Locks and Barriers

INF4140

06.09.12

Lecture 2

Practical Stuff

Compulsory assignment 1

- Deadline: Friday September 28 at 18.00
- It is 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 the *critical section*
  - Protect reading and writing to *shared variables*
- *Barriers*
  - Iterative algorithms:
    Processes must *synchronize* between each iteration
  - Coordination using *flags*

Review: await-example: Producer/Consumer

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;
  }
}
```

Global Invariant: `c <= p <= c+1`

Local Invariant (Producer): `0 <= p <= n`

An invariant holds in *all states* in the history of the program.

Main question:

How can we implement critical sections / conditional critical sections?

- using *locks* and low-level operations
- active waiting (later semaphores and passive waiting)
- various solutions and properties

Access to Critical Section (CS)

- Several processes compete for access to a shared resource
- Only one process can have access at a time
- Possible examples:
  - Execution of bank transactions
  - Access to a printer
- The solution 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 ;
           CS;
          CSexit ;
          not CS
     }
}
```

Assumption: A process which enters the CS will eventually leave it.

Naive solution

```
int in = 1;        — 1 or 2

process p1 {
    while (true) {
        while (in=2) skip;
        CS
        in = 2;
        not CS
    }
}

process p2 {
    while (true) {
        while (in=1) skip;
        CS
        in = 1;
        not CS
    }
}
```

Good solution? What is good, what is not so good?

- More than 2 processes?
- Different execution times?

Desired Properties

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 that is attempting to enter CS will eventually succeed.

NB: The three first are safety properties, the last a liveness property. (SAFETY: no bad state – LIVENESS: something good will happen.)

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 usually called a *global invariant*.

Atomic sections

Used for synchronization of processes

- General form:

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

  - `B`: Synchronization condition
  - Executed atomically when `B` is true
- Unconditional critical section (`B` is **true**):

$$< \texttt{S;} >$$

  `S` executed atomically
- Conditional synchronization:

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

Critical section using locks

```
bool lock = false;

process p[i=1 to n] {
     while (true) {
          < await (!lock) lock = true; >
          CS
          lock = false;
          not CS
     }
}
```

Safety properties:

- Mutex
- Absence of deadlock
- Absence of unnecessary waiting

What about taking away the angle brackets <...>?

“Test & Set”

Test & Set is a method for implementing *conditional atomic action*:

```
bool TS(lock) {
     < bool initial = lock ;
         lock = true;
         return initial; >
}
```

- Effect of `TS(lock)`:

  The variable `lock` will always have value `true` after `TS(lock)`, but the return value will vary between `true` and `false`.

- Exists as an *atomic* instruction on many machines.

Critical section using TS and Spin-lock

```
bool lock = false;

process p[i=1 to n] {
    while (true) {
        while (TS(lock)) skip;
        CS
        lock = false;
        not CS
    }
}
```

Spin-lock:
Processes use TS to enter the loop.

Exit from loop happens when `lock` has value `false` before the `while` test.

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

Critical section using Test, test & Set

Lock variable `lock` is continuously written to by the processes. This can be ineffective, and a more effective solution is *test, test and set*:

```
bool lock = false;

process p[i=1 to n] {
   while (true) {
      while (lock) skip;              spin while the lock is busy
         while (TS(lock)) {           attempt to take the lock
            while (lock) skip; }      spin if it fails
      CS
      lock = false;
      not CS
   }
}
```

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

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.

What about Liveness?

So far we have not found a solution that guarantees the "Eventual Entry" property, except the very first (which did not satisfy "Absence of Unnecessary Delay").

Liveness properties

- Liveness: Something good will happen
- Typical example for sequential programs:
  Program termination
- Typical example for parallel programs:
  A given process will eventually enter the critical section

This is affected by the scheduling strategies.

Scheduling and fairness

- *Fairness:* Guarantee that the processes have an opportunity to execute.
- *Scheduling:* Strategy to determine which process has an opportunity to execute.

Example:

```
bool x = true;

co while (x); || x = false; oc
```

Unconditional fairness

A strategy for scheduling is *unconditionally fair* if each unconditional atomic action which can be chosen will eventually be chosen.

- Example: "Round robin" execution

Example:

```
bool x = true;

co while (x); || x = false; oc
```

- `x = false` is unconditional
- Must thus eventually be chosen
- This guarantees termination

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 >= 10` becomes true, this condition remains true
- This ensures termination of the program
- Example: Round robin execution

Strong fairness

A strategy for scheduling is *strongly fair* if

- it is unconditionally fair
- each conditional atomic action will eventually be chosen, assuming that the condition is true infinitely often.

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

Example:

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

With *strong fairness:*
The program will terminate, because y is true infinitely often.
With *weak fairness:*
The program need not terminate, because y is also false infinitely often.

Fairness for critical sections using locks

The CS solutions earlier in the lecture need an assumption about strong fairness in order to guarantee access for a given process (i):

- Steady inflow of processes which want the lock
- The value of `lock` alternates (infinitely long) 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 will now look at CS solutions where access is guaranteed for weakly fair strategies:

Fair solutions to the CS problem

- *Tie-Breaker*
- *Ticket*
- 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

```
bool in1 = false, in2 = false

process p1 {
     while (true) {
          while (in2) skip;
          in1 = true;
          CS
          in1 = false;
          not CS
     }
}

process p2 {
     while (true) {
          while (in1) skip;
          in2 = true;
          CS
          in2 = false;
          not CS
     }
}
```

Here we cannot guarantee *mutex*!

Tie-Breaker algorithm: Attempt 2

Tries to reverse the order in the entry protocol:

```
bool in1 = false, in2 = false
```

```
process p1 {
    while (true) {
        in1 = true;
        while (in2) skip;
        CS
        in1 = false;
        not CS
    }
}
```

```
process p2 {
    while (true) {
        in2 = true;
        while (in1) skip;
        CS
        in2 = false;
        not CS
    }
}
```

Here we cannot guarantee absence of *deadlock*!

Tie-Breaker algorithm: Attempt 3 (with await)

Introduces the variable `last` which tells which process last started the entry protocol.

```
bool in1 = false, in2 = false
int last = 1
```

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

```
process p2 {
   while (true) {
      in2 = true; last = 2;
      < await (!in1 or last==1);>
      CS
      in2 = false;
      not 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 await 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): Implementation

```
bool in1 = false, in2 = false
int last = 1

 process p1 {
    while (true) {
       in1 = true; last = 1;
       while (in2 and last==1) skip;
       CS
       in1 = false;
       not CS
    }
 }
```

Can generalize to many processes (see book)

Ticket algorithm

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 p[i = 1 to n] {
     while (true) {
          < turn[i] = number; number = number + 1; >
          < await (turn[i] = next);>
          CS
          < next = next + 1; >
          not CS
     }
}
```

- The first line in the loop must be performed atomically!
- The **await**-statement can be implemented as its own loop (while)
- Some machines have an *instruction*

  `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 p[i = 1 to n] {
     while (true) {
          turn[i] = FA(number, 1);
          while (turn[i] != next) skip;
          CS
          next = next + 1;
          not CS
     }
}
```

```
FA(var, incr):<int tmp = var; var = var + incr; return tmp;>
```

Without this instruction, we use an extra CS:

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

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

Ticket algorithm: Invariant

We can formulate a global invariant for processes in the ticket algorithm:

`0 < next ≤ number`

For each process `p[i]`:

- `turn[i] < number`
- if `p[i]` is in the CS then `turn[i] == next`.

For all processes `p[i]`, `p[j]` where `i != j`:

- if `turn[i] > 0` then `turn[j] != 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 n-tasks are done;        # Barrier
    }
}
```

All processes must reach the barrier 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 overloading of 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;
```

In a loop, the flags must be cleared before the next iteration.

Flag synchronization principles:

1. The process which waits for a flag is the one which will reset the flag
2. A flag will not be set before it is reset

Synchronization using flags

```
int arrive[1:n] = ([n] 0), continue[1:n] = ([n] 0);
```

```
process Worker[i = 1 to n] {
  while(true) {
    Task i;
    arrive[i] = 1;
    <await (continue[i] == 1);>
    continue[i] = 0;
  }
}
```

```
process Coordinator {
  while(true) {
    for[i = 1 to n] {
    <await (arrive[i] == 1);>
      arrive[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: semaphores (next lecture)