

Intro

INF4140 - Models of concurrency

Intro, lecture 1

Høsten 2014

29. 08. 2014



Today's agenda

Introduction

- overview
- motivation
- simple examples and considerations

Start

a bit about

- concurrent programming with critical sections and waiting, read^a also chapter 1 for some background
- interference
- the await language

^ayou!, as course participant

What this course is about

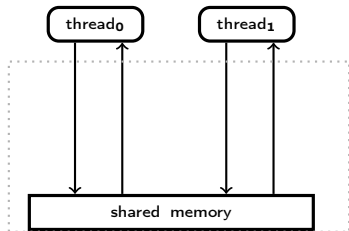
- Fundamental issues related to cooperating parallel processes
- How to think about developing parallel processes
- Various language mechanisms, design patterns, and paradigms
- Deeper understanding of parallel processes:
 - (informal and somewhat formal) analysis
 - properties

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

Parallel processes need to exchange information.

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

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



- 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

- asynchronous and synchronous message passing
- Basic mechanisms: RPC (remote procedure call), rendezvous, client/server setting, channels
- Java's mechanisms
- analysis using histories
- asynchronous systems

Why shared (global) variables?

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

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

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

Simple example

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

$$x := x + z; y := y + z;$$

Pre/post-condition

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

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

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

Simple example

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

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

Pre/post-condition

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

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

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

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

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

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

Example

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

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

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

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

Example

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

Interaction between processes

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

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

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

Synchronization

Synchronization (veeeery abstractly) = **restricting** the possible interleavings of parallel processes (so as to avoid “bad” things to happen and to achieve “positive” things)

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

Definition (Atomic)

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

Note

- A statement with at most one atomic operation, in addition to operations on local variables, can be considered atomic!
- We can do as if atomic operations do not happen concurrently!
- What is atomic depends on the language/setting: **fine-grained** and **coarse-grained** atomicity.
- e.g.: Reading and writing of a global variable is usually atomic.
- For some (high-level) languages: assignments $x := e$ atomic operations, for others, not (reading of the variables in the expression e , computation of the value e , followed by writing to x .)

Atomic operations on global variables

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

Mutual exclusion

Atomic operations on a variable cannot happen simultaneously.

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

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

- Assume:
 - each process is executed on its own processor
 - and/or: the processes run on a multi-tasking OSand 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

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

```
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”:²

- R1 must happen before W1 and
- R2 before W2

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

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

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
<hr/>					
0	-1	1	-1	1	0

²A word aside: as natural as this seems: in a number of modern architecture/modern languages & their compilers, **this is not guaranteed!** Cf. Java’s memory model

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

- 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 $x = -1 \vee x = 0 \vee x = 1$

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

Non-determinism

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

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

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

State-space explosion

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

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

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

For n processes with m atomic statements each:

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

The “at-most-once” property

Fine grained atomicity

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

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

Definition (At-most-once property)

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

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

^aor just read.

assignments with at-most-once property can be considered atomic

The “at-most-once” property

Fine grained atomicity

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

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

Definition (At-most-once property)

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

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

^aor just read.

At most once examples

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

$x := x + 1 \parallel y := x + 1$

$x := y + 1 \parallel y := x + 1 \quad \{ (x, y) \in \{(1, 1), (1, 2), (2, 1)\} \}$

$x := y + 1 \parallel x := y + 3 \parallel y := 1 \quad \{y = 1 \wedge x = 1, 2, 3, 4\}$

$r := y + 1 \parallel r' := y - 1 \parallel y := 5$

$r := x - x \parallel \dots \quad \{\text{is } r \text{ now } 0?\}$

$x := x \parallel \dots \quad \{\text{same as skip?}\}$

$\text{if } y > 0 \text{ then } y := y - 1 \text{ fi} \parallel \text{if } y > 0 \text{ then } y := y - 1 \text{ fi}$

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

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

We use the following syntax for non-parallel control-flow⁵

Declarations

```
int i = 3;  
int a[1:n];  
int a[n];6  
int a[1:n] = ([n] 1);
```

Assignments

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

Seq. composition

statement; statement

Compound statement

{statements}

Conditional

if statement

While-loop

while (condition) statement

For-loop

for [i = 0 to n - 1]statement

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

⁶corresponds to: `int a[0:n-1]`

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

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

```
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

⁷one 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

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

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

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

- note: expressions side-effect free

- 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

- A *state* in a parallel program consists of the values of the global variables at a given moment in the execution.
- Each process executes independently of the others by *modifying* global variables using atomic operations.
- An execution of a parallel program can be modelled using a *history*, i.e. a sequence of operations on global variables, or as a sequence of states.
- For non-trivial parallel programs there are *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 program⁸ is a predicate which is true for all possible histories of the program.
- Two types:
 - *safety* property: program will not reach an undesirable state
 - *liveness* property: program will reach a desirable state.
- *partial correctness*: If the program terminates, it is in a desired final state (safety property).
- *termination*: all histories are finite.⁹
- *total correctness*: The program terminates and is partially correct.

⁸the program “has” that property, the program satisfies the property ...

⁹that’s also called *strong* termination. Remember: non-determinism. 

Properties: Invariants

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

Definition (Invariant)

an **invariant** = state property, which holds for holds for all **reachable** states.

- safety property
- appropriate for also non-terminating systems (does not talk about a final state)
- *global* invariant talks about the state of many processes at once, preferably the entire system
- *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.

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!

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

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

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

¹⁰In programming languages, one could find it as `atomic{S}` or similar.

Await statement

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

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

Example

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

- *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

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

¹¹Later, a special kind of semaphore (a binary one) is also called a “mutex”. Terminology is a bit flexible sometimes.

¹²one may also see sometimes just $\text{await}(b)$: however, eval. of b better be *atomic* and under *no* circumstances must b have *side-effects* (**never, ever. Seriously**).

Typical pattern

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

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

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

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

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

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

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

Example (continued)

a:

buf: p: c: n:

b:

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

References I

- [Andrews, 2000] Andrews, G. R. (2000).
Foundations of Multithreaded, Parallel, and Distributed Programming.
Addison-Wesley.
- [Goetz et al., 2006] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. (2006).
Java Concurrency in Practice.
Addison-Wesley.
- [Lea, 1999] Lea, D. (1999).
Concurrent Programming in Java: Design Principles and Patterns.
Addison-Wesley, 2d edition.
- [Magee and Kramer, 1999] Magee, J. and Kramer, J. (1999).
Concurrency: State Models and Java Programs.
Wiley & Sons.