

INF4140 - Models of concurrency

Fall 2016

September 15, 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 Locks & barriers

5. 09. 2016

Practical Stuff

Mandatory assignment 1 (“oblig”)

- Deadline: Friday September 30 at 18.00
- 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 {
  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;
  }
}
```

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): $0 \leq p \leq N$

1.1 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) {
    CEntry      # entry protocol to CS
    CS
    CExit      # 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 {
  while (true) {
    while (in=2) {skip};
    CS;
    in := 2;
    non-CS
  }
}

process p2 {
  while (true) {
    while (in=1) {skip};
    CS;
    in := 1
    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,¹ 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*.

Atomic section

Used for synchronization of processes

- General form:

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

- 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:**²

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

¹The question for points 2 and 3, whether it's safety or liveness, is slightly up-to discussion/standpoint!

²We also use then just **await** (B) or maybe **await** B. But also in this case we assume that B is evaluated atomically.

Critical sections using “locks”

```
bool lock = false;

process [i=1 to n] {
  while (true) {
    < await ( $\neg$  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 $\langle \dots \rangle$?

“Test & Set”

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.

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**³ needed to guarantee eventual entry for a process

Variable **lock** becomes a hotspot!

³see later

A puzzle: “paranoid” entry protocol

Better safe than sorry?

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

```
bool lock := false;

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

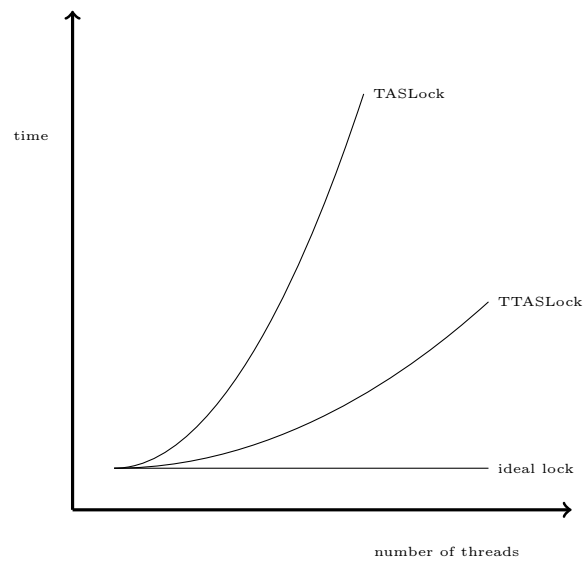
    CS;
    lock := false;
    non-CS
  }
}
```

```
bool lock := false;

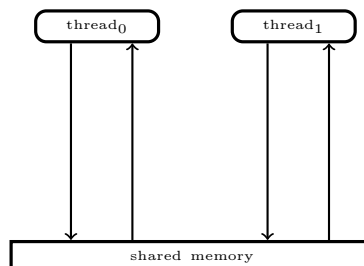
process p[i = i to n] {
  while (true) {
    while (lock) {skip}; # additional spin lock check
    while (TS(lock)) {
      while (lock) {skip}}; # + again inside the TAS loop
    CS;
    lock := false;
    non-CS
  }
}
```

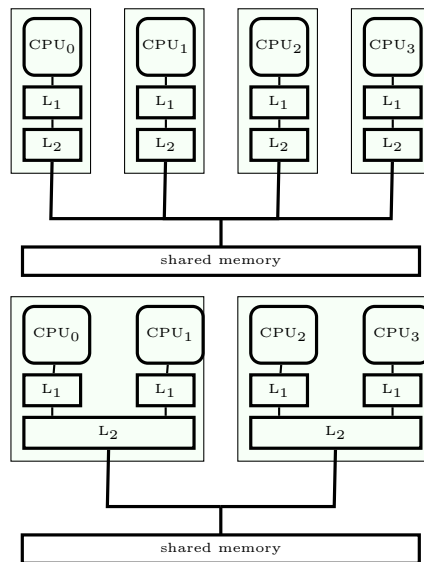
Does that make sense?

Multiprocessor performance under load (contention)



A glance at HW for shared memory





Test and test & set

- **Test-and-set** operation:
 - (Powerful) HW instruction for synchronization
 - Accesses main memory (and involves “cache synchronization”)
 - Much slower than cache access
- **Spin-loops**: faster than TAS loops
- “**Double-checked locking**”: sometimes design pattern/programming idiom for efficient CS (under certain architectures)⁴

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* $\langle \text{await (B) } S; \rangle$:

```

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.

1.2 Liveness and fairness

Liveness properties

So far: no(!) solution for “**Eventual Entry**”.⁵

Liveness

Eventually, something good will happen.

- Typical example for **sequential** programs: Program termination⁶
- Typical example for **parallel** programs: A given process will eventually enter the critical section

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

⁴depends on the HW architecture/memory model. In some architectures: does not guarantee mutex! in which case it’s an anti-pattern ...

⁵Except the very first (which did not satisfy “absence of unnecessary delay”

⁶In the first version of the slides of lecture 1, termination was defined misleadingly/too simple.

Scheduling and fairness

enabledness

Command *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?⁷
- Possible *status* changes:
 - disabled → enabled (of course),
 - but also enabled → disabled
- Differently “powerful” *forms of fairness*: guarantee of progress
 1. for actions that are always enabled
 2. for those that *stay enabled*
 3. for those whose enabledness show “on-off” behavior

Unconditional fairness

Definition 1 (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
```

- $x := \mathbf{false}$ 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

⁷provided 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).

Weak fairness

Definition 2 (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 $y \geq 10$ 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 3 (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 \Rightarrow 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 {
  while (true){
    while (in2) {skip};
    in1 := true;
    CS
    in1 := false;
    non-CS
  }
}

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

What is the global invariant here?

Problem: No *mutex*

Tie-Breaker algorithm: Attempt 2

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

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

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

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

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

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

- Problem seems to be the entry protocol
- Reverse the order: first “set”, then “test”

“Deadlock”⁸ :- (

Tie-Breaker algorithm: Attempt 3 (with await)

- Problem: both half flagged their wish to enter \Rightarrow deadlock
- Avoid deadlock: “tie-break”
- Be fair: Don’t always give priority to one specific process
- Need to know which process last started the entry protocol.
- Add new variable: last

in1 := false, in2 := false; int last

⁸Technically, 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.

```

process p1 {
  while (true){
    in1 := true;
    last := 1;
    < await ( (not in2) or
              last = 2);>
    CS
    in1 := false;
    non-CS
  }
}

```

```

process p2 {
  while (true){
    in2 := true;
    last := 2;
    < await ( (not in1) or
              last = 1);>
    CS
    in2 := false;
    non-CS
  }
}

```

Tie-Breaker algorithm

Even if the variables `in1`, `in2` and `last` can change the value while a wait-condition evaluates to true, the wait condition will *remain true*.

`p1` sees that the wait-condition is true:

- `in2 = false`
 - `in2` can eventually become **true**, but then `p2` must also set `last` to 2
 - Then the wait-condition to `p1` still holds
- `last = 2`
 - Then `last = 2` will hold until `p1` has executed

Thus we can replace the **await**-statement with a **while**-loop.

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:⁹

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

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

Ticket algorithm: Invariant

Invariants

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

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

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

$$\text{if } \text{turn}[i] > 0 \text{ then } \text{turn}[j] \neq \text{turn}[i]$$

This holds initially, and is preserved by all atomic statements.

1.3 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.

⁹Why?

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]:
  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`

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* !!

References

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