INF4140 - Models of concurrency

Fall 2016

August 31, 2016

Abstract

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

This handout version also contains more remarks and footnotes, which would clutter the slides, and which typically contains remarks and elaborations, which may be given orally in the lecture. Not included currently here is the material about weak memory models.

1 Intro

29 August 2016

1.1 Warming up

Today's agenda

Introduction

- overview
- motivation
- simple examples and considerations

Start
a bit about

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

What this course is about

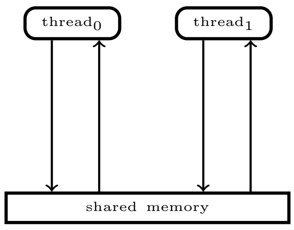
- Fundamental issues related to cooperating parallel processes
- How to think about developing parallel processes
- Various language mechanisms, design patterns, and paradigms
- Deeper understanding of parallel processes:
  - properties
  - informal analysis
  - *formal* analysis

Parallel processes

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

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

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



Course overview – part I: Shared variables

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

Course overview – part II: Communication

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

Part I: shared variables

Why shared (global) variables?

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

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

Simple example

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

before after

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

Pre/post-conditions

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

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

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

Parallel operator $\parallel$

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

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

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

Example 1.

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

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 do as if atomic operations do not happen concurrently!
- What is atomic depends on the language/setting: fine-grained and coarse-grained atomicity.
- e.g.: Reading/writing of global variables: usually atomic.[2]
- note: $x := e$: assignment statement, i.e., more than write to x!

Atomic operations on global variables

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

Mutual exclusion
Atomic operations on a variable cannot happen simultaneously.

Example

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

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

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

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

[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

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

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]

⇒ 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] $x = -1 \vee x = 0 \vee x = 1$

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

[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
- consider executions of $P_1 \parallel P_2 \parallel P_3$

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

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

For n processes with m atomic statements each:

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

The "at-most-once" property

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

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

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

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

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

At-most-once examples

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

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

1.2 The await language

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

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

[5] or just read.

Syntax

We use the following syntax for non-parallel control-flow[6]

Declarations	Assignments
`int i = 3;`	`x := e;`
`int a[1:n];`	`a[i] := e;`
`int a[n];`[7]	`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

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

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

Parallel processes

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

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

Example

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

Starts one process.

The numbers are printed in increasing order.

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

Starts n processes.

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

Read- and write-variables

- $\mathcal{V}$: *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

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

[8]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.

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

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

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

- note: expressions side-effect free

Disjoint processes

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

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

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

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

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

1.3 Semantics and properties

Semantic concepts

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

Properties

- property = predicate over programs, resp. their histories
- A (true) *property* of a program[9] is a predicate which is true for all possible histories of the program.

  Classification

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

- *partial correctness*: *If* the program terminates, it is in a desired final state (safety property).
- *termination*: all histories are finite.[10]
- *total correctness*: The program terminates and is partially correct.

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

Properties: Invariants

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

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

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

  ***proof principle:* induction**

  one can show that an invariant is correct by

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

  Note: we avoid looking at all possible executions!

How to check properties of programs?

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

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

Critical sections

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

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

Atomic operations:

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

Example The example from before can now be written as:

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

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

Conditional critical sections

Await statement

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

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

  Example 5.

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

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

2 special cases

- unconditional critical section or "mutex"[12]

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

- Condition synchronization:[13]

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

Typical pattern

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

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

- "critical statements" *not* enclosed in ⟨angle brackets⟩. 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

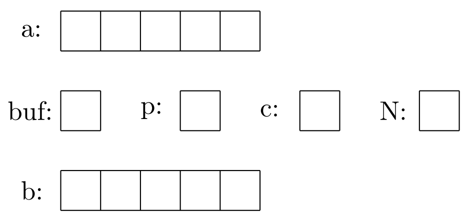
```
               int buf, p := 0; c := 0;

process Producer {                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;
 }                                   }
}                                 }
```

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

[13]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 (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`

References

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