Locks & barriers

INF4140 - Models of concurrency

Locks & barriers, lecture 2

Høsten 2014

5. 9. 2014

Mandatory assignment 1 ("oblig")

- Deadline: Friday September 26 at 18.00
- Possible to work in pairs
- Online delivery (Devilry): https://devilry.ifi.uio.no

Introduction

- Central to the course are general mechanisms and issues related to parallel programs
- **Previous class:** *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): $0 \leq p \leq N$

Critical section

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

Main question we are discussing 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
- A solution to the CS problem can be used to *implement* **await**-statements

Critical section: First approach to a solution

Operations on shared variables happen 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?

- **Mutual exclusion (Mutex):** At any time, at most one process is inside CS.
- **Absence of deadlock:** If all processes are trying to enter CS, at least one will succeed.
- **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.
- **Eventual entry:** A process attempting to enter CS will eventually succeed.

NB: The three first are safety properties,[1]
The last a liveness property.

(SAFETY: no bad state
LIVENESS: something good will happen.)

[1] point 2 and 3 are slightly up-to discussion/standpoint!

Safety: Invariants (review)

A safety property expresses that 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*.

Used for synchronization of processes

- General form:

$$< \textbf{await}\ (B)\ S; >$$

  - B: Synchronization condition
  - Executed atomically when B is true

- Unconditional critical section (B is **true**):

$$< S; >$$

  S executed atomically

- Conditional synchronization:[2]

$$< \textbf{await}\ (B); >$$

[2] We 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 (¬ 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 <...>?

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
  }
}
```

NB:

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

Strong fairness needed to guarantee eventual entry for a process

Variable lock becomes a hotspot!

A puzzle: "paranoid" entry protocol

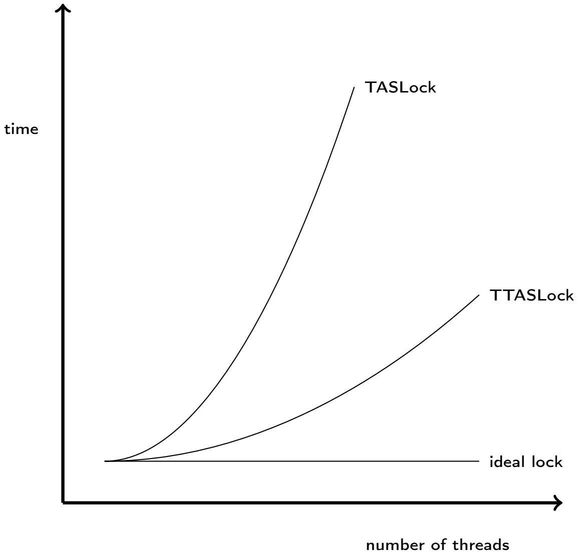
Better safe than sorry?

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

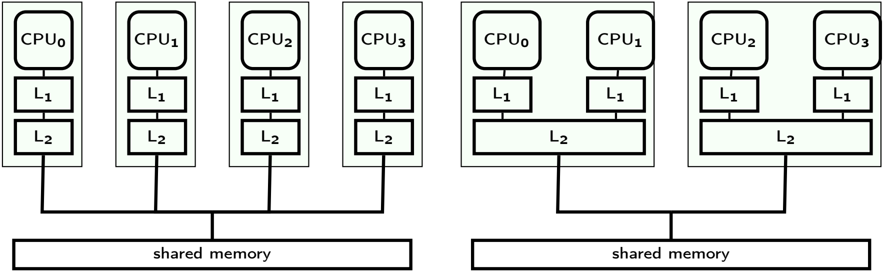
```
bool lock := false;

process p[i = i to n] {
  while (true) {
    while (lock) {skip};      # additional spin lock check
    while (TS(lock)) {
      while (lock) {skip}}; # + more 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": important design pattern/programming idiom for efficient CS (under certain architectures)[3]

[3] 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 < S;> 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.

So far: no(!) solution for "Eventual Entry"-property, except the very first (which did not satisfy "Absence of Unnecessary Delay").

- Liveness: Something good will happen
- Typical example for sequential programs: (esp. in our context) Program termination[4]
- Typical example for parallel programs:
  A given process will eventually enter the critical section

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

[4] In the first version of the slides of lecture 1, termination was defined misleadingly.

Scheduling and fairness

- 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? [5]
- Possible status changes:
  - disabled $\rightarrow$ enabled (of course),
  - but also enabled $\rightarrow$ 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

[5] provided the control-flow/program pointer stands in front of them.

Unconditional fairness

A scheduling strategy is *unconditionally fair* if each unconditional atomic action which can be chosen, 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

- This guarantees termination
- Example: "Round robin" execution
- Note: if-then-else, **while** (b) ; are *not* conditional atomic statements!

Weak fairness

Weak fairness

A scheduling strategy is *weakly fair* if

- it is 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

Example

```
bool x := true; y := false;

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

Definition (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 to assume 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**
- Weak fairness:
  Process i can read lock only when the value is **false**
- Strong fairness:
  Guarantees that i eventually sees that lock is **true**

Difficult: to make a 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

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?

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*

```
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
  }
}
```

Deadlock[6] :-(

[6] 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: 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

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

```
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.

```
process p1 {
  while (true){
    in1 := true;
    last := 1;
    while (in2 and 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.

```
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
  }
}
```

- The first line in the loop must be performed atomically!
- **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:[7]

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

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

[7] Why?

Ticket algorithm: Invariant

Invariants

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

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

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

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

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

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!!

Divides shared counter into several local variables.

```
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 := 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
    }
  }
}
```

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 fever processors than processes
  - Should not waste time executing a skip loop!
- No clear distinction between variables used for synchronization and computation!

Desirable to have a special tools for synchronization protocols

Next week we will do better: semaphores !!

References I

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