Real-Time Operating Systems (RTOS) for sensors and actuators

B. Miramond
Polytech Nice Sophia

Some existing solutions

Zephyr™

Micrium® Embedded Software

VxWorks

THREADX®

RIOT

MICROEJ®

freeRTOS

TinyOS

APACHE SOFTWARE FOUNDATION mynewt

ARM® mbed™

Contiki

NUCLEUS

Windows IoT

android things

MONGOOSE OS

snappy

IoT OS and RTOS

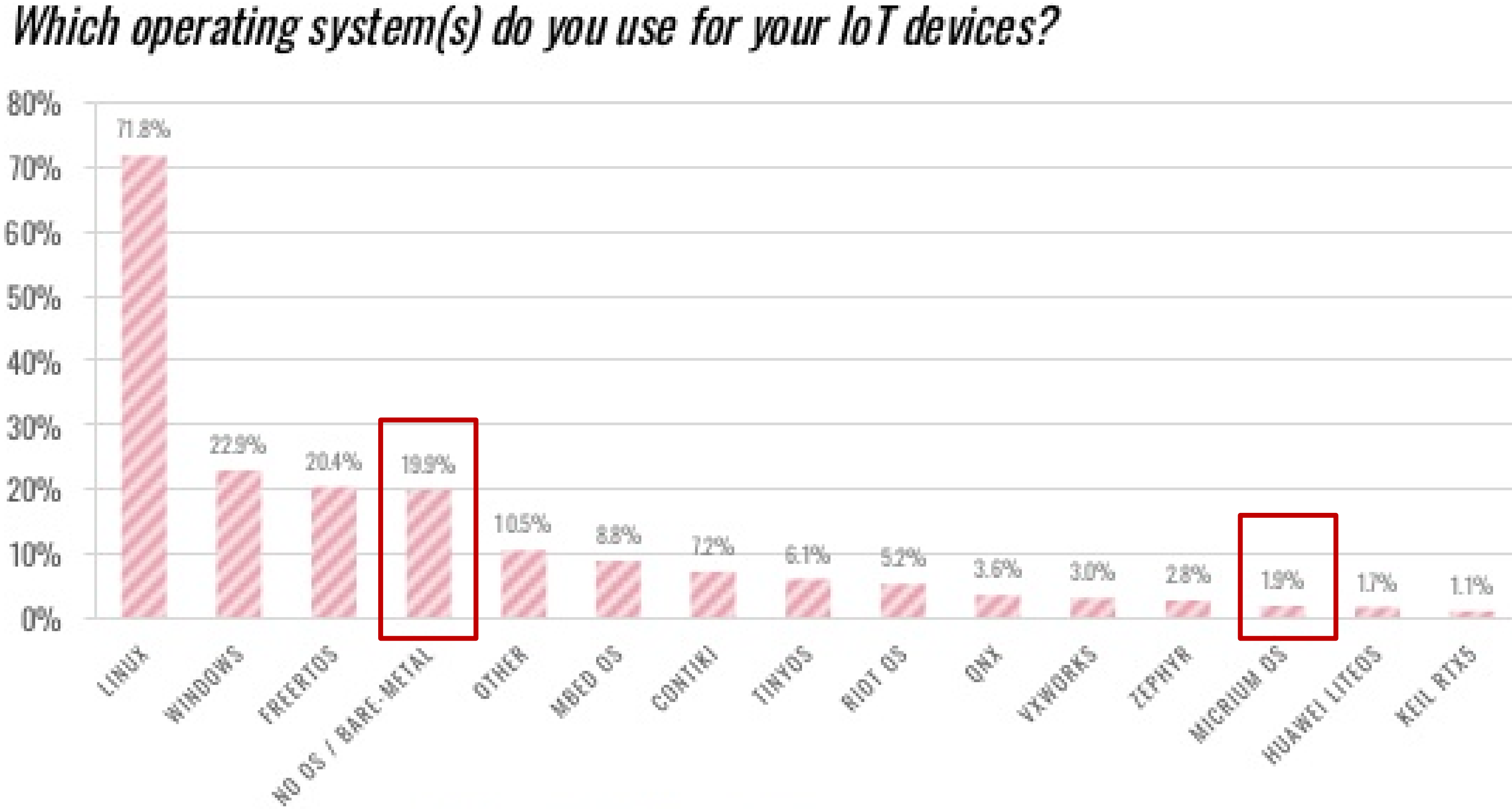
- **Open-source**

TinyOS, RIOT, Contiki, Mantis OS, Nano RK, LiteOS, FreeRTOS, Apache Mynewt, Zephyr OS, Ubuntu Core 16 (Snappy), ARM mbed, Yocto, Raspbian

- **Commercial**

Android Things, Windows 10 IoT, WindRiver VxWorks, **Micrium µC/OS**, Micro Digital SMX RTOS, MicroEJ OS, Express Logic ThreadX, TI RTOS, Freescale MQX, Mentor Graphics Nucleus RTOS, Green Hills Integrity, Particle

IoT Operating Systems



How to select one ?

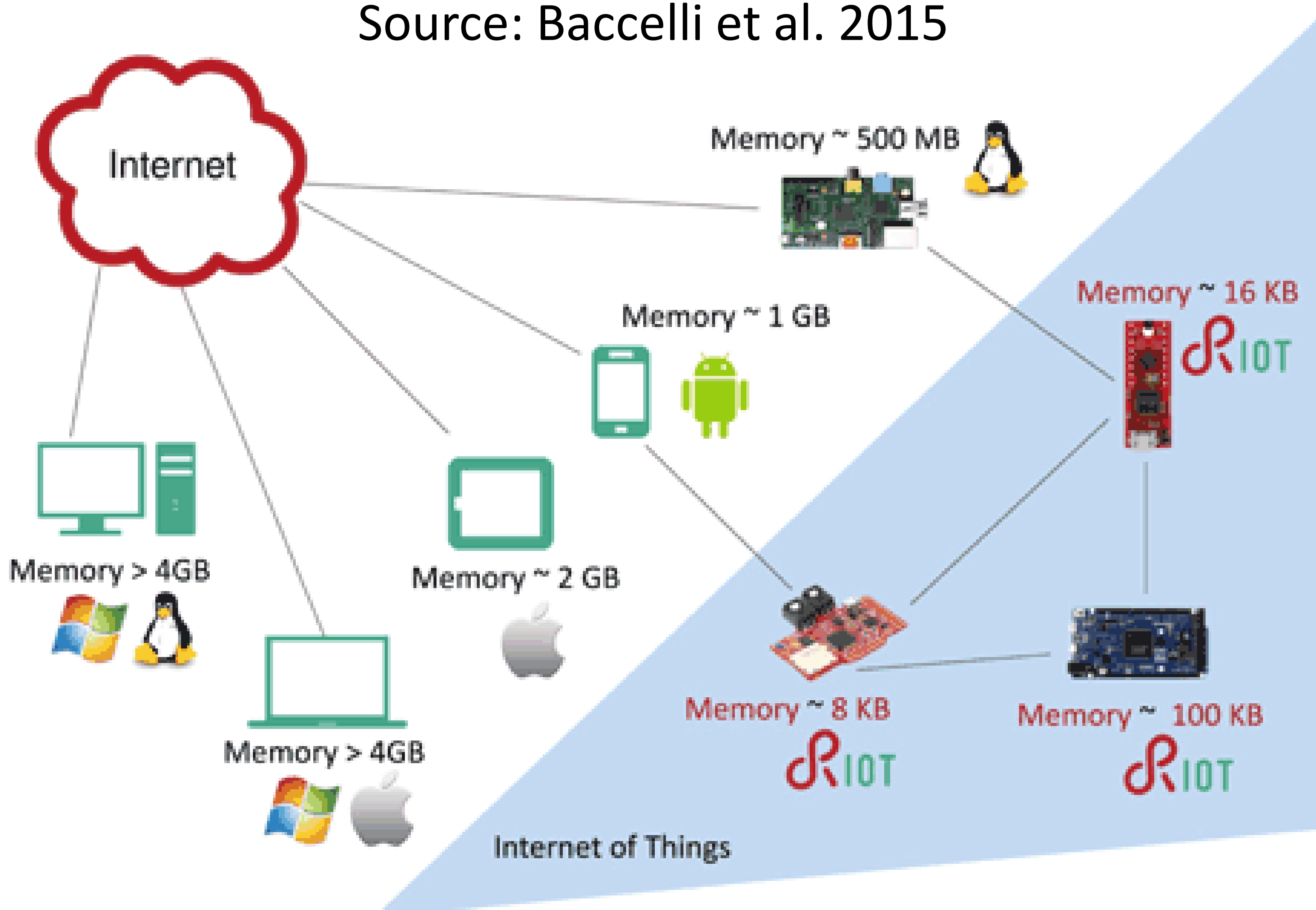
- Sensors are constrained systems that need:
  - Reduced memory footprint (ROM & RAM)
  - Low time-overhead (2 to 4 % of CPU time)
- But also
  - Reduced cost (royalties per unit)
  - Maintenance and support cost

How to select one ?

- **Footprint**: Since devices are constraint, we expect OS to have low memory, power and processing requirements. The overhead due to the OS should be minimal.
- **Portability**: OS isolates applications from the specifics of the hardware. Usually, OS is ported to different hardware platforms and interfaces to the board support package (BSP) in a standard way, such as using POSIX calls.
- **Modularity**: OS has a kernel core that's mandatory. All other functionality can be included as add-ons if so required by the application.
- **Connectivity**: OS supports different connectivity protocols, such as Ethernet, Wi-Fi, BLE, IEEE 802.15.4, and more.
- **Scalability**: OS must be scalable for any type of device. This means developers and integrators need to be familiar with only one OS for both nodes and gateways.
- **Reliability**: This is essential for mission-critical systems. Often devices are at remote locations and have to work for years without failure. Reliability also implies OS should fulfil certifications for certain applications.
- **Security**: OS has add-ons that bring security to the device by way of secure boot, SSL support, components and drivers for encryption.

Typical memory requirements

Source: Baccelli et al. 2015



Our use-case:
real-time kernel uC/OS-II

- uC/OS-II (Micro Controller Operating System)
  - 1992 (J. Labrosse)
  - 50 (8-64 bits)
  - Portable
  - ROMable (6K-24K bytes) + 1K bytes RAM
  - Scalable
  - Preemptive fixed priorities
  - Deterministic
  - With or without MMU

uc/OS-II vs. uc/OS-III

Feature	µC/OS-II	µC/OS-III
Release Date	1999–present	2009–present
Preemptive Multitasking	✓	✓
Maximum number of tasks	255	Unlimited
Number of tasks at each pri ority level	1	Unlimited
Round robin scheduling		✓

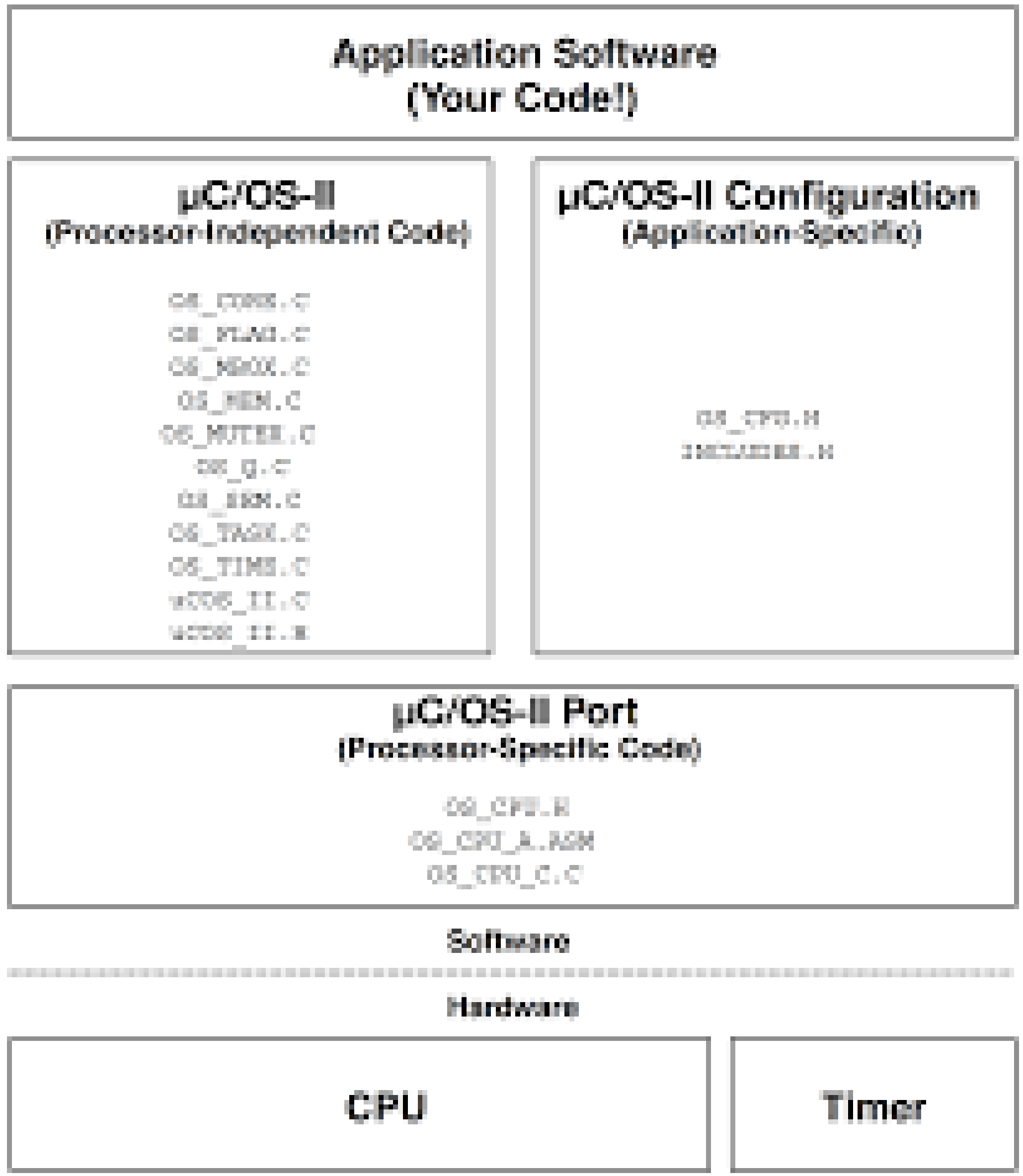
uC/OS-II Tasks

- Maximum 64 tasks
- Esclusive priorities
- No round robin (2 tasks don’t have the same priority)
- Tasks can communicate through ISR (Interrupt Service Routine)

Services of the real-time kernel

- The uC/OS kernel provides the following services:
  - Scheduling
  - Change of context
  - Management of synchronization modes (semaphores, mutex)
  - Management of the means of communication (mailbox, queue)
  - Delays and timout
- This gives a memory footprint of a few KB.

Structure of uC/OS-II



Memory footprint

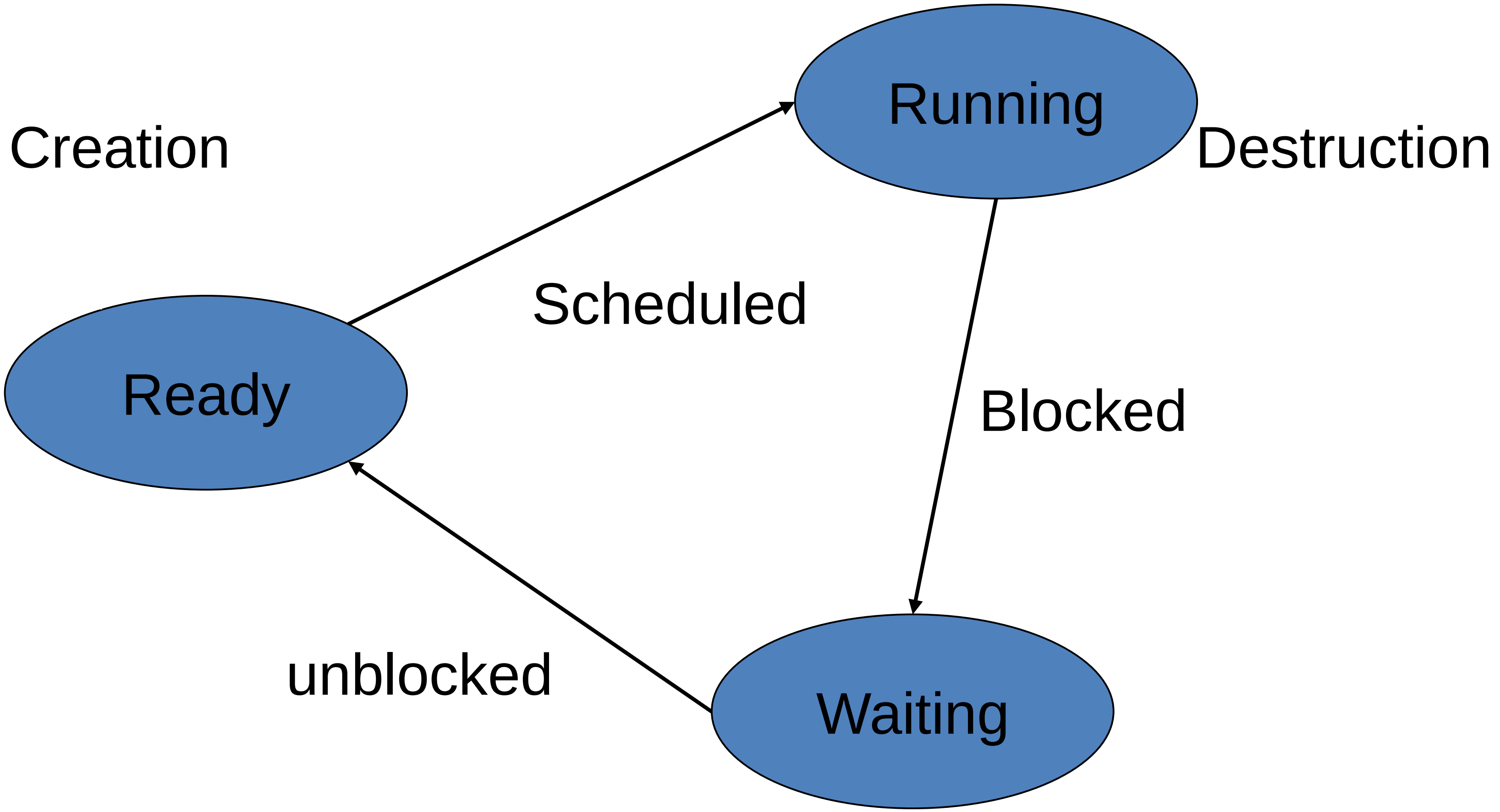
- The total size of the application with an RTOS depends on:
  - Size of the application code
  - Data (RAM) and instruction (ROM) space used by the kernel
  - SUM(Task Stack + TCB) // Task Control Block
  - SUM(ECB) / Event Control Block
  - ISR routines
- For systems with little memory, you have to pay attention to the evolution of the stack size :
  - Table and local structures with functions
  - Nested function calls
  - Number of function arguments

Tasks under ucOS-II

- The uC/OS-II Scheduler manages the following tasks:
  - User tasks
  - Plus 2 system tasks :
    - Idle Task (OS_LOWEST_PRIO)
    - Statistics Task (OS_LOWEST_PRIO - 1)
- Since the priorities are coded as an INT8U, and the priorities are not shared (no round robin), the number of possible tasks is

  64 - 2 = 62 user tasks.

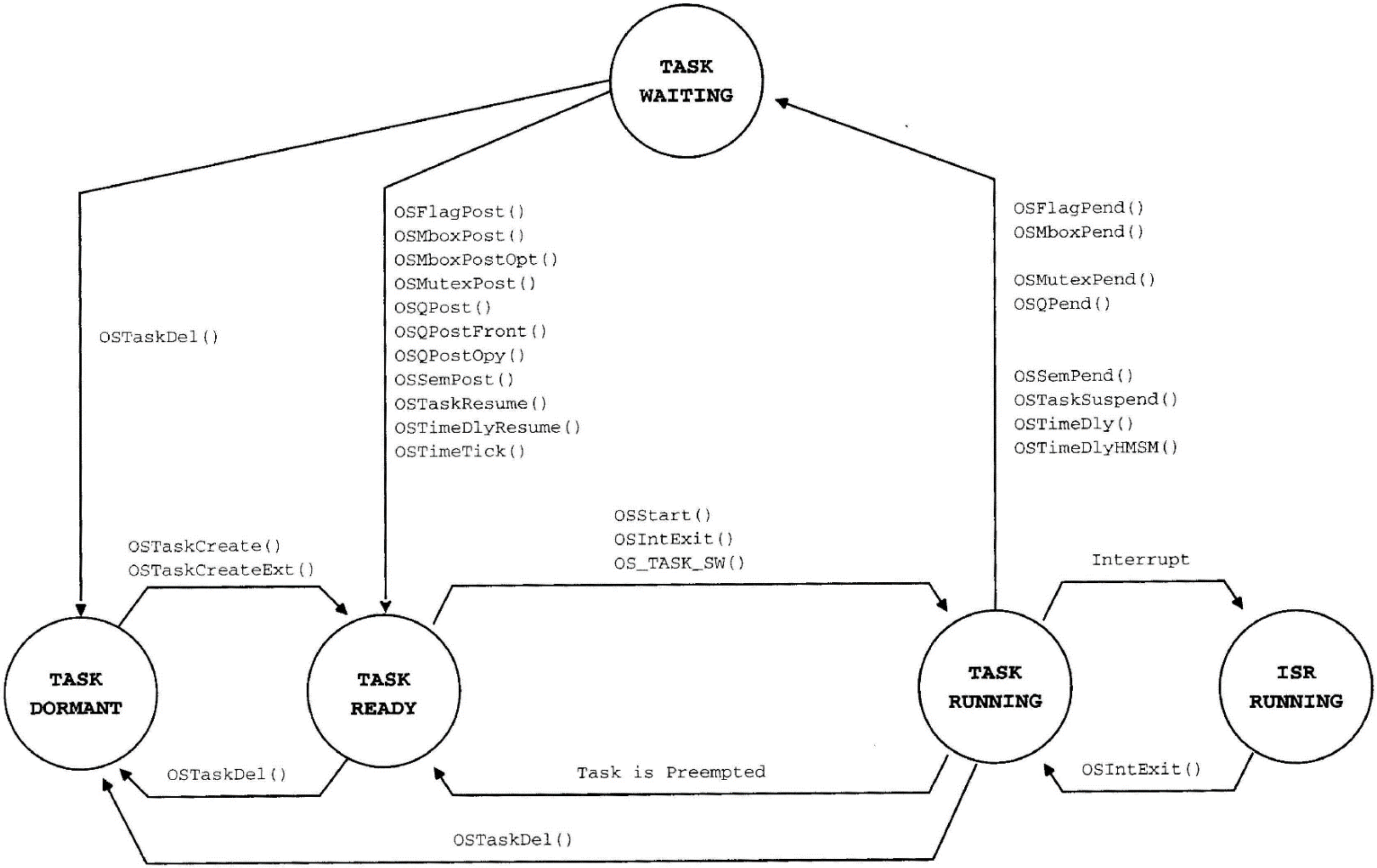
General state diagram



2 more states with uCOS-II

- **DORMANT** :
  - Before being created
  - After task destruction (OSTaskDel())
- **ISR** :
  - When an interrupt is triggered
  - On return, the current task can be pre-empted by a higher priority task (RDY state).
- WAITING :
  - Waiting for an event or timeout(OSTimeDly() or OS_X_Pend())
- READY :
  - Creation of tasks before multitasking is launched (OSStart())
  - Preempted
- RUNNING :
  - The only task using the processing resource (on MCU without multi-threading)

State diagram under ucOS



BUT WHAT IS A TASK ?

A task is managed through a data structure: TCB

The OS keeps the context of a task in a specific data structure called the **TCB**.
There is a change of context (save/restore) at each pre-emption or change of task:

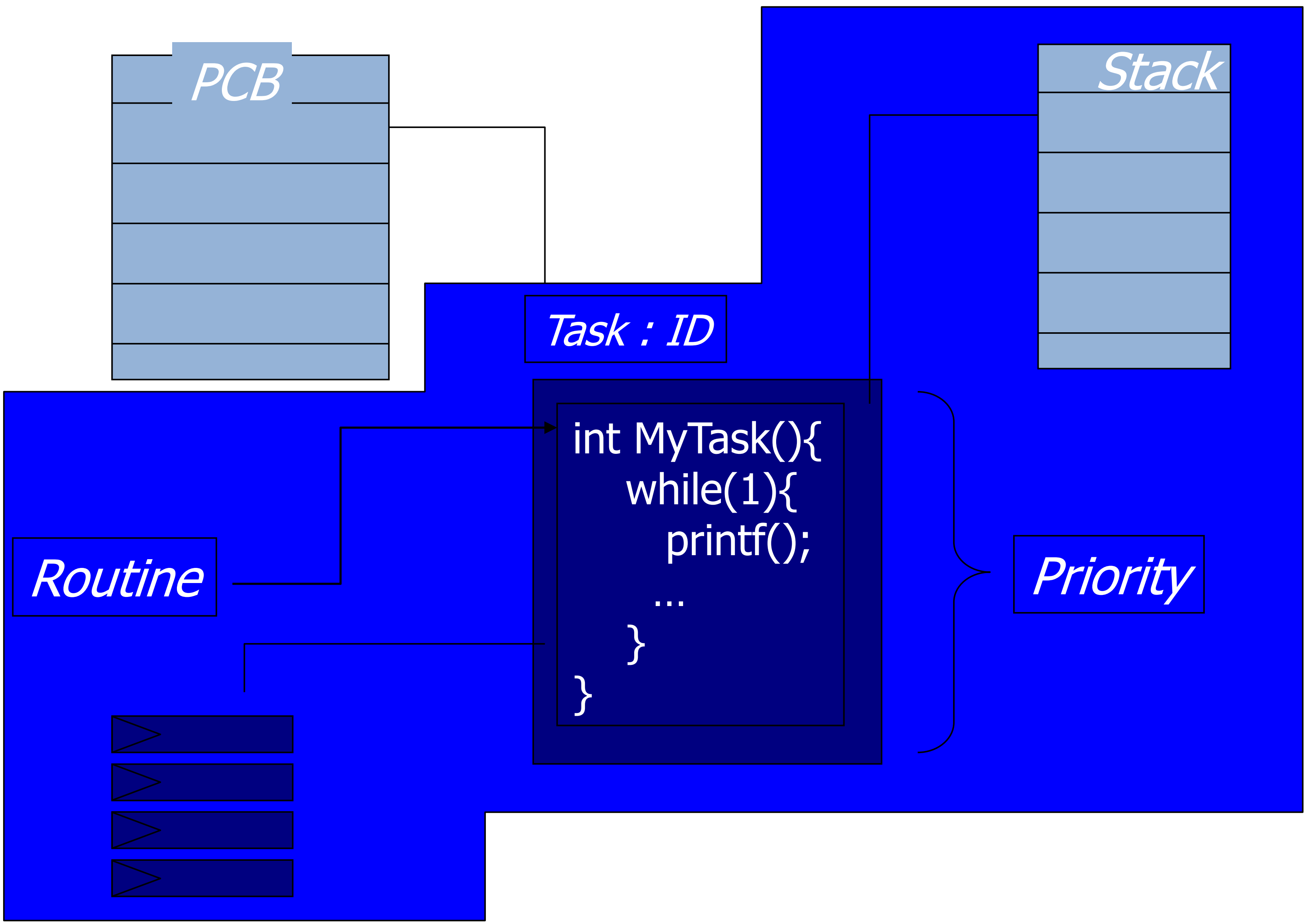
- Processor registers
- General registers: PC, SP, section registers
- Runtime stack
- Task status
- …

Task ID
State
CPU Context
Memory Context
Ressource
Scheduling data
User Information

More precisely

- The identifier of the process as it was assigned at its creation (an integer)
- One of the states of the process
- The value of the processor registers (PC, RI, SR...)
- The start and end addresses of the execution stack
- The open files,
- The synchronization services used
- The priority of the process, its queue...
- The CPU time used, the size of its stack, ...

Illustration of the task



uC-OS-II services API

- Task Management
- Time Management
- Timer Management
- Event Control Blocks
- Semaphore Management
- Mutual Exclusion Semaphores
- Event Flag Management
- Message Mailbox Management
- Message Queue Management
- Memory Management
- Porting µC/OS-II

API documentation :

- https://micrium.atlassian.net/wiki/spaces/osiidoc/overview

Fonctions to create a tasks: OSTaskCreate() and OSTaskCreateExt()

- Rule 1: Tasks must be created before launching the OSStart() function that launches the multi-tasking management mechanism.

- Rule 2: Ext version is an extended version used only for code analysis.
- The first one has 4 arguments, the second one 9.

Arguments of the task creation function

- void (*task)(void*pd): pointer on the task function
- Void * pdata : payload pointer used at the creation
- OS_STK *ptos : pointer on top of stack
- INT8U prio : priority of the tasl (limitations to 64 values)
- INT16U id : task ID (extension of the previous limitation to 64 values, else id = prio)
- OS_STK *pbos : Bottom-of-stack
- INT32U stk_size : stack size (for stack checking).
- void *pext : user data extension
- INT16U opt : uCOS_ii.h contains the list of options (OS_TASK_OPT_STK_CHK, OS_TASK_OPT_TSK_CLR, OS_TASK_OPT_SAVE_FP…). Each constante is a binary flag.

TCB

OS_STK = by default is 2 bytes long

```
******************************************************************************************************
******************************************************************************************************
*/

typedef st            b {
    OS_STK          *OSTCBStkPtr;         /* Pointer to current top of stack                          */

#if OS_TAS      EATE_EXT_EN > 0
    void            *OSTCBExtPtr;         /* Pointer to user definable data for TCB extension         */
    OS_STK          *OSTCBStkBottom;      /* Pointer to bottom of stack                               */
    INT32U           OSTCBStkSize;        /* Size of task stack (in number of stack elements)         */
    INT16U           OSTCBOpt;            /* Task options as passed by OSTaskCreateExt()              */
    INT16U           OSTCBId;             /* Task ID (0..65535)                                       */
#endif

    struct os_tcb *OSTCBNext;             /* Pointer to next     TCB in the TCB list                  */
    struct os_tcb *OSTCBPrev;             /* Pointer to previous TCB in the TCB list                  */

#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0) || (OS_SEM_EN > 0) || (OS_MUTEX_EN > 0)
    OS_EVENT        *OSTCBEventPtr;       /* Pointer to event control block                           */
#endif
```

TCB

```
#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    void           *OSTCBMsg;          /* Message received from OSMboxPost() or OSQPost()          */
#endif
```

Utilisé si l'option OSTCBOpt.OS_TASK_EN = 1

```
#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) &&                    0)
#if OS_TASK_DEL_EN > 0
    OS_FLAG_NODE   *OSTCBFlagNode;     /  Pointer to event flag node                                */
#endif
    OS_FLAGS        OSTCBFlagsRdy;     /* Event flags that made task ready to run                  */
#endif
```

wait(timeout) ou wait(event, timeout)

```
    INT16U          OSTCBDly;          /* Nbr ticks to delay task or, timeout waiting for event    */
    INT8U           OSTCBStat;         /* Task status                                              */
    INT8U           OSTCBPrio;         /* Task priority (0 == highest, 63 == lowest)               */

    INT8U           OSTCBX;            /* Bit position in group  corresponding to task priority (0..7) */
    INT8U           OSTCBY;            /* Index into ready table corresponding to task priority    */
    INT8U           OSTCBBitX;         /* Bit mask to access bit position in ready table           */
    INT8U           OSTCBBitY;         /* Bit mask to access bit position in ready group           */
```

Évite les calculs en-ligne, cf. chapitre suivant

```
#if OS_TASK_DEL_EN > 0
    BOOLEAN         OSTCBDelReq;       /* Indicates whether a task needs to delete itself          */
#endif
} OS_TCB;
```

OSTaskDel

- OSTaskDel(OS_PRIO_SELF);
- Auto Destruction of a task when it has finished processing so that it does not take up memory unnecessarily.

EXECUTION AND SCHEDULING OF TASKS ON AN RTOS

2 types of task

*Void Run-to-completion_Task (void *pdata){*

Initialization

Create another task

Create other kernel objects

Self-destruction (end of the task)

}

Void Endless-loop (void =pdata) {

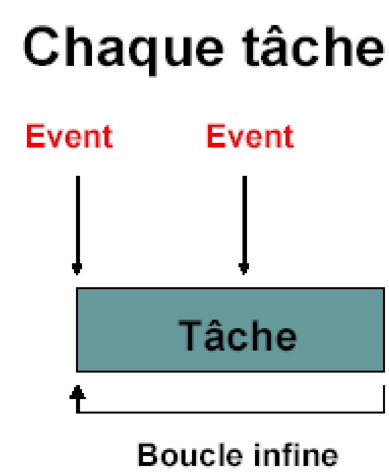
***While(1)** {*

Loop body

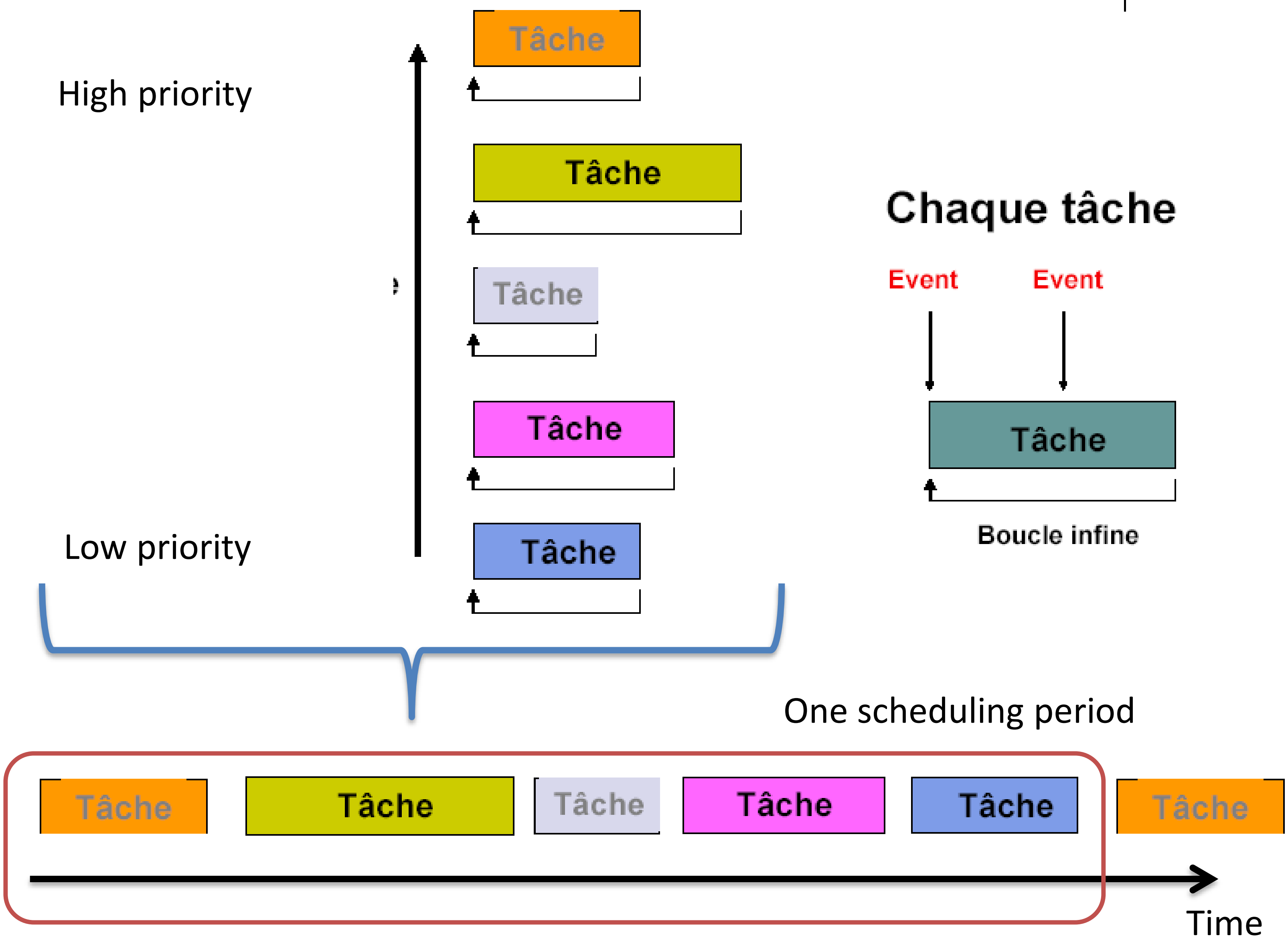
Blocking call

}

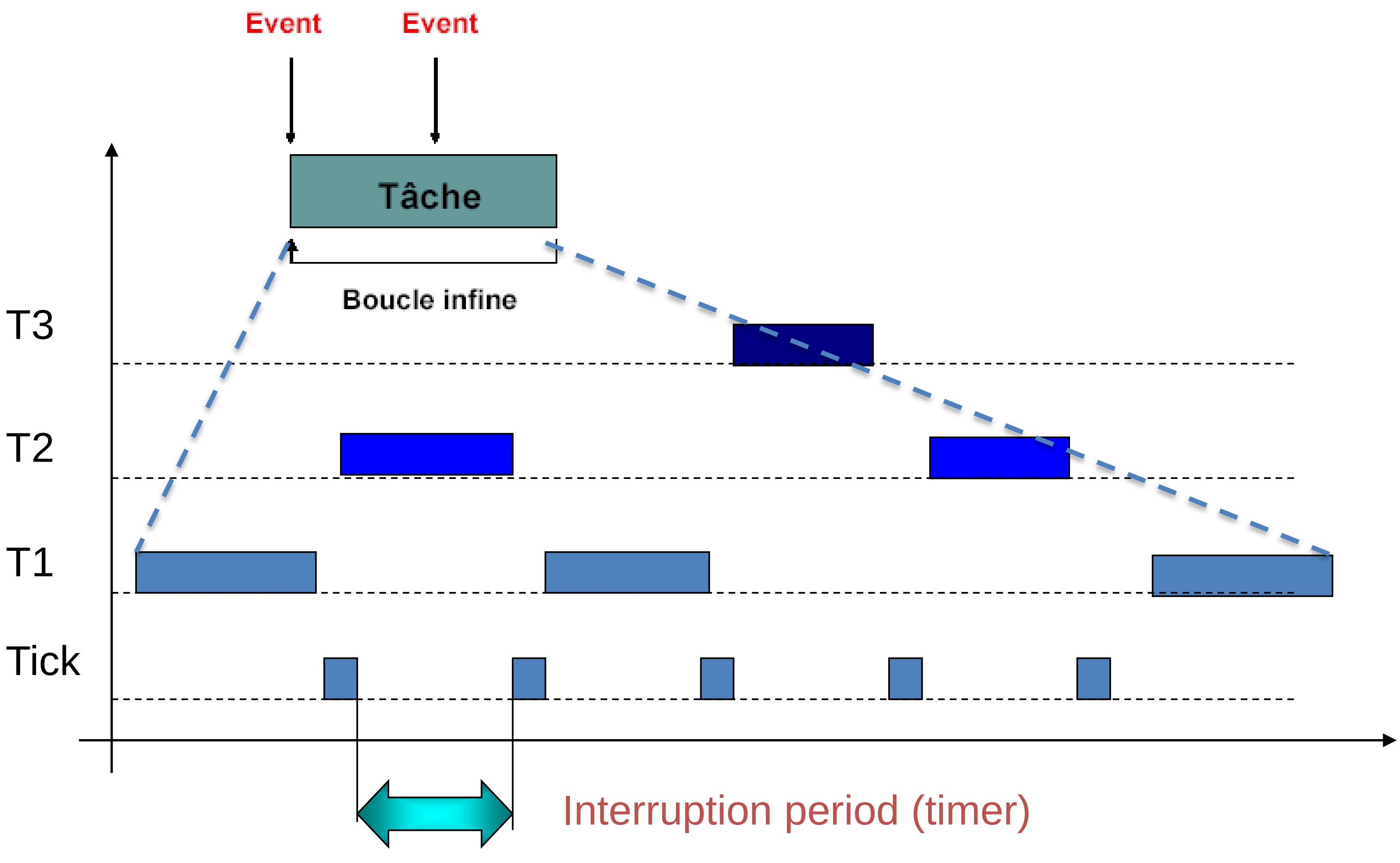
}



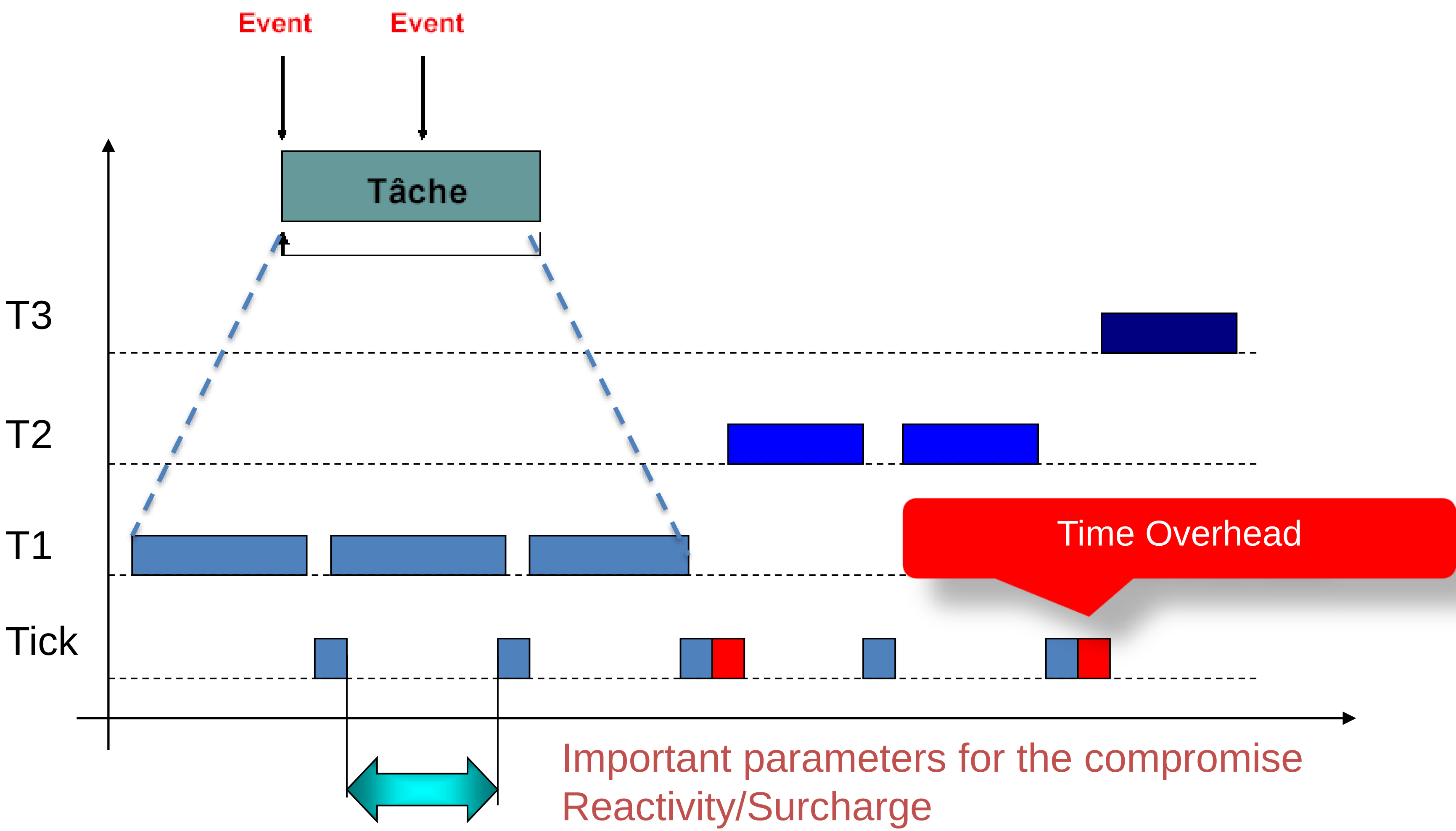
Priority and scheduling



Preemptive OS

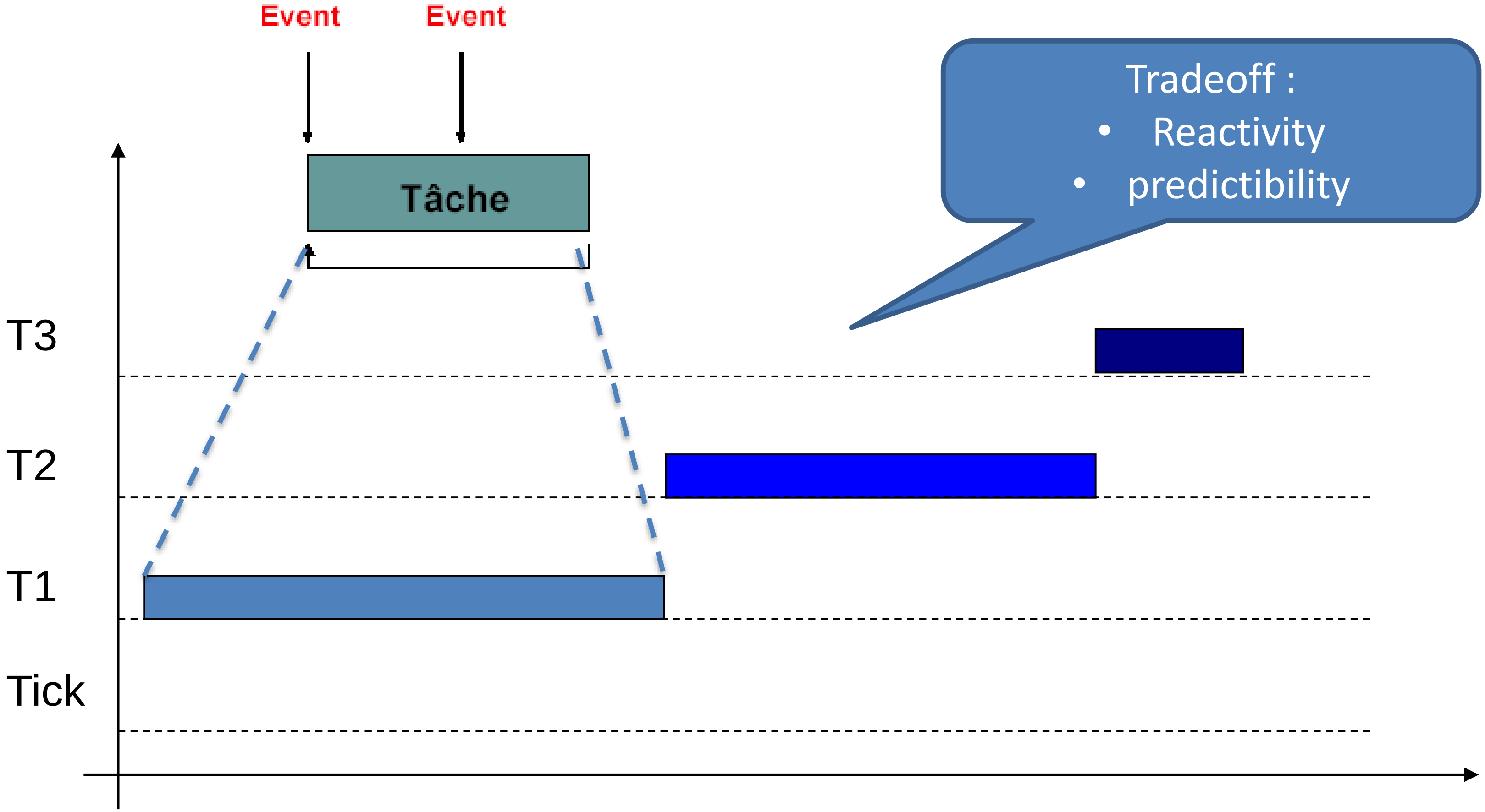


Preemptive OS with fixed-priorities



Important parameters for the compromise
Reactivity/Surcharge

Non-preemptive OS



Reentry functions

Any function called by a task
is executed in the context of the task !

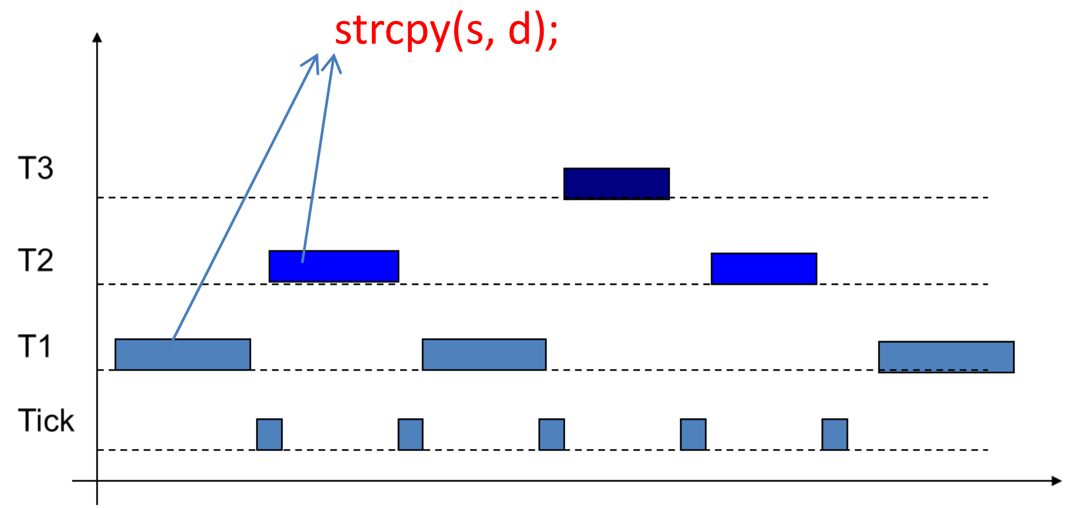
Two types of implementations are possible:

- Reentry functions :

```
void strcpy(char *dest, char *src){
    while(*dest++ = *src++);
    *dest = NULL;
}
```

- Non-reentry functions:

```
int temp;
void swap(int *x, int *y){
temp = *x;
*x = *y;
*y = temps;
}
```



Volatile and static keywords

- **Volatile**
  - Used in the case of variables whose value can change spontaneously:
    - without processor action, memory mapped devices
    - By another task in case of multithreaded software
  - This prefix tells the compiler to avoid optimizations that generate a systematic memory read instruction.
    - More often used in embedded programming
- **Static**
  - On a variable, keeps the value of the local variable (allocation out of stack)
  - On a function limits the definition of the symbol inside the object (file)

Volatile, example 3: Preemption and shared variables

- The compiler cannot have knowledge of context switches
- Whereas the shared variables can be changed at any time by another function

```
int cntr;
void task1(){
        cntr = 0;
        while(cntr == 0){
                // do something
                sleep(1);
        }
                // do something else
}
```

```
void Task2(){
        …
        cntr++;
        sleep(10);
        …
}
```

That's why it's safer to declare shared variables as **volatile**

Simple types

- Integers depend on the target architecture, so they are redefined in the specific part: in OS_CPU.h
- On the Nios MCU,
  - typedef unsigned char BOOLEAN;
  - typedef unsigned char INT8U; /* Unsigned 8 bit quantity */
  - typedef signed char INT8S; /* Signed 8 bit quantity*/
  - typedef unsigned int INT16U; /* Unsigned 16 bit quantity */
  - typedef signed int INT16S; /* Signed 16 bit quantity*/
  - typedef unsigned long INT32U; /* Unsigned 32 bit quantity */
  - typedef signed long INT32S; /* Signed 32 bit quantity */
  - typedef float FP32; /* Single precision floating point */
  - typedef double FP64; /* Double precision floating point */

Delays in a RTOS

- uCOS works in real time thanks to the notion of time.
- This time is given by a source called Clock Tick which is a periodic interruption (ISR).
- The period depends on the application.
- Rule 3: ISR = overhead with respect to the system
  - too small and priority tasks will wait longer
  - too big and the extra cost will become a handicap

OSTimeDly()

- A task can itself be put on hold / waiting state
- Calling this function causes a context shift to the next priority task
- Parameter = number of tick between 0 and 65.535
- The task will only be executed again after the time has elapsed and only if it has the highest priority.
- OSTimeDlyHMSM() does the same by taking Hours, Minutes, Seconds and Milliseconds as parameters.
- Rule 4 : Never call this function after disabling SRI!!

- What is the difference with usleep function ?

OSStart()

- It initializes variables and data structures.
- It creates the Idle() task that is always ready to run and has the lowest priority (OS_LOWEST_PRIO = 63).
- If the OS_TASK_STAT_EN and OS_TASK_CREATE_EXT tags are set to 1, it also creates a statistics task with OS_LOWEST_PRIO -1 priority.

OSStart()

- Launches multi-tasking management
- Rule 1 (reminder): You must therefore have already created at least one task
- OSStart finds the TCB of the HPT
- It calls OSStartHighRdy() which is described in OS_CPU.asm depending on the target processor.

Summary

General scheme of a program under uCOS

```
// 1 – Static allocation of memory stacks
OS_STK TaskStartStk[TASK_STK_SIZE];

// 2 – declaration of synchro/com objects
OS_EVENT *mbox, mq, mutex;

void main (void){

// 3 - Create at least one task
OSTaskCreate(   TaskStart,
                (void)*0,
                &TaskStartStk[TASK_STK_SIZE -1],
                TASK_START_PRIO);
OSStart(); // start multi-tasking
}

void TaskStart(void *pdata){
        // 4 – Create other tasks
}
```

Inter-Process Communication : IPC

Communication between tasks
i) Shared Ressources

- The easiest way to make 2 tasks communicate is to use a shared memory area.
- Especially when they run in a single address space.
- However, this requires ensuring exclusive access by each task at a given time.
- This translates into several methods:
  - a) Stop interrupts, // CRITICAL SECTION
  - b) Test-And-Set operation
  - c) Stop the scheduler,
  - d) Use a semaphore,

a) Critical Sections

- To prevent other tasks or ISRs from modifying a critical section, it is necessary to interrupt the interruptions.
- interrupt downtime = important parameter of an RTOS (interrupt latency)
- Depends on the processor and the application
- In uCOS, 2 macros are used from OS_CPU.H :
  - OS_ENTER_CRITICAL()
  - // critical section
  - OS_EXIT_CRITICAL()

Interrupt latency

- One of the most important characteristics of an RTOS = the time during which interruptions are stopped (critical sections).
- IRQs are used in an on-board environment as a means of initiating user code triggered by an external asynchronous sensor (ABS braking, sensor sample…).

Interruptions

- Interruptions are asynchronous events triggered by hardware mechanisms.
- When the CPU receives an interrupt, it saves the context of the current task and connects to the routine corresponding to the IRQ number in its interrupt vector.
- At the end of the routine, the CPU returns to :
  - The highest priority task (preemptive mode)
  - The interrupted task (non-preemptive mode)

b) Operation TAS

- This operation is used when the system does not have a RT kernel.
- Principle test and set in a single clock cycle:
  - Test of the value of a global variable
  - If Val = 0
    - The function has access to the resource
    - It sets the variable to 1 // Test and Set
- Some processors implement this service in hardware in their instruction set.

c) Stop the scheduler

- Rough solution that does not interrupt interruptions, only multi-tasking.
- For example: :
  - OSSchedLock()
  - Access to share data
  - OSSchedUnlock()

d) Semaphores

- Invented in the 1960s by Edgser Dijkstra
- Binary semaphores // Mutex
- Counting semaphores
- Three types of operations :
  - Create (initialize)
  - Wait (Pend) (sem>0)? sem-- : wait();
  - Signal (Post) (sem==0)?sem ++: notify();
- The task that launches (notify) is either
  - Highest priority (uC/OS)
  - The one who requested it first (FIFO)

Assessment of shared resource access

- For the access to a shared area, the semaphore is the least risky solution.
- If other solutions are misused, the consequences can be much more serious.
- However, for a simple access to a 32-bit variable, stopping interrupts will be less expensive than using a semaphore without changing the interrupt latency!

Deadlock

- Deadlock occurs when 2 tasks are waiting for access to resources taken by the other one (see course on scheduling).
- To avoid these situations the tasks must :
  - Acquire resources in the same order,
  - Release resources in reverse order.
  - RTOS kernels also allow you to specify a Timeout on waiting for semaphores.
    - These calls return a different error code to inform the task that the wait was not successful.

ii) Sending messages

- Message Mailbox
  - Sending a pointer to a mailbox
  - Only one letter at a time !
- Message Queue
  - Sending several messages (mailbox queue[] ;)
  - Reading in FIFO mode
  - A list of pending processes is built by the scheduler.

State diagram under ucOS