

# *Synchronization#1: Semaphores and Condition Critical Regions*

©2007 Dept Computer Science, Australian National University

## *Synchronization methods*

- **Shared memory based synchronization**
  - Semaphores + 'C', POSIX — Dijkstra
  - Conditional critical regions + Edison (experimental)
  - Monitors + Modula-1, Mesa, Dijkstra, Hoare, ...
  - Mutexes & conditional variables + POSIX
  - Synchronized methods + Java
  - Protected objects + Ada95
- **Message based synchronization**
  - Asynchronous messages + e.g. POSIX, ...
  - Synchronous messages + e.g. Ada95, CHILL, Occam2
  - Remote invocation, remote procedure call + e.g. Ada95, ...
  - Synchronization in distributed systems + e.g. CORBA, ...

©2007 Dept Computer Science, Australian National University

## Synchronization in concurrent systems

- All data is declared ...
  - ... **either** local (and protected by language, O/S, or hardware-mechanisms)
  - ... **or** it is 'out in the open' and all accesses need to be synchronized!

©2007 Dept Computer Science, Australian National University

## Synchronization in concurrent system

- Synchronization: the run-time overhead?
- Is the potential overhead justified for simple data-structures:

```
int i;  
.....  
    i++;           |   if i>n {i=0;}  
    {in one thread} |   {in another thread}
```

- Are these operations atomic?
- Do we really need to introduce full featured synchronization methods here?

©2007 Dept Computer Science, Australian National University

## Synchronization in concurrent system

```
int i;  
.....  
i++;          |   if i>n {i=0;}  
{in one thread} |   {in another thread}
```

- Depending on the hardware and the compiler, it might or might not be atomic:
  - Handling a 64-bit integer on a 8- or 16-bit controller *will not be atomic* ... but perhaps it is an 8-bit integer.
  - Any manipulations on the main memory *will usually not be atomic* ... but perhaps it is a register.
  - Broken down to a load-operate-store cycle, the operations *will usually not be atomic* ... but perhaps the processor supplies atomic operations for the actual case.
- Assuming that all 'perhapses' apply: how to expand this code?

©2007 Dept Computer Science, Australian National University

## Synchronization in concurrent system

```
int i;  
.....  
i++;          |   if i>n {i=0;}  
{in one thread} |   {in another thread}
```

- Unfortunately: the chance that such programming errors occur is usually small and often some implicit by chance synchronization in the rest of the system can prevent them appearing.
- Many effects stemming from asynchronous memory accesses are interpreted as (hardware) 'glitches', since they are usually rare but then often disastrous.
- On assembler level: synchronization by employing knowledge about the atomicity of CPU-operations and interrupt structures is nevertheless possible and done frequently.
- In anything higher than assembler level on small, predictable  $\mu$ controllers:

Measures for synchronization are required!

©2007 Dept Computer Science, Australian National University

## *Synchronization by flags*

- Assuming that any access to a word in the system is an atomic operation:
    - e.g. assigning two values (not wider than the size of word) to a memory cell simultaneously:  
**Task 1:  $x := 0;$  |      **Task 2:  $x := 5;$****
- will result in **either**  $x = 0$  **xor**  $x = 5$  — and no other value is ever observable.

©2007 Dept Computer Science, Australian National University

## *Condition synchronization by flags*

```
var Flag : boolean := false;
```

process P1; statement X;		process P2; statement A;
<b>repeat until Flag;</b>		<b>Flag := true;</b>
statement Y;		statement B;
end P1;		end P2;

- Sequence of operations:  
 $[A \mid X] \Rightarrow [B \mid Y]$

©2007 Dept Computer Science, Australian National University

## Synchronization by flags

- Assuming further that there is a shared memory area between two processes:
  - A set of processes agree on a (word-size) atomic variable operating as a flag to indicate synchronization conditions.
- Memory flag method is ok for simple condition synchronization, but ...
  - ... is not suitable for general mutual exclusion in critical sections!
  - ... busy-waiting is required to poll the synchronization condition!

**More powerful synchronization operations are required for critical sections**

©2007 Dept Computer Science, Australian National University

## Semaphores (Dijkstra 1968)

- Assuming that there is a shared memory between two processes:
  - a set of processes agree on a variable  $S$  operating as a flag to indicate synchronization conditions ... *and* ...
  - an **atomic** operation  $P$  on  $S$  —  $P$  stands for 'passeren' (Dutch for 'pass'):  
$$P(S): [\text{if } S > 0 \text{ then } S := S - 1]$$
  - an **atomic** operation  $V$  on  $S$  —  $V$  stands for 'vrygeven' (Dutch for 'to release'):  
$$V(S): [S := S + 1]$$
  - the variable  $S$  is then called a **semaphore**.
- OS-level:  $P$  is usually also suspending the current task until  $S > 0$ .  
CPU-level:  $P$  indicates whether it was successful, but the operation is not blocking.

©2007 Dept Computer Science, Australian National University

## Condition synchronization by semaphores

```
var sync : semaphore := 0;
```

```
process P1;  
  statement X;
```

```
  wait (sync);
```

```
  statement Y;  
end P1;
```

```
process P2;  
  statement A;
```

```
  signal (sync);
```

```
  statement B;  
end P2;
```

- Sequence of operations:

$[A \mid X] \Rightarrow [B \mid Y]$

©2007 Dept Computer Science, Australian National University

## Mutual exclusion by semaphores

```
var mutex : semaphore := 1;
```

```
process P1;  
  statement X;
```

```
  wait (mutex);  
  statement Y;  
  signal (mutex);
```

```
  statement Z;  
end P1;
```

```
process P2;  
  statement A;
```

```
  wait (mutex);  
  statement B;  
  signal (mutex);
```

```
  statement C;  
end P2;
```

- Sequence of operations:

$[A \mid X] \Rightarrow [B \Rightarrow Y \text{ xor } Y \Rightarrow B] \Rightarrow [C \mid Z]$

©2007 Dept Computer Science, Australian National University

## Semaphores in Ada95

```
package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True (S : in out Suspension_Object);
  procedure Set_False (S : in out Suspension_Object);
  function Current_State (S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True (S : in out Suspension_Object);
private
  ... -- not specified by the language
end Ada.Synchronous_Task_Control;
```

- only one task can be blocked at Suspend\_Until\_True!  
(‘strict version of a binary semaphore’) (Program\_Error will be raised with the second task trying to suspend itself)
  - no queues! ⇒ minimal run-time overhead

©2007 Dept Computer Science, Australian National University

## Semaphores in Ada95

```
package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set_True (S : in out Suspension_Object);
  procedure Set_False (S : in out Suspension_Object);
  function Current_State (S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True (S : in out Suspension_Object);
private
  ... -- not specified by the language
end Ada.Synchronous_Task_Control;
```

- only one task can be blocked at Suspend\_Until\_True!  
(‘strict version of a binary semaphore’) (Program\_Error will be raised with the second task trying to suspend itself)
  - no queues! ⇒ minimal run-time overhead

©2007 Dept Computer Science, Australian National University

*for very special cases only,  
in general: medieval!*

## *Semaphores in POSIX*

```
int sem_init      (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy  (sem_t *sem_location);
int sem_wait     (sem_t *sem_location);
int sem_trywait  (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);
int sem_post     (sem_t *sem_location);
int sem_getvalue (sem_t *sem_location, int *value);
```

©2007 Dept Computer Science, Australian National University

## *Semaphores in POSIX*

```
int sem_init      (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy  (sem_t *sem_location);
int sem_wait     (sem_t *sem_location);
int sem_trywait  (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);
int sem_post     (sem_t *sem_location);
int sem_getvalue (sem_t *sem_location, int *value);
```

generate semaphore for usage between processes  
(otherwise for threads of the same process only)

©2007 Dept Computer Science, Australian National University

## Semaphores in POSIX

```
int sem_init      (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy  (sem_t *sem_location);
int sem_wait     (sem_t *sem_location);
int sem_trywait  (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);
int sem_post     (sem_t *sem_location);
int sem_getvalue (sem_t *sem_location, int *value);
```

delivers the number of waiting processes as a negative integer,  
if there are processes waiting on this semaphore

©2007 Dept Computer Science, Australian National University

## Semaphores in POSIX

```
void allocate (priority_t P)
{
    sem_wait (&mutex);
    if (busy) {
        sem_post (&mutex);
        sem_wait (&cond[P]);
    }
    busy = 1;
    sem_post (&mutex);
}

sem_t mutex, cond[2];
typedef emun {low, high} priority_t;
int waiting
int busy

void deallocate (priority_t P)
{
    sem_wait (&mutex);
    busy = 0;
    sem_getvalue ( &cond[high],
                  &waiting);
    if (waiting < 0) {
        sem_post (&cond[high]);
    }
    else {
        sem_getvalue ( &cond[low],
                      &waiting);
        if (waiting < 0) {
            sem_post (&cond[low]);
        }
        else {
            sem_post (&mutex);
        }
    }
}
```

©2007 Dept Computer Science, Australian National University

## Semaphores in POSIX

```

void allocate (priority_t P)
{
    sem_wait (&mutex);
    if (busy) {
        sem_post (&mutex);
        sem_wait (&cond[P]);
    }
    busy = 1;
    sem_post (&mutex);
}

sem_t mutex, cond[2];
typedef emun {low, high} priority_t;
int waiting
int busy

void deallocate (priority_t P)
{
    sem_wait (&mutex);
    busy = 0;
    sem_getvalue (&cond[high],
                  &waiting);
    if (waiting < 0) {
        sem_post (&cond[high]);
    }
    else {
        sem_getvalue (&cond[low],
                      &waiting);
        if (waiting < 0) {
            sem_post (&cond[low]);
        }
        else {
            sem_post (&mutex);
        }
    }
}

```

Correct?

©2007 Dept Computer Science, Australian National University

## Deadlock by semaphores

with Ada.Synchronous\_Task\_Control; use Ada.Synchronous\_Task\_Control;  
X, Y : Suspension\_Object;

<pre> task B; task body B is begin     ...     Suspend_Until_True (Y);     Suspend_Until_True (X);     ... end B; </pre>	<pre> task A; task body A is begin     ...     Suspend_Until_True (X);     Suspend_Until_True (Y);     ... end A; </pre>
--	--

- Could raise a Program\_Error in Ada95.
- Produces a potential **deadlock** when implemented with general semaphores.
  - Deadlocks can be generated by all kinds of synchronization methods.

©2007 Dept Computer Science, Australian National University

## *Criticism of semaphores*

- Semaphores are not bound to any resource or method or region
  - Adding or deleting a single semaphore operation some place might stall the whole system
- Semaphores are scattered all over the code
  - hard to read, error-prone
- Semaphores are considered inadequate for non-trivial systems.
  - all concurrent languages and environments offer efficient higher-level synchronization methods

©2007 Dept Computer Science, Australian National University

## *Conditional critical regions*

- Critical regions are *a set of code sections in different processes, which are guaranteed to be **executed in mutual exclusion***:
  - Shared data structures are grouped in named regions and are tagged as being private resources.
  - Processes are prohibited from entering a critical region, when another process is active in any associated critical region.
- **Condition synchronisation** is provided by *guards*:
  - When a process wishes to enter a critical region it evaluates the guard (under mutual exclusion). If the guard evaluates false, the process is suspended / delayed.
- As with semaphores, no access order can be assumed.

©2007 Dept Computer Science, Australian National University

## Conditional critical regions

```
buffer : buffer_t;
resource critical_buffer_region : buffer;

process producer;
loop
    region critical_buffer_region
    when buffer.size < N do
        -- place in buffer etc.
    end region
end loop;
end producer

process consumer;
loop
    region critical_buffer_region
    when buffer.size > 0 do
        -- take from buffer etc.
    end region
end loop;
end consumer
```

©2007 Dept Computer Science, Australian National University

## Criticism of conditional critical regions

- All guards need to be re-evaluated, when any conditional critical region is left:
  - all involved processes are activated to test their guards
  - there is no order in the re-evaluation phase
  - potential livelocks
- As with semaphores the conditional critical regions are scattered all over the code.
  - on a larger scale: same problems as with semaphores.
- The language Edison uses conditional critical regions for synchronization in a multiprocessor environment (each process is associated with exactly one processor).

©2007 Dept Computer Science, Australian National University