Chapter 2, Section 2.1 to 2.3: Architecture and Characteristics of a RTOS kernel (μC/OS-II)

1. *Introduction*
2. *Interrupt Management*
3. *Task Management*
4. Time Management
5. Event Management
6. Memory Management
7. Porting μCOS-II

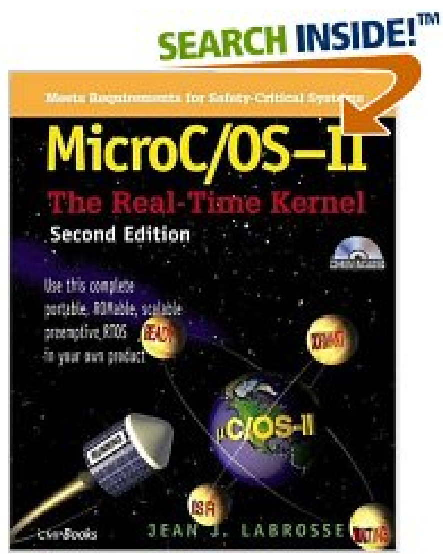
ÉCOLE POLYTECHNIQUE MONTRÉAL

1. Introduction

- **μC/OS-II is one of the Micrium products (http://www.micrium.com/)**
- **μC/OS-II is a real-time multitasking kernel, scalable and easily adaptable to different processors.**
- **It can be programmed in a ROM or a Flash memory.**
- **Also, latencies of interrupt and switching context are deterministic and low (O(μs)).**

1. Introduction

- μC/OS-II is second after VxWorks (percentage at the level of the RTOS market).
- μC/OS-II offers one the most popular books for the embedded systems topic.



ÉCOLE POLYTECHNIQUE MONTRÉAL

1. Introduction

- µC/OS-II source and object code can be used by accredited Colleges and Universities without requiring a license, as long as there is no commercial application involved. In other words, no licensing is required if µC/OS-II and/or µC/TCP-IP are used for educational use.
- You need to obtain an 'Executable Distribution License' to embed µC/OS-II in a product that is sold with the intent to make a profit or if the product is not used for education or 'peaceful' research.

1. Introduction

- **See also:**
  - **the list of products:** **http://www.micrium.com/products/uc-products.html**
  - **the partial list of client:** **http://www.micrium.com/customers/list.html**
  - **the list of porting:** **http://www.micrium.com/products/rtos/kernel/ports.html**

2. Interrupts management

- **Your Interrupt Service Routine (ISR)...**

  - Saves all the CPU registers;

  Calls OSIntEnter() or increment OSIntNesting directly;

  If (OSIntNesting == 1)

  OSTCBCur -> OSTCBStkPtr = SP; //details later

  Clears interrupting device;

  Re-enables interrupts (optional)

  Executes user code to service interrupt;

  Calls OSIntExit();

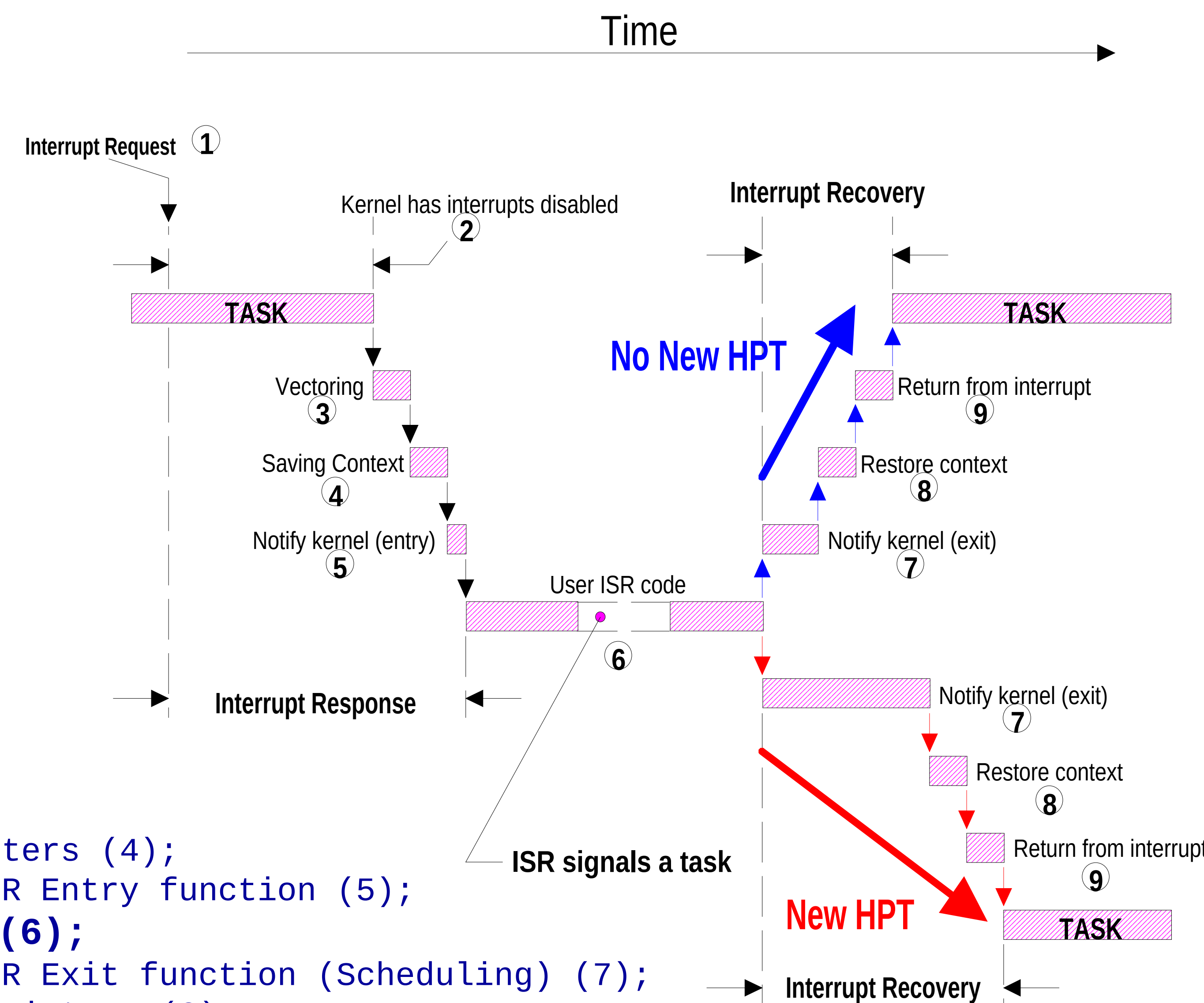
  Restores all CPU registers

  Executes a return from interrupt instruction;

2 Interruptions management

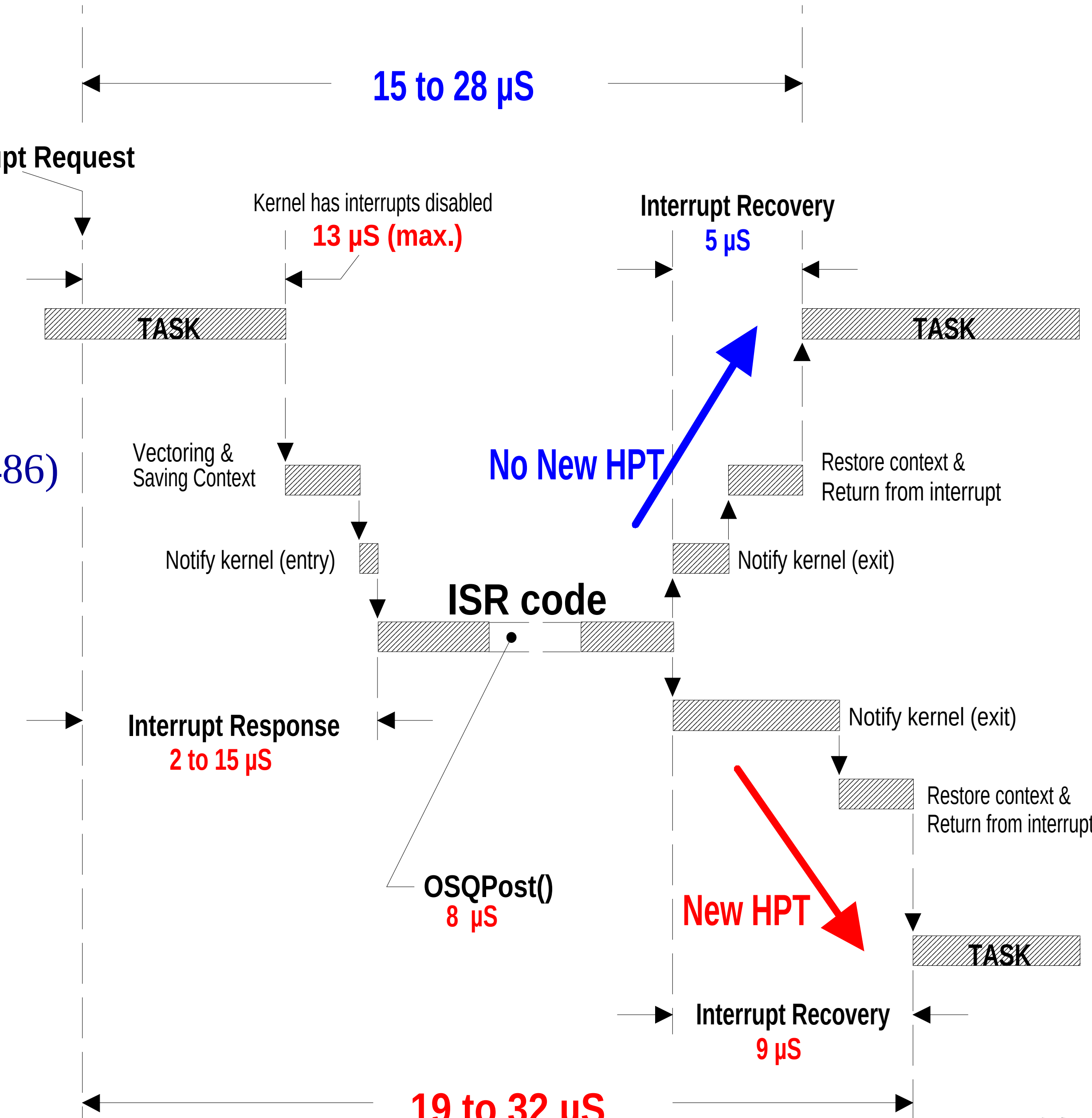
- **OSIntEnter() :**
  - Disables interrupts to gain exclusive access to the global variable OSIntNesting (current number of interrupt nested)
  - Increments OSIntNesting
  - Enables interrupts

- **OSIntExit() :**
  - Disables interrupts
  - Decrements OSIntNesting
  - If OSIntNesting == 0, determines HPT (Highest Priority Task) as the next task to be executed
  - Performs a context switch to resume HPT
  - Enables interrupts;




```
ISR (3):
    Save CPU Registers (4);
    Call Kernel ISR Entry function (5);
    Process ISR (6);
    Call Kernel ISR Exit function (Scheduling) (7);
    Restore CPU Registers (8);
    Return from Interrupt (9);
```

Servicing Interrupts
(µC/OS-II on 66 MHz 80486)



ÉCOLE POLYTECHNIQUE MONTRÉAL

3. Task Management

Overview

Chap2, Sections 2.1 à 2.3, page 10

3.1 A task and its data structure

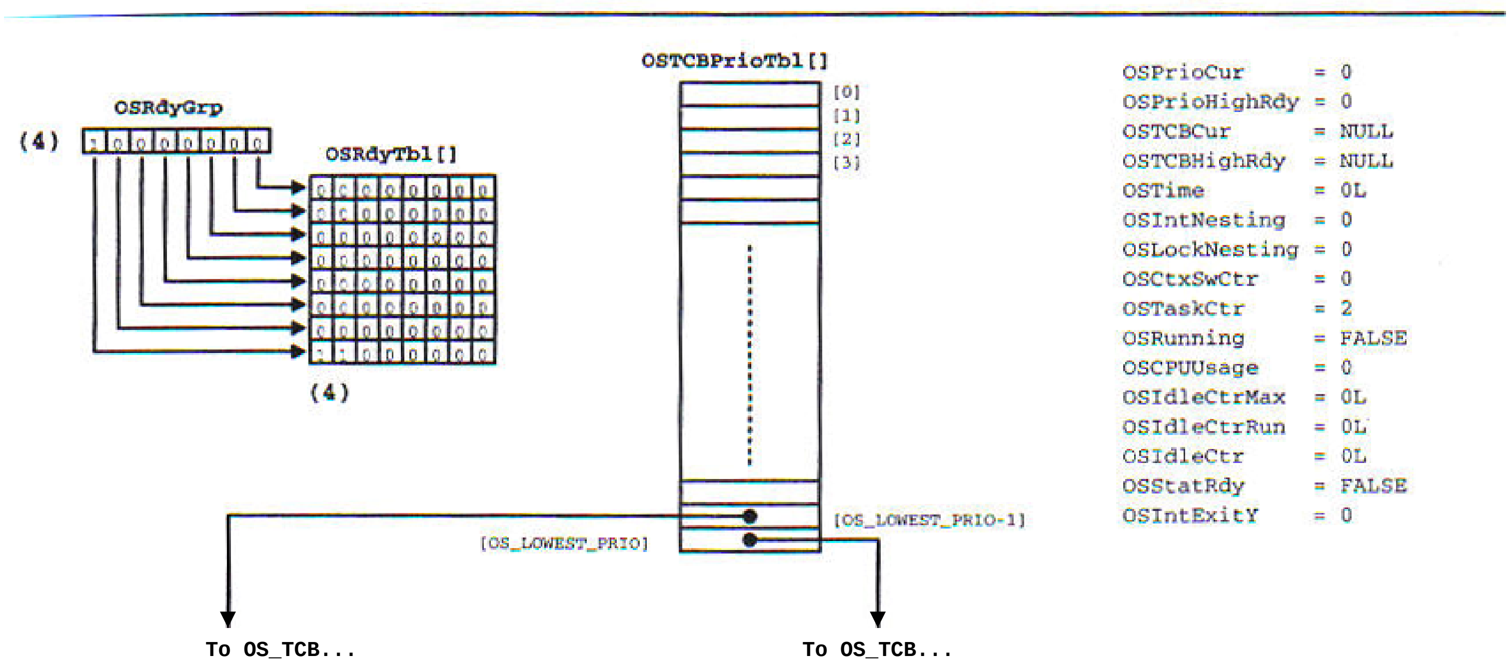
- **A µC/OS-II task is built with an infinite loop**
  - i.e. *for(;;)* or *while(1)*
  - Similar to C subroutines
  - The return type must always be declared *void*
  - The task can delete itself upon completion

3.1 A task and its data structure

- A task is composed of a TCB (Task Control Block) and a stack
- The TCB contains the task parameters (see next slide)
- The stack contains:
  - functions called by your task;
  - local variables that will be allocated by all functions called by your task,
  - nested interrupts
  - CPU register saved during the context switch

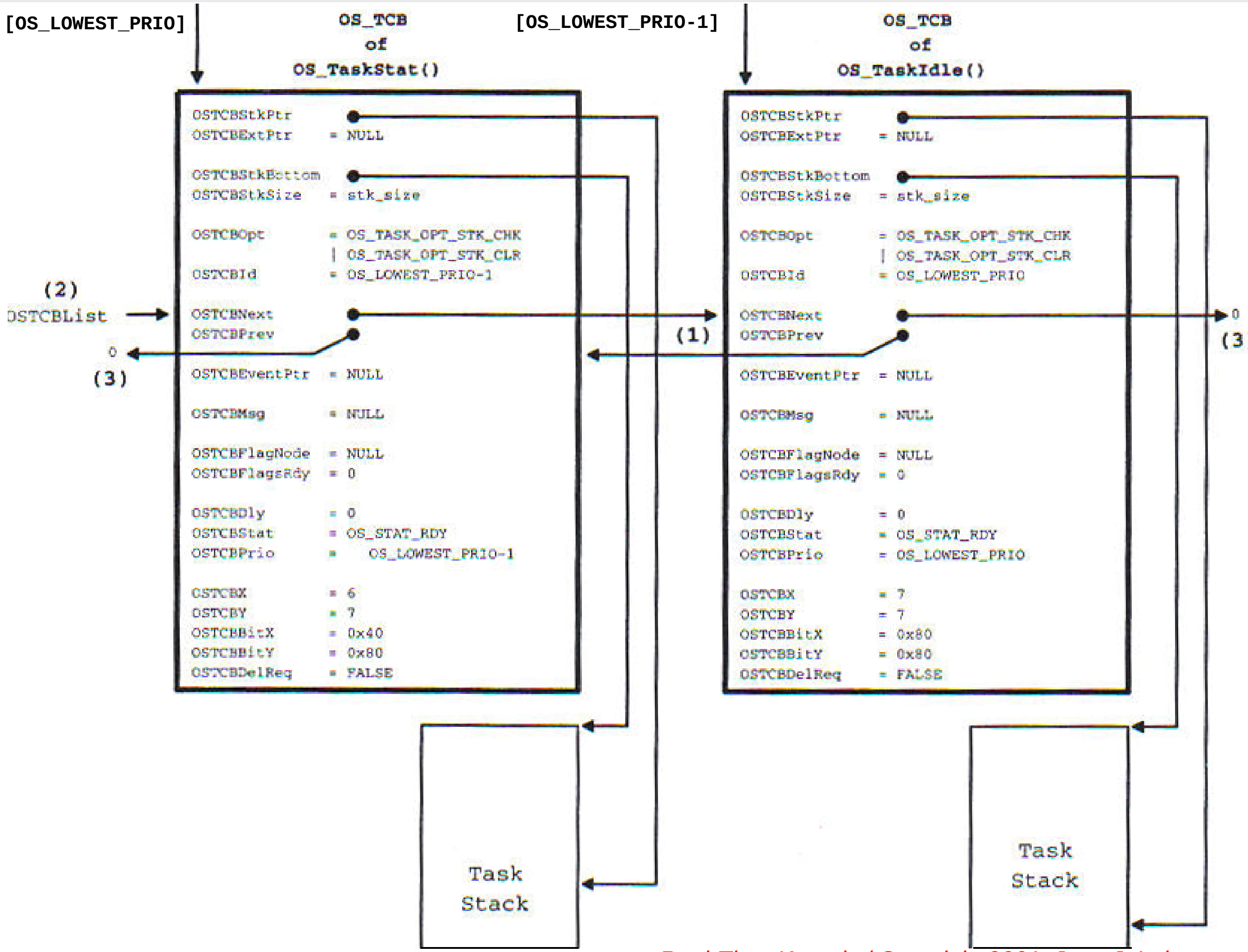
3.1 A task and its data structure

Variables and data structures after calling `OSInit()`.



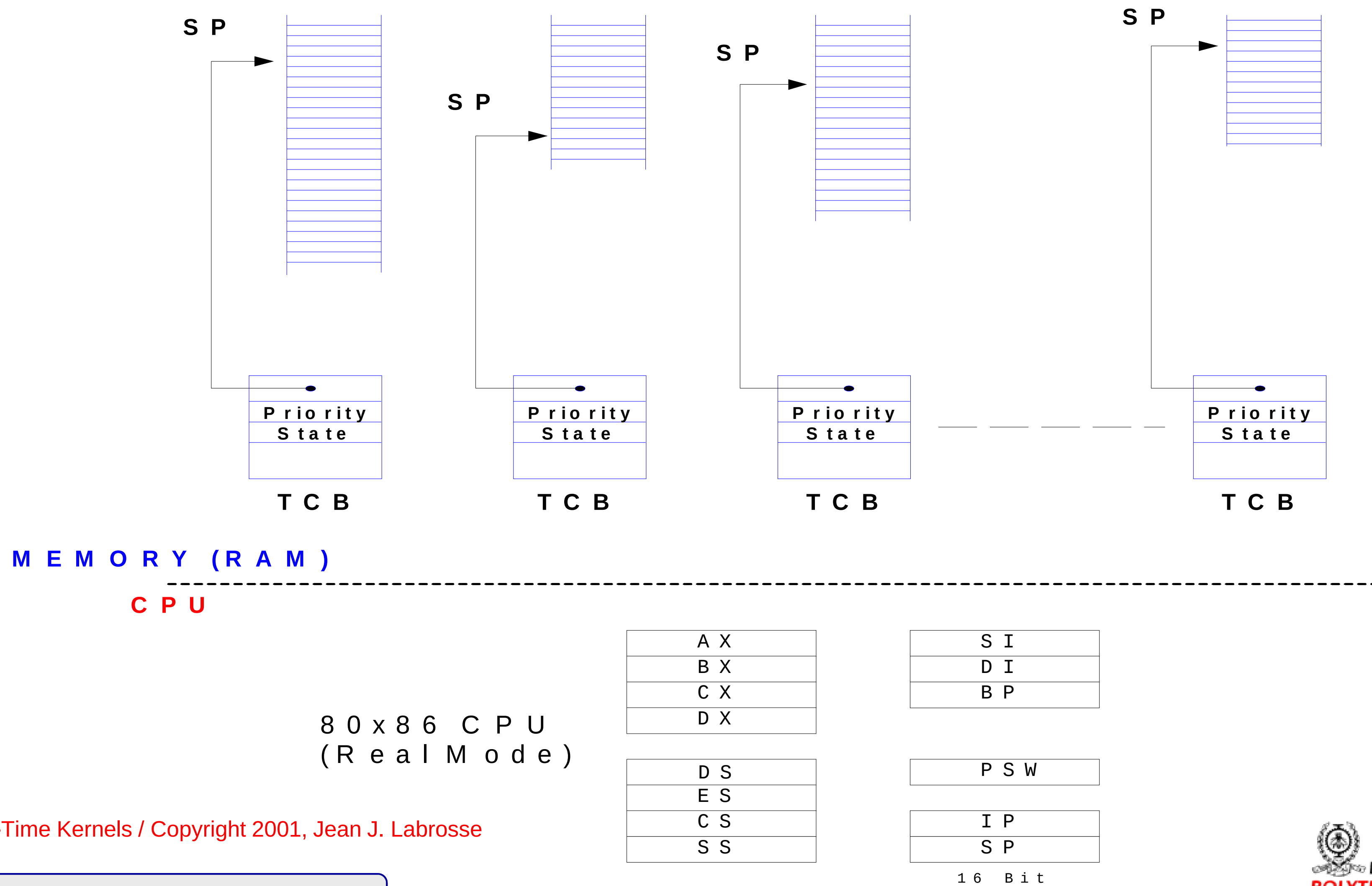
ÉCOLE POLYTECHNIQUE MONTRÉAL

3.1 A task and its data structure



ÉCOLE POLYTECHNIQUE MONTRÉAL

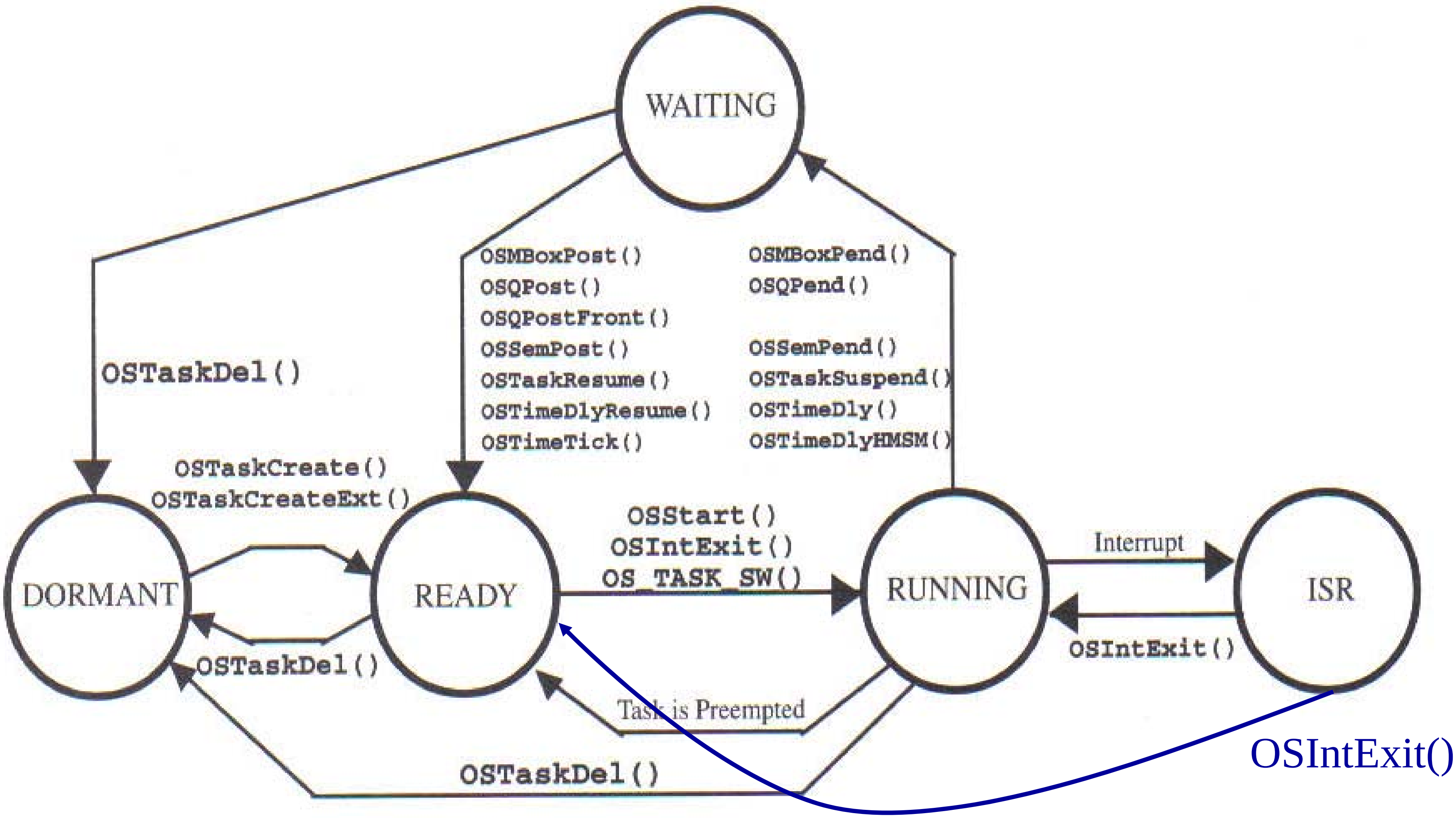
3.1 A task and its data structure



Real-Time Kernels / Copyright 2001, Jean J. Labrosse

ÉCOLE POLYTECHNIQUE MONTRÉAL

3.2 Task States



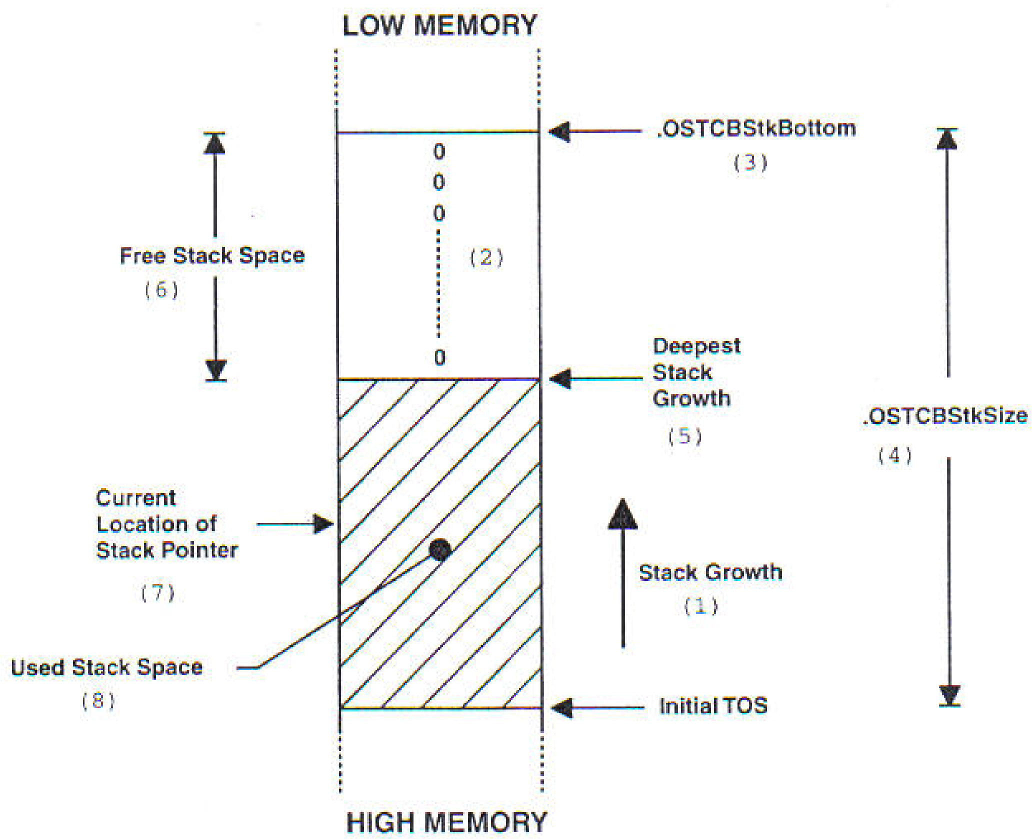
Chap2, Sections 2.1 à 2.3, page 16

3.3 Creating a task

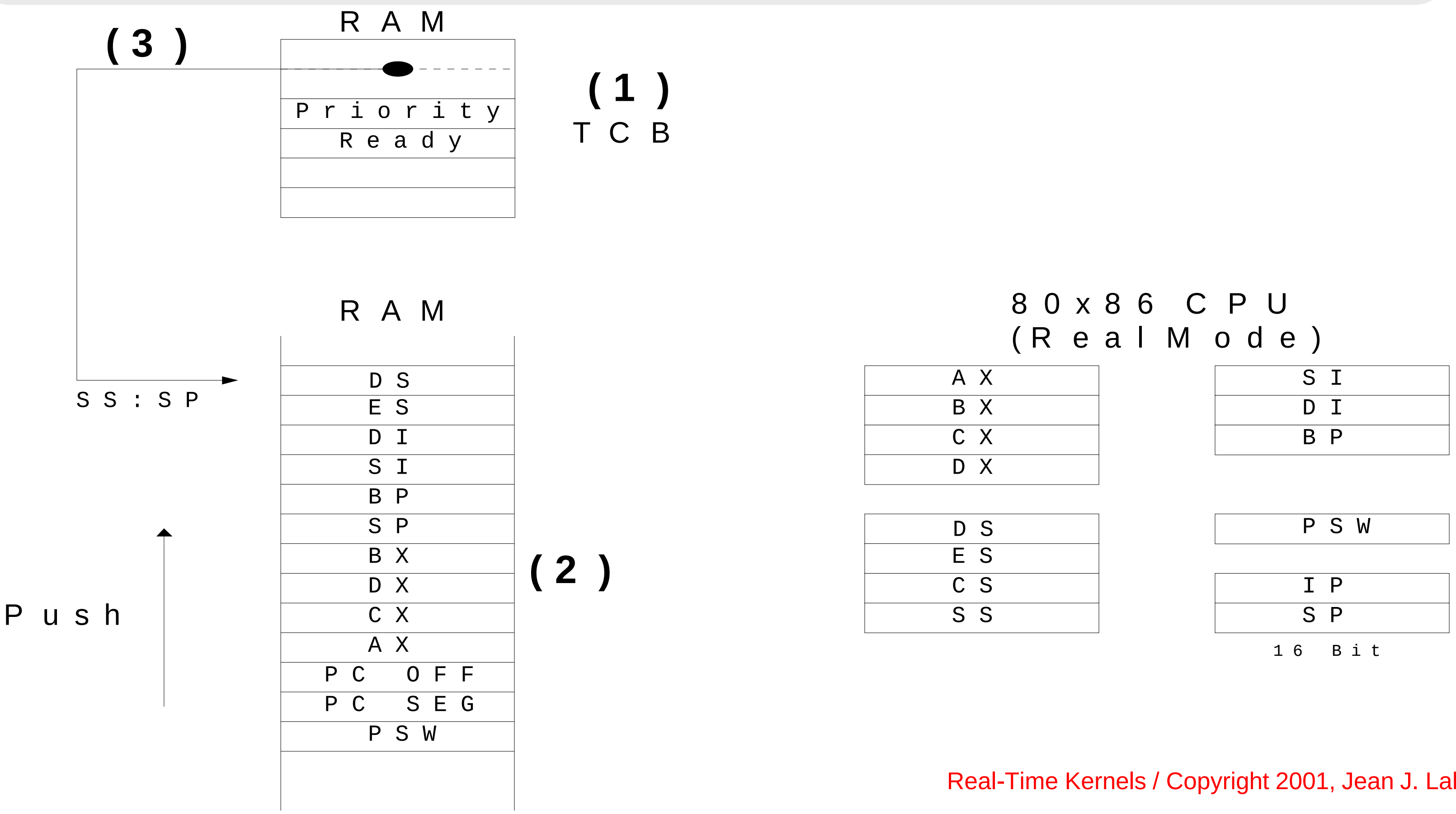
- **During the creation of a task, different operations are executed by the kernel:**
  - RAM memory allocation in the *System Memory Pool* for the stack and TCB.
  - Stack Initialization (e.g. push the first address of task code and its initial arguments, and also initialize the stack using symbols to facilitate stack checking)
  - TCB Initialization

3.3 Creating a task

Stack checking.



3.3 Creating a task



(1) Allocate TCB & Init.
(2) Init. Task Stack frame
(3) Save Top-Of-Stack (TOS) in TCB

3.3 Creating a task

- **Different ways to create a task:**
- **Examples:**
  - VxWorks: taskCreate()
  - µC/OS-II: OSTaskCreateExt()   // 9 arguments

3.3 Creating a task

```
INT8U OSTaskCreateExt(     void (*task) (void *pd),
                           void *pdata,
                           OS-STK *ptos,
                           INT8U prio,
                           INT16U id,
                           OS-STK *pbos,
                           INT32U stk-size,
                           void *pext,
                           INT16U opt);
```

- task = pointer to the task code
- pdata = pointer to an argument that is passed to your task when on the first execution
- ptos = pointer to the top of the stack
- prio = task priority
- id = unique id of the task (for future expansion)

3.3 Creating a task

Arguments of OSTaskCreateExt() (cnt'd):

- pbos = pointer to the task's bottom-of-stack (used to perform stack checking)
- stk_size = size of the stack. For example, if an entry is 4-byte wide, then a stack size of 1,000 means that the stack has 4,000 bytes.
- pext = pointer to a user-supplied data area that can be used to extend the OS_TCB of the task (e.g. floating-point registers)
- opt = specific options of OSTaskCreateExt() (see the reference manual)

3.3 Creating a task

Creation summary

```
OSTaskCreate (parameters)
{
        if (a TCB is available)
                Initialize the TCB;
        else
                Return an error code;
        Initialize the task's stack;
        Increment the number of tasks counter;
        Make task ready-to-run;
        if (OS is running)
                Call the scheduler;
}
```

3.3 Creating a task

About the size of the stack

- **The size of the stack needed by the task is application-specific. You must account:**
  - for nesting of all functions called by your task,
  - the number of local variables that will be allocated by all functions called by your task,
  - the stack requirements for all the nested interrupts,
  - the number of all the CPU registers saved during the context switch,
- **A good practice is to allocate twice of the estimation and to execute many times for critical situations. Then, see if you can decrease.**

3.3 Creating a task

Minimizing the latency

- **Creating a task increases the execution time and can be a non-deterministic operation**
  - A good practice is to create all the tasks required during the initialization (special functions) and suspend those not immediately required. They will be resumed later.
  - Also if possible (e.g. after debugging) turn the option to determine how much task space is used off (*opt* parameter in *OSTaskCreateExt*).

3.3 Creating a task

Information about a task

- **Each RTOS provides system functions to retrieve information about a task:**
  - name, ID, task code, state, priority, delay.
  - information about the stack
  - options of the task
  - contents of the general registers
  - contents of the floating point registers (FPU)

- ***OSTaskQuery()* performs such a function with µC/OS-II.**

ÉCOLE POLYTECHNIQUE MONTRÉAL

3.4 Deleting a task

- **Sometimes, it is necessary to delete a task**
- **Deleting means that the task is returned to the dormant state**
- **It is performed in two steps:**
  - Remove the task from the ready list and from all the waiting list (e.g. OSReadyTbl and OSEventTbl).
  - Deallocate the TCB and the stack. Deallocated memory can be reused.

3.4 Deleting a task

- **A task is automatically deleted when the last line of its task code is encountered**
- **One can force a task to be deleted with *OSTaskDel (INT8U prio)* where *prio* is the priority of the task to delete.**

3.4 Deleting a task

Warning

- **Sometimes, a task owns ressources such as memory buffers or a semaphore.**
- **If another task attempts to delete this task, the ressource will stay used (locked). Therefore, ressource is lost.**
- **A good pratice is to use *OSTaskDelReq(INT8U prio):***
  - **If Task A wants to delete Task B using *OSTaskDelReq*, then A asks B to *release all resources first and then to delete itself after.***

3.4 Deleting a task

Case of VxWorks

- **Before the acquisition of a resource (semaphore, mutex, etc.), the system function *TaskSafe()* can be used in order to avoid a deletion from other tasks.**

3.4 Deleting a task

Another good pratice

- **Allocation and deallocation of TCB and stack are managed by the µC/OS-II**
- **But a task is responsible to:**
  - deallocate its own allocated memory (malloc, new)
  - properly release shared resource mechanisms (mutex, semaphore, etc.)
  - close all files
  - ...

Chap2, Sections 2.1 à 2.3, page 31

3.5 Suspending/resuming a task

- **Sometimes, it is useful to suspend the execution of a task explicitly. The *Suspended State* is included within the *Waiting State*.**
- **A suspended task can be resumed later**
- **With µC/OS-II:**
  - ***OSTaskSuspend(INT8Uprio)***
  - ***OSTaskResume (INT8U prio)***

3.6 Delaying a task

- **A task must have a mechanism to delay itself for an integral number of timer ticks.**
- **Very useful for a periodic execution or a busy waiting (read the status of an external device periodically).**
- ***Delayed state* is included in the *waiting state*.**
- **In µC/OS-II, for a task to delay itself:**
  - ***OSTimeDly(ticks)*, where *ticks* is the number of clock ticks from the timer;**
  - ***OSTimeDlyHMSM(H,M,S,M)*, with *H*ours, *M*inutes, *S*econds, *M*icroseconds**

3.7 Changing the priority of a task

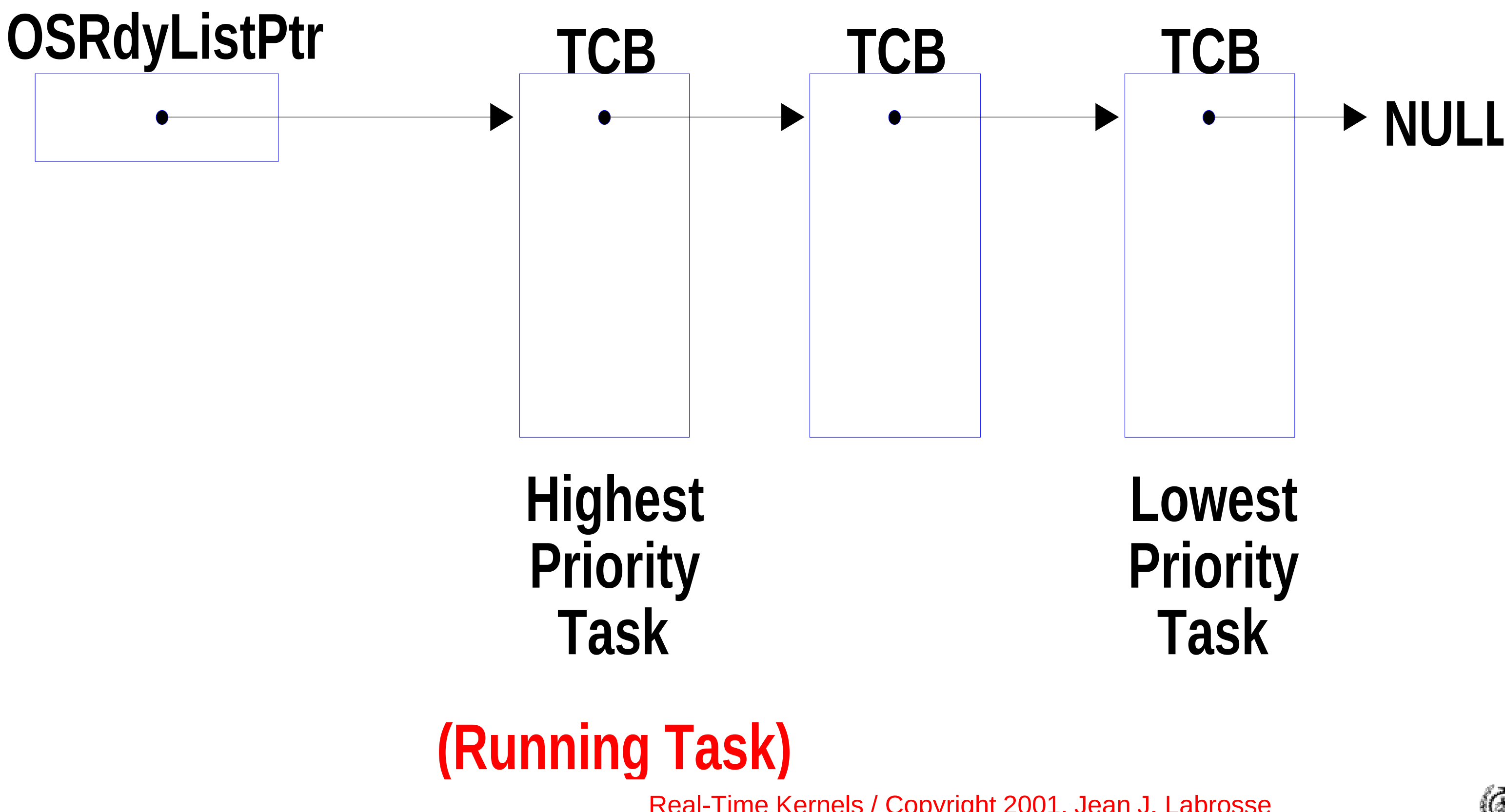
- **When you create a task, you assign the task priority. At runtime, you can change the priority of any task.**
- **In other words, you can change priorities dynamically**
- **In µC/OS-II, for a task to change its priority:**
  - ***OSTaskChange(oldprio, newprio);***

3.8 Tasks scheduling

- **A <u>queue</u> is used to maintain tasks that are ready to <u>run</u> (similarly, waiting tasks are memorized in queues, see chapter 5).**
- **The queue of "ready to run" tasks is sorted by priorities. That is, the highest priority task ready to run is selected first.**
- **More precisely, the type of queue is a bitmap. It allows the removing/deleting in O(constant).**

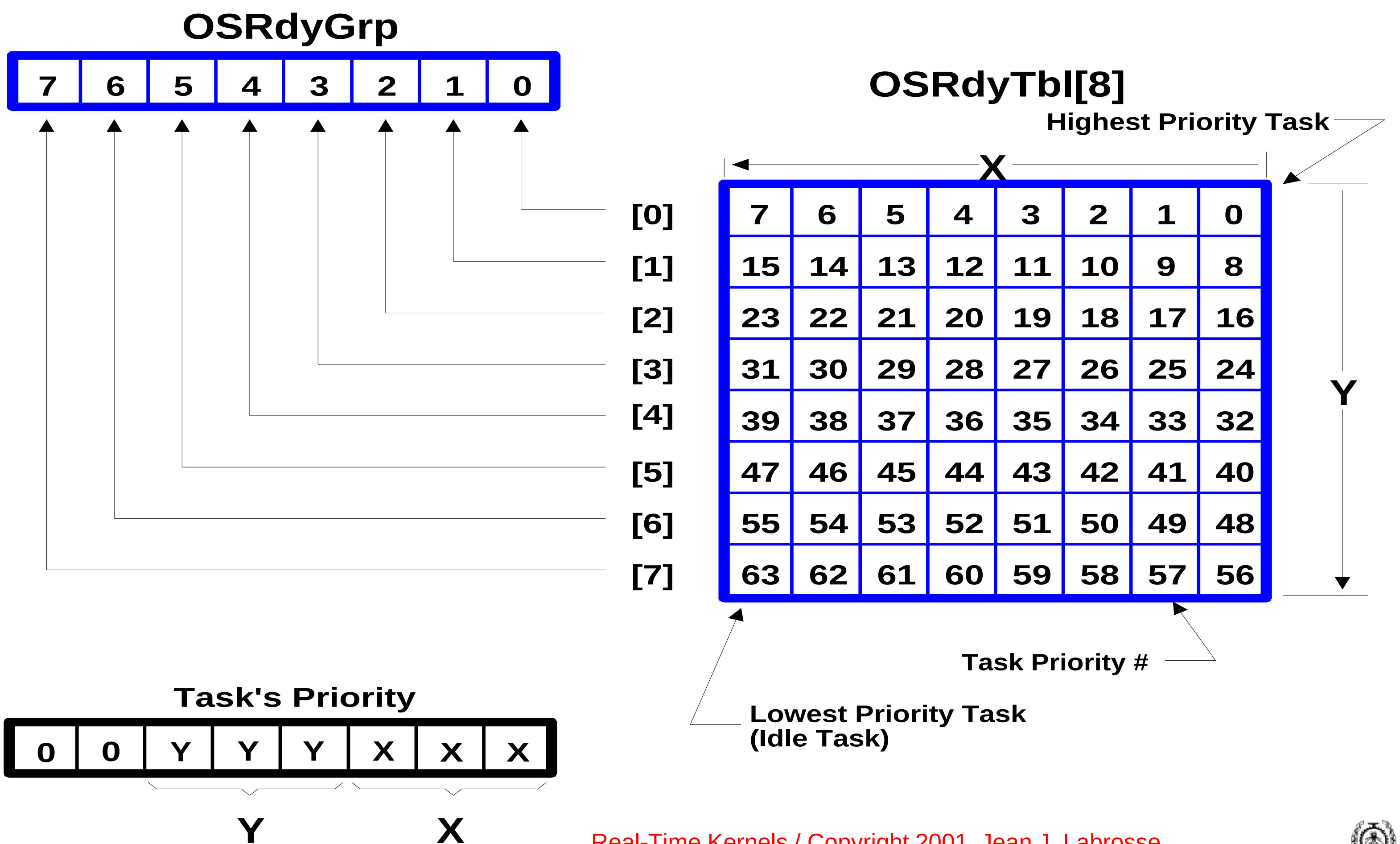
3.8 Tasks scheduling

Queue of tasks ready to run



3.8 Tasks scheduling

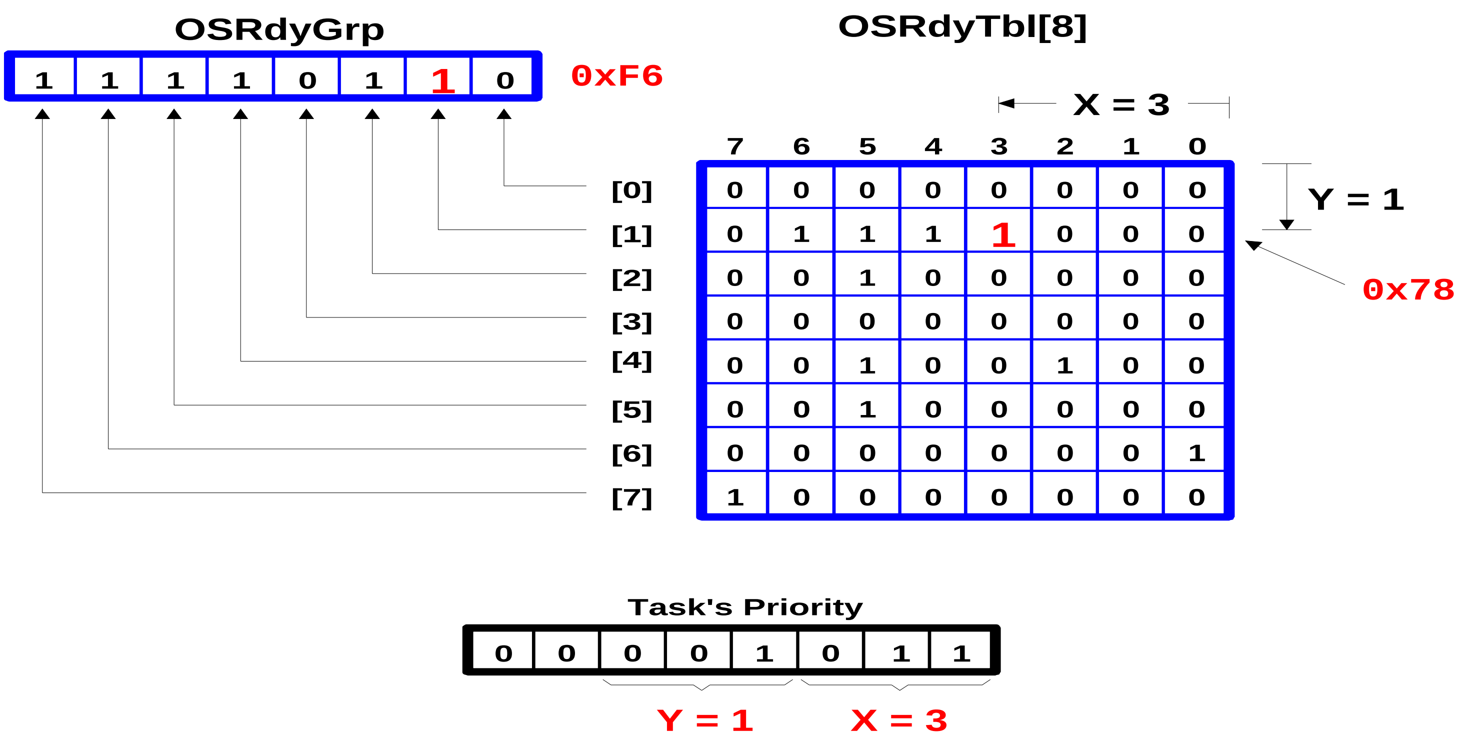
Organization of priorities



Chap2, Sections 2.1 à 2.3, page 37

3.8 Tasks scheduling

Example




```
Task's Priority = ( Y * 8) + X
Task's Priority = ( 1 * 8) + 3
Task's Priority = 11
```

3.8 Tasks scheduling

Lookup table to resolve highest priority

```
*************************************************************
* Note(s): 1) Index into table is bit pattern to resolve highest priority.
*          2) Indexed value corresponds to highest priority bit position
*(i.e. 0..7)
****************************************
INT8U const OSUnMapTbl[] = {
   0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x00-0x0F
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x10-0x1F
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x20-0x2F
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x30-0x3F
   6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x40-0x4F
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x50-0x5F
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x60-0x6F
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x70-0x7F
   7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x80-0x8F
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0x90-0x9F
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0xA0-0xAF
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0xB0-0xBF
   6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0xC0-0xCF
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0xD0-0xDF
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,   // 0xE0-0xEF
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0    // 0xF0-0xFF };
```

X = @ [0x78]

(i.e. 0x78 = OSRdyTbl[1])

Y = @ [0xF6]

(i.e. 0xF6 = OSRdyGrp)

ÉCOLE POLYTECHNIQUE MONTRÉAL

3.8 Tasks scheduling

Code to find the task ready to run with the highest priority

```
Y               = OSUnMapTbl[OSRdyGrp];
X               = OSUnMapTbl[OSRdyTbl[Y]];
HighestPriority = (Y * 8) + X;


Y (i.e. 1)      = OSUnMapTbl[0xF6];
X (i.e. 3)      = OSUnMapTbl[0x78];
HighestPriority = (1 * 8) + 3;
HighestPriority = 11
```

3.8 Tasks scheduling

- **4 constants (memorized in the TCB) are used to speedup access to OSRdyTbl and OSRdyGr:**

OSTCBY	=>	ptcb->OSTCBY = priority >> 3;
OSTCBBitY	=>	ptcb->OSTCBBitY = OSMapTbl[ptcb->OSTCBY]
OSTCBX	=>	ptcb->OSTCBX = priority & 0x07;
OSTCBBitX	=>	ptcb->OSTCBBitX = OSMapTbl[ptcb->OSTCBX];

Index	Bit Mask
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

OSMapTbl[]

3.8 Tasks scheduling

- **Code to remove a task from the ready list:**

```
if ((OSRdyTbl[ptcb->OSTCBY]  &=  ~ ptcb->OSTCBBitX) ==0)
{
   OSRdyGrp  &= ~ ptcb->OSTCBBitY;
}
```

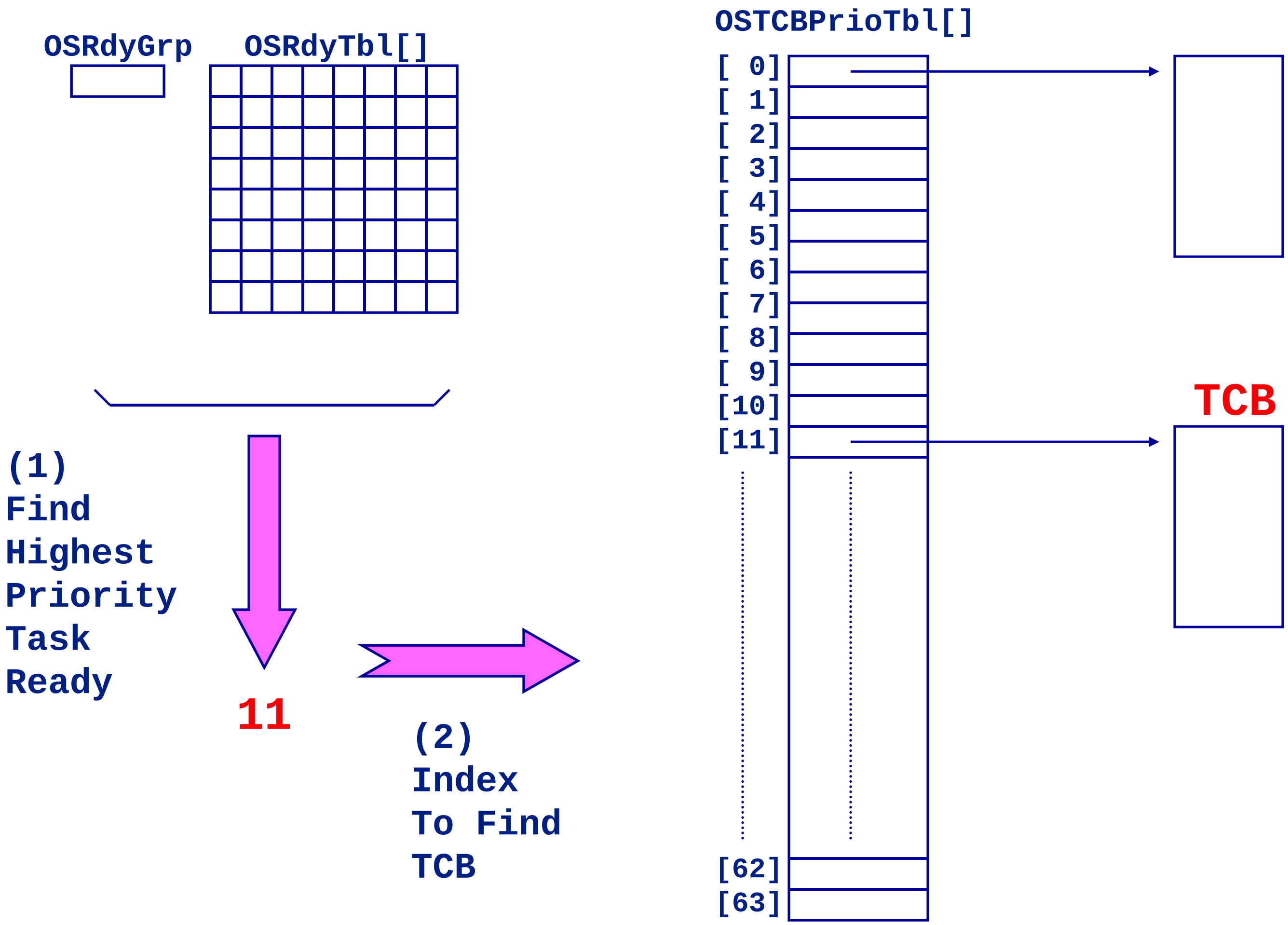
3.8 Tasks scheduling

- **Code to make a task ready to run**

```
OSRdyGrp                      |=    ptcb->OSTCBBitY;
OSRdyTbl[ptcb->OSTCBY]        |=    ptcb->OSTCBBitX;
```

3.8 Tasks scheduling

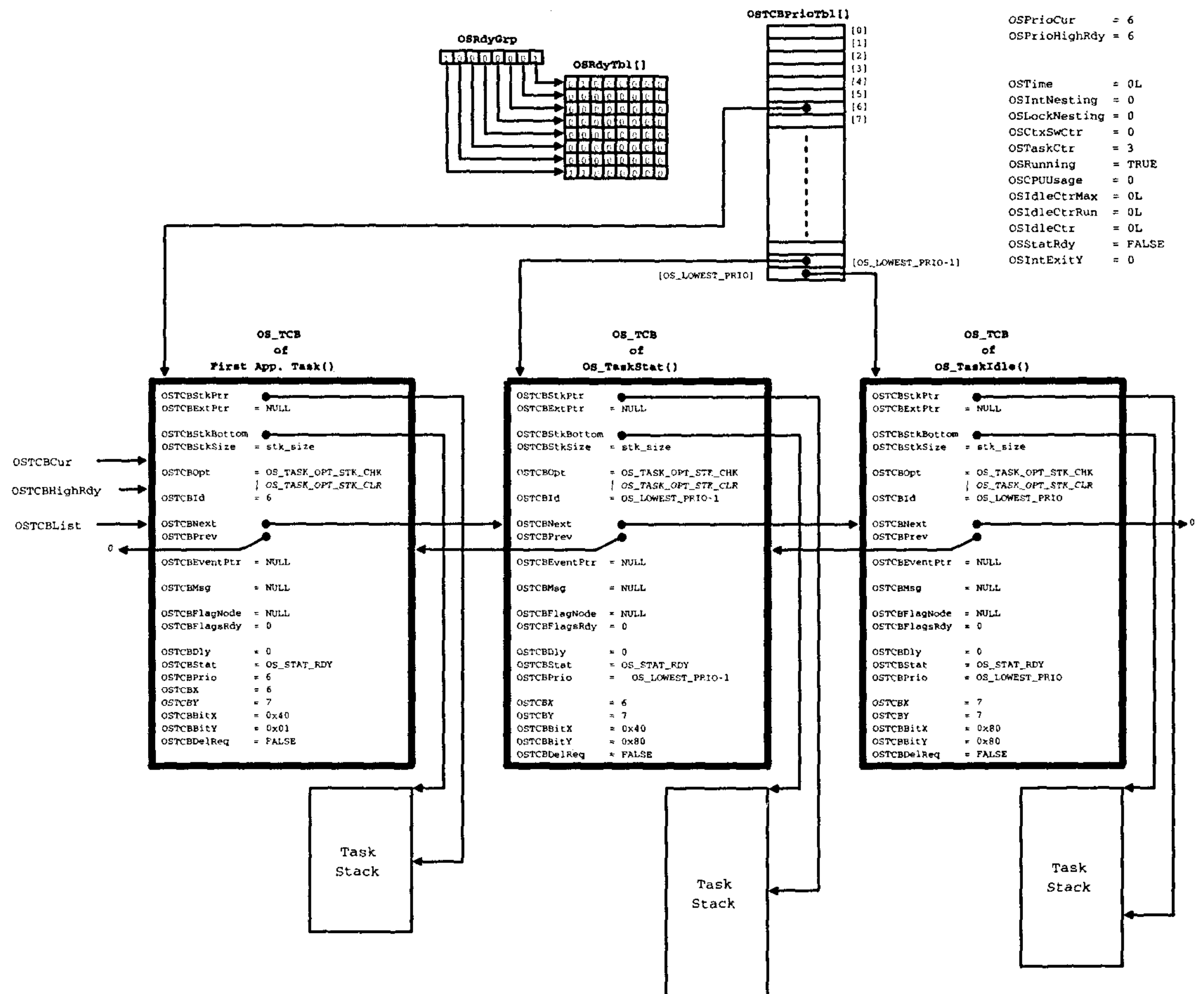
Highest priority task ready to run to its TCB



Chap2, Sections 2.1 à 2.3, page 44

In more details

Variables and data structures after calling `OSStart()`*.*



ÉCOLE POLYTECHNIQUE MONTRÉAL

3.9 Systems tasks

- **RTOS includes several system tasks:**
  - Tasks used to log system messages without having to perform I/O in the current task context. (ex. VxWorks: tLogTask)
  - Task to execute function that cannot occur at interrupt level. It must have the highest priority in the system (ex. VxWorks: tExcTask)
  - Task executed when none other are ready to run. (*idle task*) must be the lowest priority
  - Tasks to provide run-time statistics (*statistics task*)

3.9 Systems tasks

Idle task

- **The lowest priority in µC/OS-II is *OS_LOWEST_PRIO* (e.g. 63).**
- **The task named *OS_TaskIdle()* can never be deleted by applications software.**
- ***OS_TaskIdle()* increments a 32-bit counter called *OSIdleCtr*.**

3.9 Systems tasks

Statistics task

- **In µC this task is called *OS_TaskStat()* and is created if you set the configuration constant *OS_TASK_STAT_EN* to 1.**
- **When enabled, *OS_TaskStat()* executes every second and computes the percentage of CPU usage.**
- **The priority of the statistics task is OS_LOWEST_PRIO-1.**

Chap2, Sections 2.1 à 2.3, page 48

3.9 Systems tasks

Statistics task

- **To compute the percentage of CPU usage, you must initially call the function *OSStatInit()* from the highest priority task.**
- ***OSStatInit()* determines the maximum value that a 32-bit counter (named *OSIdleCtrMax*) can reach during 1 second when no other task is executed.**

3.9 Systems tasks

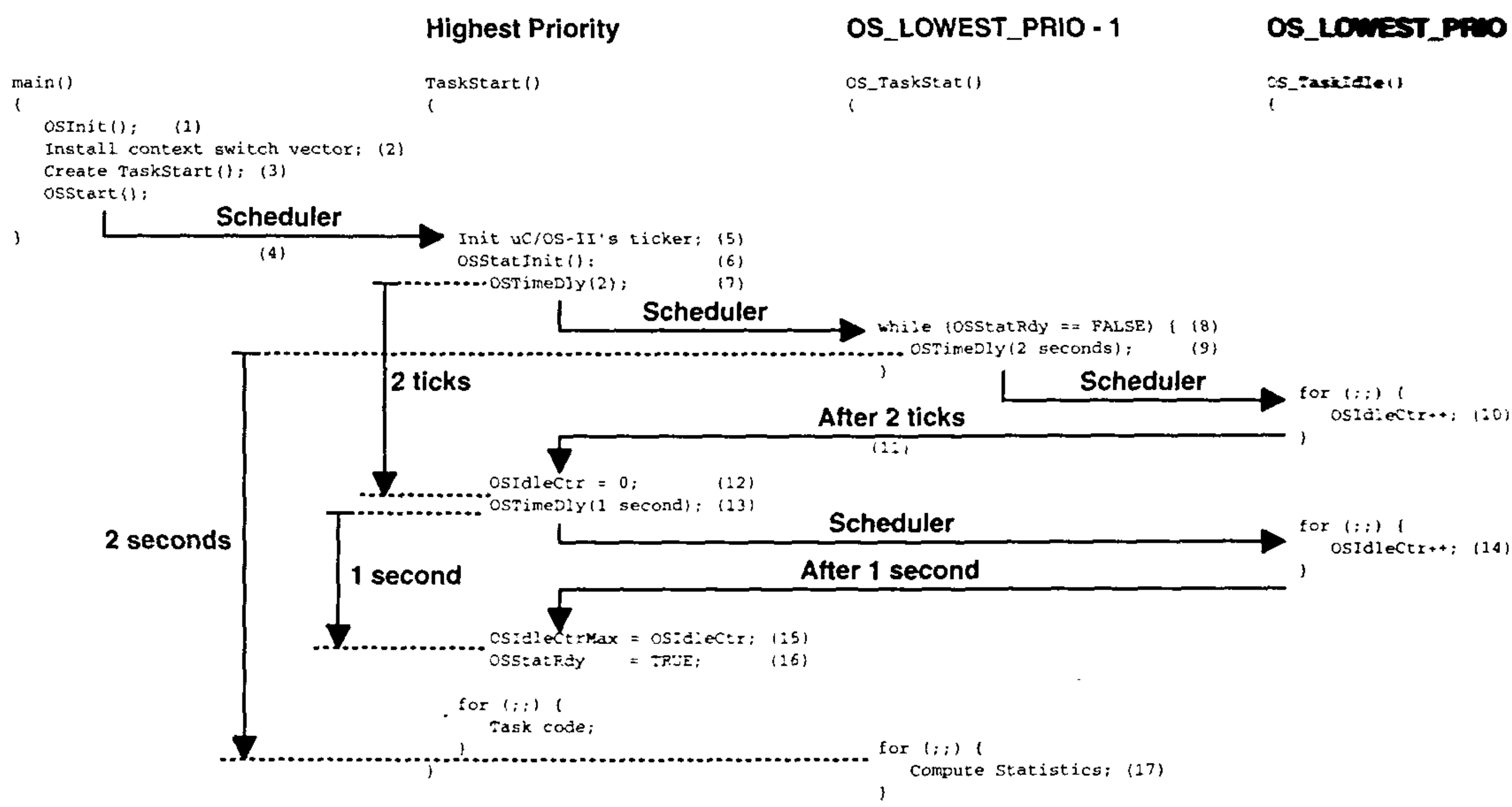
Statistics task

- **Each time the statistics task is excuted, *OS_TaskStat()* compares *OSIdleCtrMax and OSIdleCtr.***

$$OSCPUUsage(\%) = 100 * (1 - OSIdleCtr / OSIdleCtrMax)$$

- **That is, CPU usage is stored in the variable *OSCPUUsage***

Statistic task initialization.



ÉCOLE POLYTECHNIQUE MONTRÉAL