IN5170/IN9170 – Models of concurrency

Fall 2019

November 25, 2019

Abstract

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

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

Contents

1 Introduction to IN5170/IN9170

27th August 2019

General Info
Curriculum:

- Main book: **G. R. Andrews. Foundations of Multithreaded, Parallel, and Distributed Programming**. Addison Wesley, 2000 (Chapters 1 to 10).
- slides and notes from the course home page
- one or more papers to be announced

Obligs:

- 2 obligatory assignments

Exam:

- Written exam Dec. 9th

1.1 Warming up

Today's agenda

Introduction

- overview
- motivation
- simple examples and considerations

Start
a bit about

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

What this course is about

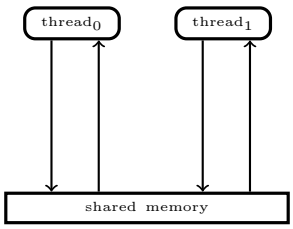
- Fundamental issues related to cooperating parallel processes
- How to think about developing parallel processes
- Various language mechanisms, design patterns, and paradigms
- Deeper understanding of parallel processes:
  - properties
  - informal analysis
  - *formal* analysis

Parallel processes

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

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

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



Course overview – part I: Shared variables

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

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
- active object languages
- Go: modern language proposal with concurrency at the heart (channels, goroutines)

Part I: shared variables

Why shared (global) variables?

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

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

Simple example

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

before after?after

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

Pre/post-conditions

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

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

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

Parallel operator $\|$

Consider: shared and non-shared programvariables, assignment.[1mm]

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

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

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

Example 1.

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

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

Interaction between processes

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

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

organization of this interaction: "*synchronization*"

Synchronization (veeery abstractly)

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

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

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

Concurrent processes: Atomic operations

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

Note

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

Atomic operations on global variables

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

Mutual exclusion
Atomic operations on a variable cannot happen simultaneously.

Example

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

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

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

Atomic read and write operations

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

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

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

4 atomic x-operations:

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

Interleaving & possible execution sequences

- "program order":
  - R1 must happen before W1 and
  - R2 before W2
- `inc` and `dec` ("-1") work process-local[3]

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

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

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

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

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

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

Non-determinism

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

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

[3] e.g.: in an arithmetic register, or a local variable (not mentioned in the code).
[4] Of course, things like $x \in \{-1, 0, 1\}$ or $-1 \leq x \leq 1$ are equally adequate formulations of the postcondition.

State-space explosion

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

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

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

For n processes with m atomic statements each:

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

The "at-most-once" property

Fine-grained atomicity
only the very most basic operations (R/W) are atomic "by nature"

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

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

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

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

At-most-once examples
In all examples:

- x, y shared variables, initially 0
- r, r' etc: local var's (registers)
- **co** and **oc** around $\ldots \parallel \ldots$ omitted

$x := x + 1 \parallel y := x + 1$ $\quad \{\ x{=}1 \wedge y \in \{(1,2)\}\ \}$
$x := y + 1 \parallel y := x + 1$ $\quad \{\ (x,y) \in \{(1,1),(1,2),(2,1)\}\ \}$
$x := y + 1 \parallel x := y + 3 \parallel y := 1$ $\quad \{\ y{=}1 \wedge x{=}1,2,3,4\ \}$
$r := y + 1 \parallel r' := y - 1 \parallel y := 5$
$r := x - x \parallel \ldots$ $\quad$ {is r now 0?}
$x := x \parallel \ldots$ $\quad$ {same as skip?}
if $y > 0$ **then** $y := y - 1$ **fi** $\parallel$ **if** $y > 0$ **then** $y := y - 1$ **fi**

1.2 The await language

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

Summary:

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

Syntax: Sequential part

We use the following syntax for non-parallel control-flow[5]

Declarations	Assignments
`int i = 3;`	`x := e;`
`int a[1:n];`	`a[i] := e;`
`int a[n];`[6]	`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*

Parallel statements

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

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

Parallel processes

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

- Processes evaluated in arbitrary order.
- Processes are declared (as methods/functions)
- side remark: the convention "declaration = start process" is *not* used in practice.[7]

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

[7] 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.

Example

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

Starts one process.

The numbers are printed in increasing order.

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

Starts n processes.

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

Read- and write-variables

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

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

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

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

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

- Note: assuming expressions side-effect free!

Disjoint processes

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

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

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

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

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

1.3 Semantics and properties

Semantic concepts ("Interleaving Semantics")

- A state in a parallel program consists of the values of the variables at a given moment in the execution.
- Each process executes independently of the others by *modifying* global variables using atomic operations.
- 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[8] is a predicate which is true for all possible histories of the program.

  Classification

  - *safety* property: program will not reach an undesirable state
  - *liveness* property: program will reach a desirable state.

- *partial correctness*: *If* the program terminates, it is in a desired final state (safety property).
- *termination*: all histories are finite.[9]
- *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 all reachable states.

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

  ***proof principle:* induction**
  one can show that an invariant is correct by

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

  Note: we avoid looking at all possible executions!

How to check properties of programs?

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

Dijkstra's dictum:
A test can only show errors, but "never" prove correctness!

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

Critical sections

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

- When the non-interference requirement does not hold: *synchronization* to restrict the possible histories.
- Synchronization gives coarser grained atomic operations.
- The notation $\langle S \rangle$ means that S is performed *atomically*.[10]

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:

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

Conditional critical sections

Await statement

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

- boolean condition b: *await condition*
- body S: executed atomically (conditionally on b)
  Example 5.
  $$\langle \textbf{await}\ (y > 0)\ y := y - 1 \rangle$$
- *synchronization*: decrement delayed until (if ever) $y > 0$ holds

2 special cases

- unconditional critical section or "mutex"[11]
  $$\langle x := 1; y := y + 1 \rangle$$
- Condition synchronization:[12]
  $$\langle \textbf{await}\ (\mathit{counter} > 0)\ \rangle$$

Typical pattern

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

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

- "critical statements" *not* enclosed in ⟨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

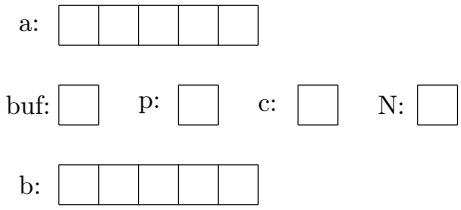
[10] In programming languages, one could find it as `atomic{`S`}` or similar.
[11] Later, a special kind of semaphore (a binary one) is also called a "mutex". Terminology is a bit flexible sometimes.
[12] One may also see sometimes just **await** (b): however, the evaluation of b better be *atomic* and under *no* circumstances must b have *side-effects* (*Never, ever. Seriously*).

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

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

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

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

Example (continued)



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

2 Locks & Barriers (week 2)

3. 09. 2019

Practical Stuff

Mandatory assignment 1 ("oblig 1")

- Deadline: Sunday September 22 at 24.00
- Online delivery (Devilry): https://devilry.ifi.uio.no

Introduction

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

Today

- Entry- and exit protocols to *critical sections*

- Protect reading and writing to *shared variables*

- *Barriers*
  - Iterative algorithms: Processes must *synchronize* between each iteration
  - Coordination using *flags*

Remember: await-example: Producer/Consumer

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

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

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

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

2.1 Critical sections

Critical section

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

Main question today:
How can we implement critical sections / conditional critical sections?

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

Access to Critical Section (CS)

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

Critical section: First approach to a solution

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

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

General pattern for CS

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

⇒ Programming advice: be aware of exceptions inside CS!

Naive solution

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


process p1 {                    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.

note: The three first are *safety* properties,[13] The last a *liveness* property.

Safety: Invariants (review)

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

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

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

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

Atomic section

Used for synchronization of processes

- General form:

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

  - 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:[14]

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

Critical sections using "locks"

```
bool lock = false;

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

Safety properties:

- Mutex
- Absence of deadlock
- Absence of unnecessary waiting

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

"Test & Set" (TAS)

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

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

Effect of TS(lock)

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

[14] 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
  }
}
```

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

Strong fairness[15] 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? (TTAS - Test, Test And Set)

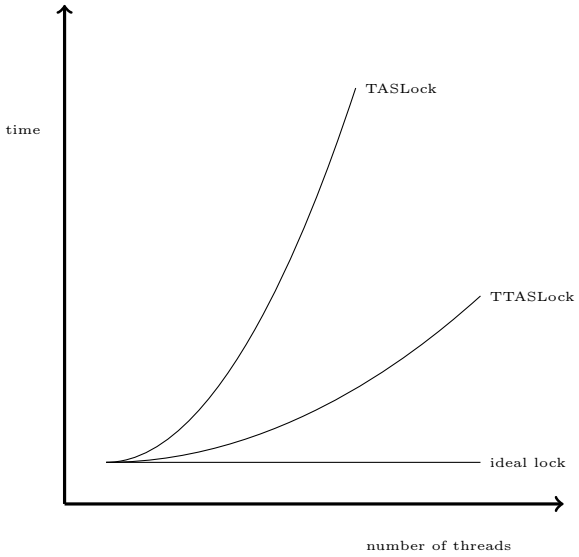
```
bool lock := false;

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

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

```
bool lock := false;

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

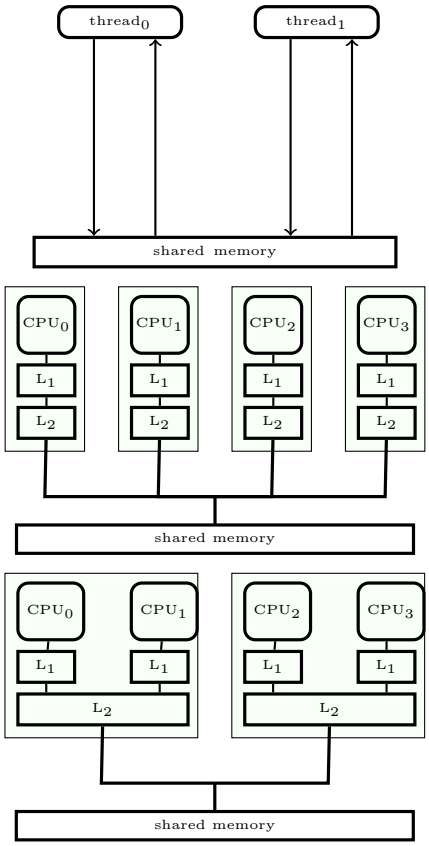
Does that make sense?

Multiprocessor performance under load (contention)

[15] see later



A glance at HW for shared memory



Test and test & set (TTAS)

- Test-and-set operation:
  - (Powerful) HW instruction for synchronization
  - Accesses main memory (and involves "cache synchronization")
  - Much slower than cache access
- Spin-loops: faster than TAS loops
- "Double-checked locking": sometimes design pattern/programming idiom for efficient CS (under certain architectures)[16]

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

Implementing await-statements

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

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

CSentry; S; **CSexit**;

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

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

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

2.2 Liveness and fairness

Liveness properties

So far: no(!) solution for "Eventual Entry".[17]

Liveness

Eventually, something good will happen.

- Typical example for sequential programs: Program termination[18]
- Typical example for parallel programs: A given process will eventually enter the critical section

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

Scheduling and fairness

enabledness (considering processes on a shared processor)

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

- Concurrent programs: often more than 1 statement enabled!

```
bool x := true;

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

Scheduling: resolving non-determinism

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

Fairness (informally)

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

Fairness notions

- Fairness: how to pick among enabled actions without being "passed over" indefinitely
- Which actions in our language are potentially non-enabled?[19]
- Possible status changes:
  - disabled → enabled (of course),

[17] Except the very first (which did not satisfy "absence of unnecessary delay"

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

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

– 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

Unconditional fairness

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

Example:

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

- x := **false** is unconditional

⇒ The action will eventually be chosen

- guarantees termination here
- Example: "Round robin" execution
- Note: if-then-else, **while** (b) ; are *not* conditional atomic statements!
- uncond. fairness formulated here based in (un)-conditional atomic actions

Weak fairness

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

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

Example:

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

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

Strong fairness

Example

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

Definition 8 (Strongly fair scheduling strategy)**.**

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

For the example:

- under strong fairness: y true ∞-often ⇒ termination
- under *weak fairness:* non-termination possible

Fairness for critical sections using locks

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

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

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

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

Fair solutions to the CS problem

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

Tie-Breaker algorithm

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

Tie-Breaker algorithm: Attempt 1

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

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

What is the global invariant here?

Problem: No *mutex*

Tie-Breaker algorithm: Attempt 2

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

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

```
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"[20] :-(

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

```
process p1 {                          in1 := false, in2 := false; int last
  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.

[20] 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 not last = 2){skip}
    CS
    in1 := false;
    non-CS
  }
}
```

Generalizable to many processes (see book)

Ticket algorithm

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

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

- Works like the "take a number" queue at the post office (with one loop)
- A customer (process) which comes in takes a number which is higher than the number of all others who are waiting
- The customer is served when a ticket window is available and the customer has the lowest ticket number.

Ticket algorithm: Sketch (n processes)

```
int number := 1; next := 1; turn[1:n] := ([n] 0);

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

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

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

Ticket algorithm: Implementation

```
int number := 1; next := 1; turn[1:n] := ([n] 0);

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

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

Without this instruction, we use an extra CS:[21]

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

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

[21] Why?

Ticket algorithm: Invariant

Invariants

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

$$0 < next \leq number$$

- What is the *local* invariant for process i:
  - before the entry: turn[i] < number
  - if p[i] in 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!! ("contention")

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

```
Worker[i]:    task i;
    arrive[i] := 1;
    < await (continue[i] = 1);>

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

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

Flag synchronization principles:

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

Synchronization using flags

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

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

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

- a bit like "message passing"
- see also semaphores next week

Combined barriers

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

Implementation of Critical Sections

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

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

Drawbacks:

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

Desirable to have special tools for synchronization protocols

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

3 Semaphores (week 3)

10. September, 2019

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

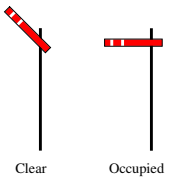
- This lecture: Semaphores (synchronization tool)
  - Used easily for mutual exclusion and condition synchronization.
  - A way to implement signaling (and scheduling).
  - implementable in many ways.
  - available in programming language libraries and OS

Outline

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

Semaphores

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



Properties

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

Concept

- *Semaphore*: special kind of shared program variable (with built-in sync. power)
- value of a semaphore: a *non-negative* integer
- can *only* be manipulated by two atomic operations:[22]

  P **and** *V*

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

- nowadays, for libraries or sys-calls: other names are preferred (up/down, wait/signal, ...)
- different "flavors" of semaphores (binary vs. counting)
- a mutex: often (basically) a synonym for binary semaphore

Syntax and semantics

- declaration of semaphores:
  - sem s; default initial value is zero
  - sem s := 1;
  - sem s[4] := ([4] 1);
- semantics[23] (via "implementation"):

  P-operation P(s)
  $\langle$**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

[22] There are different stories about what Dijkstra actually wanted *V* and *P* to stand for.
[23] Semantics generally means "meaning"

Example: Mutual exclusion (critical section)

Mutex[24] implemented by a binary semaphore

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

Note:

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

Example: Barrier synchronization

Semaphores may be used for signaling events

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

Note:

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

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

⇒ code between the P and the V

- all semaphores = 0
- code executed in mutex

Example: Producer/consumer with split binary semaphores

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

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

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

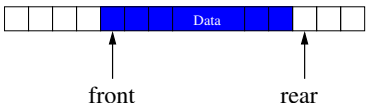
```
process Consumer {
  while (true) {
    P(full);
    data_c := buff;
    V(empty);
  }
}
```

Note:

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

Producer/consumer: Increasing buffer capacity

- previously: tight coupling, the producer must wait for the consumer to empty the buffer before it can produce a new entry.
- easy generalization: buffer of size n.
- loose coupling/asynchronous communication ⇒ "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

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

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

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

several producers or consumers?

Increasing the number of processes

- several producers and consumers.
- New synchronization problems:
  - Avoid that two producers deposits to `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

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

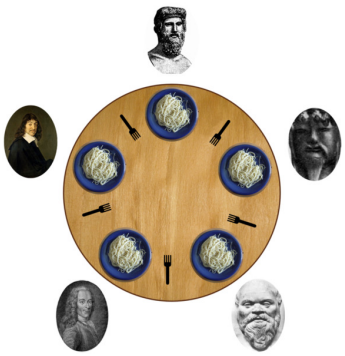
```
process Producer {
  while (true) {
    P(empty);
    P(mutexDeposit);
    buff[rear] := data;
    rear := (rear + 1) % n;
    V(mutexDeposit);
    V(full);
  }
}
```

```
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 around a circular table.
- one fork placed between each pair of philosophers
- each philosopher 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`

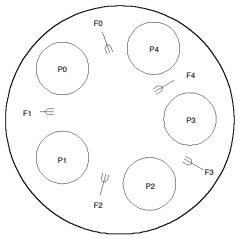
[25] image from wikipedia.org
From top: Plato, Konfuzius, Socrates, Voltaire and Descartes.

Dining philosophers: 1st attempt

- forks as semaphores
- philosophers: pick up left fork first

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

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



ok solution?

Example: Dining philosophers 2nd attempt

breaking the symmetry

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

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

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

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

Dining philosophers

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

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 classes of processes competes for access to the database
  - Readers compete with writers
  - Writers compete both with readers and other writers
- General synchronization problem:
  - readers: must wait until no writers are active in DB
  - writers: must wait until no readers or writers are active in DB
- here: two different approaches
  1. Mutex: easy to implement, but "unfair"[26]
  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);
  }
}
```

[26] The way the solution is "unfair" does not technically fit into the fairness categories we have introduced.

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

    write to DB

    V(rw);
  }
}
```

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

Readers/writers with mutex (2)

Initially:

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

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

    read from DB

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

}
```

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

    P(rw);

    write to DB


    V(rw);
  }

}
```

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

Readers/writers with mutex (3)

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

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

    read from DB

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

"Fairness"

What happens if we have a constant stream of readers? "Reader's preference"

Readers/writers with condition synchronization: overview

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

Invariant

reasonable 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, check/synchronize:
  - 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] Invariant outside critical regions on

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:

  1. associate one counter and
  2. one 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

```
int nr := 0, nw = 0;      # condition variables (as before)
sem e  := 1;              # entry semaphore
int dr := 0; sem r := 0; # delay counter + sem for reader
int dw := 0; sem w := 0; # delay counter + sem for writer

# invariant  RW: (nr = 0 ∨ nw = 0 ) ∧ nw ≤ 1
```

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

      read from DB;

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

Note: First reader does nr+1 without having e, the next ones have e.

[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
```

- This passes the control (the baton) to an appropriate process!
- SIGNAL has no P operation, each path has exactly one V oper.
- using the conditions to see who goes next.
- Called "passing the baton" technique (as in relay competition).

Overview: condition synchronization

- one semaphore for protection of shared variables (the counters)
- for each condition: a semaphore + a "delay" counter
- on entry increase delay counter if your condition is not true
- wait one your condition semaphore
- decide who is next (SIGNAL) using
  - the conditions and
  - the delay counters to see who is waiting to enter
- SIGNAL whenever someone should get a chance to enter.

4 Monitors (week 5)

2. Oct. 2019

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:[29]
  - Forgotten P or V operations
  - Too many P or V operations
  - They are shared between processes
    - Global knowledge
    - Need to examine all processes to see how a semaphore is intended

Monitors

Monitor

"Abstract data type + synchronization"

- program *modules* with *more structure* than semaphores
- monitor encapsulates data, which can only be *observed* and *modified* by the monitor's procedures.
  - contains variables that describe the *state*
  - variables can be changed only through the available procedures
- implicit mutex: only 1 procedure may be active at a time.
  - A procedure: mutex access to the data in the monitor
  - 2 procedures in the same monitor: never executed concurrently
- cooperative scheduling
- Condition synchronization:[30] is given by *condition variables*
- At a lower level of abstraction: monitors can be implemented using locks or semaphores (for instance)

[29] Same may be said about simple locks.
[30] block a process until a particular condition holds.

Usage

- process = active ⇔ 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 public procedures are important
- implementation can be changed, if visible effects remains
- Monitors and processes can be developed relatively independent ⇒ Easier to understand and develop parallel programs

Syntax & semantics

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

monitor: a form of abstract data type:

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

  call *name.opname*(*arguments*)

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

monitor invariant: describe the monitor's inner states

Condition variables

- monitors contain *special* type of variables: **cond** (condition)
- used for synchronization/to delay processes
- each such variable is associated with a *wait condition*
- "*value*" of a condition variable: *queue* of delayed processes
- value: not directly accessible by programmer
- Instead, manipulated by special operations

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

4.1 Semaphores & signalling disciplines

Implementation of semaphores

A monitor with P and V operations:

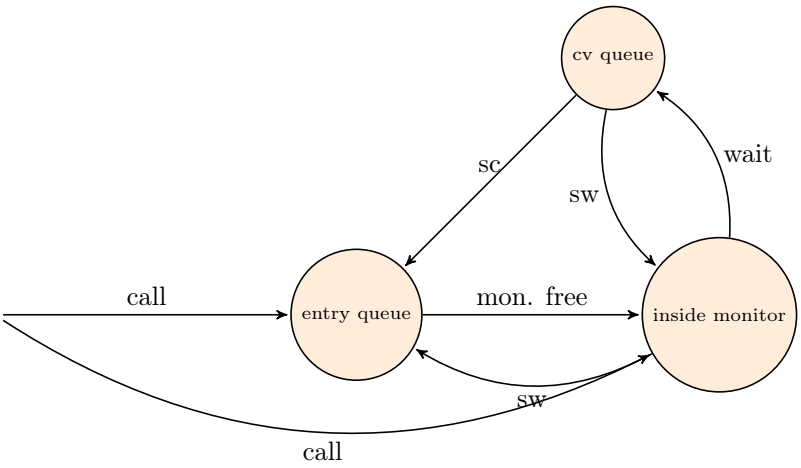
```
monitor Semaphore { # monitor invariant: s ≥ 0
  int s := 0          # value of the semaphore
  cond pos;           # wait condition

  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`: *FIFO signaling strategy*
- When a process executes `signal(cv)`, then it is inside the monitor. If a waiting process is woken up: *two active processes* in the monitor?

2 disciplines to provide mutex:

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



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

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       { # 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[31]

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);
      }
}
```

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

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

Monitor procedures:

- `request_read`: requests read access
- `release_read`: reader leaves DB
- `request_write`: requests write access
- `release_write`: writer leaves DB

Invariants and signalling

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

`nr` — number of readers
`nw` — number of writers

Invariant

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

We want `RW` to be a *monitor invariant*

- chose carefully condition variables for "communication" (waiting/signaling)

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

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

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

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

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

Invariant

- monitor invariant I: describe the monitor's inner state
- expresses relationship between monitor variables
- maintained by execution of procedures:
  - must hold: after initialization
  - must hold: when a procedure terminates
  - must hold: when we suspend execution due to a call to `wait`

  ⇒ can assume that the invariant holds *after* `wait` and when a procedure starts
- Should be as *strong* as possible

Monitor solution to reader/writer problem (6)

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

```
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
}
```

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

Do we need to protect release_read and release_write?

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)`: 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 efficient if many processes will wait for a long time
- Can give many false alarms

Prioritized waiting

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

Time server: Prioritized wait

- Uses prioritized waiting to order processes by `check`
- The process is awakened only when `tod` $\geq$ `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 in the 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

Rendezvous

Similar to a two-process barrier: *Both* parties must arrive before either can continue.[32]

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

The barber can have rendezvous with an arbitrary customer.

Organize the sync.: Identify the synchronization needs

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

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-rendezvous
  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

[32] Later, in the context of message passing, will have a closer look at making rendezvous synchronization (using channels), but the pattern "2 partners must be present at a point at the same time" is analogous.

Sleeping barber

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

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

  while (open = 0) wait(door_open);          # leave shop
  open := open - 1; signal(customer_left);
}
procedure get next customer() {              # RV with client
  barber := barber + 1; signal (barber available);
  while (chair = 0) wait(chair_occupied);
  chair := chair - 1;
}
 procedure finished_cut() {
   open := open + 1; signal(door_open);      # get rid of customer
   while (open > 0) wait(customer_left);
}
```

5 Program Analysis Part I : Sequential Programs (week 4)

24. 9. 2019

Program correctness

Is my program correct?

Central question for this lecture (Part I) and Part II.

- 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 (Part I): Basic machinery for program reasoning (Part II): Extending this machinery to the concurrent setting

Concurrent executions

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

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

- 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, not so practical for concurrent programs

- *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
    - Verification
    - (Symbolic) Model Checking

Formal Specifications

Kinds of formal specifications:

- pre/post conditions
- loop invariant
- class invariants. monitor invariants, data invariants
- global and local invariants in concurrent systems
  - may be violated inside critical regions

Formal specifications are useful

- when designing a system, using the specification as a guide
- in order to understand a program
- using the specification as a contract between different units,
  - in particular concurrent units
- for testing, using specifications in testing
  - for test case generation

States

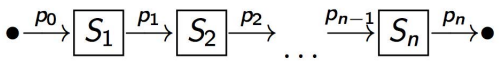
- *state* of a program consists of the values of the program variables at a point in time, example: $\{ x = 2 \wedge y = 3 \}$
- The state space of a program is given by the different values that the declared variables can take
- Sequential program: one execution thread operates on its own state space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

Example 9.

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

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 10 (from previous slide).

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

Assertions

Want to express more general properties of programs, like invariants:

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

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

Example 11. The assertion $x = y$ describes *all* states where the values of x and y are equal, like $\{x = -1 \wedge y = -1\}$, $\{x = 1 \wedge y = 1\}$, ...

Assertions

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

$x = y$	all states where x has the same value as y
$x \leq y$:	all states where the value of x is less or equal to the value of y
$x{=}2 \wedge y{=}3$	the state where x is 2 and y is 3
$x{=}2 \vee y{=}3$	the state where x is 2 or y is 3
$x{=}2 \Rightarrow y{=}3$	the state where y is 3 if x is 2
true	all states
false	no state

Example 12.

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

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

Formal analysis of programs

- Establish program properties/correctness, using a system for formal reasoning
- Help in understanding how a program behaves
- Useful for program construction
  - in particular, class and loop invariants
- Look at logics for formal analysis
- basis of analysis tools

Formal system

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

Logics and formal systems

Our formal system consists of:

- a language, formed by syntactic building blocks:
  - A set of *symbols* (constants, variables,...)
  - A set of *formulas* (meaningful combination of symbols)
- derivation machinery
  - A set of *axioms* (assumed to be true) [33]
  - A set of *inference rules*

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

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

Symbols (first-order logic with equality)

- variables: $x, y, z, \ldots$ which include
  - program variables
  - + "extra" ones including logical variables
- Relation symbols: $\leq, \geq, \ldots$
- Function symbols: $+, -, \ldots$, and constants $0, 1, 2, \ldots, \mathit{true}, \mathit{false}$
- Equality (also a relation symbol): $=$
- Connectives $\neg, \vee, \wedge, \Rightarrow$
- Quantifiers: $\forall, \exists$

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 a formula, the following are formulas:[34]

$\forall x : A(x)$	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"

[33] **NB:** An axiom can be seen as a rule with no premise.
[34] $A(x)$ to indicate that, here, A (typically) contains x.

Examples of axioms and rules (propositional logic)

Typical axioms:

$$A \vee \neg A \qquad\qquad A \Rightarrow A \qquad\qquad \mathit{true}$$

Typical rules:[35]

$$\frac{A \quad B}{A \wedge B}\ \textsc{And-I} \qquad \frac{A}{A \vee B}\ \textsc{Or-I} \qquad \frac{A \Rightarrow B \quad A}{B}\ \textsc{Impl-E/modus ponens}$$

Example 13.

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

$$\frac{x \geq 0 \Rightarrow y \geq 0 \quad x \geq 0}{y \geq 0}\ \textsc{Impl-E}$$

Important terms

- **Proof**: derivation tree where all leaf nodes are axioms
- **Theorems**: a "formula" derivable in a given proof system
- **Interpretation**: describe each formula as either *semantically true* or *false*
- **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 semantically 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 (or sequence of statements)
  - P, P_1, P', Q ...: assertions over program states
    - including use of $\neg, \wedge, \vee, \Rightarrow, \exists, \forall$
  - In above triple: P_1 pre-condition, and P_2 post-condition of S

Example 14.

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

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

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, sometimes called a contract
  - Code analysis by proving the specification (in PL)
  - Symbolic and static analysis:
    - 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 semantically *true*
    - $\{\ x = 0\ \}\ x := x + 1;\ \{\ x = 0\ \}$ should be semantically *false*

Reasoning about programs

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

Interpretation:
Interpretation ("semantics") of triples is related to program execution

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

- If the initial state of S satisfies P (P holds for the initial state of S) and
- **if**[36] S *terminates,*
- *then* Q is *true* in the final state of S

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

Example 15. $\{x = y\}$ `x := x * 2; y := y * 2;` $\{x = y\}$ is *true* if the initial state satisfies $x = y$ and if the execution terminates, then the final state satisfies $x = y$

Examples
Some true triples

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

Some non-true triples

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

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

Partial correctness

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

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

$\{ P \}\ S\ \{ \mathit{false} \}$	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[37]
- 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 holds

- Example: we want
  $$\{ x = 0 \}\ x := x + 1\ \{ x = 1 \}$$
  to be a theorem (since it was interpreted as *true*),
- but
  $$\{ x = 0 \}\ x := x + 1\ \{ x = 0 \}$$
  should *not* be a theorem (since it was interpreted as *false*)

Soundness:[38]
All theorems in PL hold

$$\vdash \{ P \}\ S\ \{ Q \} \quad \text{implies} \quad \models \{ P \}\ S\ \{ Q \} \tag{3}$$

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

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

[38] technically, we'd need a semantics for reference, otherwise it's difficult to say what a program "really" does.

Textual substitution

Substitution
$P[e/x]$ means, all free occurrences of x in P are replaced by expression e. (Notation of book: $P_{x \leftarrow e}$)

Example 16.

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

Substitution propagates into formulas:

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

Free and "non-free" variable occurrences

$P[e/x]$

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

Example 17 (Substitution).

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

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:

$$\begin{array}{rl} \{\ e = 5\ \} & \texttt{x := e}\ \ \{\ x = 5\ \} \\ \{\ (x = 5)[e/x]\ \} & \texttt{x := e}\ \ \{\ x = 5\ \} \end{array}$$

Axiom of assignment

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

Axiom for the assignment statement

$$\{\ P[e/x]\ \}\ x := e\ \{\ P\ \} \quad \textsc{Assign}$$

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

NB: e evaluated in the prestate = value of x in the poststate

PL structural inference rules

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

$$\frac{\{\ P\ \}\ S\ \{\ Q\ \} \qquad \{\ P'\ \}\ S\ \{\ Q'\ \}}{\{\ P \wedge P'\ \}\ S\ \{\ Q \wedge Q'\ \}}\ \textsc{Conjunction}$$

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

- Red: logical proof obligations

Axiom of skip

The skip statement does nothing

Axiom:

$$\{\ P\ \}\ \textbf{skip}\ \{\ P\ \}\quad \textsc{Skip-axiom}$$

Proving an assignment

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

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

The red implication is a logical proof obligation. In this course we will not prove proof obligations formally.

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

Examples

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

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

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

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

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

PL inference rules

$$\frac{\{\, P \wedge B \,\}\ S\ \{\, Q \,\} \qquad P \wedge \neg B \Rightarrow Q}{\{\, P \,\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{fi}\ \{\, Q \,\}}\ \textsc{Cond1}$$

$$\frac{\{\, P \wedge B \,\}\ S\ \{\, Q \,\} \qquad \{\, P \wedge \neg B \,\}\ S'\ \{\, Q \,\}}{\{\, P \,\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ S'\ \{\, Q \,\}}\ \textsc{Cond2}$$

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

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

Sequential composition and consequence

Backward construction over assignments:

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

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

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

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 18.

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

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:

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

$$\frac{\{P \wedge B\}\ S\ \{Q\} \qquad (P \wedge \neg B) \Rightarrow Q}{\{\, P \,\}\ \textbf{if}\ \ B\ \textbf{then}\ \ S\ \textbf{fi}\ \{\, Q \,\}}\ \textsc{Cond1}$$

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

$$x = x_0 \land x < 0 \Rightarrow (-x \geq 0 \land (-x = x_0 \lor -x = -x_0))$$

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

15. 10. 2019

6 Program Analysis: Part II Concurrency (week 7)

Program Logic (PL)

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

  "triple" (or "Hoare triple")

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

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

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

PL rules from part I: structural inference rules

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

$$\frac{\{ P \}\ S\ \{ Q \} \qquad \{ P' \}\ S\ \{ Q' \}}{\{ P \land P' \}\ S\ \{ Q \land Q' \}}\ \textsc{Conjunction}$$

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

- Red: logical proof obligations

PL rules from part I: if/while/assignment

$$\frac{\{ P \land B \}\ S\ \{ Q \} \qquad P \land \neg B \Rightarrow Q}{\{ P \}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{fi}\ \{ Q \}}\ \textsc{Cond1}$$

$$\frac{\{ P \land B \}\ S\ \{ Q \} \qquad \{ P \land \neg B \}\ S'\ \{ Q \}}{\{ P \}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ S'\ \{ Q \}}\ \textsc{Cond2}$$

$$\frac{\{ I \land B \}\ S\ \{ I \}}{\{ I \}\ \textbf{while}\ B\ \textbf{do}\ S\ \{ I \land \neg B \}}\ \textsc{While}$$

$$\{ P[e/x] \}\ x := e\ \{ P \}\quad \textsc{Assign}$$

How to actually *use* the while rule?

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

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

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

While rule

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

Can use this rule to reason about the general situation:

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

where

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

Combine While-rule with Consequence-rule to prove:

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

While rule: example

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

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

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

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

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

Await statement

Rule for await

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

Remember: we are reasoning about safety properties/partial correctness

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

Concurrent execution and interference

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:

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

Example 19 (Problem with this rule).

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

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

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

Interference problem

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

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

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

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

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

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

Interference problem

Apply the previous "parallel-composition-is-conjunction"rule again:

$$\frac{\{\ x = 0 \lor x = 2\ \}\ \langle x := x + 1\rangle\ \{\ x = 1 \lor x = 3\ \} \quad \{\ x = 0 \lor x = 1\ \}\ \langle x := x + 2\rangle\ \{\ x = 2 \lor x = 3\ \}}{\{\ PRE\ \}\ \mathbf{co}\langle x := x + 1\rangle \parallel \langle x := x + 2\rangle\ \mathbf{oc}\ \{\ POST\ \}}$$

where:

$$PRE\ : (x = 0 \lor x = 2) \land (x = 0 \lor x = 1)$$
$$POST : (x = 1 \lor x = 3) \land (x = 2 \lor x = 3)$$

which gives:

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

Concurrent execution

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

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

Interference freedom

A process interferes with (the specification of) another process, if its execution changes the assertions[39] of the other process.

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

Interference freedom

Interference freedom

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

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

is derivable in PL (= theorem).

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

$$\frac{\{\ P_1\ \}\ S_1\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ S_2\ \{\ Q_2\ \}}{\{\ P_1 \land P_2\ \}\ \mathbf{co}\ S_1 \parallel S_2\ \mathbf{oc}\ \{\ Q_1 \land Q_2\ \}}$$

Four interference freedom requirements (for 2 statements in co-oc):

$$\{P_2 \land P_1\}\ S_1\ \{P_2\} \qquad \{P_1 \land P_2\}\ S_2\ \{P_1\}$$
$$\{Q_2 \land P_1\}\ S_1\ \{Q_2\} \qquad \{Q_1 \land P_2\}\ S_2\ \{Q_1\}$$

[39] Only "critical assertions" considered, i.e. outside mutex-protected sections

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

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

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

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

Avoiding interference: Global invariants

Global inductive invariants

- An assertion that only refers to global (shared) variables
- Holds initially.
- Preserved by all assignments/transitions ("inductive")

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

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

Avoiding interference: Synchronization

- Hide critical conditions
- MUTEX to critical sections

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

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

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

Example: Producer/ consumer synchronization

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

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

PC a *global inductive invariant* of the producer/consumer?

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

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

Example: Producer

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

```
process Producer {
  int a[n];
  { I_P }                    //  entering loop
  while (p < n) {            { I_P ∧ p < n }
    < await (p = c); >       { I_P ∧ p < n ∧ p = c }
                             { I_P[p+1/p][a[p]/buf] }
    buf := a[p];             { I_P[p+1/p] }
    p := p + 1;              {I_P}
  } {I_P ∧ ¬(p < n)}         //  exit loop
     ⇔ {PC ∧ p = n}
}
```

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

Example: Consumer

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

```
process Consumer {
  int b[n];
  {I_C}                      // entering loop
  while (c < n) {            {I_C ∧ c < n}
    < await (p > c) ; >      {I_C ∧ c < n ∧ p > c}
                             {I_C[c+1/c][buf/b[c]]}
    b[c] := buf;             {I_C}[c+1/c]
    c := c + 1;              {I_C}
  } {I_C ∧ ¬(c < n)}         // exit loop
     ⇔ {PC ∧ c = n ∧ b[0 : c-1] = a[0 : c-1]}
}
```

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

Example: Producer/Consumer

The final state of the program satisfies:

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

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

Interference freedom is ensured by the global/local discipline, except for the assertions right after the await tests (which are talking also about shared variables).

The cointerference freedom requirements for these conditions give:

Combining the two `await` assertions, 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!*

Thus the cointerference freedom requirements are satisfied.

Monitor invariant

```
monitor name {
   monitor variables
   initialization
   procedures
}
```

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

Axioms for signal and continue (1)

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

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

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 + 1/nr] }
        nr := nr + 1;
        { I }
}
```

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

Axioms for Signal and Continue (2)

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

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

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

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

by assignment axiom:

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

Axioms for Signal and Continue (3)

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

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

$$\begin{aligned}\textbf{signal}(cv) :\ & \{((\#cv = 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P[\#cv - 1/\#cv])\}\\ & if\ (\#cv \neq 0)\ \#cv := \#cv - 1\\ & \{P\}\end{aligned}$$

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

Axioms for Signal and Continue (4)

Together this gives:

Axioms for monitor communication

$\{\ I[\#cv + 1/\#cv]\ \}$ **wait**(cv) $\{\ I\ \}$ WAIT

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

$\{\ P[0/\#cv]\ \}$ **signal_all**(cv) $\{\ P\ \}$ SIGNALALL

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

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

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

Example: FIFO semaphore verification (1)

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

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

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

Consider the following monitor invariant:

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

No process is waiting if the semaphore value is positive!

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

Example: FIFO semaphore verification: Psem

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

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

Example: FIFO semaphore verification (3)

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

This gives two proof obligations: If-branch:

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

Else branch:

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

Note: $(s - 1 > 0 \Rightarrow \#pos = 0)$ follows since we have $\#pos = 0$.

Example: FIFO semaphore verification: Vsem

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

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

Example: FIFO semaphore verification (5)

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

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

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

Else branch:

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

Summary

- interference proofs tricky – easy to forget something
  - critical regions help (but may reduce efficiency)
  - global invariants help
    - avoid interference by splitting in global invariant and local condition
- monitor invariants much easier – like local reasoning
  - signal/wait need special care
  - can reason about the length of the queue

7 Java concurrency (lecture 6)

8. 10. 2019

7.1 Threads in Java

Outline

1. Monitors: review
2. Threads in Java:
   - Instantiating and starting threads
     - Thread class and Runnable interface
   - Threads and interference
   - Synchronized blocks and methods: (atomic regions and monitors)
3. Example: The ornamental garden
4. Thread interaction & condition synchronization (wait and signal/notify)
5. Example: Thread interaction
6. Example: Mutual exclusion
7. Example: Readers/writers

Short recap of monitors

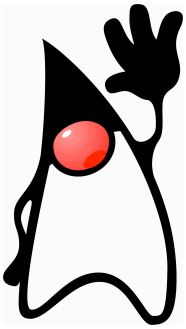
- monitor encapsulates data, which can only be observed and modified by the monitor's procedures
  - Contains variables that describe the *state*
  - variables can be accessed/changed only through the available *procedures*
- Implicit mutex: Only one procedure may be active at a time.
  - 2 procedures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

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

Java

From Wikipedia:[40]

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



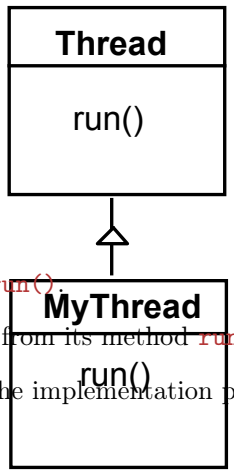
[40] But it's correct nonetheless ...

Threads in Java

A thread in Java

- thread of execution/unit of concurrency[41], and
- instance of java.lang.Thread.
- originally "green threads"
- identity, accessible via static method `Thread.CurrentThread()`[42]
- 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



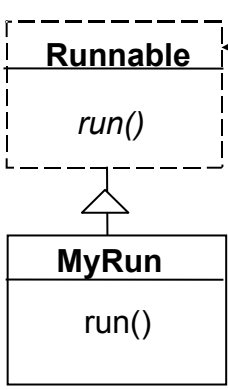
- The thread specific action happens in `run()`.
- The `Thread` class executes instructions from its method `run()`.
- The actual code executed depends on the implementation provided for `run()` in a derived class.
- start() to launch a new call stack.
- limitations?

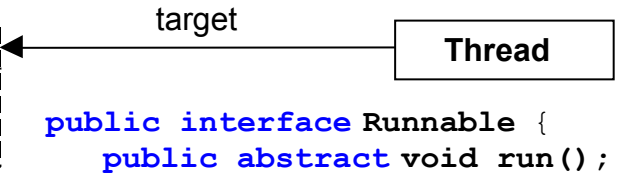
```
class MyThread extends Thread {
  public void run() {
    System.out.println(``My thread'');
  }
}
  // Creating a thread object:
  Thread a = new MyThread();
  a.start();
```

[41] as such, roughly corresponding to the concept of "processes" from previous lectures.
[42] What's the difference to `this`?

Runnable interface

- no multiple inheritance.
- *thread* and the *job* split into two classes–Thread class for the *thread-specific* code and Runnable implementation class for *job-that-should-be-run-by-a-specific-thread-code.*
- Thread class itself implements Runnable.






```
public interface Runnable {
    public abstract void run();
}

class MyRun implements Runnable{
    public void run() {
        //.....
    }
}
```

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

Runnable Example

```
class MyRunnable implements Runnable {
  public void run() {
    for(int x = 1; x < 6; x++) {
      System.out.println("Runnable running");}
    }
}

public class TestThreads {
  public static void main (String [] args) {
    MyRunnable r = new MyRunnable();
    Thread t = new Thread(r);
    t.start();
  }
}
```

Threads in Java

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

1. Define class that
   - *extends* the java.lang.Thread class or
   - *implements* the Runnable interface[43].
2. define *run* method inside the new class[44]
3. create an instance of the new class.
4. invoke *start()* on the class instance to start new thread.

- **t.run()**, valid but doesn't start a new thread.

[43] although in real world you're much more likely to implement Runnable than extend Thread.
[44] overriding, late-binding.

Interference and Java threads

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

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

t1 and t2 execute s.update() concurrently!
Interference between t1 and t2 ⇒ may lose updates to data.

Synchronization

avoid interference ⇒ "synchronize" code that modifies shared data

1. built-in lock for each object in Java
2. mutex: at most one thread t can lock o at any time.[45]
3. Java keyword : "synchronized"
4. two "flavors" of synchronization:

"synchronized method"

whole method body of m "protected"[46]:

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

"synchronized block"

```
synchronized (o) { ... }
```

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();
 s.update();
```

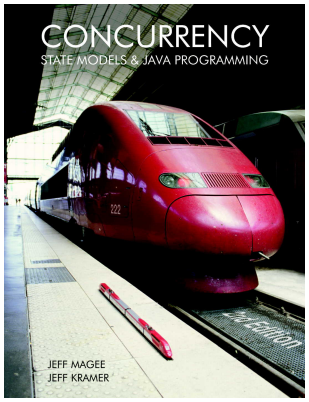
[45] but: in a re-entrant manner!
[46] assuming that other methods play according to the rules as well etc.

Key points : locks & synchronization

- "monitor" : object, whose lock is acquired.
- only one lock per object.
- only methods and blocks can be synchronized, not variables or classes.
- class can have "*synchronized*" and "*non-synchronized*" methods.
- thread can acquire more than one lock, e.g., from a synchronized method invoking another synchronized method.
- for **o.update()**, lock on **o** is acquired.
- synchronization does hurt concurrency
  - synchronized method could always be replaced with a non-synchronized method containing synchronized block.

Java Examples

Book:
Concurrency: State Models & Java Programs, 2^{nd} Edition
Jeff Magee & Jeff Kramer
Wiley



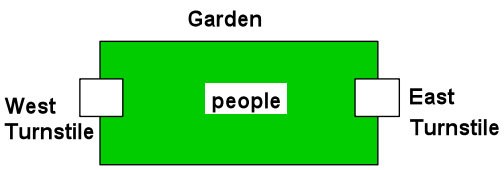
`http://www.doc.ic.ac.uk/~jnm/book/`

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

7.2 Ornamental garden

Ornamental garden problem

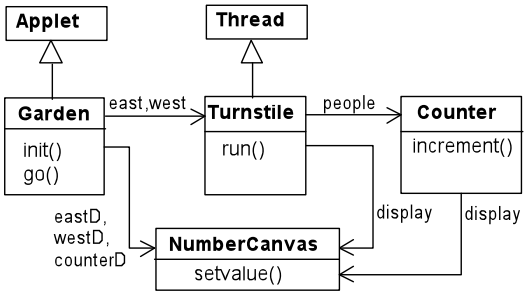
- people enter an ornamental garden through either of 2 turnstiles.
- problem: the number of people present at any time.



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

```
class Counter {

   int value = 0;
   NumberCanvas display;

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

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

Turnstile

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

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

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

Ornamental Garden Program

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

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

Ornamental Garden Program: DEMO

DEMO

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

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

Avoid interference by synchronization

```
class SynchronizedCounter extends Counter {

  SynchronizedCounter(NumberCanvas n) {
    super(n);
  }

  synchronized void increment() {
    super.increment();
  }
}
```

Mutual Exclusion: The Ornamental Garden - DEMO

DEMO

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[47]
  - *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
- these methods must be called from within a synchronized context.
- note: notify does *not* operate on a thread-identity[48]

⇒

```
Thread t = new MyThread();
...
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 locked 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[49]
  - besides that: there's re-entrance!

Example : Thread interaction

```
class ThreadA {
  public static void main(String[] args){
    ThreadB b = new ThreadB();
    b.start();

    synchronized(b) {
      try{
        b.wait();
      } catch (InterruptedException e){}
      System.out.println(``Total is :'' +b.total); %
    }
}

class ThreadT extends Thread {
  public void run () {
    synchronized(this){
      for(int i=0;i<10;i++) {
        total += i;
       }
      notify();
      }
    }
  }
```

[47] there are more

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

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

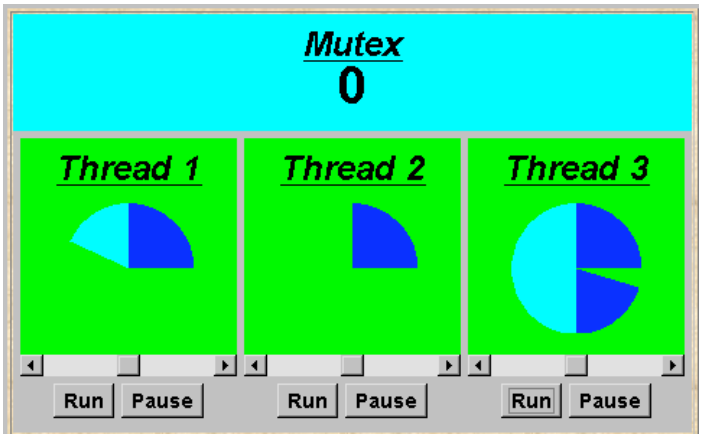
A semaphore implementation in Java

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

public class Semaphore {
  private int value;

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

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

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

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

7.4 Semaphores

Mutual exclusion with semaphores



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

Mutual exclusion with semaphores

```
class MutexLoop implements Runnable {

    Semaphore mutex;

    MutexLoop (Semaphore sema) {mutex=sema;}

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

DEMO

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

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

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

```
interface ReadWrite {

   public void acquireRead() throws InterruptedException;

   public void releaseRead();

   public void acquireWrite() throws InterruptedException;

   public void releaseWrite();
 }
```

Reader client code

```
class Reader implements Runnable {

   ReadWrite monitor_;

   Reader(ReadWrite monitor) {
     monitor_ = monitor;
    }

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

Writer client code

```
class Writer implements Runnable {

   ReadWrite monitor_;

   Writer(ReadWrite monitor) {
     monitor_ = monitor;
    }

    public void run() {
      try {
        while(true)  {
             while(!ThreadPanel.rotate());
             // begin critical section
             monitor_.acquireWrite();
```

```
            while(ThreadPanel.rotate());
            monitor_.releaseWrite();
        }
      } catch (InterruptedException e){}
    }
}
```

R/W monitor (regulate readers)

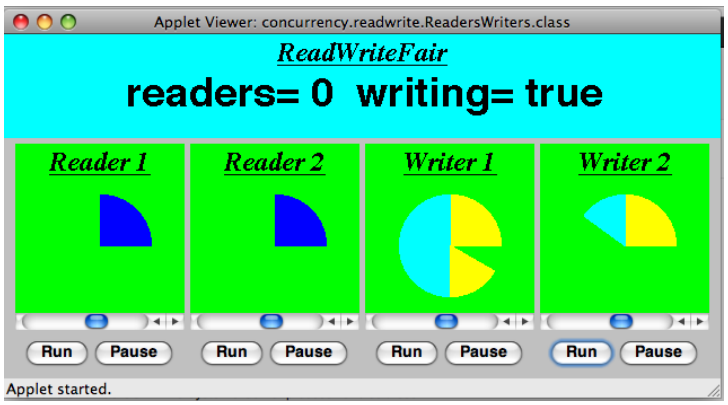
```
class ReadWriteSafe implements ReadWrite {
  private int readers =0;
  private boolean writing = false;

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

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

  public synchronized void acquireWrite() {...}

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

R/W monitor (regulate writers)

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

  public synchronized void acquireRead() {...}

  public synchronized void releaseRead() {...}

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

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

DEMO

Fairness



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

"Fairness": regulating readers

```
class ReadWriteFair implements ReadWrite {

    private int readers =0;
    private boolean writing = false;
    private int waitingW = 0; // no of waiting Writers.
    private boolean readersturn = false;

    synchronized public void acquireRead()
    throws InterruptedException {
      while (writing || (waitingW>0 && !readersturn)) wait();
      ++readers;
      }

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

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

"Fairness": regulating writers

```
class ReadWriteFair implements ReadWrite {

    private int readers =0;
    private boolean writing = false;
    private int waitingW = 0; // no of waiting Writers.
    private boolean readersturn = false;

    synchronized public void acquireRead() {...}
    synchronized public void releaseRead() {...}

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

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

Readers and Writers problem

DEMO

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

Java concurrency

- there's (much) more to it than what we discussed (synchronization, monitors) (see `java.util.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,[50] 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 programming remains a bit of an art

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

8 Message passing and channels (lecture 7)

10. Oct. 2018

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[51]
- 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

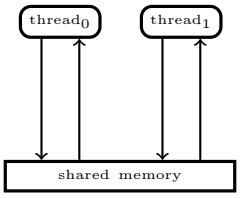
Distributed memory architectures:

- Processor has private memory and communicates over a "network" (inter-connect)
- Examples:

[50] and pointer aliasing and a weak memory model makes it worse.
[51] The dividing line is not absolute. One can make perfectly good use of channels and message passing also in a non-distributed setting.

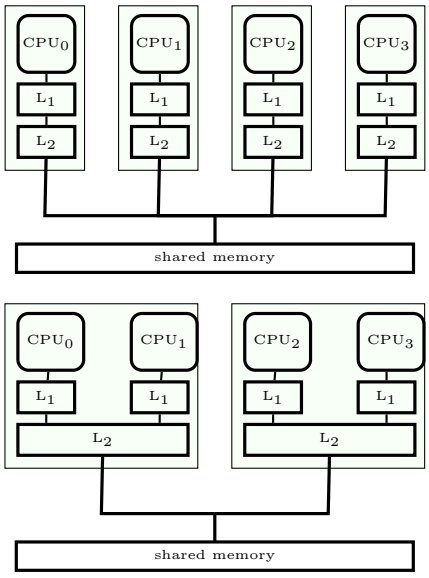
- Multicomputer: asynchronous multi-processor with distributed memory (typically contained inside one case)
- Workstation clusters: PC's in a local network
- Grid system: machines on the Internet, resource sharing
- cloud computing: cloud storage service
- NUMA-architectures
- cluster computing ...

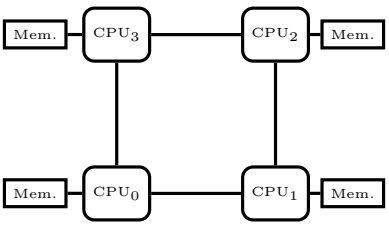
Shared memory concurrency in the real world



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

SMP, multi-core architecture, and NUMA





Concurrent vs. distributed programming

Concurrent programming:

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

Distributed programming:

- Memory is distributed ⇒ 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[52]

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

Variants: errors possible, untyped, ...

Asynchronous message passing: primitives

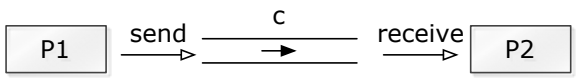
Channel declaration

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

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



Simple channel example in Go

```
func main() {
        messages := make(chan string, 0)  // declare + initialize

        go func() { messages <- "ping" }() // send
        msg := <-messages                  // receive
        fmt.Println(msg)
}
```

Short intro to the Go programming language

- programming language, executable, used by f.ex. Google
- supporting channels and asynchronous processes (function calls)
  - *go-routine*: a lightweight thread
- syntax: mix of functional language (lambda calculus) and imperative style programming (built on C).

[52] but remember also: producer-consumer problem

Some syntax details of the Go programming language

Calls

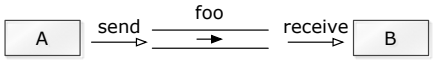
- $f(x)$ – ordinary (synchronous) function call, where f is a defined function or a functional definition
- **go** $f(x)$ – called as an asynchronous process, i.e. go-routine **Note**: the go-routine will die when its parent process dies!
- **defer** $f(x)$ – the call is delayed until the end of the process

Channels

- $chan := make(chanint, buffersize)$ – declare channel
- $chan < -x$ – send x
- $< -chan$ – receive
- example: $y :=< -chan$ – receive in y

Run command: **go run program.go** – compile and run program

Example: message passing

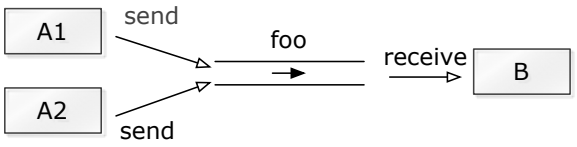


(x,y) = (1,2)

```
chan foo(int);

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

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

Example: shared channel

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




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

process A2 {
  send foo(2);
}

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

```
func main() {
        foo  := make(chan int, 10)
        go func() {
                time.Sleep(1000)
                foo <- 1        // send
        }()

        go func() {
                time.Sleep(1)
                foo <- 2
        }()
        fmt.Println("first = ", <-foo)
        fmt.Println("second = ", <-foo)
}
```

Asynchronous message passing and semaphores

Comparison with general semaphores:

channel	$\simeq$	semaphore
send	$\simeq$	**V**
receive	$\simeq$	**P**

Number of messages in queue = value of semaphore

(Ignores content of messages)

Semaphores as channels in Go

```
type dummy interface {}             // dummy type,
type Semaphore chan dummy           // type definition

func (s Semaphore) Vn (n int) {
        for i:=0; i<n; i++ {
                s <- true           // send something
        }
}
func (s Semaphore) Pn (n int) {
        for i:=0; i<n; i++ {
                <- s                // receive
        }
}

func (s Semaphore) V () {
        s.Vn(1)
}
func (s Semaphore) P () {
        s.Pn(1)
}
```

Listing 2: 5 Phils

```
package main
import ("fmt"
        "time"
        "sync"
        "math/rand"
        "andrewsbook/semchans")     // semaphores using channels


var wg sync.WaitGroup

const m = 5                 // let's make just 5
var forks  = [m]semchans.Semaphore {
        make (semchans.Semaphore,1),
        make (semchans.Semaphore,1),
        make (semchans.Semaphore,1),
        make (semchans.Semaphore,1),
        make (semchans.Semaphore,1)}
```

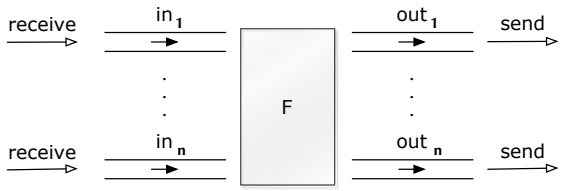
```
func main () {
        for i:=0; i<m; i++ {    // initialize the sem's
                forks[i].V()
        }
        wg.Add(m)
        for i:=0; i<m; i++ {
                go philosopher(i)
        }
        wg.Wait()

}

func philosopher(i int) {
  defer wg.Done()
  r := rand.New(rand.NewSource(99))    // random generator
  fmt.Printf("start P(%d)\n",i)
  for true {
    fmt.Printf("P(%d) is thinking\n",i)
    forks[i].P()
//    time.Sleep(time.Duration(r.Int31n(0)))  // small delay for DL
    forks[(i+1)%m].P()
    fmt.Printf("P(%d) starts eating\n",i)
    time.Sleep(time.Duration(r.Int31n(5)))  // small delay
    fmt.Printf("P(%d) finishes eating\n",i)
    forks[i].V()
    forks[(i+1)%m].V()
  }
}
```

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: naturally seen as a composition of filters.
- cf. *stream* processing/programming (feedback loops) and *dataflow programming*

Example: A single filter process

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

process **Sort** with input channels **input** and output channel **output**.

Define:

n : number of values sent to **output**. $sent[i]$: i'th value sent to **output**.

Sort predicate

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

```
in1: 1 4 9 ...
                          out: 1 2 4 5 8 9 ...
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 $\mathbf{in}_2$

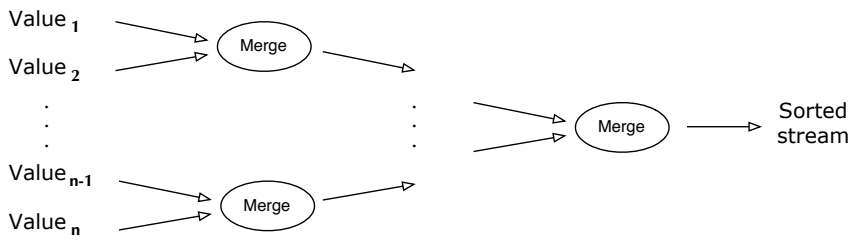
Example: Merge process

```
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.[53]

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"

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

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

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

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

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

Allocator as a monitor

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

  procedure acquire(int &id) {  # var.parameter
    if (avail = 0)
      wait(free);
    else
      avail := avail-1;
    remove(units, id);
  }

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

([Andrews, 2000, Fig. 7.6])

Allocator as a server process: code design

1. interface and "data structure"
   (a) allocator with two types of operations: get unit, free unit
   (b) 1 request channel[54] ⇒ 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)

[54] Alternatives exist

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)
```

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

```
chan values(int);

process Producer {
  int data[n];
  for [i = 0 to n-1] {
    ... # computation ...;
    synch_send values(data[i]);
} }

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

Assume both producer and consumer vary in time complexity.
Communication using `synch_send`/`receive` will block.

With *asynchronous* message passing, the waiting is reduced.

Example: deadlock using synchronous message passing

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

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

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

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

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

```
func main() {
        var wg sync.WaitGroup // wait group
        c1,c2  := make(chan int, 0),make(chan int, 0)
        wg.Add(2)             // prepare barrier
        go func() {
                defer wg.Done() // signal to barrier
                c1  <- 1        // send
                x := <- c2      // receive
                fmt.Printf("P1: x := %v\n", x)
        }()

        go func() {
                defer wg.Done()
                c2  <- 2
                x := <- c1
                fmt.Printf("P2: x := %v\n", x)
        }()
        wg.Wait()          // barrier
}
```

9 Weak Memory Models

12. 11. 2019

Concurrency

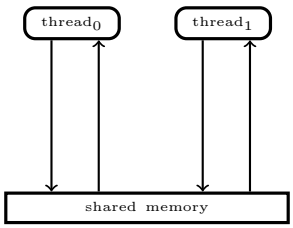
Concurrency

"Concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other" (Wikipedia)

- performance increase, better latency
- many forms of concurrency/parallelism: multi-core, multi-threading, multi-processors, distributed systems ...

9.1 Hardware architectures

Shared memory: a simplistic picture



- one way of "interacting" (i.e., communicating and synchronizing): via shared memory
- a number of threads/processors: access common memory/address space
- interacting by sequence of reads/writes (or loads/stores, etc.)

However: considerably harder *to get correct and efficient programs*

Dekker's solution to mutex

- As known, shared memory programming requires synchronization: e.g. mutual exclusion

Dekker

- simple and first known mutex algo
- here simplified

initially: $\texttt{flag}_0 = \texttt{flag}_1 = 0$

`flag0 := 1;`	`flag1 := 1;`
`if (flag1 = 0)`	`if (flag0 = 0)`
`then` CRITICAL	`then` CRITICAL

Known textbook "fact":
Dekker is a software-based solution to the mutex problem (or is it?)

A three process example

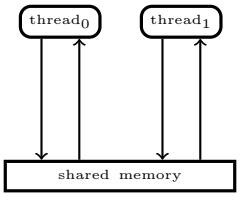
Initially: `x,y = 0`, `r`: register, local var

thread$_0$	thread$_1$	thread$_2$
`x := 1`	`while (x = 0) do skip;`	`while (y = 1) do skip`
	`y:=1`	`r:=x`

"Expected" result
Upon termination, register r of the third thread will contain $r = 1$.

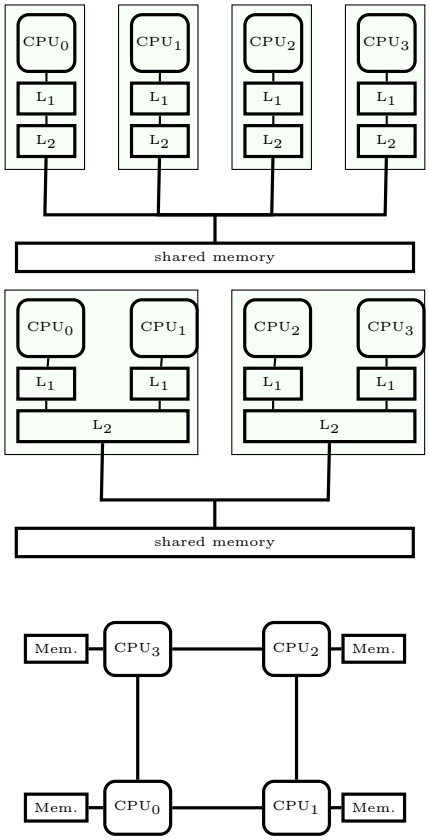
But:
Who ever said that there is only one identical copy of x that thread$_1$ and thread$_2$ operate on?

Shared memory concurrency in the real world



- the memory architecture does not reflect reality
- out-of-order executions: 2 interdependent reasons:
  1. modern HW: complex memory hierarchies, caches, buffers ...
  2. compiler optimizations

SMP, multi-core architecture, and NUMA

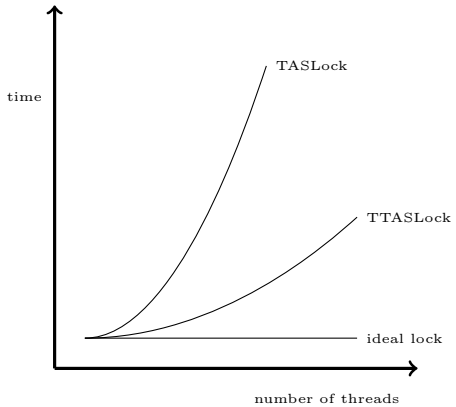


"Modern" HW architectures and performance

```
public class TASLock implements Lock {
    ...
    public void lock() {
        while (state.getAndSet(true)) { }  // spin
    }
    ...
}
```

```
public class TTASLock implements Lock {
    ...
    public void lock() {
        while (true) {
            while (state.get()) {}; //spin
            if (!state.getAndSet(true))
                return;
        }
        ...
    }
}
```

Observed behavior



(cf. [Anderson, 1990] [Herlihy and Shavit, 2008, p.470])

9.2 Compiler optimizations

Compiler optimizations

- many optimizations with different forms:

  elimination of reads, writes, sometimes synchronization statements
  re-ordering of independent, non-conflicting memory accesses
  introductions of reads

- examples
  - constant propagation
  - common sub-expression elimination
  - dead-code elimination
  - loop-optimizations
  - call-inlining
  - ... and many more

Code reodering

Initially: `x = y = 0`

thread_0	thread_1
`x := 1`	`y:= 1;`
`r1 := y`	`r2 := x;`
`print r1`	`print r2`

possible print-outs $\{(0,1),(1,0),(1,1)\}$

$\Longrightarrow$

Initially: `x = y = 0`

thread_0	thread_1
`r1 := y`	`y:= 1;`
`x := 1`	`r2 := x;`
`print r1`	`print r2`

possible print-outs $\{(0,0),(0,1),(1,0),(1,1)\}$

Common subexpression elimination

Initially: `x = 0`

thread_0	thread_1
`x := 1`	`r1 := x;`
	`r2 := x;`
	`if r1 = r2`
	`then print 1`
	`else print 2`

$\Longrightarrow$

Initially: `x = 0`

thread_0	thread_1
`x := 1`	`r1 := x;`
	`r2 := r1;`
	`if r1 = r2`
	`then print 1`
	`else print 2`

Is the transformation from the left to the right correct?

$thread_0$	$W[x] := 1;$			
$thread_1$		$R[x] = 1;$	$R[x] = 1;$	`print(1)`
$thread_0$		$W[x] := 1;$		
$thread_1$	$R[x] = 0;$		$R[x] = 1;$	`print(2)`
$thread_0$			$W[x] := 1;$	
$thread_1$	$R[x] = 0;$	$R[x] = 0;$		`print(1)`
$thread_0$				$W[x] := 1;$
$thread_1$	$R[x] = 0;$	$R[x] = 0;$	`print(1);`	

2nd prog: only 1 read from memory $\Rightarrow$ only `print(1)` possible

- transformation left-to-right ok
- transformation right-to-left: new observations, thus not ok

Compiler optimizations

***Golden rule* of compiler optimization**
Change the code (for instance re-order statements, re-group parts of the code, etc) in a way that leads to

- better performance (at least on average), but is otherwise
- unobservable to the programmer (i.e., does not introduce new observable result(s)) when executed single-threadedly, i.e. *without concurrency!* ***:-O***

In the presence of concurrency

- more forms of "interaction"

⇒ more effects become observable

- standard optimizations become observable (i.e., "break" the code, assuming a naive, standard shared memory model)

Is the *Golden Rule* outdated?

Golden rule as task description for compiler optimizers:

- Let's assume for convenience, that there is no concurrency, how can I make make the code faster ....
- and if there's concurrency? too bad, but not my fault ...

- unfair characterization
- assumes a "naive" interpretation of shared variable concurrency (interleaving semantics, SMM)

What's needed:

- golden rule must(!) still be upheld
- but: relax naive expectations on what shared memory is

⇒ *weak memory model*

DRF
golden rule: also core of "data-race free" programming principle

Compilers vs. programmers

Programmer

- wants to understand the code

⇒ profits from strong memory models

Compiler/HW

- want to optimize code/execution (re-ordering memory accesses)

⇒ take advantage of weak memory models

⟹

- What are valid (semantics-preserving) compiler-optimations?
- What is a good memory model as compromise between programmer's needs and chances for optimization

Sad facts and consequences

- incorrect concurrent code, "unexpected" behavior
  - Dekker (and other well-know mutex algo's) is incorrect on modern architectures[55]
  - in the three-processor example: $r = 1$ not guaranteed
- unclear/obstruse/informal hardware specifications, compiler optimizations may not be transparent
- understanding of the memory architecture also crucial for performance

Need for unambiguous description of the behavior of a chosen platform/language under shared memory concurrency $\Longrightarrow$ memory models

Memory (consistency) model

What's a memory model?
"A formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by a system." [Adve and Gharachorloo, 1995]

MM specifies:

- How threads interact through memory?
- Which values a read can return?
- When does a value update become visible to other threads?
- What assumptions are allowed to make about memory when writing a program or applying some program optimization?

9.3 Sequential consistency

Sequential consistency

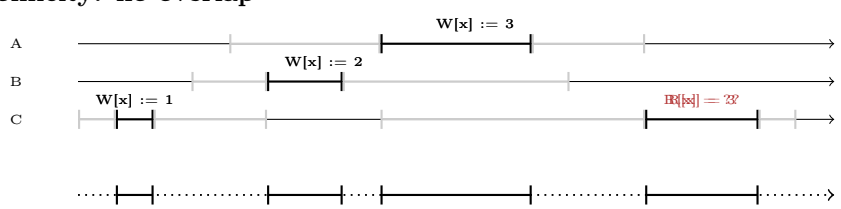
- in the previous examples: unspoken assumptions

1. Program order: statements executed in the order written/issued (Dekker).
2. atomicity: memory update is visible to everyone at the same time (3-proc-example)

Lamport [Lamport, 1979]: Sequential consistency
"...the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

- "classical" model, (one of the) oldest correctness conditions
- simple/simplistic $\Rightarrow$ (comparatively) easy to understand
- straightforward generalization: single $\Rightarrow$ multi-processor
- weak means basically "more relaxed than SC"

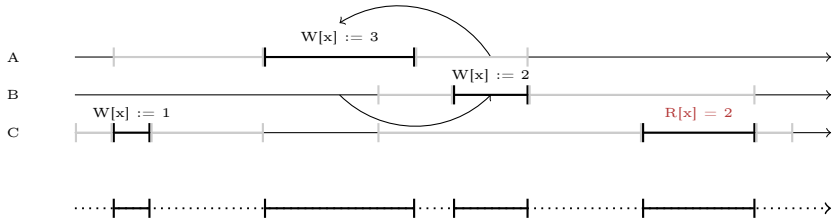
Atomicity: no overlap



Which values for x consistent with SC?

[55] Actually already since at least IBM 370.

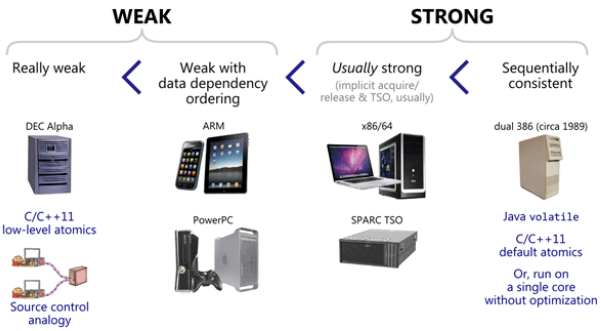
Some order consistent with the observation



- read of 2: observable under sequential consistency (as is 1, and 3)
- read of 0: contradicts program order for thread C.

10 Weak memory models

Spectrum of available architectures



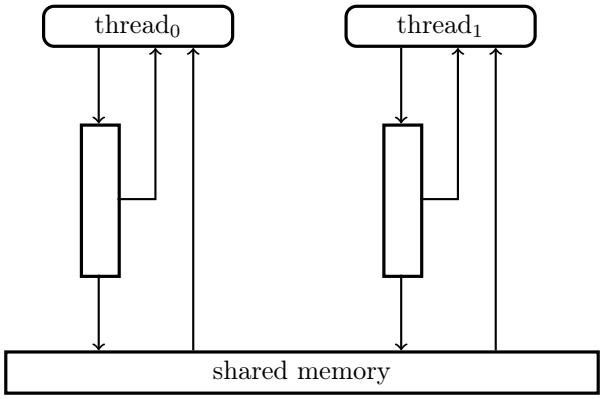
(from http://preshing.com/20120930/weak-vs-strong-memory-models)

Trivial example

thread_0	thread_1
`x := 1`	`y := 1`
`print y`	`print x`

Result?
Is the printout `0,0` observable?

Hardware optimization: Write buffers



10.1 TSO memory model (Sparc, x86-TSO)

Total store order

- TSO: SPARC, pretty old already
- x86-TSO
- see [Owell et al.,] [Sewell et al., 2010]

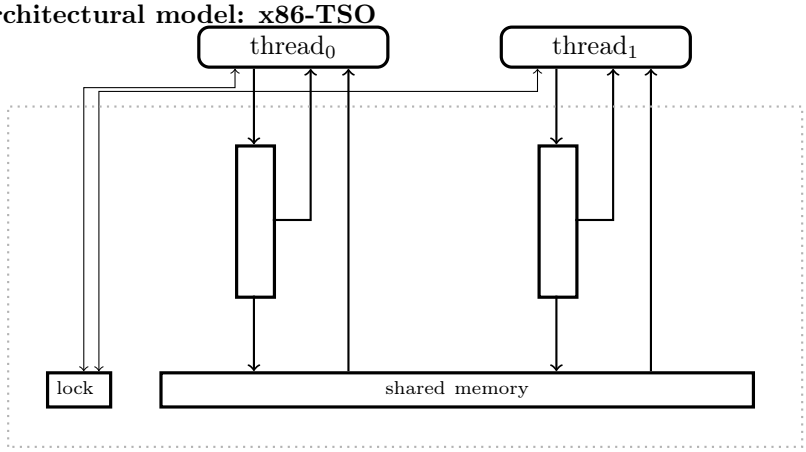
Relaxation

1. architectural: adding store buffers (aka write buffers)
2. axiomatic: relaxing program order ⇒ W-R order dropped

Architectural model: Write-buffers (IBM 370)

Architectural model: TSO (SPARC)

Architectural model: x86-TSO



Directly from Intel's spec

Intel 64/IA-32 architecture sofware developer's manual [int, 2013] (over 3000 pages long!)

- single-processor systems:
  - Reads are not reordered with other reads.
  - Writes are not reordered with older reads.
  - Reads may be reordered with older writes to different locations but not with older writes to the same location.
  - ...
- for multiple-processor system
  - Individual processors use the same ordering principles as in a single-processor system.
  - Writes by a single processor are observed in the same order by all processors.
  - Writes from an individual processor are NOT ordered with respect to the writes from other processors ...
  - Memory ordering obeys causality (memory ordering respects transitive visibility).
  - Any two stores are seen in a consistent order by processors other than those performing the store
  - *Locked instructions have a total order*

x86-TSO

- FIFO store buffer
- read = read the most recent buffered write, if it exists (else from main memory)
- buffered write: can propagate to shared memory at any time (except when lock is held by other threads).

  behavior of LOCK'ed instructions

  – obtain global lock
  – flush store buffer at the end
  – release the lock
  – note: no reading allowed by other threads if lock is held

SPARC V8 Total Store Ordering (TSO):
a read can complete before an earlier write to a different address, but a read cannot return the value of a write by another processor unless all processors have seen the write (it returns the value of *own* write before others see it)

Consequences: In a thread: for a write followed by a read (to different addresses) the order can be *swapped*

Justification: Swapping of $W - R$ is *not observable* by the programmer, it does not lead to *new, unexpected* behavior!

Example

thread	thread′
`flag := 1`	`flag′ := 1`
`A := 1`	`A := 2`
$\texttt{reg}_1$ `:= A`	$\texttt{reg}'_1$ `:= A`
$\texttt{reg}_2$ `:= flag′`	$\texttt{reg}'_2$ `:= flag`

Result?
In TSO[56]

- $(\texttt{reg}_1, \texttt{reg}'_1)$ `= (1,2)` observable (as in SC)
- $(\texttt{reg}_2, \texttt{reg}'_2)$ `= (0,0)` observable

Axiomatic description

- consider "temporal" ordering of memory commands (read/write, load/store etc)
- program order $<_p$:
  – order in which memory commands are issued by the processor
  = order in which they appear in the program code
- memory order $<_m$: order in which the commands become effective/visible in main memory

Order (and value) conditions

RR: $l_1 <_p l_2 \implies l_1 <_m l_2$

WW: $s_1 <_p s_2 \implies s_1 <_m s_2$

RW: $l_1 <_p s_2 \implies l_1 <_m s_2$

***Latest* write wins:** $val(l_1) = val(\max_{<_m}\{s_1 <_m l_1 \quad \vee \quad s_1 <_p l_1\})$

[56] Different from IBM 370, which also has write buffers, but not the possibility for a thread to read from its own write buffer

10.2 The ARM and POWER memory model

ARM and Power architecture

- ARM and POWER: similar to each other
- ARM: widely used inside smartphones and tablets (battery-friendly)
- POWER architecture = **P**erformance **O**ptimization **W**ith **E**nhanced **RISC**., main driver: IBM

  Memory model

  much weaker than x86-TSO
- exposes *multiple-copy* semantics to the programmer

"Message passing" example in POWER/ARM

thread_0 wants to pass a message over "channel" x to thread_1, shared var y used as flag.

Initially: `x = y = 0`

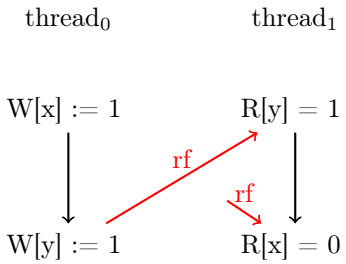
thread_0	thread_1
`x := 1`	`while (y=0) { };`
`y := 1`	`r := x`

Result?

Is the result `r = 0` observable?

- impossible in (x86-)TSO
- it would violate W-W order

Analysis of the example

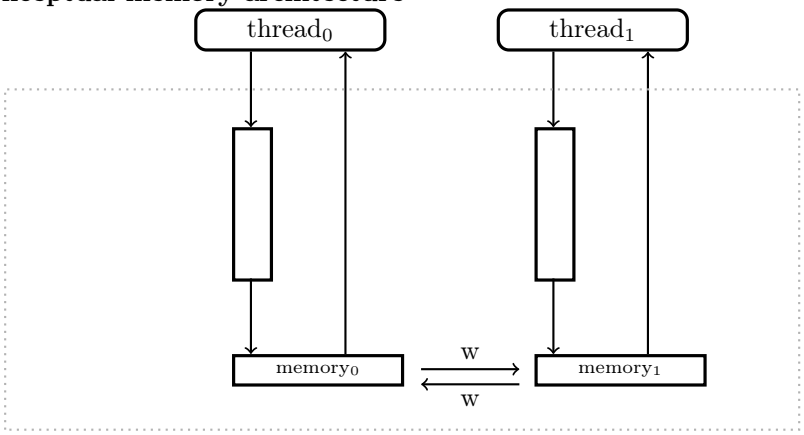


How could that happen?

1. thread does stores out of order
2. thread does loads out of order
3. store propagates between threads out of order.

Power/ARM do all three!

Conceptual memory architecture



Power and ARM order constraints

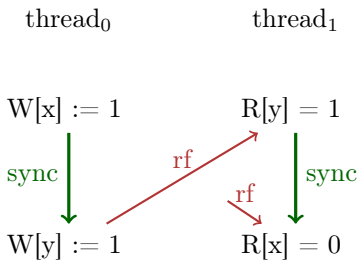
basically, program order is not preserved[57] (!) unless.

- writes to the same location
- address dependency between two loads
- dependency between a load and a store,
  1. address dependency
  2. data dependency
  3. control dependency
- use of *synchronization* instructions.

Repair of the MP example

To avoid reorder: Barriers

- heavy-weight: sync instruction (POWER)
- light-weight: lwsync



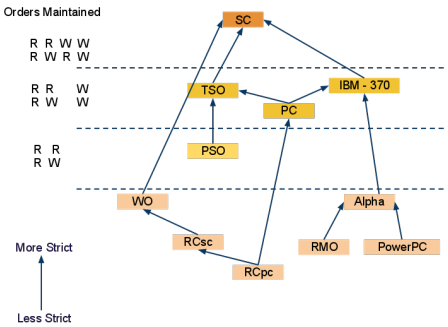
Stranger still, perhaps

thread$_0$	thread$_1$
`x := 1`	`print y`
`y := 1`	`print x`

Result?

Is the printout `y = 1, x = 0` observable?

Relationship between different models



(from http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_506_Spring_2013/10c_ks)

[57] in other words: "semicolon" etc is meaningless

10.3 The Java memory model

Java memory model

- known, influential example for a memory model for a programming language.
- specifies how Java threads interact through memory
- weak memory model
- under long development and debate
- original model (from 1995):
  - widely criticized as flawed
  - disallowing many runtime optimizations
  - no good guarantees for code safety
- more recent proposal: Java Specification Request 133 (JSR-133), part of Java 5
- see [Manson et al.,]

Correctly synchronized programs *and* others

1. Correctly synchronized programs: correctly synchronized, i.e., data-race free, programs are sequentially consistent ("*Data-race free*" model [Adve and Hill, 1990])
2. Incorrectly synchronized programs: A clear and definite semantics for incorrectly synchronized programs, without breaking Java's security/safety guarantees.

tricky balance for programs with data races:
disallowing programs violating Java's security and safety guarantees vs. flexibility still for standard compiler optimizations.

Data race free model

Data race free model
data race free programs/executions are sequentially consistent

Data race with a twist

- A data race is the "simultaneous" access by two threads to the same shared memory location, with at least one access a write.
- a program is race free if no execution reaches a race.
- a program is race free if no *sequentially consistent* execution reaches a race.
- note: the definition seems ambiguous!

Order relations

synchronizing actions: locking, unlocking, access to volatile variables

Definition 20. 1. synchronization order $<_{sync}$: total order on all synchronizing actions (in an execution)

2. synchronizes-with order: $<_{sw}$
   - an unlock action *synchronizes-with* all $<_{sync}$-subsequent lock actions by any thread
   - similarly for volatile variable accesses
3. happens-before ($<_{hb}$): transitive closure of program order and synchronizes-with order

Happens-before memory model

- simpler than/approximation of Java's memory model
- distinguising `volative` from non-`volatile` reads
- happens-before

Happens before consistency
In a given execution:

- if $R[x] <_{hb} W[X]$, then the read cannot observe the write
- if $W[X] <_{hb} R[X]$ and the read observes the write, then there does not exists a $W'[X]$ s.t. $W[X] <_{hb} W'[X] <_{hb} R[X]$

Synchronization order consistency (for `volatile`-s)

- $<_{sync}$ consistent with $<_p$.
- If $W[X] <_{hb} W'[X] <_{hb} R[X]$ then the read sees the write $W'[X]$

Incorrectly synchronized code

Initially: `x = y = 0`

thread_0	thread_1
`r1 := x`	`r2 := y`
`y := r1`	`x := r2`

- obviously: a race
- however:

  out of thin air

  observation `r1 = r2 = 42` not wished, but consistent with the happens-before model!

Happens-before: volatiles

- cf. also the "message passing" example

`ready` volatile
Initially: `x = 0, ready = false`

thread_0	thread_1
`x := 1`	`while (!ready) do skip`
`ready := true`	`r1 := x`

- `ready` volatile $\Rightarrow$ `r1` $= 1$ guaranteed

Problem with the happens-before model

Initially: `x = 0, y = 0`

thread_0	thread_1
`r1:= x`	`r2:= y`
`if (r1` $\neq$ `0)`	`if (r2` $\neq$ `0)`
`y := 42`	`x := 42`

- the program is *correctly synchronized!*

$\Rightarrow$ observation `y = x = 42` disallowed

- However: in the happens-before model, *this is allowed!*

violates the "data-race-free" model

$\Rightarrow$ add causality

Causality: second ingredient for JMM

JMM
Java memory model = happens before + causality

- circular causality is unwanted
- causality eliminates:
  - data dependence
  - control dependence

Causality and control dependency

Initially: `a = 0; b = 1`

thread$_0$	thread$_1$
`r1 := a`	`r3 := b`
`r2 := a`	`a := r3;`
`if (r1 = r2)`	
`  b := 2;`	

is $\texttt{r}_1 = \texttt{r}_2 = \texttt{r}_3 = 2$ possible?

$\Longrightarrow$

Initially: `a = 0; b = 1`

thread$_0$	thread$_1$
`b := 2`	`r3 := b;`
r1 := a	`a := r3;`
`r2 := r1`	
`if (true) ;`	

$\texttt{r}_1 = \texttt{r}_2 = \texttt{r}_3 = 2$ is sequentially consistent

Optimization breaks control dependency

Causality and data dependency

Initially: `x = y =0`

thread$_0$	thread$_1$
`r1 := x;`	`r3 := y;`
`r2 := r1∨1;`	`x := r3;`
`y := r2;`	

Is $\texttt{r}_1 = \texttt{r}_2 = \texttt{r}_3 = 1$ possible?

$\Longrightarrow$

Initially: `x = y = 0`

thread$_0$	thread$_1$
`r2 := 1;`	`r3 := y;`
y := 1	`x := r3;`
`r1 := x`	

using *global* analysis

$\vee$ = bit-wise or on integers

Optimization breaks data dependence

Summary: Un-/Desired outcomes for causality

Disallowed behavior

Initially: x = y = 0

$thread_0$	$thread_1$
r_1 := x	r_2 := y
y := r_1	x := r_2

[2em] $r_1 = r_2 = 42$

Initially: x = 0, y = 0

$thread_0$	$thread_1$
r_1 := x	r_2 := y
if ($r_1 \neq 0$)	if ($r_2 \neq 0$)
y := 42	x := 42

[2em] $r_1 = r_2 = 42$

Allowed behavior

Initially: a = 0; b = 1

$thread_0$	$thread_1$
r_1 := a	r_3 := b
r_2 := a	a := r_3;
if ($r_1 = r_2$)	
b := 2;	

is $r_1 = r_2 = r_3 = 2$ possible?

Initially: x = y =0

$thread_0$	$thread_1$
r_1 := x;	r_3 := y;
r_2 := $r_1 \vee 1$;	x := r_3;
y := r_2;	

Is $r_1 = r_2 = r_3 = 1$ possible?

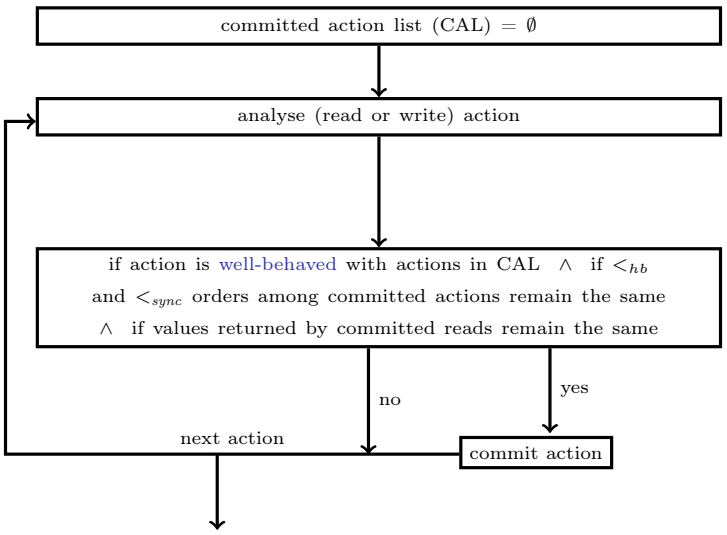
Causality and the JMM

- key of causality: well-behaved executions (i.e. consistent with SC execution)
- non-trivial, subtle definition
- writes can be done early for well-behaved executions

Well-behaved

a not yet commited read must return the value of a write which is $<_{hb}$.

Iterative algorithm for well-behaved executions



JMM impact

- considerations for implementors
  - control dependence: should not reorder a write above a non-terminating loop
  - weak memory model: semantics allow re-ordering,
  - other code transformations
    - synchronization on thread-local objects can be ignored
    - volatile fields of thread local obects: can be treated as normal fields
    - redundant synchronization can be ignored.
- Consideration for programmers
  - DRF-model: make sure that the program is correctly synchronized ⇒ don't worry about re-orderings
  - Java-spec: no guarantees whatsoever concerning pre-emptive scheduling or fairness

10.4 Go memory model

Go language and weak memory

- Go: supports shared var (but frowned upon)
- favors *message passing* (channel communication)
- "standard" modern-flavored WMM (like Java, C++11)
- based on *happens-before*
- specified in `https://golang.org/ref/mem` (in English)

Advice for average programmers[58] [Go memory model, 2016]

> "If you must read the rest of this document to understand the behavior of your program, you are being too clever.
>
> Don't be clever"

Go MM: Programs-order implies happens-before

program order [Go memory model, 2016]

> "Within a single goroutine, the happens-before order is the order expressed by the program."

- goroutine: Go-speak for thread/process/asynchronously executing function body/unit-of-concurrency

Allowed and guaranteed observability

May observation [Go memory model, 2016]
A read r of a variable v is *allowed to observe* a write w to v if both of the following hold:

1. r does not happen before w.
2. There is no other write w' to v that happens after w but before r.

Must observation [Go memory model, 2016]
r is *guaranteed to observe* w if both of the following hold:

1. w happens before r.
2. Any other write to the shared variable v either happens before w or after r.

Synchronization?

- so far: only statements without sync-power (reads, writes)
- without synchronization (and in WMM): concurrent programming impossible (beyond independent concurrency)
- a few synchronization statements in Go
  - initialization, package loads
  - Go goroutine start
  - via `sync`-package: locks and mutexes, once-operation
  - *channels*

[58] But of course participants of this course well-trained enough to make sense of the document.

Channels as communication and synchronization construct

- central in Go
- message passing: fundamental for concurrency
- cf. producer/consumer problem, bounded-buffer data structure, also *Oblig-1*

Role of channels:

Communication: one can transfer data from sender to receiver, but not only that:

Synchronization:

- receiver has to wait for value
- sender has to wait, until place free in "buffer"
- and: channels introduce "barriers"

- technically: *happens-before* relation for channel communication

Happens-before for send and receive

```
x := 1     |  y := 2
c!()       |  c?()
print y    |  print x
```

which read is guaranteed / may happen?

Message passing and happens-before

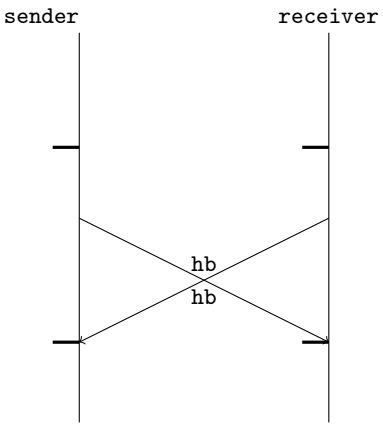
Send before receive [Go memory model, 2016]
"A send on a channel *happens before* the corresponding receive from that channel completes."

Receives before send [Go memory model, 2016]
"The kth receive on a channel with capacity C *happens before* the $k + C$th send from that channel completes."

Receives before send, unbuffered[Go memory model, 2016]
A receive from an unbuffered channel happens before the send on that

Happens-before for send and receive

```
x := 1     |  y:=2
c!()       |  c?()
print(y)   |  print x
```



Go memory model

- catch-fire / out-of-thin-air (≠ Java)
- standard: *DRF programs are SC*
- Concrete implementations:
  - more specific
  - platform dependent
  - difficult to "test"

```
[msteffen@rijkaard wmm] go run reorder.go
 1 reorders detected after 329 interations
 2 reorders detected after 694 interations
 3 reorders detected after 911 interations
 4 reorders detected after 9333 interations
 5 reorders detected after 9788 interations
 6 reorders detected after 9951 interations
 ...
```

11 Summary and conclusion

Memory/consistency models

- there are memory models for HW and SW (programming languages)
- often given informally/prose or by some "illustrative" examples (e.g., by the vendor)
- it's basically the semantics of concurrent execution with shared memory.
- interface between "software" and underlying memory hardware
- modern complex hardware ⇒ complex(!) memory models
- defines which compiler optimizations are allowed
- crucial for correctness and performance of concurrent programs

Conclusion

Take-home lesson
it's impossible(!!) to produce

- correct and
- high-performance

concurrent code without clear knowledge of the chosen platform's/language's MM

- that holds: not only for system programmers, OS-developers, compiler builders . . . but also for "garden-variety" SW developers
- reality (since long) much more complex than "naive" SC model

Take home lesson for the impatient
Avoid data races at (almost) all costs (by using synchronization)!

12 RPC and Rendezvous

Lecture 5. Nov. 2019

Outline

Today:

- remote procedure calls
  - concept, syntax, and meaning
  - examples: time server, merge filters, exchanging values
- rendezvous
  - concept, syntax, and meaning
  - examples: buffer, time server, exchanging values
- combinations of RPC, rendezvous and message passing
  - Examples: bounded buffer, readers/writers

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

Asynchronous agent systems: do not use channels, but

- two way communication gives many messages, and message names

⇒ many messages

RPC and rendezvous are better suited for client/server applications.

12.1 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.[59]
- 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.

[59] cf. RMI

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/procedures

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:[60]

call M.O(*arguments*)

Processes: as before.

Synchronization in modules

- RPC: primarily a *communication* mechanism
- within the module: in principle allowed:
  - more than one process
  - shared data

⇒ need for synchronization

- two approaches
  1. "implicit":
     - as in *monitors:* mutex built-in
     - additionally condition variables (or semaphores)
  2. "explicit":[61]
     - user-programmed mutex and synchronization (like semaphores, 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 of day")

[60] Cf. static/class methods
[61] assumed in the following

Time server code (RPC)

```
module TimeServer
  op get_time() returns int;
  op delay(int interval);
body
  int  tod := 0;          # time of day
  sem    m := 1;          # for mutex
  sem d[n] := ([n] 0);    # for delayed processes
  queue of (int waketime, int process_id) napQ;
  ## when m = 1, tod < waketime for delayed processes
  proc get_time() returns time { time := tod; }

  proc delay(int interval) {
    P(m);          # assume unique myid and i [0,n-1]
    int waketime := tod + interval;
      insert  (waketime, myid) at appropriate place in napQ;
    V(m);
    P(d[myid]);  # Wait to be awoken
  }
  process Clock ...
  .
  .
  .
end TimeServer
```

Time server code: clock process

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

12.2 Rendezvous

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:

- $O_i(v_i, \ldots v_{m_i})$ – name of operation and its formals
- 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.[62]

Semantics of input statement

Consider the following:

```
in ...
[] Oi(v_i, ..., v_{m_i}) and Bi -> Si;
...
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[63]
- Values are returned to the caller
- 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?

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

[63] this may be changed using additional syntax (`by`), see [Andrews, 2000].

- 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
```

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
```

Note: here the argument to delay is absolute time, not increment.
Note: the test "<=" implies that the caller needs to wait when tod is less than waketime.

RPC, rendezvous and message passing

We do now have several combinations:

invocation	*service*	*effect*
call	proc	procedure call (RPC)
call	in	rendezvous
send	proc	dynamic process creation, asynchronous proc. calling
send	in	asynchronous message passing

in addition (not in Andrews)

- asynchronous procedure call, wait-by-necessity, futures

Rendezvous, message passing and semaphores

Comparing input statements and receive:

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

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 using RPC and Rendezvous

```
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
```

13 Asynchronous Communication I

12. 11. 2019

Asynchronous Communication: Semantics, specification and reasoning

Where are we?

- part one: shared variable systems
  - programming
  - synchronization
  - reasoning by invariants and Hoare logic
- part two: communicating systems
  - message passing
  - channels
  - rendezvous

What is the connection?

- What is the semantic understanding of message passing?
- How can we understand concurrency?
- How to understand a system by looking at each component?
- How to specify and reason about asynchronous systems?

Overview

Clarifying the semantic questions above, by means of histories:

- describing interaction
- capturing interleaving semantics for concurrent systems
- Focus: asynchronous communication systems *without* channels

Plan today

- histories from the outside (global) view of components
  - describing overall understanding of a (sub)system
- Histories from the inside (local) view of a component
  - describing local understanding of a single process
- The connection between the inside and outside view
  - the composition rule

What kind of system? Agent or process network systems

Two flavors of message-passing concurrent systems, based on the notion of:

- processes — without self identity, but with named channels. Channels often FIFO. Examples: Go, CSP
- objects (agents) — with self identity, but without channels, sending messages to named objects through a network. In general, a network gives no FIFO guarantee, nor guarantee of successful transmission.

We next look at

- Go, briefly
- agent systems, in more detail including reasoning aspects.

Example of a language based on processes: Go

Language highlights

- design goals (/"marketing speak"?): efficiency, safety, scalability, concurrency
- here: *not a full intro* to Go

Concurrency in the Go language

- Syntax and environment adopting patterns.
- Fast compilation time.
- Remote package management.
- Online package documentation.
- Built-in concurrency primitives.
- Interfaces to replace virtual inheritance.
- Type embedding to replace non-virtual inheritance.
- Compiled into statically linked native binaries without external dependencies.
- Simple language specifications.
- Large built-in library.
- Light testing framework.

Channel comm. in Go

- no "named" sender or receiver: go-routines are anonymous
- instead Go uses channel names
- choice operator: `select`
- special syntax (of course):
  - `<- c` : receive over channel `c`
  - `c <- e` : send `e` over channel `c`
- similar non-determinism by select-case:

```
    select {                    // choice of alternatives
      case msg := <-c1:         // receive on c1 and store in msg
           ...
      case msg := <-c2:
           ...
      case msg := <-c3:
           ...
      default:                  // optional branch if
         ...                    // nothing else works
      }
}
```

Agent Systems
A High-Level Introduction

- syntax
- examples
- semantics and reasoning

Note: We focus on agent systems, 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.

Programming asynchronous agent systems
Statements for sending and receiving:

- *send statement:* `send` $A : m(e)$ means that the current agent sends message m to agent A where e is an (optional) list of actual parameters.
- *fixed receive statement:* `await` $A : m(w)$ wait for a message m from a specific agent A, and receive parameters in the variable list w. We say that the message is then consumed.
- *open receive statement:* `await` $X?m(w)$ wait for a message m from any agent X and receive parameters in w (consuming the message). The variable X will be set to the agent that sent the message.
- *choice operator* $[\,]$ to select between alternative statement lists, each starting with a receive statement, similar to **select** in Go ("angelic choice")

Here m is a message name, A the name of an agent, X an agent reference, e an expression (list), and w a variable (list).

Async. communication constructs

Syntax (s statement list, e expression/expression list)

s	::=	`send` $A : m(e)$	send to A
	|	`await` $A : m(w)$	receive from A
	|	`await` $X?m(w)$	receive from someone
	|	`await` $?m(w)$	annonymous receive
	|	$s\,[\,]\,s$	choice
	|	$x := e \mid s;s \mid (s)$	assign., seq.composition
	|	$m(e)$	local call to method m
	|	`if` e `then` s `else` s `fi`	if statement
	|	`while` e `do` s `od` | `loop` s `end`	loop statements

Example: Coin machine (from Exam 05)

Consider an agent C which changes "5 krone" coins and "1 krone" coins into "10 krone" coins. It receives *five* and *one* messages and sends out *ten* messages as soon as possible, in the sense that the number of messages sent out should equal the total amount of kroner received divided by 10.

We imagine here a fixed user agent U, both producing the *five* and *one* messages and consuming the *ten* messages. The code of the agent C is given below, using b (*balance*) as a local variable initialized to 0.

Example: Coin machine (Cont)

```
loop
  while b < 10
  do
    (await U:five; b:=b+5)
  []
    (await U:one;  b:=b+1)
  od;
  send U:ten;
  b:=b-10              // use b in next iteration
end
```

- the choice operator [ ][64]
  - selects an *enabled* branch, if any (and otherwise waits)
  - non-deterministic choice if both branches are enabled

Interleaving semantics of concurrent systems

- The behavior of a concurrent system: may be described as a set of executions,
- each execution: sequence of atomic communication events,
- other names for it: trace, history, execution, (interaction) sequence — similar to message sequence (charts) in UML

  Interleaving semantics

  Concurrency is expressed by the set of all *possible interleavings*.

- note: for each interaction sequence, all interactions are ordered sequentially, and showing their "visible" concurrent actions

Regular expressions (to express traces)

- well known and widely used "format" to describe "languages" (= sets of finite "words" over given a given "alphabet")
- formed by
  - [...] (meta-parentheses)
  - superscript * (repetition) as in $[..]^*$
  - superscript + (nonempty repetition) as in $[..]^+$
  - + (choice) [65]
  - ; (sequential composition)

Examples:

$$[a; b]^*$$

$$a + [a; b]$$

$$[a]^* + [b]^*$$

$$[a]^+ + [b]^*$$

[64] In the literature, also + as notation can often be found. [ ] taken because of "ASCII" version of □, which can be found in publications.

[65] Note: we do not use | for choice, to not confuse with parallel composition.

A way to describe (sets of) traces

Example 21 (Reg-Expr). • a, b: atomic interactions.

- Assume them to "run" concurrently

$\Rightarrow$ two possible interleavings, described by

$$[[a; b] + [b; a]] \tag{4}$$

Parallel composition of a^* and b^*:

$$[a + b]^* \tag{5}$$

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[66]
- *non-terminating* systems: infinite traces
- trace set semantics in the general case: both finite and infinite traces
- 2 conceptually important classes of properties[67]
  - 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 run-time monitoring/run-time verification
  - connection to checking (violations of) safety prop's
- our terminology: history = trace up to a *given execution point* (thus finite)
- Note: In contrast to the book, histories are here finite initial parts of a trace (prefixes)
- sets of histories are **prefix closed**:

  if a history h is in the set, then every prefix (initial part) of h is also in the set.

Sets of histories: can be used capture safety, but not liveness

Simple example: histories and trace set

Consider a system of two agents, A and B, where agent A says "hi-B" repeatedly until B replies "hi-A".

- "sloppy" B: may or may not reply, in which case there will be an infinite trace with only "hi-B" (here comma denotes $\cup$).

  Trace set: $\{[hi_B]^\infty\}, \{[hi_B]^+; [hi_A]\}$ Histories: $\{[hi_B]^*\}, \{[hi_B]^+; [hi_A]\}$
- "lazy" B: will reply eventually, but no limit on how long A may need to wait. Thus, each trace will end with "hi_A" after finitely many "hi_B"'s. Trace set: $\{[hi_B]^+; [hi_A]\}$ Histories: $\{[hi_B]^*\}, \{[hi_B]^+; [hi_A]\}$
- an "eager" B will reply within a fixed number of "hi_B"'s, for instance before A says "hi_B" three times. Trace set: $\{[hi_B]; [hi_A]\}, \{[hi_B]; [hi_B]; [hi_A]\}$ Histories: $\{\epsilon\}, \{[hi_B]\}, \{[hi_B]; [hi_A]\}, \{[hi_B]; [hi_B]\}, \{[hi_B]; [hi_B]; [hi_A]\}$

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

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

Histories = sequences of communication events

We use the following conventions

- communication events $a : Event$ is an event
- set of communication events: A (i.e. $2^{\#Event}$)
- history $h : Hist$ – sequence of events

For a given program, the set of (communication) events $Event$ is assumed to be fixed.

Definition 22 (Histories)**.** Histories (over a given set of events) are defined inductively over the constructors ϵ (empty history) and $\_;\_$ (appending of an event to the right of the history)

Functions over histories (for specification purpose)

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

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

$\#\epsilon$	$= 0$
$\#(h;x)$	$= \#h + 1$
ϵ/A	$= \epsilon$
$(h;x)/A$	$=$ `if` $x \in A$ `then` $(h/A);x$ `else` (h/A) `fi`
$h \preceq h'$	$= (h = h') \vee h \prec h'$
$h \prec \varepsilon$	$= false$
$h \prec (h';x)$	$= h \preceq h'$

Invariants and Prefix Closed Trace Sets

May use *invariants* to define trace sets:

A (history) invariant I is a predicate over a histories, supposed to hold at all times:

"At any point in an execution h the property $I(h)$ is satisfied"

It defines the following set:

$$\{h \mid I(h)\} \tag{6}$$

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

$$h \leq h' \Rightarrow (I(h') \Rightarrow I(h)) \tag{7}$$

- I history-monotonic $\Rightarrow$ the set from equation (6) is 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

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

C – coin machine
U – user

Legal histories

- not all global sequences/histories "make sense"
- depends on the programming language/communciation model
- sometimes called well-definedness, well-formedness or similar
- $legal : Hist \to Bool$

Definition 23 (Legal history)**.**

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

where m is message and h a history.

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

Example of legal history (coin machine C and user U):

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

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 message
- for each execution of a receive statement in B, update H by

$$H := H;(A{\downarrow}B{:}m)$$

  where m is the message and A the sender. The message must be of the kind requested by B.

Outside View: Global Properties

specify desired system behavior by predicate I on the global history,

Global invariant

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

Enforcement

- run-time logging the history: monitor an executing system. When $I(H)$ is violated we may
  - report it
  - stop the system, or
  - interact with the system (for inst. through fault handling)
- How to prove such properties by analysing the program?
- How can we monitor, or prove correctness properties, component-wise ?

Semantics: Inside view: Local histories

Definition 24 (Local events)**.** Events visible to an agent A, written α_A = the events local to A:

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

Definition 25 (Local history)**.** Given a global history: The local history of A, written h_A, is the subsequence of all events visible to A. (Inside A we simply write h.)

- Correspondence between global and local view:

$$h_A = H/\alpha_A$$

  i.e. at any point in an execution the history observed locally in A is the projection to A -events of the history observed globally.

- Note: Each event is visible to one, and only one, agent! This will allow compositional reasoning.

Coin Machine Example: Local Events

The events *visible to* C are:

$U{\downarrow}C{:}five$	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.

"program 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.

***Composition* rule for histories: from inside to outside view**

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

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

i.e., the global history H must be legal and correspond to all the local histories. This may be used to reason about the global history.

Local invariant A_i: a local specification of A_i is given by a predicate on the local history $I_{A_i}(h_{A_i})$ describing a property which is maintained by all local interaction points. I_{A_i} may have the form of an implication, expressing that the output events from A_i depends on a condition on its input events.

Composition rule from local invariants to a global invariant: if each agent satisfies $I_{A_i}(h_{A_i})$, the total system will satisfy the global invariant:

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

Example revisited: Coin Machine.

Let us fucus on C

```
loop
  while b < 10
  do
    (await U:five; b:=b+5)
  []
    (await U:one;  b:=b+1)
  od;
  send U:ten;
  b:=b-10              // use b in next iteration
end
```

Coin Machine:

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

$U{\downarrow}C{:}five$ -- 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 for C

Loop invariant for the outer loop:

$$sum(h/\downarrow) = sum(h/\uparrow) + b \ \wedge \ 0 \leq b < 5 \tag{8}$$

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:

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

Note: $h/\uparrow$ denotes the subsequence of messages sent (by C) **Note:** $h/\downarrow$ denotes the subsequence of messages received (by C)

Coin machine example: from local to global invariant

before each send/receive: (see eq. (19))

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

Local Invariant of C in terms of h *alone:*

$$I_C(h) = \exists b.\ (sum(h/\downarrow) = sum(h/\uparrow) + b \ \wedge \ 0 \leq b < 15) \tag{10}$$

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

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

$$I_C(H/\alpha_C) \Leftrightarrow 0 \leq sum(H/\alpha_C/\downarrow) - sum(H/\alpha_C/\uparrow) < 15 \tag{12}$$

This gives the global invariant:

$$0 \leq sum(H/\downarrow C) - sum(H/C\uparrow) < 15$$

Coin machine user U: 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:

$$I(H) = legal(H) \wedge I_C(H/\alpha_C) \wedge I_U(H/\alpha_U) \tag{13}$$

  implying:

  Overall global invariant:

$$0 \leq sum(H/U\downarrow C) - sum(H/C\uparrow U) \leq sum(H/U\uparrow C) - sum(H/C\downarrow U) \leq 10$$

  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:

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

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

Loop invariant for the inner loop:

$$rec(h) = sent(h) + b \ \wedge \ 0 \leq b < 15$$

Final Remarks

Remark 3 (Self-communication may be considered internal:)**.** *In "black-box" specifications, we consider observable events only, abstracting away from internal events. Then, the alphabeth of an agent A is restricted by requiring the other agent to be different from A:*

$$\alpha_A = \{A{\uparrow}B{:}m \texttt{ where } A \neq B\} \cup \{B{\downarrow}A{:}m \texttt{ where } A \neq B\}$$

Remark 4 (FIFO/non-FIFO ordering:)**.** *The 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).

Using Legality to Model Network Properties

If the network delivers messages in a FIFO fashion, one could capture this by strengthening the legality-concept suitably, requiring

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

where the projections $h/\uparrow$ and $h/\downarrow$ denote the subsequence of messages sent and received, respectively, and *sendevents* converts receive events to the corresponding send events.

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

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

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

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

14 Asynchronous Communication II

19.11.2019

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

Async. communication constructs

Syntax (s statement list, e expression/expression list)

s	$::= \underline{\texttt{send}}\ A : m(\vec{e})$	send to A
	$\mid \underline{\texttt{await}}\ A : m(\vec{x})$	receive from A
	$\mid \underline{\texttt{await}}\ X?m(\vec{x})$	receive from someone
	$\mid \underline{\texttt{await}}\ ?m(\vec{x})$	annonymous receive
	$\mid s\,[\,]\,s$	choice
	$\mid x := e \mid s; s \mid (s)$	assign., seq.composition
	$\mid m(\vec{e})$	local call to method m
	$\mid \underline{\texttt{if}}\ e\ \underline{\texttt{then}}\ s\ \underline{\texttt{else}}\ s\ \underline{\texttt{fi}}$	if statement
	$\mid \underline{\texttt{while}}\ e\ \underline{\texttt{do}}\ s\ \underline{\texttt{od}} \mid \underline{\texttt{loop}}\ s\ \underline{\texttt{end}}$	loop statements

Here $\vec{e}$ is a list of expressions, $\vec{x}$ a list of variables, A an agent expression, X an agent variable.
NB: $X, \vec{x}$ must be distinct variables.

Local reasoning by Hoare logic (a.k.a program logic)

We adapt Hoare logic to reason about local histories in an agent A:

- Introducing a local (logical) variable h, initialized to empty ϵ
  - h represents the *local* history of A
- For **send**/**await**-statement: define the effect on h.
  - extending the h with the corresponding communication event
- *Local reasoning*: we do not know the global invariant
  - For **await**: unknown parameter values
  - For *open receive*: unknown sender

  $\Rightarrow$ use non-deterministic assignment

$$x := \underline{\texttt{some}} \tag{14}$$

Local invariant reasoning by Hoare Logic

- each send statement $\underline{\texttt{send}}\ B : m(\vec{e})$ in A is treated as:

$$h := (h; A{\uparrow}B{:}m(\vec{e})) \tag{15}$$

- each fixed receive statement $\underline{\texttt{await}}\ B : m(\vec{x})$ in A is treated as

$$\vec{x} := \underline{\texttt{some}}; h := (h; B{\downarrow}A{:}m(\vec{x})) \tag{16}$$

  the usage of $\vec{x} := \underline{\texttt{some}}$ expresses that A may receive any values for the receive parameters

- each open receive statement $\underline{\texttt{await}}\ X?m(\vec{x})$ in A is treated as

$$X := \underline{\texttt{some}}; \underline{\texttt{await}}\ X : m(\vec{x}) \tag{17}$$

  i.e., $X := \underline{\texttt{some}}; \vec{x} := \underline{\texttt{some}}; h := (h; B{\downarrow}A{:}m(\vec{x}))$, where $X := \underline{\texttt{some}}$ expresses that A may receive the message from any agent.

Rule for non-deterministic assignments

Non-det. assignment

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

- 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

$$\frac{}{\{\ Q[h; A{\uparrow}B{:}m(\vec{e})/h]\ \}\ \texttt{send}\ B : m(\vec{e})\ \{\ Q\ \}}\ \textsc{Send}$$

$$\frac{}{\{\ \forall \vec{x}\ .\ Q[h; B{\downarrow}A{:}m(\vec{x})/h]\ \}\ \texttt{await}\ B : m(\vec{x})\ \{\ Q\ \}}\ \textsc{Receive}_1$$

$$\frac{}{\{\ \forall \vec{x}, X\ .\ Q[h; X{\downarrow}A{:}m(\vec{x})/h]\ \}\ \texttt{await}\ X?m(\vec{x})\ \{\ Q\ \}}\ \textsc{Receive}_2$$

- As before: A is the current agent/object, h the local history
- We assume that neither B nor X occur in $\vec{x}$, and that $\vec{x}$ is a list of distinct variables (which is a static check)
- No shared variables $\Rightarrow$ no interference, and Hoare reasoning can be done as usual in the sequential setting!
- Simplified version, if no parameters in await:

$$\frac{}{\{\ Q[h; (B{\downarrow}A{:}m)/h]\ \}\ \texttt{await}\ B : m\ \{\ Q\ \}}\ \textsc{Receive}$$

Hoare rules for local reasoning

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

Rule for Choice ([ ])

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

Remark: Similarly we may reason backwards over conditionals:[68]

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

Example: Coin machine (from Exam 05)

Consider an agent C which changes "5 krone" coins and "1 krone" coins into "10 krone" coins. It receives *five* and *one* messages and sends out *ten* messages as soon as possible, in the sense that the number of messages sent out should equal the total amount of kroner received divided by 10.

We imagine here a fixed user agent U, both producing the *five* and *one* messages and consuming the *ten* messages. The code of the agent C is given below, using b (*balance*) as a local variable initialized to 0.

[68] 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).

Example: Coin machine (Cont)

```
loop
  while b < 10
  do
    (await U:five; b:=b+5)
  []
    (await U:one;  b:=b+1)
  od;
  send U:ten;
  b:=b-10              // use b in next iteration
end
```

- the choice operator [ ]
  - selects an *enabled* branch, if any (and otherwise waits)
  - non-deterministic choice if both branches are enabled

Coin Machine Example: Events

$U{\uparrow}C{:}five$ -- U sends the message "five" to C
$U{\downarrow}C{:}five$ -- C consumes the message "five"

$U{\uparrow}C{:}one$ -- U sends the message "one to C
$U{\downarrow}C{:}one$ -- C consumes the message "one"

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

C – coin machine
U – user

Coin machine: local events

Invariants may refer to the local history h, which is the sequence of events visible to C that have occurred so far.

The events visible to C are:

$U{\downarrow}C{:}five$ -- 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 for C

Loop invariant for the outer loop:

$$sum(h/\downarrow) = sum(h/\uparrow) + b \ \wedge \ 0 \leq b < 5 \tag{18}$$

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:

$$sum(h/\downarrow) = sum(h/\uparrow) + b \ \wedge \ 0 \leq b < 15 \tag{19}$$

Note: $h/\uparrow$ denotes the subsequence of messages sent (by C) **Note:** $h/\downarrow$ denotes the subsequence of messages received (by C)

Inner loop

let I_i ("inner invariant") abbreviate the last invariant (19).

```
{ I_i }
while b < 10   { b < 10 ∧ I_i }
{ (I_i[(b+5)/b])[h;U↓C:five/h] ∧ (I_i[(b+1)/b])[h;U↓C:one/h] }
do { (I_i[(b+5)/b])[h;U↓C:five/h] }
     ( await U:five;   { I_i[b+5/b] }
       b:=b+5 )    { I_i }
[] { (I_i[(b+1)/b])[h;U↓C:one/h] }
   (await U:one;   b:=b+1)
   { I_i }
od;
```

Must prove the implication:

$$b < 10 \wedge I_i \Rightarrow (I_i[(b+5)/b])[h;U{\downarrow}C:five/h] \wedge (I_i[(b+1)/b])[h;U{\downarrow}C:one/h]$$

Note: From precondition I_i for the loop, we have $I_i \wedge b \geq 10$ as the postcondition to the inner loop.

Outer loop

```
{ I_o }
loop
   { I_o }
   { I_i }
   while b < 10   { b < 10 ∧ I_i }
   { (I_i[(b+5)/b])[h;U↓C:five/h] ∧ (I_i[(b+1)/b])[h;U↓C:one/h] }
   do { (I_i[(b+5)/b])[h;U↓C:five/h] }
        ( await U:five;   { I_i[b+5/b] }
          b:=b+5 )    { I_i }
   [] { (I_i[(b+1)/b])[h;U↓C:one/h] }
      (await U:one;   b:=b+1)
      { I_i }
   od;
   { I_i ∧ b ≥ 10 }
   { (I_o[b−10/b])[h;C↑U:ten/h] }
   send U:ten;
   { I_o[b−10/b] }
   b:=b−10
   { I_o }
end
```

Combining forward and backward verification, meeting in the middle.

Outer loop (2)

Verification conditions (as usual):

- before inner loop:
$$I_o \Rightarrow I_i$$
- after inner loop:
$$I_i \wedge b \geq 10 \Rightarrow (I_o[(b-10)/b])[h;C{\uparrow}U:ten/h]$$
- I_o holds initially since
$$h = \varepsilon \wedge b = 0 \Rightarrow I_o$$

Local history invariant

For each agent (A):

- Predicate $I_A(h)$ over the local communication history (h)
- Describes interactions between A and the surrounding agents
- Must be maintained by *all* history extensions in A
- Last week: Local history invariants for the different agents may be composed, giving a global invariant

Verification by 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, it suffices to ensure:

$$Q(\vec{x}, h) \Rightarrow I_A(h) \tag{20}$$

- We may assume that the local invariant is satisfied immediately *before* each send/receive point.
- we may also assume that the *last* event of h is the send/receive event.

This gives an improved strategy:

Proving the local history invariant

The improved strategy for proving that each send/receive statement occurring in agent A maintains the local invariant $I_A(h)$:

$$(\ h = (h'; a) \wedge I_A(h') \wedge Q(\vec{x}, h)\ ) \Rightarrow I_A(h) \tag{21}$$

where

- a is the event corresponding to the send/receive statement
- first conjunct $h = (h'; a)$: specifies that a is the last communication event
- $I_A(h')$: the assumption that invariant holds before event a
- $Q(\vec{x}, h)$ is the condition (on local variables $\vec{x}$ and h) right after the send/receive 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 8 from last week):

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

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

- Hoare analysis of the outer loop gave the condition $I_o[(b-10)/b]$ immediately after the send statement
- history *ends* with the event $C{\uparrow}U{:}ten$

$\Rightarrow$ Verification condition, corresponding to equation (21):

$$h = h'; (C{\uparrow}U{:}ten) \wedge I_C(h') \wedge I_o[(b-10)/b] \Rightarrow I_C(h) \tag{22}$$

Coin machine example: local history invariant

Expanding I_C and I_o in the VC from equation (22), gives:

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

Note: Gray parts not needed here.

Coin Machine Example: Summary

Correctness proofs (bottom-up):

- code
- loop invariants (Hoare analysis)
- local history invariant
- verification of local history invariant based on the Hoare analysis

Note: The choice 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 card number, x pin code, and y cash amount, provided by clients.

Example of unsuccessful cycle: $[\ C{\updownarrow}M : card\_in(n); M{\updownarrow}C : pin; C{\updownarrow}M : pin(x);\quad M{\updownarrow}C : amount; C{\updownarrow}M : amount(y); M{\updownarrow}B : M{\updownarrow}C : card\_out\ ]$

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 (by projection to the relevant alphabet):

The successful cycle:

- Client: $[C{\uparrow}M{:}card\_in(n); M{\downarrow}C{:}pin; C{\uparrow}M{:}pin(x);\quad M{\downarrow}C{:}amount; C{\uparrow}M{:}amount(y); M{\downarrow}C{:}cash(y); M{\downarrow}C{:}card\_out]$
- Mini Bank: $[C{\downarrow}M{:}card\_in(n); M{\uparrow}C{:}pin; C{\downarrow}M{:}pin(x);\quad M{\uparrow}C{:}amount; C{\downarrow}M{:}amount(y); M{\uparrow}B{:}request(n,x,y); B{\downarrow}M{:}grant; M{\uparrow}C{:}cash(y); M{\uparrow}C{:}card\_out]$
- 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)

Example of an 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 card numbers
bal -- array of account balances, indexed by card numbers

X : Agent, n:  Card_Number, 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 here 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** $[a;b]^*$ (but it is a prefix)

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

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

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 $\{h$ **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
$\{h$ **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 h''_1 is Cycle_B else h''_2 is Cycle_B }
  await X?request(n,x,y);
    { if pin[n] = x ∧ bal[n] > y then h'_1 is Cycle_B else h'_2 is Cycle_B }
  if    pin[n]=x and bal[n]>y
  then  bal[n]:=bal[n]-y;
        { (h; B↑X:grant) is Cycle_B }
        send X:grant;
  else  { (h; B↑X:deny) is Cycle_B }
        send X:deny;
  fi
  { h is Cycle_B }
end
```

$$\begin{array}{rcl} h''_1 & = & h; X{\downarrow}B{:}request(n,x,y); B{\uparrow}X{:}grant \\ h'_1 & = & h; B{\uparrow}X{:}grant \end{array}$$

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 following general rule (where R is a regExp and a and b are events):

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

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:

$$h \leq Cycle_B$$

where $h \leq R$ denotes that h is a prefix of a regular expression in R.

Some intuitive laws:

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

Central Bank: Verification of the local history invariant

$h \leq Cycle_B$

- As before, we need to ensure that the history invariant is implied after each send/receive statement.
- Here it is enough to assume the conditions after each send/receive statement in the verification of the loop invariant

This gives 2 proof conditions:

1. after send grant/deny (i.e. after **fi**)

h **is** $Cycle_B \Rightarrow h \leq Cycle_B$ which is trivial.

2. after await request

if ... **then** $(h; B{\uparrow}X{:}grant)$ **is** $Cycle_B$ **else** $(h; B{\uparrow}X{:}deny)$ **is** $Cycle_B$

$\Rightarrow h \leq Cycle_B$ which follows from $(h; a)$ **is** $R \Rightarrow h \leq R$.

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

Mini bank example: Local invariant of Client (C)

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

History invariant:

$$h_C \leq Cycle_C$$

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.

Note: The client is therefore modeled as highly non-deterministic.

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 from cycle to 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** $M{\updownarrow}B : request(n, x, y); [B{\updownarrow}M : deny; M{\updownarrow}C : card\_out + B{\updownarrow}M : grant; M{\uparrow}C{:}cash(y); [M{\downarrow}C{:}cash(y) \;|||\; M{\updownarrow}C : card\_out]]$ **fi** **some** n, x, y]*

where ||| gives all possible interleavings.

Note: 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, delayed messages/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
- we 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

15 Wrapping it up : Exams Examples

Fall, 2019

15.1 Exam 2012

Exam 2012: Snow-globes

- 4 different tasks
- precedence graph
- semaphore solution

```
process MaterialCollector[i=1 to N]{
    while (true){
    ...
    ''collect a group of materials (glass, fake snow and ceramic)''
    ...
  }
}
process GiftAssembler[i=1 to N]{
  while (true){
    ...
    ''hand-make a snow-globe''
    ...
  }
}
process BoxMaker[i=1 to N]{
  while (true){
    ...
    ''hand-make a box''
    ...
  }
}
process GiftWrapper[i=1 to N]{
  while (true){
    ...
    ''wrap a snow-globe''
    ...
  }
}
```

```
sem materials = 0, gift = 0, box = 0;

process MaterialCollector[i=1 to N]{
    while (true){
    ''collect a group of materials (glass, fake snow and ceramic)''
    V(materials);
  }
}
process GiftAssembler[i=1 to N]{
  while (true){
```

```
    P(materials);
    ''hand-make a snow-globe''
    V(gifts);
  }
}
process BoxMaker[i=1 to N]{
  while (true){
    ''hand-make a box''
     V(boxes);
  }
}
process GiftWrapper[i=1 to N]{
  while (true){
    P(gifts);
    P(boxes);
     ''wrap a snow-globe''
  }
}
```

Gift storage

- *N gift organizers*, 1 transporter
- capacity G

```
Process GiftOrganizer[i=1 to N] {
  while (true)  call Gift_Storage.put();
}

process Transporter{
  while (true)  call Gift_Storage.transport();
}
```

Monitors

```
monitor Gift_Storage {
  int g = G; # capacity of the sledge
  int counter = 0; # number of gifts in the sledge
  ...
  procedure put() {
    ...
    ''Put gift into the sledge''
    ...
   }

  procedure transport() {
    ...
    ''move sledge into the storage room, download gifts
      and return''
    ...
  }
}
```

Monitors (1)

```
Process GiftOrganizer[i=1 to N] {
  while (true)  call Gift_Storage.put(); }

process Transporter{
  while (true) call Gift_Storage.transport(); }

monitor Gift_Storage {
  int g = G; # capacity of the sledge
  int counter = 0; # number of gifts in the sledge
  cond queue; # waiting queue

  procedure put() {
    while(counter == g) wait(queue);
    ''Put gift into the sledge''
    counter = counter + 1;
    signal_all(queue);
  }

  procedure transport() {
    while(counter < g) wait(queue);
    ''move sledge into the storage room, download gifts and return''
    counter = 0;
    signal_all(queue);
  }
}
```

Monitors (2)

```
process GiftOrganizer[i=1 to N] {
  while (true)  call Gift_Storage.put(); }

process Transporter{
  while (true)  call Gift_Storage.transport(); }

monitor Gift_Storage {
  int g = G;  # capacity of the sledge
  int counter = 0; # number of gifts in the sledge
  cond queueO; # waiting queue for GiftOrganizerts
  cond queueT; # waiting queue for Transporter

  procedure put() {
    while(counter >= g) wait(queueO);
    ''Put gift into the sledge''
    counter = counter + 1;
    if(counter == g) signal(queueT);
  }
  procedure transport() {
    while(counter < g) wait(queueT);
    ''move sledge into the storage room, download gifts and return''
    counter = 0;
    signal_all(queueO);
   }
 }
```

Java

```
public class Gift_Storage {
    int g = G;
    int counter = 0;


    public synchronized void put()
                        throws InterruptedException{
        while(counter==g) wait();
        ++count;
        notifyAll();
    }

    public synchronized void transport()
                        throws InterruptedException{
        while(count<g) wait();
        count = 0;
        notifyAll();
    }
}
```

Async. message passing

- same problem, 4 agents

invariant

```
    X : Agent;  // assumed initialized to the MaterialCollector
    Y : Agent;  // assumed initialized to the GiftWrapper

  while true do
    await X:materialIsCollected;
    // ''hand-make a snow-globe''
    send Y:giftIsAssembled;
  od
```

- calculate

predicate over the history

$$isBeingAssembled(\varepsilon) = 0$$
$$isBeingAssembled(h; X \downarrow A : materialIsCollected) = isBeingAssembled(h) + 1$$
$$isBeingAssembled(h; A \uparrow Y : giftIsAssembled) = isBeingAssembled(h) - 1$$

Program analysis

- "verify" the program $\equiv$ verify the invariant.
- use Hoare-logic + rules (given)
- sequential problem

while *true* **do** $\{isBeingAssembled(h) = 0\}$
$\{isBeingAssembled(h) + 1 - 1 = 0\}$
$\{isBeingAssembled(h; X \downarrow A : materialIsCollected) - 1 = 0\}$
await $X : materialIsCollected$;
$\{isBeingAssembled(h) - 1 = 0\}$
// ‘‘hand-make a snow-globe’’
$\{isBeingAssembled(h; A \uparrow Y : giftIsAssembled) = 0\}$
send $Y : giftIsAssembled$;
$\{isBeingAssembled(h) = 0\}$
od

Note: It's mechanical, best to work from back to front.

Gift wrapper

```
    while true do
      ...
      //  ‘‘wrap a snow-globe’’
      ...
   od
```

- *random order*

```
X : Agent;  // assumed initialized to the GiftAssembler
Y : Agent;  // assumed initialized to the BoxMaker

while true do
  (await X:giftIsAssembled;
    await Y:boxIsMade)
    []
  (await Y:boxIsMade;
    await X:giftIsAssembled)

  //  ‘‘wrap a snow-globe’’
od
```

15.2 2011

Exam 2011: Ice Cream shop

- ice cream = 2 flavors
- 2 service guys
- 2 solutions required:
  - semaphore
  - RPC/rendevouz

skeleton

```
...                    # global variables
int scoops[N] = ([N] 0);

process Customer[i = 0 to N - 1]{
  while(true){
    ...
    scoops[i] = 0;  #eat
    ...
  }
}

process ChocolateMaker{
  while(true){
    ...
    scoops[...] = scoops[...] + 1  #make scoop
    ...
  }
}

process VanillaMaker{
  while(true){
    ...
    scoops[...] = scoops[...] + 1  #make scoop
```

```
    ...
  }
}
```

Skeleton

```
module IceCreamBar
  op getIceCream()                    // will be a procedure

body IcecreamBar
  proc getIceCream(){
    call startChock();
    #make the scoop
    send doneChock();
    call startVanilla();
    #make the scoop
    send doneVanilla();
  }

  process chocController{
    bool availChock = true;
    in startChoc() and availChock -> availChock = false ;
    [] doneChock -> availChock = true;
    ni
  }

  process vanillaController{
    bool availVanilla = true;
    in startVanilla() and availVanilla -> availVanilla=false;
    [] doneVanilla -> availVanilla = true;
    ni
  }
end
```

Skeleton: a comment

The solution has 2 processes to serve, like in the previous one, except that they are now *internalized*.

Also the *order* from last time is preserved: first the chocolate is made and then the vanilla.

This time it's not done by the customer himself, but as part of the procedure.

Analysis

program ("statement") `S`=

```
x = 0; y = 10;
while (x < y) {
  x = x+1; y = y-1;
}
```

- partial correctness

$$\{\texttt{true}\}S\{x == y\}$$

$$I : x \leq y \wedge even(y - x)$$

- "strengthening"?

Roaller-coaster

```
// ------------------- Passenger -----------------
C : Car; // assumed initialized to the Car agent

while true do
  ...
  send C:embark;
  await C:finished;
od
// ------------------- Car -------------------
pass : Stack[Agent]  // assumed initialized to empty stack
P : Agent

while true do
  while (size(pass) < 4) do
    await P?embark;
    pass := push(pass,P)
  od
  // ride!
  while (size(pass) > 0) do
    P := top(pass); pass := pop(pass);
    send P:finished
  od
od
```

15.3 2010

Exam 2010: House Building

- 3 tasks:
  1. Floor
  2. Walls
  3. Roof
- 2 solutions:
  - Semaphores
  - Monitor

Skeleton

```
process FloorBuilder[i=1 to N]{
  while (true){
    ...
        ‘‘build floor’’
        ...
    }
}

process WallBuilder[i=1 to N]{
    while (true){
        ...
        ‘‘build walls’’
        ...
    }
}

process RoofBuilder[i=1 to N]{
    while (true){
        ...
        ‘‘build roof’’
        ...
    }
}
```

Semaphores

```
sem floor = 0, walls = 0;

process FloorBuilder[i=1 to N]{
      while (true){
        ‘‘build floor’’
        V(floor);
    }
}

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

```
        P(floor);
        ‘‘build walls’’
        V(walls);
    }
}

process RoofBuilder[i=1 to N]{
    while (true){
        P(walls);
        ‘‘build roof’’
    }
}
```

Monitor

```
process FloorBuilder[i=1 to N]{
    ‘‘build floor’’
    Clerk.doneFloor();
}

process WallBuilder[i=1 to N]{
    Clerk.startWalls();
    ‘‘build walls’’
    Clerk.doneWalls();
}

process RoofBuilder[i=1 to N]{
    Clerk.startRoof();
    ‘‘build roof’’
}

monitor Clerk{ %not fair
    int floor = 0, walls = 0;
    cond buildWalls, buildRoof;

    procedure startWalls(){
        while(floor < 1){
            wait(buildWalls);
        }
        floor = floor - 1;
    }

    procedure startRoof(){
        while(walls < 1){
            wait(buildroof);
        }
        walls = walls - 1;
    }

    procedure doneFloor(){
        floor = floor + 1;
        signal(buildWalls);
    }

    procedure doneWalls(){
        walls = walls + 1;
        signal(buildRoof);
    }
}
```

House Building (Asynchronous Communications)

House:

- **One** floor
- **Four** walls
- **One** roof

- *Six* agents

WallBuilder

```
F : Agent;  // assumed initialized to the FloorBuilder
R : Agent;  // assumed initialized to the RoofBuilder

while true do
    await F:startWall;
    // build wall
    send R:startRoof
od
```

Program Analysis

Loop invariant: $\#(h/(\downarrow\texttt{startWall})) = \#(h/(\uparrow\texttt{startRoof}))$

$\{\#(h/(\downarrow\texttt{startWall})) + 1 = \#(h/(\uparrow\texttt{startRoof})) + 1\}$
await `R:startWall`
$\{\#(h/(\downarrow\texttt{startWall})) = \#(h/(\uparrow\texttt{startRoof})) + 1\}$
`// build wall`
$\{\#((h; W \uparrow R : \texttt{startRoof})/(\downarrow\texttt{startWall}))$
$= \#((h; W \uparrow R : \texttt{startRoof})/(\uparrow\texttt{startRoof}))\}$
sent `R:startRoof`
$\{\#(h/(\downarrow\texttt{startWall})) = \#(h/(\uparrow\texttt{startRoof}))\}$

History Invariant

$\#(h/\uparrow\texttt{startRoof}) \leq \#(h/\downarrow\texttt{startWall}) \leq \#(h/\uparrow\texttt{startRoof}) + 1$

For **await** `R:startWall`, we have:

$$\#(h/(\downarrow\texttt{startWall})) = \#(h/(\uparrow\texttt{startRoof})) + 1 \Rightarrow$$
$$\#(h/\uparrow\texttt{startRoof}) \leq \#(h/\downarrow\texttt{startWall}) \leq \#(h/\uparrow\texttt{startRoof}) + 1$$

For **sent** `R:startRoof`, we have:

$$\#(h/(\downarrow\texttt{startWall})) = \#(h/(\uparrow\texttt{startRoof})) \Rightarrow$$
$$\#(h/\uparrow\texttt{startRoof}) \leq \#(h/\downarrow\texttt{startWall}) \leq \#(h/\uparrow\texttt{startRoof}) + 1$$

RoofBuilder (Skeleton)

```
while true do
    ...
    // build roof
    ...
od
```

RoofBuilder

```
while true do
    n : Int = 0;
    while n < 4 do
       await X?:startRoof;
       n:= n + 1;
    od
    // build roof
od
```

RoofBuilder (Modified)

```
while true do
  await X? startRoof(x);
   ms.insert(x);
  if ms.get(x) = 4 then // build roof x fi
od
```

15.4 09

Chat

- identity exchange
- symmetric

skeleton

```
  .....                          # global variables
  process Chat_Client[i = 1 to M]{
  int partner_id;
  ...
  [chatting]
}
```

Semaphore

```
int exchange_id;
sem enter = 1;
sem continue = 0;

process Chat_Client[i = 1 to M]{
  int partner_id;
  while (true){
    P(enter);                          // mutex [
      if (exchange_id == 0){           // cell empty
        exchange_id = i;               // I want a partner
        V(enter)                       // mutex ]
        p(continue)                    // waiting for back-communication
        partner_id = exchange_id;      // continue
        exchange_id = 0;               // empty
        V(enter);                      // mutex
      }
      partner_id = exchange_id         // learn my partner
      exchange_id = i;                 // send my id (to partner!)
      V(continue)                      // signal, do not release P(enter)
  }
}
```

Monitor (SC)

```


monitor Chat_Server {
  bool waiting = false;
  int wait_id;
  cond not_alone;
  procedure findPartner(int id, int &partner_id) {
    if (!waiting) {
      wait_id = id;
      waiting = true;
      wait (not_alone);
      partner_id = wait_id;
      waiting = false;
    } else {
      partner_id = wait_id;
      wait_id = id;
      signal(not_alone);
    }
  }
}
```

Passing the control back

- Note: symmetry
- signalling: we cannot target one particular process[69], we can only do
  - signal all (for SC only)
  - signal
- using `not_alone`

```
monitor Chat_Server {
  int partner;
  cond not_alone, not_busy;

  procedure findPartner(int id, int &partner_id) {

    if (empty(not_alone)) {        // am I alone
      partner = id;                // remember me
      wait(not_alone);             // suspend myself
        partner_id = partner;      // read the cell
```

[69] Unless with some extra semaphores

```
    }
    else {                          // I am not alone
        partner_id = partner;       // read  the cell
        partner = id;               // write back to partner
      signal(not_alone);            //
    }
  }
}
```

Monitor

```
monitor Chat_Server {
  int partner;
  bool busy = false;
  cond not_alone, not_busy;

  procedure findPartner(int id, int &partner_id) {
    while (busy) wait(not_busy);

    if (empty(not_alone)) {     // am I alone
      partner = id;             // remember me
      wait(not_alone);          // suspend myself
        partner_id = partner;   // read the cell
        busy = false;           // start fresh by
      signal_all(not_busy);     //
    }
    else {                      // I am not alone
        busy = true;            // partner must contain chat partner
        partner_id = partner;   // read  the cell
        partner = id;           // write back to partner
      signal(not_alone);        //
    }
  }
}
```

References

[int, 2013] (2013). *Intel 64 and IA-32 Architectures Software Developer s Manual. Combined Volumes:1, 2A, 2B, 2C, 3A, 3B and 3C*. Intel.

[Adve and Gharachorloo, 1995] Adve, S. V. and Gharachorloo, K. (1995). Shared memory consistency models: A tutorial. Research Report 95/7, Digital WRL.

[Adve and Hill, 1990] Adve, S. V. and Hill, M. D. (1990). Weak ordering — a new definition. *SIGARCH Computer Architecture News*, 18(3a).

[Anderson, 1990] Anderson, T. E. (1990). The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed System*, 1(1):6–16.

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

[Go memory model, 2016] Go memory model (2016). The Go memory model. `https://golang.org/ref/mem`.

[Goetz et al., 2006] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. (2006). *Java Concurrency in Practice*. Addison-Wesley.

[Herlihy and Shavit, 2008] Herlihy, M. and Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann.

[Lamport, 1979] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691.

[Lea, 1999] Lea, D. (1999). *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 2d edition.

[Magee and Kramer, 1999] Magee, J. and Kramer, J. (1999). *Concurrency: State Models and Java Programs*. John Wiley & Sons Inc.

[Manson et al.,] Manson, J., Pugh, W., and Adve, S. V. The Java memory memory.

[Owell et al.,] Owell, S., Sarkar, S., and Sewell, P. A better x86 memory model: x86-TSO.

[Sewell et al., 2010] Sewell, P., Sarkar, S., Nardelli, F., and O.Myreen, M. (2010). x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7).

Index