Schedule of the course(s)

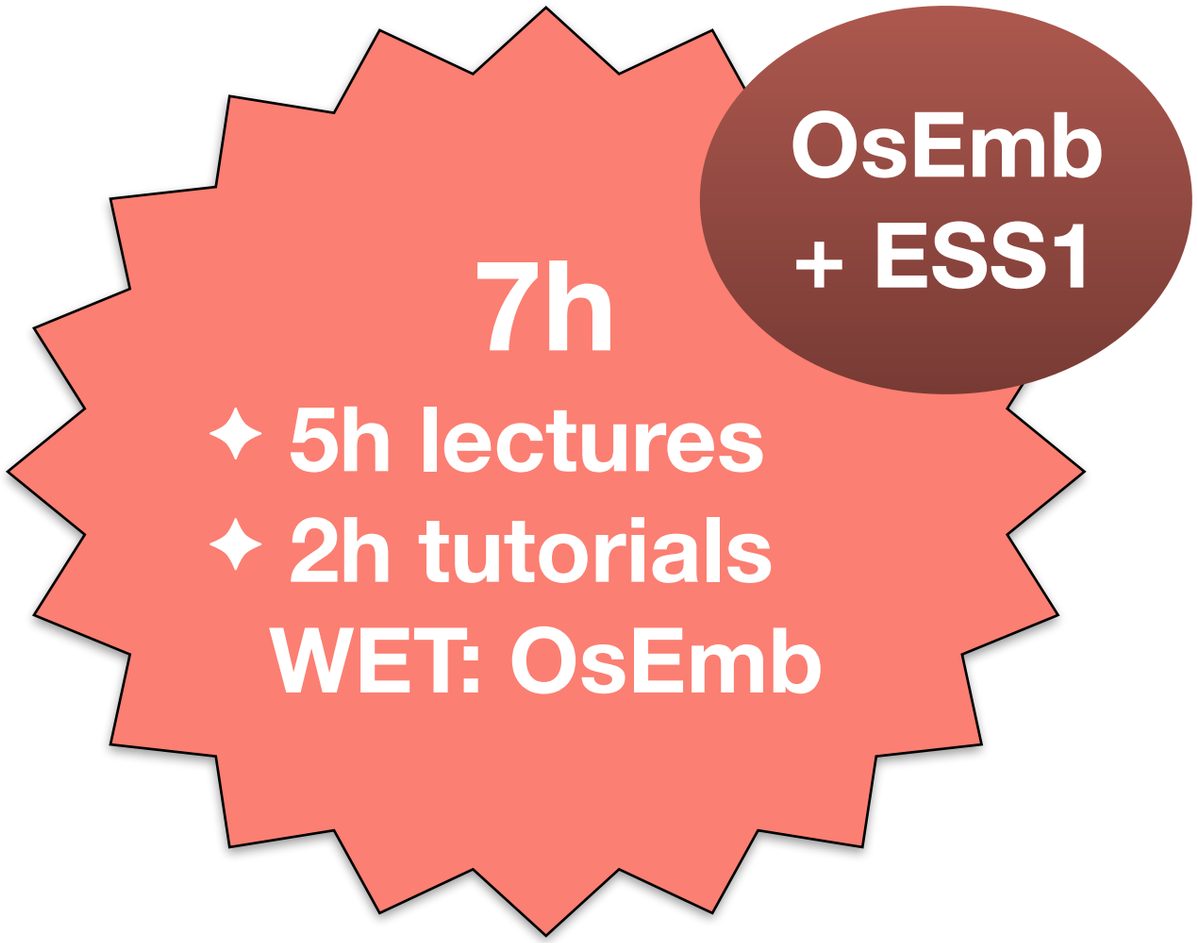
- 2 Masters in parallel:
  - Master WET (Polytech Nantes), OSEmb (7h) and LogEmb (15h).
  - Master CORO (Ecole Centrale de Nantes), ESS (30h)
- Courses:
  - Embedded/Real-Time Operating Systems (CORO/ESS — WET/OsEmb)
  - Embedded Software (CORO/ESS — WET/LogEmb)
  - Additional courses, tutorials and labs for CORO/ESS

`Documents : http://wet.rts-software.org/`

Part 1

Teacher **Jean-Luc Béchennec**

- Multiplexing task on a platform
  - specificities of embedded (RT)OS
  - interrupts
  - sequencer
  - operating system
- Scheduling
  - RM and EDF
- Synchronization
  - Hardware and data sharing: Mutex, semaphores, PCP
- Communication
  - Messaging systems vs shared variables



Part 2

Teachers **Jean-Luc Béchennec**
Sébastien Faucou
Mikaël Briday

- The C language is not your friend:
  - pitfalls
  - programming rules
- Good practices
  - Development process
  - Requirements / Test
- Application design
- Models and model checking
  - Verifying design
- C / Assembly / ABI
- Syscall / µP modes / context switch

15h
✦ **8h lectures**
✦ **3h tutorials**
✦ **4h labs**

LogEmb + ESS2

+

8h
✦ **2h tutorials**
✦ **6h labs**

ESS2

Embedded Operating Systems / ESS1

Master CORO / WET

Jean-Luc Béchennec - jean-luc.bechennec@ls2n.fr

Generalities

Computer systems

- Hardware
  - Processor / Microprocessor / Microcontroller
  - Devices to interface with the real world
  - Data acquisition (sensors connected to input devices)
  - Action (actuators connected to output devices)
- Software
  - One or many applications
  - « Runs » on the hardware
    - Get data
    - Perform computation
    - Set action

Embedded systems

- An embedded system is:
  - A computing system doing a **dedicated function** or a set of dedicated function **within an electrical/mechanical system to** control (a part of) the electrical/mechanical system.
    - Low power consumption
    - Low cost (usually)
    - Size (not always)
    - Often **real-time**
- Can be discussed but:
  - does not encompass mobile devices (smartphones, tablets) which are very close to general purpose handheld computers.

Real-time systems

- A real-time system is a computing system that must produce a response within a specified (and known) time.
  - Implies a deadline.
  - Response after the deadline is a failure.
- Usually the input is an event or a data in the physical world.
  - Output is related to that event or data.
  - Output has to be produced quick enough to be compatible with the controlled process.
- Correctness of the system depends on :
  - The functional correctness (as in any system)
  - The **timeliness** : time critical systems
- Periodic systems

Desired properties

- **Predictability** (adapted from Stankovic)
  - A system is predictable if it is possible to show/prove/demonstrate that requirements are met subject to any assumption made, for example, concerning failures and workloads
  - A RTS must be **predictable** both in term of logical results and **timing behaviour**
- **Dependability** (adapted from Avizienis *et al.*)
  - A system is dependable if it has the ability to avoid service failures that are more frequent and more severe than is acceptable.

There are different classes of RTS

- **Hard real-time**
  - **No timing failure allowed —** a missing reaction could have catastrophic consequences on the environment (human life losses, environmental pollution, ...)
- **Soft real-time**
  - Occasional failure allowed — the environment is inherently robust to occasional late reactions.
- **Firm real-time**
  - Same as soft real-time but a late result has no value

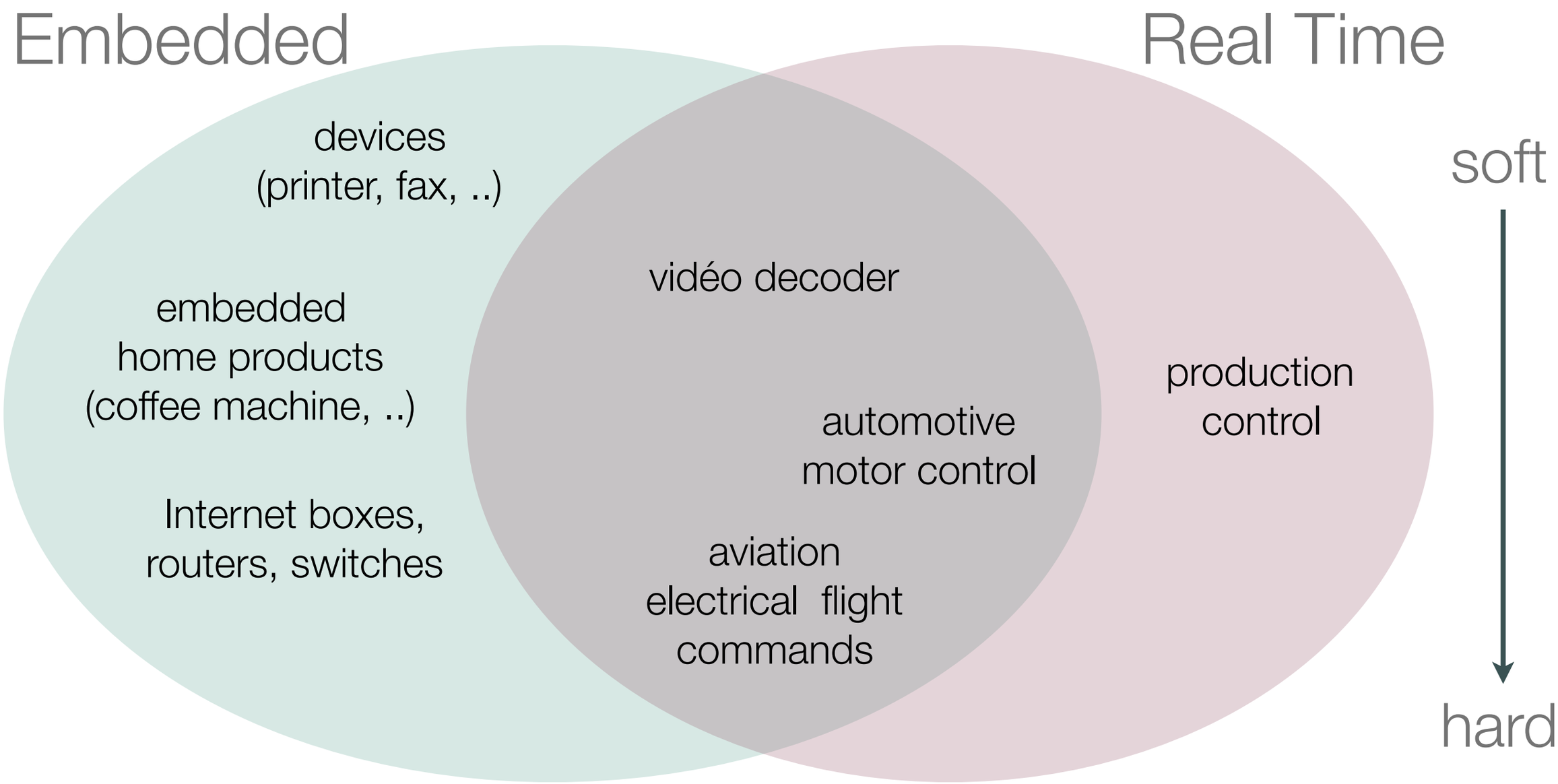
A single system can include hard real-time, firm real-time and soft real-time subsystems.

Summary

Real-time is not about being fast, it is about being on time. Be the tortoise, not the hare.

Real-time system design requires specific model, method, tools, and components. Blue screen of death is not an option when commanding a plane or a car.

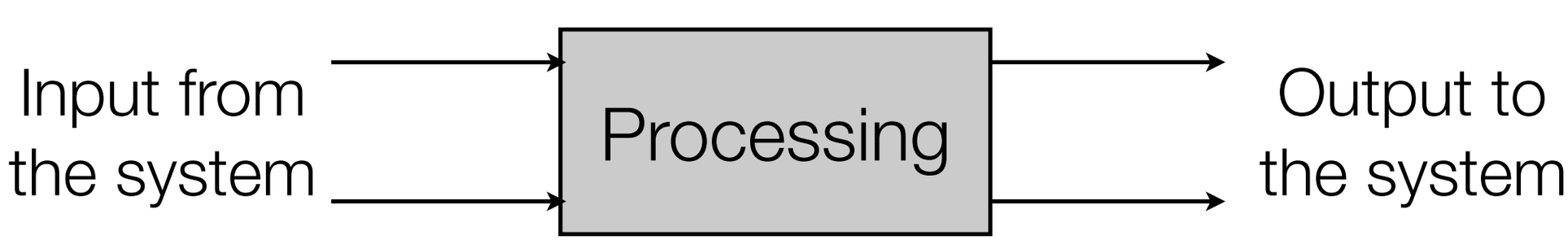
Embedded systems *vs* Real Time systems



source:
I. Puaut

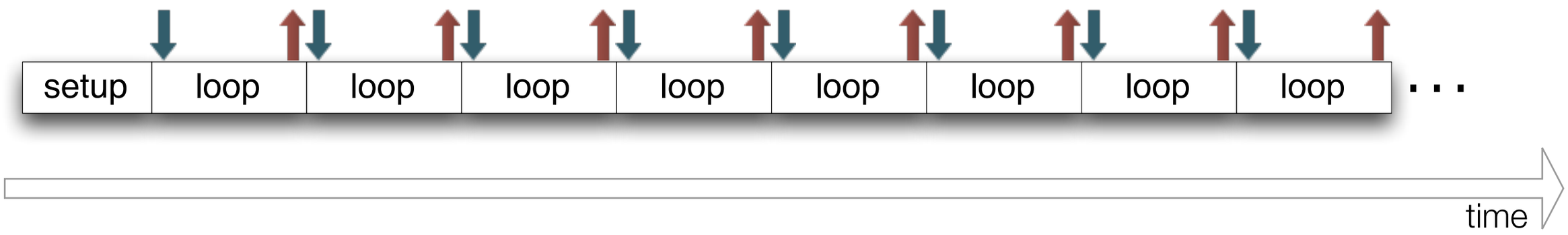
Embedded / Real-time systems

- Simple system :
  - One task only



Example

```
void setup()
{
}

void loop()
{
  /* get the potentiometer value */
  uint16_t value = analogRead(A0);

  /* dim the LED */
  analogWrite(value/4);
}
```



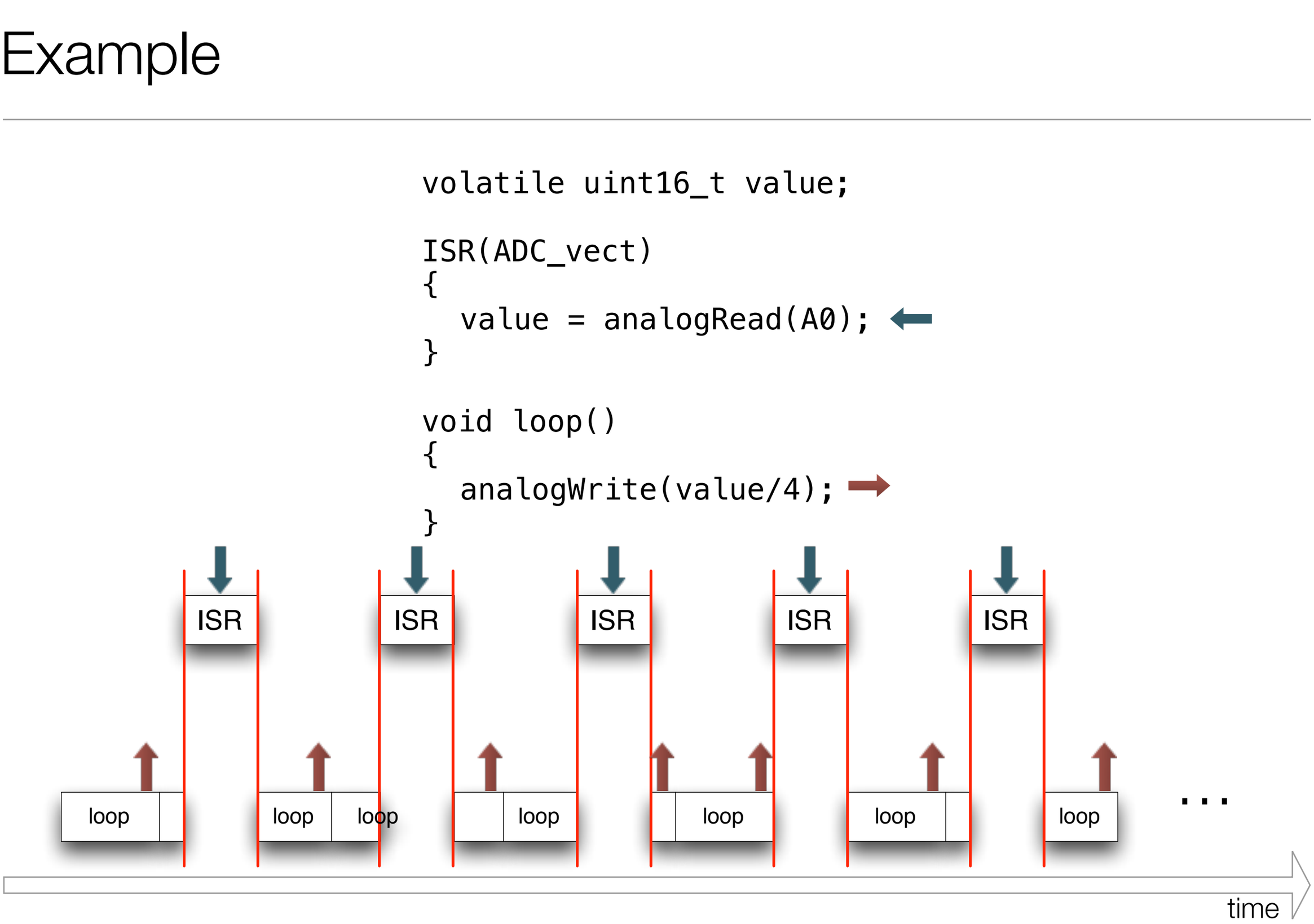
Embedded / Real-time systems

- Simple system :
  - Asynchronous

a bit of concurrency

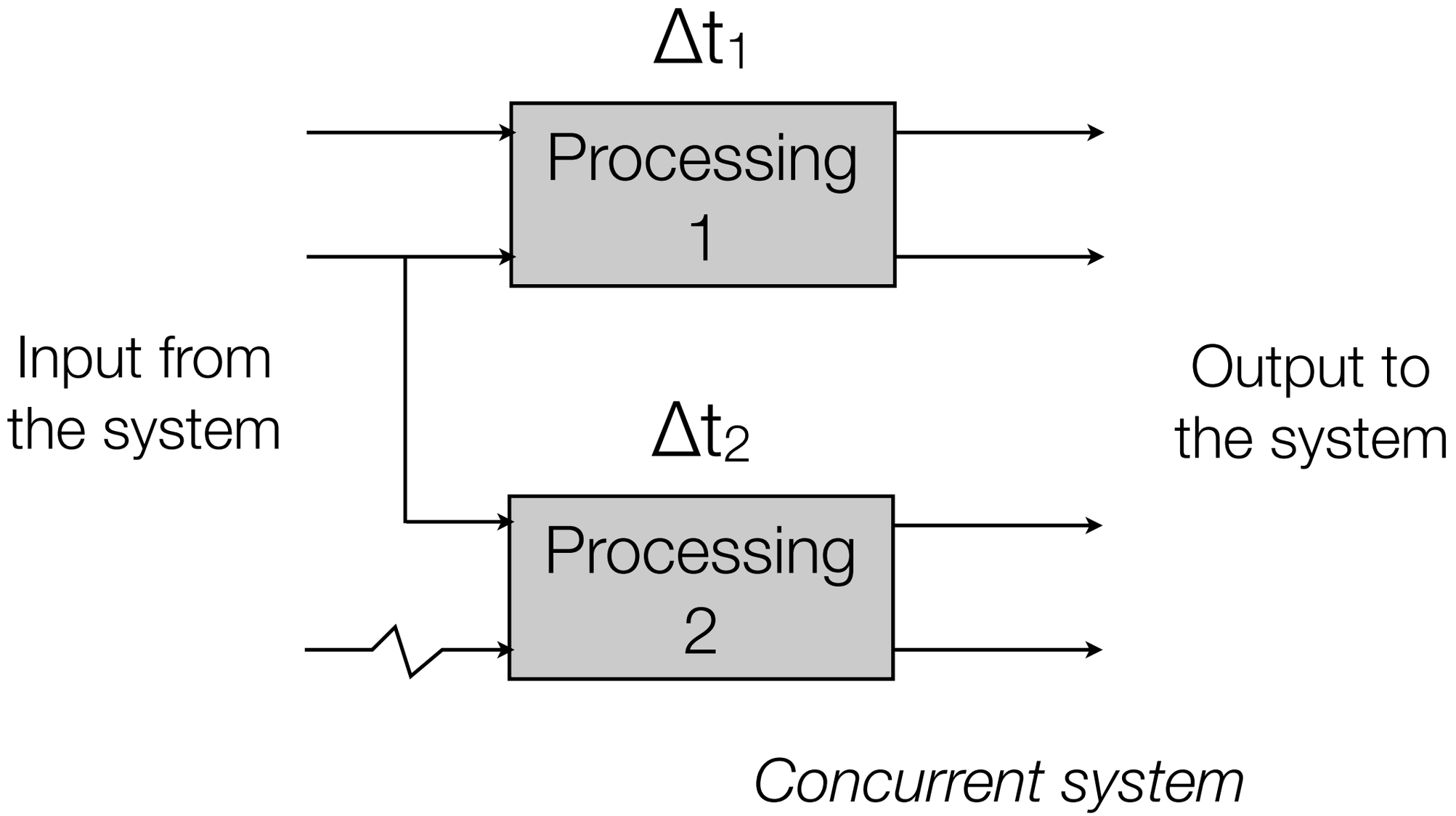
Example

```
volatile uint16_t value;

ISR(ADC_vect)
{
  value = analogRead(A0);
}

void loop()
{
  analogWrite(value/4);
}
```



Embedded / Real-time systems

- More complex :
  - Multitask



Concurrent system

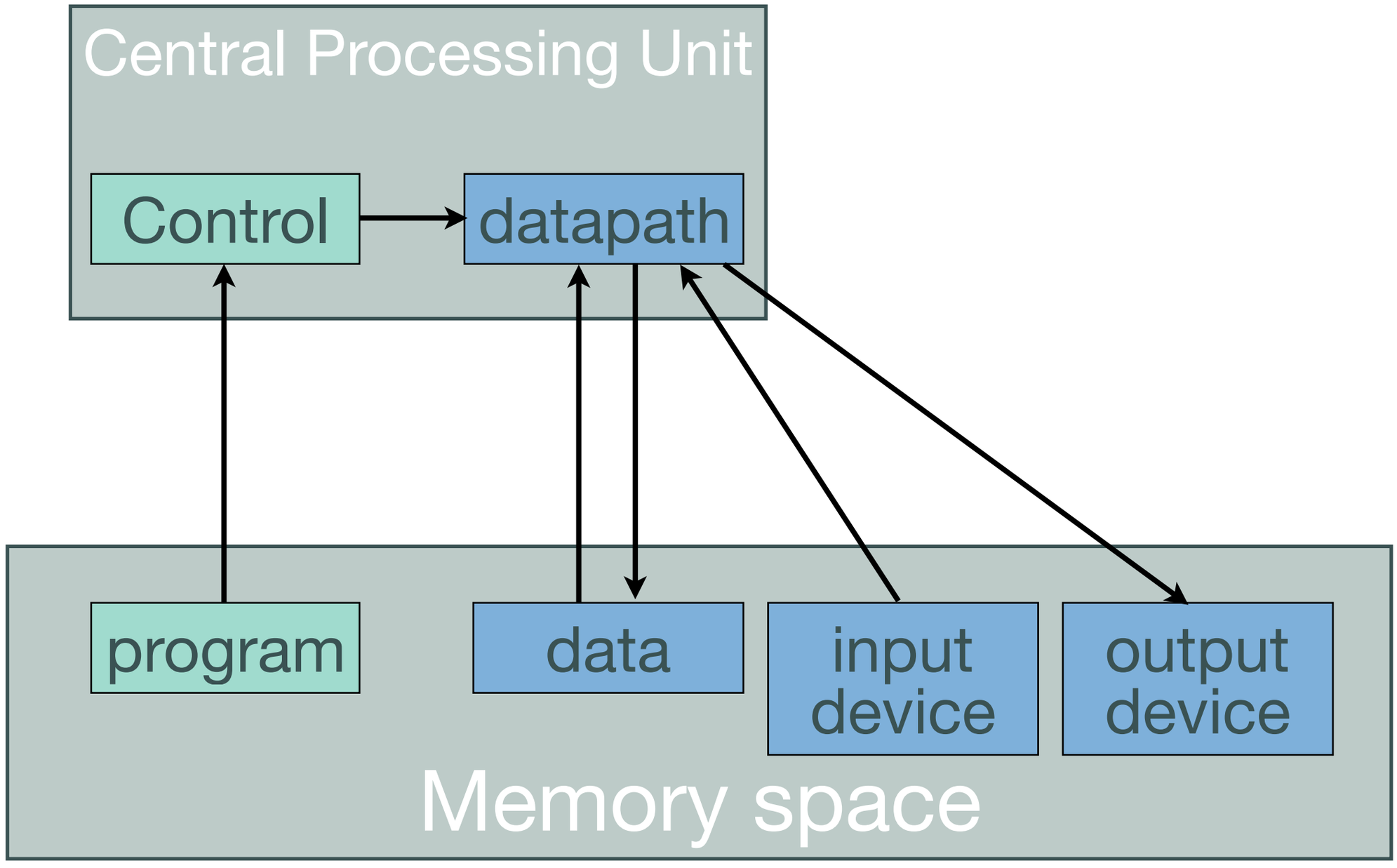
Software architecture

- When complexity of an application grows, it is necessary to **split the computing into separate tasks**
  - maintainability,
  - readability,
  - separation of concerns
  - programming abstraction

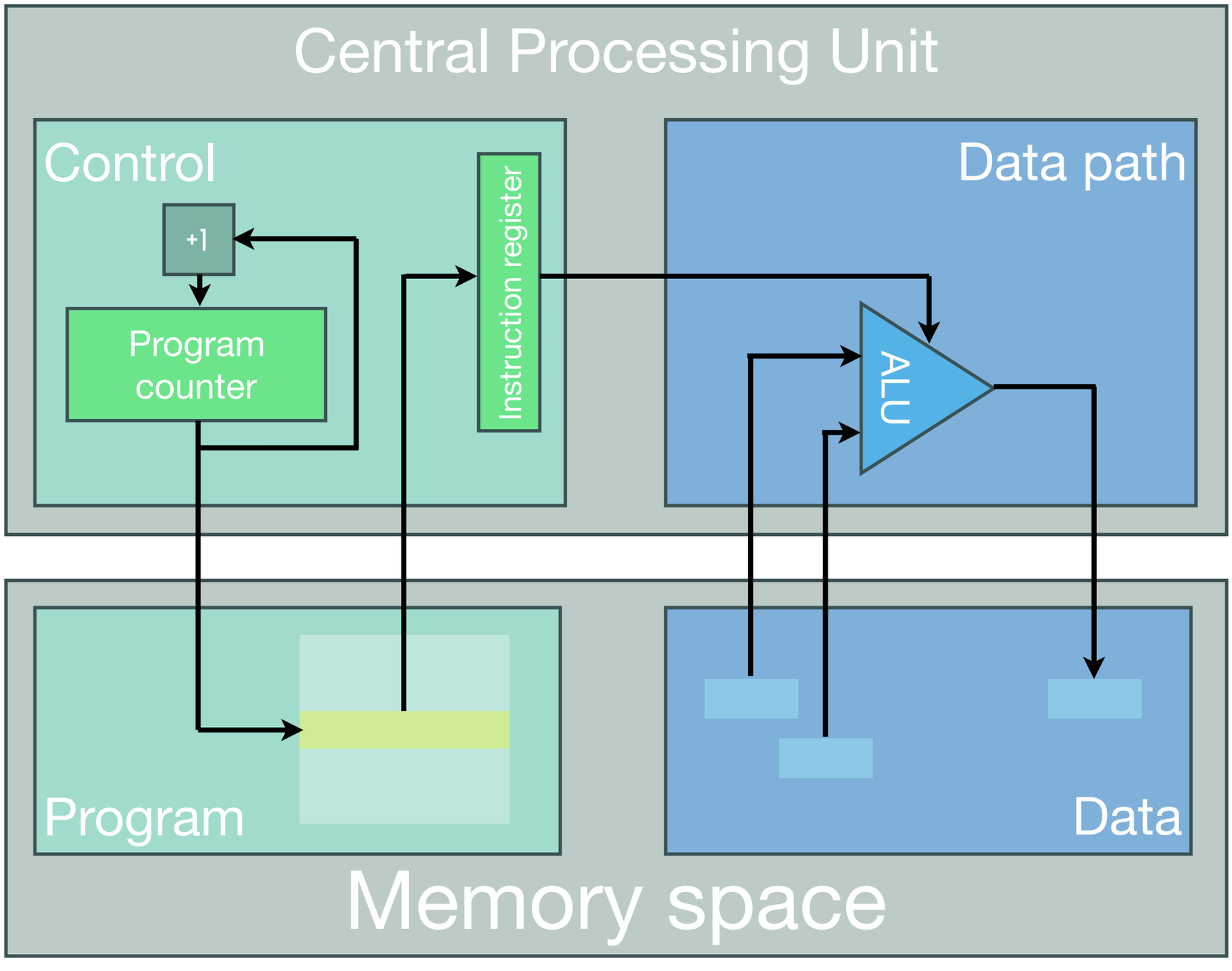
What means concurrent ?

- A computer is a sequential machine
  - A program is sequential (most of the time)
  - A program is a sequence of instructions
  - A computer executes the instructions one after the other
- Processing many tasks on the same computer means tasks are *concurrent* for their execution

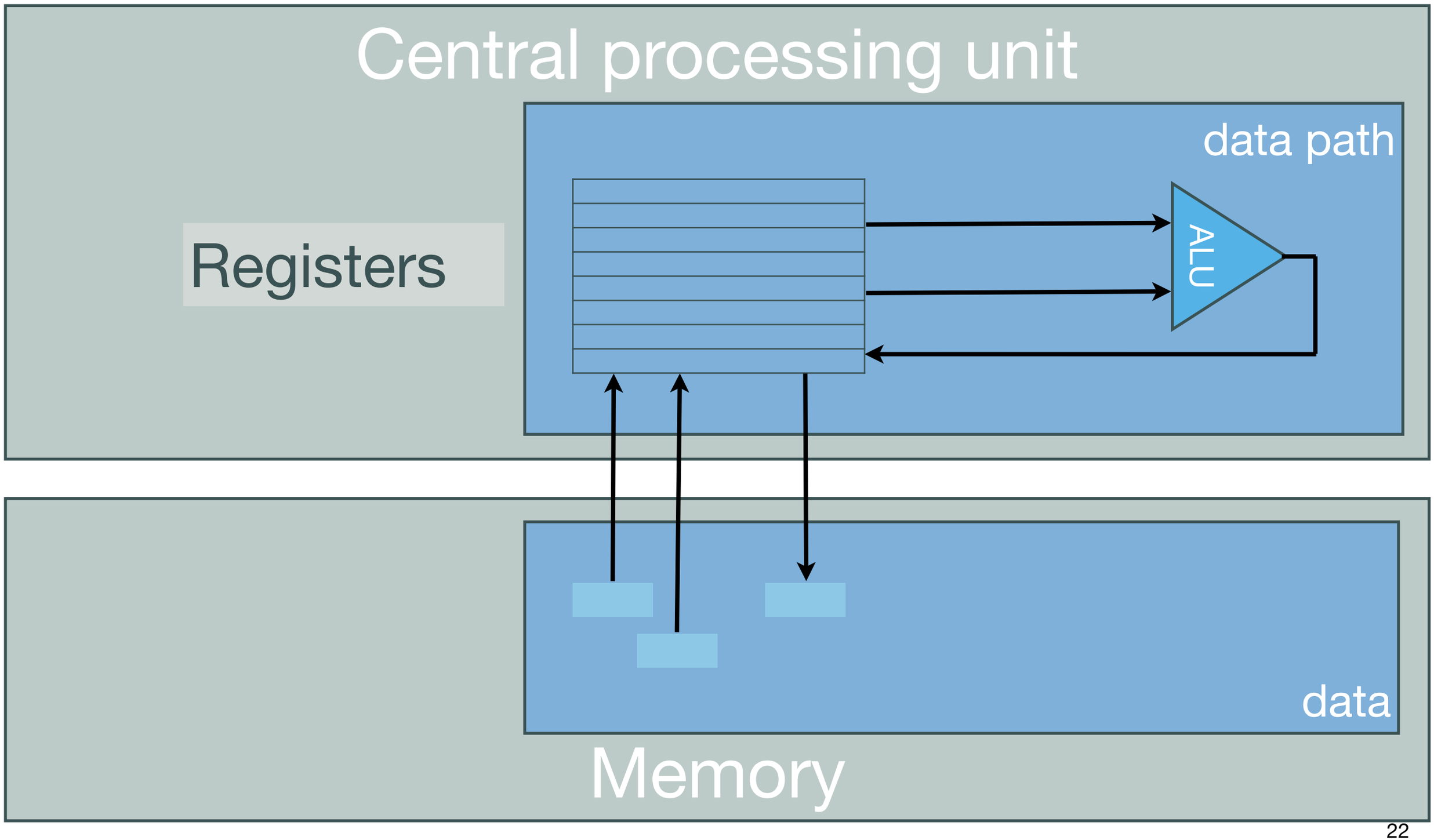
Very simplified drawing



A little bit less simple



Focus on registers



Example of the ARM Cortex (1)

- The CPU uses 16 32 bits general purpose registers: **r0** to **r15**.
- Among them, 3 registers have a special meaning:
  - **r13** is the stack pointer, **sp**
  - **r14** is the link register, **lr**
  - **r15** is the program counter, **pc**
- At last, the PSR stores some flags, among others:
  - C, the **C**arry of an arithmetic/logic operation
  - N, **N**egative output of an arithmetic/logic operation
  - V, o**V**erflow in of an arithmetic/logic operation
  - Z, **Z**ero output of an arithmetic/logic operation

This defines the context of execution

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
sp
lr
pc

psr

A bit of assembly language

```
int b[10];

int main()
{
  int i;

  for (i = 0; i < 10; i++)
  {
    b[i] = i;
  }

  return 0;
}
```

```
00000000 <main>:
   0:   4a04            ldr  r2, [pc, #16]  ; (14 <main+0x14>)
   2:   2300            movs r3, #0
   4:   f842 3023       str.w  r3, [r2, r3, lsl #2]
   8:   3301            adds r3, #1
   a:   2b0a            cmp  r3, #10
   c:   d1fa            bne.n  4 <main+0x4>
   e:   2000            movs r0, #0
  10:   4770            bx lr
  12:   bf00            nop
  14:   00000000        .word  0x00000000
```

r2←b

i←0

b[i]←i

i←i+1

compare i and 10

branch if i < 10

load return value of the function with 0

return from the function

How to manage concurrency ?

- Bare metal approach
  - Program runs directly on the hardware
  - Asynchronous events are managed by interrupt handlers
  - Concurrency between interrupt handlers and normal execution
  - Limited to small and less complex systems.

How to manage concurrency ?

- Sequencer approach
  - The sequencer is a sequential program running in loop
  - The sequencer measures the time and calls functions (tasks) according to their activation period
  - Periodic system, predictable, deterministic
  - Not very flexible, all tasks are mixed, non preemptable.

Example of a sequencer

- Starting with previous analogRead / analogWrite example, we add a LCD.
  - Let’s say we have 3 tasks each in a function:
    - task_sensor that does the analogRead to get the potentiometer value (execution time 1ms)
    - task_led that does the analogWrite to dim the led (execution time 1ms)
    - task_lcd that display the current value on a lcd (execution time 3ms)
- First we decide of the period of each task:
  - task_sensor every 5ms
  - task_led every 3ms
  - task_lcd every 10ms

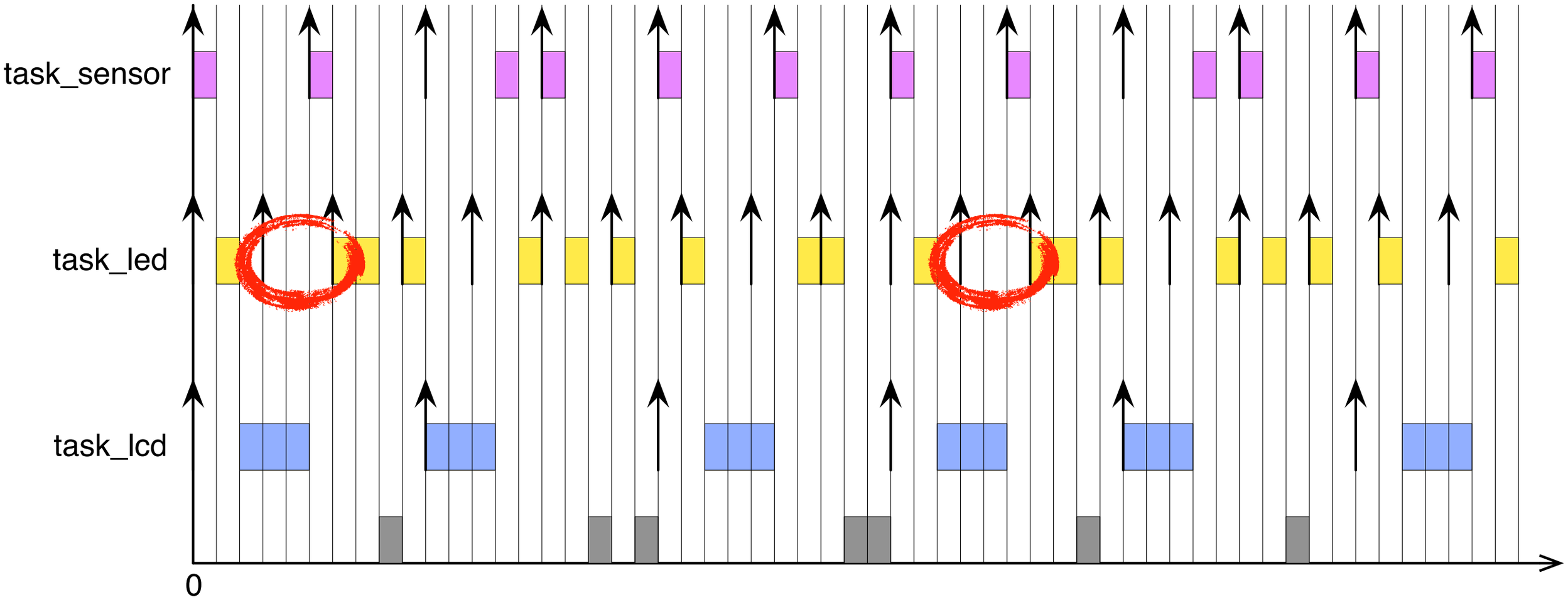
Example of a sequencer

```
uint16_t value;

void task_sensor()
{
  value = analogRead(A0);
}

void task_led()
{
  analogWrite(value/4);
}

void task_lcd()
{
  lcd.write(value);
}
```

```
void loop()
{
  static uint32_t sensorTime = 0;
  static uint32_t ledTime = 0;
  static uint32_t lcdTime = 0;

  uint32_t currentTime = millis();
  if (currentTime >= sensorTime) {
    sensorTime += 5;
    task_sensor();
  }
  currentTime = millis();
  if (currentTime >= ledTime) {
    ledTime += 3;
    task_led();
  }
  currentTime = millis();
  if (currentTime >= lcdTime) {
    lcdTime += 10;
    task_lcd();
  }
}
```

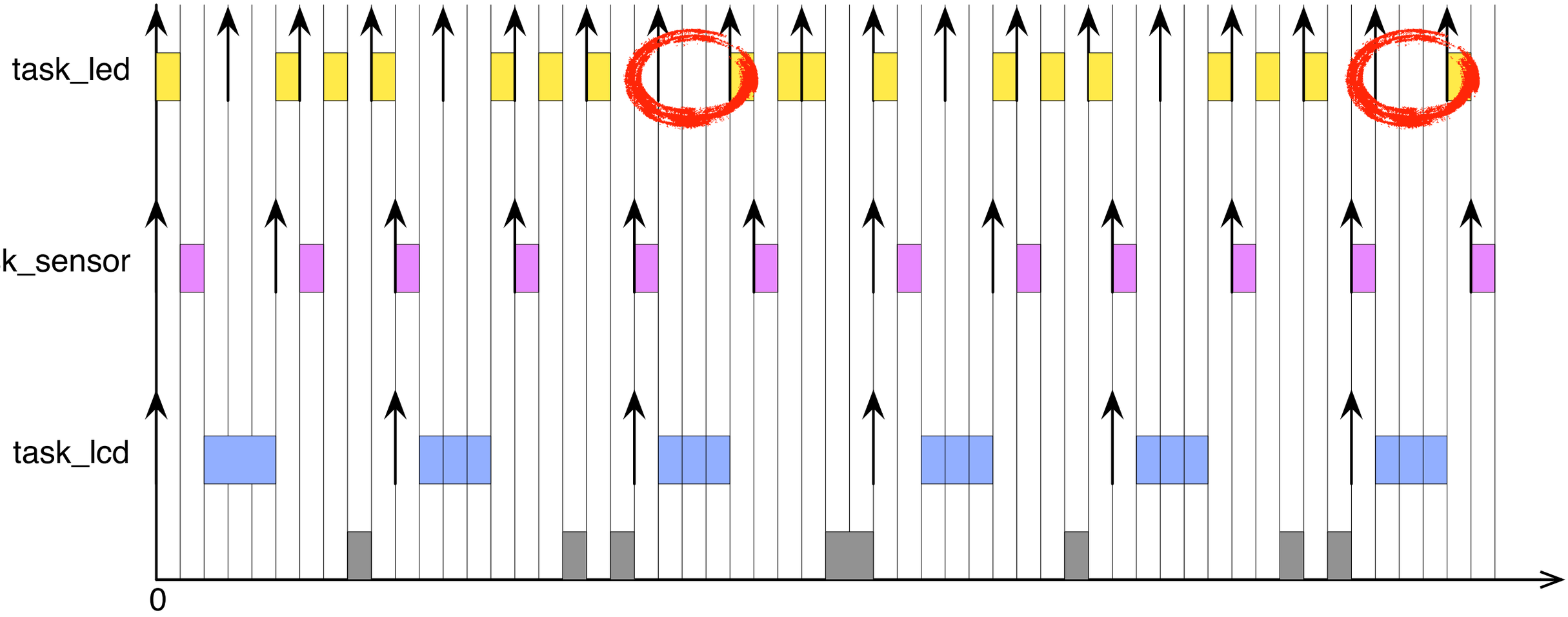
Example of a sequencer

task_sensor > task_led > task_lcd



Example of a sequencer

task_led > task_sensor > task_lcd



What does an operating system ?

- Running many task implies:
  - The computing power is shared among tasks
  - Tasks may need to synchronize and access shared resources in a safe way
  - Tasks may need to communicate

- An operating system:
  - decides what task run according to a scheduling policy
  - provides services to synchronize tasks and manage shared resources
  - provides services to communicate

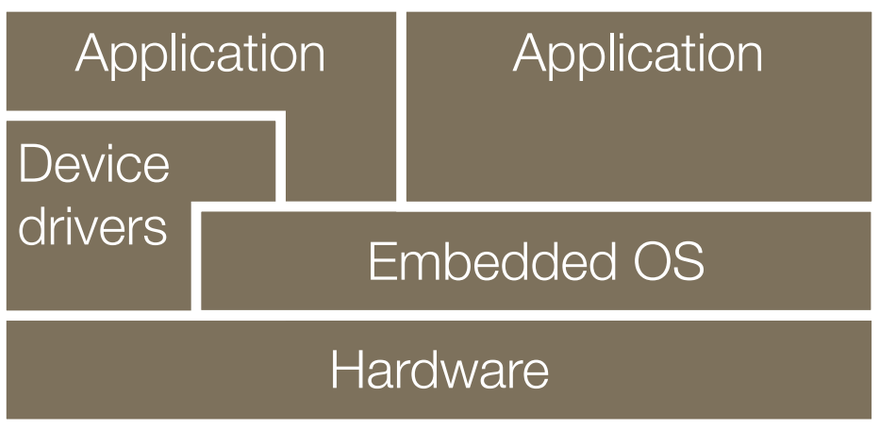
Why a desktop OS is not suited ?

- Too much functionalities not needed in embedded systems
  - eats a lot of memory space
  - needs too much computing power
- Timing variability too high
- Not designed and not tested for safety / mission-critical applications
- Not designed to be modular and configurable.

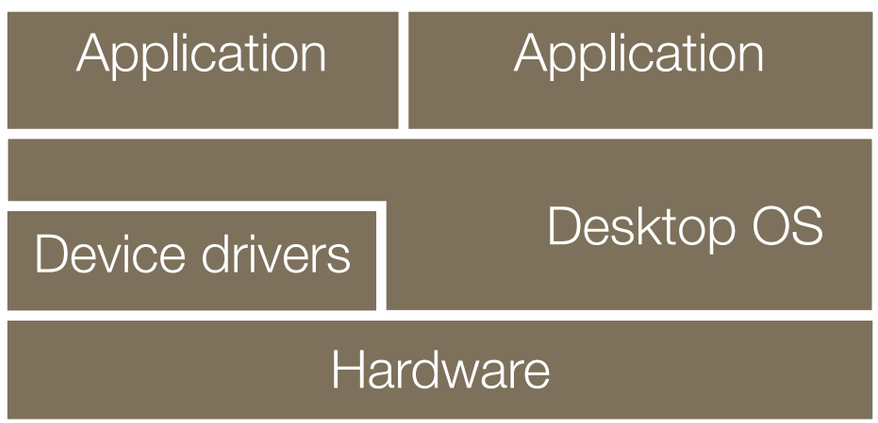
What about device drivers ?

- Device drivers are not part of the OS (exception: system timer)
- They are implemented:
  - Using **tasks scheduled by the scheduler** as any other task of the application ⇒ in real-time context this **improve predictability**.
  - Target support is easier for the OS.

Embedded system



Desktop system



Scheduling

How to manage concurrency ?

- Real-time operating system (RTOS) preemptive
  - Uses real-time *scheduling*
  - Periodic system, predictable, deterministic
  - Preemptive system :
    - A running task mais be stopped to allow a higher priority task to run and resumed later.
- Tasks of the application are separated from the RTOS
- Separation of concerns, reusability, abstraction layer.

Scheduling policies

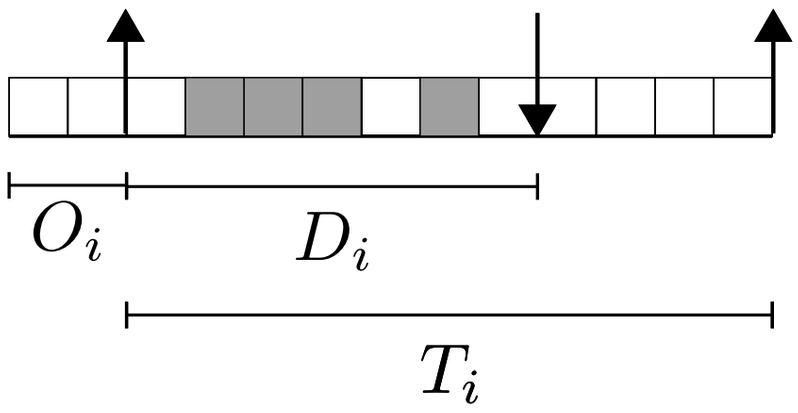
- A RTOS implements a scheduling algorithm.
  - a way to decide which task should run according to a priority
  - remark: the complete execution of a task is called a **job**
  - 2 categories : fixed priority and dynamic priority
    - fixed priority : the priority is assigned offline
    - dynamic priority : the priority is computed dynamically
      - A task may have a dynamic priority and the jobs a fixed one.
- We will see 3 scheduling policies :
  - RM : Rate monotonic (fixed)
  - DM : Deadline monotonic (fixed)
  - EDF : Earliest Deadline First (dynamic for tasks, fixed for jobs)

Definitions

- The **workload** (W) is composed of a set of tasks
- A **task** generates recurrently a potentially infinite set of jobs.
- Each **job** is a computation that must be executed on a processor. In a real-time system, a job has a WCET (Worst Case Execution Time) and a deadline.
  - as it is generally intractable to compute the WCET of a job, a conservative estimation is used
  - in critical systems, WCET estimations should not be considered as safe
- The scheduling algorithm is in charge of allocating jobs to processors so that each job receives enough computation time to finish before its deadline.

Task set, task

- A task set W is defined as a finite set of n independant tasks: $W = \{\tau_1,...,\tau_n\}$
- Each task τ_i is defined as a tuple $\tau_i = \langle O_i,T_i,C_i,D_i \rangle$ where:
  - $O_i \in N$ is the offset of the task;
  - $T_i \in N$ is the period of the task;
  - $C_i \in N$ is the WCET of the task;
  - $D_i \in N$ is the relative deadline of the task.



Jobs

- Each task τ_i generates a potentially infinite set of jobs:
  $J_i = \{\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,j}, \tau_{i,j+1}, \ldots\}$
- The task set W generates: $J_W = \tau_i \in W\ J_i = \{\tau_{1,1}, \tau_{1,2}, \ldots, \tau_{2,1}, \tau_{2,2}, \ldots\}$
- A job $\tau_{i,j}$ is characterized by:
  - $a_{i,j} = O_i + (j - 1)T_i$ its arrival time;
  - $c_{i,j} = C_i$ its computation time;
  - $d_{i,j} = a_{i,j} + D_i$ its absolute deadline;

Rate Monotonic (RM)

- Priorities are assigned according to the periods of the tasks.
- The **lower** is the period, the **higher** is the priority.
- Exemple : 3 tasks
  - t1, period 10ms
  - t2, period 20ms
  - t3, period 7ms
  - Priorities are assigned so that P(t3) > P(t1) > P(t2)

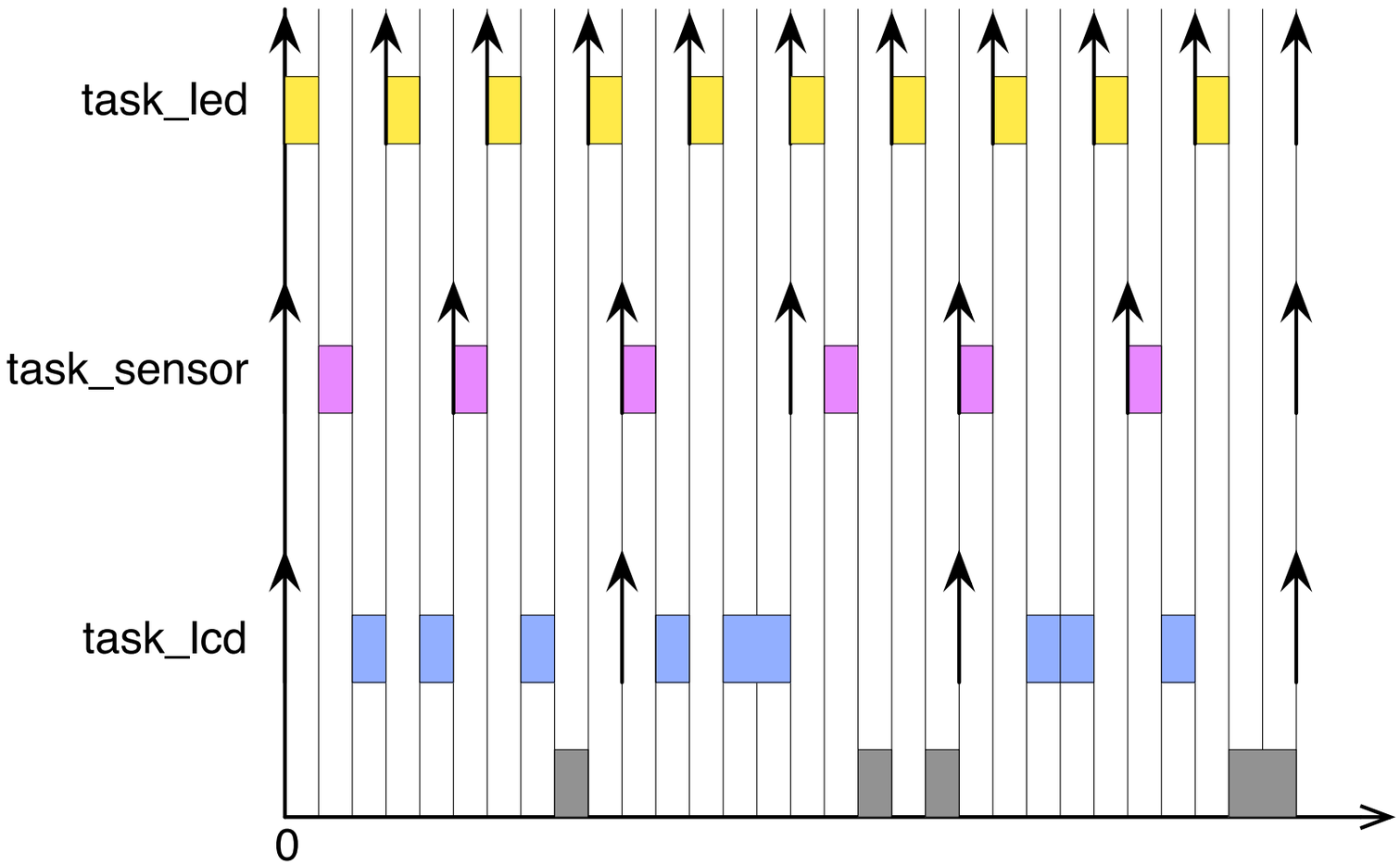
Optimality of RM

- Theorem (Optimality of RM, [LL73]) Preemptive Rate Monotonic is optimal when:
  - all tasks are independent
  - all tasks are synchronous: $\forall \tau_i\ O_i = a_{i,1} = 0$
  - all tasks are periodic: $\forall \tau_{i,j}\ a_{i,j} = (j - 1)T_i$
  - deadlines equal periods $\forall \tau_i\ D_i = T_i$
  - execution time of a task is constant: $\forall \tau_{i,j}\ c_{i,j} = C_i$
  - a job cannot voluntarily suspend itself
  - context switch time and other OS overheads are zero
  - inter-task precedence constraints are enforced
  - there is a single processing unit
  - scheduling algorithm is on-line, fixed-task priority, preemptive, work conserving

$$U = \sum_{\tau_i \in W} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

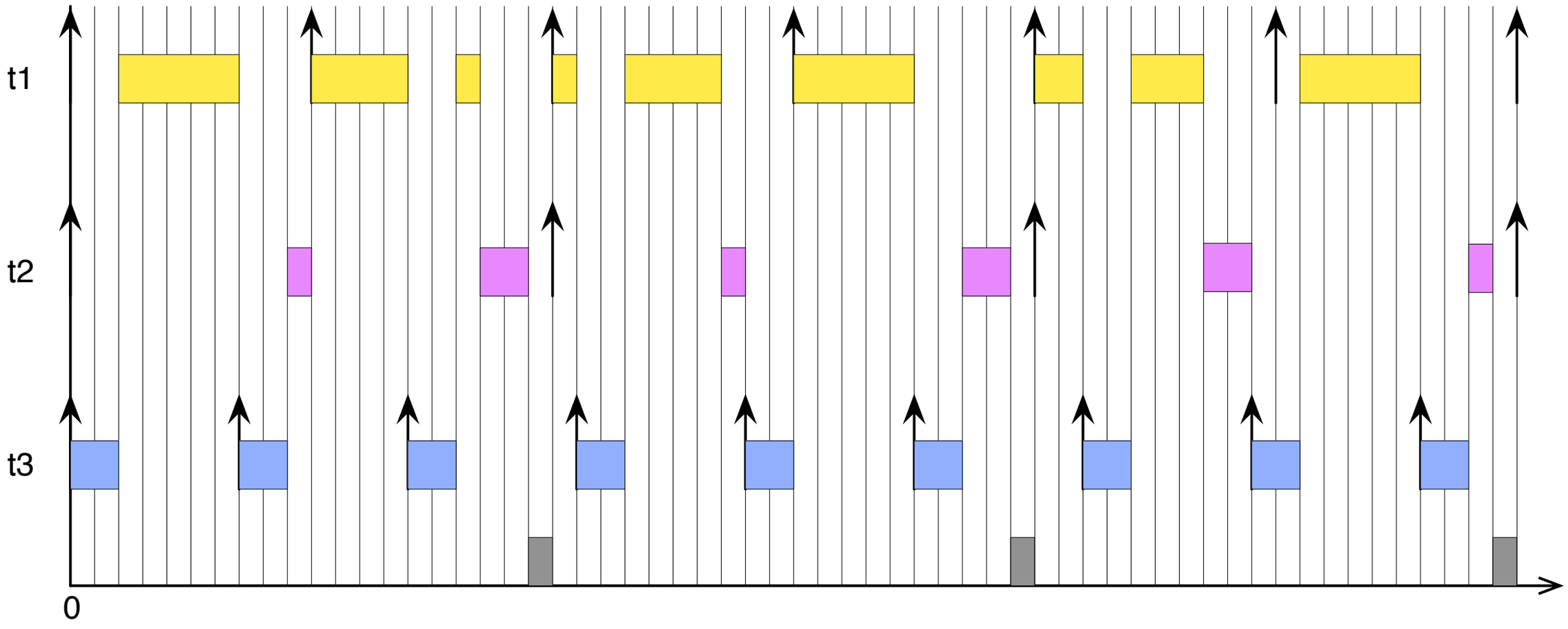
Example of rate monotonic

- From the previous example:
  - task_led: period 3ms, execution time 1ms
  - task_sensor: period 5ms, execution time 1ms
  - task_lcd: period 10ms, execution time 3ms



Another example

- t1, period 10ms, execution time 5ms
- t2, period 20ms, execution time 3ms
- t3, period 7ms, execution time 2ms



Deadline Monotonic (DM)

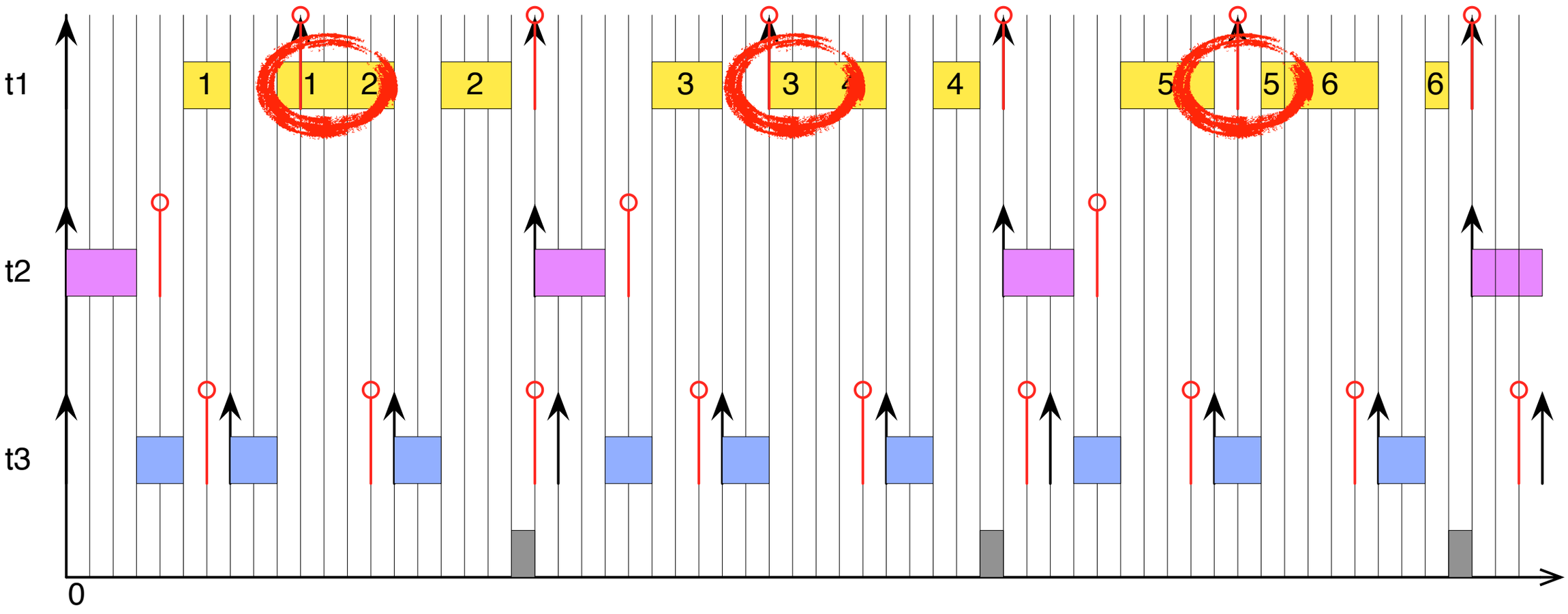
- Priorities are assigned according to the deadline of the tasks.
- The lower is the deadline, the higher is the priority.
- Exemple : 3 tasks
  - t1, period 10ms, execution time 5ms, deadline 10ms
  - t2, period 20ms, execution time 3ms, deadline 4ms
  - t3, period 7ms, execution time 2ms, deadline 6ms
- Priorities are assigned so that P(t2) > P(t3) > P(t1)

$$\forall \tau_i \in W \quad C_i + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{D_j}{T_i} \right\rceil C_j \leq D_i$$

- where hp(τ_i) is the subset of W containing all tasks with priority higher than τi:

$$hp(\tau_i) = \{\tau_j \mid \tau_j \in W \wedge D_j < D_i \}$$

Deadline monotonic (DM)

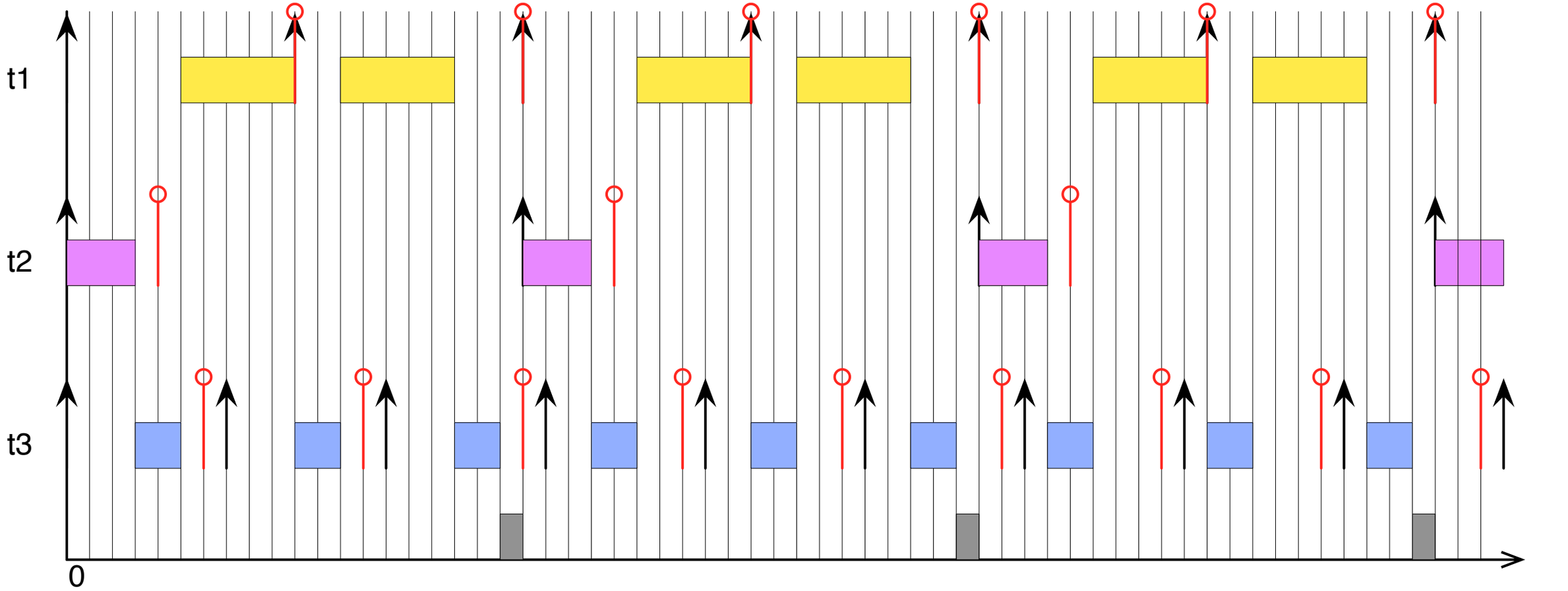


Earliest Deadline First (EDF)

- Priorities are assigned according to the deadline of the tasks at the time a rescheduling occurs.
- The lower is the deadline, the higher is the priority.
- Optimal algorithm.

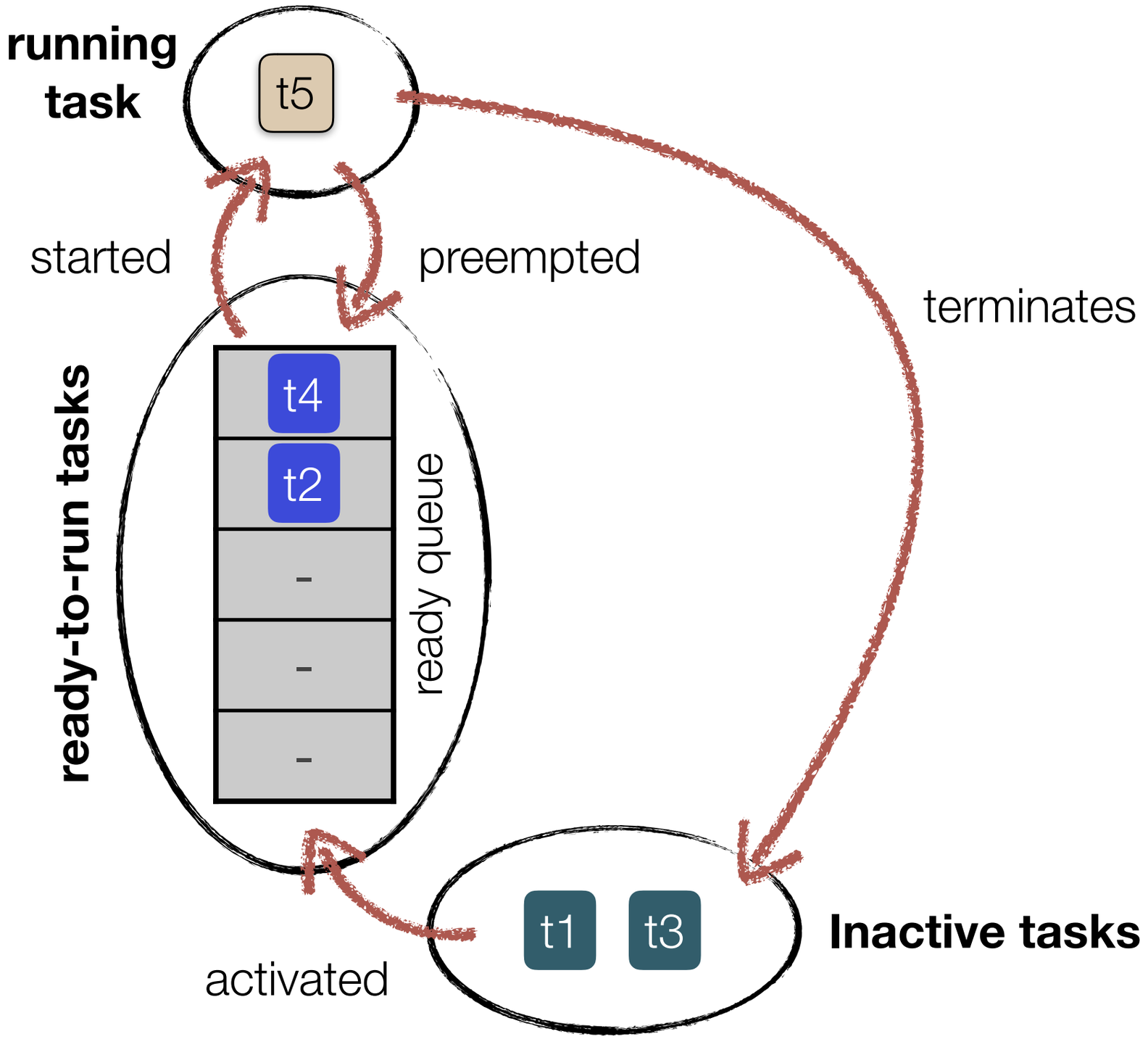
Earliest Deadline First (EDF)

- Exemple : 3 tasks
  - t1, period 10ms, execution time 5ms, deadline 10ms
  - t2, period 20ms, execution time 3ms, deadline 4ms
  - t3, period 7ms, execution time 2ms, deadline 6ms



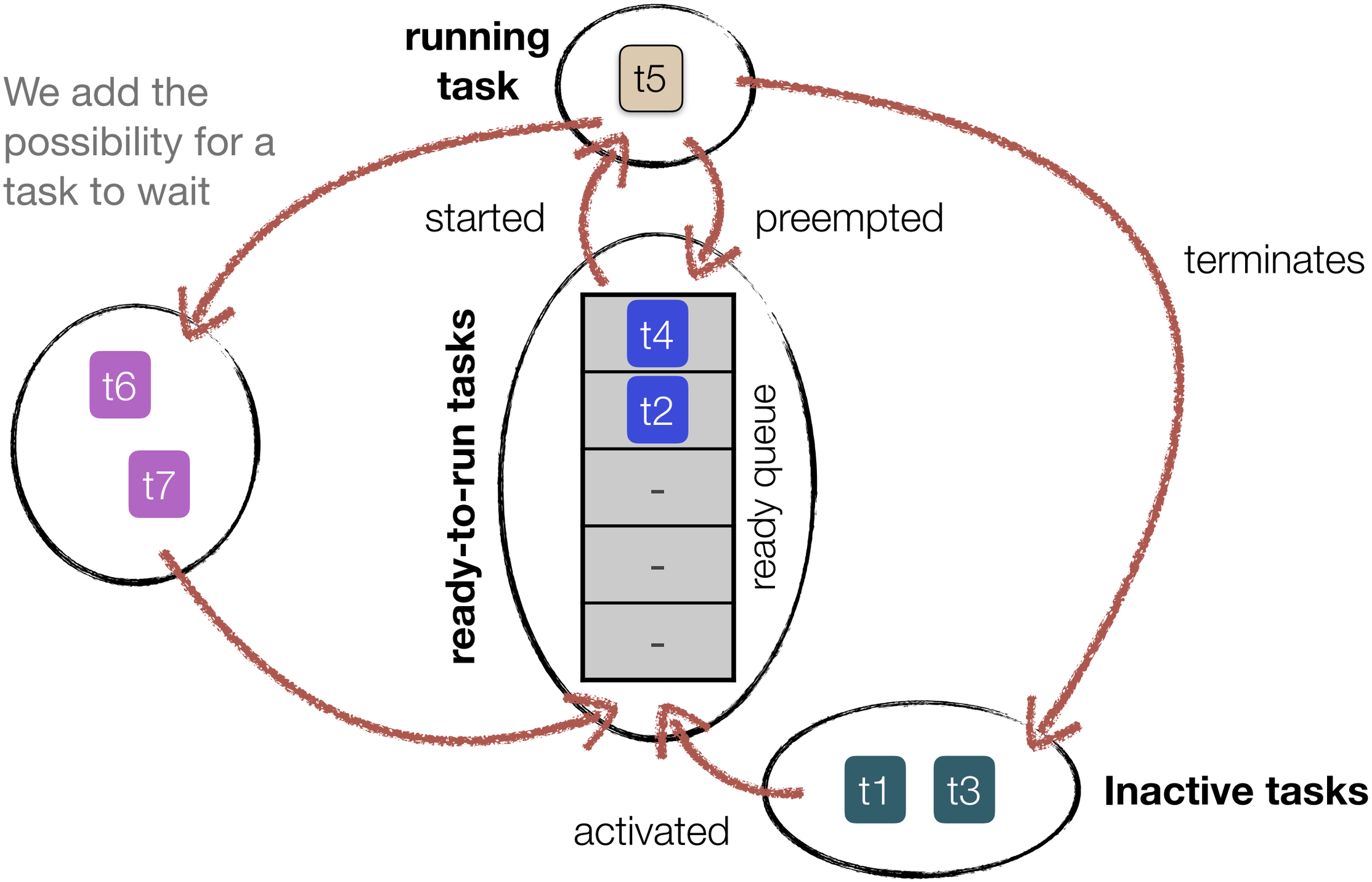
Managing tasks

- The scheduler manages a ready queue: a data structure where ready tasks are stored

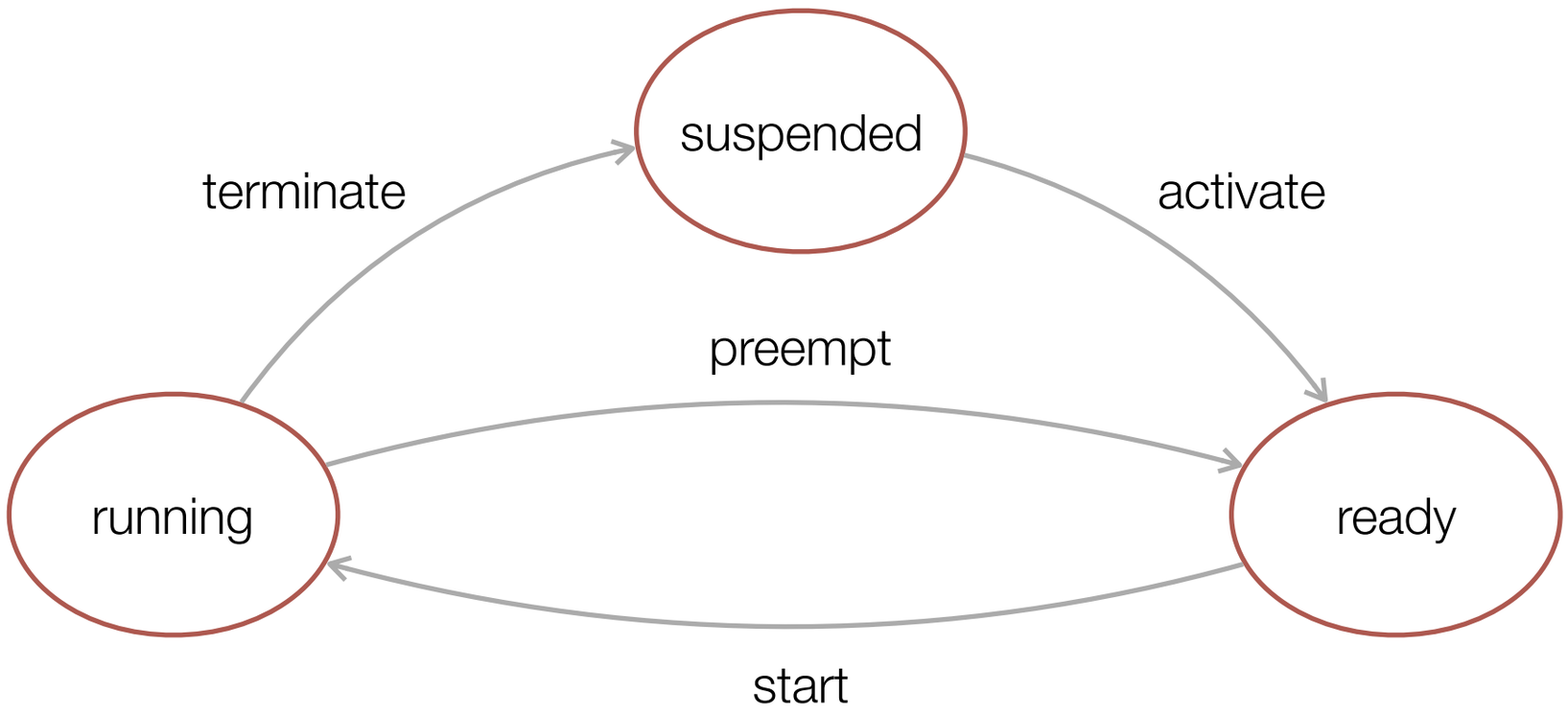


Managing tasks

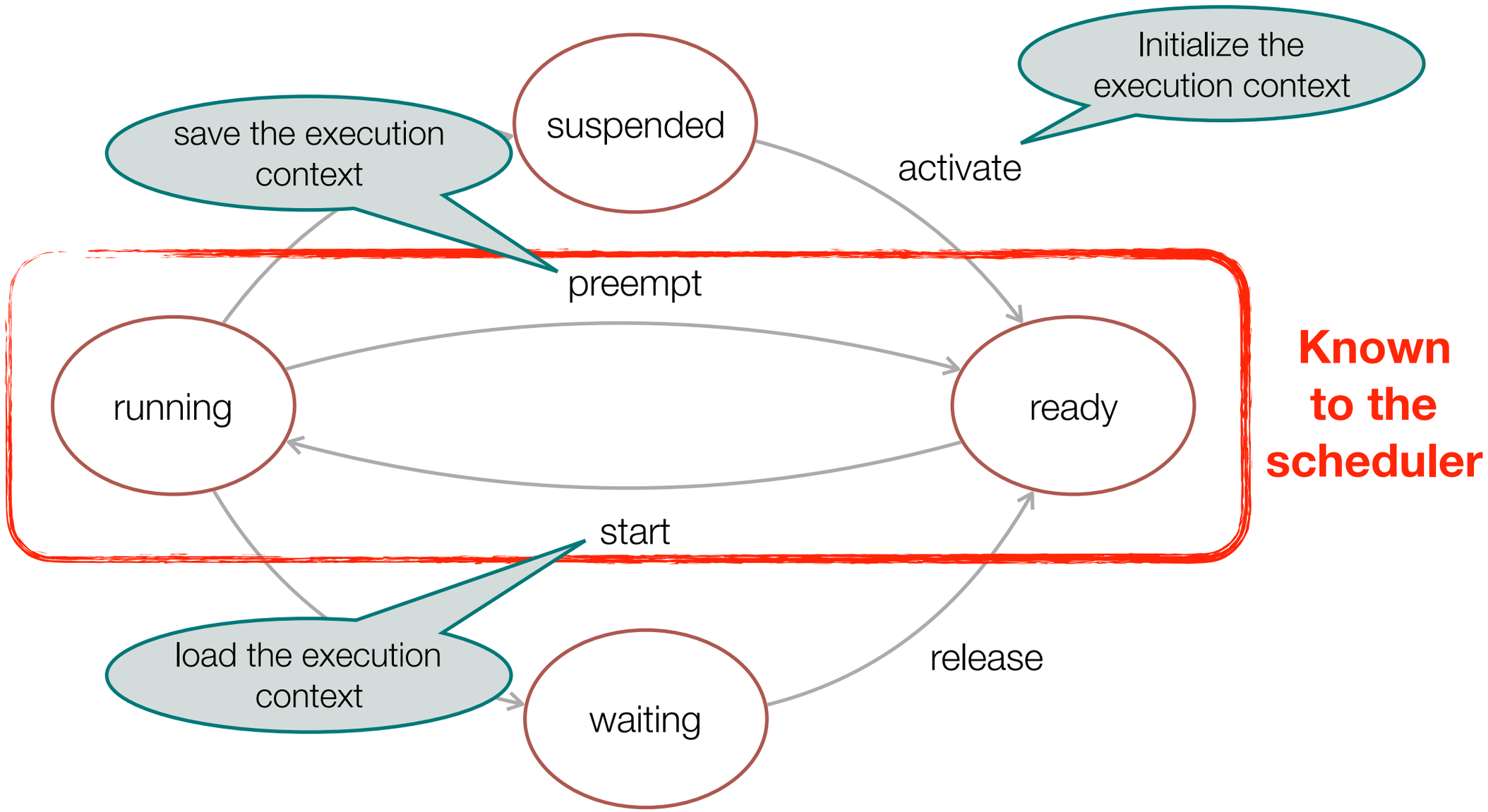
- We add the possibility for a task to wait



Task states


suspended
terminate
activate
preempt
running
ready
start

Task states


Initialize the execution context
save the execution context
suspended
activate
preempt
running
ready
start
load the execution context
waiting
release
Known to the scheduler

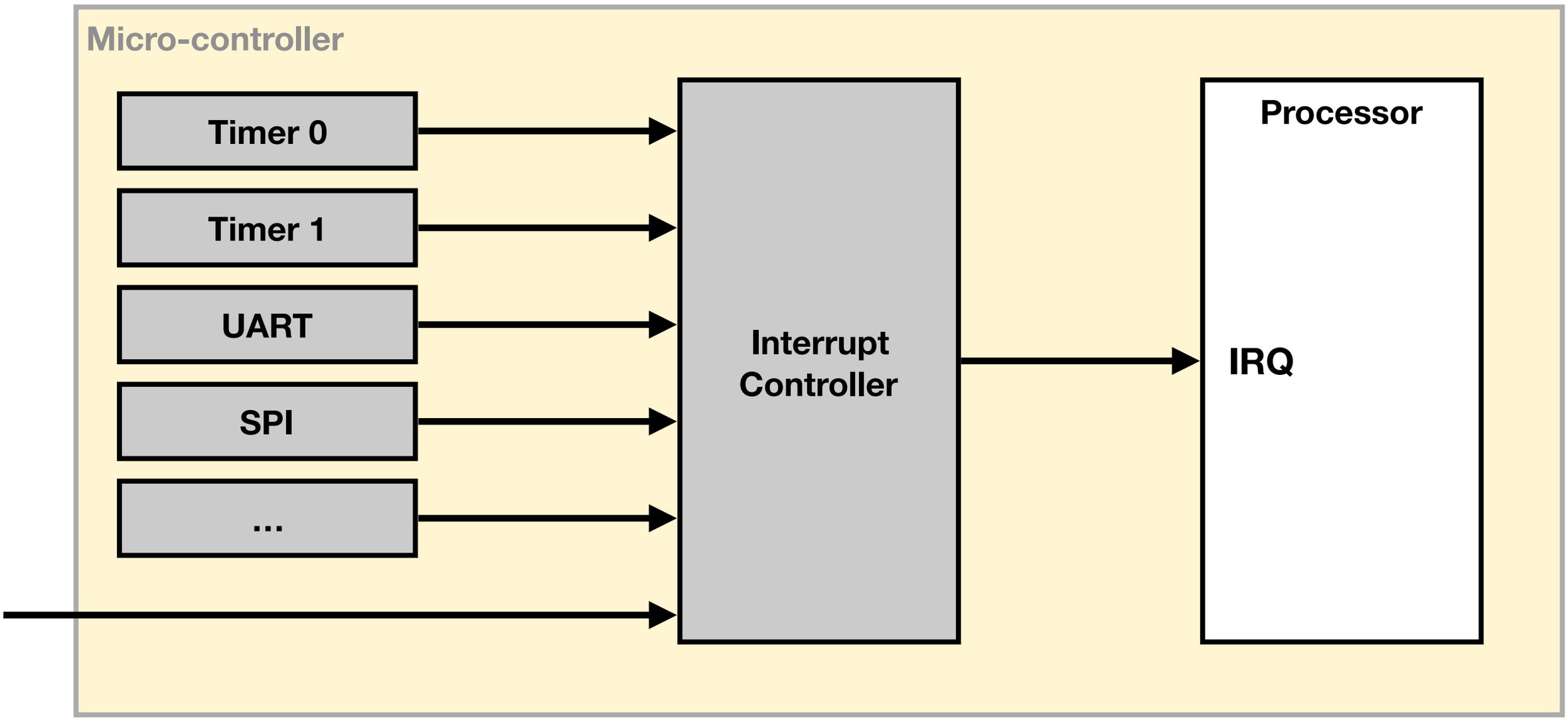
A journey in the real world

- When no task is ready something has to run
  - hidden **idle** task with the lowest possible priority
    - **always ready/running**
    - implements an infinite loop
      - can put the micro-controller in **idle/low power state**
- Periodic tasks (with precise time) / preemption
  - A hardware timer delivers an interrupt periodically (usually called the **systick**)
  - Usually a 1ms period is chosen for this interrupt but it can be adjusted to fit the needs of the application

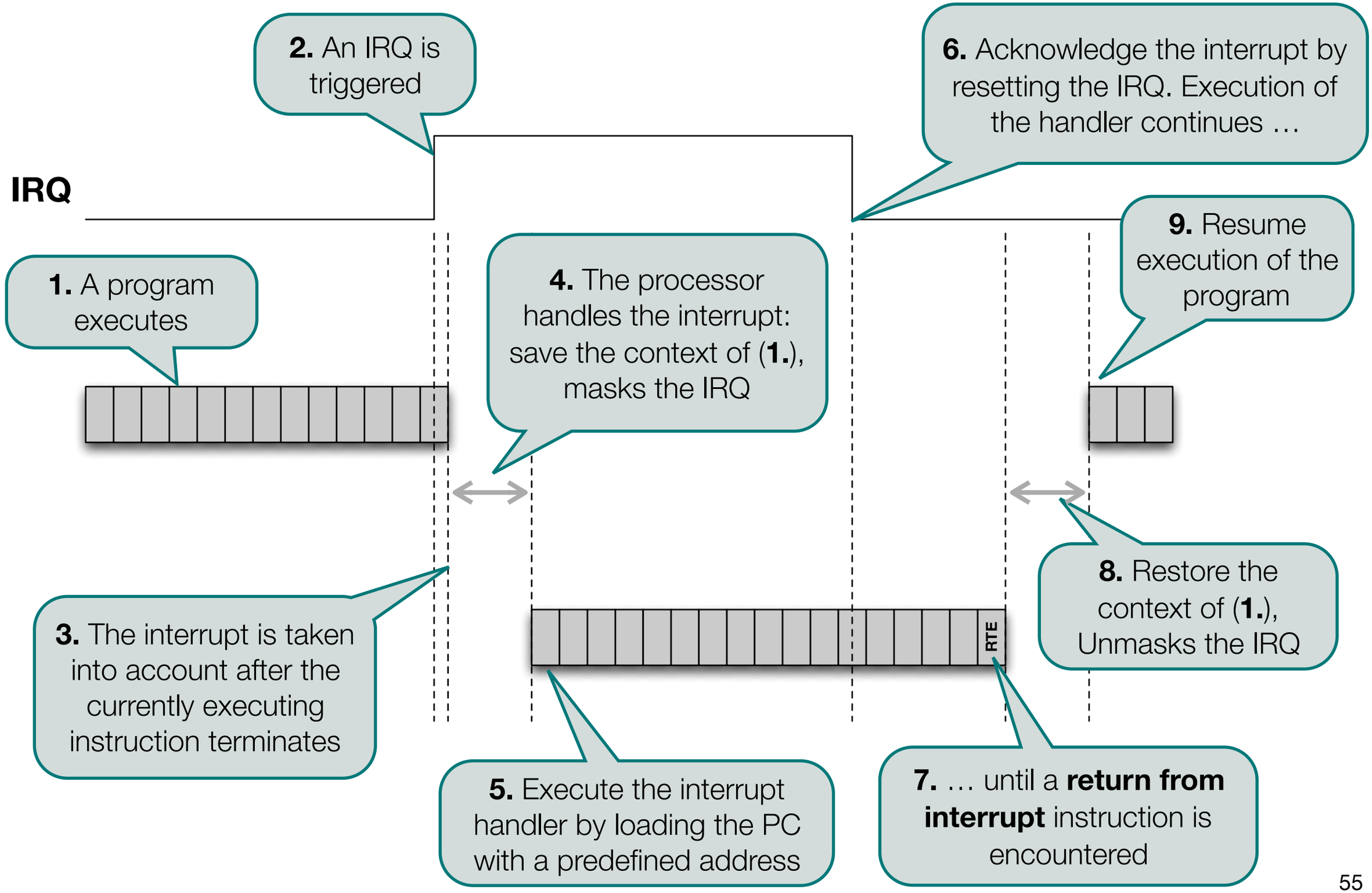
Interrupts (1)

- From hardware point of view, and interrupt line is a hardware input of the processor (**IRQ**, Interrupt ReQuest). The IRQ is a **state**
- The processor may be in states where:
  - the IRQ is ignored (aka masked) or
  - the IRQ is taken into account (unmasked)
- When the IRQ become **active** and if the interrupt is **unmasked**, the current executing program is halted and the processor executes another program called the **interrupt handler**.
- When the interrupt handler terminates, the processor goes back to the previously halted program.

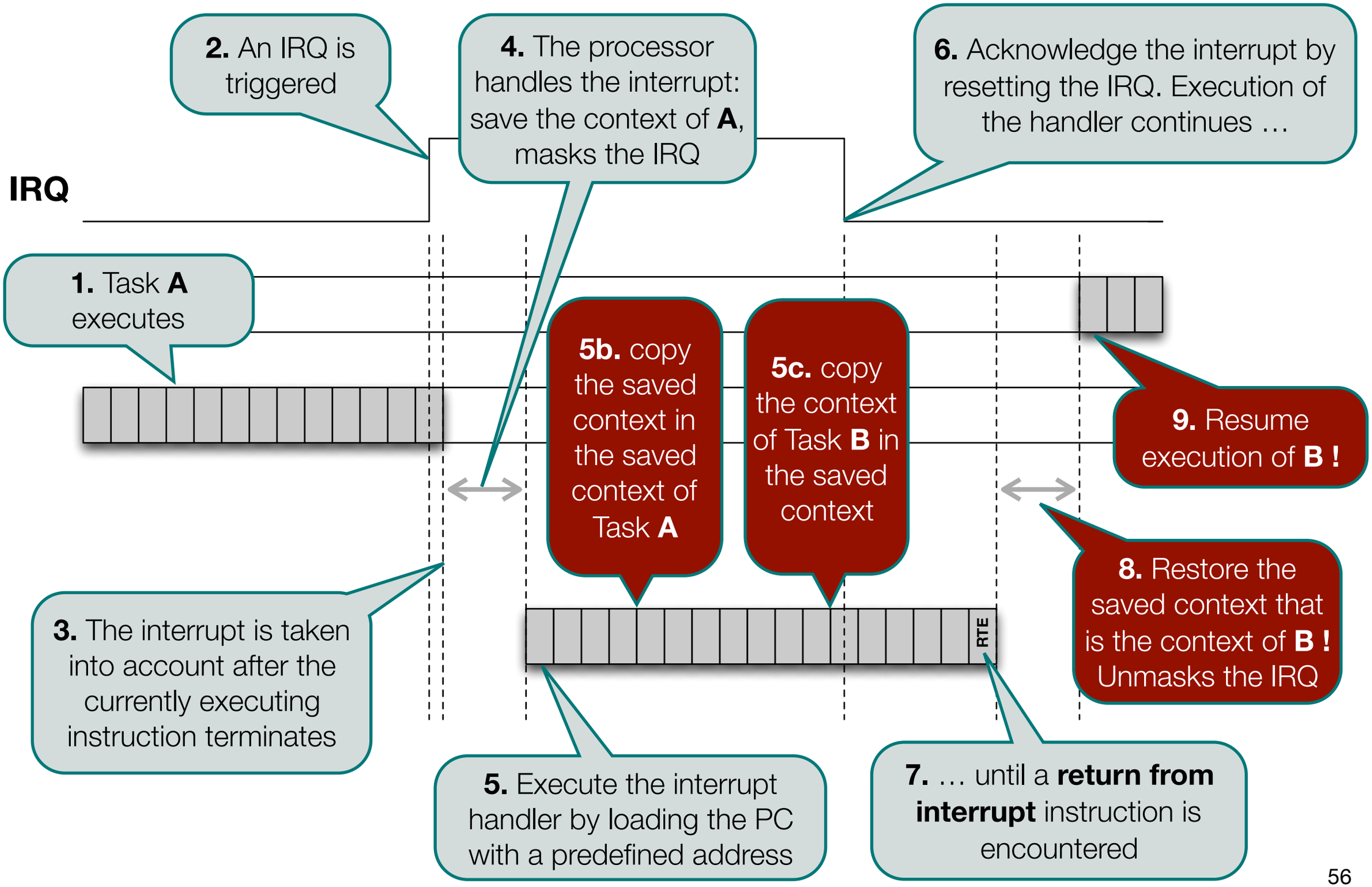
Interrupts (2)



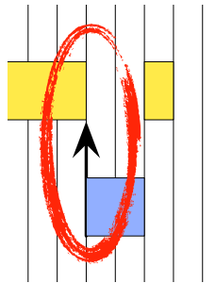
Interrupts (3)

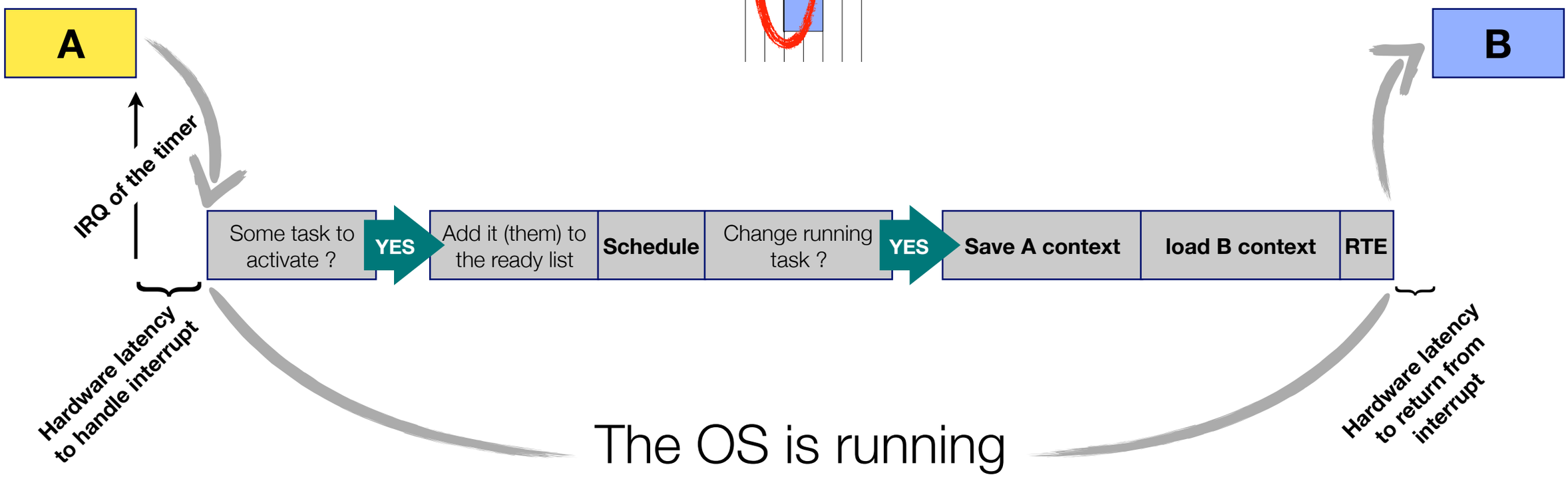


Interrupts and preemption

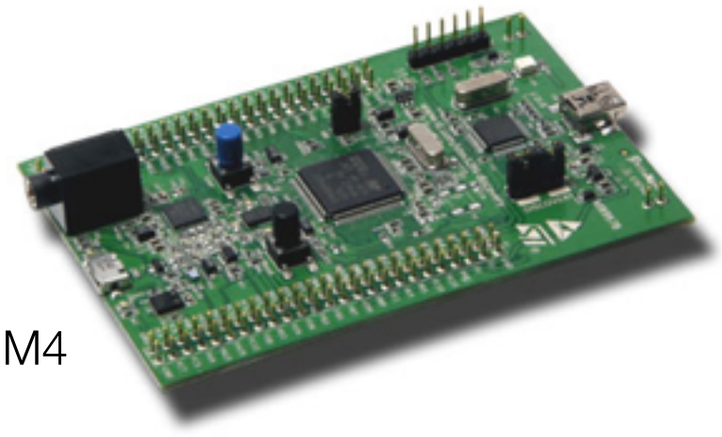


Preemption

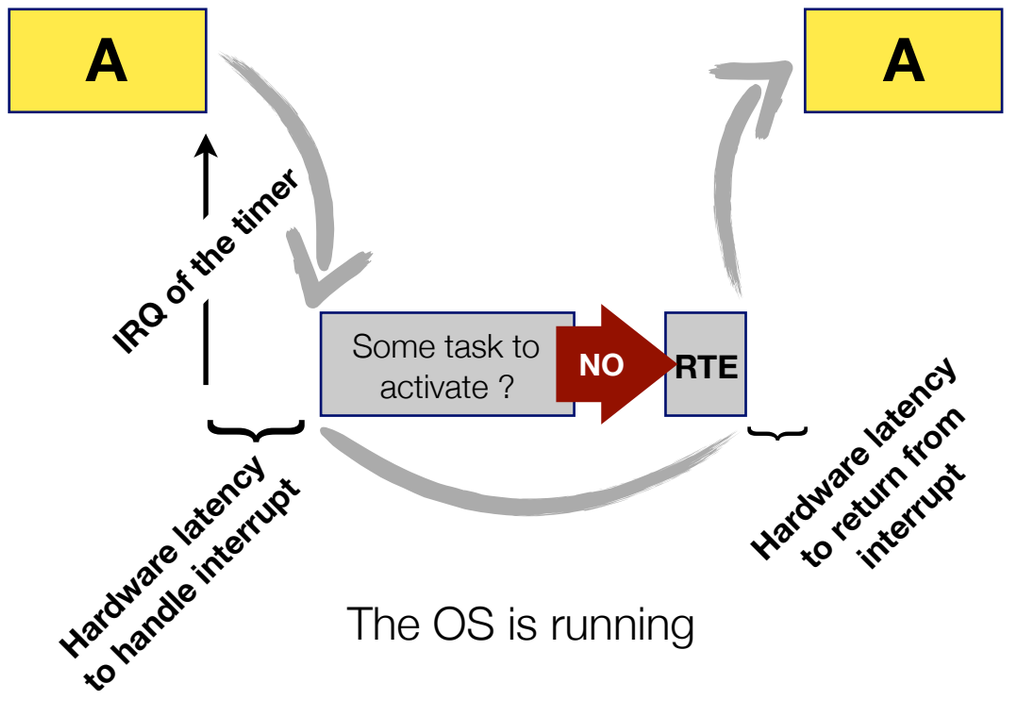




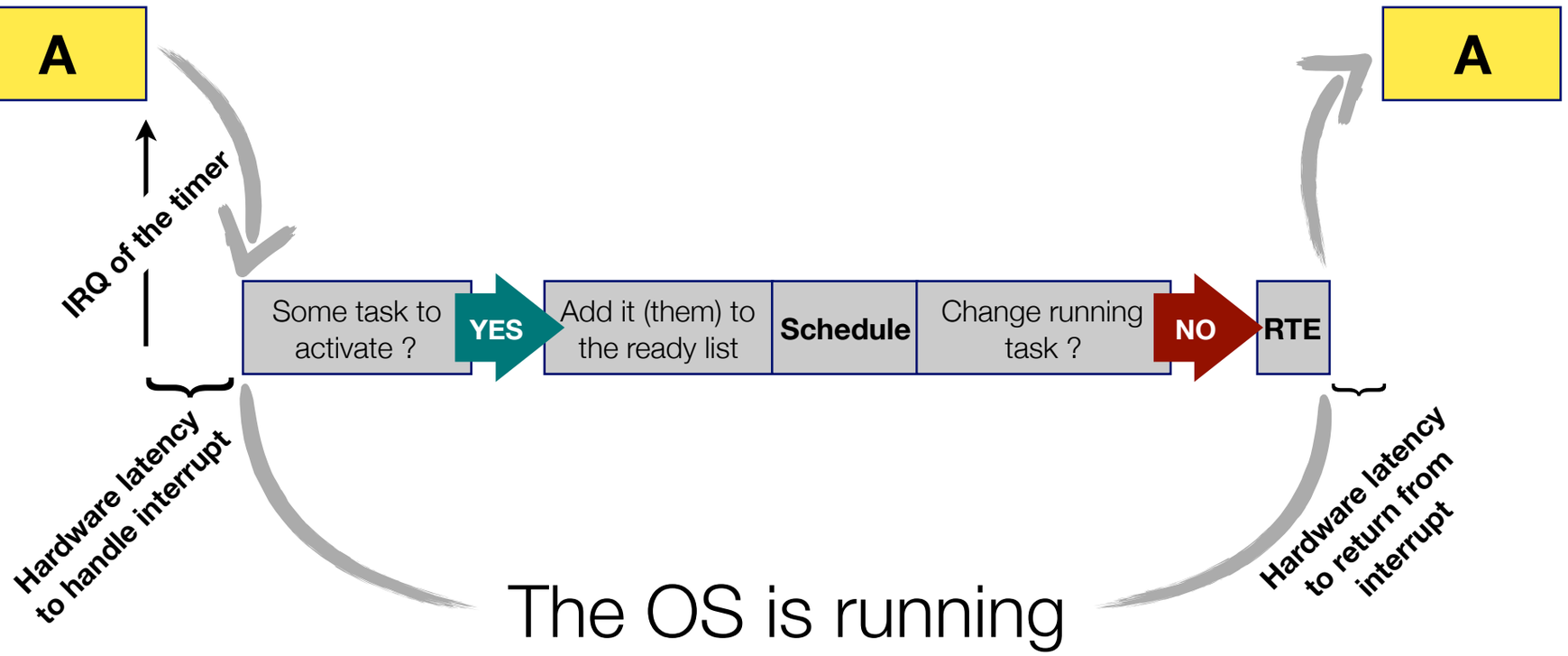
takes ~5µs on a 168MHz ARM Cortex M4



No preemption (1)



No preemption (2)



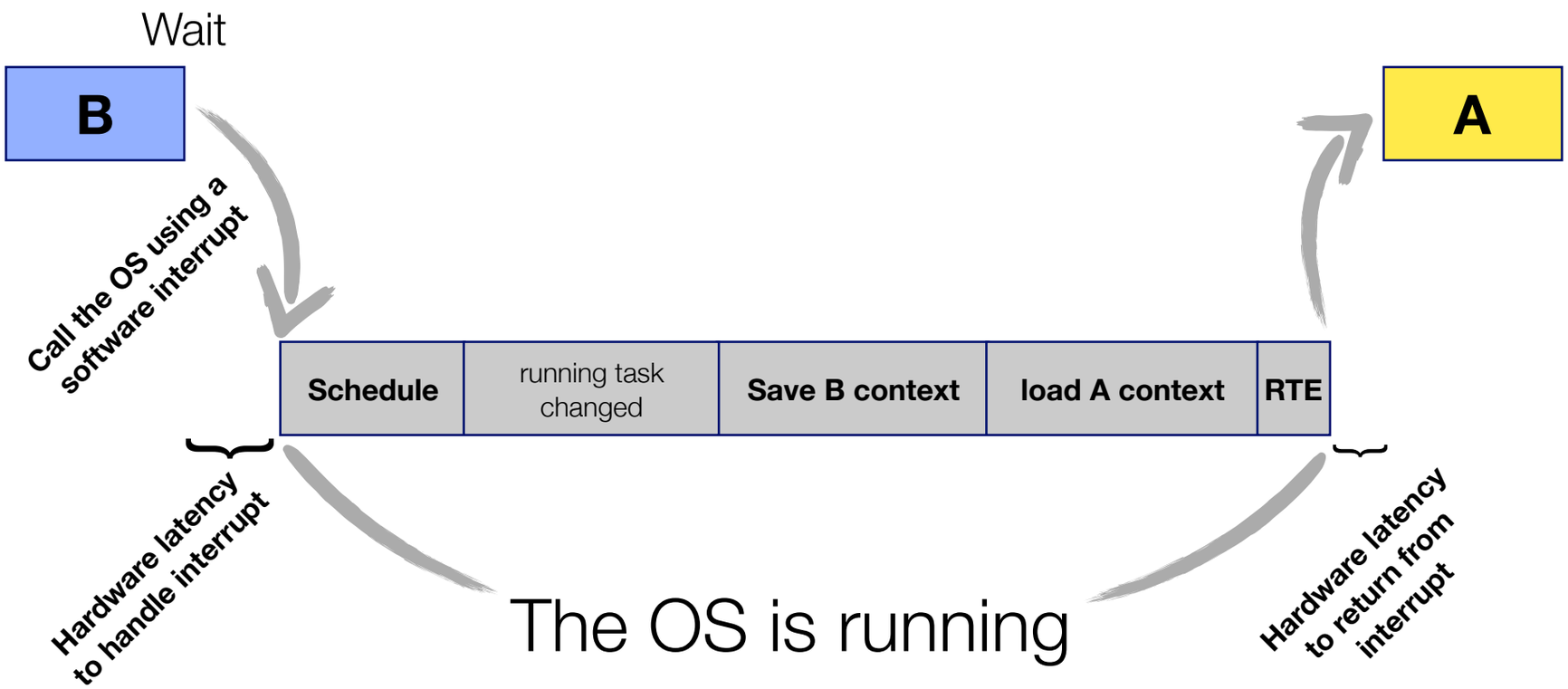
There are not only interrupts

- We saw the OS runs each time an interrupt dealing with tasks occurs but there are other occurrences that do not correspond to hardware interrupts: System Calls or System Services ➜ functions of the OS that can be called by the tasks.

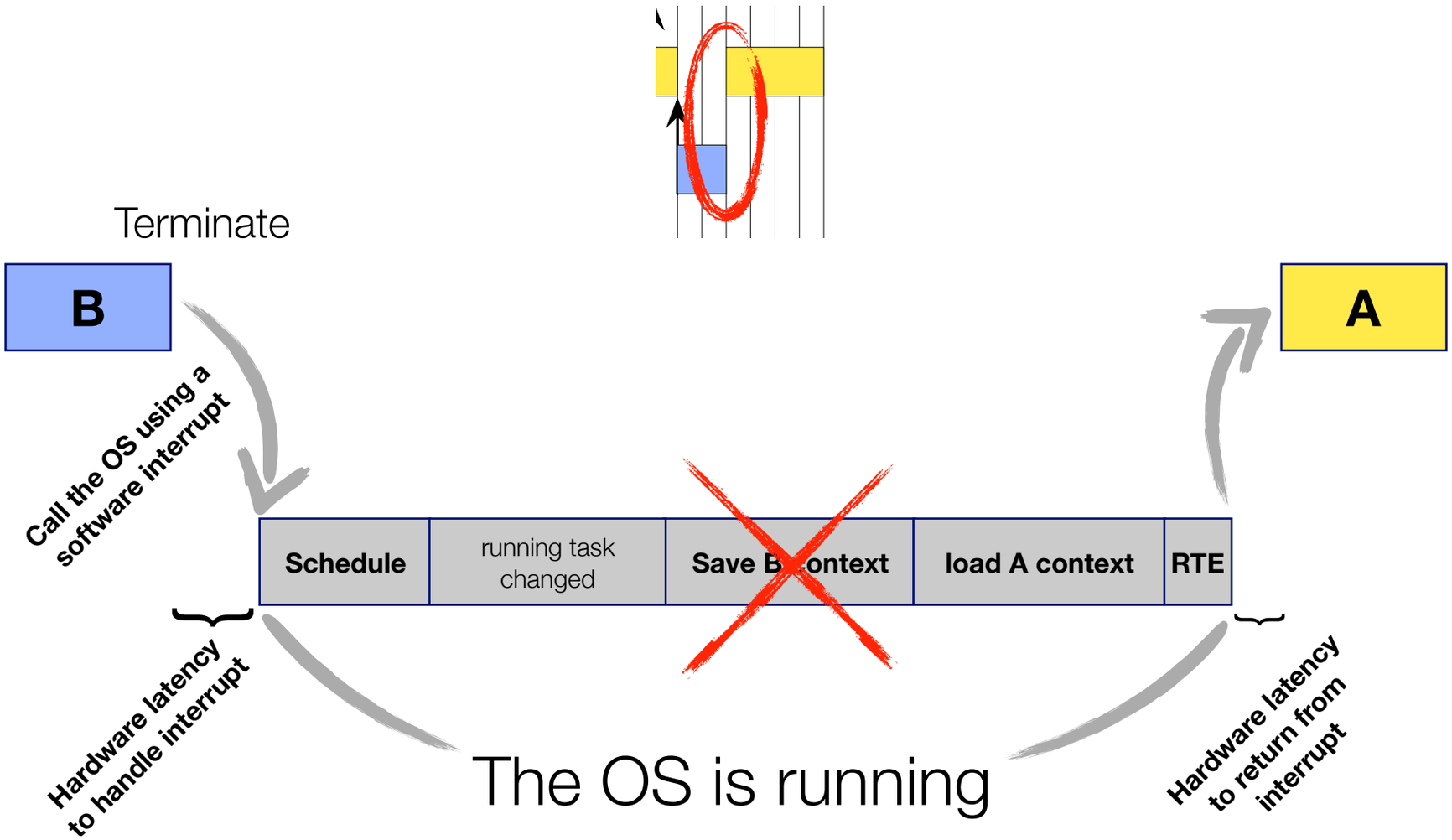
  Examples:

  - A task waits
  - A task terminates
  - A task activates another task (similar to activation by a hardware interrupt)
  - …

A task waits


Wait
B
A
Call the OS using a software interrupt
Schedule
running task changed
Save B context
load A context
RTE
Hardware latency to handle interrupt
Hardware latency to return from interrupt
The OS is running

A task terminates

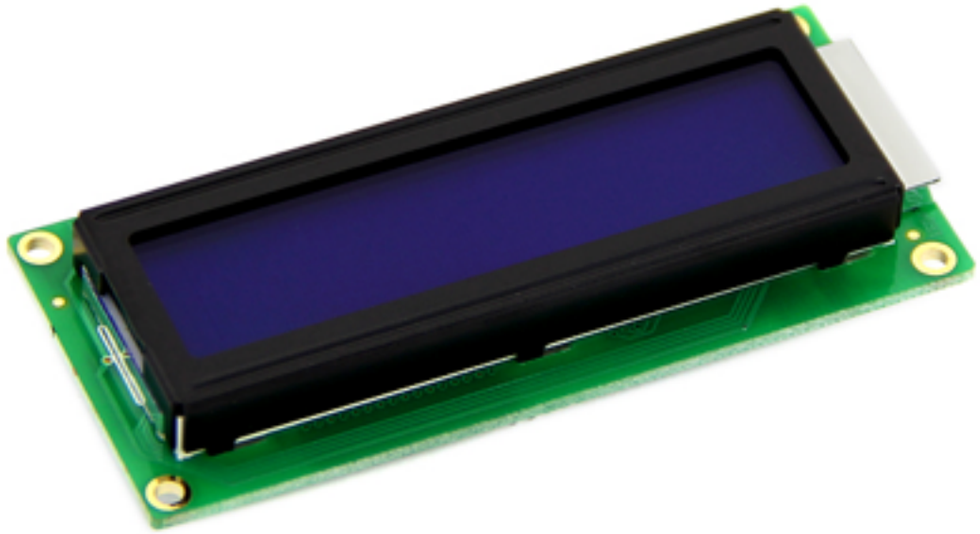


Resource sharing and Synchronization

Synchronization purpose

- Tasks may need to synchronize to
  - synchronize the execution of (part of) code to a particular event
  - manage concurrent access to a shared resource

- Example: shared access to a liquid crystal display



Resource sharing / LCD example (1)

- Access to a LCD is a communication, usually a parallel 8 bits port (using the GPIO). Usually the chip which drives the screen is a Hitachi HD44780
  - Putting a character on the LCD is a single write
  - Clearing / setting the cursor position is a single write too

- 2 periodic tasks (t1 and t2, prio(t2) > prio(t1) so that t2 can preempt t1):
  - t1 sets the cursor at column 0, line 0 and writes « MASTER »
  - t2 sets the cursor at column 0, line 1 and writes « WET »

Does it work ?

Resource sharing / LCD example (2)

Arduino like code using LiquidCrystal

```
TASK(t1)
{
  lcd.setCursor(0,0);
  lcd.print(“MASTER”);
  TerminateTask();
}
```

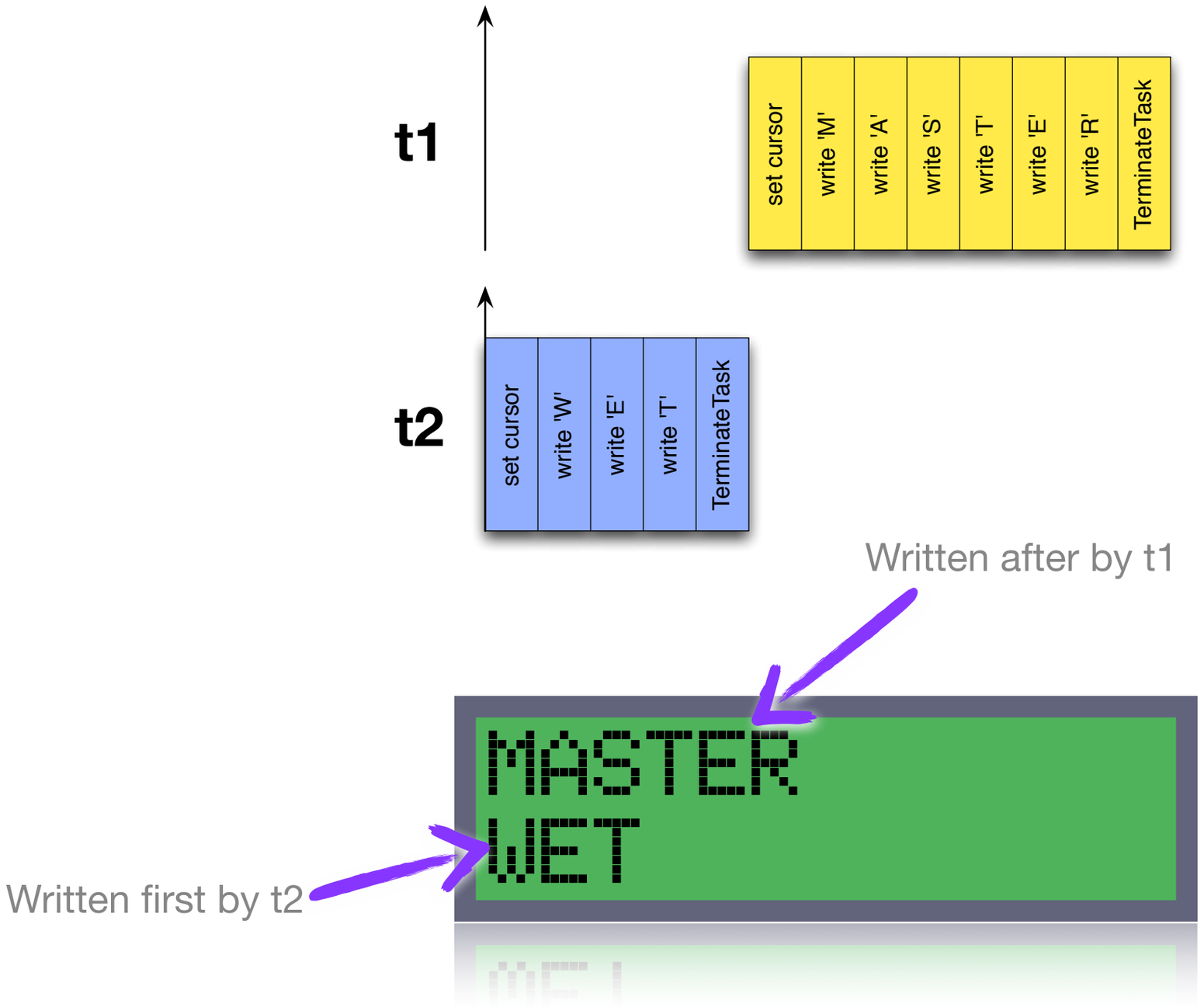
```
TASK(t2)
{
  lcd.setCursor(0,1);
  lcd.print(“WET”);
  TerminateTask();
}
```

Actually lcd.print writes chars one by one after getting

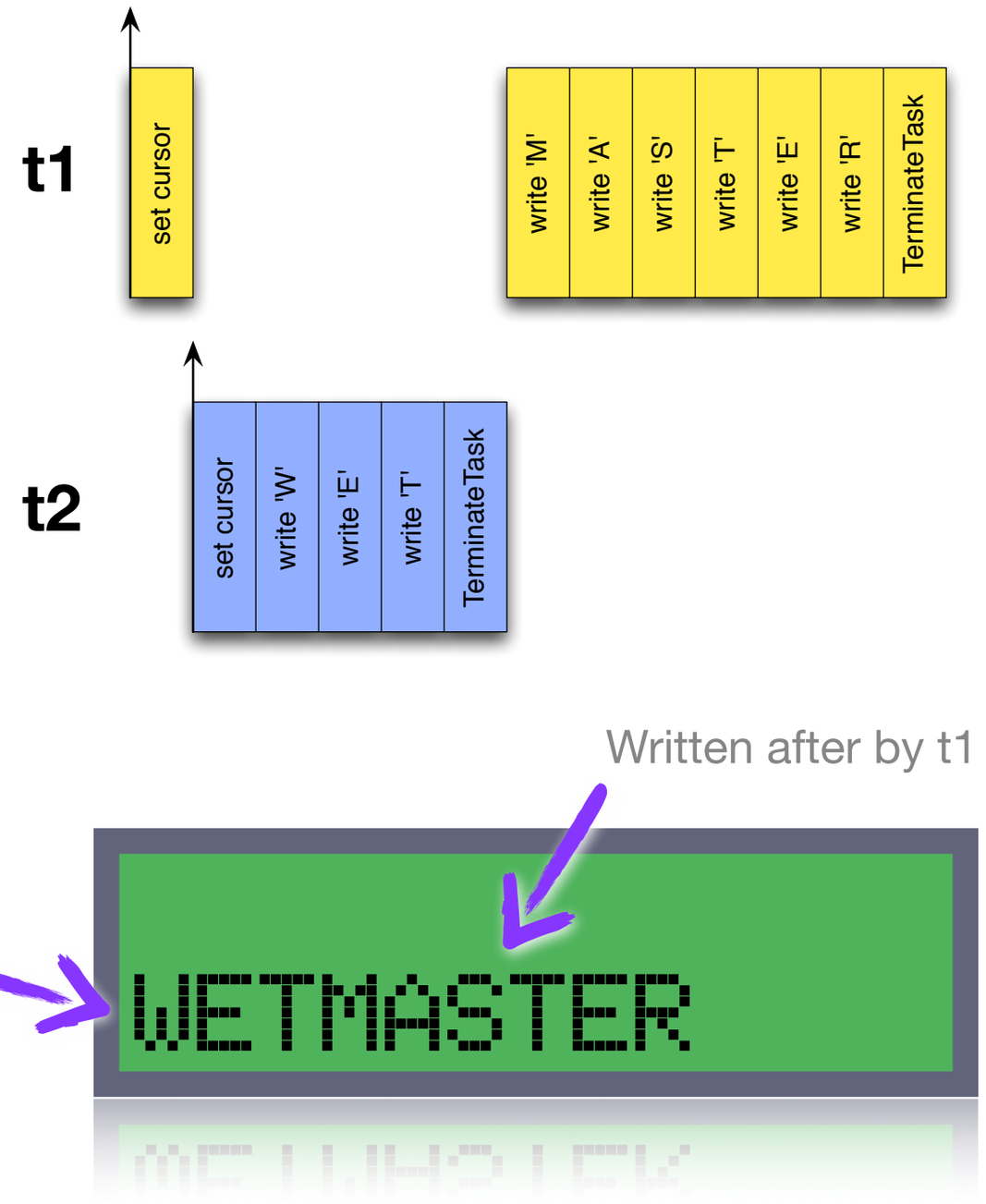
- the length of the string in the size variable
- a pointer to the string in the buffer variable

```
while (size--) {
  if (write(*buffer++)) n++;
  else break;
}
```

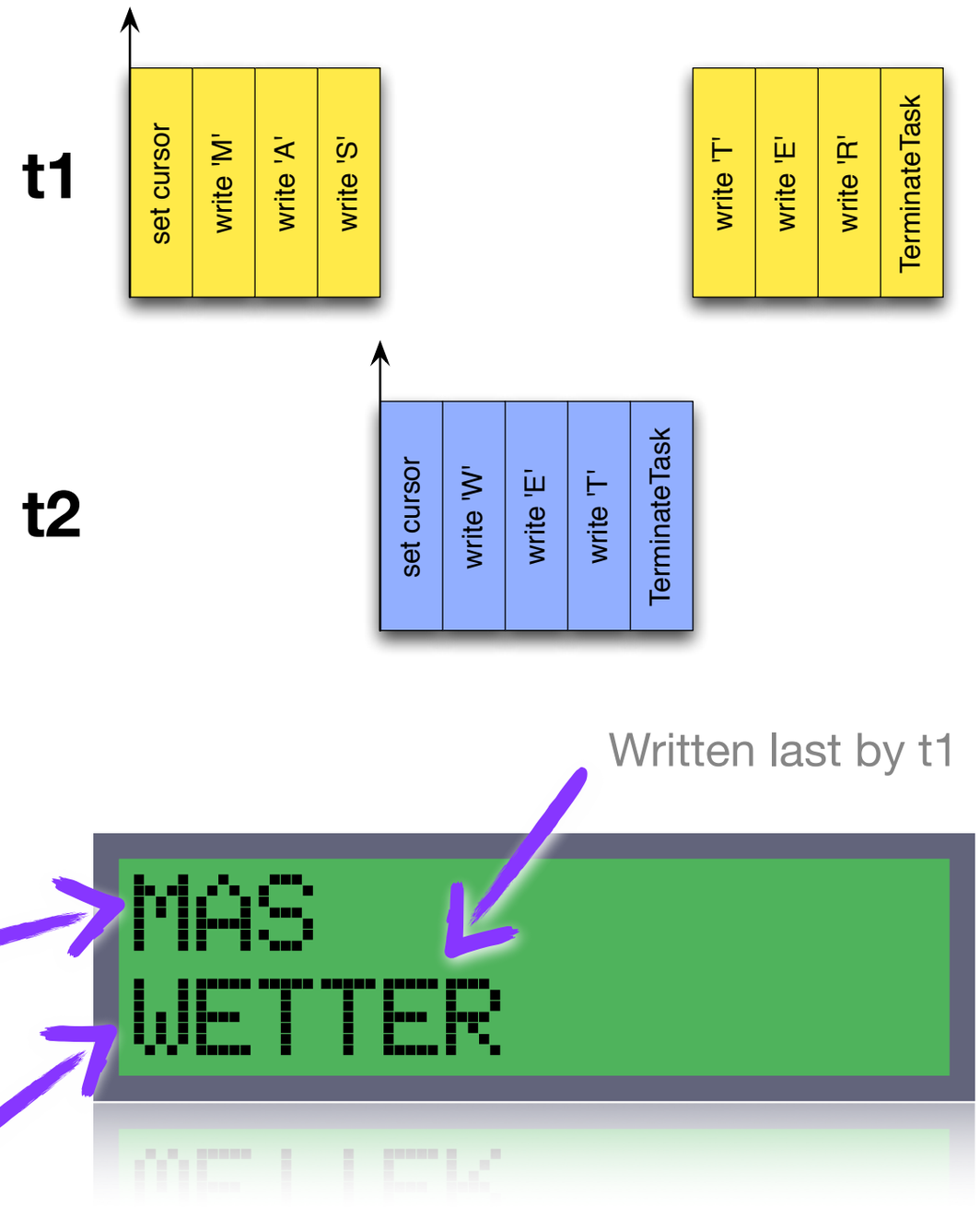
Resource sharing / LCD example (3)



Resource sharing / LCD example (4)



Resource sharing / LCD example (5)



Resource sharing / variable example (1)

```
volatile int i = 0;
```

```
TASK(t1)
{
  i++;
  TerminateTask();
}
```

```
ldr  r3, [r2]
add  r3, r3, #1
str  r3, [r2]
```

Actually i++ is not a single instruction of the processor

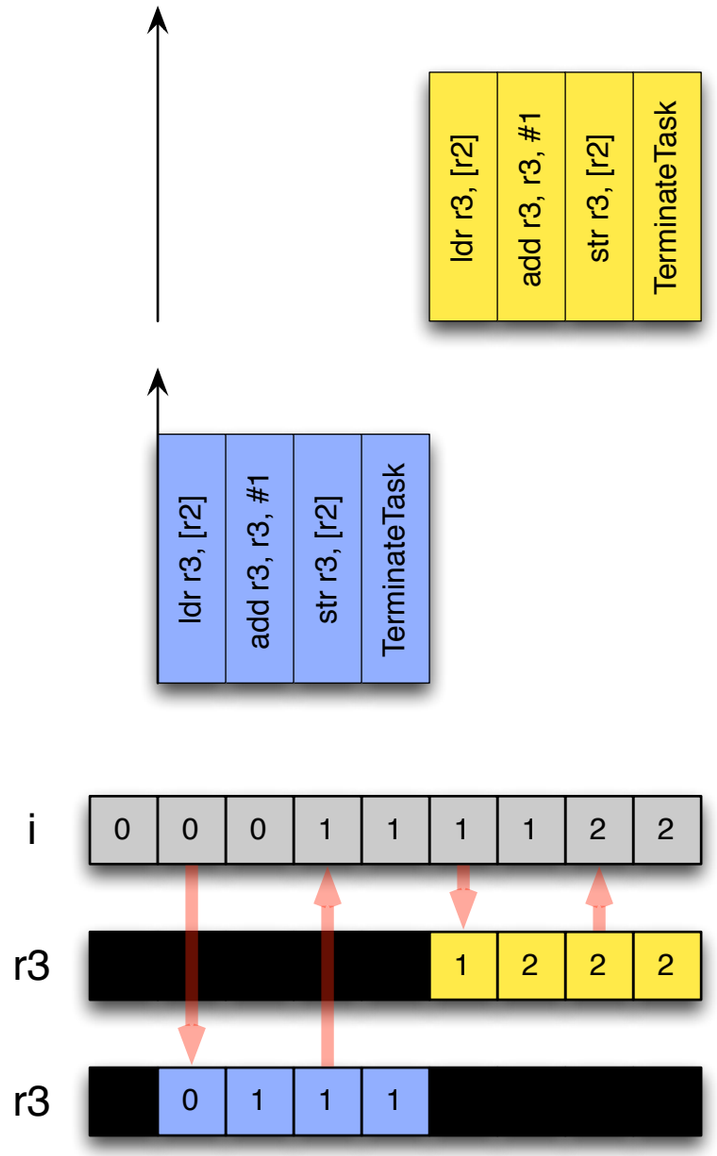
On ARM CORTEX, assuming r2 contains a pointer to i

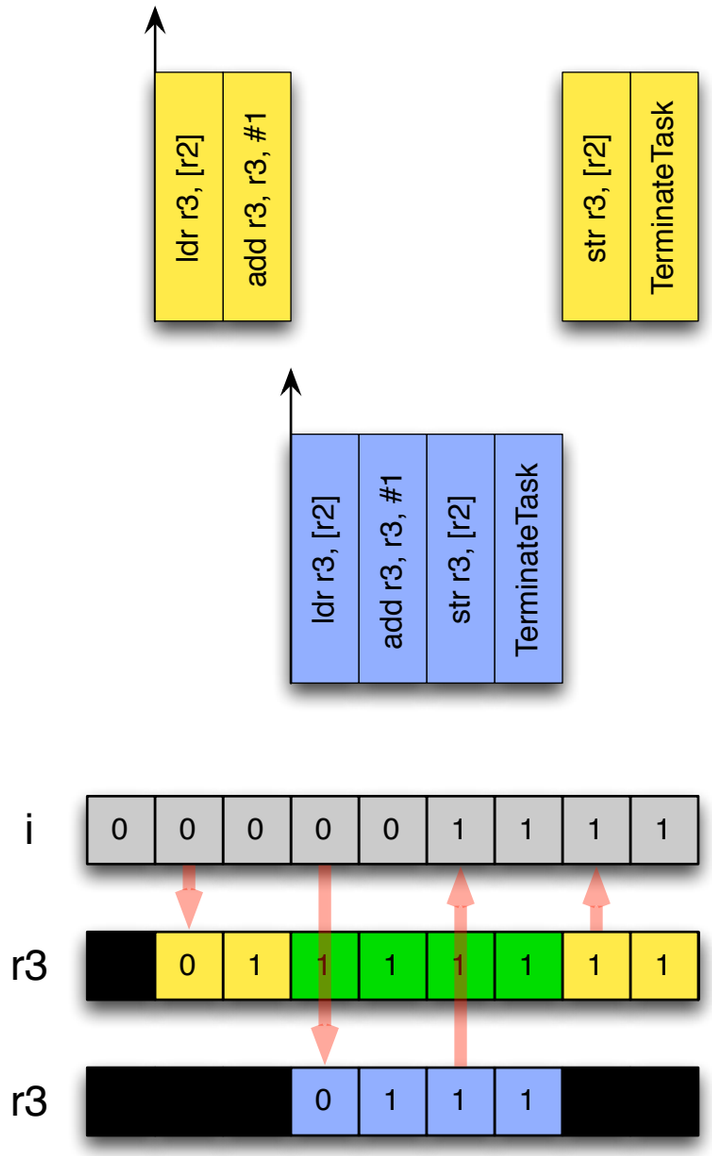
```
TASK(t2)
{
  i++;
  TerminateTask();
}
```

```
ldr  r3, [r2]
add  r3, r3, #1
str  r3, [r2]
```

prio(t2) > prio(t1)

Resource sharing / variable example (2)



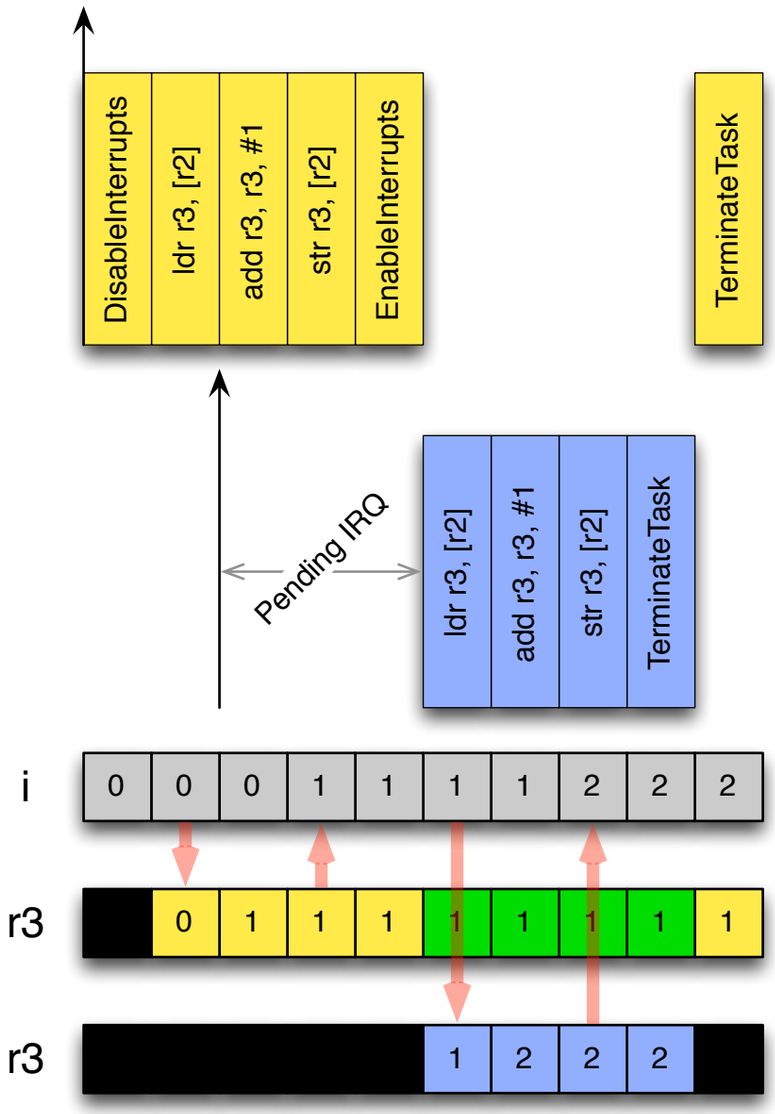


Resource sharing / no interrupts

```
volatile int i = 0;
```

```
TASK(t1)
{
  DisableInterrupts();
  i++;
  EnableInterrupts();
  TerminateTask();
}
```

```
TASK(t2)
{
  i++;
  TerminateTask();
}
```



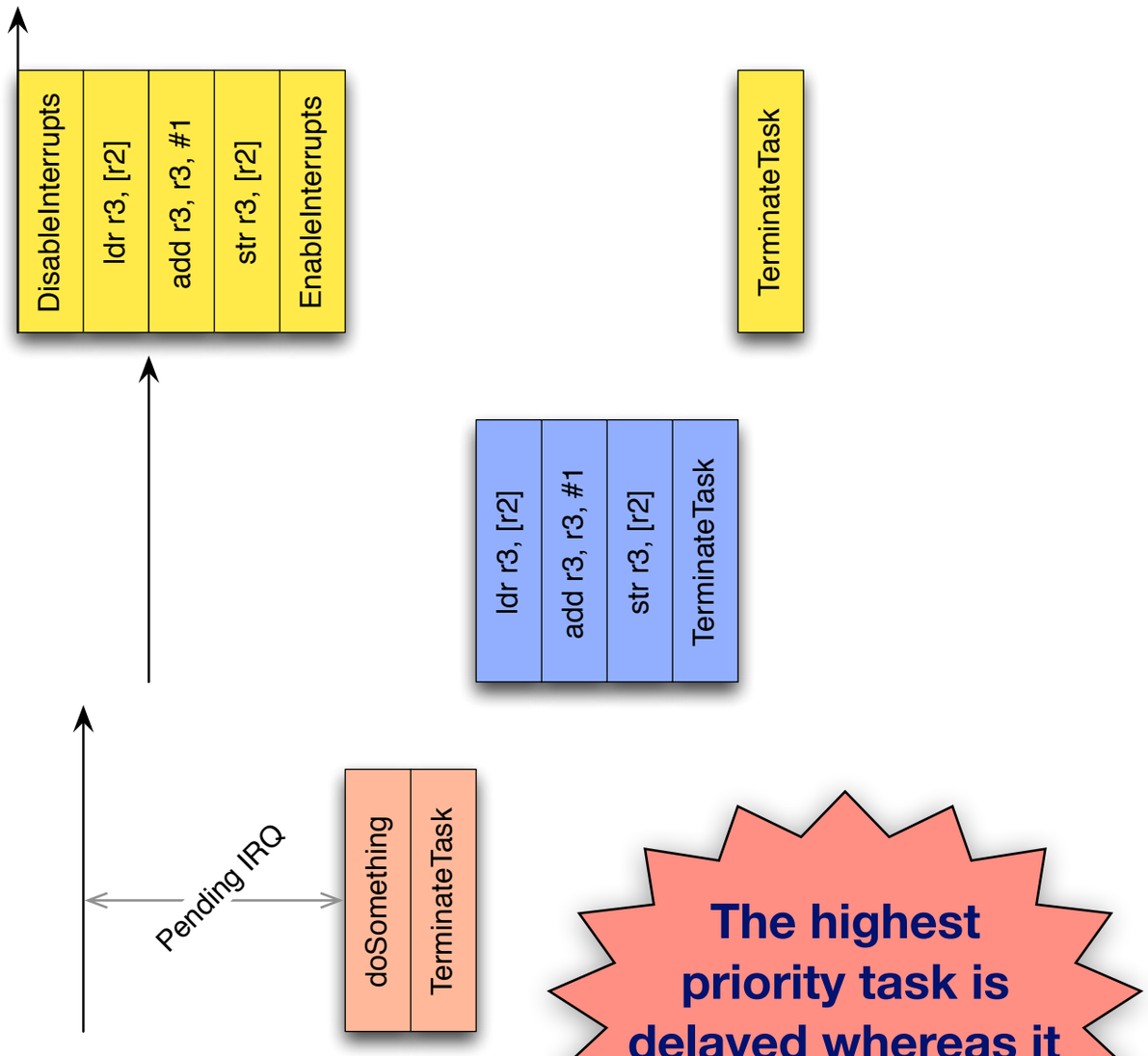
Resource sharing / no interrupts

```
volatile int i = 0;
```

```
TASK(t1)
{
  DisableInterrupts();
  i++;
  EnableInterrupts();
  TerminateTask();
}
```

```
TASK(t2)
{
  i++;
  TerminateTask();
}
```

```
TASK(unrelated)
{
  doSomething();
  TerminateTask();
}
```



prio(unrelated) > prio(t2) > prio(t1)

Resource sharing / how ?

- What we want is a mutual exclusion mechanism such that 2 or more tasks cannot execute code accessing/modifying a shared resource (device, variable) at the same time.
  - A **critical section** is an abstract portion of code delimited by proper services calls which reference the critical section (using an ID for instance).
  - more than one critical section, each with an id, can be defined for each of the shared resource.
  - When a task is within a critical section, another task cannot enter the same critical section.
  - Viewed from the other task, all the code executing within the critical section is **atomic**.
  - This rely on **atomic hardware mechanism**: Since service calls are executing with interrupts masked, it is ok (at least in monocore).

A simple mutex

- Let's say we have a special datatype called a mutex (for MUTual EXclusion)
  - `mutex m;` defines a mutex called **m**
  - `lock(m);` locks m, it should be used at the start of a critical section
  - `unlock(m);` unlocks m, it should be used at the end of a critical section
  - any task calling `lock` on a mutex already locked is put in the **waiting state**
  - when a mutex is unlocked, one or more tasks waiting for the mutex are put back in the **ready state** and a rescheduling occurs

A simple mutex : example

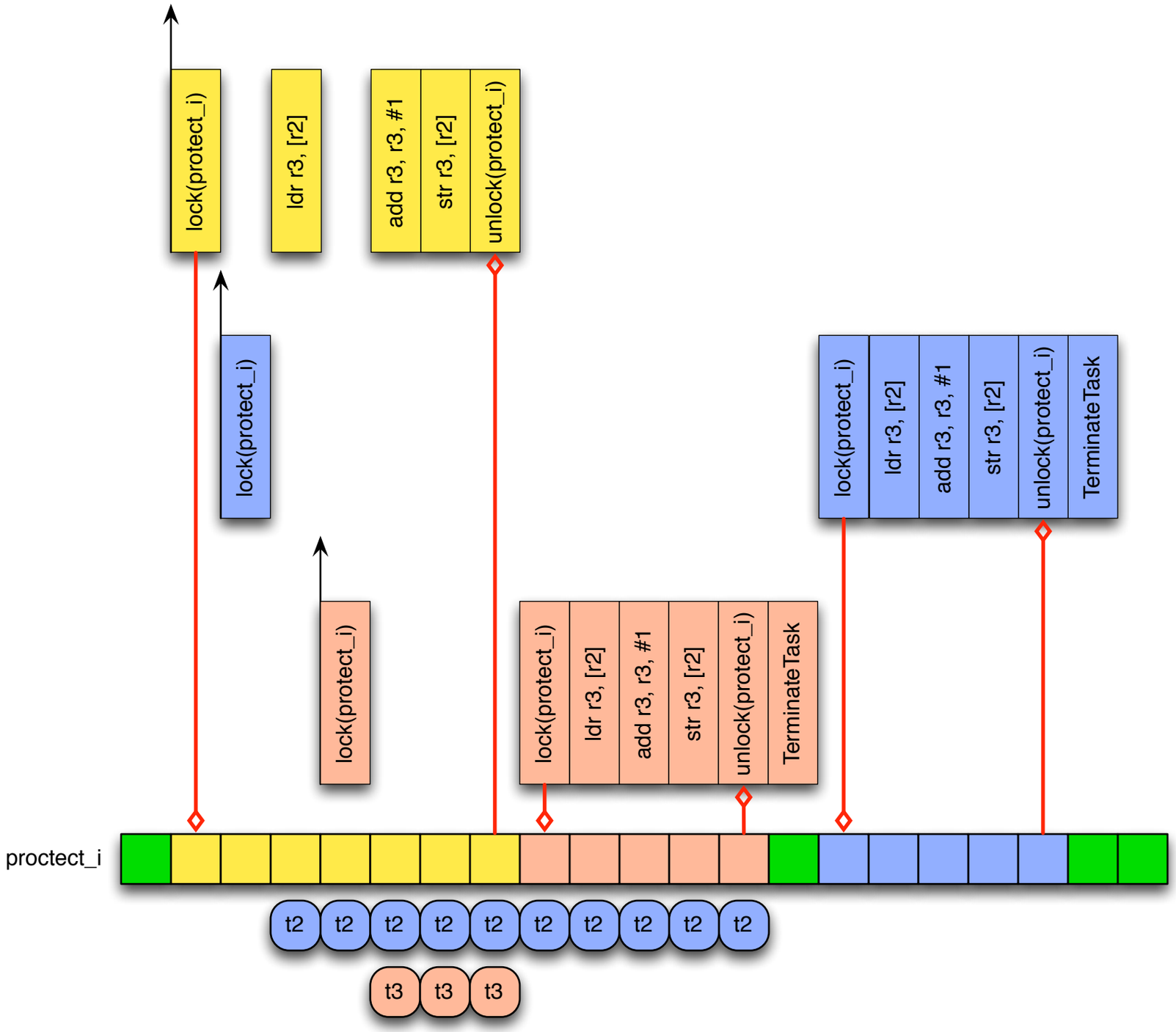
```
mutex protect_i;
```

```
TASK(t1)
{
  lock(protect_i);
  i++;
  unlock(protect_i);
  TerminateTask();
}
```

```
TASK(t2)
{
  lock(protect_i);
  i++;
  unlock(protect_i);
  TerminateTask();
}
```

```
TASK(t3)
{
  lock(protect_i);
  i++;
  unlock(protect_i);
  TerminateTask();
}
```

prio(t3) > prio(t2) > prio(t1)



Semaphore

- Proposed by Edger Dijkstra
- It allows to protect access to shared resources
- This mechanism, available in some OS Posix compliant for instance) offers 3 functions:
  - Init() : initialize the semaphore;
  - P() to test the semaphore (Probieren);
  - V() to increment the semaphore (Verhogen).
- Works like mutex but with a counter. Identical to a mutex if the counter is initialized to 1.

Semaphore

- A counter is associated to the semaphore.
  - A call to P() is used to enter the critical section:
    - If the counter is > 0, it is decremented and the resource may be taken.
    - If the counter is = 0, the task which called P() is put in the waiting state until the counter became > 0. At that time the task will be awaken and the counter will be decremented again.
- A call to V() is used release a resource:
  - The counter is incremented and a task which is waiting may be put in ready state.

Semaphore

- The counter associated to the semaphore may have a non-binary value
- example: access to a buffer:
  - Here are 2 functions to read and write the buffer. These functions may be called by concurrent tasks.
- The S1 semaphore which has the initial value of its counter equal to 5 allows up to 5 writes. After that, a task which try to write is put in the waiting state until a task does a read.

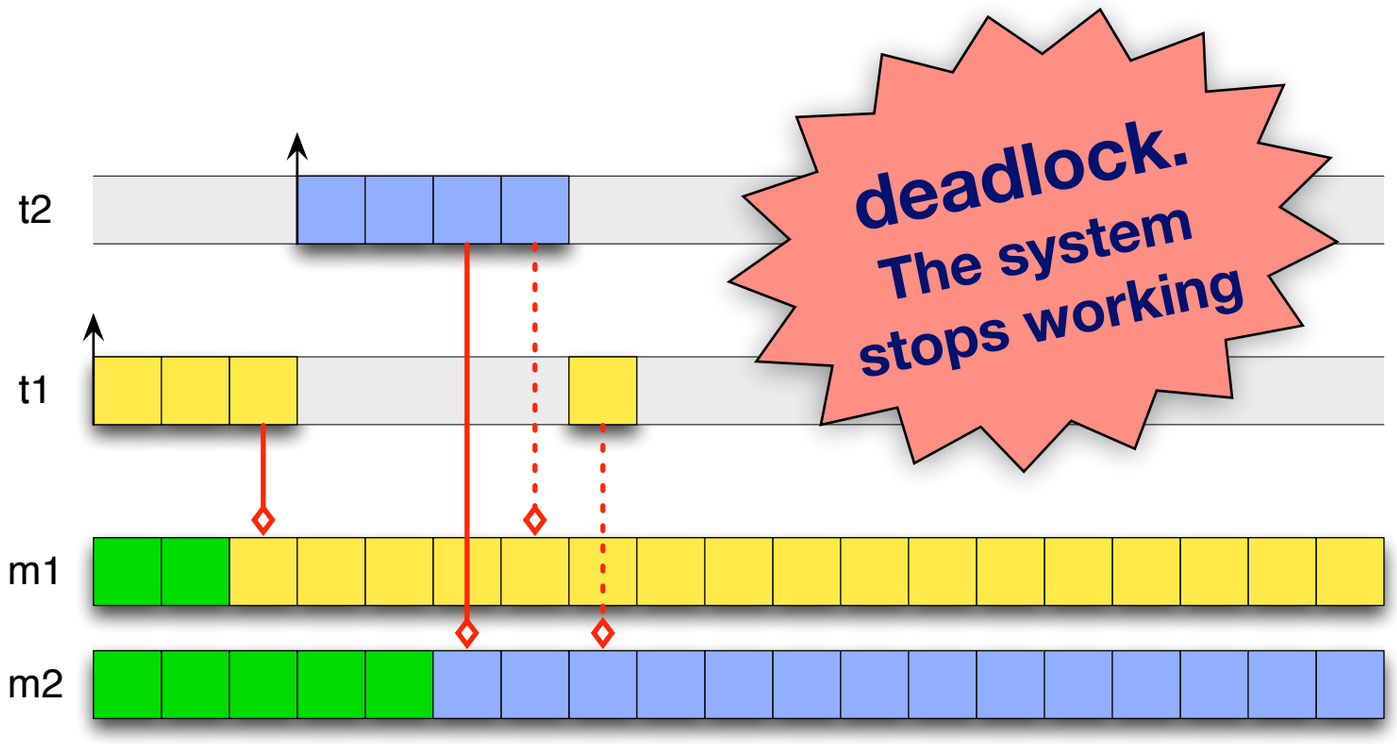
```
void init() {
    semInit(S1, 5);
}
```

```
void WriteBuffer(int data) {
   P(S1);
   //buffer write
}

void ReadBuffer(int *data) {
   V(S1);
   //buffer read
}
```

Problem: deadlock

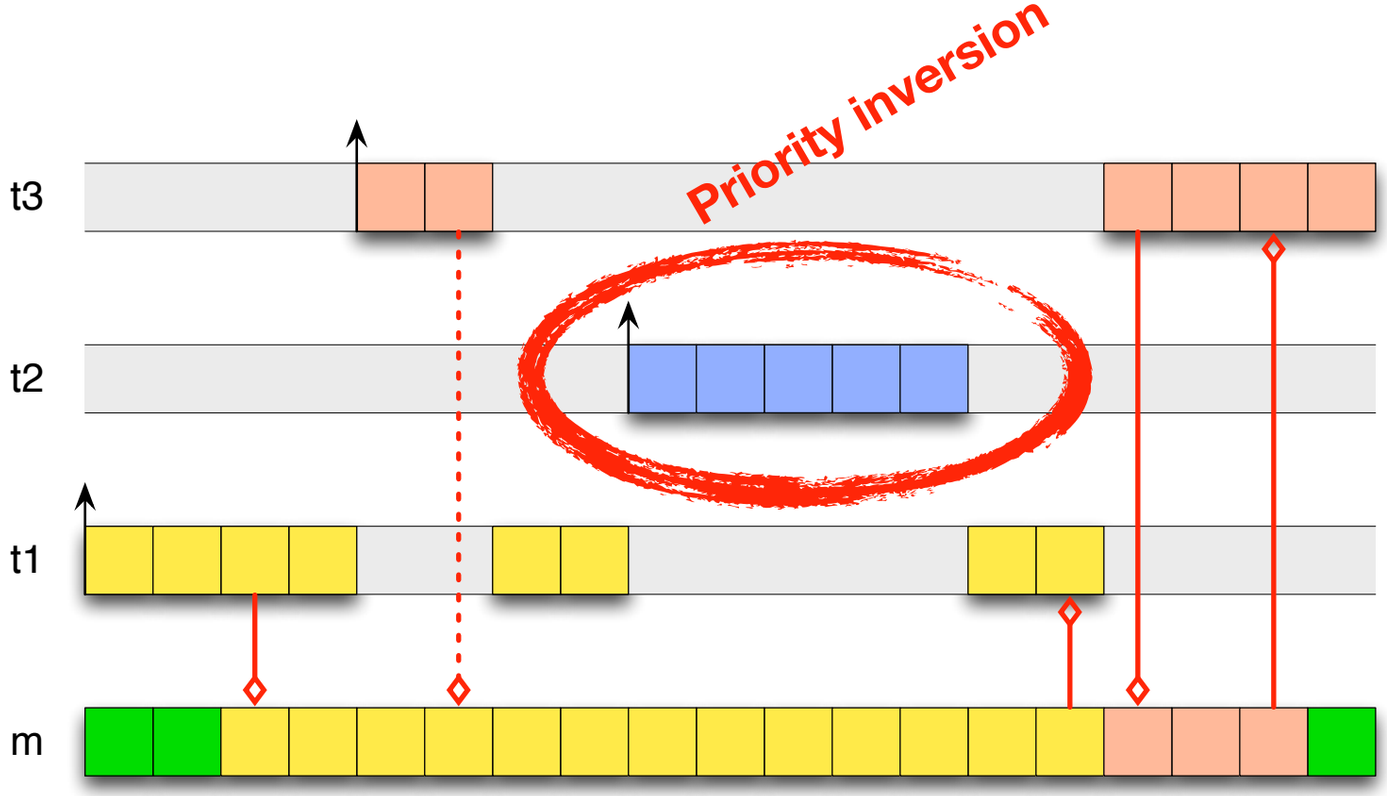
- Classical synchronization mechanisms (semaphore, mutex) may have deadlocks:
  - example: 2 tasks take 2 mutex in reverse order, prio(t2) > prio(t1)

```
mutex m1;
mutex m2;
volatile int i;
volatile int j;

TASK(t1)
{
   lock(m1);
   lock(m2);
   i++;
   j--;
   unlock(m2);
   unlock(m1);
   TerminateTask();
}

TASK(t2)
{
   lock(m2);
   lock(m1);
   i--;
   j++;
   unlock(m1);
   unlock(m2);
   TerminateTask();
}
```

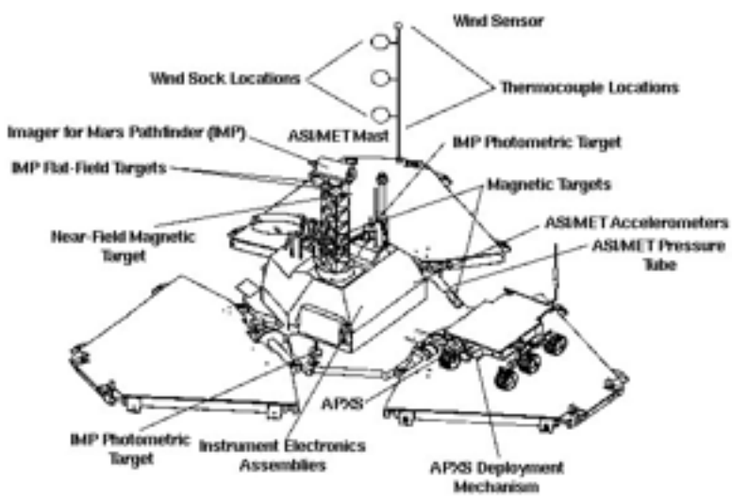


Problem: priority inversion

- Classical synchronization mechanisms (semaphore, mutex) may have the priority inversion problem:
  - A task with a lower priority may delay a higher priority task.
  - The following example shows 3 preemptable tasks.
    prio(t3) > prio(t2) > prio(t1):



Caused the failure of Mars Pathfinder Probe



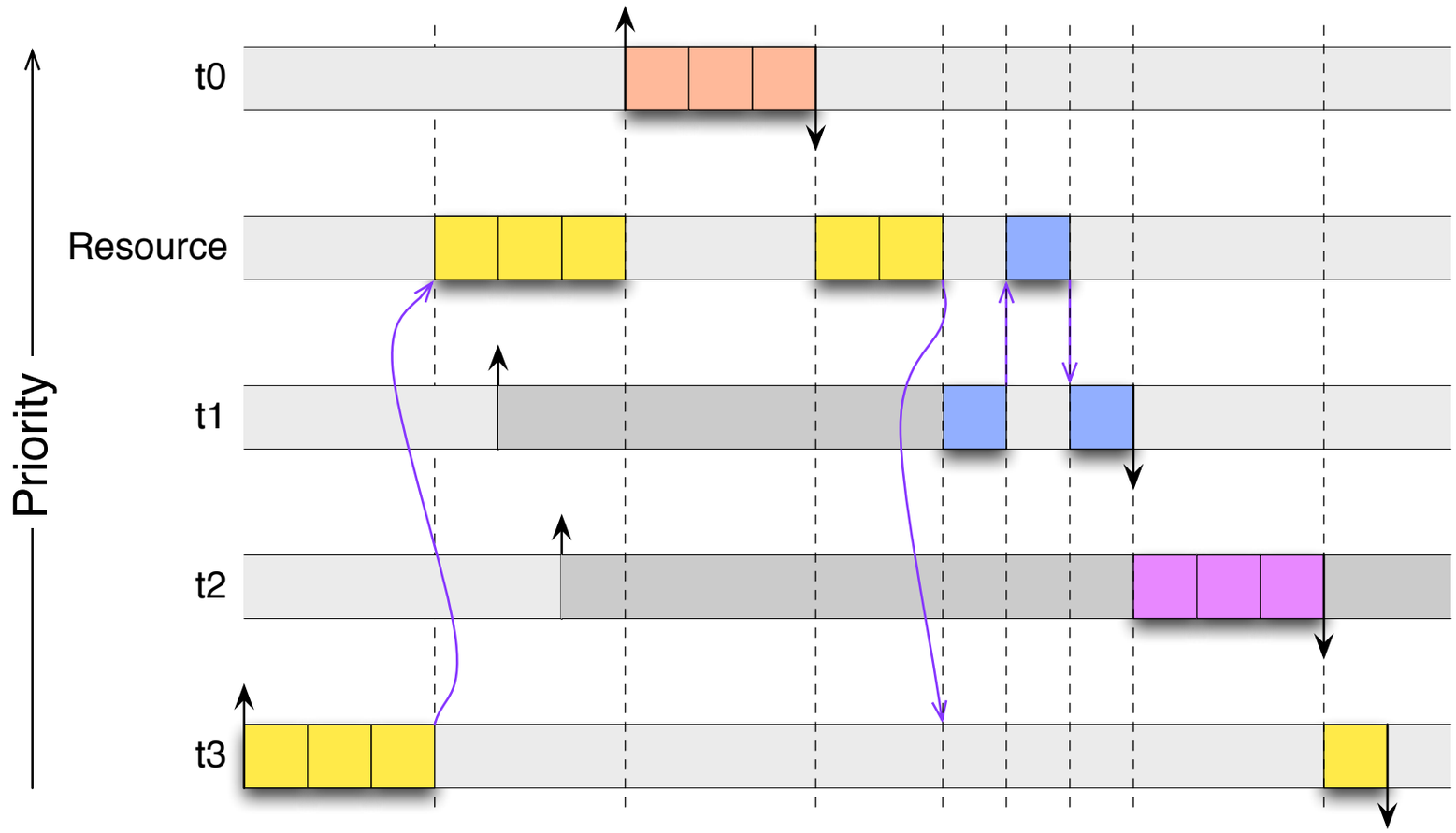
IPCP: Immediate Priority Ceiling Protocol

- Priority based protocol
  - works with fixed priority scheduling
  - the mutex (usually called a Resource) has a priority
    - the priority is the max of the priorities of tasks using the Resource
    - when a task takes the resource, **its priority is raised to the priority of the Resource** (if needed) ⇒ other tasks likely to take the Resource

      cannot preempt the task that holds the Resource
    - when the task release the Resource, **its priority is lowered to the previous priority** (if needed) and a **rescheduling occurs**.
- Deadlocks are not possible because resources are in strict ordering (the priority)
- Priority inversion is not possible because the priority changes

IPCP example

- 4 tasks
  - prio(t0) > prio(Resource) ≥ prio(t1) > prio(t2) > prio(t3)
  - t1 and t3 share a Resource



Events

- Events are a synchronization scheme.
  - An event is a flag, a boolean variable, used to signal something happened
  - It is a N producers / 1 consumer model (private events) or N producers / M consumers model (public events).
  - A set of events is called an event mask
- Not designed for critical sections
- A task may
  - set an event
  - clear an event
  - wait for an event
  - get the current events

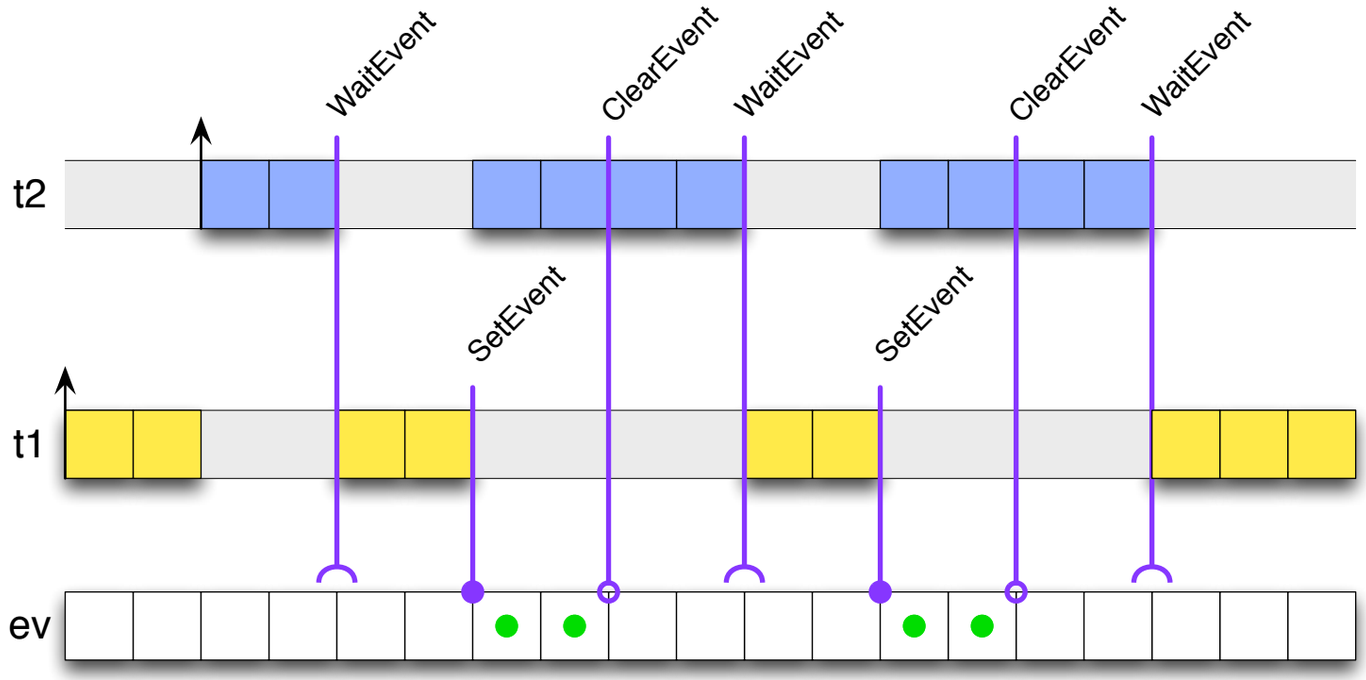
Events : example

- prio(t2) > prio(t1)

```
Event ev;
```

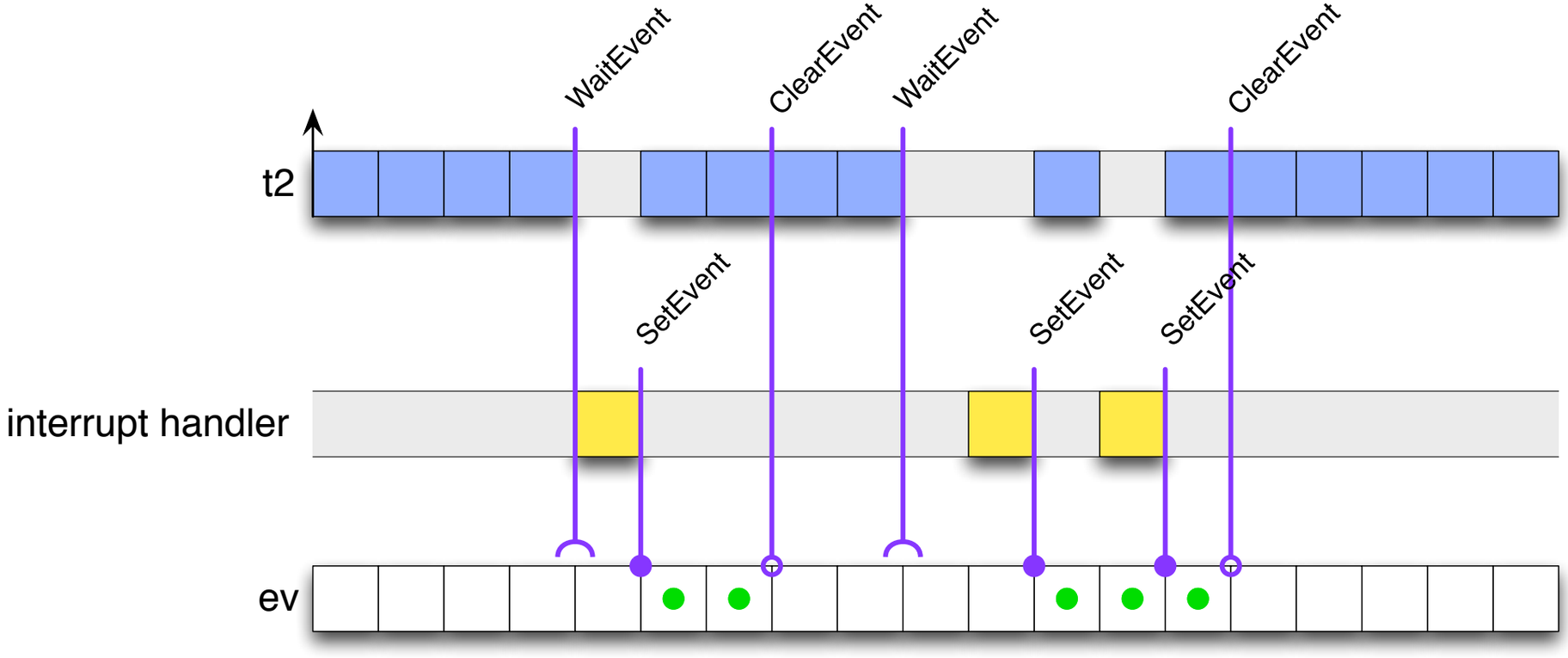
```
TASK(t1)
{
  while(1) {
    doSomeWork();
    SetEvent(t2,ev);
  }
  TerminateTask();
}
```

```
TASK(t2)
{
  while(1) {
    WaitEvent(ev);
    ClearEvent(ev);
    doSomeOtherWork();
  }
  TerminateTask();
}
```



Events may be lost

- If a second event is received while the previous one has not been taken into account, the second event is lost.
- Fixed priority scheduling solve the problem in monoprocessor system, not in multiprocessor system or if event is set by interrupt handler.



Events : implementation

- To implement this feature, an event corresponds to a binary mask: 0x01, 0x02, 0x04, ...
- An event vector is associated to 1 or more bytes. Each event is represented by one bit in this vector
- So each task owns:
  - a vector of the events set
  - a vector of the event it waits for
- Operation:
  - Event X signaling : ev_set |= mask_X;
  - Is event X arrived ? : ev_set & mask_X;
  - Wait for event X : ev_wait | mask_X;
  - Clear event X : ev_set &= ~mask_X;

Communication : shared data and messages

Shared data

- A set of global data shared among 2 or more tasks
  - simplistic and primitive approach
  - concurrency problem (we saw that in the previous section)
- Global variables are evil and should be avoided
  - System wide visibility:
    - implicit coupling of every pieces of software
    - how to be sure they are protected everywhere ?
- Hidden bugs difficult to find.

Messages (1)

- A better way to move data from one task to another
- Messages can be of **queued** kind:
  - A queue is a FIFO handling N messages
  - If a write occurs on a queue that is full, the message is lost
  - Of course, N can be 1
- Messages can be of blackboard kind:
  - A blackboard handles one message only
  - Writing a message to a blackboard overwrites the previous message
  - Reading a message from a blackboard gets the last written message
  - What if one reads a message from a blackboard that was never written ?

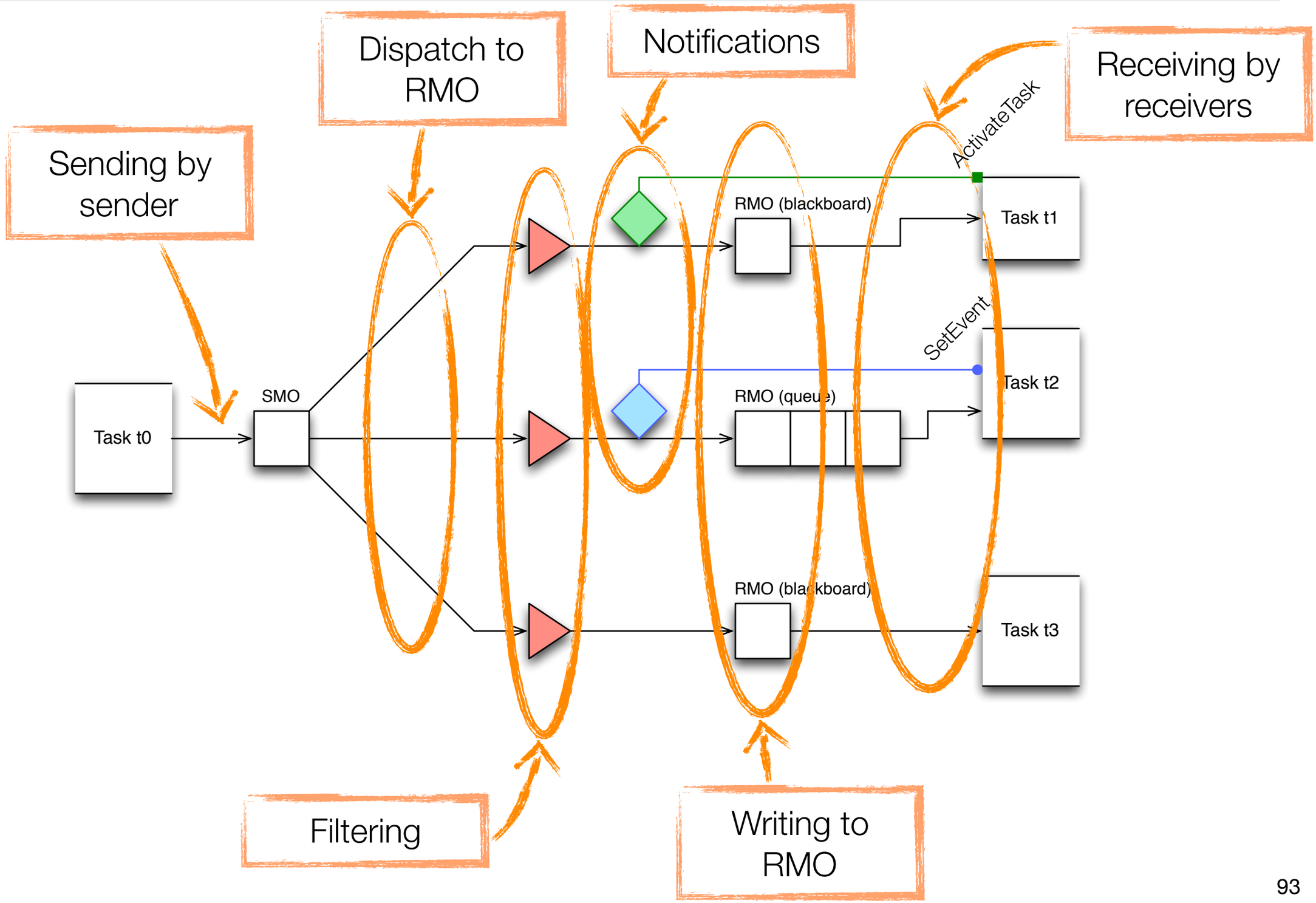
Messages (2)

- Concurrent accesses to a message is managed by the OS
  - **No need to use a mutex**, it is implicitly provides
- Synchronization may be included:
  - If a task try to read and empty queue, it is put in the WAITING state
  - If a message is received in an empty queue and a task is WAITING, then the task is put in the READY state
  - An event can be sent to a task when a message is received
  - A task may be activated when a message is received

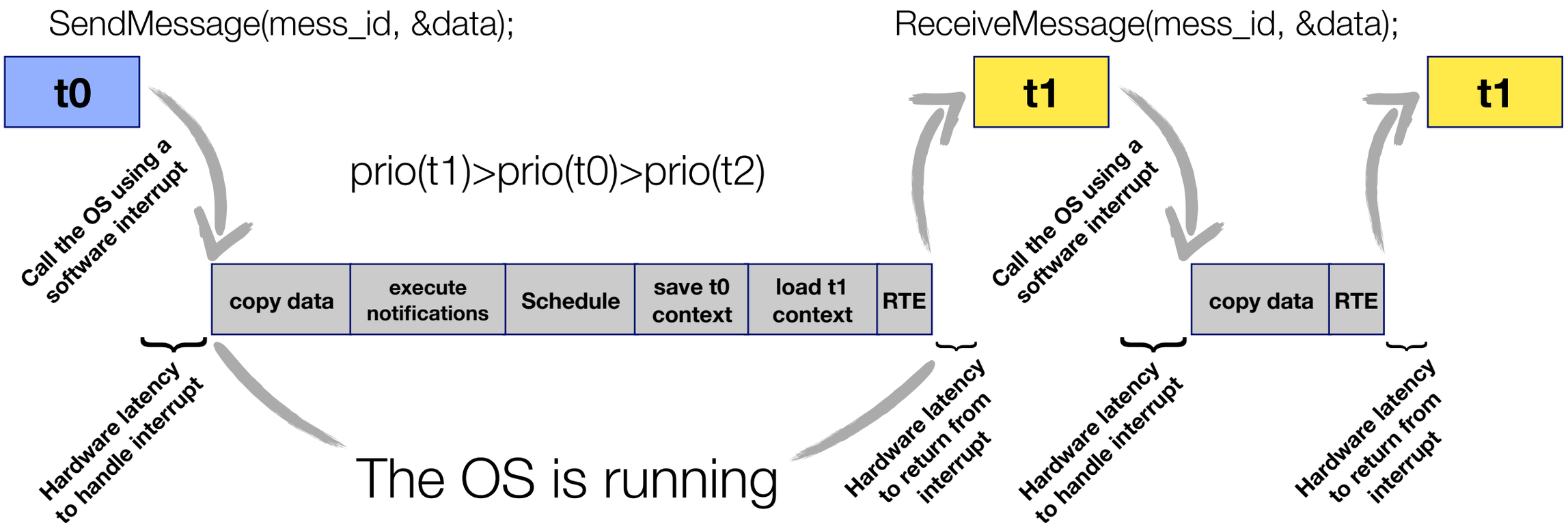
Example: OSEK/VDX messaging aka OSEK/COM

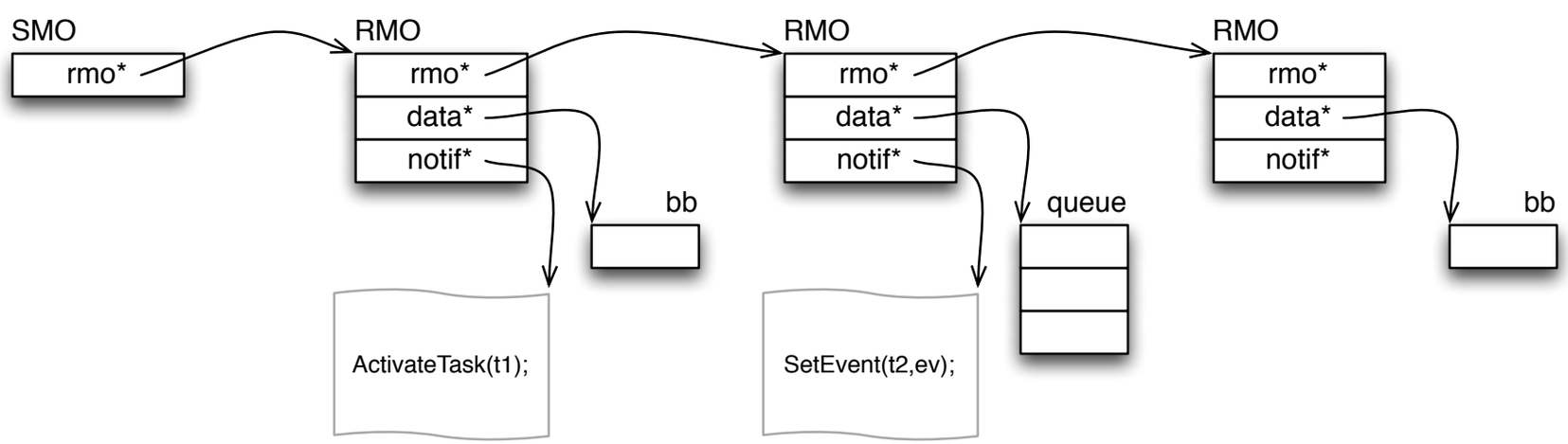
- OSEK/VDX messaging is design for communication within the same micro controller (internal communication) and for communication across a communication network (external communication).
  We will see the internal communication.
- Messages objects:
  - Sending message objects (SMO) to send messages
  - Receiving message objects (RMO), linked to sending message objects, to receive messages
- Filters:
  - Filters are associated to RMO and allows to **receive or discard** the message according to rules
- Notifications:
  - Notifications are associated to RMO and allows to **set an event** or **activate a task**.

Example: OSEK/VDX messaging



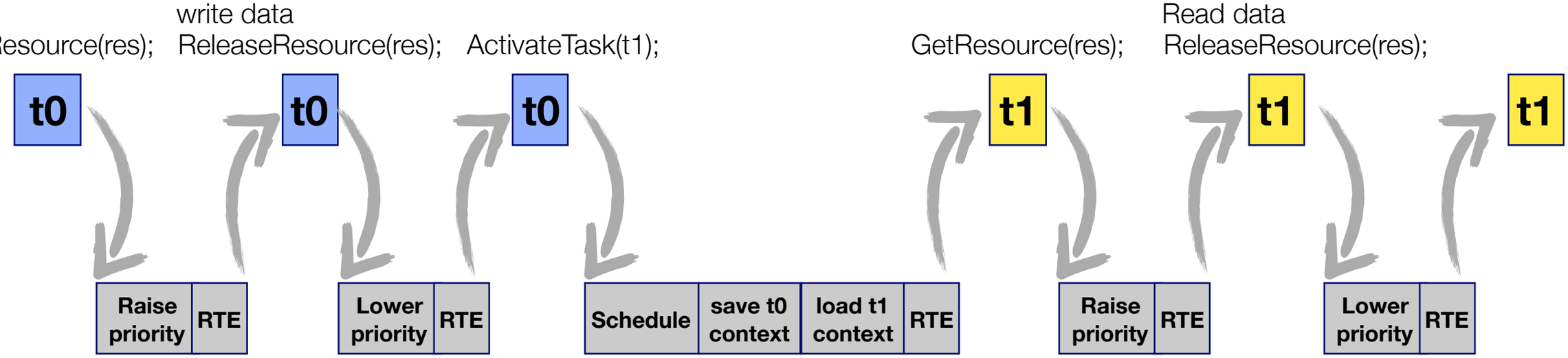
How it works


SendMessage(mess_id, &data);
ReceiveMessage(mess_id, &data);
t0
t1
t1
Call the OS using a software interrupt
prio(t1)>prio(t0)>prio(t2)
copy data
execute notifications
Schedule
save t0 context
load t1 context
RTE
copy data
RTE
Call the OS using a software interrupt
Hardware latency to handle interrupt
Hardware latency to return from interrupt
Hardware latency to handle interrupt
Hardware latency to return from interrupt
The OS is running


SMO
rmo*
RMO
rmo*
data*
notif*
bb
RMO
rmo*
data*
notif*
queue
RMO
rmo*
data*
notif*
bb
ActivateTask(t1);
SetEvent(t2,ev);

Comparison with a shared variable

prio(t1)>prio(t0)



Message filtering (1)

- It is not always useful to receive all the messages…
  - when 2 values in a row are the same
  - when the value is greater than a threshold. The message is used by a different task. For instance a control system where an alternate law is used according to the value
  - when the sender has a period *x* times smaller than the period of the receiver. In this case 1 message every *x* is received
- For these cases (an for others) OSEK/COM allows to set a filter for each receiving message.
- The message is delivered if the filter result is TRUE

Message filtering (2)

Filter	Sub-attribute	Result
ALWAYS		TRUE
NEVER		FALSE
MASKEDNEWEQUALSX	MASK, X	new & MASK == X
MASKEDNEWDIFFERSX	MASK, X	new & MASK != X
NEWISEQUAL		new == old
NEWISDIFFERENT		new != old
MASKEDNEWEQUALSMASKEDOLD	MASK	new & MASK == old & MASK
MASKEDNEWDIFFERSMASKEDOLD	MASK	new & MASK != old & MASK
NEWISWITHIN	MIN,MAX	(MIN <= new) && (new <= MAX)
NEWISOUTSIDE	MIN, MAX	(new < MIN) || (new > MAX)
NEWISGREATER		new > old
NEWISLESSOREQUAL		new <= old
NEWISLESS		new < old
NEWISGREATEROREQUAL		new >= old
ONEEVERYN	PERIOD, OFFSET	msg_number % PERIOD == OFFSET