Problem specification

The 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 in the first place.

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.
Mutual Exclusion

**Mutual exclusion: Atomic load & store operations**

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

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

```pascal
G := 1  G := G + G;
end P1;

G := 2  G := G + G;
end P2;

G := 3  G := G + G;
end P3;
```

- What is the value of G?

---

**Mutual exclusion: First attempt**

```pascal
type Task_Token is mod 2;
Turn: Task_Token := 0;

task body P0 is
begin
loop
------ non_critical_section_0;
loop exit when Turn = 0; end loop;
------ critical_section_0;
Turn := Turn + 1;
end loop;
end P0;

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

Mutual exclusion?
Deadlock?
Starvation?
Work without contention?
```

---

**Atomic load & store operations**

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

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

```pascal
G := 1  G := G + G;
end P1;

G := 2  G := G + G;
end P2;

G := 3  G := G + G;
end P3;
```

- After the first global initialisation, G can have **almost any** value between 0 and 24
- After the first global initialisation, G will have **exactly one** value between 0 and 24
- After all tasks terminated, G will have **exactly one** value between 2 and 24

---

**Work without contention?**

- No deadlock!
- No starvation!
- Locks up, if there is no contention!
type Task_Token is mod 2;  
Turn: Task_Token := 0;

begin
C1, C2: Critical_Section_State := Out_CS;

loop
  if Turn = myturn then
    Turn := Turn + 1;
    if Turn = 2
      C1 := In_CS;
      C2 := In_CS;
    end if
  end if;
  into the non-critical sections
end loop;
Mutual Exclusion: Third attempt

```plaintext
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:
      C2 := Out_CS;
      end loop;
    end P1;
  end

# Mutual exclusion!
# Potential deadlock!
```

Mutual Exclusion: Decker's Algorithm

```plaintext
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:
      C2 := Out_CS;
      end loop;
    end loop
    ---- critical_section_1;
    C1 := Out_CS;
    end loop;
    end loop
    ---- critical_section_1;
    C2 := Out_CS;
    end loop;
    end loop
    end P1;
  end

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

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 type One_Of_Two_Tasks
  (this_Task : Task_Range)
  is
    other_Task : Task_Range := this_Task + 1;
    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;
          if Turn = this_Task then
            CSS (this_Task) := In_CS;
            end loop;
            CSS (this_Task) := Out_CS;
            if end loop;
            end loop;
            CSS (this_Task) := Out_CS;
            Turn := other_Task;
            end One_of_Two_Tasks;
    end
```

Mutual Exclusion: Forth attempt

```plaintext
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:
      C2 := Out_CS;
      end loop;
    end loop
    ---- critical_section_1:
    C1 := Out_CS;
    end loop;
    end loop
    ---- critical_section_1:
    C2 := Out_CS;
    end loop;
    end loop
    end P1;
  end

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

# Making any progress?
```
Mutual Exclusion

Mutual exclusion: Peterson's Algorithm

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);
Last : Task_Range := Task_Range'First;

task type One_Of_Two_Tasks
  (this_Task : Task_Range);
task body One_Of_Two_Tasks
  other_Task : Task_Range
  := this_Task + 1;
begin
  ------ non_critical_section
  CSS (this_Task) := In_CS;
  Last := this_Task;
  loop
    exit when
    CSS (other_Task) = Out_CS;
    if Turn = other_Task then
      CSS (this_Task) := Out_CS;
      loop
        exit when Turn = this_Task;
        end loop;
    CSS (this_Task) := In_CS;
    end if;
    end loop;
  ------ critical_section
  CSS (this_Task) := Out_CS;
  Turn := other_Task;
end One_Of_Two_Tasks;

Problem specification

The 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.

Mutual exclusion: Decker's Algorithm

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);
Last : Task_Range := Task_Range'First;

task type One_Of_Two_Tasks
  (this_Task : Task_Range);
task body One_Of_Two_Tasks
  other_Task : Task_Range
  := this_Task + 1;
begin
  ------ non_critical_section
  CSS (this_Task) := In_CS;
  Last := this_Task;
  loop
    exit when
    CSS (other_Task) = Out_CS;
    if Turn = other_Task then
      CSS (this_Task) := Out_CS;
      loop
        exit when Turn = this_Task;
        end loop;
    CSS (this_Task) := In_CS;
    end if;
    end loop;
  ------ critical_section
  CSS (this_Task) := Out_CS;
  Turn := other_Task;
end One_Of_Two_Tasks;
### Mutual Exclusion: Bakery Algorithm

**The idea of the Bakery Algorithm**

A set of $N$ Processes $P_1, \ldots, P_N$ competing for mutually exclusive execution of their critical regions. Every process $P_i$ out of $P_1, \ldots, P_N$ 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: $\forall j: t_i < t_j$ or $t_i = 0$
- After a process 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?

**Type definitions**

- **Task_Range** is mod No_Of_Tasks;
- **Choosing** is array (Task_Range) of Boolean := (others => False);
- **Ticket** is array (Task_Range) of Natural := (others => 0);

**Algorithm**

```
Choosing (this_id) := True;
Ticket (this_id) := Max (Ticket) + 1;
for id in Task_Range loop
  if id /= this_id then
    loop
      exit when not Choosing (id);
  end loop;
  Ticket (this_id) := 0;
end loop;
```

---

### Beyond atomic memory access

**Realistic hardware support**

**Atomic test-and-set operations:**

- $L := C; C := 1$

**Atomic exchange operations:**

- $\text{Temp} \leftarrow L; L \leftarrow C; C \leftarrow \text{Temp}$

**Memory cell reservations:**

- $L := C; \text{ – read by using a special instruction, which puts a ‘reservation’ on C}$
- ... calculate a new value for C ...
- $C := \text{<new value>}$
  - succeeds iff C was not manipulated by other processors or devices since the reservation
Mutual exclusion: atomic test-and-set operation

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

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

G Mutual exclusion!, No deadlock!, No global live-lock!
G Works for any dynamic number of processes.
G Individual starvation possible! Busy waiting loops!
```

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

G Mutual exclusion!, No deadlock!, No global live-lock!
G Works for any dynamic number of processes.
G Individual starvation possible! Busy waiting loops!

Mutual exclusion: atomic exchange operation

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

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

G Does that work?
```

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

G Does that work?
Mutual Exclusion

Mutual exclusion: memory cell reservation

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

---

task body Pi is
L : Flag;
begin
loop
L := C; C := 1;
exit when Untouched and L = 0;
------ change process
end loop;
C := 0;
end loop;
end Pi;

---

task body Pj is
L : Flag;
begin
loop
L := C; C := 1;
exit when Untouched and L = 0;
------ change process
end loop;
C := 0;
end loop;
end Pj;

---

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

---

What is the value of Count after both programs complete?
Count: .word 0x00000000
Lock: .word 0x00000000 ; #0 means unlocked

for_enter:
  ldr r3, =Lock
  mov r1, #1

for_leave:
  ldr r4, =Count
  cmp r1, #100
  bgt end_for_leave

fail_enter:
  ldr r3, [r3]
  cbnz r0, fail_enter

fail_leave:
  ldr r2, [r4]
  sub r2, #1
  str r2, [r4]

Critical section

add r1, #1
b for_leave
end_for_enter:

fail_leave:
  ldr r2, [r4]
  add r2, #1
  str r2, [r4]

Critical section

add r1, #1
b for_leave
end_for_leave:

end_for_enter:

end_for_leave:
Mutual Exclusion

Beyond atomic hardware operations

Semaphores

Basic definition (Dijkstra 1968)

Assuming the following three conditions on a shared memory cell between processes:

- a set of processes agree on a variable $S$ operating as a flag to indicate synchronization conditions
- an atomic operation $P$ on $S$ — for ‘passeren’ (Dutch for ‘pass’):
  $P(S): \text{[as soon as } S > 0 \text{ then } S := S - 1]$ this is a potentially delaying operation
- an atomic operation $V$ on $S$ — for ‘vrygeven’ (Dutch for ‘to release’):
  $V(S): \text{[if } S > 0 \text{ then } S := S + 1]$ then the variable $S$ is called a Semaphore.
Mutual Exclusion

Beyond atomic hardware operations

Semaphores

Types of semaphores:

- **Binary semaphores**: restricted to \([0, 1]\) or \([\text{False, True}]\) resp. Multiple \texttt{V} (\texttt{signal}) calls have the same effect than a single call.
- Atomic hardware operations support binary semaphores.
- Binary semaphores are sufficient to create all other semaphore forms.
- **General semaphores** (counting semaphores): non-negative number; (range limited by the system) \(P\) and \(V\) increment and decrement the semaphore by one.
- **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: often the number of processes which are allowed inside a critical section, i.e. ‘1’.

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]
Mutual Exclusion

Semaphores

\(\text{S} : \text{Semaphore} := 1;\)

task body Pi is

begin

loop

\begin{verbatim}
------- non_critical_section_i;
wait (S);
------- critical_section_i;
signal (S);
end loop;
end Pi;
\end{verbatim}

end for_leave:

end_for_enter:

\(\text{\textit{ Works?}}\)

\(\text{\textit{\textbf{Mutual Exclusion! No deadlock! No global live-lock!}}}\)

\(\text{\textit{\textbf{Works for any dynamic number of processes}}}\)

\(\text{\textit{\textbf{Individual starvation possible!}}}\)
Summary

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