

INF4140 - Models of concurrency

Fall 2016

December 1, 2016

Abstract

This is the “handout” version of the slides for the lecture (i.e., it’s a rendering of the content of the slides in a way that does not waste so much paper when printing out). The material is found in [Andrews, 2000]. Being a handout-version of the slides, some figures and graph overlays may not be rendered in full detail, I remove most of the overlays, especially the long ones, because they don’t make sense much on a handout/paper. Scroll through the real slides instead, if one needs the overlays.

This handout version also contains more remarks and footnotes, which would clutter the slides, and which typically contains remarks and elaborations, which may be given orally in the lecture.

1 Intro

29 August 2016

1.1 Warming up

Today’s agenda

Introduction

- overview
- motivation
- simple examples and considerations

Start

a bit about

- concurrent programming with critical sections and waiting. Read also [Andrews, 2000, chapter 1] for some background
- interference
- [the await-language](#)

What this course is about

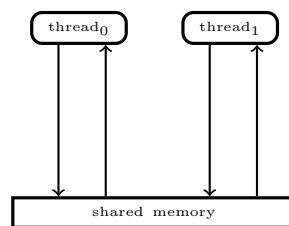
- Fundamental issues related to cooperating parallel processes
- How to think about developing parallel processes
- Various language mechanisms, design patterns, and paradigms
- Deeper understanding of parallel processes:
 - properties
 - informal analysis
 - *formal* analysis

Parallel processes

- Sequential program: one control flow thread
- Parallel/concurrent program: several control flow threads

Parallel processes need to exchange information. We will study two different ways to organize communication between processes:

- Reading from and writing to *shared variables* (part I)
- Communication with *messages* between processes (part II)



Course overview – part I: Shared variables

- atomic operations
- interference
- deadlock, livelock, liveness, fairness
- parallel programs with locks, critical sections and (active) waiting
- semaphores and passive waiting
- monitors
- formal analysis (Hoare logic), invariants
- Java: threads and synchronization

Course overview – part II: Communication

- asynchronous and synchronous message passing
- basic mechanisms: RPC (remote procedure call), rendezvous, client/server setting, channels
- Java's mechanisms
- analysis using histories
- asynchronous systems
- (Go: modern language proposal with concurrency at the heart (channels, goroutines))
- weak memory models

Part I: shared variables

Why shared (global) variables?

- reflected in the HW in conventional architectures
- there may be several CPUs inside one machine (or multi-core nowadays).
- natural interaction for **tightly coupled** systems
- used in many languages, e.g., Java's multithreading model.
- even on a single processor: use many processes, in order to get a natural partitioning
- potentially greater efficiency and/or better latency if several things happen/appear to happen "at the same time".¹

e.g.: several active windows at the same time

Simple example

Global variables: x , y , and z . Consider the following *program*:

before		after
$\{x \text{ is } a \text{ and } y \text{ is } b\} \{x = a \wedge y = b\} \quad x := x + z; y := y + z; \quad \{x = a + z \wedge y = b + z\}$		

Pre/post-conditions

- about imperative programs (fragments) \Rightarrow state-change
- the conditions describe the state of the global variables before and after a program statement
- These conditions are meant to give an understanding of the program, and are not part of the executed code.

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

If operations can be performed *independently* of one another, then concurrency may increase performance

Parallel operator \parallel

Extend the language with a construction for *parallel composition*:

$$\text{co } S_1 \parallel S_2 \parallel \dots \parallel S_n \text{ oc}$$

Execution of a parallel composition happens via the *concurrent* execution of the component processes S_1, \dots, S_n and *terminates* normally if all component processes terminate normally.

Example 1.

$$\{x = a, y = b\} \text{ co } x := x + z; \parallel y := y + z \text{ oc } \{x = a + z, y = b + z\}$$

Remark 1 (Join). *The construct abstractly described here is related to the fork-join pattern. In particular the end of the pattern, here indicate via the **oc**-construct, corresponds to a barrier or join synchronization: all participating threads, processes, tasks, ... must terminate before the rest may continue.* \square

Interaction between processes

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

- *cooperation* to obtain a result
- *competition* for common resources

organization of this interaction: "*synchronization*"

Synchronization (veeery abstractly)

restricting the possible interleavings of parallel processes (so as to avoid "bad" things to happen and to achieve "positive" things)

- increasing "**atomicity**" and **mutual exclusion** (*Mutex*): We introduce *critical sections* of which cannot be executed concurrently
- *Condition synchronization*: A process must **wait** for a specific condition to be satisfied before execution can continue.

¹Holds for concurrency in general, not just shared vars, of course.

Concurrent processes: Atomic operations

Definition 2 (Atomic). *atomic* operation: “cannot” be subdivided into smaller components.

Note

- A statement with at most one atomic operation, in addition to operations on *local* variables, can be considered atomic!
- We can do as if atomic operations do not happen concurrently!
- What is atomic depends on the language/setting: *fine-grained* and *coarse-grained* atomicity.
- e.g.: Reading/writing of global variables: usually atomic.²
- note: $x := e$: assignment statement, i.e., more than write to x !

Atomic operations on global variables

- fundamental for (shared var) concurrency
- also: process *communication* may be represented by variables: a communication channel corresponds to a variable of type vector or similar
- associated to global variables: a set of *atomic operations*
- typically: read + write,
- in hardware, e.g. LOAD/STORE
- channels as global data: *send* and *receive*
- *x-operations*: atomic operations on a variable x

Mutual exclusion

Atomic operations on a variable cannot happen simultaneously.

Example

$$\{ x = 0 \} \quad \text{co} \quad \overset{P_1}{x := x + 1} \parallel \overset{P_2}{x := x - 1} \quad \text{oc} \quad \{ ? \}$$

final state? (i.e., post-condition)

- Assume:
 - each process is executed on its own processor
 - and/or: the processes run on a multi-tasking OS
- and that x is part of a *shared* state space, i.e. a shared var
- Arithmetic operations in the two processes can be executed simultaneously, but read and write operations on x must be performed sequentially/atomically.
- *order* of these operations: dependent on relative processor speed and/or scheduling
- outcome of such programs: *difficult* to predict!
- “*race*” on x or race condition
- as for races in practice: it’s simple, avoid them at (almost) all costs

²That’s what we mostly assume in this lecture. In practice, it may be the case that not even that is atomic, for instance for “long integers” or similarly. Sometimes, only reading one machine-level “word”/byte or similar is atomic. In this lecture, as said, we don’t go into that level of details.

Atomic read and write operations

$$\{x = 0\} \quad \overset{P_1}{\text{co } x := x + 1} \parallel \overset{P_2}{x := x - 1} \text{ oc } \{?\}$$

Listing 1: Atomic steps for $x := x + 1$

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

4 atomic x -operations:

- P_1 reads (R1) value of x
- P_1 writes (W1) a value into x ,
- P_2 reads (R2) value of x , and
- P_2 writes (W2) a value into x .

Interleaving & possible execution sequences

- “program order”:
 - R1 must happen before W1 and
 - R2 before W2

- `inc` and `dec` (“-1”) work process-local³

⇒ remember (e.g.) `inc; write x` behaves “as if” atomic (alternatively `read x; inc`)

The operations can be sequenced in 6 ways (“*interleaving*”)

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

Remark 2 (Program order). *Program order means: given two statements say $stmt_1; stmt_2$, then the first statement is executed before the second: as natural as this seems: in a number of modern architecture/modern languages & their compilers, this is not guaranteed! for instance in*

$$x_1 := e_1; x_2 := e_2$$

the compiler may choose (for optimization) the swap the order of the assignment (in case e_2 does not mention x_1 and e_1 does not mention x_2). Similar “rearrangement” will effectively occur due to certain modern hardware design. Both things are related: being aware that such HWs are commonly available, an optimizing compiler may realize, that the hardware will result in certain reorderings when scheduling instructions, the language specification may guarantee weaker guarantees to the programmer than under “program order”. Those are called weak memory models. They allows the compiler more aggressive optimizations. If the programmer insists (for part of the program, perhaps), the compiler needs to inject additional code, that enforces appropriate synchronization. Such synchronization operations are supported by the hardware, but obviously come at a cost, slowing down execution. Java’s memory model is a (rather complex) weak memory model. □

Non-determinism

- final states of the program (in x): $\{0, 1, -1\}$
- *Non-determinism*: result can vary depending on factors *outside* the program code
 - timing of the execution
 - scheduler
- as (post)-condition:⁴ $x = -1 \vee x = 0 \vee x = 1$

$$\{ \} \quad x := 0; \text{co } x := x + 1 \parallel x := x - 1 \text{ oc}; \quad \{x = -1 \vee x = 0 \vee x = 1\}$$

³e.g.: in an arithmetic register, or a local variable (not mentioned in the code).

⁴Of course, things like $x \in \{-1, 0, 1\}$ or $-1 \leq x \leq 1$ are equally adequate formulations of the postcondition.

State-space explosion

- Assume 3 processes, each with the same number of atomic operations
- consider executions of $P_1 \parallel P_2 \parallel P_3$

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

- different executions can lead to different final states.
- even for simple systems: *impossible* to consider every possible execution (factorial explosion)

For n processes with m atomic statements each:

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

The “at-most-once” property

Fine grained atomicity

only the very most basic operations (R/W) are atomic “by nature”

- however: some non-atomic interactions *appear* to be atomic.
- note: expressions do only read-access (\neq statements)
- *critical reference* (in an expression e): a variable changed by another process
- e without critical reference \Rightarrow evaluation of e as if atomic

Definition 3 (At-most-once property). $x := e$ satisfies the “*amo*”-property if

1. e contains *no* critical reference
2. e with *at most one* crit. reference & x not *referenced*⁵ by other proc's

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

At-most-once examples

- In all examples: initially $x = y = 0$. And r, r' etc: local var's (registers)
- `co` and `oc` around $\dots \parallel \dots$ omitted

```
x := x + 1 || y := x + 1
x := y + 1 || y := x + 1    { (x, y) ∈ {(1, 1), (1, 2), (2, 1)} }
x := y + 1 || x := y + 3 || y := 1    { y = 1 ∧ x = 1, 2, 3, 4 }
r := y + 1 || r' := y - 1 || y := 5
r := x - x || ...    {is r now 0?}
x := x || ...    {same as skip?}
if y > 0 then y := y - 1 fi || if y > 0 then y := y - 1 fi
```

1.2 The await language

The course's first programming language: the await-language

- the usual sequential, imperative constructions such as assignment, if-, for- and while-statements
- **cobegin**-construction for parallel activity
- processes
- critical sections
- **await**-statements for (active) waiting and conditional critical sections

⁵or just read.

Syntax

We use the following syntax for non-parallel control-flow⁶

Declarations

```
int i = 3;
int a[1:n];
int a[n];7
int a[1:n] = ([n] 1);
```

Assignments

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

Seq. composition

statement; statement

Compound statement

{statements}

Conditional

if statement

While-loop

while (condition) statement

For-loop

for [i = 0 to n - 1] statement

Parallel statements

$$\text{co } S_1 \parallel S_2 \parallel \dots \parallel S_n \text{ oc}$$

- The statement(s) of each “arm” S_i are executed *in parallel* with those of the other arms.
- Termination: when all “arms” S_i have terminated (“join” synchronization)

Parallel processes

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

- Processes evaluated in arbitrary order.
- Processes are declared (as methods/functions)
- side remark: the convention “declaration = start process” is *not* used in practice.⁸

Example

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

Starts one process.

The numbers are printed in increasing order.

```
process bar2[i=1 to n] {
  write(i);
}
```

Starts n processes.

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

Read- and write-variables

- $\mathcal{V} : \text{statement} \rightarrow \text{variable set}$: set of global variables in a statement (also for expressions)
- $\mathcal{W} : \text{statement} \rightarrow \text{variable set}$ set of global *write*-variables

⁶The book uses more C/Java kind of conventions, like = for assignment and == for logical equality.

⁸one typically separates declaration/definition from “activation” (with good reasons). Note: even *instantiation* of a runnable interface in Java starts a process. Initialization (filling in initial data into a process) is tricky business.

$$\begin{aligned}
\mathcal{V}(x := e) &= \mathcal{V}(e) \cup \{x\} \\
\mathcal{V}(S_1; S_2) &= \mathcal{V}(S_1) \cup \mathcal{V}(S_2) \\
\mathcal{V}(\text{if } b \text{ then } S) &= \mathcal{V}(b) \cup \mathcal{V}(S) \\
\mathcal{V}(\text{while } (b) S) &= \mathcal{V}(b) \cup \mathcal{V}(S)
\end{aligned}$$

\mathcal{W} analogously, except the most important difference:

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

- note: expressions side-effect free

Disjoint processes

- Parallel processes without common (=shared) global variables: without *interference*

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

- *read-only* variables: no interference.
- The following *interference criterion* is thus sufficient:

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

- cf. notion of *race* (or *race condition*)
- remember also: *critical* references/amo-property
- programming practice: **final** variables in Java

1.3 Semantics and properties

Semantic concepts

- A *state* in a parallel program consists of the values of the global variables at a given moment in the execution.
- Each process executes independently of the others by *modifying* global variables using atomic operations.
- An execution of a parallel program can be modelled using a *history*, i.e. a sequence of operations on global variables, or as a sequence of states.
- For non-trivial parallel programs: *very many possible histories*.
- synchronization: conceptually used to *limit* the possible histories/interleavings.

Properties

- property = predicate over programs, resp. their histories
- A (true) *property* of a program⁹ is a predicate which is true for all possible histories of the program.

Classification

- *safety* property: program will not reach an undesirable state
- *liveness* property: program will reach a desirable state.
- *partial correctness*: If the program terminates, it is in a desired final state (safety property).
- *termination*: all histories are finite.¹⁰
- *total correctness*: The program terminates and is partially correct.

⁹the program “has” that property, the program satisfies the property ...

¹⁰that’s also called *strong* termination. Remember: non-determinism.

Properties: Invariants

- *invariant* (adj): constant, unchanging
- cf. also “loop invariant”

Definition 4 (Invariant). an *invariant* = state property, which holds for all *reachable* states.

- safety property
- appropriate for also non-terminating systems (does not talk about a final state)
- *global* invariant talks about the states of many processes
- *local* invariant talks about the state of one process

proof principle: induction

one can show that an invariant is correct by

1. showing that it holds initially,
2. and that each atomic statement maintains it.

Note: we avoid looking at all possible executions!

How to check properties of programs?

- *Testing* or *debugging* increases confidence in a program, but gives no guarantee of correctness.
- *Operational reasoning* considers *all* histories of a program.
- *Formal analysis*: Method for reasoning about the properties of a program without considering the histories one by one.

Dijkstra’s dictum:

A test can only show errors, but “never” prove correctness!

Critical sections

Mutual exclusion: combines sequences of operations in a *critical section* which then behave like atomic operations.

- When the non-interference requirement does not hold: *synchronization* to restrict the possible histories.
- Synchronization gives coarser-grained atomic operations.
- The notation $\langle S \rangle$ means that S is performed *atomically*.¹¹

Atomic operations:

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

Example The example from before can now be written as:

$$\text{int } x := 0; \text{ co } \langle x := x + 1 \rangle \parallel \langle x := x - 1 \rangle \text{ oc } \{ x = 0 \}$$

¹¹In programming languages, one could find it as `atomic{S}` or similar.

Conditional critical sections

Await statement

$\langle \text{await } (b) \ S \rangle$

- boolean condition b : *await condition*
- body S : executed atomically (conditionally on b)

Example 5.

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

- *synchronization*: decrement [delayed](#) until (if ever) $y > 0$ holds

2 special cases

- [un](#)conditional critical section or “mutex”¹²

$\langle x := 1; y := y + 1 \rangle$

- Condition synchronization:¹³

$\langle \text{await } (counter > 0) \ \rangle$

Typical pattern

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

- “critical statements” *not* enclosed in \langle angle brackets \rangle . Why?
- *invariant*: $0 \leq counter \leq 1$ (= counter acts as “*binary lock*”)
- very bad style would be: touch counter inside “critical statements” or elsewhere (e.g. access it *not* following the “await-inc-CR-dec” pattern)
- in practice: beware(!) of [exceptions](#) in the critical statements

Example: (rather silly version of) producer/consumer synchronization

- strong *coupling*
- `buf` as shared variable (“one element buffer”)
- *synchronization*
 - coordinating the “speed” of the two procs (rather strictly here)
 - to avoid, reading data which is not yet produced
 - (related:) avoid w/r conflict on shared memory

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

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

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

¹²Later, a special kind of semaphore (a binary one) is also called a “mutex”. Terminology is a bit flexible sometimes.

¹³One may also see sometimes just `await (b)`: however, the evaluation of b better be *atomic* and under *no* circumstances must b have *side-effects* (*Never, ever. Seriously*).

Example (continued)

a:

--	--	--	--	--

buf:

--

 p:

--

 c:

--

 N:

--

b:

--	--	--	--	--

- An invariant holds in *all states* in *all* histories (traces/executions) of the program (starting in its initial state(s)).
- *Global invariant*: $c \leq p \leq c+1$
- *Local invariant (Producer)*: $0 \leq p \leq N$

2 Locks & barriers

5. 09. 2016

Practical Stuff

Mandatory assignment 1 (“oblig”)

- Deadline: Friday September 30 at 18.00
- Online delivery (Devilry): <https://devilry.ifi.uio.no>

Introduction

- Central to the course are general mechanisms and issues related to parallel programs
- **Previously**: *await language* and a simple version of the *producer/consumer* example

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*

Remember: await-example: Producer/Consumer

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

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

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

Invariants

An invariant holds in *all states* in all histories of the program.

- global invariant: $c \leq p \leq c + 1$
- local (in the producer): $0 \leq p \leq N$

2.1 Critical sections

Critical section

- Fundamental concept for concurrency
- Immensely intensively researched, many solutions
- **Critical section**: part of a program that is/needs to be “protected” against interference by other processes
- Execution under *mutual exclusion*
- Related to “atomicity”

Main question today:

How can we implement critical sections / conditional critical sections?

- Various solutions and properties/guarantees
- Using *locks* and low-level operations
- SW-only solutions? HW or OS support?
- Active waiting (later semaphores and passive waiting)

Access to Critical Section (CS)

- Several processes compete for access to a shared resource
- Only one process can have access at a time: “**mutual exclusion**” (mutex)
- Possible examples:
 - Execution of bank transactions
 - Access to a printer or other resources
 - ...
- A solution to the CS problem can be used to *implement await-statements*

Critical section: First approach to a solution

- Operations on shared variables inside the CS.
- Access to the CS must then be protected to prevent interference.

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

General pattern for CS

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

Naive solution

```
int in = 1           # possible values in {1,2}

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

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

- **entry protocol:** active/busy waiting
- **exit protocol:** atomic assignment

Good solution? A solution at all? What's good, what's less so?

- More than 2 processes?
- Different execution times?

Desired properties

1. **Mutual exclusion (Mutex):** 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.

note: The three first are *safety* properties,¹⁴ The last a *liveness* property.

Safety: Invariants (review)

safety property: a program does not reach a “bad” state. In order to prove this, we can show that the program will never leave a “good” state:

- Show that the property holds in all initial states
- Show that the program statements preserve the property

Such a (good) property is often called a *global invariant*.

Atomic section

Used for synchronization of processes

- General form:

$\langle \text{await}(B) S \rangle$

- B: Synchronization condition
- Executed atomically when B is true

- **Unconditional** critical section: (B is **true**):

$\langle S \rangle$ (1)

S executed atomically

- **Conditional synchronization**:¹⁵

$\langle \text{await}(B) \rangle$ (2)

¹⁴The question for points 2 and 3, whether it's safety or liveness, is slightly up-to discussion/standpoint!

¹⁵We also use then just **await** (B) or maybe **await** B. But also in this case we assume that B is evaluated atomically.

Critical sections using “locks”

```
bool lock = false;

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

Safety properties:

- Mutex
- Absence of deadlock
- Absence of unnecessary waiting

What about taking away the angle brackets $\langle \dots \rangle$?

“Test & Set”

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

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

Effect of TS(lock)

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

Critical section with TS and spin-lock

Spin lock:

```
bool lock := false;

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

Note: Safety: Mutex, absence of deadlock and of unnecessary delay.

Strong **fairness**¹⁶ needed to guarantee eventual entry for a process

Variable **lock** becomes a hotspot!

¹⁶see later

A puzzle: “paranoid” entry protocol

Better safe than sorry?

What about *double-checking* in the entry protocol whether it is *really, really* safe to enter?

```
bool lock := false;

process p[i = 1 to n] {
  while (true) {
    while (lock) {skip};      # additional spin-lock check
    while (TS(lock)) {skip};

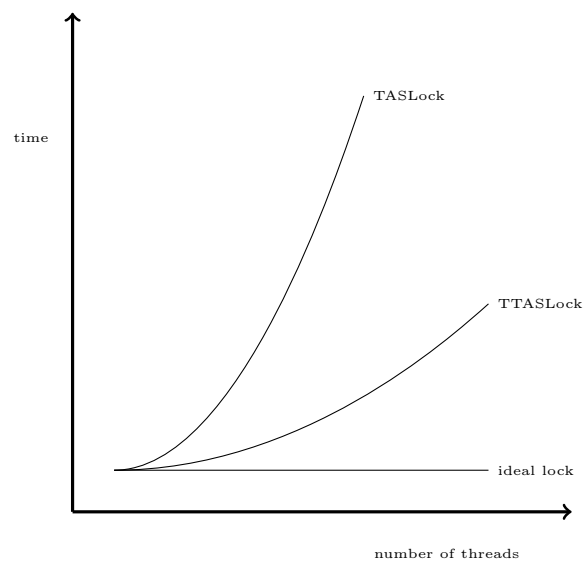
    CS;
    lock := false;
    non-CS
  }
}
```

```
bool lock := false;

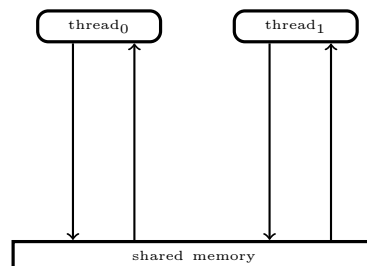
process p[i = 1 to n] {
  while (true) {
    while (lock) {skip};      # additional spin lock check
    while (TS(lock)) {
      while (lock) {skip}};  # + again inside the TAS loop
    CS;
    lock := false;
    non-CS
  }
}
```

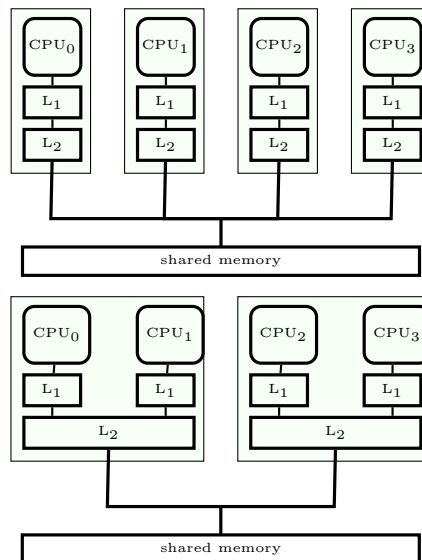
Does that make sense?

Multiprocessor performance under load (contention)



A glance at HW for shared memory





Test and test & set

- **Test-and-set** operation:
 - (Powerful) HW instruction for synchronization
 - Accesses main memory (and involves “cache synchronization”)
 - Much slower than cache access
- **Spin-loops**: faster than TAS loops
- “**Double-checked locking**”: sometimes design pattern/programming idiom for efficient CS (under certain architectures)¹⁷

Implementing await-statements

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

Then the statement $\langle S \rangle$ can be implemented by

CSentry; S; **CSexit**;

Implementation of *conditional critical section* $\langle \text{await } (B) S; \rangle$:

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

The implementation can be optimized with **Delay** between the exit and entry in the body of the **while** statement.

2.2 Liveness and fairness

Liveness properties

So far: no(!) solution for “**Eventual Entry**”.¹⁸

Liveness

Eventually, something good will happen.

- Typical example for **sequential** programs: Program termination¹⁹
- Typical example for **parallel** programs: A given process will eventually enter the critical section

Note: For parallel processes, *liveness is affected by scheduling strategies*.

¹⁷depends on the HW architecture/memory model. In some architectures: does not guarantee mutex! in which case it's an anti-pattern ...

¹⁸Except the very first (which did not satisfy “absence of unnecessary delay”

¹⁹In the first version of the slides of lecture 1, termination was defined misleadingly/too simple.

Scheduling and fairness

enabledness

Command *enabled* in a state if the statement can in principle be executed next

- Concurrent programs: often more than 1 statement enabled!

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

Scheduling: resolving non-determinism

A strategy such that for all points in an execution: if there is more than one statement enabled, pick one of them.

Fairness (informally)

enabled statements should not “systematically be neglected” (by the scheduling strategy)

Fairness notions

- Fairness: how to pick among enabled actions without being “passed over” indefinitely
- Which actions in our language are potentially non-enabled?²⁰
- Possible *status* changes:
 - disabled \rightarrow enabled (of course),
 - but also enabled \rightarrow disabled
- Differently “powerful” *forms of fairness*: guarantee of progress
 1. for actions that are always enabled
 2. for those that *stay enabled*
 3. for those whose enabledness show “on-off” behavior

Unconditional fairness

Definition 6 (Unconditional fairness). A scheduling strategy is *unconditionally fair* if each enabled *unconditional atomic action*, will eventually be chosen.

Example:

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

- $x := \text{false}$ is unconditional
- \Rightarrow The action will eventually be chosen
- guarantees termination here
 - *Example*: “Round robin” execution
 - *Note*: if-then-else, **while** (b) ; are *not* conditional atomic statements!
 - uncond. fairness formulated here based in (un)-conditional atomic actions

²⁰provided the control-flow/instruction pointer “stands in front of them”. If course, only instructions actually next for execution wrt. the concerned process are candidates. Those are the ones we meant when saying, the ones which are “in principle” executable (where it not for scheduling reasons).

Weak fairness

Definition 7 (Weak fairness). A scheduling strategy is *weakly fair* if

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

Example:

```
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
- *Example:* Round robin execution

Strong fairness

Example

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

Definition 8 (Strongly fair scheduling strategy). • unconditionally fair and

- each conditional atomic action will eventually be chosen, if the condition is true infinitely often.

For the example:

- 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**

Difficult: scheduling strategy that is both practical and strongly fair.

We look at CS solutions where access is guaranteed for *weakly* fair strategies

Fair solutions to the CS problem

- *Tie-Breaker Algorithm*
- *Ticket Algorithm*
- The book also describes the *bakery* algorithm

Tie-Breaker algorithm

- Requires no special machine instruction (like TS)
- 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

Tie-Breaker algorithm: Attempt 1

```
in1 := false, in2 := false;

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

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

What is the global invariant here?

Problem: No *mutex*

Tie-Breaker algorithm: Attempt 2

```
in1 := false, in2 := false;

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

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

```
in1 := false, in2 := false;

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

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

- Problem seems to be the entry protocol
- Reverse the order: first “set”, then “test”

“Deadlock”²¹ :- (

Tie-Breaker algorithm: Attempt 3 (with await)

- Problem: both half flagged their wish to enter \Rightarrow deadlock
- Avoid deadlock: “tie-break”
- Be fair: Don’t always give priority to one specific process
- Need to know which process last started the entry protocol.
- Add new variable: last

in1 := false, in2 := false; int last

²¹Technically, it’s more of a live-lock, since the processes still are doing “something”, namely spinning endlessly in the empty while-loops, never leaving the entry-protocol to do real work. The situation though is analogous to a “deadlock” conceptually.

```

process p1 {
  while (true){
    in1 := true;
    last := 1;
    < await ( (not in2) or
              last = 2); >
    CS
    in1 := false;
    non-CS
  }
}

```

```

process p2 {
  while (true){
    in2 := true;
    last := 2;
    < await ( (not in1) or
              last = 1); >
    CS
    in2 := false;
    non-CS
  }
}

```

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 (4)

```

process p1 {
  while (true){
    in1 := true;
    last := 1;
    while (in2 and not last = 2){ skip }
    CS
    in1 := false;
    non-CS
  }
}

```

Generalizable to many processes (see book)

Ticket algorithm

Scalability: If the Tie-Breaker algorithm is scaled up to n processes, we get a loop with $n - 1$ 2-process Tie-Breaker algorithms.

The *ticket algorithm* provides a simpler solution to the CS problem for n processes.

- Works like the “take a number” queue at the post office (with one loop)
- 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)

```
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 >;
    non-CS
  }
}
```

- loop's first line: must be **atomic**!
- **await**-statement: can be implemented as while-loop
- Some machines have an *instruction* **fetch-and-add** (FA):
FA(var, incr) = < int tmp := var; var := var + incr; return tmp;>

Ticket algorithm: Implementation

```
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;
    non-CS
  }
}
```

FA(var, incr): < int tmp := var; var := var + incr; return tmp;>

Without this instruction, we use an extra CS.²²

CSentry; turn[i]=number; number = number + 1; CSexit;

Problem with *fairness* for CS. Solved with the *bakery algorithm* (see book).

Ticket algorithm: Invariant

Invariants

- What is a *global* invariant for the ticket algorithm?
$$0 < \text{next} \leq \text{number}$$
- What is the *local* invariant for process i :
 - before the entry: $\text{turn}[i] < \text{number}$
 - if $p[i]$ in CS: then $\text{turn}[i] = \text{next}$.
- for pairs of processes $i \neq j$:
$$\text{if } \text{turn}[i] > 0 \text{ then } \text{turn}[j] \neq \text{turn}[i]$$

This holds initially, and is preserved by all atomic statements.

2.3 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.

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

All processes must reach the barrier (“join”) before any can continue.

²²Why?

Shared counter

A number of processes will synchronize the end of their tasks. Synchronization can be implemented with a shared counter:

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

Can be implemented using the FA instruction.

Disadvantages:

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

Coordination using flags

Goal: Avoid too many read- and write-operations on one variable!! (“contention”)

- Divides shared counter into several *local variables*.
- coordinator process

```
Worker [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;
```

NB: In a loop, the flags must be cleared before the next iteration!

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.

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

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

- a bit like “message passing”
- see also semaphores next week

Combined barriers

- The roles of the Worker and Coordinator processes can be *combined*.
- In a *combining tree barrier* the processes are organized in a tree structure. The processes signal arrive upwards in the tree and continue downwards in the tree.

Implementation of Critical Sections

```
bool lock = false;  
Entry: <await (!lock) lock := true>  
Critical section  
Exit: <lock := false>
```

Spin lock implementation of entry: `while (TS(lock)) skip`

Drawbacks:

- Busy waiting protocols are often complicated
- Inefficient if there are fewer processors than processes
 - Should not waste time executing a skip loop!
- No clear distinction between variables used for synchronization and computation!

Desirable to have special tools for synchronization protocols

Next week we will do better: *semaphores* !!

3 Locks & barriers

5. 09. 2016

Practical Stuff

Mandatory assignment 1 (“oblig”)

- Deadline: Friday September 30 at 18.00
- Online delivery (Devilry): <https://devilry.ifi.uio.no>

Introduction

- Central to the course are general mechanisms and issues related to parallel programs
- **Previously:** *await language* and a simple version of the *producer/consumer* example

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*

Remember: await-example: Producer/Consumer

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

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

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

Invariants

An invariant holds in *all states* in all histories of the program.

- global invariant: $c \leq p \leq c + 1$
- local (in the producer): $0 \leq p \leq N$

3.1 Critical sections

Critical section

- Fundamental concept for concurrency
- Immensely intensively researched, many solutions
- **Critical section**: part of a program that is/needs to be “protected” against interference by other processes
- Execution under *mutual exclusion*
- Related to “atomicity”

Main question today:

How can we implement critical sections / conditional critical sections?

- Various solutions and properties/guarantees
- Using *locks* and low-level operations
- SW-only solutions? HW or OS support?
- Active waiting (later semaphores and passive waiting)

Access to Critical Section (CS)

- Several processes compete for access to a shared resource
- Only one process can have access at a time: “**mutual exclusion**” (mutex)
- Possible examples:
 - Execution of bank transactions
 - Access to a printer or other resources
 - ...
- A solution to the CS problem can be used to *implement await-statements*

Critical section: First approach to a solution

- Operations on shared variables inside the CS.
- Access to the CS must then be protected to prevent interference.

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

General pattern for CS

- **Assumption:** A process which enters the CS will eventually leave it.

⇒ **Programming advice:** be aware of exceptions inside CS!

Naive solution

```
int in = 1      # possible values in {1,2}  
  
process p1 {  
  while (true) {  
    while (in=2) {skip};  
    CS;  
    in := 2;  
    non-CS  
  }  
  
process p2 {  
  while (true) {  
    while (in=1) {skip};  
    CS;  
    in := 1  
    non-CS  
  }  
}
```

- **entry protocol:** active/busy waiting
- **exit protocol:** atomic assignment

Good solution? A solution at all? What's good, what's less so?

- More than 2 processes?
- Different execution times?

Desired properties

1. **Mutual exclusion (Mutex):** 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.

note: The three first are *safety* properties,²³ The last a *liveness* property.

Safety: Invariants (review)

safety property: a program does not reach a “bad” state. In order to prove this, we can show that the program will never leave a “good” state:

- Show that the property holds in all initial states
- Show that the program statements preserve the property

Such a (good) property is often called a *global invariant*.

²³The question for points 2 and 3, whether it's safety or liveness, is slightly up-to discussion/standpoint!

Atomic section

Used for synchronization of processes

- General form:

$$\langle \text{await}(B) S \rangle$$

- B: Synchronization condition
- Executed atomically when B is true

- **Unconditional** critical section: (B is **true**):

$$\langle S \rangle \quad (3)$$

S executed atomically

- **Conditional synchronization**:²⁴

$$\langle \text{await}(B) \rangle \quad (4)$$

Critical sections using “locks”

```
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
- Absence of unnecessary waiting

What about taking away the angle brackets $\langle \dots \rangle$?

“Test & Set”

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

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

Effect of TS(lock)

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

²⁴We also use then just **await** (B) or maybe **await** B. But also in this case we assume that B is evaluated atomically.

Critical section with TS and spin-lock

Spin lock:

```
bool lock := false;

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

Note: Safety: Mutex, absence of deadlock and of unnecessary delay.

Strong **fairness**²⁵ needed to guarantee eventual entry for a process

Variable **lock** becomes a hotspot!

A puzzle: “paranoid” entry protocol

Better safe than sorry?

What about *double-checking* in the entry protocol whether it is *really, really* safe to enter?

```
bool lock := false;

process p[i = i to n] {
  while (true) {
    while (lock) {skip};          # additional spin-lock check
    while (TS(lock)) {skip};

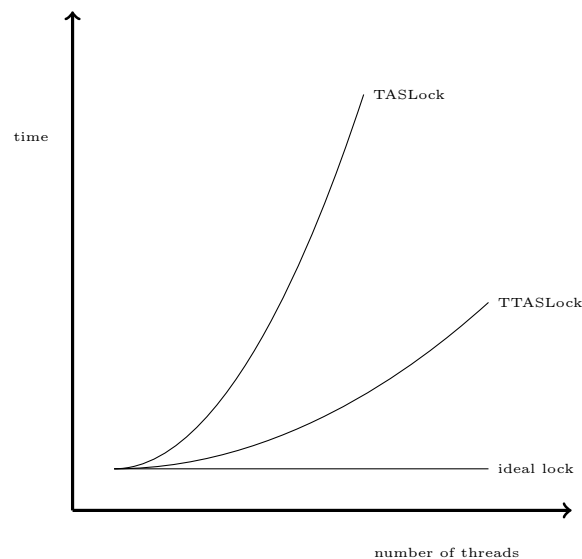
    CS;
    lock := false;
    non-CS
  }
}
```

```
bool lock := false;

process p[i = i to n] {
  while (true) {
    while (lock) {skip};          # additional spin lock check
    while (TS(lock)) {
      while (lock) {skip}};      # + again inside the TAS loop
    CS;
    lock := false;
    non-CS
  }
}
```

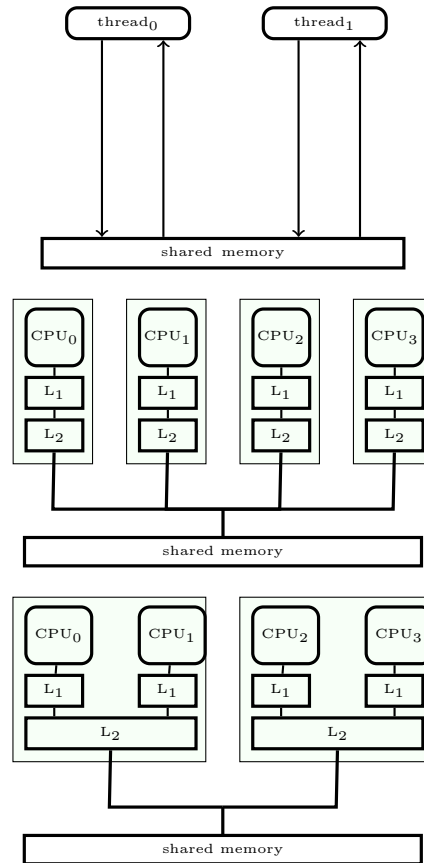
Does that make sense?

Multiprocessor performance under load (contention)



²⁵see later

A glance at HW for shared memory



Test and test & set

- **Test-and-set** operation:
 - (Powerful) HW instruction for synchronization
 - Accesses main memory (and involves “cache synchronization”)
 - Much slower than cache access
- **Spin-loops**: faster than TAS loops
- “**Double-checked locking**”: sometimes design pattern/programming idiom for efficient CS (under certain architectures)²⁶

Implementing await-statements

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

Then the statement $\langle S \rangle$ can be implemented by

CSentry; S; **CSexit**;

Implementation of *conditional critical section* $\langle \text{await } (B) S; \rangle$:

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

The implementation can be optimized with **Delay** between the exit and entry in the body of the **while** statement.

²⁶depends on the HW architecture/memory model. In some architectures: does not guarantee mutex! in which case it's an anti-pattern ...

3.2 Liveness and fairness

Liveness properties

So far: no(!) solution for “**Eventual Entry**”.²⁷

Liveness

Eventually, something good will happen.

- Typical example for **sequential** programs: Program termination²⁸
- Typical example for **parallel** programs: A given process will eventually enter the critical section

Note: For parallel processes, *liveness is affected by scheduling strategies*.

Scheduling and fairness

enabledness

Command **enabled** in a state if the statement can in principle be executed next

- Concurrent programs: often more than 1 statement enabled!

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

Scheduling: resolving non-determinism

A strategy such that for all points in an execution: if there is more than one statement enabled, pick one of them.

Fairness (informally)

enabled statements should not “systematically be neglected” (by the scheduling strategy)

Fairness notions

- Fairness: how to pick among enabled actions without being “passed over” indefinitely
- Which actions in our language are potentially non-enabled?²⁹
- Possible **status** changes:
 - disabled \rightarrow enabled (of course),
 - but also enabled \rightarrow disabled
- Differently “powerful” **forms of fairness**: guarantee of progress
 1. for actions that are always enabled
 2. for those that *stay enabled*
 3. for those whose enabledness show “on-off” behavior

²⁷Except the very first (which did not satisfy “absence of unnecessary delay”

²⁸In the first version of the slides of lecture 1, termination was defined misleadingly/too simple.

²⁹provided the control-flow/instruction pointer “stands in front of them”. If course, only instructions actually next for execution wrt. the concerned process are candidates. Those are the ones we meant when saying, the ones which are “in principle” executable (where it not for scheduling reasons).

Unconditional fairness

Definition 9 (Unconditional fairness). A scheduling strategy is *unconditionally fair* if each enabled *unconditional atomic action*, will eventually be chosen.

Example:

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

- $x := \text{false}$ is unconditional
- \Rightarrow The action will eventually be chosen
- guarantees termination here
- Example: “Round robin” execution
- Note: if-then-else, **while** (b) ; are *not* conditional atomic statements!
- uncond. fairness formulated here based in (un)-conditional atomic actions

Weak fairness

Definition 10 (Weak fairness). A scheduling strategy is *weakly fair* if

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

Example:

```
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
- Example: Round robin execution

Strong fairness

Example

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

Definition 11 (Strongly fair scheduling strategy). • unconditionally fair and

- each conditional atomic action will eventually be chosen, if the condition is true infinitely often.

For the example:

- 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**

Difficult: scheduling strategy that is both practical and strongly fair.

We look at CS solutions where access is guaranteed for *weakly* fair strategies

Fair solutions to the CS problem

- *Tie-Breaker Algorithm*
- *Ticket Algorithm*
- The book also describes the *bakery* algorithm

Tie-Breaker algorithm

- Requires no special machine instruction (like TS)
- 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

Tie-Breaker algorithm: Attempt 1

```
in1 := false , in2 := false ;

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

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

What is the global invariant here?

Problem: No *mutex*

Tie-Breaker algorithm: Attempt 2

```
in1 := false , in2 := false ;

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

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

```

in1 := false, in2 := false;

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

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

```

- Problem seems to be the entry protocol
- Reverse the order: first “set”, then “test”

“Deadlock”³⁰ :- (

Tie-Breaker algorithm: Attempt 3 (with await)

- Problem: both half flagged their wish to enter \Rightarrow deadlock
- Avoid deadlock: “tie-break”
- Be fair: Don’t always give priority to one specific process
- Need to know which process last started the entry protocol.
- Add new variable: last

```

process p1 {
  while (true){
    in1 := true;
    last := 1;
    < await ( (not in2) or
              last = 2); >
    CS
    in1 := false;
    non-CS
  }
}

```

```

process p2 {
  while (true){
    in2 := true;
    last := 2;
    < await ( (not in1) or
              last = 1); >
    CS
    in2 := false;
    non-CS
  }
}

```

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.

³⁰Technically, it’s more of a live-lock, since the processes still are doing “something”, namely spinning endlessly in the empty while-loops, never leaving the entry-protocol to do real work. The situation though is analogous to a “deadlock” conceptually.

Tie-Breaker algorithm (4)

```
process p1 {
  while (true){
    in1 := true;
    last := 1;
    while (in2 and not last = 2){skip}
    CS
    in1 := false;
    non-CS
  }
}
```

Generalizable to many processes (see book)

Ticket algorithm

Scalability: If the Tie-Breaker algorithm is scaled up to n processes, we get a loop with $n - 1$ 2-process Tie-Breaker algorithms.

The *ticket algorithm* provides a simpler solution to the CS problem for n processes.

- Works like the “take a number” queue at the post office (with one loop)
- 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)

```
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 >;
    non-CS
  }
}
```

- loop’s first line: must be **atomic**!
- **await**-statement: can be implemented as while-loop
- Some machines have an *instruction* **fetch-and-add** (FA):

FA(var, incr) = < int tmp := var; var := var + incr; return tmp; >

Ticket algorithm: Implementation

```
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;
    non-CS
  }
}
```

FA(var, incr) = < int tmp := var; var := var + incr; return tmp; >

Without this instruction, we use an extra CS:³¹

CSentry; turn[i]=number; number = number + 1; CSexit;

Problem with *fairness* for CS. Solved with the *bakery algorithm* (see book).

³¹Why?

Ticket algorithm: Invariant

Invariants

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

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

- What is the *local* invariant for process i :

- before the entry: $\text{turn}[i] < \text{number}$
- if $p[i]$ in CS: then $\text{turn}[i] = \text{next}$.

- for pairs of processes $i \neq j$:

if $\text{turn}[i] > 0$ then $\text{turn}[j] \neq \text{turn}[i]$

This holds initially, and is preserved by all atomic statements.

3.3 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.

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

All processes must reach the barrier (“join”) before any can continue.

Shared counter

A number of processes will synchronize the end of their tasks. Synchronization can be implemented with a shared counter:

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

Can be implemented using the FA instruction.

Disadvantages:

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

Coordination using flags

Goal: Avoid too many read- and write-operations on one variable!! (“contention”)

- Divides shared counter into several *local variables*.
- coordinator process

```

Worker[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;

```

NB: In a loop, the flags must be cleared before the next iteration!

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 `arrive` are initialized to 0.

```

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

```

```

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

```

- a bit like “message passing”
- see also semaphores next week

Combined barriers

- The roles of the Worker and Coordinator processes can be *combined*.
- In a *combining tree barrier* the processes are organized in a tree structure. The processes signal arrive upwards in the tree and continue downwards in the tree.

Implementation of Critical Sections

```

bool lock = false;
Entry: < await (!lock) lock := true >
Critical section
Exit: < lock := false >

```

Spin lock implementation of entry: `while (TS(lock)) skip`

Drawbacks:

- Busy waiting protocols are often complicated
- Inefficient if there are fewer processors than processes
 - Should not waste time executing a skip loop!
- No clear distinction between variables used for synchronization and computation!

Desirable to have special tools for synchronization protocols

Next week we will do better: *semaphores* !!

4 Semaphores

7 September, 2015

4.1 Semaphore as sync. construct

Overview

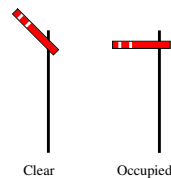
- **Last lecture:** Locks and barriers (complex techniques)
 - No clear separation between variables for synchronization and variables to compute results
 - Busy waiting
- **This lecture:** Semaphores (synchronization tool)
 - Used easily for mutual exclusion and condition synchronization.
 - A way to implement signaling (and scheduling).
 - implementable in many ways.
 - 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

- 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, similar in Windows etc.

Concept

- *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:³²

P and *V*

- **P:** (Passeren) Wait for signal – want to **pass**
 - * **effect:** **wait** until the value is greater than zero, and **decrease** the value by one
- **V:** (Vrijgeven) Signal an event – **release**
 - * **effect:** **increase** the value by one
- nowadays, for libraries or sys-calls: other names are preferred (up/down, wait/signal, ...)
- different “flavors” of semaphores (binary vs. counting)
- a mutex: often (basically) a synonym for binary semaphore

Syntax and semantics

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

- semantics³³ (via “implementation”):

P-operation **P(s)**

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

V-operation **V(s)**

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

Important: No **direct** access to the value of a semaphore.
E.g. a test like

if (*s* = 1) **then** ... **else**

is seriously *not* allowed!

Kinds of semaphores

- 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 **awaken** in the **order** they where delayed

³²There are different stories about what Dijkstra actually wanted *V* and *P* to stand for.

³³Semantics generally means “meaning”

Example: Mutual exclusion (critical section)

Mutex³⁴ implemented by a **binary semaphore**

```
sem mutex := 1;
process CS[i = 1 to n] {
  while (true) {
    P(mutex);
    criticalsection;
    V(mutex);
    noncriticalsection;
  }
}
```

Note:

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

Example: Barrier synchronization

Semaphores may be used for **signaling events**

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

Note:

- **signalling** semaphores: usually **initialized to 0** and
- **signal** with a V and then **wait** with a P

4.2 Producer/consumer

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

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

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

³⁴As mentioned: “mutex” is also used to refer to a data-structure, basically the same as binary semaphore itself.

```

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

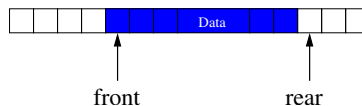
```

Note:

- remember also P/C with await + exercise 1
- **empty** and **full** are both **binary semaphores**, **together** they form a split binary semaphore.
- solution works with **several** producers/consumers

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

```

T buff[n]
int front := 0, rear := 0; # 'pointers'
sem empty := n,
sem full := 0;

```

```

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

```

```

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

```

several producers or consumers?

Increasing the number of processes

- **several producers and consumers**.
- New synchronization problems:
 - **Avoid** that two producers **deposits** to `buff[rear]` before `rear` is updated
 - **Avoid** that two consumers **fetches** from `buff[front]` before `front` is updated.
- Solution: additionally 2 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

```
T buff[n]                # array, elem's of type T
int front := 0, rear := 0; # ''pointers''
sem empty := n,
sem full  := 0;
sem mutexDeposit, mutexFetch := 1; # protect the data struct.
```

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

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

4.3 Dining philosophers

Problem: Dining philosophers introduction

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



Dining philosophers: sketch

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

now: program the actions **acquire forks** and **release forks**

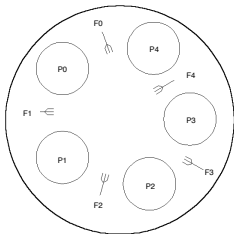
³⁵image from wikipedia.org

Dining philosophers: 1st attempt

- forks as [semaphores](#)
- philosophers: pick up left fork first

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

```
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]);  
  }  
}
```



ok solution?

Example: Dining philosophers 2nd attempt

breaking the symmetry

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

```
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]);  
  }  
}
```

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

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

Dining philosophers

- important illustration of problems with concurrency:
 - deadlock
 - but also other aspects: liveness and fairness etc.
- resource access
- connection to mutex/critical sections

4.4 Readers/writers

Example: Readers/Writers overview

- Classical synchronization problem
- **Reader** and **writer** processes, sharing access to a “**database**”
 - readers: read-only from the database
 - writers: update (and read from) the database
- R/R access unproblematic, W/W or W/R: interference
 - **writers** need **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 competes 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**”³⁶
 2. **Condition synchronization**:
 - Using a **split binary semaphore**
 - Easy to adapt to different scheduling strategies

Readers/writers with mutex (1)

sem rw := 1

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

³⁶The way the solution is “unfair” does not technically fit into the fairness categories we have introduced.

```

process Writer [i=1 to N] {
  while (true) {
    ...
    P(rw);

    write to DB

    V(rw);
  }
}

```

- safety ok
- but: unnecessarily cautious
- We want **more than one reader** simultaneously.

Readers/writers with mutex (2)

Initially:

```

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

```

```

process Reader [i=1 to M] {
  while (true) {
    ...
    < nr := nr + 1;
    if (nr=1) P(rw) >;

    read from DB

    < nr := nr - 1;
    if (nr=0) V(rw) >;
  }
}

```

```

process Writer [i=1 to N] {
  while (true) {
    ...

    P(rw);

    write to DB

    V(rw);
  }
}

```

Semaphore **inside** await statement? It's perhaps a bit strange, but works.

Readers/writers with mutex (3)

```

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 from DB

    P(mutexR)
    nr := nr - 1;
    if (nr=0) V(rw);
    V(mutexR)
  }
}

```

“Fairness”

What happens if we have a constant **stream** of readers? “Reader’s preference”

Readers/writers with condition synchronization: overview

- previous **mutex** solution solved **two** separate synchronization problems
 - **Readers and. writers** for access to the **database**
 - **Reader vs. reader** for access to the **counter**
- Now: a solution based on **condition synchronization**

Invariant

reasonable invariant³⁷

1. When **a writer** access the DB, **no one else** can
 2. When **no writers** access the DB, **one or more** readers may
- introduce two counters:
 - **nr**: number of active readers
 - **nw**: number of active writers

The invariant may be:

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

Code for “counting” readers and writers

Reader:

```
< nr := nr + 1; >  
read from DB  
< nr := nr - 1; >
```

Writer:

```
< nw := nw + 1; >  
write to DB  
< nw := nw - 1; >
```

- maintain **invariant** \Rightarrow add **sync-code**
- decrease counters: not dangerous
- before increasing, check/synchronize:
 - before increasing **nr**: **nw = 0**
 - before increasing **nw**: **nr = 0 and nw = 0**

condition synchronization: without semaphores

Initially:

```
int nr := 0;  # number of active readers  
int nw := 0;  # number of active writers  
sem rw := 1   # lock for reader/writer mutex  
  
## Invariant RW: (nr = 0 or nw = 0) and nw <= 1
```

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

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

³⁷2nd point: technically, not an invariant.

Condition synchr.: converting to split binary semaphores

implementation of `await`'s: possible via [split binary semaphores](#)

- May be used to implement different synchronization problems with different guards $B_1, B_2 \dots$

General pattern

- `entry`³⁸ semaphore e , initialized to 1
- For each guard B_i :
 1. associate 1 counter and
 2. 1 delay-semaphoreboth initialized to 0
 - * semaphore: delay the processes waiting for B_i
 - * counter: count the number of processes waiting for B_i

⇒ for readers/writers problem: 3 semaphores and 2 counters:

```
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 synchr.: 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 `CR` because either:
 - the process checks itself,
 - or the process is only `signaled` if B_i holds
- and another `pitfall`: Avoid `deadlock` by checking the counters before the delay semaphores are signaled.
 - 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 synchr.: Reader

```
int nr := 0, nw = 0;      # condition 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 ∨ nw = 0) ∧ 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 from DB;

    P(e); nr:=nr -1; SIGNAL; # < nr:=nr-1 >
  }
}
```

³⁸Entry to the administrative CS's, not entry to data-base access

With condition synchronization: Writer

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

    write to DB;

    P(e);nw:=nw -1; SIGNAL # < nw:=nw-1>
  }
}
```

With condition synchronization: Signalling

- SIGNAL

```
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
```

5 Monitors

19. September 2016

Overview

- Concurrent execution of different processes
- Communication by *shared variables*
- Processes may *interfere* `x := 0; co x := x + 1 || x := x + 2 oc` final value of x will be 1, 2, or 3
- **await** language – **atomic regions** `x := 0; co <x := x + 1> || <x := x + 2> oc` final value of x will be 3
- special tools for **synchronization**: Last week: semaphores **Today**: monitors

Outline

- Semaphores: review
- Monitors:
 - Main ideas
 - Syntax and semantics
 - * Condition variables
 - * Signaling disciplines for monitors
 - Synchronization problems:
 - * Bounded buffer
 - * Readers/writers
 - * Interval timer
 - * Shortest-job next scheduling
 - * Sleeping barber

Semaphores

- Used as “synchronization variables”
- **Declaration:** *sem s = 1;*
- **Manipulation:** Only two operations, $P(s)$ and $V(s)$
- **Advantage:** Separation of “business” and synchronization code
- **Disadvantage:** Programming with semaphores can be tricky:³⁹
 - Forgotten P or V operations
 - Too many P or V operations
 - They are shared between processes
 - * Global knowledge
 - * Need to examine all processes to see how a semaphore is intended

Monitors

Monitor

“Abstract data type + synchronization”

- program *modules* with *more structure* than semaphores
- monitor *encapsulates* data, which can only be *observed* and *modified* by the monitor’s *procedures*.
 - contains *variables* that describe the *state*
 - variables can be *changed only* through the available procedures
- implicit *mutex*: only 1 procedure may be active at a time.
 - A procedure: mutex access to the data in the monitor
 - 2 procedures in the same monitor: never executed concurrently
- *cooperative* scheduling
- **Condition synchronization:**⁴⁰ is given by *condition variables*
- At a lower level of abstraction: monitors can be implemented using locks or semaphores (for instance)

Usage

- process = active \Leftrightarrow Monitor: = passive/re-active
- a procedure is *active*, if a statement in the procedure is executed by some process
- *all* shared variables: inside the monitor
- processes *communicate* by calling monitor procedures
- processes do not need to know all the implementation details
 - Only the visible effects public procedures important
- implementation can be changed, if visible effects remains
- Monitors and processes can be developed relatively independent \Rightarrow **Easier to understand** and develop parallel programs

³⁹Same may be said about simple locks.

⁴⁰block a process until a particular condition holds.

Syntax & semantics

```
monitor name {  
    mon. variables    # shared global variables  
    initialization  
    procedures  
}
```

monitor: a form of [abstract data type](#):

- *only* the procedures' names visible from outside the monitor:

call *name.opname*(arguments)

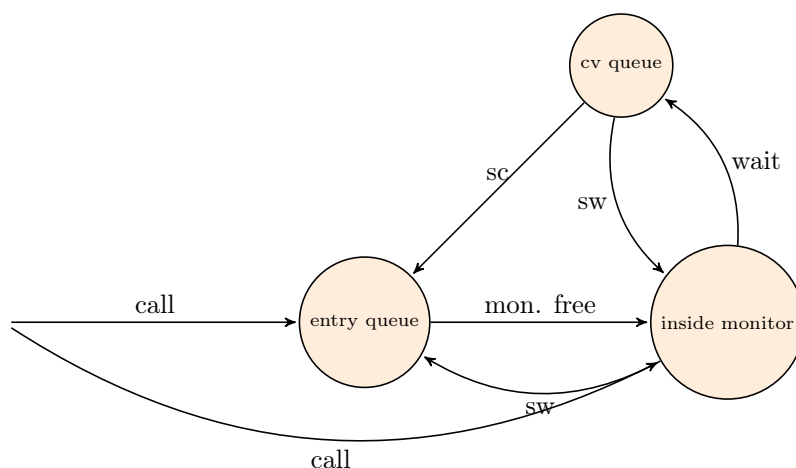
- statements *inside* a monitor: *no* access to variables *outside* the monitor
- monitor variables: [initialized](#) before the monitor is used

monitor [invariant](#): describe the monitor's inner states

Condition variables

- monitors contain *special* type of variables: **cond** (condition)
- used for synchronization/to [delay](#) processes
- each such [variable](#) is associated with a *wait condition*
- “*value*” of a condition variable: [queue](#) of delayed processes
- **value**: 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 process to wait in the queue to cv  
signal(cv);        # wakes up a process in the queue to cv  
signal_all(cv);    # wakes up all processes in the queue to cv
```



Remark 3. The figure is schematic and combines the “transitions” of signal-and-wait and signal-and-continue in a single diagram. The corresponding transition, here labelled *SW* and *SC* are the state changes caused by being signalled in the corresponding discipline. □

5.1 Semaphores & signalling disciplines

Implementation of semaphores

A **monitor** with *P* and *V* operations:

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

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

  procedure Vsem() {
    s := s+1;
    signal (pos); }
}
```

Signaling disciplines

- **signal** on a condition variable **cv** roughly has the following effect:
 - **empty queue**: no effect
 - the **process** at the head of the queue to **cv** is **woken up**
- **wait** and **signal**: *FIFO signaling strategy*
- When a process executes **signal(cv)**, then it is inside the monitor. If a waiting process is woken up: *two active processes* in the monitor?

2 disciplines to provide mutex:

- **Signal and Wait (SW)**: the signaller waits, and the signalled process gets to execute immediately
- **Signal and Continue (SC)**: the signaller continues, and the signalled process executes later

Signalling disciplines

Is this a **FIFO** semaphore assuming **SW** or **SC**?

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

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

  procedure Vsem() {
    s := s+1;
    signal (pos); }
}
```

Signalling disciplines: **FIFO** semaphore for **SW**

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

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

  procedure Vsem() {
    s := s+1;
    signal (pos); }
}
```

```

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

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

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

```

FIFO semaphore

FIFO semaphore with SC: can be achieved by explicit transfer of control inside the monitor (forward the condition).

```

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);
  }
}

```

5.2 Bounded buffer

Bounded buffer synchronization (1)

- buffer of size n (“channel”, “pipe”)
- **producer**: performs **put** operations on the buffer.
- **consumer**: performs **get** operations on the buffer.
- **count**: number of items in the buffer
- two access operations (“methods”)
 - **put** operations must **wait** if buffer **full**
 - **get** operations must **wait** if buffer **empty**
- assume SC discipline⁴¹

Bounded buffer synchronization (2)

- When a process is **woken up**, it **goes back** to the monitor’s **entry queue**
 - **Competes** with other processes for entry to the monitor
 - Arbitrary **delay** between awakening and start of execution
- ⇒ *re-test* the wait condition, when execution starts
 - E.g.: **put** process wakes up when the buffer is not full
 - * Other processes can perform **put** operations before the awakened process starts up
 - * Must therefore **re-check** that the buffer is not full

⁴¹It’s the commonly used one in practical languages/OS.

Bounded buffer synchronization monitors (3)

```
monitor Bounded_Buffer {
    typeT buf[n]; int count := 0;
    cond not_full, not_empty;

    procedure put(typeT data){
        while (count = n) wait(not_full);
        # Put element into buf
        count := count + 1; signal(not_empty);
    }

    procedure get(typeT &result) {
        while (count = 0) wait(not_empty);
        # Get element from buf
        count := count - 1; signal(not_full);
    }
}
```

Bounded buffer synchronization: client-sides

```
process Producer[i = 1 to M]{
    while (true){
        ...
        call Bounded_Buffer.put(data);
    }
}
process Consumer[i = 1 to N]{
    while (true){
        ...
        call Bounded_Buffer.get(result);
    }
}
```

5.3 Readers/writers problem

Readers/writers problem

- Reader and writer processes share a common resource (“database”)
- Reader’s transactions can read data from the DB
- Write transactions can read and update data in the DB
- Assume:
 - DB is initially consistent and that
 - Each transaction, seen in isolation, maintains consistency
- To avoid interference between transactions, we require that
 - writers: exclusive access to the DB.
 - No writer: an arbitrary number of readers can access simultaneously

Monitor solution to the reader/writer problem (2)

- database should not be encapsulated in a monitor, as the readers will not get shared access
- monitor instead regulates access of the processes
- processes don’t enter the critical section (DB) until they have passed the RW_Controller monitor

Monitor procedures:

- request_read: requests read access

- **release_read**: reader leaves DB
- **request_write**: requests write access
- **release_write**: writer leaves DB

Invariants and signalling

Assume that we have **two counters** as local variables in the monitor:

nr — number of readers
nw — number of writers

Invariant

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

We want RW to be a *monitor invariant*

- chose carefully **condition variables** for “communication” (waiting/signaling)

Let **two condition variables** **oktoread** og **oktowrite** regulate waiting readers and waiting writers, respectively.

```
monitor RW_Controller { # RW (nr = 0 or nw = 0) and nw ≤ 1
  int nr:=0, nw:=0
  cond oktoread ; # signalled when nw = 0
  cond oktowrite; # sig'ed when nr = 0 and nw = 0

  procedure request_read() {
    while (nw > 0) wait(oktoread);
    nr := nr + 1;
  }
  procedure release_read() {
    nr := nr - 1;
    if nr = 0 signal (oktowrite);
  }

  procedure request_write() {
    while (nr > 0 or nw > 0) wait(oktowrite);
    nw := nw + 1;
  }

  procedure release_write() {
    nw := nw - 1;
    signal(oktowrite); # wake up 1 writer
    signal_all(oktoread); # wake up all readers
  }
}
```

Invariant

- **monitor invariant I**: describe the monitor’s inner state
- expresses relationship between monitor variables
- maintained by execution of procedures:
 - must hold: **after initialization**
 - must hold: when a **procedure terminates**
 - must hold: when we **suspend** execution due to a call to **wait**
- ⇒ can **assume** that the invariant holds *after wait* and when a **procedure starts**
- Should be as *strong* as possible

Monitor solution to reader/writer problem (6)

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

```
procedure request_read() {  
    # May assume that the invariant holds here  
    while (nw > 0) {  
        # the invariant holds here  
        wait(oktoread);  
        # May assume that the invariant holds here  
    }  
    # Here, we know that nw = 0...  
    nr := nr + 1;  
    # ...thus: invariant also holds after increasing nr  
}
```

Do we need $nr \geq 0$ and $nw \geq 0$?

Do we need to protect `release_read` and `release_write`?

5.4 Time server

Time server

- Monitor that enables sleeping for a given amount of time
- Resource: a logical clock (`tod`)
- Provides two operations:
 - `delay(interval)`: caller wishes to sleep for `interval` time
 - `tick` increments the logical clock with one tick Called by the hardware, preferably with high execution priority
- Each process which calls `delay` computes its own time for wakeup: `wake_time := tod + interval`;
- Waits as long as `tod < wake_time`
 - Wait condition is dependent on local variables

Covering condition:

- `all` processes are woken up when it is possible for `some` to continue
- Each process checks its condition and sleeps again if this does not hold

Time server: covering condition

Invariant: $CLOCK : tod \geq 0 \wedge tod \text{ increases monotonically by } 1$

```
monitor Timer { int tod = 0; # Time Of Day  
    cond check; # signalled when tod is increased  
  
    procedure delay(int interval) {  
        int wake_time;  
        wake_time = tod + interval;  
        while (wake_time > tod) wait(check);  
    }  
  
    procedure tick() {  
        tod = tod + 1;  
        signal_all(check);  
    }  
}
```

- Not very efficient if many processes will wait for a long time
- Can give many false alarms

Prioritized waiting

- Can also give additional argument to `wait`: `wait(cv, rank)`
 - Process waits in the queue to `cv` in ordered by the argument `rank`.
 - At `signal`: Process with lowest `rank` is awakened first
- Call to `minrank(cv)` returns the value of `rank` to the first process in the queue (with the lowest rank)
 - The queue is not modified (no process is awakened)
- Allows more efficient implementation of `Timer`

Time server: Prioritized wait

- Uses prioritized waiting to order processes by `check`
- The process is awakened only when `tod ≥ wake_time`
- Thus we do not need a `while` loop for `delay`

```
monitor Timer {
    int tod = 0; # Invariant: CLOCK
    cond check; # signalled when minrank(check) ≤ tod

    procedure delay(int interval) {
        int wake_time;
        wake_time := tod + interval;
        if (wake_time > tod) wait(check, wake_time);
    }

    procedure tick() {
        tod := tod + 1;
        while (!empty(check) && minrank(check) ≤ tod)
            signal(check);
    }
}
```

5.5 Shortest-job-next scheduling

Shortest-Job-Next allocation

- Competition for a shared resource
- A monitor administrates access to the resource
- Call to `request(time)`
 - Caller needs access for time interval `time`
 - If the resource is free: caller gets access directly
- Call to `release`
 - The resource is released
 - If waiting processes: The resource is allocated to the waiting process with lowest value of `time`
- Implemented by prioritized wait

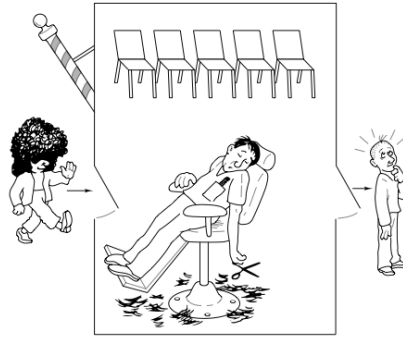
Shortest-Job-Next allocation (2)

```
monitor Shortest_Job_Next {
    bool free = true;
    cond turn;

    procedure request(int time) {
        if (free)
            free := false
        else
            wait(turn, time)
    }

    procedure release() {
        if (empty(turn))
            free := true;
        else
            signal(turn);
    }
}
```

5.6 Sleeping barber



The story of the sleeping barber

- barbershop: with two doors and some chairs.
- customers: come in through one door and leave through the other. Only one customer sits in the barber chair at a time.
- Without customers: barber sleeps in one of the chairs.
- When a customer arrives and the barber sleeps \Rightarrow barber is woken up and the customer takes a seat.
- barber busy \Rightarrow the customer takes a nap
- Once served, barber lets customer out the exit door.
- If there are waiting customers, one of these is woken up. Otherwise the barber sleeps again.

Interface

Assume the following *monitor procedures*

Client: `get_haircut`: called by the customer, returns when haircut is done

Server: barber calls:

- `get_next_customer`: called by the barber to serve a customer
- `finish_haircut`: called by the barber to let a customer out of the barbershop

Rendez-vous

Similar to a [two-process barrier](#): *Both* parties must arrive before either can continue.⁴²

- The barber must wait for a customer
- Customer must wait until the barber is available

The barber can have rendezvous with an arbitrary customer.

Organize the sync.: Identify the synchronization needs

1. barber must wait until
 - (a) customer sits in chair
 - (b) customer left barbershop
2. customer must wait until
 - (a) the barber is available
 - (b) the barber opens the exit door

⁴²Later, in the context of message passing, will have a closer look at making rendez-vous synchronization (using channels), but the pattern “2 partners must be present at a point at the same time” is analogous.

client perspective:

- two phases (during `get_haircut`)
 1. “entering”
 - trying to get hold of barber,
 - sleep otherwise
 2. “leaving”:
- between the phases: `suspended`

Processes signal when one of the wait conditions is satisfied.

Organize the synchronization: state

3 var’s to synchronize the processes:

`barber`, `chair` and `open` (initially 0)

binary variables, alternating between 0 and 1:

- for entry-`rendevouz`
 1. `barber = 1` : the barber is ready for a new customer
 2. `chair = 1`: the customer sits in a chair, the barber hasn’t begun to work
- for exit-sync
 3. `open = 1`: exit door is open, the customer has not yet left

Sleeping barber

```
monitor Barber_Shop {
  int barber := 0, chair := 0, open := 0;
  cond barber_available;      # signalled when barber > 0
  cond chair_occupied;        # signalled when chair > 0
  cond door_open;             # signalled when open > 0
  cond customer_left;         # signalled when open = 0

  procedure get_haircut() {
    while (barber = 0) wait(barber_available); # RV with barber
    barber := barber + 1;
    chair := chair + 1; signal(chair_occupied);

    while (open = 0) wait(door_open);          # leave shop
    open := open + 1; signal(customer_left);
  }

  procedure get_next_customer() {              # RV with client
    barber := barber + 1; signal(barber_available);
    while (chair = 0) wait(chair_occupied);
    chair := chair + 1;
  }

  procedure finished_cut() {
    open := open + 1; signal(door_open);       # get rid of customer
    while (open > 0) wait(customer_left);
  }
}
```

6 Program analysis

26. 9. 2016

Program correctness

Is my program correct? Central question for this and the next lecture.

- Does a given program behave as intended?
- Surprising behavior?

$x := 5; \{ x = 5 \} \langle x := x + 1 \rangle; \{ x = ? \}$

- clear: $x = 5$ *immediately* after first assignment

- Will this still hold when the second assignment is executed?
 - Depends on other processes
- What will be the final value of x ?

Today: Basic machinery for program reasoning **Next week:** Extending this machinery to the concurrent setting

Concurrent executions

- Concurrent program: several threads operating on (here) *shared* variables
- Parallel updates to x and y :

$$\text{co } \langle x := x \times 3; \rangle \parallel \langle y := y \times 2; \rangle \text{ oc}$$
- Every (concurrent) execution can be written as a sequence of atomic operations (gives one history)
- Two possible histories for the above program
- Generally, if n processes executes m atomic operations each:

$$\frac{(n * m)!}{m!^n} \quad \text{If } n=3 \text{ and } m=4: \frac{(3 * 4)!}{4!^3} = 34650$$

How to verify program properties?

- *Testing* or *debugging* increases confidence in the program correctness, but does not guarantee *correctness*
 - Program testing can be an effective way to show the presence of bugs, but not their absence
- *Operational reasoning* (exhaustive case analysis) tries all possible executions of a program
- *Formal analysis* (assertional reasoning) allows to *deduce* the correctness of a program without executing it
 - *Specification* of program behavior
 - Formal argument that the specification is correct

States

- *state* of a program consists of the values of the program variables at a point in time, example: $\{ x = 2 \wedge y = 3 \}$
- The *state space* of a program is given by the different values that the declared variables can take
- Sequential program: one execution thread operates on its own state space
- The state may be *changed* by assignments (“imperative”)

Example 12.

$$\{ x = 5 \wedge y = 5 \} x := x * 2; \{ x = 10 \wedge y = 5 \} y := y * 2; \{ x = 10 \wedge y = 10 \}$$

Executions

- Given program S as sequence $S_1; S_2; \dots; S_n$, starting in a state p_0 :

$$\bullet \xrightarrow{p_0} \boxed{S_1} \xrightarrow{p_1} \boxed{S_2} \xrightarrow{p_2} \dots \xrightarrow{p_{n-1}} \boxed{S_n} \xrightarrow{p_n} \bullet$$

where p_1, p_2, \dots, p_n are the different states during execution

- Can be documented by: $\{p_0\}S_1\{p_1\}S_2\{p_2\} \dots \{p_{n-1}\}S_n\{p_n\}$
- p_0, p_n gives an external specification of the program: $\{p_0\}S\{p_n\}$
- We often refer to p_0 as the *initial* state and p_n as the *final* state

Example 13 (from previous slide).

$$\{ x = 5 \wedge y = 5 \} x := x * 2; y := y * 2; \{ x = 10 \wedge y = 10 \}$$

Assertions

Want to express more **general** properties of programs, like

$$\{ x = y \} x := x * 2; y := y * 2; \{ x = y \}$$

- If the assertion $x = y$ holds, when the program *starts*, $x = y$ will also hold when/if the program *terminates*
- Does not talk about specific, concrete *values* of x and y , but about *relations* between their values
- Assertions characterise **sets** of states

Example 14. The assertion $x = y$ describes *all* states where the values of x and y are equal, like $\{x = -1 \wedge y = -1\}$, $\{x = 1 \wedge y = 1\}$, ...

Assertions

- state assertion P : *set* of states where P is true:

$x = y$	All states where x has the same value as y
$x \leq y$:	All states where the value of x is less or equal to the value of y
$x = 2 \wedge y = 3$	Only one state (if x and y are the only variables)
<i>true</i>	All states
<i>false</i>	No state

Example 15.

$$\{ x = y \} x := x * 2; \{ x = 2 * y \} y := y * 2; \{ x = y \}$$

Assertions may or may not say something correct for the behavior of a program (fragment). In this example, the assertions say something correct.

Formal analysis of programs

- establish program properties/correctness, using a system for formal reasoning
- Help in understanding how a program behaves
- Useful for program construction
- Look at logics for formal analysis
- basis of analysis **tools**

Formal system

- *Axioms*: Defines the meaning of individual program statements
- *Rules*: Derive the meaning of a program from the individual statements in the program

Logics and formal systems

Our formal system consists of:

- syntactic building blocks:
 - A set of *symbols* (constants, variables,...)
 - A set of *formulas* (meaningful combination of symbols)
- derivation machinery
 - A set of *axioms* (assumed to be true)
 - A set of *inference rules*

Inference rule⁴³

$$\frac{H_1 \quad \dots \quad H_n}{C}$$

- H_i : *premise* (sometimes called *assumption*), and C : *conclusion*
- intention: conclusion is true if all the premises are true
- The inference rules specify how to derive additional formulas from axioms and other formulas.

Symbols

- variables: x, y, z, \dots (which include **program** variables + “extra” ones)
- Relation symbols: \leq, \geq, \dots
- Function symbols: $+, -, \dots$, and constants $0, 1, 2, \dots, \text{true}, \text{false}$
- Equality (also a relation symbol): $=$

Formulas of first-order logic

Meaningful combination of symbols

Assume that A and B are formulas, then the following are also formulas:

- $\neg A$ means “not A ”
- $A \vee B$ means “ A or B ”
- $A \wedge B$ means “ A and B ”
- $A \Rightarrow B$ means “ A implies B ”

If x is a variable and A , the following are formulas:⁴⁴

- $\forall x : A(x)$ means “ A is true for all values of x ”
- $\exists x : A(x)$ means “there is (at least) one value of x such that A is true”

Examples of axioms and rules (no programs involved yet)

Typical axioms:

- $A \vee \neg A$
- $A \Rightarrow A$

Typical rules:⁴⁵

$$\frac{A \quad B}{A \wedge B} \text{ AND-I} \quad \frac{A}{A \vee B} \text{ OR-I} \quad \frac{A \Rightarrow B \quad A}{B} \text{ IMPL-E/MODUS PONENS}$$

Example 16.

$$\frac{\frac{x = 5 \quad y = 5}{x = 5 \wedge y = 5} \text{ AND-I} \quad \frac{x = 5}{x = 5 \vee y = 5} \text{ OR-I}}{x \geq 0 \Rightarrow y \geq 0 \quad x \geq 0} \text{ IMPL-E}$$

$$y \geq 0$$

⁴³axiom = rule with no premises

⁴⁴ $A(x)$ to indicate that, here, A (typically) contains x .

⁴⁵The “names” of the rules are written on the right of the rule, they serve for “identification”. By some convention, “I” stands for rules *introducing* some logical connector, “E” for *eliminating* one.

Important terms

- **Interpretation:** describe each formula as either *true* or *false*
- **Proof:** derivation tree where all leaf nodes are axioms
- **Theorems:** a “formula” derivable in a given proof system
- **Soundness** (of the logic): If we can prove (“derive”) some formula P (in the logic) then P is actually (semantically) true
- **Completeness:** If a formula P is true, it can be proven

Program Logic (PL)

- PL lets us *express* and *prove* properties about programs
- *Formulas* are of the form

“Hoare triple”

$$\{ P_1 \} S \{ P_2 \}$$

- S : program statement(s)
- $P, P_1, P', Q \dots$: assertions over program states (including $\neg, \wedge, \vee, \exists, \forall$)
- In above triple P_1 : pre-condition, and P_2 post-condition of S

Example 17.

$$\{ x = y \} \text{ x := x * 2; y := y * 2; } \{ x = y \}$$

The proof system PL (Hoare logic)

- Express and prove program properties
- $\{P\} S \{Q\}$
 - P, Q may be seen as a **specification** of the program S
 - Code analysis by proving the specification (in PL)
 - No need to execute the code in order to do the analysis
 - An **interpretation** maps triples to *true* or *false*
 - * $\{ x = 0 \} x := x + 1; \{ x = 1 \}$ should be *true*
 - * $\{ x = 0 \} x := x + 1; \{ x = 0 \}$ should be *false*

Reasoning about programs

- Basic idea: *Specify* what the program is supposed to do (pre- and post-conditions)
- Pre- and post-conditions are given as assertions over the program state
- use PL for a mathematical argument that the program satisfies its specification

Interpretation:

Interpretation (“semantics”) of triples is related to program execution

Partial correctness interpretation

$\{ P \} S \{ Q \}$ is true/holds:

- If the initial state of S satisfies P (P holds for the initial state of S) and
- **if**⁴⁶ S terminates,
- then Q is *true* in the final state of S

Expresses *partial correctness* (termination of S is assumed)

Example 18. $\{x = y\} \text{ x := x * 2; y := y * 2; } \{x = y\}$ is *true* if the initial state satisfies $x = y$ and, in case the execution terminates, then the final state satisfies $x = y$

⁴⁶Thus: if S does not terminate, all bets are off. . .

Examples

Some true triples

$$\begin{aligned}
 & \{ x = 0 \} x := x + 1; \{ x = 1 \} \\
 & \{ x = 4 \} x := 5; \{ x = 5 \} \\
 & \{ \text{true} \} x := 5; \{ x = 5 \} \\
 & \{ y = 4 \} x := 5; \{ y = 4 \} \\
 & \{ x = 4 \} x := x + 1; \{ x = 5 \} \\
 & \{ x = a \wedge y = b \} x = x + y; \{ x = a + b \wedge y = b \} \\
 & \{ x = 4 \wedge y = 7 \} x := x + 1; \{ x = 5 \wedge y = 7 \} \\
 & \{ x = y \} x := x + 1; y := y + 1; \{ x = y \}
 \end{aligned}$$

Some non-true triples

$$\begin{aligned}
 & \{ x = 0 \} x := x + 1; \{ x = 0 \} \\
 & \{ x = 4 \} x := 5; \{ x = 4 \} \\
 & \{ x = y \} x := x + 1; y := y - 1; \{ x = y \} \\
 & \{ x > y \} x := x + 1; y := y + 1; \{ x < y \}
 \end{aligned}$$

Partial correctness

- The interpretation of $\{ P \} S \{ Q \}$ assumes/ignores termination of S , termination is not proven.
- The pre/post specification (P, Q) express *safety* properties

The state assertion *true* can be viewed as all states. The assertion *false* can be viewed as no state. What does each of the following triple express?

$$\begin{aligned}
 \{ P \} S \{ \text{false} \} & \quad S \text{ does not terminate} \\
 \{ P \} S \{ \text{true} \} & \quad \text{trivially true} \\
 \{ \text{true} \} S \{ Q \} & \quad Q \text{ holds after } S \text{ in any case} \\
 & \quad \text{(provided } S \text{ terminates)} \\
 \{ \text{false} \} S \{ Q \} & \quad \text{trivially true}
 \end{aligned}$$

Proof system PL

A proof system consists of *axioms* and *rules*
here: structural analysis of programs

- Axioms for basic statements:
 - $x := e, \text{skip}, \dots$
- Rules for composed statements:
 - $S_1; S_2, \text{if}, \text{while}, \text{await}, \text{co} \dots \text{oc}, \dots$

Formulas in PL

- formulas = triples
- theorems = derivable formulas⁴⁷
- hopefully: all derivable formulas are also “really” (= semantically) true
- derivation: starting from *axioms*, using derivation *rules*
-

$$\frac{H_1 \quad H_2 \quad \dots \quad H_n}{C}$$

- axioms: can be seen as rules without premises

⁴⁷The terminology is standard from general logic. A “theorem” in an derivation system is a *derivable* formula. In an ill-defined (i.e., unsound) derivation or proof system, theorems may thus be not true.

Soundness

If a triple $\{ P \} S \{ Q \}$ is a *theorem* in PL (i.e., derivable), the triple holds

- Example: we want

$$\{ x = 0 \} x := x + 1 \{ x = 1 \}$$

to be a theorem (since it was interpreted as *true*),

- but

$$\{ x = 0 \} x := x + 1 \{ x = 0 \}$$

should *not* be a theorem (since it was interpreted as *false*)

Soundness:⁴⁸

All theorems in PL hold

$$\vdash \{ P \} S \{ Q \} \text{ implies } \models \{ P \} S \{ Q \} \quad (5)$$

If we can use PL to prove some property of a program, then this property will hold for [all](#) executions of the program.

Textual substitution

Substitution

$P[e/x]$ means, all free occurrences of x in P are replaced by expression e .

Example 19.

$$\begin{aligned} (x = 1)[(x + 1)/x] &\Leftrightarrow x + 1 = 1 \\ (x + y = a)[(y + x)/y] &\Leftrightarrow x + (y + x) = a \\ (y = a)[(x + y)/x] &\Leftrightarrow y = a \end{aligned}$$

Substitution propagates into formulas:

$$\begin{aligned} (\neg A)[e/x] &\Leftrightarrow \neg(A[e/x]) \\ (A \wedge B)[e/x] &\Leftrightarrow A[e/x] \wedge B[e/x] \\ (A \vee B)[e/x] &\Leftrightarrow A[e/x] \vee B[e/x] \end{aligned}$$

Free and “non-free” variable occurrences

$P[e/x]$

- Only *free* occurrences of x are substituted
- Variable occurrences may be *bound* by quantifiers, then that occurrence of the variable is not free (but bound)

Example 20 (Substitution).

$$\begin{aligned} (\exists y : x + y > 0)[1/x] &\Leftrightarrow \exists y : 1 + y > 0 \\ (\exists x : x + y > 0)[1/x] &\Leftrightarrow \exists x : x + y > 0 \\ (\exists x : x + y > 0)[x/y] &\Leftrightarrow \exists z : z + x > 0 \end{aligned}$$

Correspondingly for \forall

The assignment axiom – Motivation

Given by backward construction over the assignment:

- Given the postcondition to the assignment, we may derive the precondition!

What is the precondition?

$$\{ ? \} x := e \{ x = 5 \}$$

If the assignment $x = e$ should terminate in a state where x has the value 5, the expression e must have the value 5 before the assignment:

$$\begin{aligned} \{ e = 5 \} \quad x := e \quad \{ x = 5 \} \\ \{ (x = 5)[e/x] \} \quad x := e \quad \{ x = 5 \} \end{aligned}$$

⁴⁸technically, we'd need a semantics for reference, otherwise it's difficult to say what a program “really” does.

Axiom of assignment

“Backwards reasoning.” Given a postcondition, we may construct the precondition:

Axiom for the assignment statement

$$\{ P[e/x] \} x := e \{ P \} \quad \text{ASSIGN}$$

If the assignment $x := e$ should lead to a state that satisfies P , the state before the assignment must satisfy P where x is replaced by e .

Proving an assignment

To prove the triple $\{ P \} x := e \{ Q \}$ in PL, we must show that the precondition P implies $Q[e/x]$

$$\frac{P \Rightarrow Q[e/x] \quad \{ Q[e/x] \} x := e \{ Q \}}{\{ P \} x := e \{ Q \}}$$

The blue implication is a logical proof obligation. In this course we only convince ourselves that these are true (we do not prove them formally).

- $Q[e/x]$ is the largest set of states such that the assignment is guaranteed to terminate with Q
- largest set corresponds to **weakest condition** \Rightarrow **weakest-precondition** reasoning
- We must show that the set of states P is within this set

Examples

$$\frac{true \Rightarrow 1 = 1}{\{ true \} x := 1 \{ x = 1 \}}$$

$$\frac{x = 0 \Rightarrow x + 1 = 1}{\{ x = 0 \} x := x + 1 \{ x = 1 \}}$$

$$\frac{(x = a \wedge y = b) \Rightarrow x + y = a + b \wedge y = b}{\{ x = a \wedge y = b \} x := x + y \{ x = a + b \wedge y = b \}}$$

$$\frac{x = a \Rightarrow 0 * y + x = a}{\{ x = a \} q := 0 \{ q * y + x = a \}}$$

$$\frac{y > 0 \Rightarrow y \geq 0}{\{ y > 0 \} x := y \{ x \geq 0 \}}$$

Axiom of skip

The skip statement does nothing

Axiom:

$$\{ P \} \text{skip} \{ P \} \quad \text{SKIP}$$

PL inference rules

$$\begin{array}{c}
\frac{\{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\{P\} S_1; S_2 \{Q\}} \text{SEQ} \\
\frac{\{P \wedge B\} S \{Q\} \quad P \wedge \neg B \Rightarrow Q}{\{P\} \text{ if } B \text{ then } S \{Q\}} \text{COND}' \\
\frac{\{I \wedge B\} S \{I\}}{\{I\} \text{ while } B \text{ do } S \{I \wedge \neg B\}} \text{WHILE} \\
\frac{P' \Rightarrow P \quad \{P\} S \{Q\} \quad Q \Rightarrow Q'}{\{P'\} S \{Q'\}} \text{CONSEQUENCE} \\
\frac{\{P\} S \{Q\} \quad \{P'\} S \{Q'\}}{\{P \wedge P'\} S \{Q \wedge Q'\}} \text{CONJUNCTION}
\end{array}$$

- **Blue**: logical proof obligations
- the rule for **while** needs a *loop invariant*!
- **for-loop**: exercise 2.22!

Sequential composition and consequence

Backward construction over assignments:

$$\frac{x = y \Rightarrow 2x = 2y}{\frac{\{x = y\} x := 2x \{x = 2y\} \quad \{(x = y)[2y/y]\} y := 2y \{x = y\}}{\{x = y\} x := 2x; y := 2y \{x = y\}}}$$

Sometimes we don't bother to write down the assignment axiom:

$$\frac{(q * y) + x = a \Rightarrow ((q + 1) * y) + x - y = a}{\frac{\{(q * y) + x = a\} x := x - y; \{((q + 1) * y) + x = a\}}{\{(q * y) + x = a\} x := x - y; q := q + 1 \{(q * y) + x = a\}}}$$

Logical variables

- Do *not* occur in program text
- Used only in *assertions*
- May be used to “freeze” initial values of variables
- May then talk about these values in the postcondition

Example 21.

$$\{x = x_0\} \text{ if } (x < 0) \text{ then } x := -x \{x \geq 0 \wedge (x = x_0 \vee x = -x_0)\}$$

where $(x = x_0 \vee x = -x_0)$ states that

- the final value of x equals the initial value, *or*
- the final value of x is the negation of the initial value

Example: if statement

Verification of:

$$\{ x = x_0 \} \text{ if } (x < 0) \text{ then } x := -x \{ x \geq 0 \wedge (x = x_0 \vee x = -x_0) \}$$

$$\frac{\{ P \wedge B \} S \{ Q \} \quad (P \wedge \neg B) \Rightarrow Q}{\{ P \} \text{ if } B \text{ then } S \{ Q \}} \text{COND'}$$

- $\{ P \wedge B \} S \{ Q \}$: $\{ x = x_0 \wedge x < 0 \} x := -x \{ x \geq 0 \wedge (x = x_0 \vee x = -x_0) \}$ Backward construction (assignment axiom) gives the implication:

$$x = x_0 \wedge x < 0 \Rightarrow (-x \geq 0 \wedge (-x = x_0 \vee -x = -x_0))$$

- $P \wedge \neg B \Rightarrow Q$: $x = x_0 \wedge x \geq 0 \Rightarrow (x \geq 0 \wedge (x = x_0 \vee x = -x_0))$

03. 10. 2016

7 Program Analysis

Program Logic (PL)

- PL lets us *express* and *prove* properties about programs
- *Formulas* are on the form

“triple” (or “Hoare triple”)

$$\{ P \} S \{ Q \}$$

- S : program statement(s)
- P and Q : assertions over program states
- P : Pre-condition
- Q : Post-condition

If we can use PL to prove some property of a program, then this property will hold for all executions of the program

PL rules from last week

$$\begin{array}{c} \frac{\{ P \} S_1 \{ R \} \quad \{ R \} S_2 \{ Q \}}{\{ P \} S_1; S_2 \{ Q \}} \text{SEQ} \\ \frac{\{ P \wedge B \} S \{ Q \} \quad P \wedge \neg B \Rightarrow Q}{\{ P \} \text{ if } B \text{ then } S \{ Q \}} \text{COND'} \\ \frac{\{ I \wedge B \} S \{ I \}}{\{ I \} \text{ while } B \text{ do } S \{ I \wedge \neg B \}} \text{WHILE} \\ \frac{P' \Rightarrow P \quad \{ P \} S \{ Q \} \quad Q \Rightarrow Q'}{\{ P' \} S \{ Q' \}} \text{CONSEQUENCE} \\ \frac{\{ P \} S \{ Q \} \quad \{ P' \} S \{ Q' \}}{\{ P \wedge P' \} S \{ Q \wedge Q' \}} \text{CONJUNCTION} \end{array}$$

How to actually *use* the while rule?

- Cannot control the execution in the same manner as for **if**-statements
 - Cannot tell *from the code* how many times the loop body will be executed (not a “syntax-directed” rule)

$$\{ y \geq 0 \} \text{ while } (y > 0) \ y := y - 1$$
 - Cannot speak about the state after the first, second, third ... iteration
- **Solution:** Find an assertion I that is maintained by the loop body
 - *Loop invariant:* express a property preserved by the loop
- Often hard to find suitable loop invariants
 - The course is *not* an exercise in finding complicated invariants
 - “suitable:
 1. must be preserved by the body, i.e., it must be actually an invariant
 2. must be strong enough to imply the desired post-condition
 3. Note: both “true” and “false” are loop invariants for partial correctness! Both typically fail to be suitable (i.e. they are basically useless invariants).

While rule

$$\frac{\{ I \wedge B \} S \{ I \}}{\{ I \} \text{ while } B \text{ do } S \{ I \wedge \neg B \}} \text{ WHILE}$$

Can use this rule to reason about the general situation:

$$\{ P \} \text{ while } B \text{ do } S \{ Q \}$$

where

- P need not be the loop invariant
- Q need not match $(I \wedge \neg B)$ syntactically

Combine WHILE-rule with CONSEQUENCE-rule to prove:

- **Entry:** $P \Rightarrow I$
- **Loop:** $\{ I \wedge B \} S \{ I \}$
- **Exit:** $I \wedge \neg B \Rightarrow Q$

While rule: example

$$\{ 0 \leq n \} \ k := 0; \{ k \leq n \} \text{ while } (k < n) \ k := k + 1; \{ k = n \}$$

Composition rule splits a proof in two: assignment and loop. Let $k \leq n$ be the loop invariant

- **Entry:** $k \leq n$ follows from itself
- **Loop:**

$$\frac{k < n \Rightarrow k + 1 \leq n}{\{ k \leq n \wedge k < n \} \ k := k + 1 \{ k \leq n \}}$$

- **Exit:** $(k \leq n \wedge \neg(k < n)) \Rightarrow k = n$

Await statement

Rule for await

$$\frac{\{ P \wedge B \} S \{ Q \}}{\{ P \} \langle \text{await}(B) S \rangle \{ Q \}} \text{AWAIT}$$

Remember: we are reasoning about safety properties/partial correctness

- **termination** is assumed/ignored
- the rule does not ensure waiting nor progress

Concurrent execution

Assume two statements S_1 and S_2 such that:

$$\{ P_1 \} \langle S_1 \rangle \{ Q_1 \} \quad \text{and} \quad \{ P_2 \} \langle S_2 \rangle \{ Q_2 \}$$

Note: to avoid further complications right now: S_i 's are enclosed into “ \langle atomic brackets \rangle ”.
First **attempt** for a $\text{co} \dots \text{oc}$ rule in PL:

$$\frac{\{ P_1 \} \langle S_1 \rangle \{ Q_1 \} \quad \{ P_2 \} \langle S_2 \rangle \{ Q_2 \}}{\{ P_1 \wedge P_2 \} \text{co} \langle S_1 \rangle \parallel \langle S_2 \rangle \text{oc} \{ Q_1 \wedge Q_2 \}} \text{PAR}$$

Example 22 (Problem with this rule).

$$\frac{\{ x = 0 \} \langle x := x + 1 \rangle \{ x = 1 \} \quad \{ x = 0 \} \langle x := x + 2 \rangle \{ x = 2 \}}{\{ x = 0 \} \text{co} \langle x := x + 1 \rangle \parallel \langle x := x + 2 \rangle \text{oc} \{ x = 1 \wedge x = 2 \}}$$

but this conclusion is not true: the postcondition should be $x = 3$!

1>This first attempt may seem plausible. One has two programs, both with its “own” precondition. Therefore, if they run in parallel, they start in a common state, obviously. That may be characterized by the conjunction. Alternatively, one may use the *same* precondition. There is not much difference between those two ways of thinking (due to strengthening of preconditions). Indeed, the **precondition** in this line of reasoning is not problematic. Note however, that conceptually we are thinking in a *forward* way, we are not currently reason like “assume you are in a given post-state, ...”. But the forward reasoning fits better to the following illustrating example. 2>Different ways to analyze what’s exactly wrong. But the important observation is: **that it’s plain wrong**. Remember **Soundness**. The break of soundness is illustrated by the following example. Linear logic, resources: “I win 100 dollar \wedge I win 100 dollar”. The rule, if it were true, would be nice: compositionality. For independent variables (i.e., local ones) it would be true. So, the reason, why concurrency is hard/compositional reasoning does not work, are **shared variables**. The absence of such problem will later be called **interference free**. It will not be defined for processes alone, but for specifications insofar: an annotated program is interference free if the *conditions* of parallel processes are not disturbed. It’s like an invariant.

Interference problem

$$S_1 \quad \{ x = 0 \} \langle x := x + 1 \rangle \{ x = 1 \}$$

$$S_2 \quad \{ x = 0 \} \langle x := x + 2 \rangle \{ x = 2 \}$$

- execution of S_2 interferes with pre- and postconditions of S_1
 - The assertion $x = 0$ need not hold when S_1 starts execution
- execution of S_1 interferes with pre- and postconditions of S_2
 - The assertion $x = 0$ need not hold when S_2 starts execution

Solution: *weaken* the assertions to account for the other process:

$$S_1 \quad \{ x = 0 \vee x = 2 \} \langle x := x + 1 \rangle \{ x = 1 \vee x = 3 \}$$

$$S_2 \quad \{ x = 0 \vee x = 1 \} \langle x := x + 2 \rangle \{ x = 2 \vee x = 3 \}$$

Interference problem

Apply the previous “parallel-composition-is-conjunction” rule again:

$$\frac{\begin{array}{c} \{ x = 0 \vee x = 2 \} \langle x := x + 1 \rangle \{ x = 1 \vee x = 3 \} \\ \{ x = 0 \vee x = 1 \} \langle x := x + 2 \rangle \{ x = 2 \vee x = 3 \} \end{array}}{\{ PRE \} \text{ co } \langle x := x + 1 \rangle \parallel \langle x := x + 2 \rangle \text{ oc } \{ POST \}}$$

where:

$$PRE : (x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)$$

$$POST : (x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)$$

which gives:

$$\{ x = 0 \} \text{ co } \parallel x = x + 1 \parallel \langle x := x + 2 \rangle \text{ oc } \{ x = 3 \}$$

Concurrent execution

Assume $\{ P_i \} S_i \{ Q_i \}$ for all S_1, \dots, S_n

$$\frac{\{ P_i \} S_i \{ Q_i \} \text{ are interference free}}{\{ P_1 \wedge \dots \wedge P_n \} \text{ co } S_1 \parallel \dots \parallel S_n \text{ oc } \{ Q_1 \wedge \dots \wedge Q_n \}} \text{Cooc}$$

Interference freedom

A process **interferes** with (the specification of) another process, if its execution changes the assertions⁴⁹ of the other process.

- assertions inside awaits: not endangered
- *critical assertions* or **critical conditions**: assertions outside await statement bodies.⁵⁰

Interference freedom

Interference freedom

- S : statement in some process, with pre-condition $pre(S)$
- C : critical assertion in **another** process
- S **does not interfere** with C , if

$$\vdash \{ C \wedge pre(S) \} S \{ C \}$$

is derivable in PL (= theorem).

“ C is *invariant* under the execution of the other process”

$$\frac{\{ P_1 \} S_1 \{ Q_1 \} \quad \{ P_2 \} S_2 \{ Q_2 \}}{\{ P_1 \wedge P_2 \} \text{ co } S_1 \parallel S_2 \text{ oc } \{ Q_1 \wedge Q_2 \}}$$

Four interference freedom requirements:

$$\frac{\{ P_2 \wedge P_1 \} S_1 \{ P_2 \} \quad \{ P_1 \wedge P_2 \} S_2 \{ P_1 \}}{\{ Q_2 \wedge P_1 \} S_1 \{ Q_2 \} \quad \{ Q_1 \wedge P_2 \} S_2 \{ Q_1 \}}$$

⁴⁹Only “critical assertions” considered

⁵⁰More generally one could say: outside mutex-protected sections.

“Avoiding” interference: Weakening assertions

$$\begin{aligned} S_1 : \{ x = 0 \} < x := x + 1; > \{ x = 1 \} \\ S_2 : \{ x = 0 \} < x := x + 2; > \{ x = 2 \} \end{aligned}$$

Here we have interference, for instance the precondition of S_1 is not maintained by execution of S_2 :

$$\{ (x = 0) \wedge (x = 0) \} x := x + 2 \{ x = 0 \}$$

is not true

However, after **weakening**:

$$\begin{aligned} S_1 : \{ x = 0 \vee x = 2 \} < x := x + 1; > \{ x = 1 \vee x = 3 \} \\ S_2 : \{ x = 0 \vee x = 1 \} < x := x + 2; > \{ x = 2 \vee x = 3 \} \end{aligned}$$

$$\{ (x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1) \} x := x + 2 \{ x = 0 \vee x = 2 \}$$

(Correspondingly for the other three critical conditions)

Avoiding interference: Disjoint variables

- *V set*: global variables referred to (i.e. read or written) by a process
- *W set*: global variables written to by a process
- *Reference set*: global variables in critical (Hoare) assertions/conditions of one process

S_1 and S_2 : in 2 different processes. **No interference**, if:

- *W set* of S_1 is disjoint from reference set of S_2
- *W set* of S_2 is disjoint from reference set of S_1

Alas: variables in a critical condition of one process will often be among the written variables of another

Avoiding interference: Global invariants

Global inductive invariants

- Some condition that only refers to global (shared) variables
- Holds initially.
- Preserved by all assignments/transitions (“inductive”)

“Separation of concerns: We **avoid interference** if critical conditions are on the form $\{ I \wedge L \}$ where:

- I is a global invariant
- L only refers to local variables of the considered process

Avoiding interference: Synchronization

- Hide critical conditions
- MUTEX to critical sections

$$\text{co } \dots; S; \dots \parallel \dots; S_1; \{ C \} S_2; \dots \text{oc}$$

S might interfere with C Hide the critical condition by a critical region:

$$\text{co } \dots; S; \dots \parallel \dots; \langle S_1; \{ C \} S_2 \rangle; \dots \text{oc}$$

Example: Producer/ consumer synchronization

Let process **Producer** deliver data to a **Consumer** process

$$PC : c \leq p \leq c + 1 \wedge (p = c + 1) \Rightarrow (buf = a[p - 1])$$

PC a global inductive invariant of the producer/consumer?

```

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

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

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

```

Example: Producer

Loop invariant of Producer: $I_P : PC \wedge p \leq n$

```

process Producer {
  int a[n];
  {  $I_P$  } // entering loop
  while (p < n) {
    < await (p = c) ; > {  $I_P \wedge p < n$  }
    buf := a[p]; {  $I_P \wedge p < n \wedge p = c$  }
    p := p + 1; {  $I_P[p + 1/p][a[p]/buf]$  }
  } {  $I_P$  } {  $I_P[p + 1/p]$  }
  } {  $I_P \wedge \neg(p < n)$  } // exit loop
  } {  $I_P$  }
  }  $\Leftrightarrow \{PC \wedge p = n\}$ 
}

```

Proof obligation: $I_P \wedge p < n \wedge p = c \Rightarrow I_P[p + 1/p][a[p]/buf]$

Example: Consumer

Loop invariant of Consumer: $I_C : PC \wedge c \leq n \wedge b[0 : c - 1] = a[0 : c - 1]$

```

process Consumer {
  int b[n];
  {  $I_C$  } // entering loop
  while (c < n) {
    < await (p > c) ; > {  $I_C \wedge c < n$  }
    b[c] := buf; {  $I_C \wedge c < n \wedge p > c$  }
    c := c + 1; {  $I_C[c + 1/c][buf/b[c]]$  }
  } {  $I_C$  } {  $I_C[c + 1/c]$  }
  } {  $I_C \wedge \neg(c < n)$  } // exit loop
  } {  $I_C$  }
  }  $\Leftrightarrow \{PC \wedge c = n \wedge b[0 : c - 1] = a[0 : c - 1]\}$ 
}

```

Proof Obligation: $I_C \wedge c < n \wedge p > c \Rightarrow I_C[c + 1/c][buf/b[c]]$

Example: Producer/Consumer

The final state of the program satisfies:

$$PC \wedge p = n \wedge c = n \wedge b[0 : c - 1] = a[0 : c - 1]$$

which ensures that all elements in **a** are received and occur in the same order in **b**

Interference freedom is ensured by the global invariant and **await**-statements

Combining the two assertions after the **await** statements, we get:

$$I_P \wedge p < n \wedge p = c \wedge I_C \wedge c < n \wedge p > c$$

which gives **false**! At any time, only one process can be after the await statement!

Monitor invariant

```
monitor name {
  monitor variables      # shared global variable
  initialization         # for the monitor's procedures
  procedures
}
```

- A monitor invariant (I): describe the monitor's inner state
- Express relationship between monitor variables
- Maintained by execution of procedures:
 - Must hold after initialization
 - Must hold when a procedure terminates
 - Must hold when we **suspend** execution due to a call to **wait**
 - Can **assume** that the invariant holds *after wait* and when a procedure starts
- Should be as *strong* as possible!

Axioms for signal and continue (1)

Assume that the monitor invariant I and predicate P *does not* mention cv . Then we can set up the following axioms:

$$\begin{array}{ll} \{ I \} \text{wait}(cv) \{ I \} & \\ \{ P \} \text{signal}(cv) \{ P \} & \text{for arbitrary } P \\ \{ P \} \text{signal_all}(cv) \{ P \} & \text{for arbitrary } P \end{array}$$

Monitor solution to reader/writer problem

Verification of the invariant over `request_read`

$$I : (nr = 0 \vee nw = 0) \wedge nw \leq 1$$

```
procedure request_read() {
  { I }
  while (nw > 0) {      { I ∧ nw > 0 }
    { I } wait(oktoread); { I }
  } { I ∧ nw = 0 }
  { I[nr + 1/nr] }
  nr := nr + 1;
  { I }
}
```

$(I \wedge nw > 0) \Rightarrow I (I \wedge nw = 0) \Rightarrow I[nr + 1/nr]$ 1>The invariant we had earlier already, it's the obvious one.

Axioms for Signal and Continue (2)

Assume that the invariant can mention the number of processes in the queue to a condition variable.

- Let $\#cv$ be the number of proc's waiting in the queue to cv .
- The test `empty(cv)` thus corresponds to $\#cv = 0$

`wait(cv)` is *modelled* as an extension of the queue followed by processor release:

$$\text{wait}(cv) : \{ ? \} \#cv := \#cv + 1; \{ I \} \text{"sleep"} \{ I \}$$

by **assignment** axiom:

$$\text{wait}(cv) : \{ I[\#cv + 1/\#cv]; \#cv := \#cv + 1; \{ I \} \text{"sleep"} \{ I \}$$

Axioms for Signal and Continue (3)

$signal(cv)$ can be modelled as a reduction of the queue, if the queue is not empty:

$$signal(cv) : \{ ? \} \text{ if } (\#cv \neq 0) \#cv := \#cv - 1 \{ P \}$$

$$signal(cv) : \{ ((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P[\#cv - 1/\#cv]) \} \\ \text{if } (\#cv \neq 0) \#cv := \#cv - 1 \\ \{ P \}$$

- $signal_all(cv) : \{ P[0/\#cv] \} \#cv := 0 \{ P \}$

Axioms for Signal and Continue (4)

Together this gives:

Axioms for monitor communication

$$\begin{aligned} & \{ I[\#cv + 1/\#cv] \} wait(cv) \{ I \} \quad \text{WAIT} \\ & \{ ((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P[\#cv - 1/\#cv]) \} signal(cv) \{ P \} \quad \text{SIGNAL} \\ & \{ P[0/\#cv] \} signal_all(cv) \{ P \} \quad \text{SIGNALALL} \end{aligned}$$

If we know that $\#cv \neq 0$ whenever we signal, then the axiom for $signal(cv)$ be simplified to:

$$\{ P[\#cv - 1/\#cv] \} signal(cv) \{ P \}$$

Note! $\#cv$ is not allowed in statements! Only used for reasoning

Example: FIFO semaphore verification (1)

```
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);
  }
}
```

Consider the following monitor invariant:

$$s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

No process is waiting if the semaphore value is positive!

1>The example is from the monitor chapter. This is a monitor solution for FIFO-semaphores, even under the weak s&c signalling discipline. It's "forwarding the condition"

Example: FIFO semaphore verification: Psem

$$I : s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Psem() {
  {I}
  if (s=0) {I ∧ s = 0}
    {I[#pos + 1/#pos]} wait(pos); {I}
  else {I ∧ s ≠ 0}
    {I[s - 1/s]} s := s-1; {I}
  {I}
}
```

Example: FIFO semaphore verification (3)

$$I : s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

This gives two proof obligations: If-branch:

$$\begin{array}{ll} (I \wedge s = 0) & \Rightarrow I[\#pos + 1/\#pos] \\ s = 0 & \Rightarrow s \geq 0 \wedge (s > 0 \Rightarrow \#pos + 1 = 0) \\ s = 0 & \Rightarrow s \geq 0 \end{array}$$

Else branch:

$$\begin{array}{ll} (I \wedge s \neq 0) & \Rightarrow I[s - 1/s] \\ (s > 0 \wedge \#pos = 0) & \Rightarrow s - 1 \geq 0 \wedge (s - 1 > 0 \Rightarrow \#pos = 0) \\ (s > 0 \wedge \#pos = 0) & \Rightarrow s > 0 \wedge \#pos = 0 \end{array}$$

Example: FIFO semaphore verification: Vsem

$$I : s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

```
procedure Vsem() {  
  {I}  
  if empty(pos) {I ∧ #pos = 0}  
    {I[s + 1/s]} s:=s+1; {I}  
  else {I ∧ #pos ≠ 0}  
    {I[#pos - 1/#pos]} signal(pos); {I}  
  {I}  
}
```

Example: FIFO semaphore verification (5)

$$I : s \geq 0 \wedge (s > 0 \Rightarrow \#pos = 0)$$

As above, this gives two proof obligations: If-branch:

$$\begin{array}{ll} (I \wedge \#pos = 0) & \Rightarrow I[s + 1/s] \\ (s \geq 0 \wedge \#pos = 0) & \Rightarrow s + 1 \geq 0 \wedge (s + 1 > 0 \Rightarrow \#pos = 0) \\ (s \geq 0 \wedge \#pos = 0) & \Rightarrow s + 1 \geq 0 \wedge \#pos = 0 \end{array}$$

Else branch:

$$\begin{array}{ll} (I \wedge \#pos \neq 0) & \Rightarrow I[\#pos - 1/\#pos] \\ (s = 0 \wedge \#pos \neq 0) & \Rightarrow s \geq 0 \wedge (s > 0 \Rightarrow \#pos - 1 = 0) \\ s = 0 & \Rightarrow s \geq 0 \end{array}$$

8 Java concurrency

10. 10. 2016

8.1 Threads in Java

Outline

1. [Monitors](#): review
2. [Threads in Java](#):
 - Thread classes and Runnable interfaces
 - Interference and Java threads
 - [Synchronized blocks and methods](#): (atomic regions and monitors)
3. Example: The ornamental garden
4. Thread communication & condition synchronization (wait and signal/notify)
5. Example: Mutual exclusion
6. Example: Readers/writers

Short recap of monitors

- monitor **encapsulates** data, which can only be **observed** and **modified** by the monitor's procedures
 - Contains variables that describe the *state*
 - variables can be accessed/changed only through the available *procedures*
- Implicit mutex: Only one procedure may be active at a time.
 - 2 procedures in the same monitor: never executed concurrently
- **Condition synchronization**: **block** a process **until** a particular **condition holds**, achieved through *condition variables*.

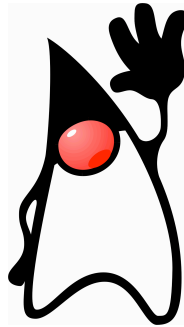
Signaling disciplines

- **Signal and wait (SW)**: the signaller waits, and the signalled process gets to execute immediately
- **Signal and continue (SC)**: the signaller continues, and the signalled process executes later

Java

From Wikipedia.⁵¹

"... Java is a general-purpose, *concurrent*, class-based, object-oriented language ..."



Threads in Java

A **thread** in Java

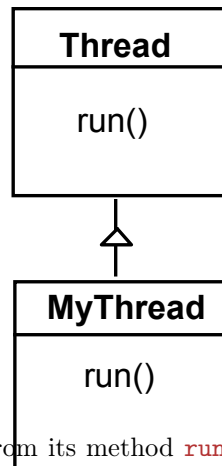
- unit of concurrency⁵²
- originally “green threads”
- identity, accessible via static method `Thread.currentThread()`⁵³
- has its own stack / execution context
- access to shared state
- shared mutable state: heap structured into objects
 - privacy restrictions possible
 - what are **private** fields?
- may be **created** (and “deleted”) dynamically

⁵¹But it's correct nonetheless ...

⁵²as such, roughly corresponding to the concept of “processes” from previous lectures.

⁵³What's the difference to **this**?

Thread class

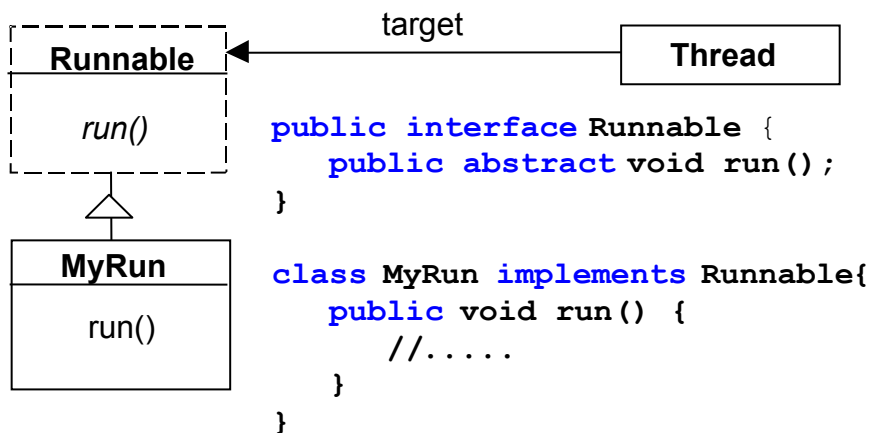


The `Thread` class executes instructions from its method `run()`. The actual code executed depends on the implementation provided for `run()` in a derived class.

```
class MyThread extends Thread {
    public void run() {
        //.....
    }
}
// Creating a thread object:
Thread a = new MyThread();
a.start();
```

Runnable interface

no multiple inheritance \Rightarrow , often implement the `run()` method in a class not derived from `Thread` but from the interface `Runnable`.



```
// Creating a thread object:
Runnable b = new MyRun();
new Thread(b).start();
```

Threads in Java

steps to *create* a thread and get it running:

1. Define class that
 - *extends* the Java `Thread` class or
 - *implements* the `Runnable` interface
2. define `run` method inside the new class⁵⁴
3. create an instance of the new class.
4. *start* the thread.

⁵⁴overriding, late-binding.

Interference and Java threads

```
...
class Store {
    private int data = 0;
    public void update() { data++; }
}
...

// in a method:
Store s = new Store(); // the threads below have access to s
t1 = new FooThread(s); t1.start();
t2 = new FooThread(s); t2.start();
```

t1 and **t2** execute `s.update()` concurrently!

Interference between **t1** and **t2** \Rightarrow may lose updates to `data`.

Synchronization

avoid interference \Rightarrow threads “synchronize” access to shared data

1. One **unique lock** for each object *o*.
2. mutex: at most one thread *t* can lock *o* at any time.⁵⁵
3. two “flavors” of synchronization:

“synchronized block”

```
synchronized (o) { B }
```

synchronized method

whole method body of *m* “protected”⁵⁶:

```
synchronized Type m(...) { ... }
```

Protecting the initialization

Solution to earlier problem: **lock** the `Store` objects before executing problematic method:

```
class Store {
    private int data = 0;

    public void update() {
        synchronized (this) { data++; }
    }
}
```

or

```
class Store {
    private int data = 0;

    public synchronized void update() { data++; }
}
...

// inside a method:
Store s = new Store();
```

Java Examples

Book:

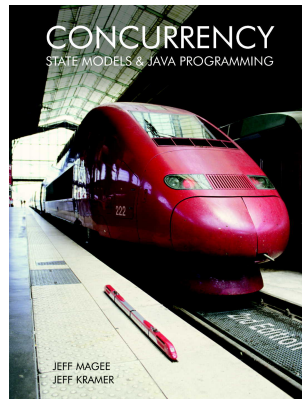
Concurrency: State Models & Java Programs, 2nd Edition

Jeff Magee & Jeff Kramer

Wiley

⁵⁵but: in a re-entrant manner!

⁵⁶assuming that other methods play according to the rules as well etc.



<http://www.doc.ic.ac.uk/~jnm/book/>

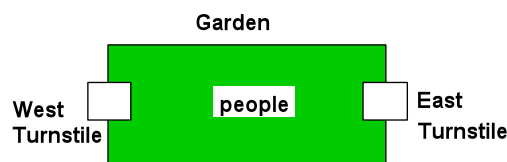
Examples in Java:

http://www.doc.ic.ac.uk/~jnm/book/book_applets

8.2 Ornamental garden

Ornamental garden problem

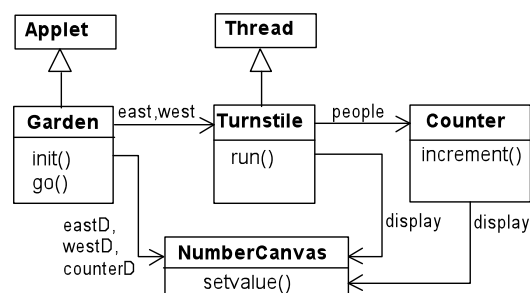
- people enter an ornamental garden through either of 2 **turnstiles**.
- problem: the number of people present at any time.



The concurrent program consists of:

- 2 threads
- shared counter object

Ornamental garden problem: Class diagram



The **Turnstile** thread simulates the periodic arrival of a visitor to the garden every second by sleeping for a second and then invoking the **increment()** method of the **counter** object.

Counter

```
class Counter {
    int value = 0;
    NumberCanvas display;

    Counter(NumberCanvas n) {
        display = n;
        display.setvalue(value);
    }

    void increment() {
        int temp = value;           // read[v]
        Simulate.HWinterrupt();
        value = temp + 1;           // write[v+1]
        display.setvalue(value);
    }
}
```

Turnstile

```
class Turnstile extends Thread {
    NumberCanvas display; // interface
    Counter people;       // shared data

    Turnstile(NumberCanvas n, Counter c) { // constructor
        display = n;
        people = c;
    }

    public void run() {
        try {
            display.setvalue(0);
            for (int i = 1; i <= Garden.MAX; i++) {
                Thread.sleep(500); // 0.5 second
                display.setvalue(i);
                people.increment(); // increment the counter
            }
        } catch (InterruptedException e) { }
    }
}
```

Ornamental Garden Program

The **Counter** object and **Turnstile** threads are created by the `go()` method of the **Garden** applet:

```
private void go() {
    counter = new Counter(counterD);
    west = new Turnstile(westD, counter);
    east = new Turnstile(eastD, counter);
    west.start();
    east.start();
}
```

Ornamental Garden Program: DEMO



DEMO

After the **East** and **West** turnstile threads have each **incremented** its counter **20** times, the garden people counter is **not the sum** of the counts displayed. **Counter** increments have been lost. **Why?**

http://www.doc.ic.ac.uk/~jnm/book/book_applets/Garden.html

Avoid interference by synchronization

```
class SynchronizedCounter extends Counter {  
    SynchronizedCounter(NumberCanvas n) {  
        super(n);  
    }  
    synchronized void increment() {  
        super.increment();  
    }  
}
```

Mutual Exclusion: The Ornamental Garden - DEMO



DEMO

8.3 Thread communication, monitors, and signaling

Monitors

- *each* object
 - has attached to it a unique *lock*
 - and thus: can act as *monitor*
- 3 important monitor operations⁵⁷
 - *o.wait()*: release lock on *o*, enter *o*'s wait queue and wait
 - *o.notify()*: wake up one thread in *o*'s wait queue
 - *o.notifyAll()*: wake up all threads in *o*'s wait queue
- executable by a thread “inside” the monitor represented by *o*
- executing thread must hold the lock of *o*/ executed within *synchronized* portions of code
- typical use: `this.wait()` etc.
- note: notify does *not* operate on a thread-identity⁵⁸

⇒

```
Thread t = new MyThread();  
...  
t.notify(); // mostly to be nonsense
```

⁵⁷there are more

⁵⁸technically, a thread identity is represented by a “thread object” though. Note also : `Thread.suspend()` and `Thread.resume()` are deprecated.

Condition synchronization, scheduling, and signaling

- quite **simple**/weak form of monitors in Java
- **only one** (implicit) condition variable per object: availability of the lock. threads that wait on *o* (*o.wait()*) are in this queue
- no built-in support for general-purpose condition variables.
- ordering of wait “queue”: implementation-dependent (usually FIFO)
- signaling discipline: **S & C**
- awakened thread: **no** advantage in competing for the lock to *o*.
- note: monitor-protection not enforced (!)
 - **private** field modifier \neq instance private
 - not all methods need to be synchronized⁵⁹
 - besides that: there’s **re-entrance!**

A semaphore implementation in Java

```
// down() = P operation
// up()   = V operation

public class Semaphore {
    private int value;

    public Semaphore (int initial) {
        value = initial;
    }

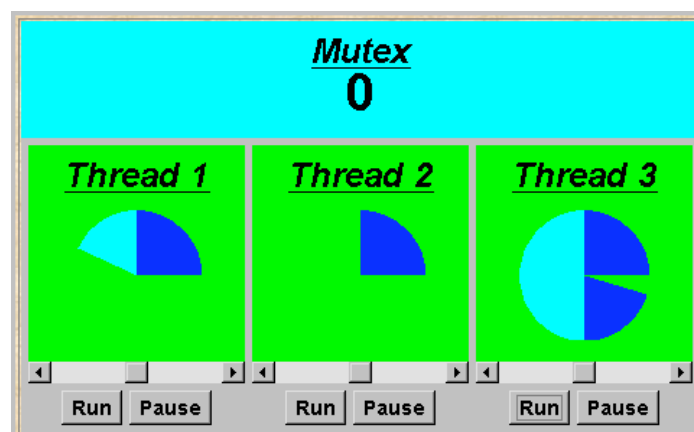
    synchronized public void up() {
        ++value;
        notifyAll();
    }

    synchronized public void down() throws InterruptedException {
        while (value == 0) wait(); // the well-known while-cond-wait pattern
        --value;
    }
}
```

- cf. also `java.util.concurrent.Semaphore` (acquire/release + more methods)

8.4 Semaphores

Mutual exclusion with semaphores



The graphics shows waiting and active phases, plus value of mutex. **Note:** Light blue for active phase, other colours for waiting.

⁵⁹remember: find of oblig-1.

Mutual exclusion with semaphores

```
class MutexLoop implements Runnable {  
    Semaphore mutex;  
  
    MutexLoop (Semaphore sema) {mutex=sema;}  
  
    public void run() {  
        try {  
            while(true) {  
                while(!ThreadPanel.rotate());  
                // get mutual exclusion  
                mutex.down();  
                while(ThreadPanel.rotate()); //critical section  
                //release mutual exclusion  
                mutex.up();  
            }  
        } catch (InterruptedException e){}  
    }  
}
```

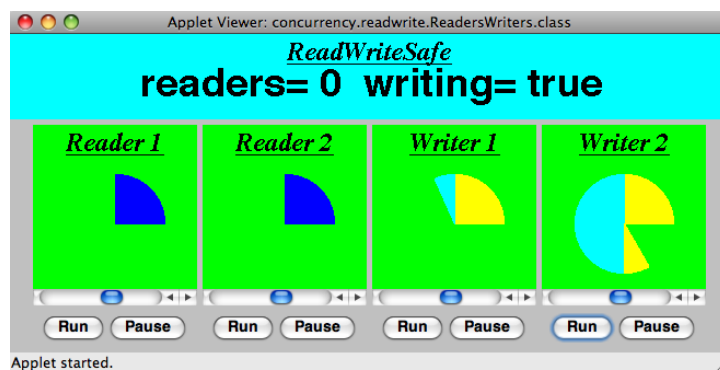
DEMO

Panel is an (old) AWT class (applet is a subclass). It's the simplest container class. The function `rotate` returns a *boolean*. It's a static method of the thread subclass `DisplayThread`.

http://www.doc.ic.ac.uk/~jnm/book/book_applets/Garden.html

8.5 Readers and writers

Readers and writers problem (again...)



A shared database is accessed by two kinds of processes. Readers execute transactions that examine the database while Writers both examine and update the database. A Writer must have exclusive access to the database; any number of Readers may concurrently access it.

Interface R/W

```
interface ReadWrite {  
    public void acquireRead() throws InterruptedException;  
    public void releaseRead();  
    public void acquireWrite() throws InterruptedException;  
    public void releaseWrite();  
}
```

Reader client code

```
class Reader implements Runnable {  
    ReadWrite monitor_  
  
    Reader(ReadWrite monitor) {  
        monitor_ = monitor;  
    }  
}
```

```

    }

    public void run() {
        try {
            while(true) {
                while(!ThreadPanel.rotate());
                // begin critical section
                monitor_.acquireRead();
                while(ThreadPanel.rotate());
                monitor_.releaseRead();
            }
        } catch (InterruptedException e){}
    }
}

```

Writer client code

```

class Writer implements Runnable {

    ReadWrite monitor_;

    Writer(ReadWrite monitor) {
        monitor_ = monitor;
    }

    public void run() {
        try {
            while(true) {
                while(!ThreadPanel.rotate());
                // begin critical section
                monitor_.acquireWrite();
                while(ThreadPanel.rotate());
                monitor_.releaseWrite();
            }
        } catch (InterruptedException e){}
    }
}

```

R/W monitor (regulate readers)

```

class ReadWriteSafe implements ReadWrite {
    private int readers = 0;
    private boolean writing = false;

    public synchronized void acquireRead()
        throws InterruptedException {
        while (writing) wait();
        ++readers;
    }

    public synchronized void releaseRead() {
        --readers;
        if(readers==0) notifyAll();
    }

    public synchronized void acquireWrite() {...}

    public synchronized void releaseWrite() {...}
}

```

R/W monitor (regulate writers)

```

class ReadWriteSafe implements ReadWrite {
    private int readers = 0;
    private boolean writing = false;

    public synchronized void acquireRead() {...}

    public synchronized void releaseRead() {...}

    public synchronized void acquireWrite()
        throws InterruptedException {
        while (readers>0 || writing) wait();
        writing = true;
    }

    public synchronized void releaseWrite() {
        writing = false;
        notifyAll();
    }
}

```

DEMO

Fairness



http://www.doc.ic.ac.uk/~jnm/book/book_applets/ReadWriteFair.html

“Fairness”: regulating readers

```
class ReadWriteFair implements ReadWrite {  
    private int readers =0;  
    private boolean writing = false;  
    private int waitingW = 0; // no of waiting Writers.  
    private boolean readersturn = false;  
  
    synchronized public void acquireRead()  
    throws InterruptedException {  
        while (writing || (waitingW>0 && !readersturn)) wait();  
        ++readers;  
    }  
  
    synchronized public void releaseRead() {  
        --readers;  
        readersturn=false;  
        if(readers==0) notifyAll();  
    }  
  
    synchronized public void acquireWrite() {...}  
    synchronized public void releaseWrite() {...}  
}
```

“Fairness”: regulating writers

```
class ReadWriteFair implements ReadWrite {  
    private int readers =0;  
    private boolean writing = false;  
    private int waitingW = 0; // no of waiting Writers.  
    private boolean readersturn = false;  
  
    synchronized public void acquireRead() {...}  
    synchronized public void releaseRead() {...}  
  
    synchronized public void acquireWrite()  
    throws InterruptedException {  
        ++waitingW;  
        while (readers>0 || writing) wait();  
        --waitingW; writing = true;  
    }  
  
    synchronized public void releaseWrite() {  
        writing = false; readersturn=true;  
        notifyAll();  
    }  
}
```

Readers and Writers problem

DEMO

http://www.doc.ic.ac.uk/~jnm/book/book_applets/ReadersWriters.html

Java concurrency

- there's (much) more to it than what we discussed (synchronization, monitors) (see `java.util.concurrent`)
- Java's memory model: since Java 1: looooong, hot debate
- connections to
 - GUI-programming (swing/awt/events) and to
 - RMI etc.
- major *clean-up*/repair since Java 5
- better “thread management”
- Lock class (allowing `new Lock()` and non block-structured locking)
- one simplification here: Java has a (complex!) *weak* memory model (out-of-order execution, compiler optimization)
- not discussed here *volatile*

General advice

shared, *mutable* state is more than a bit tricky,⁶⁰ watch out!

- work thread-local if possible
- make variables *immutable* if possible
- keep things local: encapsulate state
- learn from tried-and-tested concurrent design patterns

golden rule

never, ever allow (real, unprotected) races

- unfortunately: no silver bullet
- for instance: “synchronize everything as much as possible”: not just inefficient, but mostly nonsense

⇒ concurrent programmig remains a bit of an art

see for instance [Goetz et al., 2006] or [Lea, 1999]

9 Message passing and channels

17. Oct. 2016

9.1 Intro

Outline

Course overview:

- **Part I:** *concurrent* programming; programming with shared variables
- **Part II:** “*distributed*” programming

Outline: asynchronous and synchronous message passing

- *Concurrent* vs. *distributed* programming⁶¹
- *Asynchronous message passing*: channels, messages, primitives

⁶⁰and pointer aliasing and a weak memory model makes it worse.

⁶¹The dividing line is not absolute. One can make perfectly good use of channels and message passing also in a non-distributed setting.

- Example: filters and sorting networks
- From [monitors](#) to [client–server](#) applications
- [Comparison](#) of [message passing](#) and [monitors](#)
- About [synchronous message passing](#)

Shared memory vs. distributed memory

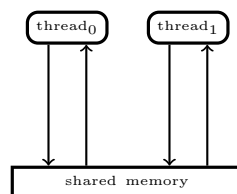
more [traditional](#) system architectures have [one](#) shared memory:

- many processors access the same physical memory
- example: fileserver with many processors on one motherboard

Distributed memory architectures:

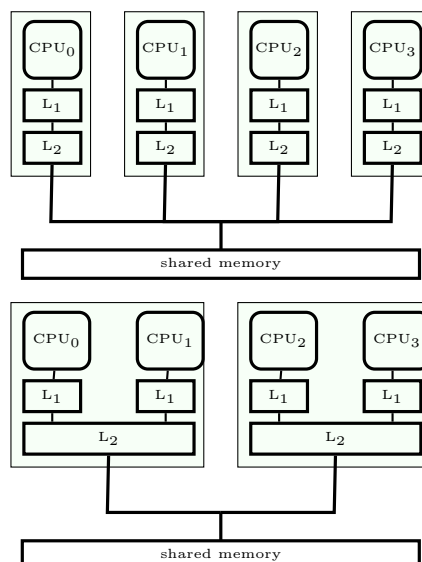
- Processor has private memory and communicates over a “network” (inter-connect)
- Examples:
 - Multicomputer: asynchronous multi-processor with distributed memory (typically contained inside one case)
 - Workstation clusters: PC’s in a local network
 - Grid system: machines on the Internet, resource sharing
 - cloud computing: cloud storage service
 - NUMA-architectures
 - cluster computing ...

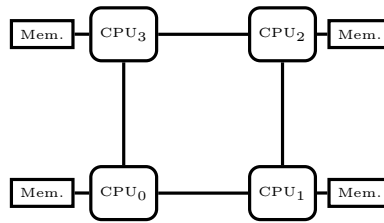
Shared memory concurrency in the real world



- the memory architecture does not reflect reality
- out-of-order executions:
 - modern systems: complex memory hierarchies, caches, buffers...
 - compiler optimizations,

SMP, multi-core architecture, and NUMA





Concurrent vs. distributed programming

Concurrent programming:

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

Distributed programming:

- Memory is distributed \Rightarrow processes cannot share variables (directly)
- Processes communicate by sending and receiving *messages* via shared *channels*
or (in future lectures): communication via *RPC* and *rendezvous*

9.2 Asynch. message passing

Asynchronous message passing: channel abstraction

Channel: abstraction, e.g., of a physical communication network⁶²

- **One-way** from sender(s) to receiver(s)
- unbounded FIFO (queue) of waiting messages
- preserves message order
- atomic access
- error-free
- typed

Variants: errors possible, untyped, ...

Asynchronous message passing: primitives

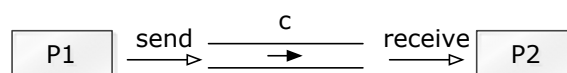
Channel declaration

$\text{chan } c(\text{type}_1 \text{id}_1, \dots, \text{type}_n \text{id}_n);$

Messages: n -tuples of values of the respective types

communication **primitives**:

- **send** $c(\text{expr}_1, \dots, \text{expr}_n);$ Non-blocking, i.e. asynchronous
- **receive** $c(\text{var}_1, \dots, \text{var}_n);$ Blocking: receiver waits until message is sent on the channel
- **empty** $(c);$ True if channel is empty



⁶²but remember also: *producer-consumer* problem

Simple channel example in Go

```
func main() {
    messages := make(chan string, 0) // declare + initialize

    go func() { messages <- "ping" }() // send
    msg := <-messages                  // receive
    fmt.Println(msg)
}
```

Short intro to the Go programming language

- programming language, executable, used by f.ex. Google
- supporting channels and asynchronous processes (function calls)
 - *go-routine*: a lightweight thread
- syntax: mix of functional language (lambda calculus) and imperative style programming (built on C).

Some syntax details of the Go programming language

Calls

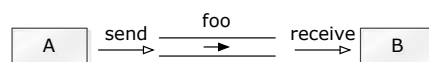
- $f(x)$ – ordinary (synchronous) function call, where f is a defined function or a functional definition
- **go** $f(x)$ – called as an asynchronous process, i.e. *go-routine* **Note**: the *go-routine* will die when its parent process dies!
- **defer** $f(x)$ – the call is delayed until the end of the process

Channels

- $chan := make(chanint, buffersize)$ – declare channel
- $chan <- x$ – send x
- $<- chan$ – receive
- example: $y := <- chan$ – receive in y

Run command: `go run program.go` – compile and run program

Example: message passing



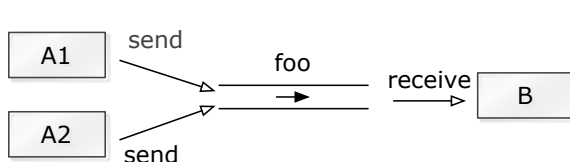
$(x,y) = (1,2)$

```
chan foo(int);

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

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

Example: shared channel



$(x,y) = (1,2) \text{ or } (2,1)$

```

process A1 {
    send foo(1);
}

process A2 {
    send foo(2);
}

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

```

```

func main() {
    foo := make(chan int, 10)
    go func() {
        time.Sleep(1000)
        foo <- 1 // send
    }()

    go func() {
        time.Sleep(1)
        foo <- 2
    }()
    fmt.Println("first _=_", <-foo)
    fmt.Println("second _=_", <-foo)
}

```

Asynchronous message passing and semaphores

Comparison with general [semaphores](#):

channel	\simeq	semaphore
send	\simeq	V
receive	\simeq	P

Number of messages in queue = value of semaphore

(Ignores content of messages)

Semaphores as channels in Go

```

type dummy interface {} // dummy type,
type Semaphore chan dummy // type definition

func (s Semaphore) Vn (n int) {
    for i:=0; i<n; i++ {
        s <- true // send something
    }
}

func (s Semaphore) Pn (n int) {
    for i:=0; i<n; i++ {
        <- s // receive
    }
}

func (s Semaphore) V () {
    s.Vn(1)
}

func (s Semaphore) P () {
    s.Pn(1)
}

```

Listing 2: 5 Phils

```

package main
import (
    "fmt"
    "time"
    "sync"
    "math/rand"
    "andrewsbook/semchans") // semaphores using channels

```

```

var wg sync.WaitGroup

const m = 5 // let's make just 5
var forks = [m]semchans.Semaphore {
    make (semchans.Semaphore,1),
    make (semchans.Semaphore,1),
    make (semchans.Semaphore,1),
    make (semchans.Semaphore,1),
    make (semchans.Semaphore,1)}

func main () {
    for i:=0; i<m; i++ { // initialize the sem's
        forks[i].V()
    }
    wg.Add(m)
    for i:=0; i<m; i++ {
        go philosopher(i)
    }
    wg.Wait()
}

func philosopher(i int) {
    defer wg.Done()
    r := rand.New(rand.NewSource(99)) // random generator
    fmt.Printf("start_P(%d)\n", i)
    for true {
        fmt.Printf("P(%d)_is_thinking\n", i)
        forks[i].P()
        // time.Sleep(time.Duration(r.Int31n(0))) // small delay for DL
        forks[(i+1)%m].P()
        fmt.Printf("P(%d)_starts_eating\n", i)
        time.Sleep(time.Duration(r.Int31n(5))) // small delay
        fmt.Printf("P(%d)_finishes_eating\n", i)
        forks[i].V()
        forks[(i+1)%m].V()
    }
}

```

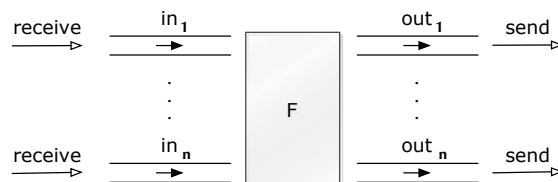
9.2.1 Filters

Filters: one-way interaction

Filter F

= process which:

- receives messages on input channels,
- sends messages on output channels, and
- output is a function of the input (and the initial state).



- A filter is specified as a predicate.
- Some computations: naturally seen as a composition of filters.
- cf. *stream* processing/programming (feedback loops) and *dataflow programming*

Example: A single filter process

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

process **Sort** with input channels **input** and output channel **output**.

Define:

n : number of values sent to **output**. $sent[i]$: i 'th value sent to **output**.

Sort predicate

$\forall i : 1 \leq i < n. (sent[i] \leq sent[i+1]) \wedge$ values sent to **output** are a *permutation* of values from **input**.

Filter for merging of streams

Problem: **Merge** two sorted input streams into one sorted stream.

Process **Merge** with input channels **in₁** and **in₂** and output channel **out**:

$in_1: 1 \ 4 \ 9 \ \dots$	$out: 1 \ 2 \ 4 \ 5 \ 8 \ 9 \ \dots$
$in_2: 2 \ 5 \ 8 \ \dots$	

Special value **EOS** marks the end of a stream.

Define: n : number of values sent to **out**. $sent[i]$: i 'th value sent to **out**.

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

in_1 and in_2 are empty $\wedge sent[n+1] = \text{EOS} \wedge \forall i : 1 \leq i < n (sent[i] \leq sent[i+1]) \wedge$ values sent to **out** are a *permutation* of values from in_1 and in_2

Example: Merge process

```

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  $\neq$  EOS and v2  $\neq$  EOS) {
    if (v1  $\leq$  v2)
      { send out(v1); receive in1(v1); }
    else
      { send out(v2); receive in2(v2); }
  }

  # consume the rest
  # of the non-empty input channel

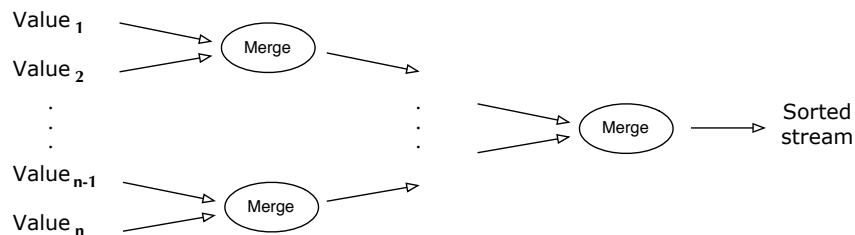
  while (v2  $\neq$  EOS)
    { send out(v2); receive in2(v2); }
  while (v1  $\neq$  EOS)
    { send out(v1); receive in1(v1); }
  send out(EOS); # add special value to out
}

```

Sorting network

We now build a **network** that sorts n numbers.

We use a **collection** of **Merge** processes with tables of shared input and output channels.



(Assume: number of input values n is a power of 2)

9.2.2 Client-servers

Client-server applications using messages

Server: process, repeatedly handling requests from client processes.

Goal: Programming client and server systems with asynchronous message passing.

```
chan request(int clientID, ...),
  reply[n](...);

client nr. i                                server
int id; # client id.

while(true) { # server loop
  receive request(id, vars);
  :
  :
  send reply[id](results);
}
```

9.2.3 Monitors

Monitor implemented using message passing

Classical **monitor**:

- controlled *access* to shared resource
- Permanent variables (monitor variables): safeguard the resource state
- access to a resource via *procedures*
- procedures: executed under *mutual exclusion*
- *condition* variables for synchronization

also implementable by **server process** + **message passing**

Called “**active monitor**” in the book: active process (loop), instead of passive procedures.⁶³

Allocator for multiple-unit resources

Multiple-unit resource: a resource consisting of multiple units

Examples: memory blocks, file blocks.

Users (clients) need resources, use them, and return them to the **allocator** (“free” the resources).

- here simplification: users get and free *one* resource at a time.
- two versions:
 1. monitor
 2. server and client processes, message passing

Allocator as monitor

Uses “**passing the condition**” pattern \Rightarrow simplifies later **translation** to a server process

Unallocated (free) **units** are **represented** as a **set**, type **set**, with operations **insert** and **remove**.

⁶³In practice: server may spawn local threads, one per request.

Recap: “semaphore monitor” with “passing the condition”

```
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);
  }
}
```

(Fig. 5.3 in Andrews [Andrews, 2000])

Allocator as a monitor

```
monitor Resource Allocator {
  int avail := MAXUNITS;
  set units := ... # initial values;
  cond free;      # signalled when process wants a unit

  procedure acquire(int &id) { # var.parameter
    if (avail = 0)
      wait(free);
    else
      avail := avail - 1;
      remove(units, id);
  }

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

([Andrews, 2000, Fig. 7.6])

Allocator as a server process: code design

1. interface and “data structure”
 - (a) allocator with two types of operations: **get** unit, **free** unit
 - (b) 1 request channel⁶⁴ \Rightarrow must be *encoded* in the arguments to a request.
2. control structure: **nested if**-statement (2 levels):
 - (a) first checks **type** operation,
 - (b) proceeds correspondingly to **monitor-if**.
3. synchronization, scheduling, and mutex
 - (a) cannot wait (**wait(free)**) when no unit is free.
 - (b) must save the request and return to it later
 \Rightarrow queue of pending requests (**queue**; **insert**, **remove**).
 - (c) request: “synchronous/blocking” call \Rightarrow “ack”-message back
 - (d) no internal parallelism \Rightarrow mutex

1>In order to design a monitor, we may follow the following 3 “design steps” to make it more systematic:
1) Interface, 2) “business logic” 3) sync./coordination

⁶⁴Alternatives exist

Channel declarations:

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

Allocator: client processes

```
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
  ...
}
```

(Fig. 7.7(b) in Andrews)

Allocator: server process

```
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)          # save request
        insert(pending, clientID);
      else { # perform request now
        avail := avail - 1;
        remove(units, unitID);
        send reply[clientID](unitID);
      }
    }
    else { # kind = RELEASE
      if empty(pending) { # return units
        avail := avail + 1; insert(units, unitID);
      } else { # allocates to waiting client
        remove(pending, clientID);
        send reply[clientID](unitID);
      }
    }
  }
}
```

Duality: monitors, 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 if statement wrt. op. type

9.3 Synchronous message passing

Synchronous message passing

Primitives:

- New primitive for sending:
synch_send c(expr₁, ..., expr_n);

Blocking send:

- sender waits until message is received by channel,
- i.e. sender and receiver “synchronize” sending and receiving of message

- Otherwise: like asynchronous message passing:

```
receive c(var1, ..., varn);
empty(c);
```

Synchronous message passing: discussion

Advantages:

- Gives maximum [size](#) of channel.

Sender synchronises with receiver \Rightarrow receiver has at most 1 pending message per channel per sender \Rightarrow 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

```
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

```
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** block on `synch_send` – [deadlock](#).

One process must be modified to do **receive** first \Rightarrow asymmetric solution.

With [asynchronous](#) message passing (`send`) all goes well.

```
func main() {
  var wg sync.WaitGroup // wait group
  c1, c2 := make(chan int, 0), make(chan int, 0)
  wg.Add(2) // prepare barrier
  go func() {
    defer wg.Done() // signal to barrier
    c1 <- 1 // send
    x := <- c2 // receive
    fmt.Printf("P1: x := %v\n", x)
  }()

  go func() {
    defer wg.Done()
    c2 <- 2
    x := <- c1
    fmt.Printf("P2: x := %v\n", x)
  }()

  wg.Wait() // barrier
}
```

INF4140 7 Nov. 2016

10 RPC and Rendezvous

Outline

- More on asynchronous message passing

- interacting processes with different **patterns of communication**
- summary
- remote procedure calls
 - concept, syntax, and meaning
 - examples: time server, merge filters, exchanging values
- rendez-vous
 - concept, syntax, and meaning
 - examples: buffer, time server, exchanging values
- combinations of RPC, rendezvous and message passing
 - Examples: bounded buffer, readers/writers

Remark 4 (Join). *RPC and rendezvous is good for client-server patterns, Andrews claims. What is new here is that it's two-way communication (but in the previous chapter, when we “simulated” a monitor via channel communication, we already used basically to channels, for the synchronous entering to the monitor) Both have a “call-notation”. The difference between rendez-vous and rpc is on the server side:*

1. for RPC, on the server, each call is served by a “new process” (at least conceptually).
2. For the rendezvous, there is already an existing process. Then caller and callee perform a rendez-vous.

Therefore there will be an **extra syntax** for the receiving side of the rendez-vous.

10.1 Message passing (cont'd)

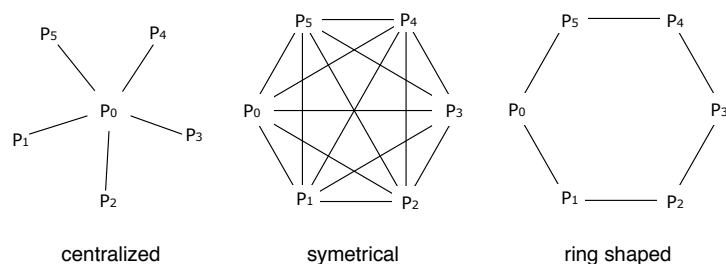
Interacting peers (processes): exchanging values example

Look at processes as **peers**.

Example: Exchanging values

- Consider n processes $P[0], \dots, P[n-1]$, $n > 1$
- every process has a **number**, stored in local variable v
- **Goal:** all processes knows the **largest** and **smallest** number.
- simplistic problem, but “characteristic” of distributed computation and **information distribution**

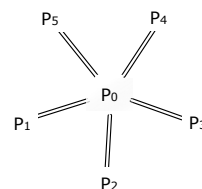
Different communication patterns



Centralized solution

Process $P[0]$ is the **coordinator process**:

- $P[0]$ does the calculation
- **The other processes** send their values to $P[0]$ and wait for a reply.



Number of *messages*:⁶⁵ (number of sends:)

P[0]: $n - 1$

P[1], ..., P[n - 1]: $(n - 1)$

Total: $(n - 1) + (n - 1) = 2(n - 1) \sim 2n$ messages
repeated “computation”

Number of *channels*: $\sim n$

Remark 5 (Join). *P[0] receives $n - 1$ messages and, sequentially afterwards, sends $n - 1$ messages.*

Now: one single line means: 1 message, also for the following slides. That has actually changed from the earlier picture.

Centralized solution: code

```

chan values(int),
      results[1..n-1](int smallest, int largest);

process P[0] { # coordinator process
  int v := ...;
  int new, smallest := v, largest := v; # initialization
  # get values and store the largest and smallest
  for [i = 1 to n-1] {
    receive values(new);
    if (new < smallest)      smallest := new;
    if (new > largest)      largest := new;
  }
  # send results
  for [i = 1 to n-1]
    send results[i](smallest, largest);
}
process P[i = 1 to n-1] {
  int v := ...;
  int smallest, largest;

  send values(v);
  receive results[i](smallest, largest);}
# Fig. 7.11 in Andrews (corrected a bug)

```

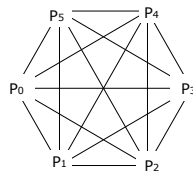
```

for i:=0; i<m; i++      {
    go P (i, values, results[i], r)
}

for i:=0; i<m; i++      {
    v = <- values
    if v > largest { largest = v}
}
fmt.Printf("largest %v\n", largest)
for i := range results {
    results[i] <- largest
}
}

```

Symmetric solution



“Single-programme, multiple data (SPMD)”-solution:

Each process executes the **same** code and shares the results with all other processes.

Number of messages: n processes sending $n - 1$ messages each, **Total:** $n(n - 1)$ messages.

Number of (bi-directional) channels: $n(n - 1)$

⁶⁵For now in the pics: 1 line = 1 message (not 1 channel), but the notation in the pics is not 100% consistent.

Symmetric solution: code

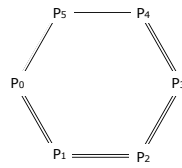
```
chan values[n](int);

process P[i = 0 to n-1] {
  int v := ...;
  int new, smallest := v, largest := v;

  # send v to all n-1 other processes
  for [j = 0 to n-1 st j ≠ i]
    send values[j](v);

  # get n-1 values
  # and store the smallest and largest.
  for [j = 1 to n-1] { # j not used in the loop
    receive values[i](new);
    if (new < smallest)    smallest := new;
    if (new > largest)    largest := new;
  }
} # Fig. 7.12 from Andrews
```

Ring solution



Almost symmetrical, except $P[0]$, $P[n-2]$ and $P[n-1]$.

Each process executes the same code and sends the results to the *next* process (if necessary).

Number of messages:
$$\begin{array}{lcl} P[0]: & 2 & \\ P[1], \dots, P[n-3]: & (n-3) \times 2 & \\ P[n-2]: & 1 & \\ P[n-1]: & 1 & \end{array} \quad 2 + 2(n-3) + 1 + 1 = 2(n-1) \text{ messages sent.}$$

Number of channels: n .

Remark 6 (Join). *That every process does the same code, that's of course not really true: P_0 is different, and then there's a conditional in the others, which checks on the process id.*

Ring solution: code (1)

```
chan values[n](int smallest, int largest);

process P[0] { # starts the exchange
  int v := ...;
  int smallest := v, largest := v;
  # send v to the next process, P[1]
  send values[1](smallest, largest);
  # get the global smallest and largest from P[n-1]
  # and send them to P[1]
  receive values[0](smallest, largest);
  send values[1](smallest, largest);
}
```

Ring solution: code (2)

```
process P[i = 1 to n-1] {
  int v := ...;
  int smallest, largest;
  # get smallest and largest so far,
  # and update them by comparing them to v
  receive values[i](smallest, largest);
  if (v < smallest) smallest := v;
  if (v > largest)  largest := v;
  # forward the result, and wait for the global result
  send values[(i+1) mod n](smallest, largest);
  if (i < n-1)
    receive values[i](smallest, largest);
  # forward the global result, but not from P[n-1] to P[0]
  if (i < n-2)
    send values[i+1](smallest, largest);
} # Fig. 7.13 from Andrews (modified)
```

Message passing: Summary

Message passing: well suited to programming **filters** and **interacting peers** (where processes communicates **one way** by one or more **channels**).

May be used for client/server applications, but:

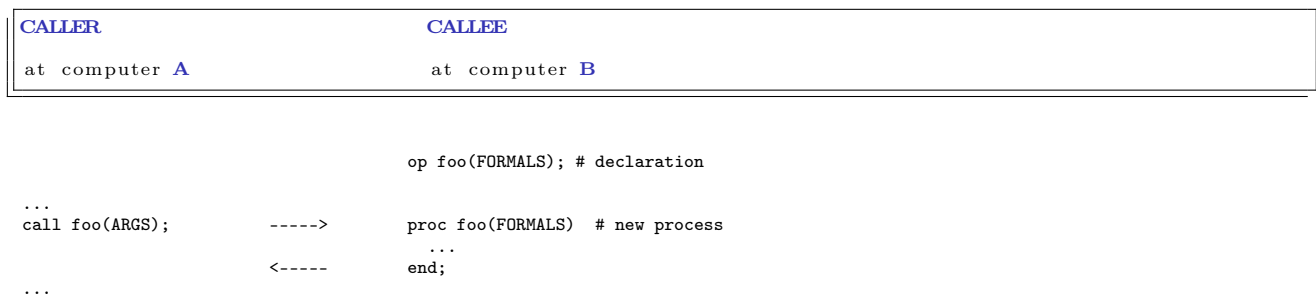
- Each client must have its own reply channel
- In general: **two way** communication needs two channels

⇒ **many channels**

RPC and **rendezvous** are better suited for **client/server** applications.

10.2 RPC

Remote Procedure Call: main idea



RPC (cont.)

RPC: combines elements from **monitors** and **message passing**

- As ordinary **procedure call**, but caller and callee may be on **different machines**.⁶⁶
- Caller: **blocked** until called procedure is done, as with monitor calls and synchronous message passing.
- **Asynchronous** programming: not supported directly
- A **new process** handles each call.
- Potentially **two way** communication: caller **sends arguments** and **receives return values**.

RPC: module, procedure, process

Module: new program component – contains both

- procedures and processes.

```
module M  
  headers of exported operations;  
body  
  variable declarations;  
  initialization code;  
  procedures for exported operations;  
  local procedures and processes;  
end M
```

Modules may be executed on **different machines**

M has: *procedures* and *processes*

- may **share variables**
- execute concurrently ⇒ *must be **synchronized** to achieve **mutex***
- May only **communicate** with processes in M' by procedures *exported* by M'

⁶⁶cf. RMI

RPC: operations

Declaration of operation O:

op O(*formal parameters.*) [returns *result*] ;

Implementation of operation O:

```
proc O(formal identifiers.) [ returns result identifier]{declaration of local variables;  
statements }
```

Call of operation O in module M:⁶⁷

call M.O(*arguments*)

Processes: as before.

Synchronization in modules

- RPC: primarily a *communication* mechanism
- within the module: in principle allowed:
 - more than one process
 - shared data

⇒ need for synchronization

- two approaches
 1. “implicit”:
 - as in *monitors*: mutex built-in
 - additionally condition variables (or semaphores)
 2. “explicit”:⁶⁸
 - user-programmed mutex and synchronization (like semaphore, local monitors etc.)

Example: Time server (RPC)

- module providing *timing services* to processes in other modules.
 - interface: two visible operations:
 - `get_time()` returns `int` – returns time of day
 - `delay(int interval)` – let the caller sleep a given number of time units
 - multiple clients: may call `get_time` and `delay` at the same time
- ⇒ Need to *protect* the variables.
- internal *process* that gets *interrupts* from machine clock and updates `tod`

Time server code (rpc)

```
module TimeServer
  op get_time() returns int;
  op delay(int interval);
body
  int tod := 0;           # time of day
  sem m := 1;             # for mutex
  sem d[n] := ([n] 0);    # for delayed processes
  queue of (int waketime, int process_id) napQ;
  ## when m = 1, tod < waketime for delayed processes
  proc get_time() returns time { time := tod; }

  proc delay(int interval) {
    P(m); # assume unique myid and i [0,n-1]
    int waketime := tod + interval;
    insert (waketime, myid) at appropriate place in napQ;
    V(m);
    P(d[myid]); # Wait to be awoken
  }
  process Clock ...
end TimeServer
```

⁶⁷Cf. static/class methods

⁶⁸assumed in the following

Time server code: clock process

```
process Clock {
  int id; start hardware timer;
  while (true) {
    wait for interrupt, then restart hardware timer
    tod := tod + 1;
    P(m);                                     # mutex
    while (tod ≥ smallest waketime on napQ) {
      remove (waketime, id) from napQ;      # book-keeping
      V(d[id]);                             # awake process
    }
    V(m);                                     # mutex
  }
}
end TimeServer # Fig. 8.1 of Andrews
```

10.3 Rendez-vous

Rendezvous

RPC:

- offers inter-module communication
- synchronization (often): must be programmed explicitly

Rendezvous:

- known from the language [Ada](#) (US DoD)
- combines communication and synchronization between processes
- *No new* process created for each call
- instead: perform ‘rendezvous’ with existing process
- operations are executed one at the time

[synch_send](#) and [receive](#) may be considered as primitive rendezvous.
cf. also [join](#)-synchronization

Rendezvous: main idea

CALLER	CALLEE
at computer A	at computer B

```
...                                     op foo(FORMALS); # declaration
call foo(ARGS);                       -----> ... # existing process
                                     in foo(FORMALS) ->
                                     BODY;
                                     <----- ni
...
```

Rendezvous: module declaration

```
module M
  op O1(types);
  ...
  op On(types);
body
  process P1 {
    variable declarations;
    while (true)                                     # standard pattern
      in O1(formals) and B1 -> S1;
      ...
      [] On(formals) and Bn -> Sn;
      ni
    }
  ... other processes
end M
```

Calls and input statements

Call:

```
call  $O_i$  ( $expr_1, \dots, expr_m$ );
```

Input statement, multiple guarded expressions:

```
in  $O_1(v_1, \dots, v_{m_1})$  and  $B_1 \rightarrow S_1$ ;  
...  
[]  $O_n(v_1, \dots, v_{m_n})$  and  $B_n \rightarrow S_n$ ;  
ni
```

The **guard** consists of:

- and B_i – **synchronization expression** (optional)
- S_i – statements (one or more)

The variables v_1, \dots, v_{m_i} may be referred by B_i and S_i may read/write to them.⁶⁹

Semantics of input statement

Consider the following:

```
in ...  
[]  $O_i(v_i, \dots, v_{m_i})$  and  $B_i \rightarrow S_i$ ;  
...  
ni
```

The guard **succeeds** when O_i is called and B_i is true (or omitted).

Execution of the in statement:

- **Delays** until a guard succeeds
- If **more than one** guard succeed, the **oldest call** is served⁷⁰
- Values are **returned** to the caller
- The the **call**- and **in**-statements terminates

Different variants

- different versions of rendezvous, depending on the language
- origin: ADA (**accept**-statement) (see [Andrews, 2000, Section 8.6])
- design variation points
 - synchronization expressions or not?
 - scheduling expressions or not?
 - can the guard inspect the *values* for input variables or not?
 - non-determinism
 - checking for *absence* of messages? priority
 - checking in more than one operation?

Bounded buffer with rendezvous

```
module BoundedBuffer  
  op deposit (TypeT), fetch (result TypeT);  
body  
  process Buffer {  
    elem buf[n];  
    int front := 0, rear := 0, count := 0;  
    while (true)  
      in deposit(item) and count < n ->  
        buf[rear] := item; count++;  
        rear := (rear+1) mod n;  
      [] fetch(item) and count > 0 ->  
        item := buf[front]; count--;  
        front := (front+1) mod n;  
    ni  
  }  
end BoundedBuffer # Fig. 8.5 of Andrews
```

⁶⁹once again: no side-effects in B_i !!!

⁷⁰this may be changed using additional syntax (**by**), see [Andrews, 2000].

Example: time server (rendezvous)

```
module TimeServer
  op get_time() returns int;
  op delay(int); # absolute waketime as argument
  op tick(); # called by the clock interrupt handler
body
  process Timer {
    int tod := 0;
    start timer;
    while (true)
      in get_time() returns time -> time := tod;
      [] delay(waketime) and waketime <= tod -> skip;
      [] tick() -> { tod++; restart timer; }
    ni
  }
end TimeServer # Fig. 8.7 of Andrews
```

RPC, rendezvous and message passing

We do now have several combinations:

invocation	service	effect
call	proc	procedure call (RPC)
call	in	rendezvous
send	proc	dynamic process creation, asynchronous proc. calling
send	in	asynchronous message passing

in addition (not in Andrews)

- asynchronous procedure call, wait-by-necessity, futures

Rendezvous, message passing and semaphores

Comparing input statements and receive:

$$\text{in } O(a_1, \dots, a_n) \rightarrow v_1=a_1, \dots, v_n=a_n \text{ ni} \iff \text{receive } O(v_1, \dots, v_n)$$

Comparing message passing and semaphores:

$$\text{send } O() \text{ and receive } O() \iff V(O) \text{ and } P(O)$$

Bounded buffer: procedures and “semaphores” (simulated by channels)

```
module BoundedBuffer
  op deposit(typeT), fetch(result typeT);
body
  elem buf[n];
  int front = 0, rear = 0;
  # local operation to simulate semaphores
  op empty(), full(), mutexD(), mutexF(); # operations
  send mutexD(); send mutexF(); # init. "semaphores" to 1
  for [i = 1 to n] # init. empty-"semaphore" to n
    send empty();

  proc deposit(item) {
    receive empty(); receive mutexD();
    buf[rear] = item; rear = (rear+1) mod n;
    send mutexD(); send full();
  }
  proc fetch(item) {
    receive full(); receive mutexF();
    item = buf[front]; front = (front+1) mod n;
    send mutexF(); send empty();
  }
end BoundedBuffer # Fig. 8.12 of Andrews
```

The primitive ?O in rendezvous

New primitive on operations, similar to empty(...) for condition variables and channels.

?O means number of pending invocations of operation O.

Useful in the input statement to give priority:

```

in
   $O_1 \dots \rightarrow S_1;$ 
[ ]
   $O_2 \dots \text{ and } (?O_1 = 0) \rightarrow S_2;$ 
ni

```

Here O_1 has a higher priority than O_2 .

Readers and writers

```

module ReadersWriters
  op read(result types); # uses RPC
  op write(types); # uses rendezvous
body
  op startread(), endread(); # local ops.
  ... database (DB)...;

  proc read(vars) {
    call startread(); # get read access
    ... read vars from DB ...;
    send endread(); # free DB
  }
  process Writer {
    int nr := 0;
    while (true)
      in startread() -> nr++;
      [] endread() -> nr--;
      [] write(vars) and nr = 0 ->
        ... write vars to DB ... ;
    ni
  }
end ReadersWriters

```

Readers and writers: prioritize writers

```

module ReadersWriters
  op read(result typeT); # uses RPC
  op write(typeT); # uses rendezvous
body
  op startread(), endread(); # local ops.
  ... database (DB)...;

  proc read(vars) {
    call startread(); # get read access
    ... read vars from DB ...;
    send endread(); # free DB
  }
  process Writer {
    int nr := 0;
    while (true)
      in startread() and ?write = 0 -> nr++;
      [] endread() -> nr--;
      [] write(vars) and nr = 0 ->
        ... write vars to DB ... ;
    ni
  }
end ReadersWriters

```

11 Asynchronous Communication I

14. 11. 2016

Asynchronous Communication: Semantics, specification and reasoning

Where are we?

- part one: shared variable systems
 - programming
 - synchronization
 - reasoning by invariants and Hoare logic
- part two: communicating systems
 - message passing
 - channels

- rendezvous

What is the connection?

- What is the semantic understanding of message passing?
- How can we understand concurrency?
- How to understand a system by looking at each component?
- How to specify and reason about asynchronous systems?

Overview

Clarifying the semantic questions above, by means of **histories**:

- describing interaction
- capturing **interleaving semantics** for concurrent systems
- **Focus**: asynchronous communication systems without channels

Plan today

- histories from the **outside** (global) view of components
 - describing overall understanding of a (sub)system
- Histories from the **inside** (local) view of a component
 - describing local understanding of a single process
- The connection between the **inside** and **outside** view
 - the **composition rule**

What kind of system? Agent network systems

Two flavors of message-passing concurrent systems, based on the notion of:

- **processes** — without self identity, but with named **channels**. Channels often FIFO.
- **objects (agents)** — with self identity, but without channels, sending messages to named objects through a **network**. In general, a network gives no FIFO guarantee, nor guarantee of successful transmission.

We use the latter here, since it is a very general setting. The process/channel setting may be obtained by representing each combination of object and message kind as a channel.

In the following we consider systems with **agents** connected by a network!

Programming asynchronous agent systems

New syntax statements for sending and receiving:

- *send statement*: **send** $A : m(e)$ means that the current agent sends message m to agent A where e is an (optional) list of actual parameters.
- *fixed receive statement*: **await** $A : m(w)$ wait for a message m from a specific agent A , and receive parameters in the variable list w . We say that the message is then **consumed**.
- *open receive statement*: **await** $X?m(w)$ wait for a message m from any agent X and receive parameters in w (consuming the message). The **variable** X will be set to the agent that sent the message.
- *choice operator* $[]$ to select between alternative statement lists, each starting with a receive statement.

Here m is a message name, A the name of an agent, e expressions, X and w variables.

Async. communication constructs

Syntax (s statement list, e expression)

$s ::=$	send $A : m(e)$	send to A
	await $A : m(w)$	receive from A
	await $X ? m(w)$	receive from someone
	await $?m(w)$	anonymous receive
	$s [] s$	choice
	$x := e \mid s; s \mid (s)$	assignment and seq. composition
	$if... \mid while... \mid ...$	other statements

Channel comm. in Go

- no “named” sender or receiver: go-routines are anonymous
- instead Go uses channel names
- choice operator: **select**
- different syntax (of course):
 - $<- c$: receive over c
 - $c <- e$: send e over c
- similar non-determinism by select-case:

```
select {
  case msg := <-c1: // comparable to our choice [ ]
                   // receive on c1 and store in msg
    ...
  case msg := <-c2:
    ...
  case msg := <-c3:
    ...
  default:         // optional branch if
    ...            // nothing else works
}
```

Example: Coin machine (from Exam 05)

Consider an agent C which changes “5 krone” coins and “1 krone” coins into “10 krone” coins. It receives *five* and *one* messages and sends out *ten* messages as soon as possible, in the sense that the number of messages sent out should equal the total amount of kroner received divided by 10.

We imagine here a fixed user agent U , both producing the *five* and *one* messages and consuming the *ten* messages. The code of the agent C is given below, using b (*balance*) as a local variable initialized to 0.

Example: Coin machine (Cont)

```
loop
  while b < 10
  do
    (await U: five; b:=b+5)
  []
    (await U: one; b:=b+1)
  od;
  send U: ten;
  b:=b-10 // use b in next iteration
end
```

- the **choice** operator $[]$ ⁷¹
 - selects an *enabled* branch, if any (and otherwise waits)
 - **non-deterministic** choice if both branches are enabled

⁷¹In the literature, also $+$ as notation can often be found. $[]$ taken because of “ASCII” version of \square , which can be found in publications.

Interleaving semantics of concurrent systems

- behavior of a concurrent system: may be described as **set of executions**,
- each execution: **sequence of atomic communication events**,
- other names for it: **trace**, history, execution, (interaction) sequence ...⁷²

Interleaving semantics

Concurrency is expressed by the set of all *possible interleavings*.

- remember also: “sequential consistency” from the WMM part.
- note: for each interaction sequence, all interactions are ordered sequentially, and their “visible” concurrency

Regular expressions (to express traces)

- very well known and widely used “format” to describe “languages” (= sets of finite “words” over given a given “alphabet”)
- formed by
 - ***** (repetition)
 - **+** (choice)
 - **;** (sequential composition)

(**Note**: we do not use **|** for choice, to not confuse with parallel composition.)

A way to describe (sets of) traces

Example 23 (Reg-Expr). • a, b : atomic interactions.

- Assume them to “run” concurrently

⇒ two possible interleavings, described by

$$[[a; b] + [b; a]] \quad (6)$$

Parallel composition of a^* and b^* :

$$(a + b)^* \quad (7)$$

Safety and liveness & traces

We may let each interaction sequence reflect all interactions in an execution, called the **trace**, and the set of all possible traces is then called the **trace set**.

- terminating system: **finite** traces⁷³
- *non-terminating* systems: infinite traces
- trace set semantics in the general case: both finite and infinite traces
- 2 conceptually important classes of properties⁷⁴
 - **safety** (“nothing wrong will happen”)
 - **liveness** (“something good will happen”)

⁷²message sequence (charts) in UML etc.

⁷³Be aware: typically an *infinite* set of finite traces.

⁷⁴Safety etc. it’s not a property, it’s a “property/class of properties”

Safety and liveness & histories

- often: concentrate on *finite traces*
- reasons
 - conceptually/theoretically simpler
 - connection to run-time monitoring/run-time verification
 - connection to checking (violations of) **safety** prop's
- our terminology: **history** = trace up to a given execution point (thus finite)
- **Note:** In contrast to the book, histories are here finite initial parts of a trace (prefixes)
- **sets of histories** are **prefix closed**:
 - if a history h is in the set, then every prefix (initial part) of h is also in the set.
- sets of histories: can be used capture safety, but **not liveness**

Simple example: histories and trace set

Consider a system of two agents, A and B , where agent A says “hi-B” repeatedly until B replies “hi-A”.

- “sloppy” B : may or may not reply, in which case there will be an infinite trace with only “hi-B” (here comma denotes \cup).

Trace set: $\{[hi_B]^\infty\}, \{[hi_B]^+ [hi_A]\}$ **Histories:** $\{[hi_B]^*, \{[hi_B]^+ [hi_A]\}$

- “lazy” B : will reply eventually, but no limit on how long A may need to wait. Thus, each trace will end with “hi-A” after finitely many “hi-B”'s. **Trace set:** $\{[hi_B]^+ [hi_A]\}$ **Histories:** $\{[hi_B]^*, \{[hi_B]^+ [hi_A]\}$

- an “eager” B will reply within a fixed number of “hi-B”'s, for instance before A says “hi-B” three times.

Trace set: $\{[hi_B] [hi_A], \{[hi_B] [hi_B] [hi_A]\}$ **Histories:** $\emptyset, \{\epsilon\}\{[hi_B]\}, \{[hi_B] [hi_A]\}, \{[hi_B] [hi_B]\}, \{[hi_B] [hi_B] [hi_A]\}$

Histories = sequences of communication events

Let use the following conventions

- communication events $a : Event$ is an event
- set of communication events: $A : 2^{Event}$
- history $h : Hist$

For a given program, the set of (communication) events $Event$ is assumed to be fixed.

Definition 24 (Histories). Histories (over the given set of events) is given inductively over the constructors ϵ (empty history) and $_;$ (appending of an event to the right of the history)

Functions over histories (for specification purpose)

function	type		
ϵ	$:$	$\rightarrow Hist$	the empty history (constructor)
$_;$	$: Hist * Event$	$\rightarrow Hist$	append right (constructor)
$\#$	$: Hist$	$\rightarrow Nat$	length
$_/_$	$: Hist * Set$	$\rightarrow Hist$	projection by set of events
$_ \preceq _$	$: Hist * Hist$	$\rightarrow Bool$	prefix relation
$_ \prec _$	$: Hist * Hist$	$\rightarrow Bool$	strict prefix relation

Inductive definitions (inductive wrt. ϵ and $_;$):

$\#\epsilon$	$= 0$
$\#(h; x)$	$= \#h + 1$
ϵ/A	$= \epsilon$
$(h; x)/A$	$= \text{if } x \in A \text{ then } (h/A); x \text{ else } (h/A) \text{ fi}$
$h \preceq h'$	$= (h = h') \vee h \prec h'$
$h \prec \epsilon$	$= false$
$h \prec (h'; x)$	$= h \preceq h'$

Invariants and Prefix Closed Trace Sets

May use *invariants* to define trace sets:

A (history) invariant I is a predicate over a histories, supposed to hold at all times:

“At any point in an execution h the property $I(h)$ is satisfied”

It defines the following set:

$$\{h \mid I(h)\} \quad (8)$$

- mostly interested in *prefix-closed invariants*!
- a history invariant is **historically monotonic**:

$$h \leq h' \Rightarrow (I(h') \Rightarrow I(h)) \quad (9)$$

- I history-monotonic \Rightarrow set from equation (8) **prefix closed**

Remark: A non-monotonic predicate I may be transformed to a monotonic one I' :

$$\begin{aligned} I'(\varepsilon) &= I(\varepsilon) \\ I'(h'; x) &= I(h') \wedge I(h'; x) \end{aligned}$$

Semantics: Outside view: global histories over events

Consider asynchronous communication by messages from one agent to another: Since message passing may take some time, the sending and receiving of a message m are semantically seen as two distinct atomic interaction events of type **Event**:

- $A \uparrow B:m$ denotes that A sends message m to B
- $A \downarrow B:m$ denotes that B receives (consumes) message m from A

A **global history**, H , is a finite sequence of such events, requiring that it is **legal**, i.e.

each reception is preceded by a corresponding send-event.

For instance, the history

$$[(A \uparrow B:hi), (A \uparrow B:hi), (A \downarrow B:hi), (A \uparrow B:hi), (B \uparrow A:hi)]$$

is legal and expresses that A has sent “hi” three times and that B has received one of these and has replied “hi”.

Note: a concrete message may also have parameters, say **messagename(parameterlist)** where the number and types of the parameters are statically checked.

Coin Machine Example: Events

$$\begin{aligned} U \uparrow C:five & \text{ --- } U \text{ sends the message “five” to } C \\ U \downarrow C:five & \text{ --- } C \text{ consumes the message “five”} \\ \\ U \uparrow C:one & \text{ --- } U \text{ sends the message “one to } C \\ U \downarrow C:one & \text{ --- } C \text{ consumes the message “one”} \\ \\ C \uparrow U:ten & \text{ --- } C \text{ sends the message “ten”} \\ C \downarrow U:ten & \text{ --- } U \text{ consumes the message “ten”} \end{aligned}$$

Legal histories

- **not** all global sequences/histories “make sense”
- depends on the programming language/communication model
- sometimes called well-definedness, well-formedness or similar
- $legal : Hist \rightarrow Bool$

Definition 25 (Legal history).

$$\begin{aligned} legal(\epsilon) &= true \\ legal(h; (A \uparrow B:m)) &= legal(h) \\ legal(h; (A \downarrow B:m)) &= legal(h) \wedge \#(h/\{A \downarrow B:m\}) < \#(h/\{A \uparrow B:m\}) \end{aligned}$$

where m is message and h a history.

- when m include **parameters**, legality ensures that the values received are the same as those sent.

Example of legal history (coin machine C and user U):

$$[(U \uparrow C: \text{five}), (U \uparrow C: \text{five}), (U \downarrow C: \text{five}), (U \downarrow C: \text{five}), (C \uparrow U: \text{ten})]$$

Outside view: logging the global history

How to “calculate” the global history at *run-time*:

- introduce a **global** variable H ,
- initialize: to empty sequence
- for each execution of a **send** statement in A , update H by

$$H := H; (A \uparrow B: m)$$

where B is the destination and m is the message

- for each execution of a **receive** statement in B , update H by

$$H := H; (A \downarrow B: m)$$

where m is the message and A the sender. The message must be of the kind requested by B .

Outside View: Global Properties

specify desired system behavior by predicate I on the global history,

Global invariant

“at any point in an execution H , property $I(H)$ is satisfied”

Enforcement

- run-time **logging** the history: **monitor** an executing system. When $I(H)$ is violated we may
 - report it
 - stop the system, or
 - interact with the system (for inst. through fault handling)
- How to **prove** such properties by analysing the program?
- How can we monitor, or prove correctness properties, **component-wise**?

Semantics: Inside view: Local histories

Definition 26 (Local events). Events **visible** to an agent A , (written α_A = the events **local** to A):

- $A \uparrow B: m$: any send-events from A (output from A)
- $B \downarrow A: m$: any reception by A (input to A)

Definition 27 (Local history). Given a global history: The **local history** of A , written h_A , is the subsequence of all events visible to A

- **Conjecture**: Correspondence between global and local view:

$$h_A = H / \alpha_A$$

i.e. at any point in an execution the history observed locally in A is the projection to A -events of the history observed globally.

- Note: Each event is visible to one, and only one, agent! This will allow compositional reasoning.

Coin Machine Example: Local Events

The events *visible to C* are:

$U \downarrow C: \text{five}$	C consumes the message “five”
$U \downarrow C: \text{one}$	C consumes the message “one”
$C \uparrow U: \text{ten}$	C sends the message “ten”

The events visible to U are:

$U \uparrow C: \text{five}$	U sends the message “five” to C
$U \uparrow C: \text{one}$	U sends the message “one” to C
$C \downarrow U: \text{ten}$	U consumes the message “ten”

How to relate local and global views

From global specification to implementation: First, set up the goal of a system: by one or more global histories. Then implement it. For each component: use the global histories to obtain a local specification, guiding the implementation work.

“construction from specifications” **From implementation to global specification:** First, make or reuse components.

Use the local knowledge for the desired components to obtain global knowledge. **Working with invariants:**

The specifications may be given as invariants over the history.

- Global invariant: in terms of all events in the system
- Local invariant (for each agent): in terms of events visible to the agent

Need *composition* rules connecting local and global invariants.

Example revisited: Coin Machine

```

loop
  while b < 10
  do
    (await U: five ; b:=b+5)
  []
    (await U: one ; b:=b+1)
  od;
  send U: ten ;
  b:=b-10          // use b in next iteration
end

```

Coin Machine:

interactions *visible to C* (i.e. those that may show up in the local history):

$U \downarrow C: \text{five}$	--	C consumes the message “five”
$U \downarrow C: \text{one}$	--	C consumes the message “one”
$C \uparrow U: \text{ten}$	--	C sends the message “ten”

Coin machine example: Loop invariants

Loop invariant for the **outer** loop:

$$\text{sum}(h / \downarrow) = \text{sum}(h / \uparrow) + b \wedge 0 \leq b < 5 \quad (10)$$

where *sum* (the sum of values in the messages) is defined as follows:

$$\begin{aligned}
 \text{sum}(\varepsilon) &= 0 \\
 \text{sum}(h; (\dots : \text{five})) &= \text{sum}(h) + 5 \\
 \text{sum}(h; (\dots : \text{one})) &= \text{sum}(h) + 1 \\
 \text{sum}(h; (\dots : \text{ten})) &= \text{sum}(h) + 10
 \end{aligned}$$

Loop invariant for the **inner** loop:

$$\text{sum}(h / \downarrow) = \text{sum}(h / \uparrow) + b \wedge 0 \leq b < 15 \quad (11)$$

Histories: from inside to outside view

From local histories to global history: if we know all the local histories h_{A_i} in a system ($i = 1 \dots n$), we have the global knowledge

$$legal(H) \wedge (\bigwedge_i h_{A_i} = H/\alpha_{A_i})$$

i.e. the global history H must be legal and correspond to all the local histories. This may be used to reason about the global history.

Local invariant A_i : a local specification of A_i is given by a predicate on the local history $I_{A_i}(h_{A_i})$ describing a property which holds before all local interaction points. I may have the form of an implication, expressing the output events from A_i depends on a condition on its input events.

From local invariants to a global invariant: if each agent satisfies $I_{A_i}(h_{A_i})$, the total system will satisfy the global invariant:

$$legal(H) \wedge (\bigwedge_i I_{A_i}(H/\alpha_{A_i}))$$

Coin machine example: from local to global invariant

before each send/receive: (see eq. (11))

$$sum(h/\downarrow) = sum(h/\uparrow) + b \wedge 0 \leq b < 15$$

Local Invariant of C in terms of h alone:

$$I_C(h) = \exists b. (sum(h/\downarrow) = sum(h/\uparrow) + b \wedge 0 \leq b < 15) \quad (12)$$

$$I_C(h) = 0 \leq sum(h/\downarrow) - sum(h/\uparrow) < 15 \quad (13)$$

For a global history H ($h = H/\alpha_C$) we have:

$$I_C(H/\alpha_C) \Leftrightarrow 0 \leq sum(H/\alpha_C/\downarrow) - sum(H/\alpha_C/\uparrow) < 15 \quad (14)$$

Shorthand notation:

$$0 \leq sum(H/\downarrow C) - sum(H/C\uparrow) < 15$$

Coin machine example: from local to global invariant

- Local invariant of a careful user U (with exact change):

$$\begin{aligned} I_U(h) &= 0 \leq sum(h/\uparrow) - sum(h/\downarrow) \leq 10 \\ I_U(H/\alpha_U) &= 0 \leq sum(H/U\uparrow) - sum(H/\downarrow U) \leq 10 \end{aligned}$$

- Global invariant of the system U and C :

$$I(H) = legal(H) \wedge I_C(H/\alpha_C) \wedge I_U(H/\alpha_U) \quad (15)$$

implying:

Overall

$$0 \leq sum(H/U\downarrow C) - sum(H/C\uparrow U) \leq sum(H/U\uparrow C) - sum(H/C\downarrow U) \leq 10$$

since $legal(H)$ gives: $sum(H/U\downarrow C) \leq sum(H/U\uparrow C)$ and $sum(H/C\downarrow U) \leq sum(H/C\uparrow U)$.

So, globally, this system will have balance ≤ 10 .

Coin machine example: Loop invariants (Alternative)

Loop invariant for the outer loop:

$$rec(h) = sent(h) + b \wedge 0 \leq b < 5$$

where rec (the total amount received) and $sent$ (the total amount sent) are defined as follows:

$$\begin{aligned} rec(\epsilon) &= 0 \\ rec(h; (U \downarrow C: five)) &= rec(h) + 5 \\ rec(h; (U \downarrow C: one)) &= rec(h) + 1 \\ rec(h; (C \uparrow U: ten)) &= rec(h) \\ \\ sent(\epsilon) &= 0 \\ sent(h; (U \downarrow C: five)) &= sent(h) \\ sent(h; (U \downarrow C: one)) &= sent(h) \\ sent(h; (C \uparrow U: ten)) &= sent(h) + 10 \end{aligned}$$

Loop invariant for the inner loop:

$$rec(h) = sent(h) + b \wedge 0 \leq b < 15$$

Legality

The above definition of legality reflects networks where you may not assume that messages sent will be delivered, and where the order of messages sent need not be the same as the order received.

Perfect networks may be reflected by a stronger concept of **legality** (see next slide).

Remark 7 (Self-communication may be considered internal:). *In “black-box” specifications, we consider observable events only, abstracting away from internal events. Then, legality of sending may be strengthened:*

$$legal(h; (A \uparrow B: m)) = legal(h) \wedge A \neq B$$

Using Legality to Model Network Properties

If the network delivers messages in a **FIFO** fashion, one could capture this by strengthening the legality-concept suitably, requiring

$$sendevents(h / \downarrow) \preceq h / \uparrow$$

where the projections h / \uparrow and h / \downarrow denote the subsequence of messages sent and received, respectively, and **sendevents** converts receive events to the corresponding send events.

$$\begin{aligned} sendevents(\epsilon) &= \epsilon \\ sendevents(h; (A \uparrow B: m)) &= sendevents(h) \\ sendevents(h; (A \downarrow B: m)) &= sendevents(h); (A \uparrow B: m) \end{aligned}$$

Channel-oriented systems can be mimicked by requiring FIFO ordering of communication for each pair of agents:

$$sendevents(h / A \downarrow B) \preceq h / A \uparrow B$$

where $A \downarrow B$ denotes the set of receive-events with A as source and B as destination, and similarly for $A \uparrow B$.

12 Asynchronous Communication II

21.11.2016

Overview: Last time

- semantics: **histories** and trace sets
- specification: **invariants** over histories
 - **global** invariants

- **local** invariants
- the connection between local and global histories
- example: **Coin machine**
 - the main program
 - formulating local invariants

Overview: Today

- Analysis of **send/await** statements
- Verifying local **history invariants**
- example: **Coin Machine**
 - proving **loop invariants**
 - the **local invariant** and a **global invariant**
- example: **Mini bank**

Agent/network systems (Repetition)

We consider general **agent/network** systems:

- Concurrent agents:
 - with self identity
 - no variables shared between agents
 - communication by message passing
- Network:
 - no channels
 - no FIFO guarantee
 - no guarantee of successful transmission

Local reasoning by Hoare logic (a.k.a program logic)

We adapt Hoare logic to reason about local histories in an agent A :

- Introducing a **local (logical) variable h** , initialized to empty ϵ
 - h represents the *local* history of A
- For **send/await**-statement: define the effect on h .
 - **extending** the h with the corresponding communication event
- **Local reasoning**: we do not know the global invariant
 - For **await**: unknown parameter values
 - For *open receive*: unknown sender

\Rightarrow use **non-deterministic** assignment

$$x := \text{some} \tag{16}$$

Local invariant reasoning by Hoare Logic

- each send statement **send** $B : m(\vec{x})$ in A is treated as:

$$h := (h; A \uparrow B : m(\vec{x})) \quad (17)$$

- each fixed receive statement **await** $B : m(\vec{x})$ in A^{75} is treated as

$$\vec{x} := \text{some}; h := (h; B \downarrow A : m(\vec{x})) \quad (18)$$

the usage of $\vec{x} := \text{some}$ expresses that A may receive any values for the receive parameters

- each open receive statement **await** $X ? m(\vec{x})$ in A is treated as

$$X := \text{some}; \text{await } X : m(\vec{x}) \quad (19)$$

i.e., $X := \text{some}; \vec{x} := \text{some}; h := (h; B \downarrow A : m(\vec{x}))$, where $X := \text{some}$ expresses that A may receive the message from any agent.

Rule for non-deterministic assignments

Non-det assignment

$$\frac{}{\{ \forall x . Q \} x := \text{some} \{ Q \}} \text{ND-ASSIGN}$$

- as said: await/send have been **expressed** by manipulating h , using non-det assignments

\Rightarrow rules for await/send statements

Derived Hoare rules for send and receive

$$\begin{array}{c} \frac{}{\{ Q[h; A \uparrow B : m(\vec{x})/h] \} \text{ send } B : m(\vec{x}) \{ Q \}} \text{SEND} \\ \frac{}{\{ \forall \vec{x} . Q[h; B \downarrow A : m(\vec{x})/h] \} \text{ await } B : m(\vec{x}) \{ Q \}} \text{RECEIVE}_1 \\ \frac{}{\{ \forall \vec{x}, X . Q[h; X \downarrow A : m(\vec{x})/h] \} \text{ await } X ? m(\vec{x}) \{ Q \}} \text{RECEIVE}_2 \end{array}$$

- As before: A is the current agent/object, h the local history
- We assume that neither B nor X occur in \vec{x} , and that \vec{x} is a list of distinct variables (which is a static check)
- **No shared variables.** \Rightarrow no interference, and Hoare reasoning can be done as usual in the sequential setting!
- Simplified version, if no parameters in await:

$$\frac{}{\{ Q[h/h; (B \downarrow A : m)] \} \text{ await } B : m \{ Q \}} \text{RECEIVE}$$

⁷⁵where \vec{x} is a sequence of variables and X an agent variable

Hoare rules for local reasoning

The Hoare rule for non-deterministic choice ($[]$) is

Rule for $[]$

$$\frac{\{P_1\} S_1 \{Q\} \quad \{P_2\} S_2 \{Q\}}{\{P_1 \wedge P_2\} (S_1 [] S_2) \{Q\}} \text{NONDET}$$

Remark: We may reason similarly backwards over conditionals.⁷⁶

$$\frac{\{P_1\} S_1 \{Q\} \quad \{P_2\} S_2 \{Q\}}{\{(b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_2)\} \text{ if } b \text{ then } S_1 \text{ else } S_2 \text{ fi } \{Q\}} \text{IF'}$$

Coin machine: local events

Invariants may refer to the **local history** h , which is the sequence of events visible to C that have occurred so far. The events visible to C are:

$U \downarrow C : \text{five}$	--	C consumes the message “five”
$U \downarrow C : \text{one}$	--	C consumes the message “one”
$C \uparrow U : \text{ten}$	--	C sends the message “ten”

Inner loop

let I_i (“inner invariant”) abbreviate equation (11)

```

{ I_i }
while b < 10 { b < 10 ∧ I_i }
{ (I_i[(b+5)/b])[h; U ↓ C : five/h] ∧ (I_i[(b+1)/b])[h; U ↓ C : one/h] }
do
  ( await U: five; { I_i[b+5/b] }
    b:=b+5 ) { I_i }
[]
( await U: one; b:=b+1 )
{ I_i }
od;

```

Must prove the implication:

$$b < 10 \wedge I_i \Rightarrow (I_i[(b+5)/b])[h; U \downarrow C : \text{five}/h] \wedge (I_i[(b+1)/b])[h; U \downarrow C : \text{one}/h]$$

Note: From precondition I_i for the loop, we have $I_i \wedge b \geq 10$ as the postcondition to the inner loop.

Outer loop

```

{ I_o }
loop
  { I_o }
  { I_i }
  while b < 10 { b < 10 ∧ I_i }
  { (I_i[(b+5)/b])[h; U ↓ C : five/h] ∧ (I_i[(b+1)/b])[h; U ↓ C : one/h] }
  do
    ( await U: five; { I_i[b+5/b] }
      b:=b+5 ) { I_i }
  []
  ( await U: one; b:=b+1 )
  { I_i }
od;
{ I_i ∧ b ≥ 10 }
{ (I_o[b-10/b])[h; C ↑ U : ten/h] }
send U: ten;
{ I_o[b-10/b] }
b:=b-10
{ I_o }
end

```

Backwards verification.

⁷⁶We used actually a *different* formulation for the rule for conditionals. Both formulations are equivalent in the sense that (together with the other rules, in particular CONSEQUENCE, one can prove the same properties.

Outer loop (2)

Verification conditions (as usual):

- $I_o \Rightarrow I_i$, and
- $I_i \wedge b \geq 10 \Rightarrow (I_o[(b-10)/b])[h; C \uparrow U : ten/h]$
- I_o holds initially since $h = \varepsilon \wedge b = 0 \Rightarrow I_o$

Local history invariant

For each agent (A):

- Predicate $I_A(h)$ over the local communication history (h)
- Describes interactions between A and the surrounding agents
- Must be maintained by *all* history extensions in A
- Last week: Local history invariants for the different agents may be composed, giving a global invariant

Verification idea: “induction”:

Init: Ensure that $I_A(h)$ holds **initially** (i.e., with $h = \varepsilon$)

Preservation: Ensure that $I_A(h)$ holds **after** each **send/await**-statement, assuming that $I_A(h)$ holds before each such statement

Local history invariant reasoning

- to prove properties of the code in agent A
- for instance: loop invariants etc
- the conditions may refer to the **local state** \vec{x} (a list of variables) and the local history h , e.g., $Q(\vec{x}, h)$.

The local history invariant $I_A(h)$

- must hold immediately *after* each send/receive

\Rightarrow if reasoning gives the condition $Q(v, h)$ immediately after a send or receive statement, we basically need to ensure:

$$Q(\vec{x}, h) \Rightarrow I_A(h) \quad (20)$$

- we may **assume** that the invariant is satisfied immediately *before* each send/receive point.
- we may also assume that the *last* event of h is the send/receive event.

Proving the local history invariant

We get 3 verification conditions (+ one for the “beginning”)

$$I_A(\epsilon) \quad (21)$$

$$(h = (h'; A \uparrow B : m(e)) \wedge I_A(h') \wedge Q(\vec{x}, h)) \Rightarrow I_A(h) \quad (22)$$

$$(h = (h'; B \downarrow A : m(\vec{y})) \wedge I_A(h') \wedge Q(\vec{x}, h)) \Rightarrow I_A(h) \quad (23)$$

$$(h = (h'; X \downarrow A : m(\vec{y})) \wedge I_A(h') \wedge Q(\vec{x}, h)) \Rightarrow I_A(h) \quad (24)$$

in all three cases: Q is the condition right after the send-, resp. the await-statement, and

- $I_A(_)$: local history invariant of A
- first conjunct $h = \dots$: specifies last communication step
- $I_A(h')$: assumption that invariant holds before the comm.-statement
- 3 communication/sync. statements to consider: **send** $B : m(e)$, **await** $B : m(\vec{y})$, and **await** $X ? m(\vec{y})$

Coin machine example: local history invariant

For the coin machine C , consider the local history invariant $I_C(h)$ from last week (see equation (13)):

$$I_C(h) = 0 \leq \text{sum}(h/\downarrow) - \text{sum}(h/\uparrow) < 15$$

Consider the statement **send** $U : \text{ten}$ in C

- Hoare analysis of the outer loop gave the condition $I_o[(b-10)/b]$ immediately after the statement
- history *ends* with the event $C \uparrow U : \text{ten}$

\Rightarrow Verification condition, corresponding to equation (22):

$$h = h'; (C \uparrow U : \text{ten}) \wedge I_C(h') \wedge I_o[(b-10)/b] \Rightarrow I_C(h) \quad (25)$$

Coin machine example: local history invariant

Expanding I_C and I_o in the VC from equation (25), gives:

$$\frac{\begin{array}{l} h = h'; (C \uparrow U : \text{ten}) \wedge \\ I_C(h') \wedge \\ I_o[(b-10)/b] \\ \Rightarrow I_C(h) \end{array}}{\begin{array}{l} h = h'; (C \uparrow U : \text{ten}) \wedge \\ (0 \leq \text{sum}(h'/\downarrow) - \text{sum}(h'/\uparrow) < 15) \wedge \\ (\text{sum}(h/\downarrow) = \text{sum}(h/\uparrow) + b - 10 \wedge 0 \leq b - 10 < 5) \\ \Rightarrow 0 \leq \text{sum}(h/\downarrow) - \text{sum}(h/\uparrow) < 15 \end{array}} \\ \frac{\begin{array}{l} (b = (\text{sum}(h/\downarrow) - \text{sum}(h/\uparrow)) \wedge \\ 0 \leq b - 10 < 5) \\ \Rightarrow 0 \leq b - 10 < 15 \end{array}}{\Rightarrow 0 \leq b - 10 < 15}$$

Note: Gray parts not needed here.

Coin Machine Example: Summary**Correctness proofs (bottom-up):**

- code
- loop invariants (Hoare analysis)
- local history invariant
- verification of local history invariant based on the Hoare analysis

Note: The $[\]$ -construct was useful (basically necessary) for programming service-oriented systems, and had a simple proof rule.

Example: “Mini bank” (ATM): Informal specification

Client cycle: The client C is making these messages

- put in card, give pin, give amount to withdraw, take cash, take card

Mini Bank cycle: The mini bank M is making these messages

to client: ask for pin, ask for withdrawal, give cash, return card

to central bank: request of withdrawal

Central Bank cycle: The central bank B is making these messages

to mini bank: grant a request for payment, or deny it

There may be many mini banks talking to the same central bank, and there may be many clients using each mini bank (but the mini bank must handle one client at a time).

Mini bank example: Global histories

Consider a client C , mini bank M and central bank B : [Example of successful cycle:](#)

$[C \uparrow M : \text{card_in}(n); M \uparrow C : \text{pin}; C \downarrow M : \text{pin}(x); \quad M \uparrow C : \text{amount}; C \downarrow M : \text{amount}(y); M \downarrow B : \text{request}(n, x, y); B \downarrow M : \text{grant};$
 $M \uparrow C : \text{cash}(y); M \downarrow C : \text{card_out}]$

where n is name, x pin code, and y cash amount, provided by clients. [Example of unsuccessful cycle:](#)

$[C \uparrow M : \text{card_in}(n); M \uparrow C : \text{amount}; C \downarrow M : \text{amount}(y); M \downarrow B : \text{request}(n, x, y); B \downarrow M : \text{deny}; \quad M \uparrow C : \text{card_out}]$

Notation: $A \uparrow B : m$ denotes the sequence $A \uparrow B : m; A \downarrow B : m$

Mini bank example: Local histories (1)

From the global histories above, we may extract the corresponding local histories: [The successful cycle:](#)

- Client: $[C \uparrow M : \text{card_in}(n); M \downarrow C : \text{pin}; C \uparrow M : \text{pin}(x); \quad M \downarrow C : \text{amount}; C \uparrow M : \text{amount}(y); M \downarrow C : \text{cash}(y); M \downarrow C : \text{card_out}]$
- Mini Bank: $[C \downarrow M : \text{card_in}(n); M \uparrow C : \text{pin}; C \downarrow M : \text{pin}(x); \quad M \uparrow C : \text{amount}; C \downarrow M : \text{amount}(y); M \uparrow B : \text{request}(n, x, y);$
 $B \downarrow M : \text{grant}; M \uparrow C : \text{cash}(y); M \uparrow C : \text{card_out}]$
- Central Bank: $[M \downarrow B : \text{request}(n, x, y); B \uparrow M : \text{grant}]$

The local histories may be used as guidelines when implementing the different agents.

Mini bank example: Local histories (2)

[The unsuccessful cycle:](#)

- Client: $[C \uparrow M : \text{card_in}(n); M \downarrow C : \text{pin}; C \uparrow M : \text{pin}(x); \quad M \downarrow C : \text{amount}; C \uparrow M : \text{amount}(y); M \downarrow C : \text{card_out}]$
- Mini Bank: $[C \downarrow M : \text{card_in}(n); M \uparrow C : \text{pin}; C \downarrow M : \text{pin}(x); \quad M \uparrow C : \text{amount}; C \downarrow M : \text{amount}(y); M \uparrow B : \text{request}(n, x, y);$
 $B \downarrow M : \text{deny}; M \uparrow C : \text{card_out}]$
- Central Bank: $[M \downarrow B : \text{request}(n, x, y); B \uparrow M : \text{deny}]$

Note: many other executions possible, say when clients behaves differently, difficult to describe all at a global level (remember the formula of week 1).

Mini bank example: implementation of Central Bank

Sketch of simple central bank. [Program variables:](#)

```

pin -- array of pin codes, indexed by client names
bal -- array of account balances, indexed by client names
X : Agent, n: Client_Name, x: Pin_Code, y: Natural

```

```

Loop
  await X?request(n,x,y);
  if pin[n]=x and bal[n]>y
  then bal[n]:=bal[n]-y;
    send X:grant;
  else send X:deny
  fi
end

```

Note: the mini bank X may vary with each iteration.

Mini bank example: Central Bank (B)

Consider the (extended) regular expression $Cycle_B$ defined by:

$$[X \downarrow B : \text{request}(n, x, y); [B \uparrow X : \text{grant} + B \uparrow X : \text{deny}] \text{ some } X, n, x, y]^*$$

- with $+$ for choice, $[...]^*$ for repetition
- Defines cycles: *request* answered with either *grant* or *deny*
- notation $[\text{regExp} \text{ some } X, n, x, y]^*$ means that the values of X , n , x , and y are fixed in each cycle, but may vary from cycle to cycle.

Notation: Given an extended regular expression R . Let $h \text{ is } R$ denote that h matches the structure described by R . Example (for events a , b , and c):

- we have $(a; b; a; b) \text{ is } [a; b]^*$
- we have $(a; c; a; b) \text{ is } [a; [b + c]]^*$
- we do *not* have $(a; b; a) \text{ is } [a; b]^*$ (but it is a prefix)

Loop invariant of Central Bank (B): Let $Cycle_B$ denote the regular expression:

$$[X \downarrow B : request(n, x, y); [B \uparrow X : grant + B \uparrow X : deny] \text{ some } X, n, x, y]^*$$

Loop invariant: h is $Cycle_B$

Proof of loop invariant (entry condition): Must prove that it is satisfied initially: ε is Cycle_B , which is trivial.

Proof of loop invariant (invariance):

```

loop {hisCycleB}
  await X?request(n,x,y);
  if pin[n]=x and bal[n]>y
    then bal[n]:=bal[n]-y; send X:grant;
    else send X:deny
  fi
{hisCycleB}
end

```

Loop invariant of the central bank (B):

```

loop
  {  $h$  is  $Cycle_B$  }
  {  $\forall X, n, x, y. \text{if } pin[n] = x \wedge bal[n] > y \text{ then } h_1'' \text{ is } Cycle_B \text{ else } h_2'' \text{ is } Cycle_B$  }
  await  $X?$ request( $n, x, y$ );
  { if  $pin[n] = x \wedge bal[n] > y$  then  $h_1'$  is  $Cycle_B$  else  $h_2'$  is  $Cycle_B$  }
  if  $pin[n] = x$  and  $bal[n] > y$ 
  then  $bal[n] := bal[n] - y$ ;
    {  $(h; B \uparrow X : grant)$  is  $Cycle_B$  }
    send  $X$ :grant;
  else {  $(h; B \uparrow X : deny)$  is  $Cycle_B$  }
    send  $X$ :deny;
  fi
  {  $h$  is  $Cycle_B$  }
end

```

$$\begin{aligned} h_1'' &= h; X \downarrow B: request(n, x, y); B \uparrow X: grant \\ h_1' &= h; B \uparrow X: grant \end{aligned}$$

Analogously (with *deny*) for h'_2 and h''_2

Hoare analysis of central bank loop (cont.)

Verification condition:
$$h \text{ is } Cycle_B \Rightarrow \forall X, n, x, y. \text{ if } pin[n] = x \wedge bal[n] > y \\ \text{ then } (h; X \downarrow B : request(n, x, y); B \uparrow X : grant) \text{ is } Cycle_B \\ \text{ else } (h; X \downarrow B : request(n, x, y); B \uparrow X : deny) \text{ is } Cycle_B$$

where $Cycle_B$ is

$$[X \downarrow B: request(n, x, y); [B \uparrow X: grant + B \uparrow X: deny] \text{ some } X, n, x, y]^*$$

The condition follows by the general rule (regExp R and events a and b):

$$h \text{ is } R^* \wedge (a; b) \text{ is } R \Rightarrow (h; a; b) \text{ is } R^*$$

since $(X \downarrow B : request(n, x, y); B \uparrow X : grant)$ is $Cycle_B$ and $(X \downarrow B : request(n, x, y); B \uparrow X : deny)$ is $Cycle_B$

Local history invariant for the central bank (B)

$Cycle_B$ is

$$[X \downarrow B: request(n, x, y); [B \uparrow X: grant + B \uparrow X: deny] \text{ some } X, n, x, y]^*$$

Define the history invariant for B by:

$$h \leq Cycle_B$$

Let $h < R$ denote that h is a prefix of the structure described by R .

- intuition: if $h \leq R$ we may find some extension h' such that $(h; h')$ is R
- h is $R \Rightarrow h \leq R$ (but not vice versa)
- $(h; a)$ is $R \Rightarrow h \leq R$
- Example: $(a; b; a) \leq [a; b]^*$

Central Bank: Verification of the local history invariant

$$h \leq Cycle_B$$

- As before, we need to ensure that the history invariant is implied after each send/receive statement.
- Here it is enough to assume the conditions after each send/receive statement in the verification of the loop invariant

This gives 2 proof conditions:

1. **after send grant/deny** (i.e. after **fi**)

$$h \text{ is } Cycle_B \Rightarrow h \leq Cycle_B \quad \text{which is trivial.}$$

2. **after await request**

$$\text{if } \dots \text{ then } (h; B \uparrow X : grant) \text{ is } Cycle_B \text{ else } (h; B \uparrow X : deny) \text{ is } Cycle_B \\ \Rightarrow h \leq Cycle_B \quad \text{which follows from } (h; a) \text{ is } R \Rightarrow h \leq R.$$

Note: We have now proved that the implementation of B satisfies the local history invariant, $h \leq Cycle_B$.

Mini bank example: Local invariant of Client (C)

$$Cycle_C: [\quad C \downarrow X : card_in(n) + X \downarrow C : pin; C \uparrow X : pin(x) + X \downarrow C : amount; C \uparrow X : amount(y') + X \downarrow C : cash(y) \\ + X \downarrow C : card_out \text{ some } X, y, y']^*$$

History invariant:

$$h_C \leq Cycle_C$$

Note: The values of C , n and x are fixed in each cycle.

Note: The client is willing to receive cash and cards, and give card, at any time, and will respond to pin , and $amount$ messages from a mini bank X in a sensible way, without knowing the protocol of the particular mini bank. This is captured by $+$ for different choices.

Mini bank example: Local invariant for Mini bank (M)

$$Cycle_M: [\quad C \downarrow M : card_in(n); M \uparrow C : pin; C \downarrow M : pin(x); \quad M \uparrow C : amount; C \downarrow M : amount(y); \quad \text{if } y \leq 0 \text{ then } \varepsilon \text{ else } \\ M \uparrow B : request(n, x, y); [B \downarrow M : deny + B \downarrow M : grant; M \uparrow C : cash(y)] \text{ fi}; \quad M \uparrow C : card_out \text{ some } C, n, x, y]^*$$

History invariant:

$$h_M \leq Cycle_M$$

Note: communication with a fixed central bank. The client may vary with each cycle.

Mini bank example: obtaining a global invariant

Consider the parallel composition of C, B, M . Global invariant:

$$legal(H) \wedge H/\alpha_C \leq Cycle_C \wedge H/\alpha_M \leq Cycle_M \wedge H/\alpha_B \leq Cycle_B$$

Assuming no other agents, this invariant may *almost* be formulated by:

$$H \leq [C \downarrow M : card_in(n); M \uparrow C : pin; C \downarrow M : pin(x); \quad M \uparrow C : amount; C \downarrow M : amount(y); \quad \text{if } y \leq 0 \text{ then } M \uparrow C : card_out \\ \text{else } M \uparrow B : request(n, x, y); [B \downarrow M : deny; M \uparrow C : card_out + B \downarrow M : grant; M \uparrow C : cash(y); [M \downarrow C : cash(y) ||| M \uparrow C : card_out]] \text{ fi} \\ \text{some } n, x, y]^*$$

where $|||$ gives all possible interleavings. However, we have no guarantee that the cash and the card events are received by C before another cycle starts. Any next client may actually take the cash of C .

For proper clients it works OK, but improper clients may cause the Mini Bank to misbehave. Need to incorporate assumptions on the clients, or make an improved mini bank.

Improved mini bank based on a discussion of the global invariant

The analysis so far has discovered some weaknesses:

- The mini bank does not know when the client has taken his cash, and it may even start a new cycle with another client before the cash of the previous cycle is removed. This may be undesired, and we may introduce a new event, say $cash_taken$ from C to M , representing the removal of cash by the client. (This will enable the mini bank to decide to take the cash back within a given amount of time.)
- A similar discussion applies to the removal of the card, and one may introduce a new event, say $card_taken$ from C to M , so that the mini bank knows when a card has been removed. (This will enable the mini bank to decide to take the card back within a given amount of time.)
- A client may send improper or unexpected events. These may be lying in the network unless the mini bank receives them, and say, ignores them. For instance an old misplaced amount message may be received in (and interfere with) a later cycle. An improved mini bank could react to such message by terminating the cycle, and in between cycles it could ignore all messages (except $card_in$).

Summary

Concurrent agent systems, without network restrictions (need not be FIFO, message loss possible).

- [Histories](#) used for semantics, specification and reasoning
- correspondence between [global and local histories](#), both ways
- parallel [composition](#) from local history invariants
- [extension of Hoare logic](#) with send/receive statements
- [avoid interference](#), may reason as in the sequential setting
- [Bank example](#), showing
 - global histories may be used to exemplify the system, from which we obtain local histories, from which we get useful [coding help](#)
 - [specification](#) of local history invariants
 - [verification](#) of local history invariants from Hoare logic + [verification conditions](#) (one for each send/receive statement)
 - [composition](#) of local history invariants to a [global invariant](#)

13 Active Objects

28.11.2016

Aims for this lecture

[Distributed object-oriented systems and introduction to Creol/ABS](#)

- Consider the combination of OO, concurrency, and distribution
- Understanding active objects
 - interacting by *asynchronous method calls*
- A short introduction of (a variant of) *Creol*⁷⁷ using small example programs

Note: Inheritance and dynamic object creation not considered here.

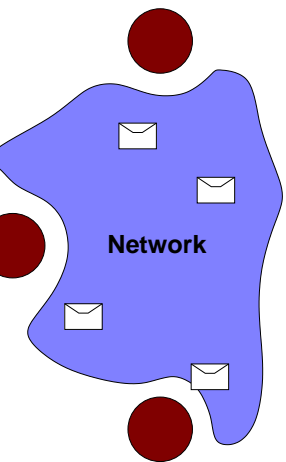
⁷⁷References:

- <http://heim.ifi.uio.no/~creol/>
- <http://tools.hats-project.eu/>
- Johnsen & Owe: An Asynchronous Communication Model for Distributed Concurrent Objects Software and Systems Modeling, 6(1): 39-58, 2007. Springer
- Johnsen, Blanchette, Kyas & Owe: Intra-Object versus Inter-Object Concurrency and Reasoning in Creol. Electron. Notes Theor. Comput. Sci. 243 (2009), 89-103.

- **Openness** : encapsulation
 - Implementation of other objects is not necessary known.

Open Distributed Systems Interaction with other objects is through interfaces.

- ODS *dominate* critical infrastructure in society: bank systems, air traffic control, etc.
- ODS: *complex, error prone*, and robustness is *poorly understood*



Challenges with OO languages for **modern systems**

Modern systems are often large and complex, with distributed, autonomous units connected through different kinds of networks.

- OO + concurrency
 - synchronization, blocking, deadlock
- OO + asynchronous communication
 - messages on top of OO or method-based communication? better than RPC/RMI?
- OO + distribution
 - efficient interaction (passive/active waiting),
- OO + openness
 - restricted knowledge of other objects
- OO + scalability
 - management of large systems

Active and Passive Objects

Passive objects

- Execute their methods in the caller's thread of control (e.g., Java)
- In multithreaded applications, must take care of synchronization
 - Shared variable interference for non-synchronized methods
- If two objects call the same object, race condition may occur

Active (or concurrent) objects

- Execute their methods in their own thread of control (e.g., Actors)
- Communication is asynchronous
- Call and return are decoupled (future variables)
- Cooperative multitasking, specified using schedulers

Creol: A Concurrent Object Model

- OO modeling language that targets open distributed systems
- All objects are **active** (or concurrent), but may receive requests
 - Need easy way to combine **active** and **passive/reactive** behavior
- We don't always know how objects are implemented
 - Separate **specification** (interface) from **implementation** (class)
 - Object variables are **typed by interface**, not by class
- **No assumptions** about the (network) environment
 - Communication may be unordered
 - Communication may be delayed
 - Execution should adapt to possible delays in the environment
- Synchronization decided by the caller
 - **Method invocations** may be *synchronous* or *asynchronous*

Interfaces as types

- Object variables (pointers) are *typed by interfaces* (other variables are typed by data types)
- *Mutual dependency*: An interface may require a *cointerface*
 - Only objects of cointerface type may call declared methods
 - Explicit keyword *caller* (identity of calling object)
 - Supports callbacks to the caller through the cointerface
 - **The cointerface is the minimal type of caller**
- All object interaction is *controlled* by interfaces
 - *No explicit hiding* needed at the class level
 - Interfaces provide behavioral specifications
 - A class may implement a number of interfaces
- **Type safety**: no “method not understood” errors, and all parameters are type correct

Interface syntax

- Declares a set of method signatures
- With *cointerface* requirement

```
interface I inherits  $\bar{I}$  begin
  with J           // cointerface J
     $\overline{MtdSig}$ 
end
```

- Method signatures (\overline{MtdSig}) of the form:

```
op m (in  $\bar{x} : \bar{I}$  out  $\bar{y} : \bar{I}$ )
```

- method name *m* with **in-parameters** \bar{x} and **out-parameters** \bar{y}
- **Local data structures** inside an object is defined by **data types**,
 - including lists, sets and user-defined data types and
 - predefined types such as *Bool*, *Int*, *String*...

Interfaces: Example

- Consider the **mini bank** example from last week
- We have **Client**, **MiniBank**, and **CentralBank** objects
- **Clients** may support the following interface:

```
interface Client begin
  with MiniBank      -- the cointerface
    op pin(out p : Int)
    op amount(out a : Int)
end
```

- only **MiniBank** objects may call the **pin** and **amount** methods

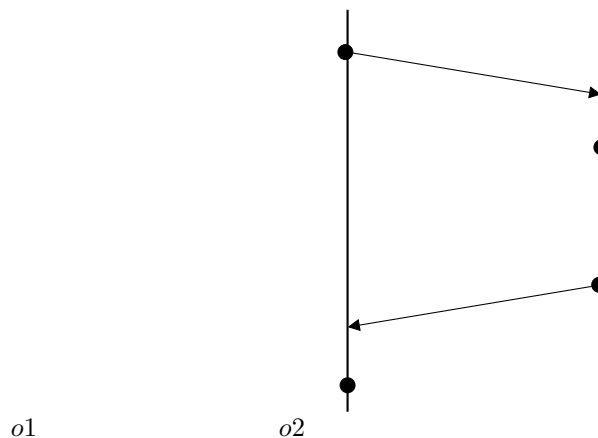
Interfaces: Example (cont.)

MiniBank and **CentralBank** interfaces:

```
interface MiniBank begin
  with Client
    op withdraw(in name : String out result : Bool)
end

interface CentralBank begin
  with MiniBank
    op request(in name : String, pin : Int, amount : Int
              out result : Bool)
end
```

Asynchronous Communication Model



- Object *o1* calls some method on object *o2*
- In *o2*: Arbitrary delay after invocation arrival and method startup
- In *o1*: Arbitrary delay after completion arrival and reading the return

***o1* may do something else while waiting for *o2* to respond**

Main ideas of Creol: Programming perspective

Main ideas: Overall

- interaction by method calls
- method executions (**processes**) may be suspended
- queue of suspended method executions (**the process queue**)

Main ideas: Method interaction

- Asynchronous communication
- Avoid undesired inactivity
 - Other processes may execute while a process waits for a reply
- Combine active and reactive behavior

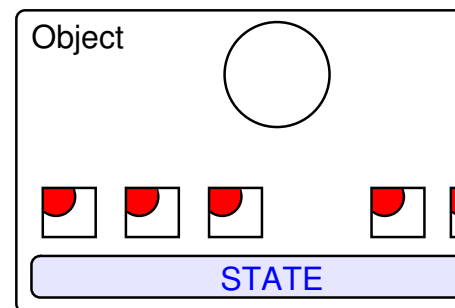
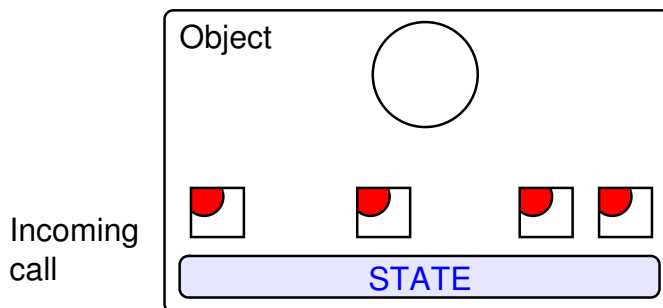
In the language, this is achieved by statements for

- **asynchronous method calls** and
- **processor release points**

Release points enable *interleaving* of *active* and *reactive* code **Note:** No need for signaling / notification

Execution inside a Creol Object

- **Concurrent objects** encapsulate a processor
- Execution in objects should *adapt* to environment delays
- At most *one active process* at a time
- *Implicit scheduling* between internal processes inside an object



Internal Processes in Concurrent Objects

- **Object:** *state* + *active process* + *suspended processes*
- **Process** (method activation):
 - code + local variable bindings (local state)
- **Asynchronous** invocation: *t!o.m(In)*, *In* a list of expressions
 - The *label t* identifies the call

Reading the result: *t?(Out)*, *Out* a list of variables

- **Processor release points**
 - Declared by **await** statements: **await guard**
 - *Guards* can be
 - * *t?*
 - * Boolean condition
 - * and also method call
 - If a *guard* evaluates to **false**, the active process is *suspended*
 - If no process is active, any suspended process may be *activated* if its guard evaluates to **true**.

Statements for object communication

- Objects communicate through method invocations *only*
- *Different ways to invoke* a method *m*
- Decided by caller — *not* at method declaration site
- **Guarded** invocation:

t!*o.m(In); ...; await t?; t?(Out)*

- **Label-free abbreviations** for standard patterns:
 - *o.m(In; Out) ≡ t!o.m(In); t?(Out)* — **synchronous call**
 - **await** *o.m(In; Out) ≡ t!o.m(In); await t?; t?(Out)*
 - *!o.m(In)* — no reply needed (one-way message passing)
- **Internal calls:** *m(In; Out)*, *t!m(In)*, *!m(In)* Internal calls may also be asynchronous/guarded

From now on: only label-free calls

Creol syntax

<i>Syntactic categories.</i>	<i>Definitions.</i>
	$g ::= \phi \mid t? \mid g_1 \wedge g_2$
t in Label	$p ::= o.m \mid m$
g in Guard	$S ::= s \mid s; S$
p in MtdCall	$s ::= \text{skip} \mid \text{begin } S \text{ end} \mid S_1 \square S_2 \text{ } s_1 \square s_2$
S in ComList	$ x := e \mid x := \text{new } \text{classname}(e)$
s in Com	$ \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ end}$
x in VarList	$ \text{while } b \text{ do } S \text{ end}$
e in ExprList	$!p(e) \mid t!p(e) \mid t?(x) \mid p(e; x)$
m in Mtd	$ \text{await } g \mid \text{await } p(e; x) !p(e) \mid t!p(e) \mid t?(x) \mid p(e; x)$
o in ObjExpr	$ \text{await } g \mid \text{await } p(e; x)$
b in BoolExpr	$ \text{releaser} \text{release}$
	$e ::= x \mid \text{this} \mid \text{caller} \mid \text{null} \mid e = e \mid f(e) \mid \dots$

- Omit the functional language for expressions **e** here: booleans, integers, strings, lists, sets, maps, etc

Example: CentralBank implementation

```
class Bank implements CentralBank begin
  var pin -- pin codes, indexed by name
  var bal -- balances, indexed by name

  with MiniBank
    op request(in name : String, pin : Int, amount : Int
              out result : Bool) ==
      result := (pin[name] = pin && bal[name] >= amount)
  end
end
```

Example: MiniBank implementation

```
class MiniBank(bank : CentralBank) implements MiniBank begin
  with Client
    op withdraw(in name : String out result : Bool) ==
      var amount : Int, pin : Int;
      caller.pin(;pin); caller.amount(;amount)
      await bank.request(name, pin, amount; result)
  end
end
```

- method calls `caller.pin(...)` and `caller.amount(...)` are type safe by cointerface requirements
- **await** statement: passive waiting for reply from CentralBank

Example: Client implementation

Optimistic client:

```
class Person(m : MiniBank) implements Client begin
  var name : String, pin : Int;

  op init == !run()

  op run() == success : Bool;
    await m.withdraw(name;success); -- active behavior
    if (success = false) then !run() end

  with MiniBank
    op pin(out p : Int) == p := pin
    op amount(out a : Int) == a := 1000
  end
end
```

- Assuming communication with a fixed minibank m

Main ideas of Creol: Programming perspective

- **concurrent objects** (each with its own virtual processor)
- a notion of **asynchronous methods calls**, avoids blocking, using **processor release points**
- high level **process control**
 - no explicit signaling/notification
 - busy waiting avoided!
- openness by a notion of **multiple interfacing**
- type-safe call-backs due to **cointerfaces**

Remark: abstraction by **behavioral interfaces**

Example: Buffer

```
interface Buffer begin
  with Producer op put(in x : Int)
  with Consumer op get(out x : Int)
end

class OneSlotBuffer implements Buffer begin
  var value : Int, full : Bool;
  op init == full := false
  with Producer
    op put(in x:Int) == await ¬full; value:= x; full:= true
  with Consumer
    op get(out x:Int) == await full; x:= value; full:= false
  end
end
```

- **init**: initialization code executed at object creation
- synchronization by boolean **await**

Example: Buffer (cont.)

Illustrating alternation between active and reactive behavior

```
class Consumer(buf: Buffer) implements Consumer begin
  var sum : Int := 0;

  op init == !run()

  op run() == var j : Int;
    while true do await buf.get(;j); sum := sum + j end
  with Any op getSum(out s : Int) == s := sum
end
```

- Call to `buf.get`:
 - Asynchronous
 - **await**: processor release
 - Incoming calls to `getSum` can be served while waiting for reply from `buf`
- Interface `Any`: supertype of all interfaces
 - Any object can call `getSum`

Readers/Writers example (Simple implementation)

```
interface RW begin with RWClient    op OR() — open read    op OW() — open write    op CR() — close read    op CW() — close write end
```

```
class RW implements RW begin var r: Int:=0; var w: Int:=0; with RWClient    op OR() == await w=0; r:= r+1    op OW() == await w=0 and r=0; w:= w+1    op CR() == r:= r-1    op CW() == w:= w-1 end
```

Note: A client may do asynchronous calls to `OR/OW` and synchronous calls to `CR/CW`.

Readers/Writers example (version 2)

```
class RW(db : DataBase) implements RW begin
  var readers : Set[Reader] := {}, writer : Writer := null,
    pr : Int := 0; // number of pending calls to db.read

  with Reader
    op OR() == await writer = null; readers := readers ∪ caller
    op CR() == readers := readers \ caller
    op read(in key : Int out result : Int) ==
      await caller ∈ readers;
      pr := pr + 1; await db.read(key;result); pr := pr - 1;

  with Writer
    op OW() == await (writer = null && readers = {} && pr = 0);
      writer := caller
    op CW() == await caller = writer; writer := null
    op write(in key : Int, value : Int) ==
      await caller = writer; db.write(key,value);

end
```

RW example, remarks (version 2)

- read and write operations on database may be declared with cointerface `RW`
- Weaker assumptions about `Reader` and `Writer` behavior than in the first version
 - Here we actually check that only registered readers/writers do read/write operations on the database
- The database is assumed to store integer values indexed by `key`
- Counting the number of pending calls to `db.read` (variable `pr`)

- A reader may call `CR` before all `read` invocations are completed
- For writing activity, we know that there are no pending calls to `db.write` when `writer` is `null`. Why?
- The solution is unfair: writers may starve
- Still, after completing `OW`, we assume that writers will eventually call `CW`. Correspondingly for readers

Summary: Active Objects

- Passive objects usually execute their methods in the thread of control of the caller (Java)
- In multithreaded applications, we must take care of proper synchronization
- Active objects execute their methods in their own thread of control
- Communication is asynchronous
- synchronous communication possible by means of asynchronous communication primitives
- Call and return are decoupled by the use of *labels*
- Usually, active objects use *cooperative* multitasking.
- Cooperative multitasking is specified using *schedulers*. Our scheduler will just randomly pick a next process.

Other versions of Creol/ABS

- with time
- with probabilities
- dynamic class upgrades
- futures (call labels as first class citizens)
- (single and) multiple inheritance
- traits and deltas for software product lines
- groups (sharing a CPU)
- abstract objects defined by object sets
- awareness and modeling of (different) underlying networks
- cloud awareness
- security awareness

For Hoare-style reasoning see [Din and Owe, 2014, Din et al., 2012]

Other PMA Courses

Spring:

- INF3230/INF4231 - Formal modeling and analysis of communicating systems [rewriting logic - language and tool Maude](#)
- INF5110 - Compiler construction (each spring)
- INF5140/INF9140 - Specification and verification of parallel systems. ('15, '17,...) [Automatic verification using model checking techniques](#)
- INF5906/INF9906 - Selected topics in static analysis. ('16, '18,...) [analysis of programs at compile time](#)

Fall:

- INF5130/INF9130 - Selected topics in rewriting logic ('15, '17,...)

References

- [Andrews, 2000] Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley.
- [Din et al., 2012] Din, C. C., Dovland, J., Johnsen, E. B., and Owe, O. (2012). Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256.
- [Din and Owe, 2014] Din, C. C. and Owe, O. (2014). A sound and complete reasoning system for asynchronous communication with shared futures. *J. Log. Algebr. Meth. Program.*, 83(5-6):360–383.
- [Goetz et al., 2006] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. (2006). *Java Concurrency in Practice*. Addison-Wesley.
- [Lea, 1999] Lea, D. (1999). *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 2d edition.
- [Magee and Kramer, 1999] Magee, J. and Kramer, J. (1999). *Concurrency: State Models and Java Programs*. John Wiley & Sons Inc.