TDDC47

Real-time and Concurrent Programming

Lecture 3: Mutual exclusion (cont'd) & monitors

Simin Nadjm-Tehrani

Real-time Systems Laboratory

Department of Computer and Information Science
Linköping University

This lecture

- We will continue with presentation on Semaphores
- We move on to the next level of abstraction: Monitors
- We will return to the analysis of the methods based on busy waiting: Peterson's algorithm

Solving ME with semaphore

```
var mutex: semaphore;
(* initially 1 *)
process Pi;
  loop
     wait(mutex);
     critical_section;
     signal(mutex);
     non_critical_section;
  end
end Pi;
```


Recall: Properties

- Semaphore variable is always initialised as non-negative
- Wait and Signal are implemented as atomic operations
- Which process to wake up among all suspended ones is not specified

Spin locks

- When busy waiting is used to implement semaphore operations
- This was the original definition of wait & signal introduced by Dijkstra :

```
wait(s): while s ≤ 0 do nothing;
         s = s-1
 signal(s): s = s+1
```


Properties

- Wait and Signal are implemented as atomic operations
- Semaphore is always initialised as non-negative
- Which process to wake up among all suspended ones is not specified

How to implement?

```
process P1;     process P2;     process P3;
…               …               …
wait(s)         wait(s)         wait(s)
…               …               …
signal(s)       signal(s)       signal(s)
…               …               …
end;            end;            end;
```

Queue of suspended processes:

Semaphores vs. Busy waiting

- For long critical sections, semaphores more efficient in using CPU
- Better code organisation, less errors?
- What about reasoning about correctness, issues with deadlock and starvation?
- We will come back to these…

This lecture

- We will continue with presentation on Semaphores
- We move on to the next level of abstraction: Monitors
- We will return to the analysis of the methods based on busy waiting: Peterson's algorithm

What is a monitor?

- A programming abstraction consisting of:
  - Data structure on which programmer can define operations – to be run one at a time
  - Condition variables for synchronisation
- Encapsulates shared data that several processes can operate upon
- In addition: automatic mutual exclusion
- Pre object-orientation!

[Hoare 74]

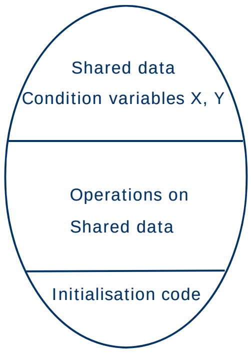
Condition variables

- Declared as special synchronisation variables:
  Condition X;
- With two designated operations:

Wait(X): suspend the calling process

Signal(X): if there are suspended processes on this variable, wake one up

Overview

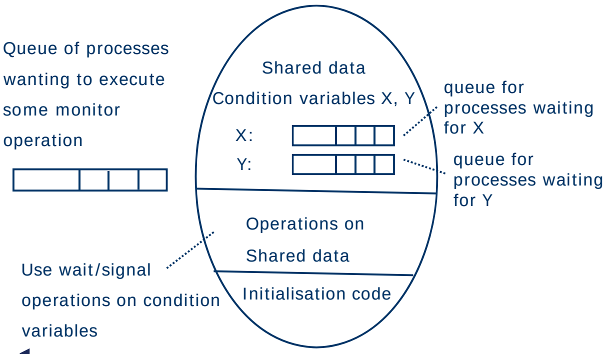


Properties

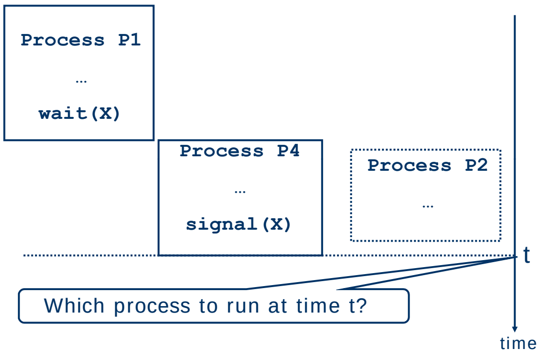
- **wait** and **signal** can be called within any of the operations

Note:
- The condition variable has no values assigned to it
- The queue associated with it is the main synchronisation mechanism
- Different semantics from semaphore operations for **wait** and **signal**

Process queues



How does it work?



Options

- Original Hoare monitor: let the woken up process (P1) continue

  What if there are several processes waiting on X?

- Pragmatic solution (Java): let the signalling process continue, and wake up P1 once P4 is suspended/exits

  P1 has to check for condition X when woken up!

Example: Bounded buffer

```
(* in some language that supports monitors *)

monitor BoundedBuffer;
Buf: array [0..SizeOfBuffer] of integer;
Base, Top: integer;
Count: integer;
NotFull, NotEmpty: condition;
operation Append(E: integer);
...
end Append;
operation Take(var E: integer);
...
end Take;
begin
<initialize> (* set Base,Top,Count to 0 *)
end BoundedBuffer;
```


Operation Append

```
operation Append (E: integer);
begin
if Count == SizeOfBuffer + 1 then
wait(NotFull);
Buff[Top] = E;
Top = (Top + 1) mod SizeOfBuffer;
Count = Count + 1;
signal(NotEmpty)
end Append;
```

Operation Take

```
operation Take (var E: integer);
begin
if Count == 0 then
wait(NotEmpty);
E = Buff[Base];
Base = (Base + 1) mod SizeOfBuffer;
Count = Count - 1;
signal(NotFull)
end Take;
```


Producer-Consumer problem

```
process Producer;
var Current:
  integer;
begin
  loop
  Produce(Current);
  Append(Current)
  end
end Producer;
```

```
process Consumer;
var Current:
  integer;
begin
  loop
  Take(Current);
  Consume(Current)
  end
end Consumer;
```


Summary

- Monitors have the same power as semaphores but are at a higher level of abstraction
  - Exercise: Try implementing producer-consumer solution with semaphores!
- Monitor has different mechanisms for handling synchronisation and for data communication
- Mutually exclusive access to data automatic, but matching waits and signals still a problem!

This lecture

- We will continue with presentation on Semaphores
- We move on to the next level of abstraction: Monitors
- We will return to the analysis of the methods based on busy waiting: Peterson's algorithm

Peterson's algorithm

```
process P1
  loop
     flag1 = up
     turn = 2
     while flag2 == up and turn == 2 do
          nothing
     end
     critical-section
     flag1 = down
     non-critical-section
  end
```

How do we show that it actually works?

Recall: last lecture

- How does one argue about correctness of Peterson's algorithm?
- Will show that
  - Processes respect mutual exclusion
  - A process will not be waiting to enter its critical section indefinitely

Questions?