Introduction

INF4140

30.08.12

Lecture 1

Models of concurrency

Lecturers

Crystal Din

office: 8167 email: crystald@ifi.uio.no phone: 22 84 08 52

Lizeth Tapia

office: 8167 email: sltarifa@ifi.uio.no phone: 22 84 08 52

Homepage for the course

`http://www.uio.no/studier/emner/matnat/ifi/INF4140/h12`

Syllabus (see the homepage for details)

- Gregory R. Andrews: *Foundations of Multithreaded, Parallel, and Distributed Programming*
- The paper: Intra-Object versus Inter-Object: concurrency and Reasoning in Creol

Models of concurrency

Lectures

Thur. 12:15 - 14:00, Room Shell, Ole-Johan Dahls Hus

Exercise course

- Tutor: Crystal Din and Lizeth Tapia
- Mondays 14:15 - 16:00, Room Java, Ole-Johan Dahls Hus
- Starts next week.

Evaluation

- Three compulsory assignments which must be approved.
- Final exam: 17 December at 14:30 (4 hours).

Today's agenda

Introduction

- contents of the course
- motivation: why is this course important
- some simple examples and considerations

Start

- a bit about concurrent programming with critical sections and waiting
- 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 and paradigms
- Deeper understanding of parallel processes: (informal and somewhat formal) analysis, properties

Parallel processes

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

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

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

Overview of topics in the course - 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

Overview of topics in the course - 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
- standard examples

Part I: shared variables

Language mechanisms and theory for processes which operate on *shared global variables*.

Why use shared variables?

- Here's the situation: There may be several CPUs inside one machine.
- natural interaction for tightly coupled systems
- used in many important languages, as in Java's concept of threads
- do as if one has many processes, in order to get a natural partitioning
- can achieve greater efficiency if several things happen at the same time

e.g. several active windows at the same time

Simple example

We have global variables x, y, and z. Consider the following program:

pre　　　　　　　　　　　　　　　　post

{x is a and y is b}　$x := x + z; y := y + z;$　{x is a+z and y is b+z}

If we have operations that can be performed *independently of one another*, then it can be advantageous to perform these concurrently.

- The conditions describe the state of the global variables before and after the program statement and are called, respectively, *pre- and post-conditions*.
- These conditions are meant to give an understanding of the program, and are not part of the executed code.

Can we use parallelism here?

Parallel operator

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

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

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

Example Thus we can write an example as follows:

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

Interaction between processes

Processes which live in the same system 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*.

- *Mutual exclusion (Mutex).* We introduce *critical sections* of program instructions which can *not* be executed concurrently.
- *Condition synchronization.* A process must wait for a specific condition to be satisfied before execution can continue.

Concurrent processes: Atomic operations

Definition from the book: An operation is atomic if it cannot be subdivided into smaller components.

Alternative definition: An operation is atomic if it can be understood without dividing it into smaller components.

What is atomic depends on which language we work with: fine-grained and coarse-grained atomicity.

e.g.: Reading and writing of a global variable is usually atomic. Some (high-level) languages can have assignment $x := e$ as one atomic operation, others as several: reading of the variables in the expression e, computation of the value e, followed by writing to x.

Note: A statement with at most one atomic component operation, in addition to operations on local variables, can be considered atomic!

Note: We can do as if atomic operations do not happen concurrently!

Atomic operations: global variables

The treatment of global variables is fundamental:
Also, the *communication* between processes can be represented by variables, e.g. a communication channel as a variable of type vector.
We associate with each global variable a set of *atomic operations*, e.g. reading and writing to normal global variables, sending and receiving of communication channels, etc. Atomic operations on a variable x are called *x-operations*.

Mutual exclusion

Atomic operations on a variable cannot happen simultaneously.

Example

Consider the following program:

$$\begin{array}{ccc} P_1 & & P_2 \\ \{x{=}{=}0\} & \textbf{co}\ x := x+1 \| x := x-1\ \textbf{oc}; & \{?\} \end{array}$$

What will be the final state here?

- We assume that each process is executed on its own processor, with its own registers, and that x is part of a shared state space with global variables.
- Arithmetic operations in the two processes can be executed simultaneously, but read and write operations on x must be performed sequentially.
- The order of these operations is dependent on relative processor speed.
- The outcome of such programs thus becomes very difficult to predict!

Atomic read and write operations

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

There are 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. R1 must happen before W1 and R2 before W2, so these operations can be sequenced in 6 ways:

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

From this table we can obtain the final state of the program: $x = -1 \vee x = 0 \vee x = 1$. The program is thus *non-deterministic*: the result can vary from execution to execution.

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

Number of possible executions

If we have 3 processes, each with a given number of atomic operations, we will obtain the following number of possible executions:

process 1	process 2	process 3	number of executions
2	2	2	90
3	3	3	1680
4	4	4	34 650
5	5	5	756 756

NB: Different executions can lead to different final states.

Impossible, even for quite simple systems, to consider every possible execution!

Different executions

For n processes with m atomic statements each, the formula is

$$\frac{(n * m)!}{m!^n}$$

The "at-most-once" property

Definition. If an expression e in one process does not reference a variable altered by another process, expression evaluation will appear to be atomic. An assignment $x := e$ satisfies the property if either e satisfies the property and x is not referenced by others, or e does not reference any shared variables and x can be read or written by other processes.

Such expressions/statements can be considered atomic!

Examples:

$x := 0; y := 0; \textbf{co}\ x := x + 1 || y := x + 1\ \textbf{oc}$

$x := 0; y := 0; \textbf{co}\ x := y + 1 || y := x + 1\ \textbf{oc};\ \{\text{x and y is 1 or 2}\}$

$x := 0; y := 0; \textbf{co}\ x := y + 1 || x := y + 3 || y := 1\ \textbf{oc};\ \{y = 1 \wedge x = 1, 2, 3, 4\}$

$\textbf{co}\ z := y + 1 || z' := y - 1 || y := 5\ \textbf{oc}$

$z := x - x || \ldots\ \{\text{is z now 0?}\}$

$x := x || \ldots\ \{\text{same as skip?}\}$

$\textbf{if y} > \textbf{0 then y} := \textbf{y} - \textbf{1 fi} || \textbf{if y} > \textbf{0 then y} := \textbf{y} - \textbf{1 fi}$

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

Programming language: Syntax

We use the following syntax for basic constructions

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

Compound statement	`{ statements }`
Conditional	`if (condition) statement`
While-loop	`while (condition) statement`
For-loop	`for [i=0 to n-1] statement`

---

[1] corresponds to: `int a[0:n-1]`

Parallel statements

$$co S_1 || S_2 || \dots || S_n oc;$$

- Each arm S_i contains a program statement which is executed in parallel with the other arms.
- The **co**-statement terminates when all the arms S_i have terminated.
- The next instruction after the **co**-statement is executed after the **co**-statement has terminated.

Parallel processes

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

- Processes run in the background
- Processes evaluated in arbitrary order.
- Processes are declared (as methods/functions)

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

Let $\mathcal{V}: \textit{statement} \longrightarrow \textit{variable set}$ be a syntactic function which computes the set of global variables which are referenced in the program. $\mathcal{W}: \textit{statement} \longrightarrow \textit{variable set}$ is the set of (write)variables which can be changed by the program.

$$
\begin{array}{ll}
\mathcal{V}[\texttt{v:=e}] = \mathcal{V}[\texttt{e}] \cup \{\texttt{v}\} & \qquad \mathcal{W}[\texttt{v:=e}] = \{\texttt{v}\} \\
\mathcal{V}[S_1; S_2] = \mathcal{V}[S_1] \cup \mathcal{V}[S_2] & \qquad \mathcal{W}[S_1; S_2] = \mathcal{W}[S_1] \cup \mathcal{W}[S_2] \\
\mathcal{V}[\texttt{if (b) } S\ ] = \mathcal{V}[\texttt{b}] \cup \mathcal{V}[S] & \qquad \mathcal{W}[\texttt{if (b) } S\ ] = \mathcal{W}[S] \\
\mathcal{V}[\texttt{while (b) } S\ ] = \mathcal{V}[\texttt{b}] \cup \mathcal{V}[S] & \qquad \mathcal{W}[\texttt{while (b) S }] = \mathcal{W}[S]
\end{array}
$$

where $\mathcal{V}[\texttt{e}]$ is the set of variables in expression e.

Disjoint processes

Parallel processes are without interference if they are disjoint, i.e. without common global variables:

$$\mathcal{V}[S_1] \cap \mathcal{V}[S_2] = \emptyset$$

Meanwhile, variables which are only read cannot give rise to interference. The following *interference criterion* is thus sufficient:

$$\mathcal{V}[S_1] \cap \mathcal{W}[S_2] = \mathcal{W}[S_1] \cap \mathcal{V}[S_2] = \emptyset$$

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 there are *very many possible histories*.
- Synchronization is used to *limit* the possible histories.

Properties

A *property* of a program is a predicate which is true for all possible histories of the program.

- Two types:
    - *Safety properties* say that the program will not reach an undesirable state
    - *Liveness properties* say that the program will reach a desirable state.
- *Partial correctness*: The program reaches a desired final state if the program terminates (safety property).
- *Termination*: All histories have finite length.
- *Total correctness*: The program terminates and is partially correct.

Properties: Invariants

Definition A common property for all states which can be reached during execution, i.e. a property which holds at any time.

- safety property
- appropriate for 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
- one can show that an invariant is correct by showing that it holds initially, 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 our 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.

A test can show an error, but can never prove correctness!

Critical sections

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

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

Atomic operations:

- Internal states are *not visible* to other processes.
- Variables *cannot* be changed by other processes.

Example The example from before can now be written as:

$$\text{int } x := 0;\ \textbf{co}\ < x := x + 1 > \,||\, < x := x - 1 >\ \textbf{oc};\ \{\text{x is 0 here}\}$$

Conditional critical sections

Introduce the following expression:

`< await (B) S; >`

- The Boolean expression `B` specifies an await condition.
- The angle brackets indicate that the body `S` is executed as an atomic operation.

Example `< await (y > 0) y := y-1; >`

The variable `y` is first decremented when the condition `y > 0` holds.

Conditional critical sections (2)

- Mutex:

```
< x := x+1 ; y := y+1 ; >
```

- Condition synchronization:

```
< await (counter > 0) ; >
```

We can use `await` to specify both synchronization methods:

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

Invariant: $0 \le counter \le 1$

Example: producer/consumer synchronization

Let `Producer` be a process which delivers data to a `Consumer` process.

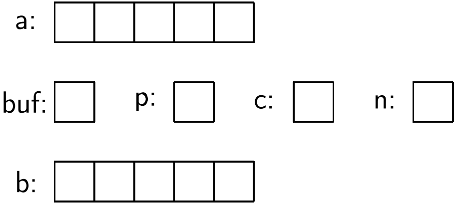
```
int buf, p := 0, c := 0;
```

```
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;
  }
}
```

This type of synchronization is usually called *busy waiting*.

Example (continued)



Global Invariant: `c <= p <= c+1`
Local Invariant (Producer): `0 <= p <= n`
An invariant holds in *all states* in the history of the program.