INF4140 - Models of concurrency

Høsten 2014

November 24, 2014

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, 1991]. 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. Not included here is the material about weak memory models.

1 Intro

29. 08. 2014

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[1] 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:
  - (informal and somewhat formal) analysis
  - properties

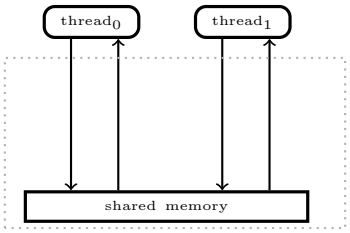
[1] you!, as course particpant

Parallel processes

- Sequential program: one control flow thread
- Parallel 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 of the course)
- Communication with *messages* between processes (part II of the course)



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

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 important 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".[2]

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

[2]Holds for concurrency in general, not just shared vars, of course.

Simple example

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

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

Pre/post-condition

- executing a program (resp. a program fragment) $\Rightarrow$ 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 $\|$

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

$$\texttt{co}\ \ S_1 \parallel S_2 \parallel \ldots \parallel S_n\ \ \texttt{oc}$$

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

Example 1.

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

Interaction between processes

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

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

The organization of this interaction is what we will call *synchronization*.

Synchronization

Synchronization (veeeery 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 can*not* be executed concurrently
- *Condition synchronization:* A process must wait for a specific condition to be satisfied before execution can continue.

Concurrent processes: Atomic operations

Definition 2 (Atomic)**.** An operation is *atomic* if it 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 and writing of a global variable is usually atomic.[3]
- For some (high-level) languages: assignments $x := e$ atomic operations, for others, not (reading of the variables in the expression e, computation of the value e, followed by writing to x.)

[3] That's what we 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 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.
- associated to global variables: a set of *atomic operations*
- typically: read + write,
- in HW, e.g. LOAD/STORE
- channels as gobal data: *send* and *receive*
- *x-operations*: atomic operations on a variable x

Mutual exclusion
Atomic operations on a variable cannot happen simultaneously.

Example

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

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

Atomic read and write operations

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

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

```
1  read x;
2  inc;
3  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":[4]
  - R1 must happen before W1 and
  - R2 before W2
- `inc` and `dec` ("-1") work process-local[5]

$\Rightarrow$ remember (e.g.) `inc; write x` behaves "as if" atomic (alternatively `read x; inc`)

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

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:[6] $x = -1 \vee x = 0 \vee x = 1$

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

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

For n processes with m atomic statements each:

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

[4] A word aside: as natural as this seems: in a number of modern architecture/modern languages & their compilers, *this is not guaranteed!* Cf. Java's memory model, or weak memory models in general.

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

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

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 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* crit. reference
2. e with *at most one* crit. reference & x not *referenced*[7] by other proc's

assigments 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 $\ldots \parallel \ldots$ omitted

$$
\begin{array}{ll}
x := x + 1 \parallel y := x + 1 & \\
x := y + 1 \parallel y := x + 1 & \{\ (x, y) \in \{(1,1),(1,2),(2,1)\}\ \} \\
x := y + 1 \parallel x := y + 3 \parallel y := 1 & \{y{=}1 \wedge x{=}1,2,3,4\} \\
r := y + 1 \parallel r' := y - 1 \parallel y := 5 & \\
r := x - x \parallel \ldots & \{\text{is r now 0?}\} \\
x := x \parallel \ldots & \{\text{same as skip?}\} \\
\texttt{if } y > 0 \texttt{ then } y := y - 1 \texttt{ fi} \parallel \texttt{if } y > 0 \texttt{ then } y := y - 1 \texttt{ fi} &
\end{array}
$$

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

Syntax
We use the following syntax for non-parallel control-flow[8]

Declarations	Assignments
`int i = 3;`	`x := e;`
`int a[1:n];`	`a[i] := e;`
`int a[n];`[9]	`a[n]++;`
`int a[1:n] = ([n] 1);`	`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*

[7] or just read.
[8] The book uses more C/Java kind of conventions, like = for assignment and == for logical equality.

Parallel statements

$$\texttt{co}\, S_1 \parallel S_2 \parallel \ldots \parallel S_n\, \texttt{oc}$$

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

Parallel processes

```
1  process foo {
2    int sum := 0;
3    for [i=1 to 10]
4      sum +:= 1;
5    x := sum;
6  }
```

- Processes evaluated in arbitrary order.
- Processes are declared (as methods/functions)
- side remark: the convention "declaration = start process" is *not* used in practice.[10]

Example

```
process bar1 dir0o
for [i = 1 to n]
write(i); }
```

Starts one process.

The numbers are printed in increasing order.

```
process bar2[i=1 to n] dir0o
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

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

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

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

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

- note: expressions side-effect free

[10] one 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.

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 program[11] is a predicate which is true for all possible histories of the program.
- Two types:
  - *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.[12]
- *total correctness*: The program terminates and is partially correct.

Properties: Invariants

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

  Definition 4 (Invariant)**.** an *invariant* = state property, which holds for 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 state of many processes at once, preferably the entire system

[11] the program "has" that property, the program satisfies the property ...
[12] that's also called *strong* termination. Remember: non-determinism.

- *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 parallel processes does not hold, we use *synchronization* to restrict the possible histories.
- Synchronization gives coarse-grained atomic operations.
- The notation $\langle S \rangle$ means that S is performed *atomically.*[13]

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:

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

Conditional critical sections

Await statement

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

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

  Example 5.

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

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

[13] In programming languages, one could find it as $\texttt{atomic}\{S\}$ or similar.

2 special cases

- unconditional critical section or "mutex"[14]

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

- Condition synchronization:[15]

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

Typical pattern

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

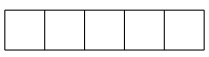
- "critical statements" *not* enclosed in ⟨angle brackets⟩. Why?
- *invariant:* $0 \leq \mathit{counter} \leq 1$ (= counter acts as "*binary lock*")
- very bad style would be: touch counter inside "critical statements" or elsewhere (e.g. access it *not* following the "await-inc-CR-dec" pattern)
- in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

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

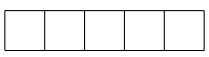
- strong *coupling*
- `buf` as shared variable ("one element buffer")
- *synchronization*
  - coordinating the "speed" of the two procs (rather strictly here)
  - to avoid, reading data which is not yet produced
  - (related:) avoid w/r conflict on shared memory

```
1
2                int buf, p := 0; c := 0;
3
4  process Producer {                  process Consumer {
5    int a[N];...                      int b[N];...
6    while (p < N) {                   while (c < N) {
7      < await (p = c) ; >              < await (p > c) ; >
8      buf := a[p];                      b[c] := buf;
9        p := p+1;                          c := c+1;
10   }                                   }
11 }                                   }
```

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`

[14] Later, a special kind of semaphore (a binary one) is also called a "mutex". Terminology is a bit flexible sometimes.

[15] one may also see sometimes just `await`(b): however, eval. of b better be *atomic* and under *no* circumstances must b have *side-effects* (*never, ever. Seriously*).

2 Locks & barriers

5. 9. 2014

Practical Stuff

Mandatory assignment 1 ("oblig")

- Deadline: Friday September 26 at 18.00
- Possible to work in pairs
- Online delivery (Devilry): https://devilry.ifi.uio.no

Introduction

- Central to the course are general mechanisms and issues related to parallel programs
- **Previous class:** *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

```
1
2                int buf, p := 0; c := 0;
3
4  process Producer {                  process Consumer {
5    int a[N];...                      int b[N];...
6    while (p < N) {                   while (c < N) {
7      < await (p = c) ; >              < await (p > c) ; >
8      buf := a[p];                      b[c] := buf;
9        p := p+1;                         c := c+1;
10   }                                   }
11 }                                   }
```

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 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 we are discussing 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
- A solution to the CS problem can be used to *implement* **await**-statements

Critical section: First approach to a solution

Operations on shared variables happen 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 {                    process p2 {
  while (true) {                  while (true) {
    while (in=2) {skip};            while (in=1) {skip};
    CS;                             CS;
    in := 2;                        in := 1
    non-CS                          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.

NB: The three first are safety properties,[16] The last a liveness property.
(SAFETY: no bad state, LIVENESS: something good will happen.)

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

Safety: Invariants (review)

A safety property expresses that 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 sections

Used for synchronization of processes

- General form:

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

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

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

$$\langle S \rangle \tag{1}$$

  S executed atomically

- Conditional synchronization:[17]

$$\langle \texttt{await}(B) \rangle \tag{2}$$

Critical sections using locks

```
1  bool lock = false;
2
3  process [i=1 to n] {
4    while (true) {
5      < await (¬ lock) lock := true >;
6      CS;
7      lock := false;
8      non CS;
9    }
10 }
```

Safety properties:

- Mutex
- Absence of deadlock
- Absence of unnecessary waiting

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

"Test & Set"

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

```
1  TS(lock) {
2    < bool initial := lock;
3    lock := true >;
4    return initial
5  }
```

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.

[17] We 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
  }
}
```

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

Strong fairness needed to guarantee eventual entry for a process

Variable lock becomes a hotspot!

A puzzle: "paranoid" entry protocol

Better safe than sorry?

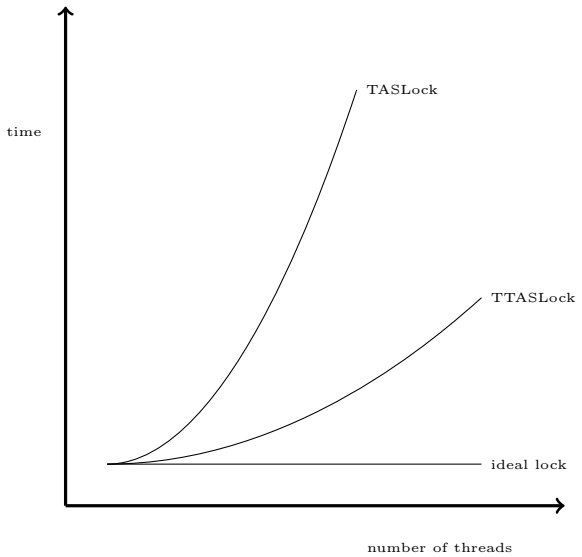
What about *double-checking* in the entry protocol whether it is *really, really* safe to enter?

```
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
  }
}
```

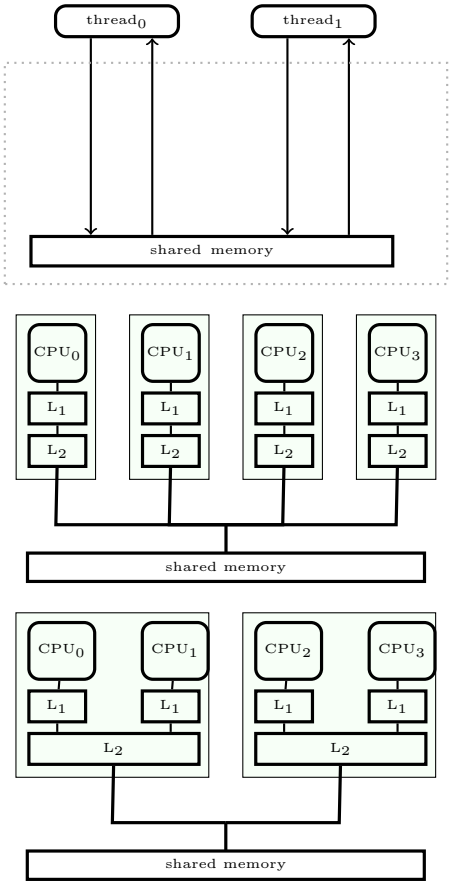
```
bool lock := false;

process p[i = i to n] {
  while (true) {
    while (lock) {skip};     # additional spin lock check
    while (TS(lock)) {
      while (lock) {skip}}; # + more 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": important design pattern/programming idiom for efficient CS (under certain architectures)[18]

Implementing await-statements

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

Then the statement < S;> can be implemented by

CSentry; S; **CSexit**;

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

```
1  CSentry;
2    while (!B) {CSexit ; CSentry};
3    S;
4  CSexit;
```

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

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

2.2 Liveness and fairness

Liveness properties

So far: no(!) solution for "Eventual Entry"-property, except the very first (which did not satisfy "Absence of Unnecessary Delay").

- Liveness: Something good will happen
- Typical example for sequential programs: (esp. in our context) Program termination[19]
- Typical example for parallel programs: A given process will eventually enter the critical section

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

Scheduling and fairness

- A command is *enabled* in a state if the statement can in principle be executed next
- Concurrent programs: often more than 1 statement enabled!

```
1  bool x := true;
2
3  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 → enabled (of course),
  - but also enabled → 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

[19] In the first version of the slides of lecture 1, termination was defined misleadingly.
[20] provided the control-flow/program pointer stands in front of them.

Unconditional fairness

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

Example:

```
1  bool x := true;
2
3  co  while (x){ skip }; || x := false co
```

- x := **false** is unconditional

⇒ The action will eventually be chosen

- This guarantees termination
- Example: "Round robin" execution
- Note: if-then-else, **while** (b) ; are *not* conditional atomic statements!

Weak fairness

Weak fairness

A scheduling strategy is *weakly fair* if

- it is 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:

```
1  bool x = true, int y = 0;
2
3  co while (x) y = y + 1; || < await y ≥ 10; > x = false; oc
```

- When y ≥ 10 becomes true, this condition remains true
- This ensures termination of the program
- Example: Round robin execution

Strong fairness

Example

```
1    bool x := true; y := false;
2
3    co
4        while (x) {y:=true; y:=false}
5    ||
6        < await(y) x:=false >
7    oc
```

Definition 6 (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 ⇒ termination
- under *weak fairness:* non-termination possible

Fairness for critical sections using locks

The CS solutions shown need to assume 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**
- Weak fairness: Process i can read lock only when the value is **false**
- Strong fairness: Guarantees that i eventually sees that lock is **true**

Difficult: to make a 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

```
1  in1 := false, in2 := false;
2
3  process p1 {                          process p2 {
4    while (true){                         while (true) {
5      while (in2) {skip};                   while (in1) {skip};
6      in1 := true;                            in2 := true;
7      CS                                      CS ;
8      in1 := false;                           in2 := false;
9      non-CS                                  non-CS
10   }                                     }
11 }                                   }
```

What is the global invariant here?

Problem: No *mutex*

Tie-Breaker algorithm: Attempt 2

```
1  in1 := false, in2 := false;
2
3  process p1 {                          process p2 {
4    while (true){                         while (true) {
5      while (in2) {skip};                   while (in1) {skip};
6      in1 := true;                            in2 := true;
7      CS                                      CS ;
8      in1 := false;                           in2 := false;
9      non-CS                                  non-CS
10   }                                     }
11 }                                   }
```

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

process p1 {                          process p2 {
  while (true){                         while (true) {
    in1 := true;                          in2 := true;
    while (in2) {skip};                   while (in1) {skip};
    CS                                    CS ;
    in1 := false;                         in2 := false;
    non-CS                                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 ⇒ 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

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.

[21] Technically, 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 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
  }
}
```

- The first line in the loop must be performed atomically!
- **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).

[22] Why?

Ticket algorithm: Invariant

Invariants

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

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

- What is the *local* invariant for process i:
  - turn[i] $<$ number
  - if p[i] is in the CS then turn[i] = next.
- for pairs of processes $i \neq j$:

  if turn[i] > 0 **then** turn[j] $\neq$ 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.

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!!

Divides shared counter into several local variables.

```
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 := 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
    }
  }
}
```

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 fever processors than processes
  - Should not waste time executing a skip loop!
- No clear distinction between variables used for synchronization and computation!

Desirable to have a special tools for synchronization protocols

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

3 Semaphores

12 September, 2014

3.1 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

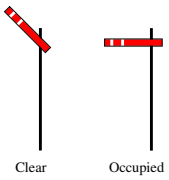
- This lecture: Semaphores (synchronization tool)
  - Used easily for mutual exclusion and condition synchronization.
  - A way to implement signaling and (scheduling).
  - Can be implemented in many ways.

Outline

- Semaphores: Syntax and semantics
- Synchronization examples:
  - Mutual exclusion (Critical Section)
  - 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:[23]

  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

[23] There are different stories about what Dijkstra actually wanted V and P to stand for.

Syntax and semantics

- declaration of semaphores:
  - sem s; default initial value is zero
  - sem s := 1;
  - sem s[4] := ([4] 1);
- semantics[24] (via "implementation"):

  P-operation P(s)

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

  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

Example: Mutual exclusion (critical section)
Mutex[25] implemented by a binary semaphore

```
9   sem mutex := 1;
10  process CS[i = 1 to n] {
11    while (true) {
12      P(mutex);
13        criticalsection;
14      V(mutex);
15      noncriticalsection;
16  }
```

Note:

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

[24] meaning
[25] As mentioned: "mutex" is also used to refer to a data-structure, basically the same as binary semaphore itself.

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

3.2 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

$\Rightarrow$ code between the P and the V

- all semaphores = 0
- code executed in mutex

Example: Producer/consumer with split binary semaphores

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

```
1  process Producer {
2    while (true) {
3      P(empty);
4      buff := data;
5      V(full);
6    }
7  }
```

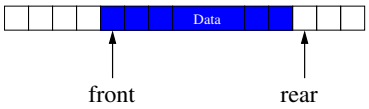
```
1  process Consumer {
2    while (true) {
3      P(full);
4      buff := data;
5      V(empty);
6    }
7  }
```

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

- previous example: strong 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 communcation ⇒ "buffering"
  - ring-buffer, typically represented
    - by an array
    - + two integers `rear` and `front`.
  - semaphores to keep track of the number of free/used slots ⇒general semaphore



Producer/consumer: increased buffer capacity

```
1   T buf[n]                    # array, elements of type T
2   int front := 0, rear := 0; # ''pointers''
3   sem empty := n,
4   sem full = 0;
```

```
1  process Producer {
2    while (true) {
3       P(empty);
4      buff[rear] := data;
5      rear := (rear + 1) % n;
6       V(full);
7    }
8  }
```

```
1  process Consumer {
2    while (true) {
3       P(full);
4      result := buff[front];
5      front  := (front + 1) % n
6       V(empty);
7    }
8  }
```

several producers or consumers?

Increasing the number of processes

- several producers and consumers.
- New synchronization problems:
  - Avoid that two producers deposits to `buf[rear]` before `rear` is updated
  - Avoid that two consumers fetches from `buf[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

```
1   T buf[n]                          # array, elem's of type T
2   int front := 0, rear := 0;        # ''pointers''
3   sem empty := n,
4   sem full = 0;
5   sem mutexDeposit, mutexFetch := 1; # protect the data stuct.
```

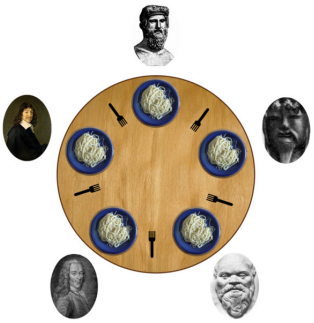
```
process Producer {
  while (true) {
    P(empty);
    P(mutexDeposit);
    buff[rear] := data;
    rear := (rear + 1) % n;
    V(mutexDeposit);
    V(full);
  }
}
```

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

3.3 Dining philosophers

Problem: Dining philosophers introduction

- famous sync. problem (Dijkstra)
- Five philosophers sit 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`

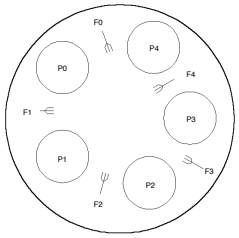
Dining philosophers: 1st attempt

- forks as semaphores
- let the philosophers pick up the left fork first

[26] image from wikipedia.org

```
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 philospher (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 philosphers

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

3.4 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 importantclasses 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"
  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);
  }
}
```

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

    write to DB

    V(rw);
  }
}
```

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

Readers/writers with mutex (2)

Initially:

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

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

    read from DB

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

}
```

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

    P(rw);

    write to DB


    V(rw);
  }

}
```

Semaphore inside await statement? Don't try that at home.

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 invariant[27]

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 or nw = 0) 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 though:
  - before increasing `nr`: `nw = 0`
  - before increasing `nw`: `nr = 0 and nw = 0`

condition synchronization: without semaphores

Initially:

```
int nr := 0;    # nunber 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>
  }
}
```

[27] 2nd 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...

  General pattern

  - entry[28] semaphore e, initialized to 1
  - For each guard B_i:
    - associate 1 counter and
    - 1 delay-semaphore

    both 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

```
1  int nr := 0, nw = 0;          # condition variables (as before)
2  sem e := 1;                   # delay semaphore
3  int dr := 0; sem r := 0;      # delay counter + sem for reader
4  int dw := 0; sem w := 0;      # delay counter + sem for writer
5  # invariant RW: (nr = 0 ∨ nw = 0 ) ∧ nw ≤ 1
```

```
1  process Reader [i=1 to M]{   # entry condition: nw = 0
2    while (true) {
3      ...
4      P(e);
5      if (nw > 0) { dr := dr + 1;   # < await (nw=0)
6                    V(e);           #     nr:=nr+1 >
7                    P(r)};
8      nr:=nr+1; SIGNAL;
9
10     read from DB;
11
12     P(e); nr:=nr -1; SIGNAL;      # < nr:=nr-1 >
13   }
14 }
```

[28] Entry 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);                          # < await (nr=0 ∧ nw=0)
      if (nr > 0 or nw > 0) {        #     nw:=nw+1 >
          dw := dw + 1;
          V(e);
          P(w) };
      nw:=nw+1; SIGNAL;

      write 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
```

4 Monitors

19. Sep 2014

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:
  - Forgotten P or V operations
  - Too many P or V operations
  - They are shared between processes
    - Global knowledge
    - May need to examine all processes to see how a semaphore works

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 a 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
- Condition synchronization:[29] is given by *condition variables*
- At a lower level of abstraction: monitors can be implemented using locks or semaphores

Usage

- processs = 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 of the called procedure are important
- the implementation can be changed, if visible effect remains the same
- Monitors and processes can be developed relatively independent $\Rightarrow$ Easier to understand and develop parallel programs

[29] block 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*)

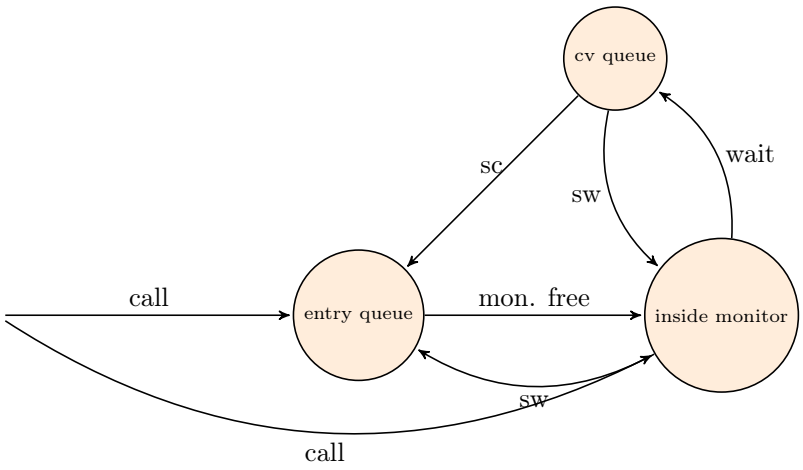
- statements *inside* a monitor: *no* access to variables *outside* the monitor
- monitor variables: initialized before the monitor is used

monitor invariant: used to describe the monitor's inner states

Condition variables

- monitors contain *special* type of variables: `cond` (condition)
- used for synchronizaton/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, manipulate it by special operations

```
cond cv;          # declares a condition variable cv
empty(cv);        # asks if the queue on cv is empty
wait(cv);         # causes the 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
```



4.1 Semaphores & signalling disciplines

Implementation of semaphores

A monitor with P and V operations:

```
monitor Semaphore { # monitor invariant: s ≥ 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` constitute a *FIFO signaling strategy*
- When a process executes `signal(cv)`, then it is inside the monitor. If a waiting process is woken up, there would be *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 ≥ 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 ≥ 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_fifo { # 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);
  }
}
```

4.2 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 discipline[30]

[30] It's the commonly used one in practical languages/OS.

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

  $\Longrightarrow$ 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

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);
      }
}
```

4.3 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 cannot be encapsulated in a monitor, as the readers will not get shared access
- monitor instead used to give access to 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

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

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.

```
1  monitor RW_Controller {  # RW  (nr = 0 or nw = 0) and nw ≤ 1
2    int nr:=0, nw:=0
3    cond oktoread ;    # signalled when nw = 0
4    cond oktowrite;    # sig'ed when nr = 0 and nw = 0
5
6    procedure request_read() {
7      while (nw > 0) wait(oktoread);
8      nr := nr + 1;
9    }
10   procedure release_read() {
11     nr := nr - 1;
12     if nr = 0 signal (oktowrite);
13   }
14
15   procedure request_write() {
16     while (nr > 0 or nw > 0) wait(oktowrite);
17     nw := nw + 1;
18   }
19
20   procedure release_write() {
21     nw := nw -1;
22     signal(oktowrite);      # wake up 1 writer
23     signal_all(oktoread);   # wake up all readers
24   }
25 }
```

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`

  $\Rightarrow$ 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)

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

```
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
}
```

4.4 Time server

Time server

- Monitor that enables sleeping for a given amount of time
- Resource: a logical clock (`tod`)
- Provides two operations:
  - `delay(interval)` the 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` 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 effective 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` $\geq$ `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);
    }
}
```

4.5 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);
}
```

4.6 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 the he barber chair at a time.
- Without customers: barber sleeps in one of the chairs.
- When a customer arrives and the barber sleeps ⇒ barber is woken up and the customer takes a seat.
- barber busy ⇒ 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.[31]

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

The barber can have rendezvous with an arbitrary customer.

[31] Later, 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.

Organize the synch.: 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

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

```
1  monitor Barber_Shop {
2    int barber := 0, chair := 0, open := 0;
3    cond barber_available;          # signalled when barber > 0
4    cond chair_occupied;            # signalled when chair > 0
5    cond door_open;                 # signalled when open > 0
6    cond customer_left;             # signalled when open = 0
7
8  procedure get haircut() {
9    while (barber = 0) wait(barber_available); # RV with barber
10   barber := barber - 1;
11   chair := chair + 1; signal(chair_occupied);
12
13   while (open = 0) wait(door_open);              # leave shop
14   open := open - 1; signal(customer_left);
15 }
16 procedure get next customer() {                  # RV with client
17   barber := barber + 1; signal (barber available);
18   while (chair = 0) wait(chair_occupied);
19   chair := chair - 1;
20 }
21  procedure finished_cut() {
22    open := open + 1; signal(door_open);          # get rid of customer
23    while (open > 0) wait(customer_left);
24 }
```

5 Program analysis

26.9.2014

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:

$$\texttt{co}\ \langle x := x \times 3;\rangle \parallel \langle y := y \times 2;\rangle\ \texttt{oc}$$

- 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:

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

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

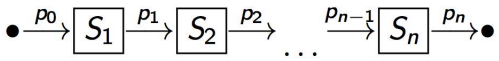
- A *state* of a program consists of the values of the program variables at a point in time, example: $\{\ x = 2 \wedge y = 3\ \}$
- The *state space* of a program is given by the different values that the declared variables can take
- Sequential program: one execution thread operates on its own state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

Example 7.

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

Executions

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



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

- Can be documented by: $\{p_0\}S_1\{p_1\}S_2\{p_2\}\ldots\{p_{n-1}\}S_n\{p_n\}$
- 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 8 (from previous slide).

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

Assertions

Want to express more general properties of programs, like

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

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

Example 9. 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

- An assertion P can be viewed as a *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)
true	All states
false	No state

Example 10.

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

Then this must also hold for particular values of x and y satisfying the initial assertion, like $x = y = 5$

Formal analysis of programs

- Establish program properties, 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 tool

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:

- A set of *symbols* (constants, variables,...)
- A set of *formulas* (meaningful combination of symbols)
- A set of *axioms* (assumed to be true)
- A set of *inference rules* of the form:

  Inference rule

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

- Where each H_i is an *assumption*, and C is the *conclusion*
- The conclusion is true if all the assumptions are true
- The inference rules specify how to derive additional true formulas from axioms and other true formulas.

Symbols

- (program + extra) variables: $x, y, z, ...$
- Relation symbols: $\leq, \geq, \dots$
- Function symbols: $+, -, \dots$, and constants $0, 1, 2, \dots, \mathit{true}, \mathit{false}$
- 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:[32]

$\forall x : A(x)$	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

Typical axioms:

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

Typical rules:

$$\frac{A \qquad B}{A \wedge B}\ \textsc{And-I} \qquad \frac{A}{A \vee B}\ \textsc{Or-I} \qquad \frac{A \Rightarrow B \qquad A}{B}\ \textsc{Or-E}$$

Example 11.

$$\frac{x = 5 \qquad y = 5}{x = 5 \wedge y = 5}\ \textsc{And-I} \qquad \frac{x = 5}{x = 5 \vee y = 5}\ \textsc{Or-I}$$

$$\frac{x \geq 0 \Rightarrow y \geq 0 \qquad x \geq 0}{y \geq 0}\ \textsc{Or-E}$$

---

[32] $A(x)$ to indicate that, here, A (typically) contains x.

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 ...: 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 12.

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

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 amathematical argument that the program satisfies its specification

Interpretation

Interpretation ("semantics") of triples is related to code execution

Partial correctness interpretation

$\{P\}\ S\ \{Q\}$ is *true*/holds, if the following is the case:

- If the initial state of S satisfies P (P holds for the initial state of S),
- and if[33] S *terminates,*
- *then* Q is *true* in the final state of S

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

Example 13. $\{x = y\}\ \texttt{x} := \texttt{x} * 2; \texttt{y} := \texttt{y} * 2;\ \{x = y\}$ is *true* if the initial state satisfies $x = y$ and, in case the execution terminates, then the final state satisfies $x = y$

[33] Thus: if S does not terminate, all bets are off...

Examples

Some true formulas:

$$\{\ x = 0\ \}\ x := x + 1;\ \{\ x = 1\ \}$$
$$\{\ x = 4\ \}\ x := 5;\ \{\ x = 5\ \}$$
$$\{\ \texttt{true}\ \}\ x := 5;\ \{\ x = 5\ \}$$
$$\{\ y = 4\ \}\ x := 5;\ \{\ y = 4\ \}$$
$$\{\ x = 4\ \}\ x := x + 1;\ \{\ x = 5\ \}$$
$$\{\ x = a \wedge y = b\ \}\ x = x + y;\ \{\ x = a + b \wedge y = b\ \}$$
$$\{\ x = 4 \wedge y = 7\ \}\ x := x + 1;\ \{\ x = 5 \wedge y = 7\ \}$$
$$\{\ x = y\ \}\ x := x + 1; y := y + 1;\ \{\ x = y\ \}$$

Some formulas that are not *true*:

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

Partial correctness

- The interpretation of $\{\ P\ \}\ S\ \{\ Q\ \}$ assumes/ignores termination of S, termination is not proven.
- The assertions (P, Q) express *safety* properties
- The pre- and postconditions *restrict* possible states

The 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?

$\{\ P\ \}\ S\ \{\ \mathit{false}\ \}$	S does not terminate
$\{\ P\ \}\ S\ \{\ \mathit{true}\ \}$	trivially true
$\{\ \mathit{true}\ \}\ S\ \{\ Q\ \}$	Q holds after S in any case (provided S terminates)
$\{\ \mathit{false}\ \}\ S\ \{\ Q\ \}$	trivially true

Proof system PL

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

- Axioms for basic statements:
  - $x := e$, `skip`,...
- Rules for composed statements:
  - S_1;S_2, `if`, `while`, `await`, `co`...`oc`, ...

Formulas in PL

- formulas = triples
- theorems = derivable formulas
- hopefully: all derivable formulas are also "really" (= semantically) true
- derivation: starting from axioms, using derivation rules
- $$\frac{H_1 \quad H_2 \quad \ldots \quad H_n}{C}$$
- axioms: can be seen as rules without premises

Soundness

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

- 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: All theorems in PL are true

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

(Textual) substitution

$P_{x \leftarrow e}$ means, all free occurrences of x in P are replaced by expression e.

Example 14.

$$\begin{array}{lcl} (x = 1)_{x \leftarrow (x+1)} & \Leftrightarrow & x + 1 = 1 \\ (x + y = a)_{y \leftarrow (y+x)} & \Leftrightarrow & x + (y + x) = a \\ (y = a)_{x \leftarrow (x+y)} & \Leftrightarrow & y = a \end{array}$$

Substitution propagates into formulas:

$$\begin{array}{lcl} (\neg A)_{x \leftarrow e} & \Leftrightarrow & \neg (A_{x \leftarrow e}) \\ (A \wedge B)_{x \leftarrow e} & \Leftrightarrow & A_{x \leftarrow e} \wedge B_{x \leftarrow e} \\ (A \vee B)_{x \leftarrow e} & \Leftrightarrow & A_{x \leftarrow e} \vee B_{x \leftarrow e} \end{array}$$

Remark on textual substitution

$P_{x \leftarrow e}$

- 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 15 (Substitution).

$$\begin{array}{lcl} (\exists y : x + y > 0)_{x \leftarrow 1} & \Leftrightarrow & \exists y : 1 + y > 0 \\ (\exists x : x + y > 0)_{x \leftarrow 1} & \Leftrightarrow & \exists x : x + y > 0 \\ (\exists x : x + y > 0)_{y \leftarrow x} & \Leftrightarrow & \exists z : z + x > 0 \end{array}$$

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:

$$\begin{array}{rcl} \{ e = 5 \} & \texttt{x := e} & \{ x = 5 \} \\ \{ (x = 5)_{x \leftarrow e} \} & \texttt{x := e} & \{ x = 5 \} \end{array}$$

Axiom of assignment

"Backwards reasoning:" Given a postcondition, we may construct the precondition:

Axiom for the assignment statement

$$\{ P_{x \leftarrow e} \}\ x := e\ \{ P \} \quad \textsc{Assign}$$

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 $Q_{x \leftarrow e}$

$$\frac{P \Rightarrow Q_{x \leftarrow e} \qquad \{ Q_{x \leftarrow e} \}\ x := e\ \{ Q \}}{\{ P \}\ x := e\ \{ Q \}}$$

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

- $Q_{x \leftarrow e}$ 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:

$$\{ P \}\ \texttt{skip}\ \{ P \} \quad \textsc{Skip}$$

PL inference rules

$$\frac{\{ P \}\ S_1\ \{ R \} \qquad \{ R \}\ S_2\ \{ Q \}}{\{ P \}\ S_1; S_2\ \{ Q \}}\ \textsc{Seq}$$

$$\frac{\{ P \wedge B \}\ S\ \{ Q \} \qquad P \wedge \neg B \Rightarrow Q}{\{ P \}\ \texttt{if}\ B\ \texttt{then}\ S\ \{ Q \}}\ \textsc{Cond}'$$

$$\frac{\{ I \wedge B \}\ S\ \{ I \}}{\{ I \}\ \texttt{while}\ B\ \texttt{do}\ S\ \{ I \wedge \neg B \}}\ \textsc{While}$$

$$\frac{\{ P \}\ S\ \{ Q \} \qquad P' \Rightarrow P \qquad Q \Rightarrow Q'}{\{ P' \}\ S\ \{ Q' \}}\ \textsc{Consequence}$$

- 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:

$$\frac{\dfrac{x = y \Rightarrow 2 * x = 2 * y}{\{\ x = y\ \}\ x := x * 2\ \{\ x = 2 * y\ \}} \qquad \{\ (x = y)_{y \leftarrow 2y}\ \}\ y := y * 2\ \{\ x = y\ \}}{\{\ x = y\ \}\ x := x * 2; y := y * 2\ \{\ x = y\ \}}$$

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

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

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 16.

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

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:

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

$$\frac{\{P \wedge B\}\ S\ \{Q\} \qquad (P \wedge \neg B) \Rightarrow Q}{\{\ P\ \}\ \texttt{if}\ B\ \texttt{then}\ S\ \{\ Q\ \}}\ \textsc{Cond}'$$

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

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

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

3.10.2014

6 Program Analysis

Program Logic (PL)

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

  "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

$$\frac{\{\ P\ \}\ S_1\ \{\ R\ \} \qquad \{\ R\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ S_1; S_2\ \{\ Q\ \}}\ \textsc{Seq}$$

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ Q\ \} \qquad P \wedge \neg B \Rightarrow Q}{\{\ P\ \}\ \texttt{if}\ B\ \texttt{then}\ S\ \{\ Q\ \}}\ \textsc{Cond}'$$

$$\frac{\{\ I \wedge B\ \}\ S\ \{\ I\ \}}{\{\ I\ \}\ \texttt{while}\ B\ \texttt{do}\ S\ \{\ I \wedge \neg B\ \}}\ \textsc{While}$$

$$\frac{\{\ P\ \}\ S\ \{\ Q\ \} \qquad P' \Rightarrow P \qquad Q \Rightarrow Q'}{\{\ P'\ \}\ S\ \{\ Q'\ \}}\ \textsc{Consequence}$$

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

    $$\{\ y \geq 0\ \}\ \texttt{while}\ (y > 0)\ y := y - 1$$

  - 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
  - This course is *not* an exercise in finding complicated invariants

While rule

$$\frac{\{\, I \wedge B \,\}\ S\ \{\, I \,\}}{\{\, I \,\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{\, I \wedge \neg B \,\}}\ \textsc{While}$$

Can use this rule to reason about the more general case:

$$\{\, P \,\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{\, Q \,\}$$

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

$$\{\, 0 \le n \,\}\ k := 0;\ \{\, k \le n \,\}\ \texttt{while}\ \ (k < n)\ k := k + 1;\ \{\, k = n \,\}$$

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

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

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

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

Await statement

Rule for await

$$\frac{\{\, P \wedge B \,\}\ S\ \{\, Q \,\}}{\{\, P \,\}\ \langle \texttt{await}(B)\ S \rangle\ \{\, Q \,\}}\ \textsc{Await}$$

Remember: we are reasoning about safety properties

- Termination is assumed/ignored
- the rule does not speak about waiting or 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 "⟨atomic brackets⟩".
First attempt for a co . . . oc rule in PL:

$$\frac{\{\, P_1 \,\}\ \langle S_1 \rangle\ \{\, Q_1 \,\} \qquad \{\, P_2 \,\}\ \langle S_2 \rangle\ \{\, Q_2 \,\}}{\{\, P_1 \wedge P_2 \,\}\ \texttt{co}\langle S_1 \rangle \parallel \langle S_2 \rangle\ \texttt{oc}\ \{\, Q_1 \wedge Q_2 \,\}}\ \textsc{Par}$$

Example 17 (Problem with this rule).

$$\frac{\{\, x = 0 \,\}\ \langle x := x + 1 \rangle\ \{\, x = 1 \,\} \qquad \{\, x = 0 \,\}\ \langle x := x + 2 \rangle\ \{\, x = 2 \,\}}{\{\, x = 0 \,\}\ \texttt{co}\langle x := x + 1 \rangle \parallel \langle x = x + 2 \rangle\ \texttt{oc}\ \{\, x = 1 \wedge x = 2 \,\}}$$

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

1>The 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 precodition in this line of reasoning is not problematic. Note however, that conceptially 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 ∧ I win 100 dollar". The rule, if it were true, wo be nice: compositionality. For indepdented 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 absense of such problem will later be called interference free. It will not be defined for processes, but for specifications insofar: interference free if the pre- and post-conditions of parallel processes are not disturbed.

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

Now we can try to apply the rule:

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

where:

$$\begin{array}{rl} PRE & : (x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1) \\ POST & : (x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)\end{array}$$

which gives:

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

Concurrent execution

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

$$\frac{\{\ P_i\ \}\ S_i\ \{\ Q_i\ \} \quad \text{are } \textit{interference free}}{\{\ P_1 \wedge \ldots \wedge P_n\ \}\ \texttt{co}\, S_1 \parallel \ldots \parallel S_n\, \texttt{oc}\ \{\ Q_1 \wedge \ldots \wedge Q_n\ \}}\ \textsc{Cooc}$$

Interference freedom

A process interferes with (the specification of) another process, if its execution changes the values of the assertions[34] of the other process.

[34] Only "critical assertions" considered

- assertions inside awaits: not endagered
- critical assertions or critical conditions: assertions outside await statement bodies.[35]

Interference freedom

Interference freedom

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

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

is derivable in PL (= theorem).

"C is *invariant* under the execution of the other process"

$$\frac{\{\ P_1\ \}\ S_1\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ S_2\ \{\ Q_2\ \}}{\{\ P_1 \wedge P_2\ \}\ \mathsf{co}\ S_1 \parallel S_2\ \mathsf{oc}\ \{\ Q_1 \wedge Q_2\ \}}$$

Four interference freedom requirements:

$$\begin{array}{ll} \{P_2 \wedge P_1\}\ S_1\ \{P_2\} & \{P_1 \wedge P_2\}\ S_2\ \{P_1\} \\ \{Q_2 \wedge P_1\}\ S_1\ \{Q_2\} & \{Q_1 \wedge P_2\}\ S_2\ \{Q_1\} \end{array}$$

"Avoiding" interference: Weakening assertions

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

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:

$$\begin{array}{ll} S_1: & \{\ x = 0 \vee x = 2\ \}\ \langle x := x + 1\rangle\ \{\ x = 1 \vee x = 3\ \} \\ S_2: & \{\ x = 0 \vee x = 1\ \}\ \langle x := x + 2\rangle\ \{\ x = 2 \vee x = 3\ \} \end{array}$$

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

(Correspondingly for the other three critical conditions)

Avoiding interference: Disjoint variables

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

[35] More generally one could say: outside mutex-protected sections.

Avoiding interference: Global invariants

global invariants

- Some conditions. that only refer to global (shared) variables
- Holds initially
- Preserved by all assignments

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

$$\texttt{co}\ldots;S;\ldots \parallel \ldots;S_1;\{\ C\ \}S_2;\ldots\texttt{oc}$$

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

$$\texttt{co}\ldots;S;\ldots \parallel \ldots;\langle S_1;\{\ C\ \}S_2\rangle;\ldots\texttt{oc}$$

Example: Producer/ consumer synchronization

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

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

Let PC be a *global invariant* of the program:

```
1
2                int buf, p := 0; c := 0;
3
4  process Producer {                  process Consumer {
5    int a[N];...                      int b[N];...
6    while (p < N) {                   while (c < N) {
7      < await (p = c) ; >              < await (p > c) ; >
8      buf := a[p];                      b[c] := buf;
9        p := p+1;                         c := c+1;
10   }                                   }
11 }                                 }
```

Example: Producer

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

```
process Producer dir0o
  int a[n];
  { I_P dir }                    //  entering loop
  while (p < n) dir0o            { I_P ∧ p < n }
    < await (p == c); >          { I_P ∧ p < n ∧ p = c }
                                 { I_{P p←p+1 buf←a[p]} }
    buf = a[p];                  { I_{P p←p+1} }
    p = p + 1;                   dir0oI_P dir
  dir dir0oI_P ∧ ¬(p < n)dir     //  exit loop
     ⇔ dir0oPC ∧ p = ndir
dir
```

Proof obligation: $\{\ I_P \wedge p < n \wedge p = c\ \} \Rightarrow \{\ I_P\ \}_{p \leftarrow p+1 buf \leftarrow a[p]}$

Example: Consumer

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

```
process Consumer dir0o
  int b[n];
  dir0oI_C dir                    // entering loop
  while (c < n) dir0o             dir0oI_C ∧ c < n dir
    < await (p > c) ; >           dir0oI_C ∧ c < n ∧ p > c dir
                                  dir0oI_C dir_{c←c+1, b[c]←buf}
    b[c] = buf;                   dir0oI_C dir_{c←c+1}
    c = c + 1;                    dir0oI_C dir
  dir dir0oI_C ∧ ¬(c < n) dir     // exit loop
      ⇔ dir0oPC ∧ c = n ∧ b[0:c-1] = a[0:c-1] dir
dir
```

Proof Obligation: $dir0oI_C \wedge c < n \wedge p > c dir \Rightarrow dir0oI_C dir_{c \leftarrow c+1, b[c] \leftarrow buf}$

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
If we combine 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 dir0o
   monitor variables      # shared global variable
   initialization         # for the monitor's procedures
   procedures
dir
```

- A monitor invariant (I): used to 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 *doe not* mention `cv`. Then we can set up the following axioms:

$$\{ I \}\ wait(cv)\ \{ I \}$$
$$\{ P \}\ signal(cv)\ \{ P \} \qquad \text{for arbitrary } P$$
$$\{ P \}\ signal\_all(cv)\ \{ P \} \qquad \text{for arbitrary } P$$

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←nr+1} }
        nr = nr + 1;
        { I }
}
```

$(I \wedge nw > 0) \Rightarrow I$ $(I \wedge nw = 0) \Rightarrow I_{nr \leftarrow nr+1}$ 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:

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

by assignment axiom:

$$wait(cv) : \{I_{\#cv \leftarrow \#cv+1}; \#cv := \#cv + 1; \{\ I\ \}\ \text{“}sleep''\{\ I\ \}$$

Axioms for Signal and Continue (3)

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

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

$$\begin{aligned} \texttt{signal}(cv) : \quad & \{((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P_{\#cv \leftarrow \#cv-1}\} \\ & if\ (\#cv \neq 0)\ \#cv := \#cv - 1 \\ & \{P\} \end{aligned}$$

- $signal\_all(cv)$: $\{\ P_{\#cv \leftarrow 0}\ \}\ \#cv := 0\ \{P\}$

Axioms for Signal and Continue (4)

Together this gives:

Axioms for monitor communication

$\{\ I_{\#cv \leftarrow (\#cv+1)}\ \}$ `wait`(cv) $\{\ I\ \}$ WAIT

$\{\ ((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P_{\#cv \leftarrow (\#cv-1)})\ \}$ `signal`(cv) $\{\ P\ \}$ SIGNAL

$\{\ P_{\#cv \leftarrow 0}\ \}$ `signal_all`(cv) $\{\ P\ \}$ SIGNALALL

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

$$\{\ P_{\#cv \leftarrow (\#cv-1)}\ \}\ \texttt{signal}(cv)\ \{\ P\ \}$$

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

Example: FIFO semaphore verification (1)

```
1  monitor Semaphore_fifo { # monitor invariant: s ≥ 0
2    int s := 0;            # value of the semaphore
3    cond pos;              # wait condition
4
5    procedure Psem() {
6      if   (s=0)
7           wait (pos);
8      else
9           s := s - 1
10   }
11
12
13   procedure Vsem() {
14     if   empty(pos)
15          s := s + 1
16     else
17          signal(pos);
18   }
19 }
```

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-semaphore even under the weak s&c signalling discipline.

Example: FIFO semaphore verification: Psem

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

```
procedure Psem() {
{I}
   if (s=0) {I ∧ s = 0}
       {I_{#pos ←(#pos+1)}} wait(pos); {I}
   else {I ∧ s ≠ 0}
      {I_{s ←(s−1)}} s := s-1; {I}
{I}
}
```

Example: FIFO semaphore verification (3)

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

This gives two proof obligations: If-branch:

$$\begin{array}{lcl} (I \wedge s = 0) & \Rightarrow & I_{\#pos \leftarrow (\#pos+1)} \\ s = 0 & \Rightarrow & s \geq 0 \wedge (s > 0 \Rightarrow \#pos + 1 = 0) \\ s = 0 & \Rightarrow & s \geq 0 \end{array}$$

Else branch:

$$\begin{array}{lcl} (I \wedge s \neq 0) & \Rightarrow & I_{s \leftarrow (s-1)} \\ (s > 0 \wedge \#pos = 0) & \Rightarrow & s - 1 \geq 0 \wedge (s - 1 \geq 0 \Rightarrow \#pos = 0) \\ (s > 0 \wedge \#pos = 0) & \Rightarrow & s > 0 \wedge \#pos = 0 \end{array}$$

Example: FIFO semaphore verification: Vsem

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

```
procedure Vsem() dir0o
{I}
   if empty(pos) {I ∧ #pos = 0}
       {I_{s ←(s+1)}}s:=s+1; {I}
   else {I ∧ #pos ≠ 0}
      {I_{#pos ←(#pos−1)}} signal(pos); {I}
{I}
dir
```

Example: FIFO semaphore verification (5)

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

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

$$\begin{array}{lcl} (I \wedge \#pos = 0) & \Rightarrow & I_{s \leftarrow (s+1)} \\ (s \geq 0 \wedge \#pos = 0) & \Rightarrow & s + 1 \geq 0 \wedge (s + 1 > 0 \Rightarrow \#pos = 0) \\ (s \geq 0 \wedge \#pos = 0) & \Rightarrow & s + 1 \geq 0 \wedge \#pos = 0 \end{array}$$

Else branch:

$$\begin{array}{lcl} (I \wedge \#pos \neq 0) & \Rightarrow & I_{\#pos \leftarrow (\#pos-1)} \\ (s = 0 \wedge \#pos \neq 0) & \Rightarrow & s \geq 0 \wedge (s > 0 \Rightarrow \#pos - 1 = 0) \\ s = 0 & \Rightarrow & s \geq 0 \end{array}$$

7 Java concurrency

10. 10. 2014

7.1 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 a 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*.

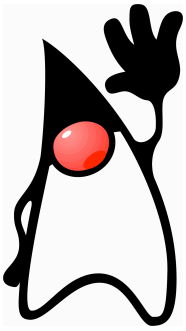
  Signaling disciplines

  - 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:[36]

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



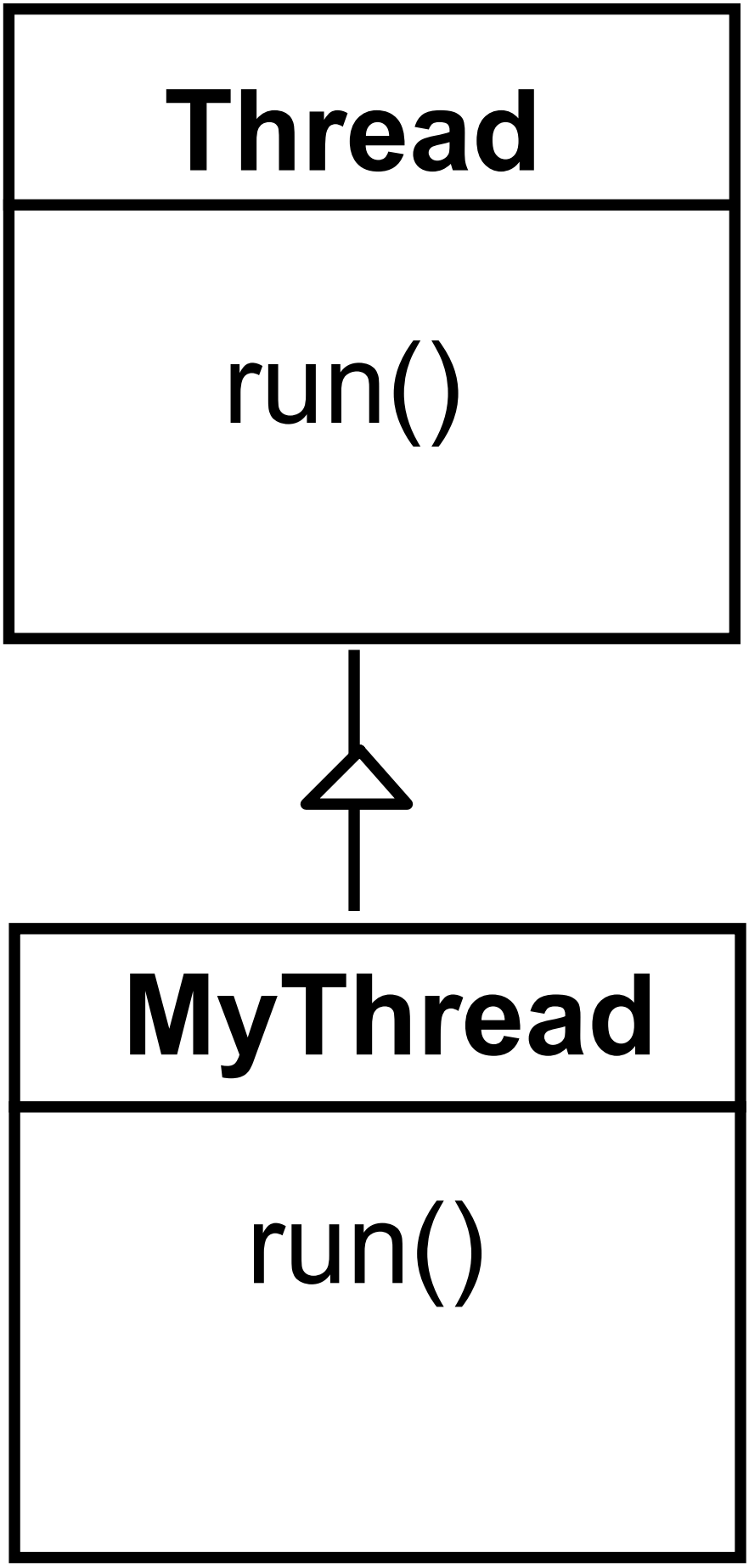
Threads in Java

A thread in Java

- unit of concurrency[37]
- identity, accessible via static method `Thread.CurrentThread()`[38]
- 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

Thread class

[36] But it's correct nonetheless ...
[37] as such, roughly corresponding to the concept of "processes" from previous lecctures.
[38] What's the difference to `this`?

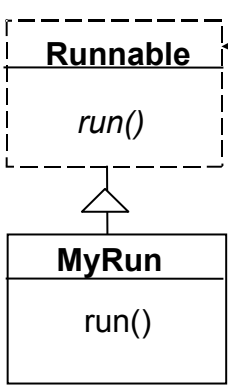


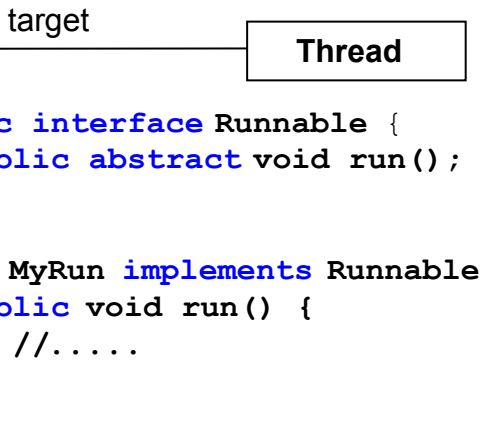
The `Thread` class executes instructions from its method `run()`. The actual code executed depends on the implementation provided for `run()` in a derived class.

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

Runnable interface

As Java does not support multiple inheritance, we 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 in Java 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 class[39]
3. create an instance of the new class.
4. *start* the thread.

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 ⇒ may lose updates to data.

[39] overriding, late-binding.

Synchronization

avoid interference ⇒ 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.[40]
3. 2 "flavors"

"synchronized block"

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

synchronized method

whole method body of m "protected"[41]:

```
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:

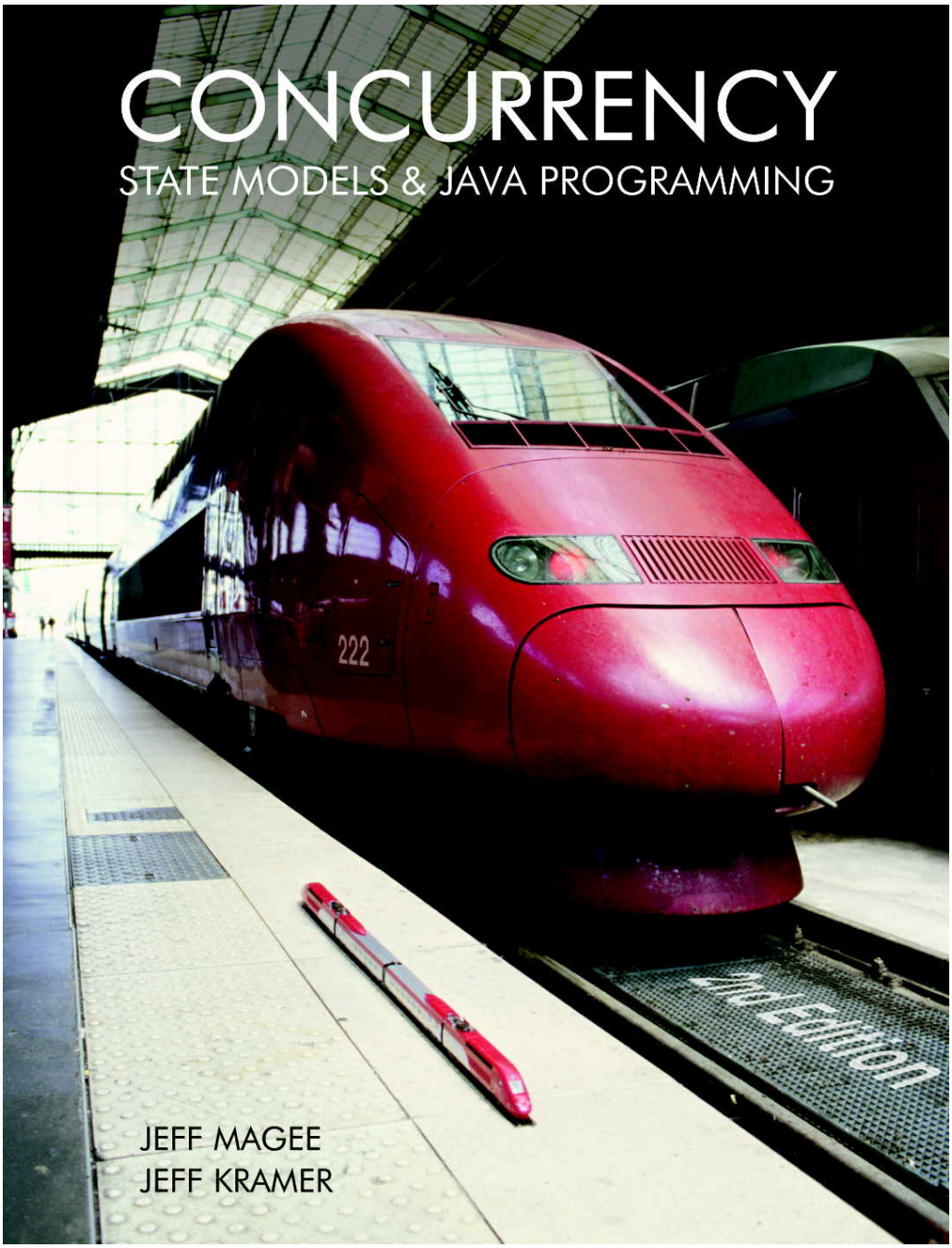
Concurrency: State Models & Java Programs, 2^{nd} Edition

Jeff Magee & Jeff Kramer

Wiley

[40] but: in a re-entrant manner!

[41] assuming that other methods play according to the rules as well etc.

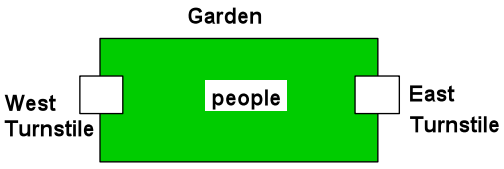


Examples in Java:
`http://www.doc.ic.ac.uk/~jnm/book/`

7.2 Ornamental garden

Ornamental garden problem

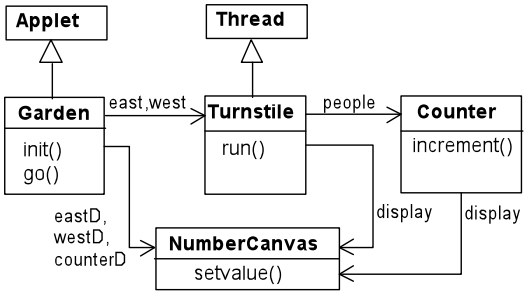
- people enter an ornamental garden through either of 2 turnstiles.
- problem: the number of people present at any time.



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

```
1
2  class Counter {
3
4     int value = 0;
5     NumberCanvas display;
6
7     Counter(NumberCanvas n) {
8       display = n;
9       display.setvalue(value);
10     }
11
12     void increment() {
13       int temp = value;              // read[v]
14       Simulate.HWinterrupt();
15       value = temp + 1;              // write[v+1]
16            display.setvalue(value);
17       }
18  }
```

Turnstile

```
1
2  class Turnstile extends Thread {
3    NumberCanvas display;    // interface
4    Counter people;          // shared data
5
6    Turnstile(NumberCanvas n, Counter c) { // constructor
7       display = n;
8       people = c;
9    }
10
11   public void run() {
12      try {
13        display.setvalue(0);
14        for (int i = 1; i <= Garden.MAX; i++) {
15          Thread.sleep(500); // 0.5 second
16          display.setvalue(i);
17          people.increment();  // increment the counter
18        }
19      } catch (InterruptedException e) { }
20    }
21 }
```

Ornamental Garden Program

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

```
1  private void go() {
2    counter = new Counter(counterD);
3    west = new Turnstile(westD,counter);
4    east = new Turnstile(eastD,counter);
5    west.start();
6    east.start();
7  }
```

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?

Avoid interference by synchronization

```
1
2  class SynchronizedCounter extends Counter {
3
4    SynchronizedCounter(NumberCanvas n) {
5      super(n);
6    }
7
8    synchronized void increment() {
9      super.increment();
10   }
11 }
```

Mutual Exclusion: The Ornamental Garden - DEMO

DEMO

7.3 Thread communication, monitors, and signaling

Monitors

- *each* object
  - has attached to it a unique *lock*
  - and thus: can act as monitor
- 3 important monitor operations[42]
  - *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*

[42] there are more

- 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-identity[43]

⇒
```
1   Thread t = new MyThread();
2   ...
3   t.notify();;      // mostly to be nonsense
```

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 ≠ instance private
  - not all methods need to be synchronized[44]
  - besides that: there's re-entrance!

A semaphore implementation in Java

```
1   // down() = P operation
2   // up()   = V operation
3
4   public class Semaphore {
5       private int value;
6
7       public Semaphore (int initial) {
8           value = initial;
9       }
10
11      synchronized public void up() {
12          ++value;
13          notifyAll();}
14
15      synchronized public void down() throws InterruptedException {
16          while (value==0) wait(); // the well-known while-cond-wait pattern
17          --value;}
18  }
```

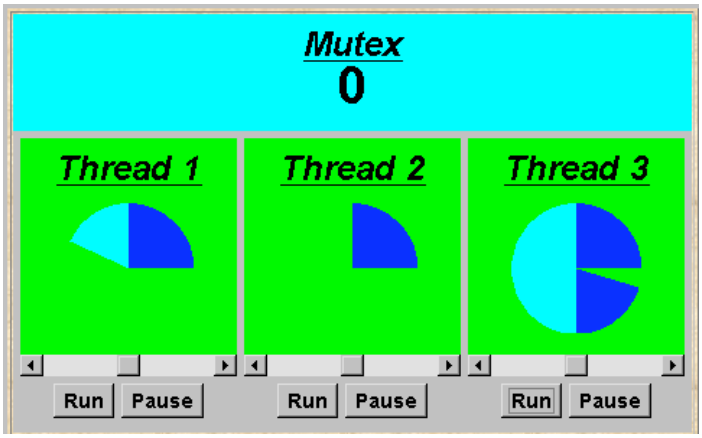
- cf. also `java.util.concurrency.Semaphore` (acquire/release + more methods)

7.4 Semaphores

Mutual exclusion with sempahores

[43] technically, a thread identity is represented by a "thread object" though. Note also : `Thread.suspend()` and `Thread.resume()` are deprecated.

[44] remember: `find` of oblig-1.



Mutual exclusion with sempahores

```
1
2  class MutexLoop implements Runnable {
3
4      Semaphore mutex;
5
6      MutexLoop (Semaphore sema) {mutex=sema;}
7
8      public void run() {
9        try {
10         while(true)  {
11           while(!ThreadPanel.rotate());
12            // get mutual exclusion
13            mutex.down();
14            while(ThreadPanel.rotate()); //critical section
15            //release mutual exclusion
16            mutex.up();
17         }
18       } catch(InterruptedException e){}
19     }
20 }
```

DEMO

7.5 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

```
1
2  interface ReadWrite {
3
4     public void acquireRead() throws InterruptedException;
5
6     public void releaseRead();
```

```
7
8     public void acquireWrite() throws InterruptedException;
9
10    public void releaseWrite();
11  }
```

Reader client code

```
1
2   class Reader implements Runnable {
3
4     ReadWrite monitor_;
5
6     Reader(ReadWrite monitor) {
7       monitor_ = monitor;
8      }
9
10     public void run() {
11       try {
12         while(true)  {
13              while(!ThreadPanel.rotate());
14              // begin critical section
15              monitor_.acquireRead();
16              while(ThreadPanel.rotate());
17              monitor_.releaseRead();
18         }
19       } catch (InterruptedException e){}
20     }
21  }
```

Writer client code

```
1
2   class Writer implements Runnable {
3
4     ReadWrite monitor_;
5
6     Writer(ReadWrite monitor) {
7       monitor_ = monitor;
8      }
9
10     public void run() {
11       try {
12         while(true)  {
13              while(!ThreadPanel.rotate());
14              // begin critical section
15              monitor_.acquireWrite();
16              while(ThreadPanel.rotate());
17              monitor_.releaseWrite();
18         }
19       } catch (InterruptedException e){}
20     }
21  }
```

R/W monitor (regulate readers)

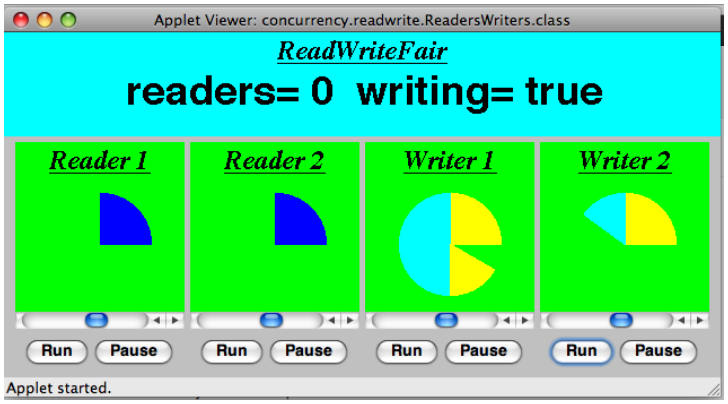
```
1
2   class ReadWriteSafe implements ReadWrite {
3     private int readers =0;
4     private boolean writing = false;
5
6     public synchronized void acquireRead()
7                 throws InterruptedException {
8       while (writing) wait();
9       ++readers;
10    }
11
12    public synchronized void releaseRead() {
13      --readers;
14      if(readers==0) notifyAll();
15    }
16
17    public synchronized void acquireWrite() {...}
18
19    public synchronized void releaseWrite() {...}
20  }
```

R/W monitor (regulate writers)

```
1
2  class ReadWriteSafe implements ReadWrite {
3    private int readers =0;
4    private boolean writing = false;
5
6    public synchronized void acquireRead() {...}
7
8    public synchronized void releaseRead() {...}
9
10   public synchronized void acquireWrite()
11               throws InterruptedException {
12     while (readers>0 || writing) wait();
13     writing = true;
14   }
15
16   public synchronized void releaseWrite() {
17     writing = false;
18     notifyAll();
19   }
20 }
```

DEMO

Fairness



"Fairness": regulating readers

```
1
2  class ReadWriteFair implements ReadWrite {
3
4      private int readers =0;
5      private boolean writing = false;
6      private int waitingW = 0; // no of waiting Writers.
7      private boolean readersturn = false;
8
9      synchronized public void acquireRead()
10     throws InterruptedException {
11         while (writing || (waitingW>0 && !readersturn)) wait();
12         ++readers;
13       }
14
15     synchronized public void releaseRead() {
16         --readers;
17         readersturn=false;
18         if(readers==0) notifyAll();
19     }
20
21     synchronized public void acquireWrite() {...}
22     synchronized public void releaseWrite() {...}
23  }
```

"Fairness": regulating writers

```
1
2  class ReadWriteFair implements ReadWrite {
3
4      private int readers =0;
5      private boolean writing = false;
6      private int waitingW = 0; // no of waiting Writers.
7      private boolean readersturn = false;
8
9      synchronized public void acquireRead() {...}
```

```
10      synchronized public void releaseRead() {...}
11
12      synchronized public void acquireWrite()
13      throws InterruptedException {
14          ++waitingW;
15          while (readers>0 || writing) wait();
16          --waitingW; writing = true;
17        }
18
19      synchronized public void releaseWrite() {
20          writing = false; readersturn=true;
21          notifyAll();
22      }
23  }
```

Readers and Writers problem

DEMO

Java concurrency

- there's (much) more to it than what we discussed (synchronization, monitors) (see `java.util.concurrency`)
- Java's memory model: since Java 1: loooong, 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,[45] 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]

8 Message passing and channels

17. Oct. 2014

[45] and pointer aliasing and a weak memory model makes it worse.

8.1 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 programming[46]
- Asynchronous message passing: channels, messages, primitives
- 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

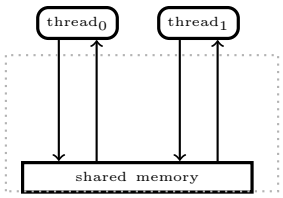
more traditional system architectures have one shared memory:

- 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 ...

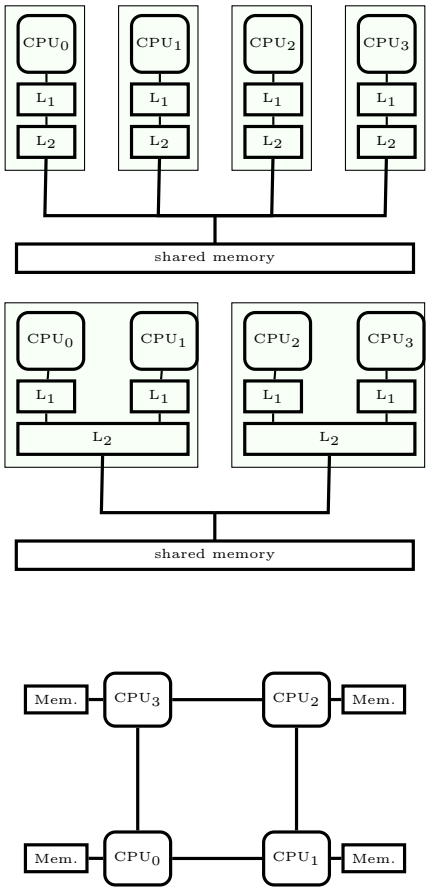
Shared memory concurrency in the real world



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

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

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 ⇒ processes cannot share variables (directly)
- Processes communicate by sending and receiving *messages* via shared *channels*

  or (in future lectures): communication via *RPC* and *rendezvous*

8.2 Asynch. message passing

Asynchronous message passing: channel abstraction

Channel: abstraction, e.g., of a physical communication network[47]

- 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, ...

[47] but remember also: producer-consumer problem

Asynchronous message passing: primitives

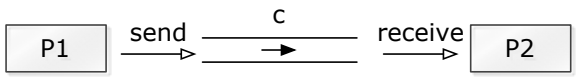
Channel declaration

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

Messages: n-tuples of values of the respective types

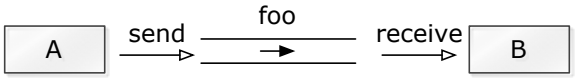
communication primitives:

- send $c(\text{expr}_1, \ldots, \text{expr}_n)$; Non-blocking, i.e. asynchronous
- receive $c(\text{var}_1, \ldots, \text{var}_n)$; Blocking: receiver waits until message is sent on the channel
- empty (c); True if channel is empty



Example: message passing

(x,y) = (1,2)

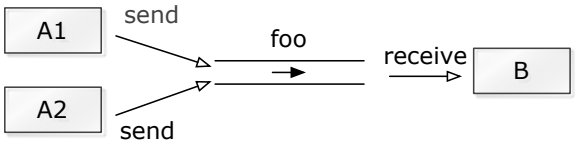



```
1  chan foo(int);
2
3  process A {
4    send foo(1);
5    send foo(2);
6  }
```

```
7
8  process B {
9    receive foo(x);
10   receive foo(y);
11 }
```

Example: shared channel

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




```
1  process A1 {
2    send foo(1);
3  }
4
```

```
5  process A2 {
6    send foo(2);
7  }
8
9  process B {
```

```
10   receive foo(x);
11   receive foo(y);
12 }
```

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

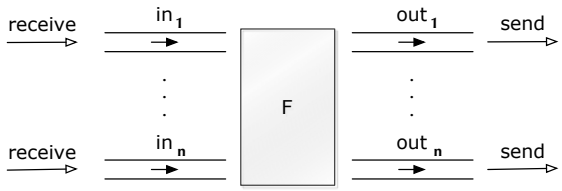
(Ignores content of messages)

8.2.1 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 can naturally be 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

$\forall i : 1 \leq i < n.\ \big(sent[i] \leq sent[i+1]\big)\ \wedge$ 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 $\mathbf{in}_1$ and $\mathbf{in}_2$ and output channel **out**:

```
1  in1: 1 4 9 ...
2                          out: 1 2 4 5 8 9 ...
3  in2: 2 5 8 ...
```

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*:

$\mathbf{in}_1$ and $\mathbf{in}_2$ are empty $\wedge$ $sent[n+1] = \mathbf{EOS}\ \wedge\ \ \forall i : 1 \leq i < n\big(sent[i] \leq sent[i+1]\big)\ \wedge$ values sent to **out** are a *permutation* of values from $\mathbf{in}_1$ and $\mathbf{in}_2$

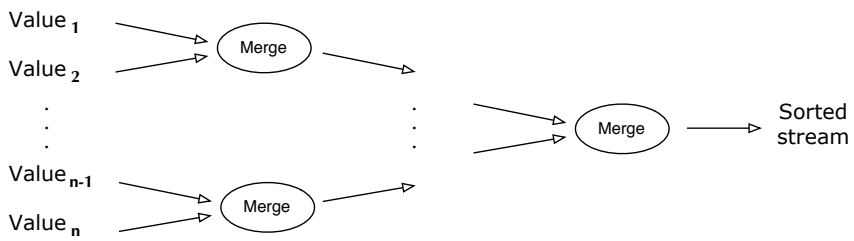
Example: Merge process

```
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 ≠ EOS and v2 ≠ EOS) {
    if (v1 ≤ v2)
      { send out(v1); receive in1(v1); }
    else                          # (v1 > v2)
      { send out(v2); receive in2(v2); }
  }

                                  # consume the rest
                                  # of the non-empty input channel
  while (v2 ≠ EOS)
    { send out(v2); receive in2(v2); }
  while (v1 ≠ 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)

8.2.2 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
send request(i,args);      ⟶       receive request(id,vars);
:                          :          :
receive reply[i](vars);    ⟵       send reply[id](results);
                                    }
```

8.2.3 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.[48]

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 ⇒ simplifies later translation to a server process

Unallocated (free) units are represented as a set, type **set**, with operations **insert** and **remove**.

Recap: "semaphore monitor" with "passing the condition"

```
1  monitor Semaphore_fifo { # monitor invariant: s ≥ 0
2    int s := 0;            # value of the semaphore
3    cond pos;              # wait condition
4
5    procedure Psem() {
6      if   (s=0)
7           wait (pos);
8      else
9           s := s - 1
10   }
11
12
13   procedure Vsem() {
14     if   empty(pos)
15          s := s + 1
16     else
17          signal(pos);
18   }
19 }
```

(Fig. 5.3 in Andrews [Andrews, 2000])

Allocator as a monitor

```
1  monitor Resource_Allocator {
2    int avail := MAXUNITS;
3    set units := ... # initial values;
4    cond free;       # signalled when process wants a unit
5
6    procedure acquire(int &id) {  # var.parameter
7      if (avail = 0)
8        wait(free);
9      else
10       avail := avail-1;
11     remove(units, id);
12   }
13
14   procedure release(int id) {
15     insert(units, id);
16     if (empty(free))
17       avail := avail+1;
18     else
19       signal(free);            # passing the condition
20   }
21 }
```

([Andrews, 2000, Fig. 7.6])

[48] In practice: server may spawn local threads, one per request.

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 channel[49] ⇒ 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
   ⇒ queue of pending requests (**queue**; **insert**, **remove**).
   (c) request: "synchronous/blocking" call ⇒ "ack"-message back
   (d) no internal parallelism ⇒ mutex

1>In order to design a monitor, we may follow the following 3 "design steps" to make it more systematic: 1) Inteface, 2) "business logic" 3) sync./coordination

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);
} } } }                              # Fig. 7.7 in Andrews (rewritten)
```

[49] Alternatives exist

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

8.3 Synchronous message passing

Synchronous message passing

Primitives:

- New primitive for sending:

  synch_send c($\text{expr}_1, \ldots, \text{expr}_n$);

  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($\text{var}_1, \ldots, \text{var}_n$);

  empty(c);

Synchronous message passing: discussion

Advantages:

- Gives maximum size of channel.

  Sender synchronises with receiver ⇒ receiver has at most 1 pending message per channel per sender ⇒ sender has at most 1 unsent message

Disadvantages:

- reduced parallellism: when 2 processes communicate, 1 is always blocked.
- higher risk of deadlock.

Example: blocking with synchronous message passing

```
1  chan values(int);
2
3  process Producer {
4    int data[n];
5    for [i = 0 to n-1] {
6      ... # computation ...;
7      synch_send values(data[i]);
8  } }
9
10 process Consumer {
11   int results[n];
12   for [i = 0 to n-1] {
13     receive values(results[i]);
14     ... # computation ...;
15 } }
```

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

```
1  chan in1(int), in2(int);
2
3  process P1 {
4    int v1 = 1, v2;
5    synch_send in2(v1);
6    receive in1(v2);
7  }
8
9  process P2 {
10   int v1, v2 = 2;
11   synch_send in1(v2);
12   receive in2(v1);
13 }
```

P1 and **P2** block on **synch_send** – deadlock.
One process must be modified to do **receive** first
⇒ asymmetric solution.

With asynchronous message passing (**send**) all goes well.

9 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

9.1 Message passing (cont'd)

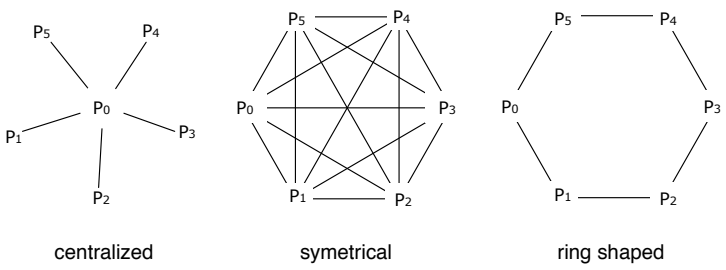
Interacting peers (processes): exchanging values example

Look at processes as peers.

Example: Exchanging values

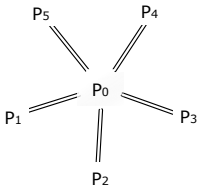
- Consider n processes P[0], ..., P[$n-1$], $n > 1$
- every process has a number, stored in local variable v
- Goal: all processes knows the largest and smallest number.
- simplistic problem, but "characteristic" of distributed computation and information distribution

Different communication patters



Centralized solution

Process P[0] is the coordinator process:



- P[0] does the calculation
- The other processes sends their values to P[0] and waits for a reply.

Number of *messages:*[50](number of send:)

P[0]: $n-1$

P[1], ..., P[$n-1$]: $(n-1)$

Total: $(n-1)+(n-1) = 2(n-1)$ messages

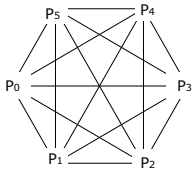
repeated "computation"

Number of *channels:* n

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)
```

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: $n(n-1)$

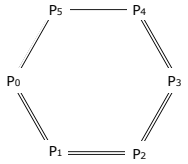
Symmetric solution: code

```
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
```

[50] For now in the pics: 1 line = 1 message (not 1 channel), but the notation in the pics is not 100% consistent.

```
12    # and store the smallest and largest.
13    for [j = 1 to n-1] { # j not used in the loop
14      receive values[i](new);
15      if (new < smallest)     smallest := new;
16      if (new > largest)      largest := new;
17    }
18  } # 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:

$$\begin{array}{rl} \text{P[0]:} & 2 \\ \text{P[1], ..., P[}n-3\text{]:} & (n-3)\times 2 \\ \text{P[}n-2\text{]:} & 1 \\ \text{P[}n-1\text{]:} & 1 \end{array}$$

$2 + 2(n-3) + 1 + 1 = 2(n-1)$ messages sent.

Number of channels: n .

Ring solution: code (1)

```
1  chan values[n](int smallest, int largest);
2
3  process P[0] { # starts the exchange
4    int v := ...;
5    int smallest := v, largest := v;
6    # send v to the next process, P[1]
7    send values[1](smallest, largest);
8    # get the global smallest and largest from P[n-1]
9    #    and send them to P[1]
10   receive values[0](smallest, largest);
11   send values[1](smallest, largest);
12 }
```

Ring solution: code (2)

```
1  process P[i = 1 to n-1] {
2    int v := ...;
3    int smallest, largest;
4    # get smallest and largest so far,
5    #    and update them by comparing them to v
6    receive values[i](smallest, largest)
7    if (v < smallest) smallest := v;
8    if (v > largest)   largest := v;
9    # forward the result, and wait for the global result
10   send values[(i+1) mod n](smallest, largest);
11   if (i < n-1)
12     receive values[i](smallest, largest);
13   # forward the global result, but not from P[n-1] to P[0]
14   if (i < n-2)
15     send values[i+1](smallest, largest);
16 } # 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.

9.2 RPC

Remote Procedure Call: main idea

```
CALLER                          CALLEE

at computer A                   at computer B
```

```
                                op foo(FORMALS); # declaration

...
call foo(ARGS);          ----->  proc foo(FORMALS)  # new process
                                   ...
                         <-----  end;
...
```

RPC (cont.)

RPC: combines elements from monitors and message passing

- As ordinary procedure call, but caller and callee may be on different machines.[51]
- 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'

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:[52]

call M.O(*arguments*)

Processes: as before.

[51] cf. RMI
[52] Cf. static/class methods

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":[53]
     - user-programmed mutex and synchronization (like semaphorse, 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)

```
1  module TimeServer
2    op get_time() returns int;
3    op delay(int interval);
4  body
5    int  tod := 0;          # time of day
6    sem    m := 1;          # for mutex
7    sem d[n] := ([n] 0);    # for delayed processes
8    queue of (int waketime, int process_id) napQ;
9    ## when m = 1, tod < waketime for delayed processes
10   proc get_time() returns time { time := tod; }
11
12   proc delay(int interval) {
13     P(m);          # assume unique myid and i [0,n-1]
14     int waketime := tod + interval;
15       insert  (waketime, myid) at appropriate place in napQ;
16     V(m);
17     P(d[myid]);  # Wait to be awoken
18   }
19   process Clock ...
     .
     .
20
21 end TimeServer
```

Time server code: clock process

```
1  process Clock {
2      int id; start hardware timer;
3      while (true) {
4        wait for interrupt, then restart hardware timer
5        tod := tod + 1;
6        P(m);                                    # mutex
7        while (tod ≥ smallest waketime on napQ) {
8          remove  (waketime, id) from napQ;      # book-keeping
9          V(d[id]);                              # awake process
10       }
11       V(m);                                    # mutex
12   } }
13 end TimeServer # Fig. 8.1 of Andrews
```

[53] assumed in the following

9.3 Rendez-vouz

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

...                             ... # existing process
call foo(ARGS);         ----->  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 Oi (expr1, ..., exprm);
```

Input statement, multiple guarded expressions:

```
    in O1(v1, ... vm1) and B1 -> S1;
    ...
    [] On(v1, ... vmn) and Bn -> Sn;
    ni
```

The guard consists of:

- and B_i – synchronization expression (optional)
- S_i – statements (one or more)

The variables $v_1, \ldots, v_{m_i}$ may be referred by B_i and S_i may read/write to them.[54]

Semantics of input statement

Consider the following:

```
in ...
[] O_i(v_i, ..., v_{m_i}) and B_i -> 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 served[55]
- 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
```

[54] once again: no side-effects in B!!!

[55] this 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
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:

$$\text{in O}(a_1, \ldots, a_n) \text{ ->}v_1{=}a_1, \ldots, v_n{=}a_n \text{ ni} \quad \Longleftrightarrow \quad \text{receive O}(v_1, \ldots, v_n)$$

Comparing message passing and semaphores:

$$\text{send O() and receive O()} \quad \Longleftrightarrow \quad \text{V(O) and P(O)}$$

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
        O1 ...              -> S1;
    [ ]
        O2 ... and (?O1 = 0) -> S2;

    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
```

10 Asynchronous Communication I

7.11.2014

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 view of components
  - describing overall understanding of a (sub)system
- Histories from the inside 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 kinds of settings for concurrent systems, based on the notion of:

- processes — without self identity, but with named channels. Channels often FIFO.
- object (agent) — 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 agent/network systems!

Programming asynchronous agent systems

New syntax statements for sending and receiving:

- *send statement:* `send` $B : m(e)$ means that the current agent sends message m to agent B where e is an (optional) list of actual parameters.
- *fixed receive statement:* **await** $B : m(w)$ wait for a message m from a specific agent B, 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, starting with receive statements.

Here m is a message name, B the name of an agent, e expressions, X and w variables.

Example: Coin machine

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
end
```

- choice operator [ ][56]
  - selects 1 *enabled* branch
  - non-deterministic choice if both branches are enabled

Interleaving semantics of concurrent systems

- behavior of a concurrent system: may be described as set of executions,
- 1 execution: sequence of atomic interaction events,
- other names for it: trace, history, execution, (interaction) sequence ...[57]

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

- very well known and widely used "format" to descibe "languages" (= sets finite "words" over given a given "alphabet")
-

A way to describe (sets of) traces

Example 18 (Reg-Expr). • a, b: atomic interactions.

- Assume them to "run" concurrently

$\Rightarrow$ two possible interleavings, described by

$$[[a.b] + [b.a]] \tag{3}$$

Parallel composition of a^* and b^*:

$$(a + b)^* \tag{4}$$

Remark: notation for reg-expr's
Different notations exist. E.g.: some write $a|b$ for the *alternative/non-deterministic* choice between a and b. We use $+$ instead

- to avoid confusion with parallel composition
- be consistent with common use of regexp. for describing concurrent behavior

Note: earlier version of this lecture used $|$.

[56] In the literature, also + as notation can often be found.
[57] message sequence (charts) in UML etc.

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 traces[58]
- *non-terminating* systems: infinite traces
- trace set semantics in the general case: both finite and infinite traces
- 2 conceptually important classes of properties[59]
  - safety ("nothing wrong will happen")
  - liveness ("something good will happen")

Safety and liveness & histories

- often: concentrate on *finite traces*
- reasons
  - conceptually/theoretically simpler
  - connection to monitoring
  - 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".

	traces	histories
a "sloppy" B may or may not give a reply, in which case there will be an infinite trace with only "hi-B" a "lazy" B will reply eventually, but there is no limit on how long A may need to wait. Thus, each trace will end with "hi_A" after finitely many "hi_B"'s. an "eager" B will reply within a fixed number of "hi_B"'s, for instance before A says "hi_B" three times.	$hi_B{}^{\infty} + hi_B{}^{+}hi_A$	$hi_B{}^{*} + hi_B{}^{+}hi_A$

- a "sloppy" B may or may not give a reply, in which case there will be an infinite trace with only "hi-B" (here comma denotes union).

  Trace set: $\{[hi_B]^{\infty}\}, \{[hi_B]^{+}\,[hi_A]\}$ Histories: $\{[hi_B]^{*}\}, \{[hi_B]^{+}\,[hi_A]\}$
- a "lazy" B will reply eventually, but there is 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: $\emptyset, \{[hi_B]\}, \{[hi_B]\,[hi_A]\}, \{[hi_B]\,[hi_B]\}, \{[hi_B]\,[hi_B]\,[hi_A]\}$

[58] Be aware: typically an *infinite* set of finite traces.

[59] Safety etc. it's not a property, it's a "property/class of properties"

Histories

Let use the following conventions

- events $x : Event$ is an event,
- set of events: $A : 2^{Event}$
- history $h : Hist$

A set of events is assumed to be fixed.

Definition 19 (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

function	type		
ϵ	$:$	$\to Hist$	the empty history (constructor)
$\_;\_$	$: Hist * Event$	$\to Hist$	append right (constructor)
$\#\_$	$: Hist$	$\to Nat$	length
$\_/\_$	$: Hist * Set$	$\to Hist$	projection by set of events
$\_ \leq \_$	$: Hist * Hist$	$\to Bool$	prefix relation
$\_ < \_$	$: Hist * Hist$	$\to Bool$	strict prefix relation

Inductive definitions (inductive wrt. ε and $\_;\_$):

$$\begin{array}{ll} \#\epsilon & = 0 \\ \#(h;x) & = \#h + 1 \\ \epsilon/A & = \epsilon \\ (h;x)/s & = \texttt{if } x \in A \texttt{ then } (h/s);x \texttt{ else } (h/s) \texttt{ fi} \\ h \leq h' & = (h = h') \vee h < h' \\ h < \varepsilon & = false \\ h < (h';x) & = h \leq h' \end{array}$$

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:

$$\{h \mid I(h)\} \tag{5}$$

- mostly interested in *prefix-closed invariants*!
- a history invariant is historically monotonic:

$$h \leq h' \Rightarrow (I(h') \Rightarrow I(h)) \tag{6}$$

- I history-monotonic $\Rightarrow$ set from equation (5) prefix closed

Remark: A non-monotonic predicate I may be transformed to a monotonic one I':

$$\begin{array}{ll} I'(\varepsilon) & = I(\varepsilon) \\ I'(h';x) & = I(h') \wedge I(h';x) \end{array}$$

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

$$\begin{array}{lll} U{\uparrow}C : five & -- & U \text{ sends the message "five" to C} \\ U{\downarrow}C : five & -- & C \text{ consumes the message "five"} \\ & & \\ U{\uparrow}C : one & -- & U \text{ sends the message "one to C} \\ U{\downarrow}C : one & -- & C \text{ consumes the message "one"} \\ & & \\ C{\uparrow}U : ten & -- & C \text{ sends the message "ten"} \\ C{\downarrow}U : ten & -- & U \text{ consumes the message "ten"} \end{array}$$

Legal histories

- note all global sequences/histories "make sense"
- depens on the programming language/communciation model
- sometimes called well-definedness, well-formedness or similar
- $legal : Hist \to Bool$

Definition 20 (Legal history)**.**

$$\begin{array}{rl} legal(\epsilon) & = true \\ legal(h; (A{\uparrow}B : m)) & = legal(h) \\ legal(h; (A{\downarrow}B : m)) & = legal(h) \wedge \\ & \quad \#(h/\{A{\downarrow}B : m\}) < \#(h/\{A{\uparrow}B : m\}) \end{array}$$

where m is message and h a history.

- should m include parameters, legality ensures that the values received are the same as those sent.

Example (coin machine C user U):

$$[(U{\uparrow}C : five), (U{\uparrow}C : five), (U{\downarrow}C : five), (U{\downarrow}C : five), (C{\uparrow}U : ten)]$$

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 l 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

Global invariant: By a predicate I on the global history, we may specify desired system behavior:

"at any point in an execution H the property $I(H)$ is satisfied"

- By logging the history at run-time, as above, we may 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 21 (Local events)**.** The events visible to an agent A, denoted α_A, are the events local to A, i.e.:

- $A{\uparrow}B : m$: any send-events from A. (output by A)
- $B{\downarrow}A : m$: any reception by A. (input by A)

Definition 22 (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.
- Each event is visible to one, and only one, agent!

Coin Machine Example: Local Events

The events visible to C are:

$U{\downarrow}C : five$	C consumes the message "five"
$U{\downarrow}C : one$	C consumes the message "one"
$C{\uparrow}U : ten$	C sends the message "ten"

The events visible to U are:

$U{\uparrow}C : five$	U sends the message "five" to C
$U{\uparrow}C : one$	U sends the message "one to C
$C{\downarrow}U : ten$	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: Sloppy coin machine

```
1  loop
2    while b < 10
3    do
4      (await U:five; b:=b+5)
5    []
6      (await U:one;  b:=b+1)
7    od;
8    send U:ten;
9    b:=b−10
10 end
```

interactions visible to C (i.e. those that may show up in the local history):

$U{\downarrow}C : five$ −− C consumes the message "five"
$U{\downarrow}C : one$ −− C consumes the message "one"
$C{\uparrow}U : ten$ −− C sends the message "ten"

Coin machine example: Loop invariants

Loop invariant for the outer loop:

$$sum(h/\downarrow) = sum(h/\uparrow) + b \ \wedge \ 0 \leq b < 5 \tag{7}$$

where sum (the sum of values in the messages) is defined as follows:

$$\begin{array}{lcl} sum(\varepsilon) & = & 0 \\ sum(h;(\ldots : five)) & = & sum(h) + 5 \\ sum(h;(\ldots : one)) & = & sum(h) + 1 \\ sum(h;(\ldots : ten)) & = & sum(h) + 10 \end{array}$$

Loop invariant for the inner loop:

$$sum(h/\downarrow) = sum(h/\uparrow) + b \ \wedge \ 0 \leq b < 15 \tag{8}$$

Histories: from inside to outside view

From local histories to global history: if we know all the local histories h_{Ai} in a system ($i = 1...n$), we have

$$legal(H) \wedge_i \ h_{A_i} = H/\alpha_{A_i}$$

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 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:

$$legal(H) \wedge_i \ I_{A_i}(H/\alpha_{A_i})$$

Coin machine example: from local to global invariant

before each send/receive: (see eq. (8))

$$sum(h/\downarrow) = sum(h/\uparrow) + b \ \wedge \ 0 \leq b < 15$$

Local Invariant of C in terms of h alone:

$$I_C(h) = \exists b.\ (sum(h/\downarrow) = sum(h/\uparrow) + b \ \wedge \ 0 \leq b < 15) \tag{9}$$

$$I_C(h) = 0 \leq sum(h/\downarrow) - sum(h/\uparrow) < 15 \tag{10}$$

For a global history H ($h = H/\alpha_C$):

$$I_C(H/\alpha_C) = 0 \leq sum(H/\alpha_C/\downarrow) - sum(H/\alpha_C/\uparrow) < 15 \tag{11}$$

Shorthand notation: $I_C(H/\alpha_C) = 0 \leq sum(H/{\downarrow}C) - sum(H/C{\uparrow}) < 15$

Coin machine example: from local to global invariant

- Local Invariant of a careful user U (with exact change):

$$I_U(h) = 0 \leq sum(h/\uparrow) - sum(h/\downarrow) \leq 10$$
$$I_U(H/\alpha_U) = 0 \leq sum(H/U\uparrow) - sum(H/\downarrow U) \leq 10$$

- Global Invariant of the system U and C:

$$I(H) = legal(H) \wedge I_C(H/\alpha_C) \wedge I_U(H/\alpha_U) \tag{12}$$

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:

$$\begin{array}{lcl}
rec(\varepsilon) & = & 0 \\
rec(h;(U\downarrow C : five)) & = & rec(h) + 5 \\
rec(h;(U\downarrow C : one)) & = & rec(h) + 1 \\
rec(h;(C\uparrow U : ten)) & = & rec(h) \\
sent(\varepsilon) & = & 0 \\
sent(h;(U\downarrow C : five)) & = & sent(h) \\
sent(h;(U\downarrow C : one)) & = & sent(h) \\
sent(h;(C\uparrow U : ten)) & = & sent(h) + 10
\end{array}$$

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: 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

$$sendevents(h/\downarrow) \ \leq \ h/\uparrow$$

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.

$$\begin{array}{ll}
sendevents(\varepsilon) & = \varepsilon \\
sendevents(h;(A\uparrow B : m)) & = sendevents(h) \\
sendevents(h;(A\downarrow B : m)) & = sendevents(h);(A\uparrow B : m)
\end{array}$$

Channel-oriented systems can be mimicked by requiring FIFO ordering of communication for each pair of agents:

$$sendevents(h/A\downarrow B) \leq \ h/A\uparrow B$$

where $A\downarrow B$ denotes the set of receive-events with A as source and B as destination, and similarly for $A\uparrow B$.

11 Asynchronous Communication II

14.11.2014

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 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

$$x := \textbf{some} \tag{13}$$

where variable x may be given any (type correct) value

Local invariant reasoning by Hoare Logic

- each send statement **send** $B : m$ in A is treated as:

$$h := (h; A{\uparrow}B : m) \tag{14}$$

- each fixed receive statement **await** $B : m(\vec{x})$ in A[60] is treated as

$$\vec{x} := \textbf{ some } ; h := (h; B{\downarrow}A : m(\vec{x})) \tag{15}$$

  the usage of $\vec{x} :=$ **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

$$X := \textbf{ some } ; \textbf{await } X : m(\vec{x}) \tag{16}$$

  where the usage of $X :=$ **some** expresses that A may receive the message from any agent

Rule for non-deterministic assignments

Non-det assignment

$$\frac{}{\{\ \forall x\ .\ Q\ \}\ x := \textbf{ some }\ \{\ Q\ \}}\ \textsc{ND-Assign}$$

- 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

$$\frac{}{\{\ Q_{h\leftarrow h;A{\uparrow}B:m}\ \}\ \textbf{send } B : m\ \{\ Q\ \}}\ \textsc{Send}$$

$$\frac{}{\{\ \forall \vec{x}\ .\ Q_{h\leftarrow h;B{\downarrow}A:m(\vec{x})}\ \}\ \textbf{await } B : m(\vec{x})\ \{\ Q\ \}}\ \textsc{Receive}_1$$

$$\frac{}{\{\ \forall \vec{x}, X\ .\ Q_{h\leftarrow h;X{\downarrow}A:m(\vec{x})}\ \}\ \textbf{await } X\,?\,m(\vec{x})\ \{\ Q\ \}}\ \textsc{Receive}_2$$

- As before: A is 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.
- 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:

$$\frac{}{\{\ Q_{h\leftarrow h;(B{\downarrow}A:m)}\ \}\ \textbf{await } B : m\ \{\ Q\ \}}\ \textsc{Receive}$$

Hoare rules for local reasoning

The Hoare rule for non-deterministic choice ([ ]) is

Rule for [ ]

$$\frac{\{\ P_1\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P_2\ \}\ S_2\ \{\ Q\ \}}{\{\ P_1 \wedge P_2\ \}\ (S_1[\,]S_2)\ \{\ Q\ \}}\ \textsc{Nondet}$$

Remark: We may reason similarly backwards over conditionals:[61]

$$\frac{\{\ P_1\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P_2\ \}\ S_2\ \{\ Q\ \}}{\{\ (b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_2)\ \}\ \textbf{if } \ b\ \textbf{ then } \ S_1\ \textbf{ else } \ S_2\ \textbf{ fi } \ \{\ Q\ \}}\ \textsc{If}'$$

[60] where $\vec{x}$ is a sequence of variables

[61] We 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.

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:

$$\begin{array}{lll} U{\downarrow}C : five & -- & C \text{ consumes the message "five"} \\ U{\downarrow}C : one & -- & C \text{ consumes the message "one"} \\ C{\uparrow}U : ten & -- & C \text{ sends the message "ten"} \end{array}$$

Inner loop

let I_i ("inner invariant") abbreviate equation (8)

```
1   { I_i }
2   while b < 10   { b ≤ 10 ∧ I_i }
3   { (I_i b←(b+5))_{h←h;U↓C:five} ∧ (I_i b←(b+1))_{h←h;U↓C:one} }
4   do
5       ( await U:five;   { I_{i5←b+1} }
6         b:=b+5 )
7   []
8     (await U:one;   b:=b+1)
9     { I_i }
10  od;
11  { I_i ∧ b ≥ 10 }
12  { (I_{ob←b−10})_{h←h;C↑U:ten} }
13  send U:ten;
```

Must prove the implication:

$$b < 10 \wedge I_i \Rightarrow (I_{i\ b\leftarrow(b+5)})_{h\leftarrow h;U{\downarrow}C:five} \wedge (I_{i\ b\leftarrow(b+1)})_{h\leftarrow h;U{\downarrow}C:one}$$

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

```
1   { I_o }
2   loop
3     { I_o }
4     { I_i }
5     while b < 10   { b ≤ 10 ∧ I_i }
6     { (I_i b←(b+5))_{h←h;U↓C:five} ∧ (I_i b←(b+1))_{h←h;U↓C:one} }
7     do
8         ( await U:five;   { I_{i5←b+1} }
9           b:=b+5 )
10    []
11      (await U:one;   b:=b+1)
12      { I_i }
13    od;
14    { I_i ∧ b ≥ 10 }
15    { (I_{ob←b−10})_{h←h;C↑U:ten} }
16    send U:ten;
17    { I_{ob←b−10} }
18    b:=b−10
19    { I_o }
20  end
```

Verification conditions (as usual):

- $I_o \Rightarrow I_i$, and
- $I_i \wedge b \geq 10 \Rightarrow (I_{o\ b\leftarrow(b-10)})_{h\leftarrow h;C{\uparrow}U:ten}$
- 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:

$$Q(\vec{x}, h) \Rightarrow I_A(h) \tag{17}$$

- 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

- $I_A(\_)$: local history invariant of A
- first conjunct $h = ...$: specifies last communication step
- $I_A(h')$: assumption that invariant holds before the comm.-statement
- 3 communcation/sync. statements: `send` $B : m(e)$, **await** $B\, m(\vec{x})$, and **await** $X\,?\, m(\vec{x}) \Rightarrow$

3 kinds of verification conditions

$$(\ h = (h'; A{\uparrow}B : m(e)) \wedge I_A(h') \wedge Q(\vec{x}, h)\ ) \Rightarrow I_A(h) \tag{18}$$
$$(\ h = (h'; B{\downarrow}A : m(\vec{y})) \wedge I_A(h') \wedge Q(\vec{x}, h)\ ) \Rightarrow I_A(h) \tag{19}$$
$$(\ h = (h'; X{\downarrow}A : m(\vec{y})) \wedge I_A(h') \wedge Q(\vec{x}, h)\ ) \Rightarrow I_A(h) \tag{20}$$

in all three cases: Q is the condition right after the send-, resp. the await-statement

Coin machine example: local history invariant

For the coin machine C, consider the local history invariant $I_C(h)$ from last week (see equation (10)):

$$I_C(h) = 0 \leq sum(h/{\downarrow}) - sum(h/{\uparrow}) < 15$$

Consider the statement **send** $U : ten$ in C

- Hoare analysis of the outer loop gave the condition $I_{o\ b\leftarrow(b-10)}$ immediately after the statement
- history *ends* with the event $C{\uparrow}U : ten$

$\Rightarrow$ Verification condition, corresponding to equation (18):

$$h = h'; (C{\uparrow}U : ten) \wedge I_C(h') \wedge I_{o\ b\leftarrow(b-10)} \Rightarrow I_C(h) \tag{21}$$

Coin machine example: local history invariant

Expanding I_c and I_o in the VC from equation (21), and using definition of *sum* and using $(sum(h'/{\downarrow}) - sum(h'/{\uparrow}) = b$ in the last step

$$\begin{array}{l}
h = h'; (C{\uparrow}U : ten) \wedge \\
I_C(h') \wedge \\
I_{o\ b\leftarrow(b-10)} \\
\quad \Rightarrow I_C(h) \\
\hline
h = h'; (C{\uparrow}U : ten) \wedge \\
(0 \leq sum(h'/{\downarrow}) - sum(h'/{\uparrow}) < 15) \wedge \\
(sum(h/{\downarrow}) = sum(h/{\uparrow}) + b - 10 \wedge 0 \leq b - 10 < 5) \\
\quad \Rightarrow 0 \leq sum(h/{\downarrow}) - sum(h/{\uparrow}) < 15 \\
\hline
h = h'; (C{\uparrow}U : ten) \wedge \\
(0 \leq sum(h'/{\downarrow}) - sum(h'/{\uparrow}) < 15) \wedge \\
(sum(h'/{\downarrow}) = sum(h'/{\uparrow}) + 10 + b - 10 \wedge 0 \leq b - 10 < 5) \\
\quad \Rightarrow 0 \leq sum(h'/{\downarrow}) - sum(h'/{\uparrow}) - 10 < 15 \\
\hline
(0 \leq b < 15)\ \wedge\ 0 \leq b - 10 < 5) \\
\quad \Rightarrow 0 \leq b - 10 < 15
\end{array}$$

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:
$[\ C\updownarrow M : card\_in(n), M\updownarrow C : pin, C\updownarrow M : pin(x),\quad M\updownarrow C : amount, C\updownarrow M : amount(y), M\updownarrow B : request(n,x,y), B\updownarrow M : grant, M\updownarrow C : cash(y), M\updownarrow C : card\_out\ ]$
where n is name, x pin code, and y cash amount, provided by clients. Example of unsuccessful cycle: $[\ C\updownarrow M : card\_in(n), M\updownarrow C : amount, C\updownarrow M : amount(y), M\updownarrow B : request(n,x,y), B\updownarrow M : deny,\quad M\updownarrow C : card\_out\ ]$

Notation: $A\updownarrow B : m$ 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: $[C\uparrow M : card\_in(n), M\downarrow C : pin, C\uparrow M : pin(x),\quad M\downarrow C : amount, C\uparrow M : amount(y), M\downarrow C : cash(y), M\downarrow C : card\_o$
- Mini Bank: $[C\downarrow M : card\_in(n), M\uparrow C : pin, C\downarrow M : pin(x),\quad M\uparrow C : amount, C\downarrow M : amount(y), M\uparrow B : request(n,x,y), B\downarrow M : grant, M\uparrow C : cash(y), M\uparrow C : card\_out]$
- Central Bank: $[M\downarrow B : request(n,x,y), B\uparrow M : grant]$

The local histories may be used as guidelines when implementing the different agents.

Mini bank example: Local histories (2)

The unsuccessful cycle:

- Client: $[C\uparrow M : card\_in(n), M\downarrow C : pin, C\uparrow M : pin(x),\quad M\downarrow C : amount, C\uparrow M : amount(y), M\downarrow C : card\_out]$
- Mini Bank: $[C\downarrow M : card\_in(n), M\uparrow C : pin, C\downarrow M : pin(x),\quad M\uparrow C : amount, C\downarrow M : amount(y), M\uparrow B : request(n,x,y), B\downarrow M : deny, M\uparrow C : card\_out]$
- Central Bank: $[M\downarrow B : request(n,x,y), B\uparrow M : deny]$

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 nam

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:

$$[\ X{\downarrow}B : request(n,x,y), [\ B{\uparrow}X : grant\ +\ B{\uparrow}X : deny\ ]\ \textbf{some}\ X,n,x,y\ ]^*$$

- with + for choice, $[...]^*$ for repetition
- Defines cycles: *request* answered with either *grant* or *deny*
- notation $[regExp\ \textbf{some}\ X,n,x,y]^*$ 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 **is** R denote that h matches the structure described by R. Example (for events a, b, and c):

- we have $(a;b;a;b)$ **is** $[a,b]^*$
- we have $(a;c;a;b)$ **is** $[a,[b|c]]^*$
- we do *not* have $(a;b;a)$ **is** $[a,b]^*$

Loop invariant of Central Bank (B): Let $Cycle_B$ denote the regular expression:

$$[\ X{\downarrow}B : request(n,x,y), [\ B{\uparrow}X : grant\ +\ B{\uparrow}X : deny\ ]\ \textbf{some}\ X,n,x,y\ ]^*$$

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 $\{h\,\textbf{is}\,Cycle_B\}$
  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
$\{h\,\textbf{is}\,Cycle_B\}$
end

Loop invariant of the central bank (B):

```
loop
  { h is Cycle_B }
  { ∀ X,n,x,y . if pin[n] = x ∧ bal[n] > y then h1'' is Cycle_B else h2'' is Cycle_B }
  await X?request(n,x,y);
    { if pin[n] = x ∧ bal[n] > y then h1' is Cycle_B else h2' is Cycle_B }
  if    pin[n]=x and bal[n]>y

   then bal[n]:=bal[n]-y;
        { (h; B↑X : grant) is Cycle_B }
        send  X:grant;
        { (h; B↑X : grant) is Cycle_B }
   else
     { (h; B↑X : deny) is Cycle_B }
  fi
  { h is Cycle_B }
end
```

$$h_1'' = h; X{\downarrow}B : request(n,x,y); B{\uparrow}X : grant$$
$$h_1' = h; B{\uparrow}X : grant$$

Analogously (with *deny*) for h_2' and h_2''

Hoare analysis of central bank loop (cont.)

Verification condition:
$$h \,\mathbf{is}\, Cycle_B \Rightarrow \forall X,n,x,y \,.\, \mathbf{if}\ pin[n] = x \wedge bal[n] > y$$
$$\mathbf{then}\ (h; X{\downarrow}B : request(n,x,y); B{\uparrow}X : grant)\,\mathbf{is}\,Cycle_B$$
$$\mathbf{else}\ \ (h; X{\downarrow}B : request(n,x,y); B{\uparrow}X : deny)\,\mathbf{is}\,Cycle_B$$

where $Cycle_B$ is

$$[\ X{\downarrow}B : request(n,x,y), [\ B{\uparrow}X : grant\ +\ B{\uparrow}X : deny\ ]\ \mathbf{some}\ X,n,x,y\ ]^*$$

The condition follows by the general rule (regExp R and events a and b):

$$h\ \mathbf{is}\ R^* \wedge (a;b)\ \mathbf{is}\ R \Rightarrow (h;a;b)\ \mathbf{is}\ R^*$$

since $(X{\downarrow}B : request(n,x,y); B{\uparrow}X : grant)\,\mathbf{is}\,Cycle_B$ and $(X{\downarrow}B : request(n,x,y); B{\uparrow}X : deny)\,\mathbf{is}\,Cycle_B$

Local history invariant for the central bank (B)

$Cycle_B$ is

$$[\ X{\downarrow}B : request(n,x,y), [\ B{\uparrow}X : grant\ +\ B{\uparrow}X : deny\ ]\ \mathbf{some}\ X,n,x,y\ ]^*$$

Define the history invariant for B by:

$$h \le Cycle_B$$

Let $h \le R$ denote that h is a prefix of the structure described by R.

- intuition: if $h \le R$ we may find some extension h' such that $(h;h')$ **is** R
- h **is** $R \Rightarrow h \le R$ (but not vice versa)
- $(h;a)$ **is** $R \Rightarrow h \le R$
- Example: $(a;b;a) \ \le\ [a,b]^*$

Central Bank: Verification of the local history invariant

$h \le Cycle_B$

- 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 **is** $Cycle_B \Rightarrow h \le Cycle_B$ which is trivial.

2. after await request
if ... **then** $(h; B{\uparrow}X : grant)\,\mathbf{is}\,Cycle_B$ **else** $(h; B{\uparrow}X : deny)\,\mathbf{is}\,Cycle_B$
$\Rightarrow h \le Cycle_B$ which follows from $(h;a)\,\mathbf{is}\,R \Rightarrow h \le R$.

Note: We have now proved that the implementation of B satisfies the local history invariant, $h \ \le Cycle_B$.

Mini bank example: Local invariant of Client (C)

$Cycle_C$: $[\ \ C{\uparrow}X : card\_in(n)\ +\ X{\downarrow}C : pin, C{\uparrow}X : pin(x)\ +\ X{\downarrow}C : amount, C{\uparrow}X : amount(y')\ +\ X{\downarrow}C : cash(y)$
$+\ X{\downarrow}C : card\_out\ \ \mathbf{some}\ X,y,y'\ ]^*$

History invariant:

$$h_C \le Cycle_C$$

Note: The values of C, n and x are fixed from cycle to 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)

$Cycle_M$: [$C{\downarrow}M : card\_in(n), M{\uparrow}C : pin, C{\downarrow}M : pin(x)$, $M{\uparrow}C : amount, C{\downarrow}M : amount(y)$, **if** $y \leq 0$ **then** ε **else** $M{\uparrow}B : request(n, x, y), [B{\downarrow}M : deny \ + \ B{\downarrow}M : grant, M{\uparrow}C : cash(y)$] **fi** , $M{\uparrow}C : card\_out$ **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:
$H \leq$[$C{\updownarrow}M : card\_in(n), M{\updownarrow}C : pin, C{\updownarrow}M : pin(x)$, $M{\updownarrow}C : amount, C{\updownarrow}M : amount(y)$, **if** $y \leq 0$ **then** $M{\updownarrow}C : card\_out$ **else** $M{\updownarrow}B : request(n, x, y), [B{\updownarrow}M : deny, M{\updownarrow}C : card\_out \ + \ B{\updownarrow}M : grant, M{\uparrow}C : cash(y), [M{\downarrow}C : cash(y)$ ||| $M{\updownarrow}C : ca$ **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

References

References

[Andrews, 1991] Andrews, G. R. (1991). *Concurrent Programming: Principles and Practice.* Benjaming/Cummings Publishing Company.

[Andrews, 2000] Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley.

[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.* Wiley & Sons.