Asynchronism

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

[Patterson17]
David A. Patterson & John L. Hennessy
Computer Organization and Design – The Hardware/Software Interface
Chapter 4 “The Processor”,
Chapter 6 “Parallel Processors from Client to Cloud”
ARM edition, Morgan Kaufmann 2017
Asynchronism

Why?

*How do you handle your communication flow?*
Asynchronism

Why?

*How do you handle your communication flow?*

- Do you have times when you check certain communication?
- Is certain communication interrupting you? – at any time?
- Do you assign “importance levels” to your communication channels/sources?
STM32L476 Discovery

CPU

... running its sequence of machine instructions.
Asynchronism

STM32L476 Discovery

CPU

... running its sequence of machine instructions.

How to interact with all the other devices inside the MCU?
STM32L476 Discovery

- Current meter to MCU: 60 nA ... 50 mA
- "9 axis" motion sensor (underneath display):
  - 3 axis accelerometer
  - 3 axis gyroscope
  - 3 axis magnetometer
- Headphone jack
- USB OTG
- Microphone
- Multiplexed 24 bit \(\Sigma \Delta\)-DAC converter with stereo power amp

© 2021 Uwe R. Zimmer, The Australian National University
Asynchronism

STM32L476 Discovery

Debugger state
User LEDs
Reset
OTG LEDs
Power
Over current
LCD
Joystick
Asynchronism

STM32L476 Discovery

CPU

... running its sequence of machine instructions.

How to interact with all the other devices inside the

MCU

... and then with all the devices on the board?
STM32L476 Discovery

CPU

... running its sequence of machine instructions.

MCU

... and then with the devices inside the

rest of the world

... which is connected to the board?
Polling

*Sequential machine instructions*

- All external devices need to be “checked” by asking for their status.
- This should usually happen (semi-) regularly.
Polling

Sequential machine instructions

- All external devices need to be “checked” by asking for their status.
- This should usually happen (semi-) regularly.

- This will lead to a loop of polling requests.

- Maximal latencies can be calculated straightforward.
- Simplicity of design (with small number of devices).
- Fastest option with small number of devices (like: one).
- All devices will need to wait their turn ... even if this device is the only one with new data!
- The “main” program transforms into one large loop which can be hard to handle in terms of scalable program design.
- Events or data can be missed.
Interrupts

- One or multiple lines wired directly into the sequencer

**Required for:**
- Pre-emptive scheduling, Timer driven actions, Transient hardware interactions, ...
- Usually preceded by an external logic ("interrupt controller") which accumulates and encodes all external requests.

On interrupt (if unmasked):
- CPU stops normal sequencer flow.
- Lookup of interrupt handler’s address
- Current IP and state pushed onto stack.
- IP set to interrupt handler.
We successfully interrupted a sequence of operations ...
Interrupt processing

Interrupt handler
Interrupt processing

Interrupt handler

Program

Stack

Code

PC

...
Interrupt processing

Interrupt handler

Program

Stack

Code

- PC

Push registers
Declare local variables

…

………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………………

……………………………
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
  critical code ..
Interrupt processing

Interrupt handler

Program

Stack

Code

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
  critical code ..
Remove local variables

Registers

Local variables
Return address
Context
Parameters
Local variables
Return address
Context
Parameters
Global variables
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
critical code ..
Remove local variables
Pop registers
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
  critical code ..
Remove local variables
Pop registers

Program

Stack

Code

FP

Base

Local variables
Return address
Context
Parameters

Global variables
Return address
Context
Parameters

Bahia Honda Rail Bridge (Creative Commons Attribution-ShareAlike 3.0, Photography by MrX at English Wikipedia)
We successfully interrupted a sequence of operations …

…and now the trick to get to the other side.
Interrupt processing

Interrupt handler

Program

Interrupt handler

Stack

Code

PC

SP

FP

Base

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables
Interrupt processing

Interrupt handler

Program

Stack

Code

...
Interrupt processing

Interrupt handler

The CPU hardware (!) did that, before anything was changed
Interrupt processing

Interrupt handler

Push registers
Declare local variables

Program

Stack

Code

- PC

Push registers
Declare local variables

Interrupt processing

Interrupt handler
Interrupt processing

Interrupt handler

Program

Stack

Code

Push registers
Declare local variables
Run handler code
.. do some I/O ..
.. or run some time
critical code ..
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
.. do some I/O ..
.. or run some time
critical code ..
Remove local variables

Program

Stack

Code

Registers
Flags
PC
Local variables
Return address
Context
Parameters
Local variables
Return address
Context
Parameters
Global variables
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
critical code ..
Remove local variables
Pop registers
Interrupt processing

Interrupt handler

Push registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
  critical code ..
Remove local variables
Pop registers
Return from interrupt
Interrupt processing

Interrupt handler

Program

Stack

Code

- PC

- SP

- FP

- Base

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables
Interrupt processing

Interrupt handler

Program

Interrupt handler

Stack

Code

PC

SP

FP

- LR is loaded with a special value
Interrupt processing

Interrupt handler

Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)
Interrupt processing

Interrupt handler

Program

Stack

Code

- SP

PC

Local variables

Registers

Scratch registers

Flags

PC

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

Clear interrupt flag (Adjust priorities)
(Re-enable interrupt)
Push other registers
Declare local variables
Interrupt processing

Interrupt handler

Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)
Push other registers
Declare local variables
Run handler code

.. do some I/O ..
.. or run some time critical code ..
Interrupt processing

Interrupt handler

Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)
Push other registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
  critical code ..
Remove local variables
Pop other registers
Interrupt processing

Interrupt handler

Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)
Push other registers
Declare local variables
Run handler code
   .. do some I/O ..
   .. or run some time
   critical code ..
Remove local variables
Pop other registers
Return ("bx lr")
Interrupt processing

Interrupt handler

Program

Stack

Code

Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)
Push other registers
Declare local variables
Run handler code
  .. do some I/O ..
  .. or run some time
critical code ..
Remove local variables
Pop other registers
Return ("bx lr")
**Interrupt handler**

**Things to consider**

- Interrupt handler code can be interrupted as well.
- Are you allowing to interrupt an interrupt handler with an interrupt on the same priority level (e.g. the same interrupt)?
- Can you overrun a stack with interrupt handlers?
Interrupt handler

Things to consider

- Interrupt handler code can be interrupted as well.
- Are you allowing to interrupt an interrupt handler with an interrupt on the same priority level (e.g. the same interrupt)?
- Can you overrun a stack with interrupt handlers?

- Can we have one of those?
Multiple programs

If we can execute interrupt handler code “concurrently” to our “main” program:

Can we then also have multiple “main” programs?
Context switch

Dispatcher

Process 1

PCB

PID ... ...

Code

SP

FP

Return address

Context

Parameters

Local variables

Global variables

Stack

Process 2

PCB

PID SP ...

Code

Stack

Context-switch-variables

Registers

Flags

PC

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

Dispatcher
Context switch

Process 1

PCB

PID

…

…

…

Code

Stack

FP

Base

SP

…

Process 2

PCB

PID

SP

…

…

…

Code

Stack

Context-switch-variables

Registers

Flags

PC

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables
Context switch

Dispatcher

Push registers
Declare local variables

Process 1

PCB

Code

Stack

Context-switch-variables

Registers

Flags

PC

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

Base

Process 2

PCB

Code

Stack

Context-switch-variables

Registers

Flags

PC

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

Base
Context switch

Process 1

Dispatcher

Push registers  
Declare local variables  
Store SP to PCB 1

Process 2
Context switch

Dispatcher

Push registers
Declare local variables
Store SP to PCB 1
Scheduler
**Context switch**

- **Dispatcher**
  - Push registers
  - Declare local variables
  - Store SP to PCB 1
  - Scheduler
  - Load SP from PCB 2

**Process 1**
- PCB
  - PID
  - SP
  - Local variables
  - Return address
  - Context
  - Parameters
- Code
- Stack
- Context-switch-variables
- Registers
- Flags
- PC
- Base

**Process 2**
- PCB
  - PID
  - Local variables
  - Return address
  - Context
  - Parameters
- Code
- Stack
- Context-switch-variables
- Registers
- Flags
- PC
- Base
- Global variables
Context switch

Dispatcher

Push registers
Declare local variables
Store SP to PCB 1
Scheduler
Load SP from PCB 2
Remove local variables

Process 1

PCB

Code

Stack

Context-switch-variables

Registers

Flags

PC

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

PC

Registers

Flags

PC

Local variables

Return address

Context

Parameters

Local variables

Return address

Context

Parameters

Global variables

Base

SP

Process 2
**Context switch**

Process 1

- PCB
  - PID
  - SP
  - ...

- Code
- Stack

- Context-switch-variables
  - Registers
  - Flags
  - PC
  - Local variables
  - Return address
  - Context
  - Parameters
  - Local variables
  - Return address
  - Context
  - Parameters
  - Global variables

- PC

Process 2

- PCB
  - PID
  - ...

- Code
- Stack

- FP
- SP

- Flags
- PC
- Local variables
- Return address
- Context
- Parameters
- Local variables
- Return address
- Context
- Parameters
- Global variables

Push registers
Declare local variables
Store SP to PCB 1
Scheduler
Load SP from PCB 2
Remove local variables
Pop registers
Context switch

Dispatcher

Push registers
Declare local variables
Store SP to PCB 1
Scheduler
Load SP from PCB 2
Remove local variables
Pop registers
Return from interrupt

Process 1

PCB
PID | SP | ...

Code

Stack

Context-switch-variables
Registers
Flags
PC
Local variables
Return address
Context
Parameters
Local variables
Return address
Context
Parameters
Global variables

Process 2

PCB
PID | ...

Code

Stack

FP
Base
PID
PCB
Multi-tasking and Contention

Anything else could go wrong?
Multi-tasking and Contention

Anything else could go wrong?

-if there is neither communication nor contention between concurrent parts ... all is easy ... and boring.

What happens if concurrent programs share data?
Shared variables

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

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

task body P1 is
  begin
    G := 1
    G := G + G;
  end P1;

task body P2 is
  begin
    G := 2
    G := G + G;
  end P2;

task body P3 is
  begin
    G := 3
    G := G + G;
  end P3;
```

What is the value of G?
Assumption 1: every individual base memory cell (word) load and store access is atomic
Assumption 2: there is no atomic combined load-store access

What is the value in memory cell G after all three programs complete?
Shared variables

This is terrible!

Nobody is their right mind would analyse a program like that.

 wink ... are we missing something?

 wink ... is there an elegant way out?
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?
Mutual exclusion ... or the lack thereof

Count: .word 0x00000000

```asm
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:

end_for_leave:
```

What is the value at address Count after both programs complete?
Mutual exclusion ... or the lack thereof

Count: `word 0x00000000`

```assembly
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:
```
Mutual exclusion ... or the lack thereof

Mutual exclusion ... or the lack thereof

Count: `.word 0x00000000

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

for_enter:
  cmp   r1, #100
  bgt   end_for_enter

enter_critical_fail:
  ldrex r2, [r4] ; tag [r4] as exclusive
  add   r2, #1
  strex r0, r2, [r4] ; only if untouched
  cmp   r0, #0
  bne   enter_critical_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_critical_fail:
  ldrex r2, [r4] ; tag [r4] as exclusive
  sub   r2, #1
  strex r0, r2, [r4] ; only if untouched
  cmp   r0, #0
  bne   leave_critical_fail
  add   r1, #1
  b     for_leave

end_for_leave:
```

What is the value at address Count after both programs complete?
Mutual exclusion ... or the lack thereof

Count: .word 0x00000000

ldr r4, =Count
mov r1, #1

for_enter:
    cmp r1, #100
    bgt end_for_enter

enter_critical_fail:
    ldrex r2, [r4] ; tag [r4] as exclusive
    add r2, #1
    strex r0, r2, [r4] ; only if untouched
    cmp r0, #0
    bne enter_critical_fail
    add r1, #1
    b for_enter

end_for_enter:

for_leave:
    cmp r1, #100
    bgt end_for_leave

leave_critical_fail:
    ldrex r2, [r4] ; tag [r4] as exclusive
    sub r2, #1
    strex r0, r2, [r4] ; only if untouched
    cmp r0, #0
    bne leave_critical_fail
    add r1, #1
    b for_leave

end_for_leave:

What is the value at address Count after both programs complete?
Count: `.word 0x00000000`

```assembly
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:

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:
```

Negotiate who goes first

Critical section

Indicate critical section completed
Count: .word 0x00000000
Lock: .word 0x00000000 ; #0 means unlocked

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

  cmp r1, #100
  bgt end_for_enter

fail_lock_enter:
  ldr r0, [r3]
  cmp r0, #0
  bne fail_lock_enter ; if locked

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

  add r1, #1
  b for_enter

end_for_enter:

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

  cmp r1, #100
  bgt end_for_leave

fail_lock_leave:
  ldr r0, [r3]
  cmp r0, #0
  bne fail_lock_leave ; if locked

  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

    for_enter:
        cmp  r1, #100
        bgt  end_for_enter

    fail_lock_enter:
        ldr  r0, [r3]
        cmp  r0, #0
        bne  fail_lock_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:

    for_leave:
        cmp  r1, #100
        bgt  end_for_leave

    fail_lock_leave:
        ldr  r0, [r3]
        cmp  r0, #0
        bne  fail_lock_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:
\end{verbatim}

Critical section

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

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

#### fail_lock_enter:
- `ldrex r0, [r3]`
- `cmp r0, #0`
- `bne fail_lock_enter ; if locked`
- `mov r5, #1`
- `strex r5, r0, [r3] ; try lock`
- `cmp r5, #0`
- `bne fail_lock_enter ; if touched`
- `dmb ; sync memory`

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

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

#### fail_lock_leave:
- `ldrex r0, [r3]`
- `cmp r0, #0`
- `bne fail_lock_leave ; if locked`
- `mov r0, #1`
- `strex r5, r0, [r3] ; try lock`
- `cmp r5, #0`
- `bne fail_lock_leave ; if touched`
- `dmb ; sync memory`

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

### end_for_enter:
- `add r1, #1`
- `b for_enter`

### end_for_leave:
- `add r1, #1`
- `b for_leave`
Count: `.word 0x00000000
Lock: `.word 0x00000000 ; #0 means unlocked

for_enter:
  cmp r1, #100
  bgt end_for_enter

fail_lock_enter:
  ldrex r0, [r3]
  cmp r0, #0
  bne fail_lock_enter ; if locked
  mov r0, #1 ; lock value
  strex r5, r0, [r3] ; try lock
  cmp r5, #0
  bne fail_lock_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

end_for_enter:

for_leave:
  cmp r1, #100
  bgt end_for_leave

fail_lock_leave:
  ldrex r0, [r3]
  cmp r0, #0
  bne fail_lock_leave ; if locked
  mov r0, #1 ; lock value
  strex r5, r0, [r3] ; try lock
  cmp r5, #0
  bne fail_lock_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

end_for_leave:

Any context switch needs to clear reservations.
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

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

```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;

--------

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: memory cell reservation

\[
\text{type Flag is Natural range 0..1; C : Flag := 0;}
\]

\[
\text{task body Pi is}
L : Flag;
\begin{align*}
\text{begin} \\
\text{loop} \\
\text{loop} \\
L & : \overset{R}{=} C; C : \overset{T}{=} 1; \\
\text{exit when Untouched and } L = 0; \\
\text{------ change process} \\
\text{end loop;} \\
\text{------ critical_section_i;} \\
C := 0; \\
\text{end loop;} \\
\text{end Pi;}
\end{align*}
\]

\[
\text{task body Pj is}
L : Flag;
\text{begin} \\
\text{loop} \\
\text{loop} \\
L & : \overset{R}{=} C; C : \overset{T}{=} 1; \\
\text{exit when Untouched and } L = 0; \\
\text{------ change process} \\
\text{end loop;} \\
\text{------ critical_section_j;} \\
C := 0; \\
\text{end loop;} \\
\text{end Pj;}
\]

Does that work?
Mutual exclusion: memory cell reservation

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

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

Any context switch needs to clear reservations

task body Pj is
L : Flag;
begin
loop
  L := R * C; C := T * 1;
  exit when Untouched and 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!
Count: `.word 0x00000000`  
Lock: `.word 0x00000000`; #0 means unlocked

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

for_enter:
    cmp r1, #100  
bgt end_for_enter

fail_lock_enter:
    ldrex r0, [r3]  
cmp r0, #0  
bne fail_lock_enter ; if locked  
mov r0, #1 ; lock value  
strex r5, r0, [r3] ; try lock  
cmp r5, #0  
bne fail_lock_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

end_for_enter:

for_leave:
    cmp r1, #100  
bgt end_for_leave

fail_lock_leave:
    ldrex r0, [r3]  
cmp r0, #0  
bne fail_lock_leave ; if locked  
mov r0, #1 ; lock value  
strex r5, r0, [r3] ; try lock  
cmp r5, #0  
bne fail_lock_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

end_for_leave:
```

Any context switch needs to clear reservations.

Asks for permission.

Critical section.

© 2021 Uwe R. Zimmer, The Australian National University
Mutual exclusion ... or the lack thereof

Count: \texttt{.word 0x00000000}

\begin{verbatim}
  ldr  r4, =Count
  mov  r1, #1
  for_enter:
      cmp  r1, #100
      bgt  end_for_enter
  enter_critical_fail:
      ldrex  r2, [r4] ; tag [r4] as exclusive
      add  r2, #1
      strex  r0, r2, [r4] ; only if untouched
      cmp  r0, #0
      bne  enter_critical_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_critical_fail:
      ldrex  r2, [r4] ; tag [r4] as exclusive
      sub  r2, #1
      strex  r0, r2, [r4] ; only if untouched
      cmp  r0, #0
      bne  leave_critical_fail
      add  r1, #1
      b  for_leave
  end_for_leave:
\end{verbatim}

What is the value at address Count after both programs complete?
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’):
  
  $$P(S): \text{[as soon as } S > 0 \text{ then } S := S - 1\text{]}$$  
  ◆ this is a potentially delaying operation

• an atomic operation $V$ on $S$ — for ‘vrygeven’ (Dutch for ‘to release’):
  
  $$V(S): \text{[}S := S + 1\text{]}$$

◆ *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 \textbf{Wait} on $S$: (aka ‘Suspend\_Until\_True’, ‘sem\_wait’, …)
  
  Process $P_i : \textbf{Wait} (S)$:
  
  \[
  \text{if } S > 0 \text{ then } S := S - 1 \\
  \text{else } \text{suspend } P_i \text{ on } S
  \]

- an atomic operation \textbf{Signal} on $S$: (aka ‘Set\_True’, ‘sem\_post’, …)
  
  Process $P_i : \textbf{Signal} (S)$:
  
  \[
  \text{if } \exists P_j \text{ suspended on } S \text{ then } \text{release } P_j \\
  \text{else } S := S + 1
  \]

\textit{then} the variable $S$ is called a \textbf{Semaphore} in a scheduling environment.
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’. 
Semaphore: `.word 0x00000001

```assembly
ldr r3, =Semaphore
...
wait (Semaphore)
...
...
...
Critical section
...
signal (Semaphore)
...
...
...
```

```assembly
ldr r3, =Semaphore
...
wait (Semaphore)
...
...
...
Critical section
...
signal (Semaphore)
...
...
```
Semaphore: `.word 0x00000001`

```assembly

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

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

signal (Semaphore)

Critical section

Critical section
```

...
Semaphore: `.word 0x00000001`

- `ldr r3, =Semaphore`
  ...

**wait_1:**
- `ldrex r0, [r3]`
- `cmp r0, #0`
- `beq wait_1`; if `Semaphore = 0`
- `sub r0, #1`; dec `Semaphore`
- `strex r1, r0, [r3]`; try update
- `cmp r1, #0`
- `bne wait_1`; if touched
- `dmb wait_1`; sync memory

- ...
- ...
- ...

**wait_2:**
- `ldrex r0, [r3]`
- `cmp r0, #0`
- `beq wait_2`; if `Semaphore = 0`
- `sub r0, #1`; dec `Semaphore`
- `strex r1, r0, [r3]`; try update
- `cmp r1, #0`
- `bne wait_2`; if touched
- `dmb wait_2`; sync memory

- ...
- ...
- ...

Any context switch needs to clear reservations

Critical section

`signal (Semaphore)`

Critical section

`signal (Semaphore)`
Semaphore: .word 0x00000001

wait_1:
  ldr r3, =Semaphore
  ...
  ldrex r0, [r3]
  cmp r0, #0 ; if Semaphore = 0
  beq wait_1
  sub r0, #1 ; dec Semaphore
  strex r1, r0, [r3] ; try update
  cmp r1, #0
  bne wait_1 ; if touched
  dmb ; sync memory
  ...
  ...
  ...
  ldr r0, [r3]
  add r0, #1 ; inc Semaphore
  str r0, [r3] ; update

wait_2:
  ldr r3, =Semaphore
  ...
  ldrex r0, [r3]
  cmp r0, #0 ; if Semaphore = 0
  beq wait_2
  sub r0, #1 ; dec Semaphore
  strex r1, r0, [r3] ; try update
  cmp r1, #0
  bne wait_2 ; if touched
  dmb ; sync memory
  ...
  ...
  ...
  ldr r0, [r3]
  add r0, #1 ; inc Semaphore
  str r0, [r3] ; update
  ...
  ...

Critical section

Critical section
Semaphore: `.word` 0x00000001

```assembly
ldr  r3, =Semaphore

wait_1:
  ldrex r0, [r3]
  cmp  r0, #0
  beq  wait_1 ; if Semaphore = 0
  sub  r0, #1 ; dec Semaphore
  strex r1, r0, [r3] ; try update
  cmp  r1, #0
  bne  wait_1 ; if touched
  dmb  ; sync memory

signal_1:
  ldrex r0, [r3]
  add  r0, #1 ; inc Semaphore
  strex r1, r0, [r3] ; try update
  cmp  r1, #0
  bne  signal_1 ; if touched
  dmb  ; sync memory
```

Any context switch needs to clear reservations

```assembly
ldr  r3, =Semaphore

wait_2:
  ldrex r0, [r3]
  cmp  r0, #0
  beq  wait_2 ; if Semaphore = 0
  sub  r0, #1 ; dec Semaphore
  strex r1, r0, [r3] ; try update
  cmp  r1, #0
  bne  wait_2 ; if touched
  dmb  ; sync memory

signal_2:
  ldrex r0, [r3]
  add  r0, #1 ; inc Semaphore
  strex r1, r0, [r3] ; try update
  cmp  r1, #0
  bne  signal_2 ; if touched
  dmb  ; sync memory
```

Critical section

Critical section

Any context switch needs to clear reservations

Critical section

Critical section

Any context switch needs to clear reservations

Any context switch needs to clear reservations
Semaphores

S : Semaphore := 1;

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

\begin{verbatim}
  task body Pj is
  begin
    loop
      ----- non_critical_section_j;
      wait (S);
      ----- critical_section_j;
      signal (S);
    end loop;
  end Pj;
\end{verbatim}

Works?
Asynchronism

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;

- 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?
Semaphores

S1, S2 : Semaphore := 1;

\[
\begin{align*}
\text{task body } & \Pi \text{ is} \\
\text{begin} \\
\text{loop} \\
\quad \text{---- non\_critical\_section\_i;} \\
\quad \text{wait } (S1); \\
\quad \text{wait } (S2); \\
\quad \text{---- critical\_section\_i;} \\
\quad \text{signal } (S2); \\
\quad \text{signal } (S1); \\
\quad \text{end loop;} \\
\text{end } \Pi;
\end{align*}
\]

\[
\begin{align*}
\text{task body } & \text{ Pj is} \\
\text{begin} \\
\text{loop} \\
\quad \text{---- non\_critical\_section\_j;} \\
\quad \text{wait } (S2); \\
\quad \text{wait } (S1); \\
\quad \text{---- critical\_section\_j;} \\
\quad \text{signal } (S1); \\
\quad \text{signal } (S2); \\
\quad \text{end loop;} \\
\text{end } \Pi;
\end{align*}
\]

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

S1, S2 : Semaphore := 1;

\begin{verbatim}
  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;
\end{verbatim}

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

Concurrent programming languages offer higher abstraction and safer synchronization mechanisms.
Summary

Asynchronism

• Interrupts & Exceptions
  • Concept
  • Hardware/Software interaction
  • Recursive interrupts

• Concurrency & Synchronization
  • Race conditions
  • Synchronization
  • Passing data