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.
Mutual Exclusion

Mutual exclusion: Atomic load & store operations

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

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

Mutual Exclusion

Mutual exclusion: First attempt

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;

Mutual exclusion?
Deadlock?
Starvation?
Work without contention?
**Mutual Exclusion**

**Mutual exclusion: First attempt**

```plaintext
type Task_Token is mod 2;
Turn: Task_Token := 0;
task body P1 is
begin
  loop
    ------ non_critical_section_1;
    if Turn = myTurn then
      Scatter;
      C1 := Out_CS;
    end if
    loop
      ------ critical_section_1;
      C2 := In_CS;
      end loop;
      end P1;
  end loop;
end
```

**Critical_Section_State**

```plaintext
is
  C1, C2: Critical_Section_State := Out_CS;
```

**Turn := Turn + 1;**

**exit when C2 = Out_CS;**

**When:**

- **Mutual exclusion!**
- **No deadlock!**
- **No starvation!**
- **Inefficient!**

```plaintext
scatter:
if Turn = myTurn then
  Scatter;
  C1 := Out_CS;
end if
```

**Non-critical sections:**

```plaintext
------ non_critical_section_1;
```

**Critical sections:**

```plaintext
------ critical_section_1;
```

**exit when C2 = Out_CS;**

**end loop;**

**C1 := In_CS;**

**loop**

**C2 := In_CS;**

**end loop;**

**P1;**

```

**Critical exclusion: Second 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;
    exit when C2 = Out_CS;
    C1 := In_CS;
    ------ critical_section_1;
    C2 := Out_CS;
    end loop;
    end P1;
  end loop;
end
```

```
------ non_critical_section_2;
```

```
------ critical_section_2;
```

```
exit when C1 = Out_CS;
C2 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_1;
```

```
------ critical_section_1;
```

```
exit when C2 = Out_CS;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_2;
```

```plaintext
------ critical_section_2;
```

```
exit when C1 = Out_CS;
C1 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_1;
```

```
------ critical_section_1;
```

```
exit when C2 = Out_CS;
C2 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_2;
```

```plaintext
------ critical_section_2;
```

```
exit when C1 = Out_CS;
C1 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_1;
```

```
------ critical_section_1;
```

```
exit when C2 = Out_CS;
C2 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_2;
```

```plaintext
------ critical_section_2;
```

```
exit when C1 = Out_CS;
C1 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_1;
```

```
------ critical_section_1;
```

```
exit when C2 = Out_CS;
C2 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_2;
```

```plaintext
------ critical_section_2;
```

```
exit when C1 = Out_CS;
C1 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_1;
```

```
------ critical_section_1;
```

```
exit when C2 = Out_CS;
C2 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_2;
```

```plaintext
------ critical_section_2;
```

```
exit when C1 = Out_CS;
C1 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_1;
```

```
------ critical_section_1;
```

```
exit when C2 = Out_CS;
C2 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_2;
```

```plaintext
------ critical_section_2;
```

```
exit when C1 = Out_CS;
C1 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_1;
```

```
------ critical_section_1;
```

```
exit when C2 = Out_CS;
C2 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_2;
```

```plaintext
------ critical_section_2;
```

```
exit when C1 = Out_CS;
C1 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_1;
```

```
------ critical_section_1;
```

```
exit when C2 = Out_CS;
C2 := In_CS;
------ critical_section_2;
C2 := Out_CS;
end loop;
end P1;
end
```

```
------ non_critical_section_2;
```

```plaintext
------ critical_section_2;
```
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;
    critical_section_1;
    C1 := Out_CS;
  end loop;
  end P1;

  MutualExclusion!
  Potential deadlock!
```

Mutual exclusion: Fourth 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;
    critical_section_1;
    C1 := Out_CS;
  end loop;
  end P1;

  MutualExclusion!
  NoDeadlock!
  Potential starvation!
  Potential global livelock!
```

Mutual exclusion: Decker's Algorithm

```plaintext
type One_Of_Two_Tasks is (this_Task  : Task_Range);
type Task_Range is mod 2;
type Critical_Section_State is (In_CS, Out_CS);

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

  MutualExclusion!
  NoDeadlock!
  Potential starvation!
  Potential global livelock!
```
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);
Turn : Task_Range := Task_Range'First;
task type One_Of_Two_Tasks
    (this_Task : Task_Range);
task body One_Of_Two_Tasks is
    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
        or
        else
          Last /= this_Task;
    end loop;
    ------ critical section
    CSS (this_Task) := Out_CS;
    end

end One_Of_Two_Tasks;

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 is
    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
          or
          else
            Last /= this_Task;
      end loop;
      ------ critical section
      CSS (this_Task) := Out_CS;
      end
    end

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.
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 \( t_i' \); \( \forall j \neq i \)
  - \( P_i \) is allowed to enter the critical section if \( \forall j \neq i: t_j < t_j' \) or \( t_j = 0 \)
- After a process left a critical section:
  - \( P_i \) resets its \( t_i \) to 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?

Mutual exclusion: Bakery Algorithm

Constant Positive := ...;

No of Tasks := constant Positive := ...

Type Task_Range is mod No of Tasks;

Choosing := array (Task_Range) of Boolean := (others => False);

Ticket := array (Task_Range) of Natural := (others => 0);

Task type P (this_id: Task_Range);

Task body P is begin
  loop
    exit when Ticket (this_id) = Ticket (id) or else
      Ticket (this_id) := Ticket (id) and then this_id < id;
      end loop;
    if this_id < id then
      Ticket (this_id) := 0;
      end loop;
    end loop;
  end loop;
end P;

Beyond atomic memory access

Realistic hardware support

Atomic test-and-set operations:
- \([ L \leftarrow C; C := 1]\)

Atomic exchange operations:
- \([ \text{Temp} := L; L := C; C := \text{Temp} ]\)

Memory cell reservations:
- \([ L := C; C := \text{new value} ]\)
  - succeeds if \( C \) was not manipulated by other processors or devices since the reservation

© 2018 Uwe R. Zimmer, The Australian National University page 222 of 757 (chapter 2: "Mutual Exclusion" up to page 224)
**Mutual Exclusion**

### Mutual exclusion: atomic test-and-set operation

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

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

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

Does that work?

---

### Mutual exclusion: atomic exchange operation

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

task body Pi is
L : Flag := 1;
begi
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;

task body Pj is
L : Flag := 1;
begi
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;
```

Does that work?

---

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: memory cell reservation

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

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

Does that work?

Mutual exclusion: memory cell reservation

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

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

Mutual exclusion, No deadlock!, No global live-lock!

Works for any dynamic number of processes.

Individual starvation possible! Busy waiting loops!

Any context switch needs to clear reservations...

Mutual Exclusion... or the lack thereof

Count : Integer := 0;

task body Enter is
begin
for i := 1 .. 100 loop
Count := Count + 1;
end loop;
end Enter;
end

What is the value of Count after both programs complete?

Negotiate who goes first

Critical section

Critical section

Indicate critical section completed

ldr r4, =Count
mov r1, #1
for_enter:
cmp r1, #100
bgt end_for_enter

ldr r2, [r4]
add r2, #1
b for_enter

ldr r4, =Count
mov r1, #1
for_leave:
cmp r1, #100
bgt end_for_leave

ldr r2, [r4]
sub r2, #1
b for_leave
Mutual Exclusion

Beyond atomic hardware operations

Semaphores

Basic definition (Dijkstra 1968)

- 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’):
  \( V(S); [S := S + 1] \)

\( v \) then the variable \( S \) is called a Semaphore.

Suspended on \( S \) then release

Agree on a variable
Beyond atomic hardware operations

Semaphores

Types of semaphores:
- **Binary semaphores**: restricted to [0, 1] or [False, True] resp. Multiple V (Signal) calls have the same effect than a single call.
- Atomic hardware operations support binary semaphores.
- **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’.

```
; For enter
cmp r1, #100
bgt end_for_enter

; For leave
cmp r1, #100
bgt end_for_leave

for_enter:
for_leave:

wait_1:
ldr r0, [r3]
cbz r0, wait_1 ; if Semaphore = 0
sub r0, #1 ; dec Semaphore
str r0, [r3] ; update

wait_2:
ldr r0, [r3]
cbz r0, wait_2 ; if Semaphore = 0
sub r0, #1 ; dec Semaphore
str r0, [r3] ; update

add r1, #1
b for_enter
end_for_enter:
add r1, #1
b for_leave
end_for_leave:
```

© 2018 Uwe R. Zimmer, The Australian National University page 244 of 757 (chapter 2: "Mutual Exclusion" up to page 252)
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;

Works?

Mutual exclusion! No deadlock! No global live-lock!

Works for any dynamic number of processes

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

Works too?

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