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?

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"
ASM edition, Morgan Kaufmann 2017
Asynchronism

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 straight forward.
- 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.

We successfully interrupted a sequence of operations …
Asynchronism

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

We successfully interrupted a sequence of operations ...

and now the trick to get to the other side.

The CPU hardware (!) did that before anything was changed.

The Bahia Honda Rail Bridge (Creative Commons Attribution-ShareAlike 3.0, Photography by MrX at English Wikipedia)
Asynchronism

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!

Busy! Do Not Disturb!

Asynchronism

Interrupt handling

Multiple programs

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

- Can we then also have multiple "main" programs?

Asynchronism

Context switch

Dispatcher

Process 1

Process 2
Asynchronism

Context switch

Dispatcher

Push registers

Declare local variables

Store SP to PCB 1

Scheduler

Process 1

Push registers

Declare local variables

Store SP to PCB 1

Scheduler

Process 2

Push registers

Declare local variables

Store SP to PCB 2

Remove local variables

Pop registers

Return from interrupt

PCB

PC

Context-

variables

Registers

Flags

PC

PC

Context-

variables

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Return address

Context

Parameters

Global variables

Base

Local variables

Shared variables

Atomic load & store operations

Assumption 1: every individual base memory cell (word)load and store accesses 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

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

G := 0 + G;
end P2;

What is the value of G?
Asynchronism

This is terrible!

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

What is the value of Count after both programs complete?

```
Count := Count + 1;
```

```
add r1, r1, #100
```

```
for_leave:
```

```
end_for_leave:
```

```
for_enter:
```

```
end_for_enter:
```

```
ldrex
```

```
lde
```

Indicate critical section completed
### Asynchronism

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

- **Type**: Flag is Natural range 0..1; C : Flag := 0;
- **Task Body**:
  - $P_i$:
    - L : Flag;
    - begin
      - loop
        - O : C := 0; C := C + 1;
        - exit when L > 0;
        - — change process
        - end loop
        - critical_section_i;
        - C := 0;
    - end
    - Pi;
  - $P_j$:
    - L : Flag;
    - begin
      - loop
        - O : C := 0; C := C + 1;
        - exit when L > 0;
        - — change process
        - end loop
        - critical_section_j;
        - C := 0;
    - end
    - Pj;

- **Notes**:
  - Mutual exclusion! No deadlock, No global livelock!
  - 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**:
  - $P_i$:
    - L : Flag;
    - begin
      - loop
        - O : C := 0; C := C + 1;
        - exit when L > 0;
        - — change process
        - end loop
        - critical_section_i;
        - C := 0;
    - end
    - Pi;
  - $P_j$:
    - L : Flag;
    - begin
      - loop
        - O : C := 0; C := C + 1;
        - exit when L > 0;
        - — change process
        - end loop
        - critical_section_j;
        - C := 0;
    - end
    - Pj;

- **Notes**:
  - Mutual exclusion! No deadlock, No global livelock!
  - 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**:
  - $P_i$:
    - L : Flag;
    - begin
      - loop
        - O : C := 0; C := C + 1;
        - exit when L = 0;
        - — change process
        - end loop
        - critical_section_i;
        - C := 0;
    - end
    - Pi;
  - $P_j$:
    - L : Flag;
    - begin
      - loop
        - O : C := 0; C := C + 1;
        - exit when L = 0;
        - — change process
        - end loop
        - critical_section_j;
        - C := 0;
    - end
    - Pj;

- **Notes**:
  - Mutual exclusion! No deadlock, No global livelock!
  - Works for any dynamic number of processes.
  - Individual starvation possible! Busy waiting loops!
Mutual exclusion ... or the lack thereof

Beyond atomic hardware operations

Semaphores

Basic definition (Dijkstra 1968): A semaphore is an integer that needs to clear reservations as soon as \( S > 0 \) then \( S := S - 1 \) for 'vrygeven' (Dutch for 'to release').

\[
\begin{align*}
\text{Critical section} & : [S := S + 1] \\
\text{wait_1} & : [\text{Signal}(S)] \\
\text{wait_2} & : [\text{Signal}(S)] \\
\text{end_for_enter} & : [S := S + 1] \\
\text{end_for_leave} & : [\text{Signal}(S)]
\end{align*}
\]

Types of semaphores:

- **Binary semaphores**: restricted to \([0, 1]\) or \([\text{False, True}]\) resp.
- **Quantity semaphores**: The increment (and decrement) value for the variable \( S \) — for 'vrygeven' (Dutch for 'to release').

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

- An atomic operation \( P \text{.P; P; P; P} \) agree on available semaphore values to indicate synchronization conditions.
- An atomic operation \( \text{P; P; P; P} \) agree on available semaphore values to indicate synchronization conditions.
- An atomic operation \( \text{P; P; P; P} \) agree on available semaphore values to indicate synchronization conditions.

\[
\begin{align*}
\text{P; P; P; P} & : \text{wait(S)} \quad \text{if} \quad S > 0 \quad \text{then} \quad S := S - 1 \quad \text{else suspend} \quad P \quad \text{on} \quad S \\
\text{P; P; P; P} & : \text{wait(S)} \quad \text{if} \quad S > 0 \quad \text{then} \quad S := S - 1 \quad \text{else suspend} \quad P \quad \text{on} \quad S
\end{align*}
\]

Semaphores

... as supplied by operating systems and runtime environments.

- A set of processes \( P \) agree on available semaphore values as a flag to indicate synchronization conditions.
- An atomic operation \( \text{P; P; P; P} \) agree on available semaphore values as a flag to indicate synchronization conditions.
- An atomic operation \( \text{P; P; P; P} \) agree on available semaphore values as a flag to indicate synchronization conditions.

Semaphore: \( \quad \text{wait}(S) \quad \text{if} \quad S > 0 \quad \text{then} \quad S := S - 1 \quad \text{else suspend} \quad P \quad \text{on} \quad S \)

Semaphore: \( \quad \text{wait}(S) \quad \text{if} \quad S > 0 \quad \text{then} \quad S := S - 1 \quad \text{else suspend} \quad P \quad \text{on} \quad S \)
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

S1, S2 : Semaphore := 1;

task body Pi is
begin
  loop
    non_critical_section_i;
    wait (S1);
    critical_section_i;
    signal (S1);
  end loop;
end

e Works!

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

Works for any dynamic number of processes.

Individual starvation possible!

Deadlock possible!