Mutual Exclusion

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.

References for this chapter

[Ben-Ari06]
M. Ben-Ari
Principles of Concurrent and Distributed Programming
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

\[ G : \text{Natural} := 0; \quad \text{-- assumed to be mapped on a 1-word cell in memory} \]

<table>
<thead>
<tr>
<th>task body P1 is</th>
<th>task body P2 is</th>
<th>task body P3 is</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin G := 1</td>
<td>begin G := 2</td>
<td>begin G := 3</td>
</tr>
<tr>
<td>G := G + G;</td>
<td>G := G + G;</td>
<td>G := G + G;</td>
</tr>
<tr>
<td>end P1;</td>
<td>end P2;</td>
<td>end P3;</td>
</tr>
</tbody>
</table>

What is the value of \( G \)?

Mutual Exclusion

Mutual exclusion: First attempt

<table>
<thead>
<tr>
<th>type Task_Token is mod 2;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turn: Task_Token := 0;</td>
</tr>
<tr>
<td>task body P0 is</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>loop</td>
</tr>
<tr>
<td>----- non_critical_section_0;</td>
</tr>
<tr>
<td>loop exit when Turn = 0; end loop;</td>
</tr>
<tr>
<td>----- critical_section_0;</td>
</tr>
<tr>
<td>Turn := Turn + 1;</td>
</tr>
<tr>
<td>end loop;</td>
</tr>
<tr>
<td>end P0;</td>
</tr>
</tbody>
</table>

Mutual exclusion?
Deadlock?
Starvation?
Work without contention?
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;
  end loop;
end;
task body P1 is
begin
  loop
    non_critical_section_1:
    loop exit when Turn = 1; end loop;
  end loop;
end;
end

end

{ Mutual exclusion!  
No deadlock!  
No starvation!  
Inefficient!

Any better?

© 2019 Uwe R. Zimmer, The Australian National University page 210 of 757 (chapter 2: "Mutual Exclusion" up to page 252)

---

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;
  end loop;
end;
task body P2 is
begin
  loop
    non_critical_section_2:
    loop exit when C1 = Out_CS; end loop;
  end loop;
end;
end

end

{ Mutual exclusion!  
Any better?

© 2019 Uwe R. Zimmer, The Australian National University page 211 of 757 (chapter 2: "Mutual Exclusion" up to page 252)

---

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

end

{ Mutual exclusion!  
Any better?

© 2019 Uwe R. Zimmer, The Australian National University page 212 of 757 (chapter 2: "Mutual Exclusion" up to page 252)

---

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

end

{ Mutual exclusion!  
Any better?

© 2019 Uwe R. Zimmer, The Australian National University page 213 of 757 (chapter 2: "Mutual Exclusion" up to page 252)
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;
    exit when C2 = Out_CS;
    end loop;
    ------ critical_section_1;
    C1 := Out_CS;
  end loop;
end P1;

=> Mutual exclusion!
=> No deadlock!

=> Potential starvation!
=> Potential global livelock!

Mutual Exclusion: Third attempt

Mutual Exclusion: Forth 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;
    exit when C2 = Out_CS;
    end loop;
    ------ critical_section_1;
    C1 := Out_CS;
  end loop;
end P1;

=> Mutual exclusion!
=> Potential starvation!

=> Potential deadlock!

Making any progress?

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);
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;
      exit when CSS (other_Task) = Out_CS;
      if Turn = other_Task then
        CSS (this_Task) := Out_CS;
        loop
          exit when Turn = this_Task;
          CSS (this_Task) := In_CS;
          end loop;
        CSS (this_Task) := Out_CS;
        end if;
        end loop;
      ------ critical_section
      CSS (this_Task) := In_CS;
      end One_Of_Two_Tasks;
Mutual Exclusion: Peterson's Algorithm

```pascal
procedure One_Of_Two_Tasks (this_Task : Task_Range);

begin
  ---- non_critical_section
  CSS (this_Task) := In_CS;
  Last := this_Task;
  loop
    exit when CSS (other_Task) = Out_CS
    or Last /= this_Task;
  end loop;
  ---- critical section
  CSS (this_Task) := Out_CS;
end One_Of_Two_Tasks;
```

```pascal
type
  Task_Range is mod 2;
  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;

```

Mutual exclusion: Decker's Algorithm

```pascal
procedure One_Of_Two_Tasks (this_Task : Task_Range);

begin
  ---- non_critical_section
  CSS (this_Task) := In_CS;
  Last := this_Task;
  loop
    exit when CSS (other_Task) = Out_CS
    or (Turn = other_Task and
        CSS (other_Task) = Out_CS
        and
        begin
          ---- critical section
          CSS (this_Task) := Out_CS;
          TURN := other_Task;
        end
    )
  end loop;
  ---- critical section
  CSS (this_Task) := Out_CS;
end One_Of_Two_Tasks;
```

```pascal
type
  Task_Range is mod 2;
  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;
```

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

The idea of the Bakery Algorithm

A set of $N$ Processes $P_1, ..., P_N$, competing for mutually exclusive execution of their critical regions. Every process $P_i$ out of $P_1, ..., P_N$ 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 $t_j > t_i$; $\forall j \neq i$
  - $P_i$ is allowed to enter the critical section iff: $\forall j \neq i: t_j < t_i$ or $t_j = 0$
- After a process left a critical section:
  - $P_i$ resets its $t_i = 0$

Issues:
- How can you ensure that processes won’t read each other’s ticket numbers while still calculating?
- How can you ensure that no two processes draw the same number?

Mutual exclusion: Bakery Algorithm

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);

for id in Task_Range loop
  if id /= this_id then
    exit when not Choosing (id);
  end loop;
end for;

if id = this_id then
  loop
    Ticket (this_id) := Max (Ticket) + 1;
  end loop;
end if;

if Ticket (id) = 0 or else
  Ticket (this_id) < Ticket (id)
else
  Ticket (this_id) = Ticket (id)
end if;

Critical Section:

for id in Task_Range loop
  if id /= this_id then
    Ticket (this_id) := 0;
  end loop;
end for;

Ticket (this_id) := Ticket (id);

Beyond atomic memory access

Realistic hardware support

Atomic test-and-set operations:
- $L := C; C := L$

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

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

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

task body Pi is
L : Flag := 1;
begin
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
```

Does that work?

Mutual exclusion: atomic test-and-set operation

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

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

Mutual exclusion: atomic test-and-set operation

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

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

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

task body Pi is
L : Flag := 1;
begin
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
```

Does that work?
Mutual Exclusion

Mutual exclusion: memory cell reservation

```plaintext
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;
  end loop;
  C := 0;
  end loop;
end Pi;

Does that work?
```

Mutual exclusion: memory cell reservation

```plaintext
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;
  end loop;
  C := 0;
  end loop;
end Pj;

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

Count: .word 0x00000000

```plaintext
1dr r4, =Count
mov r1, #1
for_for:  
cmp r1, #100
  bgt end_for_for
end_for:
```

Mutual exclusion ... or the lack thereof

```plaintext
Count : Integer := 0;

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

  What is the value of Count after both programs complete?
```

Negotiate who goes first

```plaintext
1dr r2, [r4]
add r2, #1
str r2, [r4]
```

Indicate critical section completed

```plaintext
add r1, #1
b for_for
end_for_for:
```

Type Flag is Natural range 0..1; C : Flag := 0;
Count: .word 0x00000000  
Lock: .word 0x00000000; #0 means unlocked
  
1dr r3, =lock  
1dr r4, =count  
mov r1, #1  
for_enter:  
cmp r1, #100  
bgt end_for_enter  
fail_enter:  
1dr r0, [r3]  
cbnz r0, fail_enter; if locked  
  
1dr r2, [r4]  
add r2, #1  
str r2, [r4]  
Critical section  

end_for_enter:  
add r1, #1  
b for_enter  

Fail Leave:  
1dr r0, [r3]  
cbnz r0, fail_leave; if locked  
  
1dr r2, [r4]  
add r2, #1  
str r2, [r4]  
Critical section  

dmb; sync memory  

add r1, #1  
b for_leave  
end_for_leave:  

Any context switch needs to clear reservations

Count: .word 0x00000000  
Lock: .word 0x00000000; #0 means unlocked
  
1dr r3, =Lock  
1dr r4, =Count  
mov r1, #1  
for_enter:  
cmp r1, #100  
bgt end_for_enter  
fail_enter:  
1dr r0, [r3]  
cbnz r0, fail_enter; if locked  
  
1dr r2, [r4]  
add r2, #1  
str r2, [r4]  
Critical section  

end_for_enter:  
add r1, #1  
b for_enter  

Fail Leave:  
1dr r0, [r3]  
cbnz r0, fail_leave; if locked  
  
1dr r2, [r4]  
add r2, #1  
str r2, [r4]  
Critical section  

dmb; sync memory  

add r1, #1  
b for_leave  
end_for_leave:  

Any context switch needs to clear reservations
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’): \( V(S); [S := S + 1] \)

Then the variable \( S \) is called a Semaphore.

Light weight solution – sometimes referred to as “lock-free” or “lockless”.

© 2019 Uwe R. Zimmer, The Australian National University page 239 of 757
### Mutual Exclusion

**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.**

- **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’.

#### Code Snippet

```
for_enter:
    cmp r1, #100
    bgt end_for_enter

for_leave:
    cmp r1, #100
    bgt end_for_leave

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

    /* Critical Section */

    add r1, #1
    b for_enter
    end_for_enter:

wait_2:  
    ldr r0, [r3]
    cbz r0, wait_2 ; if Semaphore = 0
    sub r0, #1 ; dec Semaphore
    strex r0, [r3] ; try update
    cbnz r0, wait_2 ; if touched
    dmb ; sync_memory

    /* Critical Section */

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

---

© 2019 Uwe R. Zimmer, The Australian National University
Mutual Exclusion

Semaphores

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

\begin{align*}
\text{task body } & P_i \text{ is} \\
& \begin{align*}
& \text{begin} \\
& \quad \text{loop} \\
& \quad \quad \text{------ non_critical_section}_i; \\
& \quad \quad \text{wait} (S); \\
& \quad \quad \text{------ critical_section}_i; \\
& \quad \quad \text{signal} (S); \\
& \quad \text{end loop;} \\
& \text{end } P_i;
\end{align*}
\end{align*}

\( \Rightarrow \) Works?

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

\begin{align*}
\text{task body } & P_j \text{ is} \\
& \begin{align*}
& \text{begin} \\
& \quad \text{loop} \\
& \quad \quad \text{------ non_critical_section}_j; \\
& \quad \quad \text{wait} (S); \\
& \quad \quad \text{------ critical_section}_j; \\
& \quad \quad \text{signal} (S); \\
& \quad \text{end loop;} \\
& \text{end } P_j;
\end{align*}
\end{align*}

\( \Rightarrow \) Mutual exclusion? No deadlock? No global live-lock?

\( \Rightarrow \) Works for any dynamic number of processes

\( \Rightarrow \) 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!