Australian National University

Distributed Synchronization

S01

Condition Synchronization by Flags

Semaphores

Conditional Critical Regions

Synchronization Methods

Shared memory-based synchronization

- Semaphores: C, POSIX
- Conditional critical regions: Edison (experimental)
- Monitors: Modula-1, Mesa
- Mutexes & conditional variables: POSIX
- Synchronized methods: Java, C#, …
- Protected objects: Ada
- Atomic blocks: Chapel, X10

Synchronization Methods

Message-based synchronization

- Asynchronous messages: POSIX, …
- Synchronous messages: Ada, CHILL, Occam2, MPI, …
- Remote procedure call: Ada, …

Motivation: Side Effects

Operations have side effects which are visible … either:

… locally only

(and protected by runtime-, OS-, or hardware-mechanisms) or

… outside the current process

If side effects transcend the local process then all forms of access need to be synchronized.

Motivation: Side Effects

```
int i; -- declare globally to multiple threads

// Thread 1                    // Thread 2
i++;                           if i > n {i = 0;}
```

What's the worst that can happen?

Motivation: Side Effects

```
int i; -- declare globally to multiple threads

// Thread 1
i++;
```

```
// Thread 2
if i > n {i = 0;}
```


- Handling a 64-bit integer on a 8- or 16-bit controller may not be atomic
- Unaligned manipulations on the main memory may not be atomic
- Broken down to a load-operate-store cycle, the operations will usually not be atomic
- Most schedulers interrupt threads irrespective of shared data operations
- Local caches may not be coherent
- Even if all assumptions hold: how to expand this code?

Motivation: Side Effects

- Chance synchronization in the rest of the system might prevent small programming errors from affecting correctness.
- Errors stemming from asynchronous memory accesses are often interpreted as (hardware) "glitches", since they are rare, yet disastrous.
- On assembler level on very simple CPU architectures: synchronization by exploiting knowledge of atomicity of CPU-operations and interrupt structures is possible.
- Anything higher than assembler level on single-core, predictable µ-controllers: measures for synchronization are required!

Condition Synchronization by Flags

Assumption: word-access atomicity:

i.e. assigning two values (not wider than the size of a ‘word’) to an aligned memory cell concurrently:


```
x := 0 | x := 500
```

will result in either `x = 0 or x = 500` (no other value is ever observable)

Condition 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 as a flag to indicate synchronization condition:


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



```
process P1;                          process P2;
  X;                                   A;
  repeat until Flag;                   Flag := true;
  Y;                                   B;
end P1;                              end P2;
```

"Happens-before" relations (ordering): A ⇀ B; [X | A] ⇀ Y; [X, Y | B]

Condition Synchronization by Flags

Flags are OK for simple condition synchronization, but …

- 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

Semaphores

- A set of processes agree on a shared variable `S` operating as a flag to indicate synchronization conditions
- Atomic operation `P` on `S` — for *passeren* (Dutch for 'pass'):
  `P(S): [when S > 0 then S := S - 1]` — blocking operation
  – aka: 'Wait', 'Suspend_Until_True', 'sem_wait', …
- Atomic operation `V` on `S` — for *vrijgeven* (Dutch for 'to release'):
  `V(S): [S := S + 1]`
  – aka 'Signal', 'Set-True', 'sem_post', …

E. Dijkstra *Over seinpalen*

Condition Synchronization by Semaphores

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



```
process P1;
  X;
  wait (sync);
  Y;
end P1;
```



```
process P2;
  A;
  signal (sync);
  B;
end P2;
```


"Happens-before" relations: A ⇀ B; [X | A] ⇀ Y; [X, Y | B]

Mutual Exclusion by Semaphores

```
mutex : semaphore := 1;

process P1;
  X;
  wait (mutex);
    Y; -- critical section
  signal (mutex);
  Z;
end P1;

process Q;
  A;
  wait (mutex);
    B; -- critical section
  signal (mutex);
  C;
end Q;
```


"Happens-before" relations:

$A \rightharpoonup B \rightharpoonup C$; $X \rightharpoonup Y \rightharpoonup Z$;

[X, Z | A, B, C]; [A, C | X, Y, Z];

¬[Y | B]

Semaphores in Ada

```
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! (Program_Error is raised if second task tries to suspend itself)
  - no queues!
  - minimal run-time overhead (single machine instruction)

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);
```


- `pshared` is a boolean indicating whether the semaphore is to be shared between processes
- *`value` indicates the number of waiting processes as a negative integer in case the semaphore value is zero