Each instruction belongs to either a critical or non-critical section.

**Mutual Exclusion**

The general mutual exclusion scenario

- **Assumption 1**: every individual base memory cell (read) load and store access is atomic
- **Assumption 2**: there is no atomic combined load-store-access.

### Deadlock?

- Locks up, if there is no contention!

### Work without contention?

- No deadlock: 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 its critical sections must succeed eventually
- Efficiency: The division which process may enter the critical section must be made efficiently in all cases, i.e., also when there is no contention in the first place

### Assumption 2: there is no contention of two or more processes

- Processes do not delay indefinitely in critical sections.
Mutual Exclusion: Second attempt

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

P1
begin
loop
----- non_critical_section_1;
C1 := In_CS;
loop
exit when C1 = Out_CS;
C1 := Out_CS;
end loop;
end P1;

P2
begin
loop
----- non_critical_section_2;
C2 := In_CS;
loop
exit when C2 = Out_CS;
C2 := Out_CS;
end loop;
end P2;
```

Mutual exclusion: Third attempt

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

P1
begin
loop
----- non_critical_section_1;
C1 := In_CS;
loop
exit when C1 = Out_CS;
C1 := Out_CS;
end loop;
end P1;

P2
begin
loop
----- non_critical_section_2;
C2 := In_CS;
loop
exit when C2 = Out_CS;
C2 := Out_CS;
end loop;
end P2;
```

Mutual exclusion: Decker's Algorithm

```plaintext
type Task_Range is mod 2;
type Critical_Section_State is (In_CS, Out_CS);
CSS : array (Task_Range) of Critical_Section_State := (others => Out_CS);
Turn : Task_Range := Task_Range’First;

task One_Of_Two_Tasks
begin
loop
----- non_critical_section;
CSS (this_Task) := In_CS;
loop
exit when CSS (other_Task) = Out_CS;
if Turn = other_Task then
CSS (this_Task) := Out_CS;
end loop;
end loop;
end if;
----- critical section;
CSS (this_Task) := Out_CS;
Turn := other_Task;
end of Two_Tasks;
```

Peterson's Algorithm

```plaintext
type Task_Range is mod 2;
type Critical_Section_State is (In_CS, Out_CS);
CSS : array (Task_Range) of Critical_Section_State := (others => Out_CS);
Turn : Task_Range := Task_Range’First;

task One_Of_Two_Tasks
begin
----- non_critical_section;
CSS (this_Task) := In_CS;
loop
exit when CSS (other_Task) = Out_CS;
if Turn = other_Task then
CSS (this_Task) := Out_CS;
end loop;
end loop;
end if;
----- critical section;
CSS (this_Task) := Out_CS;
Turn := other_Task;
end of Two_Tasks;
```
Mutual Exclusion: Bakery Algorithm

Problem specification

The general mutual exclusion scenario

- N processes execute (definite) instruction sequences concurrently. Each instruction belongs to either a critical or a noncritical section.
- Safety property: Mutual exclusion: Instructions from critical sections of two or more processes must never be interleaved!
- More general properties:
  - No deadlocks: None or multiple processes try to enter their critical sections then exactly one of them must succeed.
  - No starvation: Every process that tries to enter one of its 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.

Mutual exclusion: Bakery Algorithm

No_of_Tasks : constant Positive := 2;  
Choosing : array (Task_Range) of Boolean := (others => False);  
Ticket : array (Task_Range) of Natural := (others => 0);  

| No_of_Tasks | constant Positive := 2; |
| Choosing | array (Task_Range) of Boolean := (others => False); |
| Ticket | array (Task_Range) of Natural := (others => 0); |

task type P (this_id: Task_Range);  

begin
  loop
    exit when Ticket (id) = Ticket (id) or else Ticket (this_id) < Ticket (id) or Ticket (this_id) = 0;
    exit when this_id = id;  
    end loop;
  end

The idea of the Bakery Algorithm

A set of N processes P_1, P_N competing for mutually exclusive execution of their critical sections. Every process P_i with id i, 0 ≤ i < N, supplies a globally readable number C (ticket) initialized to 0.

- Before process P_i enters a critical section:  
  \[ P_i \text{ draws a unique number } C_i \text{ s.t. } \forall j: C_i \neq C_j \text{ if } i \neq j \]  
- If it is allowed to enter the critical section iff:  
  \[ \forall j: C_i \neq C_j \text{ or } i = 0 \]

- After a process left a critical section:
  \[ P_i \text{ resets its ticket } C_i = 0 \]

- 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?

Beyond atomic memory access

Realistic hardware support

Atomic test-and-set operations:
- \[ 0 \leq C \leq W \]

Atomic exchange operations:
- \[ \text{Temp} := L \text{ or } C = \text{Temp} \]

Memory cell reservations:
- All odd quantities are read by having a special instruction which puts a 'reservation' on C
- \[ C \text{ - synchronous value} \]
- \[ C \text{ - asynchronous value} \]

Does that work?
Mutual Exclusion: Memory Cell Reservation

```plaintext
Mutual_exclusion: memory cell reservation

type Flag := 0; C : Flag := 0;

task Body Pi is
  L : Flag := 1;
  begin
    loop
      Temp := L; C := Temp;
      exit when L = 0;
      -- change process
    end loop;
    critical_section_i;
    L := 1; C := 0;
    end Pi;
  end

end

Any context switch needs to clear reservations
untouched and L = 0;
```

© 2018 Uwe R. Zimmer, The Australian National University
Mutual Exclusion

Beyond atomic hardware operations

Semicopes

Types of semaphores:

- Binary semaphores restricted to 0 (False, Ture) or 1 (False, True).
- Multiple V (Signal) calls leave the same effect as a single call.
- Any atomic operation on a semaphore must be atomic.
- Binary semaphores are sufficient to create all other semaphore forms.
- General semaphores (counting semaphores) have a non-negative number, range limited by the system, P and V return and decrement the semaphore by one.
- Quantity semaphores. The increment (and decrement) value for the semaphore is specified as an argument with P and V.

All types of semaphores must be initialized:

- The number of processes which are allowed inside a critical section, i.e. T.

Semaphore

A set of processes P_i, P_j agree on a variable r_i, n.

Count: r_i, =Count

for_leave: cmp r_i, #0

bgt end_for_leave

Any context switch needs to clear reservations

Critical section

end_for_leave:

Sema: =Sema

Sema: =Sema

Sema: =Sema

Sema: =Sema

Critical section

for_leave: cmp r_i, #0

bgt end_for_leave

Any context switch needs to clear reservations

Critical section

end_for_leave:

Sema: =Sema

Sema: =Sema

Sema: =Sema

Sema: =Sema

Critical section

for_leave: cmp r_i, #0

bgt end_for_leave

Any context switch needs to clear reservations

Critical section

end_for_leave:
Mutual Exclusion

S1, S2 : Semaphore := 1;
task body Pi is
begin
  loop
    non_critical_section_i;
    wait (S1);
    critical_section_i;
    signal (S2);
  end loop;
end Pi;
e Works too!

Summary

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