Mutual Exclusion

Uwe R. Zimmer - The Australian National University
References for this chapter

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

健全性 properties ‘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.
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!

- 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.
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

\[
\text{task body } P1 \text{ is} \\
\text{begin} \\
\text{G := 1} \\
\text{G := G + G;} \\
\text{end } P1;
\]

\[
\text{task body } P2 \text{ is} \\
\text{begin} \\
\text{G := 2} \\
\text{G := G + G;} \\
\text{end } P2;
\]

\[
\text{task body } P3 \text{ is} \\
\text{begin} \\
\text{G := 3} \\
\text{G := G + G;} \\
\text{end } P3;
\]

What is the value of G?
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

```plaintext
begin
  G := 1
  G := G + G;
end P1;
```

```plaintext
begin
  G := 2
  G := G + G;
end P2;
```

```plaintext
begin
  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
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;

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?
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;

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

Mutual exclusion: First attempt

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

begin
  loop
    loop exit when Turn = 0; end loop,
    Turn := Turn + 1;
  end loop;
end P0;

task body P1 is
begin
  loop
    loop exit when Turn = 1; end loop,
    Turn := Turn + 1;
  end loop;
end P1;

Mutual exclusion!
No deadlock!
No starvation!
Inefficient!

scatter:
if Turn = myTurn then
  Turn := Turn + 1;
end if
into the non-critical sections
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 := 0ut_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;
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!
**Mutual Exclusion**

### 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;
    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;
```

Any better?
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!
Potential deadlock!
**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;
      C2 := In_CS; C1 := In_CS;
    end loop;
    ---- critical_section_1;
    C1 := Out_CS;
  end loop;
end P1;

Making any progress?

```
Mutual exclusion: Forth attempt

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

\begin{verbatim}
task body P1 is
begin
    loop
        ------ non_critical_section_1;
        C1 := In_CS;
    loop
        exit when C2 = Out_CS;
        C1 := Out_CS; C1 := In_CS;
    end loop;
        ------ critical_section_1;
        C1 := Out_CS;
    end loop;
    end P1;
end

\end{verbatim}

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

\begin{itemize}
  \item Mutual exclusion!
  \item No Deadlock!
  \item Potential starvation!
  \item Potential global livelock!
\end{itemize}
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;
  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

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
(task : Task_Range);

task body One_Of_Two_Tasks is
  other_Task : Task_Range := this_Task + 1;
  begin
    ------ non_critical_section

    Mutual exclusion!  No starvation!
    No deadlock!  No livelock!

    CSS (this_Task) := In_CS;
    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: 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 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 One_Of_Two_Tasks;
**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 is
  other_Task : Task_Range
    := this_Task + 1;

begin
  ------ non_critical_section

  Mutual exclusion!  No starvation!
  No deadlock!  No livelock!

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 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 $t_i > t_j \; \forall j \neq i$
  - $P_i$ is allowed to enter the critical section iff: $\forall j \neq i : t_i < t_j$ or $t_j = 0$
- 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?
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);

task type P (this_id: Task_Range);
task body P is
begin
  loop
    non_critical_section_1;
    Choosing (this_id) := True;
    Ticket (this_id) := Max (Ticket) + 1;
    Choosing (this_id) := False;
    for id in Task_Range loop
      if id /= this_id then
        loop
          exit when not Choosing (id);
        end loop;
      end if;
    end loop;
    for id in Task_Range loop
      exit when Ticket (id) = 0
        or else Ticket (this_id) < Ticket (id)
        or else (Ticket (this_id) = Ticket (id)
          and then this_id < id);
      end loop;
    end loop;
    critical_section_1;
    Ticket (this_id) := 0;
  end loop;
end P;
Mutual exclusion: Bakery Algorithm

```vba
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
      non_critical_section_1;
      Choosing (this_id) := True;
      Ticket (this_id) := Max (Ticket) + 1;
      Choosing (this_id) := False;
      for id in Task_Range loop
        if id /= this_id then
          loop
            exit when not Choosing (id);
          end loop;
        end if;
      end loop;
      critical_section_1;
      Ticket (this_id) := 0;
    end loop;
  end P;
```

Mutual exclusion!
No deadlock!
No starvation!
No livelock!
Works for N processes!

Extensive and communication intensive protocol
(even if there is no contention)
Beyond atomic memory access

Realistic hardware support

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

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

Memory cell **reservations:**
- \(L \xrightarrow{R} C\); – read by using a *special instruction*, which puts a ‘reservation’ on \(C\)
- … calculate a *<new value>* for \(C\) …
- \(C \xrightarrow{T} \text{<new value>}\);
  – succeeds iff \(C\) was not manipulated by other processors or devices since the reservation
Mutual exclusion: atomic test-and-set operation

```vhdl
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 L = 0;
    ------ change process
  end loop;
  ------ critical_section_i;
  C := 0;
end loop;
end Pi;

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

Does that work?
Mutual exclusion: atomic test-and-set operation

```ada
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 L = 0;
        ------ change process
      end loop;
      ------ critical_section_i;
      C := 0;
    end loop;
  end Pi;

  task body Pj is
    L : Flag;
    begin
      loop
        loop
          [L := C; C := 1];
          exit when L = 0;
          ------ change process
        end loop;
        ------ critical_section_j;
        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!
Mutual exclusion: atomic exchange operation

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;

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;

Does that work?
Mutual exclusion: atomic exchange operation

define 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;

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;

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

Works for any dynamic number of processes.

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;
begintask body Pj 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 loop;
end Pj;

Does that work?
Mutual exclusion: memory cell reservation

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

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

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

  task body Leave is
  begin
    for i := 1 .. 100 loop
      Count := Count - 1;
    end loop;
  end Leave;
```

What is the value of Count after both programs complete?
Negotiate who goes first

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
  str   r2, [r4]

  add   r1, #1
  b     for_enter

end_for_enter:
```

```
ldr   r4, =Count
mov   r1, #1

for_leave:
  cmp   r1, #100
  bgt   end_for_leave

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

  add   r1, #1
  b     for_leave

end_for_leave:
```

Count: .word 0x00000000
Count: .word 0x00000000
Lock: .word 0x00000000 ; #0 means unlocked

ldr r3, =Lock
ldr r4, =Count
mov r1, #1

for_enter:
cmp r1, #100
bgt end_for_enter

fail_enter:
   ldr r0, [r3]
cbnz r0, fail_enter ; if locked

   ldr r2, [r4]
   add r2, #1
   str r2, [r4]
   add r1, #1
   b for_enter

end_for_enter:

for_leave:
cmp r1, #100
bgt end_for_leave

fail_leave:
   ldr r0, [r3]
cbnz r0, fail_leave ; if locked

   ldr r2, [r4]
   sub r2, #1
   str r2, [r4]
   add r1, #1
   b for_leave

end_for_leave:
Count: `.word 0x00000000
Lock: `.word 0x00000000 ; #0 means unlocked

```
  ldr  r3, =Lock
  ldr  r4, =Count
  mov  r1, #1

for_enter:
  cmp  r1, #100
  bgt  end_for_enter

fail_enter:
  ldr  r0, [r3]
  cbnz r0, fail_enter ; if locked
  mov  r0, #1 ; lock value
  str  r0, [r3] ; lock

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

add  r1, #1
b    for_enter

end_for_enter:
```

```
  ldr  r3, =Lock
  ldr  r4, =Count
  mov  r1, #1

for_leave:
  cmp  r1, #100
  bgt  end_for_leave

fail_leave:
  ldr  r0, [r3]
  cbnz r0, fail_leave ; if locked
  mov  r0, #1 ; lock value
  str  r0, [r3] ; lock

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

add  r1, #1
b    for_leave

end_for_leave:
```
Count: \texttt{.word} 0x00000000
Lock: \texttt{.word} 0x00000000 ; #0 means unlocked

\begin{verbatim}
  ldr r3, =Lock
  ldr r4, =Count
  mov r1, #1

  \textbf{for\_enter:}
  cmp r1, #100
  bgt end\_for\_enter

  fail\_enter:
  ldrex r0, [r3]
  cbnz r0, fail\_enter ; if locked
  mov r0, #1 ; lock value
  strex r0, [r3] ; try lock
  cbnz r0, fail\_enter ; if touched
  dmb ; sync memory

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

  \textbf{Critical section}

  add r1, #1
  b for\_enter

  end\_for\_enter:

  for\_leave:
  cmp r1, #100
  bgt end\_for\_leave

  fail\_leave:
  ldrex r0, [r3]
  cbnz r0, fail\_leave ; if locked
  mov r0, #1 ; lock value
  strex r0, [r3] ; try lock
  cbnz r0, fail\_leave ; if touched
  dmb ; sync memory

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

  Critical section

  add r1, #1
  b for\_leave

  end\_for\_leave:
\end{verbatim}

Any context switch needs to clear reservations

Any context switch needs to clear reservations

Critical section

Critical section
Count: .word 0x00000000
Lock: .word 0x00000000 ; #0 means unlocked

for_enter:
  cmp r1, #100
  bgt fail_enter

fail_enter:
  ldrex r0, [r3]
  cbnz r0, fail_enter ; if locked
  mov r0, #1 ; lock value
  strex r0, [r3] ; try lock
  cbnz r0, fail_enter ; if touched
  dmb ; sync memory

  ldr r2, [r4]
  add r2, #1
  str r2, [r4]
  dmb ; sync memory
  mov r0, #0 ; unlock value
  str r0, [r3] ; unlock
  add r1, #1
  b for_enter

for_leave:
  cmp r1, #100
  bgt fail_leave

fail_leave:
  ldrex r0, [r3]
  cbnz r0, fail_leave ; if locked
  mov r0, #1 ; lock value
  strex r0, [r3] ; try lock
  cbnz r0, fail_leave ; if touched
  dmb ; sync memory

  ldr r2, [r4]
  sub r2, #1
  str r2, [r4]
  dmb ; sync memory
  mov r0, #0 ; unlock value
  str r0, [r3] ; unlock
  add r1, #1
  b for_leave

© 2018 Uwe R. Zimmer, The Australian National University
Asks for permission

Any context switch needs to clear reservations

Critical section

Critical section
Mutual Exclusion

Mutual exclusion

Count: `.word 0x00000000`

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

enter_strex_fail:
  ldrex r2, [r4] ; tag [r4] as exclusive
  add r2, #1
  strex r2, [r4] ; only if untouched
  cbnz r2, enter_strex_fail
  add r1, #1
  b for_enter

end_for_enter:
```

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

leave_strex_fail:
  ldrex r2, [r4] ; tag [r4] as exclusive
  sub r2, #1
  strex r2, [r4] ; only if untouched
  cbnz r2, leave_strex_fail
  add r1, #1
  b for_leave

end_for_leave:
```

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

G Asks for forgiveness

Any context switch needs to clear reservations
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): \begin{cases} 
  \text{as soon as } S > 0 \text{ then } S := S - 1 
  
  \end{cases}$

  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.
Beyond atomic hardware operations

Semaphores

... as supplied by operating systems and runtime environments

• a set of processes $P_1 \ldots P_N$ agree on a variable $S$ operating as a flag to indicate synchronization conditions

• an atomic operation $\texttt{Wait}$ on $S$: (aka ‘Suspend\_Until\_True’, ‘sem\_wait’, …)

  Process $P_i : \texttt{Wait} \ (S)$:

  \[
  \text{if } S > 0 \text{ then } S := S - 1 \\
  \text{else suspend } P_i \text{ on } S
  \]

• an atomic operation $\texttt{Signal}$ on $S$: (aka ‘Set\_True’, ‘sem\_post’, …)

  Process $P_i : \texttt{Signal} \ (S)$:

  \[
  \text{if } \exists P_j \text{ suspended on } S \text{ then release } P_j \\
  \text{else } S := S + 1
  \]

then the variable $S$ is called a Semaphore in a scheduling environment.
Beyond atomic hardware operations

Semaphores

Types of semaphores:

- **Binary semaphores**: restricted to \([0, 1]\) or \([\text{False, True}]\) 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’.
for_enter:
  cmp r1, #100
  bgt end_for_enter

wait_1:
  ldr r0, [r3]
  cbz r0, wait_1 ; if Semaphore = 0

add r1, #1
  b for_enter

end_for_enter:

for_leave:
  cmp r1, #100
  bgt end_for_leave

wait_2:
  ldr r0, [r3]
  cbz r0, wait_2 ; if Semaphore = 0

add r1, #1
  b for_leave

end_for_leave:
Count: .word 0x00000000
Sema: .word 0x00000001

    ldr    r3, =Sema
    ldr    r4, =Count
    mov    r1, #1

_for_enter:
    cmp    r1, #100
    bgt    end_for_enter

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

    add    r1, #1
    b      for_enter

end_for_enter:

_for_leave:
    cmp    r1, #100
    bgt    end_for_leave

    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_leave

end_for_leave:
Any context switch needs to clear reservations.

```assembly
Count: .word 0x00000000
Sema:  .word 0x00000001

ldr r3, =Sema
ldr r4, =Count
mov r1, #1

for_enter:
    cmp r1, #100
    bgt end_for_enter

wait_1:
    ldrex r0, [r3]
    cbz r0, wait_1 ; if Semaphore = 0
    sub r0, #1 ; dec Semaphore
    strex r0, [r3] ; try update
    cbnz r0, wait_1 ; if touched
    dmb ; sync memory

    ... 

    add r1, #1
    b for_enter

end_for_enter:
```

```assembly
for_leave:
    cmp r1, #100
    bgt end_for_leave

wait_2:
    ldrex 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

    ... 

    add r1, #1
    b for_leave

end_for_leave:
```
Count: .word 0x00000000
Sema: .word 0x00000001

for_enter:
  ldr r3, =Sema
  ldr r4, =Count
  mov r1, #1
  for_enter:
  cmp r1, #100
  bgt end_for_enter

wait_1:
  wait_1:
  ldr r0, [r3]
  cbz r0, wait_1 ; if Semaphore = 0
  sub r0, #1 ; dec Semaphore
  strex r0, [r3] ; try update
  cbnz r0, wait_1 ; if touched
  dmb ; sync memory

  ... Critical section ...

  ldr r0, [r3]
  add r0, #1 ; inc Semaphore
  str r0, [r3] ; update

  add r1, #1
  b for_enter

end_for_enter:

for_leave:
  ldr r3, =Sema
  ldr r4, =Count
  mov r1, #1
  for_leave:
  cmp r1, #100
  bgt end_for_leave

wait_2:
  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 ...

  ldr r0, [r3]
  add r0, #1 ; inc Semaphore
  str r0, [r3] ; update

  add r1, #1
  b for_leave

end_for_leave:

Any context switch needs to clear reservations.
Count: .word 0x00000000
Sema: .word 0x00000001

```assembly
ldr r3, =Sema
ldr r4, =Count
mov r1, #1
for_enter:
  cmp r1, #100
  bgt end_for_enter

wait_1:
  ldrex r0, [r3]
  cbz r0, wait_1 ; if Semaphore = 0
  sub r0, #1 ; dec Semaphore
  strex r0, [r3] ; try update
  cbnz r0, [r3] ; if touched
  dmb ; sync memory

... Critical section ...

signal_1:
  ldrex r0, [r3]
  add r0, #1 ; inc Semaphore
  strex r0, [r3] ; try update
  cbnz r0, signal_1 ; if touched
  dmb ; sync memory
  add r1, #1
  b for_enter

end_for_enter:
```

```assembly
ldr r3, =Sema
ldr r4, =Count
mov r1, #1
for_leave:
  cmp r1, #100
  bgt end_for_leave

wait_2:
  ldrex 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 ...

signal_2:
  ldrex r0, [r3]
  add r0, #1 ; inc Semaphore
  strex r0, [r3] ; try update
  cbnz r0, signal_2 ; if touched
  dmb ; sync memory
  add r1, #1
  b for_leave

end_for_leave:
```

Any context switch needs to clear reservations.
Semaphores

\[ S : \text{Semaphore} := 1; \]

\begin{align*}
\text{task body } \text{Pi is} \\
\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 Pi;}
\end{align*}

\begin{align*}
\text{task body } \text{Pj is} \\
\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 Pj;}
\end{align*}

Does it work?
Semaphores

S : Semaphore := 1;

```dine
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!
- Works for any dynamic number of processes
- Individual starvation possible!
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;

Works too?
Mutual Exclusion

Semaphores

S1, S2 : Semaphore := 1;

\[
\text{task body } Pi \text{ is} \\
\quad \text{begin} \\
\qquad \text{loop} \\
\qquad \quad \text{------ non_critical_section_i;} \\
\qquad \quad \text{wait } (S1); \\
\qquad \quad \text{wait } (S2); \\
\qquad \quad \text{------ critical_section_i;} \\
\qquad \quad \text{signal } (S2); \\
\qquad \quad \text{signal } (S1); \\
\qquad \text{end loop}; \\
\quad \text{end } Pi;
\]

\[
\text{task body } Pj \text{ is} \\
\quad \text{begin} \\
\qquad \text{loop} \\
\qquad \quad \text{------ non_critical_section_j;} \\
\qquad \quad \text{wait } (S2); \\
\qquad \quad \text{wait } (S1); \\
\qquad \quad \text{------ critical_section_j;} \\
\qquad \quad \text{signal } (S1); \\
\qquad \quad \text{signal } (S2); \\
\qquad \text{end loop}; \\
\quad \text{end } Pj;
\]

- Mutual exclusion!, No global live-lock!
- Works for any dynamic number of processes.
- Individual starvation possible!
- Deadlock 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