Introduction to real time systems

Fabien Calcado, Thomas Megel

Email: *fabien.calcado@gmail.com*
thomas.megel@thalesgroup.com

Presentation outlines

- **Reminder on fundamentals of Operating systems**
- **Real time concepts**
- **Architecture of real time systems**
- **Scheduling in real time systems**
  - Scheduling of independent tasks
  - Scheduling of dependent tasks on mono and multi processor systems

Course content (n°1)

- **Reminders**
  - Background on software development
  - Multitask systems
- **Parallelism management (reminders /supplements)**
  - Communication and control of concurrency
    - Mutex / semaphore

Outlines

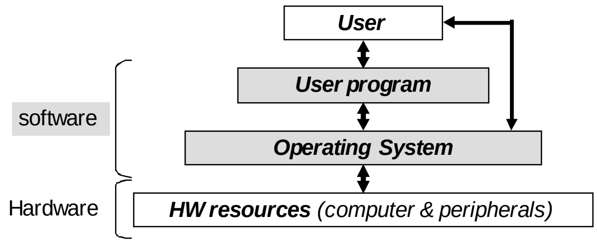
- **Background**
- **Multitask system**
- **Parallelism management**

Background

- **The primary purpose of an information processing system is to achieve a mission :**
  - Implementation of functions
  - *Set* of coded instructions (program)
  - Coded and organized informations (data)
- **Design**
  - Necessity to use hardware resources
    - HW => static configuration of resources
  - Softwares
    - Specific SW

Background

- **Computer science systems**



Background

- **Operating System**
  - Set of programs to execute and to manage « physical resources » of a computer
    - **To drive** (software-driven) computer elements and **to coordinate** exchanges of information
    - **To execute** high level **commands** from user (direct commands) or from applications launched by user (indirect commands)
    - **To secure**, it forbids actions form user that could threaten its integrity

Background

- **Software quality**
  - Efficiency
    - To execute functions required with corresponding performance
  - Reliability
    - Correctness, complete and safe
  - Testability
    - understandable, readable, organized, self-describing
  - Portable
    - On different platforms
  - Maintainability
    - corrections
  - Reuse
    - For product policy
  - Certifiable
    - By providing proofs of its correct behavior

Background

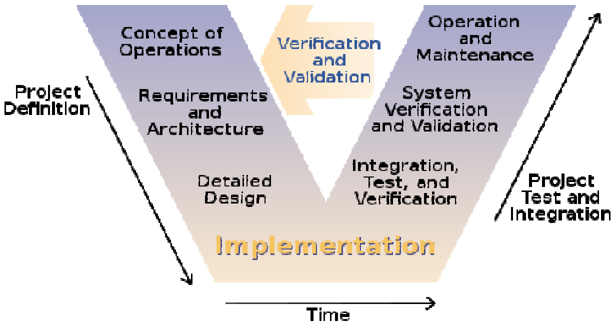
- **It implies:**
  - A design compliant to the requirements of the mission
  - A implementation compliant to the design
- **To analyze the specifications:**
  - With methods and rules
  - With basic techniques
  - Design/ programming / implementation

Background

- **Modular approach :**
  - Allows a reduced complexity of the problem
  - Allows to divide the workload
- **Methods to ensure coherency of this approach:**
  - Logical coherency
    - Categorization of problems, hardware or software
  - Temporal coherency
    - synchronization, sequencing of instructions (computing)
  - Procedural coherency
    - Algorithms organization
  - Data coherency for common part
    - Object oriented
  - Functional coherency
    - 1 functionality for each module

Background

- **V-model:**



Background

- **V-model :**
  - Need analysis
    - **Experts of usage domain**
    - **Environment, role, resources, requirements**
      - » What do we want? At which cost?
  - Global specification (functional)
    - **Set of requirements**
    - **Description of the system, not related to its implementation**
      - » Expected outputs of the system with specific inputs ➔ What it has to do ?
  - Design (detailed architecture)
    - **Decomposition of software, interface specification, description of the component design**
      - » How will it do it ?

Background

- **V-model :**
  - Programming
    - Realization step
      - » Far to be the most important part
  - Unit test
    - To ensure the correct behavior of a module
      - » To be compliant of individual specification
  - Integration and test
    - To gather all modules to validate the overall system

Background

- **V-model**
  - Verification and validation (Software)
    - Compliance to the needs, to meet the requirements
    - Analysis, tests
    - Software errors
      - » Most errors came from a wrong design
      - » Most errors are revealed by the customers
    - Cost
      - » The software development represents of a big part of the overall cost
      - » ~ 40 à 60% of expenses can be related to tests and correction of the software!

Background

- **Examples of catastrophic failures due to bugs**
  - In 1996, Ariane 5 rocket had exploded during the flight
    - The Navigation system used was identical than the one used in Ariane 4 but not tests on Ariane 5…
      - » 800 000 frs savings on the preparating cost
  - In 2000, in medicine, a program to measure radiation has provided wrong values
    - It costs the life to 8 patients and ~20 people lightly injured
  - In 2009, dozen thousand bank accounts of customer from BNP Paribas have been credited by error
  - In 2013, Toyota throttle sw design causes the death to several dozen of people

Outlines

- **Background**
- **Multitask system**
- **Parallelism management**

Multitask programming

- **System**
  - contains :
    - Several resources
      - » CPU(s), memory, hard drives, network cards …
    - Each can deliver one function at a time
  - To realize :
    - several functionalities
    - Application = 1 or several tasks (1 or several functions)
      - » Independent or not
      - » With different occurrences of release or not entirely defined

➔Need to <u>share</u> resources between different tasks to expose parallelism

Multitask programming

- **System**
  - Software architecture
    - Set of tasks (programs) to execute concurrently
  - Hardware architecture
    - Set of restricted computing resources (CPUs) which are interconnected
      - » mono/multi processor architecture
      - » shared or distributed memory architecture

Multitask programming

- **System implementation**
  - It consists in <u>allocating tasks</u> on several resources <u>over the time</u>
    - Allocating over the time is called tasks scheduling
    - Real time context➔ scheduling shall satisfy any temporal constraints of a set of tasks
  - Terminology
    - The **scheduling** is the management of the tasks' execution on the resources of the system
      - » sequencing, interleaving…
    - The **scheduling policy** is the rule to organize the execution of tasks

Multitask programming

- **Monoprocessor case**
  - A computer:
    - 1 processor, a memory and other peripheral resources
  - Execute anything in <u>one task</u> (loop programming)
    - Cyclic programming : only one release of task
  - Advantages :
    - Easy to implement
    - Simple verification (deterministic)
  - Drawbacks :
    - Slow and complex design
    - Weak usage of resources
    - Weak scalability and reuse

Multiprocessor case

- A program needs
  - One or several virtual processor (process/thread)
  - A virtual memory (addressing space)
  - Virtual resources
  - UNIX process example
    - » 1 process = 1 « virtual processor »
- To execute several program in parallel
- **OS** is the program in charge of multi-programming
  - Program isolation (partitioning property)
  - <u>Resources sharing</u>

Program isolation

- To prevent unexpected failure of a program
  - Isolation of memory access (MMU)
  - Resource accesses secured
    - » System services (kernel)
    - » Ensure a correct use of resources
  - Defensive programming
    - » Check of deadline miss (via watchdog), interrupts management
- Safety and security → similar conception
- Safety ≠ perfect system (too expensive)

Resources sharing

- « spatial » allocation
  - Possible if there are several resources
    - » CPUs, Memory…
- « temporal » allocation
  - Over the time
  - Mandatory if there is only one resource
    - » one CPU, hard drive, one serial port…

→CPU(s) scheduling : manage task execution on one or several processors of the system

Crucial needs

- Communication between tasks
  - Data transfers
- Synchronization of tasks
  - Add a constraint to the scheduling and to the instruction sequencing

Most important properties

- Data coherency
- Execution **determinism**
  - Independently of the task parallelism

Sources of non-coherence

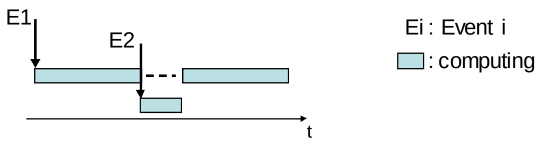
- Do not come from parallelism…
- … but from interactions between programs <u>executed in parallel</u>
  - Shared memory
  - Shared resources
  - Communications (sequencing)…
- Example : race condition

Interaction model with interfaces

- Polling
  - Regularly send requests to peripheral(s)
  - Implemented with an infinite loop
  - Advantage
    - » Easy to implement
  - Drawbacks
    - » No scalability if too many instances
    - » Unavailable data ➔ waste of time requesting it to the driver of a peripheral

Interaction model with interfaces

- Interrupt-based interactions
  - Event causing a change in the execution of a program
    - » Need to handle different time scales
    - » Input / Output interruption, clocks, external signals (watchdog)
  - Illustration
    - » Execution related to E2 with higher priority than E1



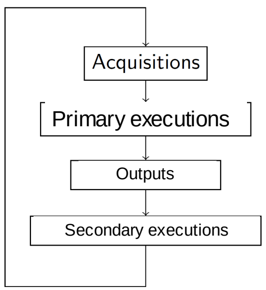
Interaction model with interfaces

- Interrupt-based interactions
  - Advantages
    - » Large flexibility
    - » Easy-medium to implement
    - » Possible optimization
  - Drawbacks
    - » Data coherency (interleaving)
    - » Feasibility (miss of important timing constraint)
    - » Resources sharing (deadlock / livelock problem)

- **Interaction model with interfaces**
  - Interrupt-based interactions
    - Can be related to exception (faults, trap, abort)
      - » Internal causes of a program
      - » Example : erroneous instruction, access to unimplemented memory zone, zero division,…
      - » Be careful of *Out of Order processor (OoO)*

- **Interaction model**
  - Multitask system case
    - Several tasks (programs or sequence of instructions)
    - Switching of tasks
      - » To halt a task (e.g. in waiting) to execute another one
      - » Interruption (periodic) triggered by a timer (clock)

- **Loop programming**
  - To avoid problem related to multi-task paradigm
    - Static control flow
    - No preemption



- **Loop programming**
  - Advantages
    - « easy » to implement
    - Cycle accurate
  - Drawbacks
    - Not flexible
    - Not optimal
    - Slow
  - Example : three tasks A, B, C
    - A can be divided in A1 and A2
    - C can be divided in C1, C2 and C3
      - » Dependency problem with respect to processor speed !

Multitask programming

- **Parallel composition of a program**

$$P = P_1 * P_2$$

– Let P_1 et P_2 known, what can we say about P ?
– To characterize explicit or implicit interactions
  - Asynchronous case: product possible or not
  - Synchronous case: synchronous product of automatons
– Loop programming provides a non flexible composition but easy to implement, with low performances
– An interrupt can lead to a « desynchronization »

Outlines

- **Background**
- **Multitask system**
- **Parallelism management**

Parallelism management

- **Example with bank account update:**

val: INTEGER

```
PROCESS Creditor (c: INTEGER){
[4]      val ← val + c
}
```

```
PROCESS Debtor (d: INTEGER){
[1] if val < d then
[2]       Write (« overdraft »)
    endif
[3] val ← val – d
}
```

– Problem if we execute:
  - Val = 5, debtor(6), creditor(4)
  - Sequence: [1], [4], [2], [3]... Overdraft notified ! (val = 3)
– Problem if we execute :
  - Val = 5, debtor(4) in // debtor(3)
  - Sequence : [1a], [1b], [3a], [3b] …. No overdraft found ! (val = -2)

Parallelism management

- **Interactions between programs (reminder)**
  – Problems in multi-task systems are related to the interactions between tasks executed in parallel and not related to parallelism
  – Resource sharing
    - To ensure that the parallel execution of several tasks leads to the same outputs than a sequential execution of them
  – Communication
    - To ensure that a well-defined protocol exists and is strictly applied to share informations between programs

Parallelism management

- **Why synchronize?**
  - To solve memory coherency problems for the data communication (shared memory)
  - To specify dependency between task executions
    - To control task execution order
      - » Ex : producer / consumer (ease the control of a thread to another one is running)
      - » Ex: peripheral commands / hardware (to ensure we do not send two contrary orders to the same controller)
  - Generally: to solve race conditions on a shared resource
    - Software or hardware

Parallelism management

- **Communication mechanisms**
  - Shared memory, FIFO pipes, asynchronous mailbox, circular buffer…
  - A shared memory zone is mandatory to realize a communication between two tasks
    - Can be hidden by the kernel
      - » Important mechanism to implement
  - Be careful of « low level » problems
    - A C language instruction = several assembly instructions!
      - » Example n°1: a variable, two tasks
        - ➔ the first one adds, the other one subtracts

Parallelism management

- **Definition : critical section**
  - Task entering in a code sequence using resources which can be used by other tasks but not at the same time with the other ones
    - A common example of shared resource is a set of memory blocks
    - To ensure a specific part of code is executed in a sequential way
  - Be careful with critical sections, they penalizes the parallelism rate
    - One must try to minimize their use

Parallelism management

- **Definitions of seriability et atomicity**
  - A and B are two (computing) tasks
  - Seriability
    - A // B independent of the scheduling
    - A // B = A,B = B,A
  - A is atomic for B if
    - A cannot be in // with B
    - A cannot be preempted in favour of B
    - B cannot observe intermediary states of A during its execution
    - A takes zero duration in B point of view

- **Remarks**
  - Atomicity periods decrease the parallelism rate
    - They shall not imply deadline misses
    - They shall be short on multicore processor
  - Atomicity avoids some interactions
    - Do not solve A,B = B,A
    - Example : parallel decomposition of code for Morse application

- **Remarks**
  - Example : to encode Morse code in parallel
    - Chain to encode is « SOS »
    - « S » = « ... » , « O » = « - - - »
    - Two threads, one encodes « S » the other one « O »
    - Without taking any precautions : « ....-.--. »
    - Compliant with atomicity : « ......--- »
    - Compliance with order: « ...---... »

    ➔ Critical section (mutex) solves atomicity but not order problems

- **What to do in front of coherency problems ?**
  - A and B must be atomic to each other
    - Pessimistic synchronization : Prevention (critical section)
      - » Atomicity : we avoid the problem
    - Optimistic synchronization : recovery (timestamps)
      - » We detect the problem (incoherent data ➔ coherent data)
    - Depends on the probability to execute A and B at the same time?
      - » timestamps: risk not to end

- **What to do in front of coherency problems ?**
  - Recovery:
    - Copy the date *(timestamp)*
    - Copy the data
    - Compute new data

    ** Begin atomicity **
    - Copy the current date
      - » Does timestamp has been changed ?
    - If unchanged

      ➔ to modify data and update the date

    ** End atomicity **

    Else do it again

Parallelism management

- **A solution for the problem of mutual exclusion meet these properties:**
  - Not CPU speed dependent (program durations)
  - Two processes (or more) cannot simultaneously enter in critical section
  - When a process is outside its critical section and does not intend to enter in it, it shall not prevent another one to go in critical section
  - Two processes shall not permanently prevent each other to enter to a critical section
    - deadlock situation
  - A process shall always enter in critical section in a duration bounded in time
    - starvation situation

Parallelism management

- **Semaphore:**
  - A semaphore is an object on which only 2 atomic commands are possible
    - **P**(sem) : « sem » semaphore value decreased
      - ***Blocked if the value < 0 (bound)***
    - **V**(sem) : « sem » semaphore value increased
      - ***Allow releasing a process blocked by P (pass)***

Note : come from dutch words *Passeren* (to take) , *Vrygeven* (to release, to give)

Parallelism management

- **Mutual exclusion (mutex), a specific semaphore:**
  - Binary semaphore initialized at 1
  - Its role is to protect a critical section (=> race condition)
  - Allow the access to different shared variables
    - To associate one semaphore of mutual exclusion for each distinct set of shared variables

$mutex1, mutex2 : INIT(TRUE)$

P(mutex1)
…
{critical section n°1}
…
V(mutex1)
…
P(mutex2)
…
{critical section n°2}
…
V(mutex2)

Parallelism management

- **Private semaphore**
  - When each task is authorized to only use one P or V primitive
    - We said it is a private semaphore (particular case)
  - Interpretation
    - The process corresponding to the **P** primitive is waiting for a signal from the process corresponding to the **V** primitive
  - Property
    - If the receiving process is too early, it is blocked
    - If the signal is send to early, it is memorized

Parallelism management

- **Use case of a private semaphore**
  - A process shall be enabled by another one one (event-triggered)
    - Only one process can execute the **P** primitive
    - Other processes can execute **V** operation

Enabling process 1	Enabling process 2	Process to enable
. . .	. . .	. . .
V(sp)	**V**(sp)	**P**(sp)
. . .	. . .	. . .

Parallelism management

- **Problem with semaphores**
  - Two tasks A and B, two semaphores S1 and S2 with M(S1) = M(S2) = 1
  - The sequence is the following
    - A : P(S1)
    - B : P(S2)
    - A : P(S2) /* A is blocked in P */
    - B : P(S1) /* B is blocked in P */
  - Remark
    - It is a general problem
      - » A is blocked and it is B who can change this situation
      - » B is blocked and it is A who can change this situation
        - ➔ The situation cannot evolve
    - Deadlock situation is also possible with only one semaphore (interrupt handler =>TP n°2)

Parallelism management

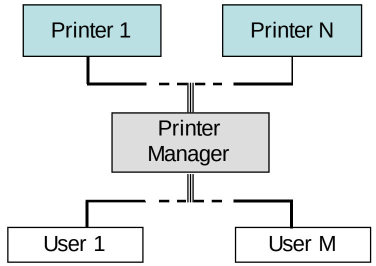
- **Solutions**
  - Recovery
    - To cancel one call to P
    - can only be achieved if we can go back in task execution
    - can only be achieved if we can restore data of the task
    - To cancel all operations the task has done
    - In practical: task detection, and removal of concerned ones
    - Same problem as with timestamps…
  - Prevention
    - Complex problem but in particular cases, there are simple solutions
    - Expression of requirements
      - » There is only one task request for the use of all resources needed

Parallelism management

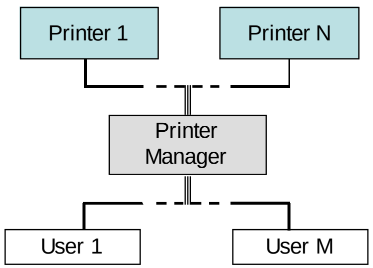
- **Partially ordered resources**
  - The task can do successive requests which target comparable resources
  - Successive requests imply an **order**
  - Demonstration
    - The task cannot be blocked when using the resource which has the highest rank
    - The condition is not necessary
      - » A : P(m1) , P(m2) , P(m3)
      - » B : P(m1) , P(m3) , P(m2)

- **Common semaphores:**
  - Used as resource counter
    - Not limited to 0 or 1 contrary to semaphore of mutual exclusion
  - Semaphore values
    - Initial : corresponds to the maximum capacity
    - Current : number of current capacity
  - P primitive allows requesting (taking) a resource
    - Blocked if no resource is available
  - V primitive release a resource
    - To notify the resource availability and eventually to release a waiting process

- **Example of a printer pool:**
  - Initial value of the semaphore (?)
  - What are the M user processes (?)
  - What the manager is supposed to do (?)



- **Example of a printer pool:**
  - Initial value of the semaphore ➔ N (number of resources)
  - The M user processes request P( )
  - The manager do V( ) to release a resource

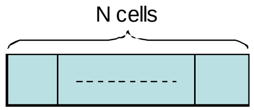


- **Producer / Consumer:**
  - The system exhibits N places to store data
    - ***Producer*** processes provide data to these places
    - ***Consumer*** processes use data and release the corresponding place
  - A semaphore is necessary to synchronize both type of processes
    - To stop a producer if there is no place
    - To stop a consumer if there is no data available

Parallelism management

- **Example with a Read / Write buffer :**
  - E Process writes data in the buffer
  - L Process reads data in the buffer
  - Initial value(s) of the semaphore(s) ?

E Process
```
…
Write
…
```

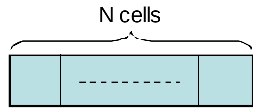


L Process
```
…
Read
…
```


Parallelism management

- **Example with a Read / Write buffer :**
  - E Process writes data in the buffer
  - L Process reads data in the buffer
  - Initial values of semaphores ➔ Free=N, Occ=0

E Process
```
P(Free)
Write
V(Occ)
```



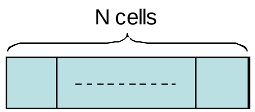
L Process
```
P(Occ)
Read
V(Free)
```


Parallelism management

- **Example with a Read / Write buffer :**
  - Use case with several producers and one consumer
    - What type of problem happens?

E1 Process
```
P(Free)
…
Write
…
V(Occ)
```

E2 Process
```
P(Free)
…
Write
…
V(Occ)
```



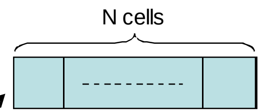
L Process
```
P(Occ)
…
Read
…
V(Free)
```


Parallelism management

- **Example with a Read / Write buffer :**
  - Use case with several producers and one consumer
    - ➔ Mutual exclusion problem

E1 Process
```
P(Free)
P(mutex)
Write
V(mutex)
V(Occ)
```

E2 Process
```
P(Free)
P(mutex)
Write
V(mutex)
V(Occ)
```

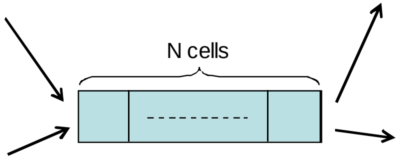


L Process
```
P(Occ)
Read
V(Free)
```

- **Example with a Read / Write buffer :**
  - Use case with several producers and consumers
    - What type of problem happens?

E1 Process
```
P(Free)
...
Write
...
V(Free)
```

E2 Process
```
P(Free)
...
Write
...
V(Occ)
```



L1 Process
```
P(Occ)
...
Read
...
V(Free)
```

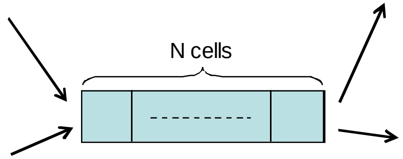
L2 Process
```
P(Occ)
...
Read
...
V(Free)
```


Parallelism management

- **Example with a Read / Write buffer :**
  - Use case with several producers and consumers
    - ➔ Mutual exclusion problems

E1 Process
```
P(Free)
P(mutexW)
Write
V(mutexW)
V(Occ)
```

E2 Process
```
P(Free)
P(mutexW)
Write
V(mutexW)
V(Occ)
```



L1 Process
```
P(Occ)
P(mutexR)
Read
V(mutexR)
V(Free)
```

L2 Process
```
P(Occ)
P(mutexR)
Read
V(mutexR)
V(Free)
```