

# Mutual Exclusion

©2007 Dept Computer Science, Australian National University

## The general mutual exclusion scenario

- Scenario
  - $N$  processes execute (infinite) instruction sequences concurrently.
  - Each instruction belongs to either a *critical* or *non-critical section*.
- Safety property 'Mutual exclusion':
  - Instructions from critical sections of two or more processes must never be interleaved!
- More required properties:
  - **No deadlocks**: If one or multiple processes try to enter their critical sections then *exactly one* of them must succeed.
  - **No starvation**: Every process which tries to enter one of his critical sections must *succeed eventually*.
  - **Efficiency**: The decision which process may enter the critical section must be made *efficiently* in all cases, i.e. also when there is no contention.

©2007 Dept Computer Science, Australian National University

## The general mutual exclusion scenario

- Scenario
  - $N$  processes execute (infinite) instruction sequences concurrently.
  - Each instruction belongs to either a *critical* or *non-critical section*.
- Safety property ‘Mutual exclusion’:
  - Instructions from critical sections of two or more processes must never be interleaved!
- Further Assumptions:
  - Pre- and post-protocols can be executed before and after each critical section.
  - Processes may *delay infinitely in non-critical sections*.
  - Processes do *not delay infinitely in critical sections*.

©2007 Dept Computer Science, Australian National University

## Atomic load & store operations

- Assumptions:
  - every individual base memory cell (word) **load** and **store** access is atomic
  - there is *no* atomic combined **load-store** access

$G$  : Natural := 0; -- assumed to be mapped on a 1-word cell in memory

task body P1 is		task body P2 is		task body P3 is
begin		begin		begin
$G := 1$		$G := 2$		$G := 3$
$G := G + G;$		$G := G + G;$		$G := G + G;$
end P1;		end P2;		end P3;

After execution  $G$  can have *many* values between 2 and 24  
After execution,  $G$  will have *exactly one* value between 2 and 24

©2007 Dept Computer Science, Australian National University

## Mutual exclusion: first attempt

Turn: Positive range 1..2 := 1;

```
task body P1 is
begin
  loop
    -- non_critical_section_1;
    loop exit when Turn = 1; end loop;
    -- critical_section_1;
    Turn := 2;
  end loop;
end P1;
```

```
task body P2 is
begin
  loop
    -- non_critical_section_2;
    loop exit when Turn = 2; end loop;
    -- critical_section_2;
    Turn := 1;
  end loop;
end P2;
```

- Mutual exclusion
- No starvation
- No deadlock
- Locks up, if there is no contention!

©2007 Dept Computer Science, Australian National University

## Mutual exclusion: second attempt!

type Critical\_Section\_State is (In\_CS, Out\_CS);  
C1, C2: Critical\_Section\_State := Out\_CS;

```
task body P1 is
begin
  loop
    -- non_critical_section_1;
    loop
      exit when C2 = Out_CS;
    end loop;
    C1 := In_CS;
    -- critical_section_1;
    C1 := Out_CS;
  end loop;
end P1;
```

```
task body P2 is
begin
  loop
    -- non_critical_section_2;
    loop
      exit when C1 = Out_CS;
    end loop;
    C2 := In_CS;
    -- critical_section_2;
    C2 := Out_CS;
  end loop;
end P2;
```

- No mutual exclusion!

©2007 Dept Computer Science, Australian National University

## Mutual exclusion: third attempt!

```
type Critical_Section_State is (In_CS, Out_CS);  
C1, C2: Critical_Section_State := Out_CS;
```

```
task body P1 is  
begin  
  loop  
    -- non_critical_section_1;  
    C1 := In_CS;  
    loop  
      exit when C2 = Out_CS;  
    end loop;  
    -- critical_section_1;  
    C1 := Out_CS;  
  end loop;  
end P1;
```

```
task body P2 is  
begin  
  loop  
    -- non_critical_section_2;  
    C2 := In_CS;  
    loop  
      exit when C1 = Out_CS;  
    end loop;  
    -- critical_section_2;  
    C2 := Out_CS;  
  end loop;  
end P2;
```

- Mutual exclusion
- Deadlock possible!

©2007 Dept Computer Science, Australian National University

## Mutual exclusion: fourth attempt!

```
type Critical_Section_State is (In_CS, Out_CS);  
C1, C2: Critical_Section_State := Out_CS;
```

```
task body P1 is  
begin  
  loop  
    -- non_critical_section_1;  
    C1 := In_CS;  
    loop  
      exit when C2 = Out_CS;  
    end loop;  
    C1 := Out_CS;  
    -- critical_section_1;  
    C1 := In_CS;  
  end loop;  
end P1;
```

```
task body P2 is  
begin  
  loop  
    -- non_critical_section_2;  
    C2 := In_CS;  
    loop  
      exit when C1 = Out_CS;  
    end loop;  
    C2 := Out_CS;  
    -- critical_section_2;  
    C2 := In_CS;  
  end loop;  
end P2;
```

- Mutual exclusion
- No deadlock
- Individual starvation
- Global livelock

©2007 Dept Computer Science, Australian National University

## Mutual exclusion: Dekker's Algorithm

type Critical\_Section\_State is (In\_CS, Out\_CS);  
 C1, C2: Critical\_Section\_State := Out\_CS; Turn : Positive range 1..2 := 1;

<pre> task body P1 is begin   loop     -- non_critical_section_1;     C1 := In_CS;     loop       exit when C2 = Out_CS;       if Turn = 2 then         C1 := Out_CS;         loop exit when Turn = 1;       end loop;       C1 := In_CS;     end if;     end loop;     -- critical_section_1;     C1 := Out_CS; Turn := 2;   end loop; end P1; </pre>	<pre> task body P2 is begin   loop     -- non_critical_section_2;     C2 := In_CS;     loop       exit when C1 = Out_CS;       if Turn = 1 then         C2 := Out_CS;         loop exit when Turn = 2;       end loop;       C2 := In_CS;     end if;     end loop;     -- critical_section_2;     C2 := Out_CS; Turn := 1;   end loop; end P2; </pre>
--	--

©2007 Dept Computer Science, Australian National University

## Mutual exclusion: Dekker's Algorithm

type Critical\_Section\_State is (In\_CS, Out\_CS);  
 C1, C2: Critical\_Section\_State := Out\_CS; Turn : Positive range 1..2 := 1;

<pre> task body P1 is begin   loop     -- non_critical_section_1;     C1 := In_CS;     loop       exit when C2 = Out_CS;       if Turn = 2 then         C1 := Out_CS;         loop exit when Turn = 1;       end loop;       C1 := In_CS;     end if;     end loop;     -- critical_section_1;     C1 := Out_CS; Turn := 2;   end loop; end P1; </pre>	<pre> task body P2 is begin   loop     -- non_critical_section_2; </pre> <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> <ul style="list-style-type: none"> <li>• <b>Mutual exclusion</b></li> <li>• <b>No deadlock</b></li> <li>• <b>No starvation</b></li> <li>• <b>No livelock</b></li> </ul> </div> <pre>       end loop;       -- critical_section_2;       C2 := Out_CS; Turn := 1;     end loop;   end loop; end P2; </pre>
--	--

©2007 Dept Computer Science, Australian National University

## Mutual exclusion: Peterson's Algorithm

```

type Critical_Section_State is (In_CS, Out_CS);
C1, C2 : Critical_Section_State := Out_CS;
Last : Positive range 1..2 := 1;

task body P1 is
begin
loop
-- non_critical_section_1;
C1 := In_CS;
Last := 1;
loop
exit when C2 = Out_CS
or else Last /= 1;
end loop;
-- critical_section_1;
C1 := Out_CS;
end loop;
end P1;

```

```

task body P2 is
begin
loop
-- non_critical_section_2;
C2 := In_CS;
Last := 2;
loop
exit when C1 = Out_CS
or else Last /= 2;
end loop;
-- critical_section_2;
C2 := Out_CS;
end loop;
end P2;

```

©2007 Dept Computer Science, Australian National University

## Mutual exclusion: Peterson's Algorithm

```

type Critical_Section_State is (In_CS, Out_CS);
C1, C2 : Critical_Section_State := Out_CS;
Last : Positive range 1..2 := 1;

task body P1 is
begin
loop
-- non_critical_section_1;
C1 := In_CS;
Last := 1;
loop
exit when C2 = Out_CS
or else Last /= 1;
end loop;
-- critical_section_1;
C1 := Out_CS;
end loop;
end P1;

```

```

task body P2 is
begin

```

- **Mutual exclusion**
- **No deadlock**
- **No starvation**
- **No livelock**

**...and it's simpler!**

```

end P2;

```

©2007 Dept Computer Science, Australian National University

## General mutual exclusion

- Scenario
  - $N$  processes execute (infinite) instruction sequences concurrently.
  - Each instruction belongs to either a *critical* or *non-critical section*.
- Safety property 'Mutual exclusion':
  - Instructions from critical sections of two or more processes must never be interleaved!
- More required properties:
  - **No deadlocks**: If one or multiple processes try to enter their critical sections then *exactly one* of them must succeed.
  - **No starvation**: Every process which tries to enter one of his critical sections must *succeed eventually*.
  - **Efficiency**: The decision which process may enter the critical section must be made *efficiently* in all cases, i.e. also when there is no contention.

©2007 Dept Computer Science, Australian National University

## The Bakery Algorithm: Theory

- A set of  $N$  Processes  $P_1 \dots P_N$  competing for mutually exclusive execution of their critical regions.
- Every process  $P_i$  out of  $P_1 \dots P_N$  supplies: a globally readable number  $t_i$  ('ticket') initialized to '0'.
- Before a process  $P_i$  enters a critical section:
  - $P_i$  draws a new number  $t_i > t_j; \forall j \neq i$
  - $P_i$  is allowed to enter the critical section iff:  $\forall j \neq i : t_j < t_i$  or  $t_j = 0$
- After a process  $P_i$  left a critical section:
  - $P_i$  resets its  $t_i = 0$
- Issues:
  - Can you ensure that processes won't read each others ticket numbers while still calculating?
  - Can you ensure that no two processes draw the same number?

©2007 Dept Computer Science, Australian National University

## Bakery Algorithm: Implementation

type Choosing\_State is (Yes, No);

Choosing: array (1..N) of Choosing\_State := (others => No);

Number : array (1..N) of Natural := (others => 0);

task type P (I: Natural) is end P;

task body P is

begin

loop

-- non\_critical\_section\_1;

Choosing (I) := Yes;

Number (I) := Max (Number) + 1;

Choosing (I) := No;

- Solves mutual exclusion for N processors!

for J in 1..N loop

if J /= I then

loop

exit when Choosing (J) = No;

end loop;

loop

exit when

Number (J) = 0 or

Number (I) < Number (J) or

(Number (I) = Number (J)

and I < J);

end loop;

end if;

end loop;

-- critical\_section\_1;

Number (I) := 0;

end loop;

end P;

©2007 Dept Computer Science, Australian National University

## Bakery Algorithm: Implementation

type Choosing\_State is (Yes, No);

Choosing: array (1..N) of Choosing\_State := (others => No);

Number : array (1..N) of Natural := (others => 0);

task type P (I: Natural) is end P;

task body P is

begin

loop

-- non\_critical\_section\_1;

Choosing (I) := Yes;

Number (I) := Max (Number) + 1;

Choosing (I) := No;

- Intensive communication with all processes, even if just one process tries to enter!
  - particularly bad on distributed systems
- Ticket value only reset if all processors are out of mutual exclusion

for J in 1..N loop

if J /= I then

loop

exit when Choosing (J) = No;

end loop;

loop

exit when

Number (J) = 0 or

Number (I) < Number (J) or

(Number (I) = Number (J)

and I < J);

end loop;

end if;

end loop;

-- critical\_section\_1;

Number (I) := 0;

end loop;

end P;

©2007 Dept Computer Science, Australian National University

## Realistic hardware support

- Atomic **test-and-set** operations [Motorola 68xxx; Intel 80x86]:

$[L := C; C := 1]$

- Atomic **exchange** operations [Motorola 68xxx; Intel 80x86]:

$[Temp := L; L := C; C := Temp]$

- Memory cell **reservations** [Motorola PowerPC]:

$L := C$ ; – by using a special instruction, which puts a 'reservation' on  $C$

... calculate a <new value> for  $C$  ...

$C :=$  <new value>;

- succeeds iff  $C$  was not manipulated by other processors or devices since the reservation

©2007 Dept Computer Science, Australian National University

## ME with atomic test-and-set operation

type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is

```

L : Flag;
begin
  loop
    -- non_critical_section_i;
    loop
      [L := C; C := 1]
      exit when L = 0;
    end loop;
    -- critical_section_i;
    C := 0;
  end loop;
end Pi;

```

task body Pj is

```

L : Flag;
begin
  loop
    -- non_critical_section_j;
    loop
      [L := C; C := 1]
      exit when L = 0;
    end loop;
    -- critical_section_j;
    C := 0;
  end loop;
end Pj;

```

- Mutual exclusion!, No deadlock!, No global live-lock! – for  $N$  processes
- Individual starvation possible!

©2007 Dept Computer Science, Australian National University

## ME with atomic exchange operation

type Flag is Natural range 0..1; C : Flag := 0;

```
task body Pi is
  L : Flag := 1;
begin
  loop
    -- non_critical_section_i;
    loop
      [Temp := L; L := C; C := Temp];
      exit when L = 0;
    end loop;
    -- critical_section_i;
    [Temp := L; L := C; C := Temp];
  end loop;
end Pi;
```

```
task body Pj is
  L : Flag := 1;
begin
  loop
    -- non_critical_section_j;
    loop
      [Temp := L; L := C; C := Temp];
      exit when L = 0;
    end loop;
    -- critical_section_j;
    [Temp := L; L := C; C := Temp];
  end loop;
end Pj;
```

- Mutual exclusion!, No deadlock!, No global live-lock! – for  $N$  processes
- Individual starvation possible!

©2007 Dept Computer Science, Australian National University

## ME with memory cell reservation

type Flag is Natural range 0..1; C : Flag := 0;

```
task body Pi is
  L : Flag;
begin
  loop
    -- non_critical_section_i;
    loop
      L := C; -- reservation on C
      C := 1; -- works if untouched
      exit when Untouched and L = 0;
    end loop;
    -- critical_section_i;
    C := 0;
  end loop;
end Pi;
```

```
task body Pj is
  L : Flag;
begin
  loop
    -- non_critical_section_j;
    loop
      L := C; -- reservation on C
      C := 1; -- works if untouched
      exit when Untouched and L = 0;
    end loop;
    -- critical_section_j;
    C := 0;
  end loop;
end Pj;
```

- Mutual exclusion!, No deadlock!, No global live-lock! – for  $N$  processes
- Individual starvation possible!

©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’):  
 $P(S): [if\ S > 0\ then\ S := S - 1]$
  - 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**.

©2007 Dept Computer Science, Australian National University

## Semaphores (supplied by O/S)

- a set of processes  $P(1) \dots P(N)$  agree on a variable  $S$  operating as a flag to indicate synchronization conditions ... *and* ...
- an **atomic** operation Wait on  $S$ : — also: Suspend\_Until\_True, ‘sem\_wait’  
**Process  $P(i)$ : Wait ( $S$ ):**  
[if  $S > 0$   
then  $S := S - 1$   
else “suspend  $P(i)$  on  $S$ ”]
- an **atomic** operation Signal on  $S$ : — also: ‘Set\_True’, ‘sem\_post’  
**Process  $P(i)$ : Signal ( $S$ ):**  
[if  $\exists_j$ : “ $P(j)$  is suspended on  $S$ ”  
then “release  $P(j)$ ”  
else  $S := S + 1$ ] release order not defined

©2007 Dept Computer Science, Australian National University

## Type of Semaphores

- **General semaphores (counting semaphores):** non-negative number; (range limited by the system) P and V increment and decrement the semaphore by one.
- **Binary semaphores:** restricted to [0, 1]; Multiple V (Signal) calls have the same effect than 1 call.
  - binary semaphores are sufficient to create all other semaphore forms.
  - atomic 'test-and-set' operations support binary semaphores at hardware level.
- **Quantity semaphores:** The increment (and decrement) value for the semaphore is specified as a parameter with P and V.
- all types of semaphores must be initialized with a non-negative number:
  - often the number of processes which are allowed inside a critical section, i.e. "1".

©2007 Dept Computer Science, Australian National University

## Mutual Exclusion: Semaphores

S : Semaphore := 1;

```
task body Pi is
begin
  loop
    -- non_critical_section_i;
    wait (S) ;
    -- critical_section_i;
    signal (S);
  end loop;
end Pi;
```

```
task body Pj is
begin
  loop
    -- non_critical_section_j;
    wait (S);
    -- critical_section_j;
    signal (S);
  end loop;
end Pj;
```


- Mutual exclusion!, No deadlock!, No global live-lock! – for  $N$  processes
- Individual starvation possible!

©2007 Dept Computer Science, Australian National University

## Multiple Semaphores

S1, S2: Semaphore := 1;

```
task body Pi is
begin
  loop
    -- non_critical_section_i;
    wait (S1) ;
    wait (S2);
    -- critical_section_i;
    signal (S2);
    signal (S1);
  end loop;
end Pi;
```



```
task body Pj is
begin
  loop
    -- non_critical_section_j;
    wait (S2) ;
    wait (S1);
    -- critical_section_j;
    signal (S1);
    signal (S2);
  end loop;
end Pj;
```

- Mutual exclusion!, No global live-lock!
- Individual starvation possible!
- Possible deadlock!

©2007 Dept Computer Science, Australian National University

## Mutual Exclusion

- **Definition of mutual exclusion**
- **Atomic load and atomic store operations**
  - ... some classical errors
  - Decker's algorithm, Peterson's algorithm
  - Bakery algorithm
- **Realistic hardware support**
  - Atomic test-and-set, Atomic exchanges, Memory cell reservations
- **Semaphores**
  - Basic semaphore definition
  - Operating systems style semaphores

©2007 Dept Computer Science, Australian National University