Embedded OS Concepts

Embedded OS Implementation

Prof. Ya-Shu Chen
National Taiwan University
of Science and Technology

Objectives

- Understand
  - Context switch
  - Priority-driven scheduling
  - Interrupt handling

Multitasking

- The scheduler switches the attention of CPU among several tasks
  - Tasks logically execute concurrently by sharing the CPU
  - How much CPU share could be obtained by each task depends on the scheduling policy adopted

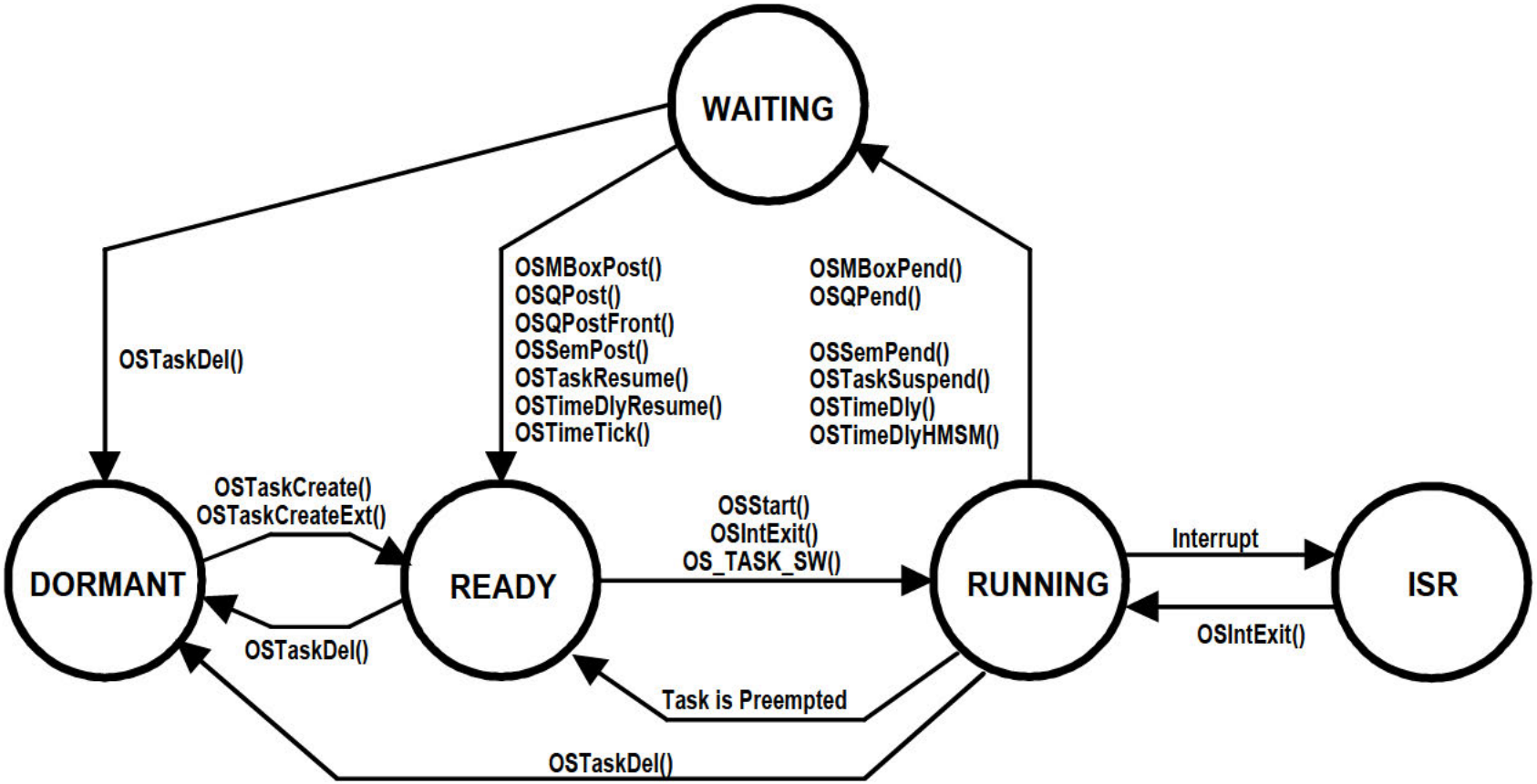
Task

- Sometimes referred to as a process
  - An active entity that does computation
- From the OS point of view, a task is of a priority, a set of registers, its own stack, and some housekeeping information

Task

- A real-time periodic task is basically an infinite loop

- There are 5 task states under uC/OS-2:
  - Dormant, ready, running, waiting, and interrupted.


WAITING
OSTaskDel()
OSMBoxPost()
OSQPost()
OSQPostFront()
OSSemPost()
OSTaskResume()
OSTimeDlyResume()
OSTimeTick()
OSMBoxPend()
OSQPend()
OSSemPend()
OSTaskSuspend()
OSTimeDly()
OSTimeDlyHMSM()
OSTaskCreate()
OSTaskCreateExt()
DORMANT
READY
OSStart()
OSIntExit()
OS_TASK_SW()
RUNNING
Interrupt
ISR
OSIntExit()
OSTaskDel()
Task is Preempted
OSTaskDel()

Kernels

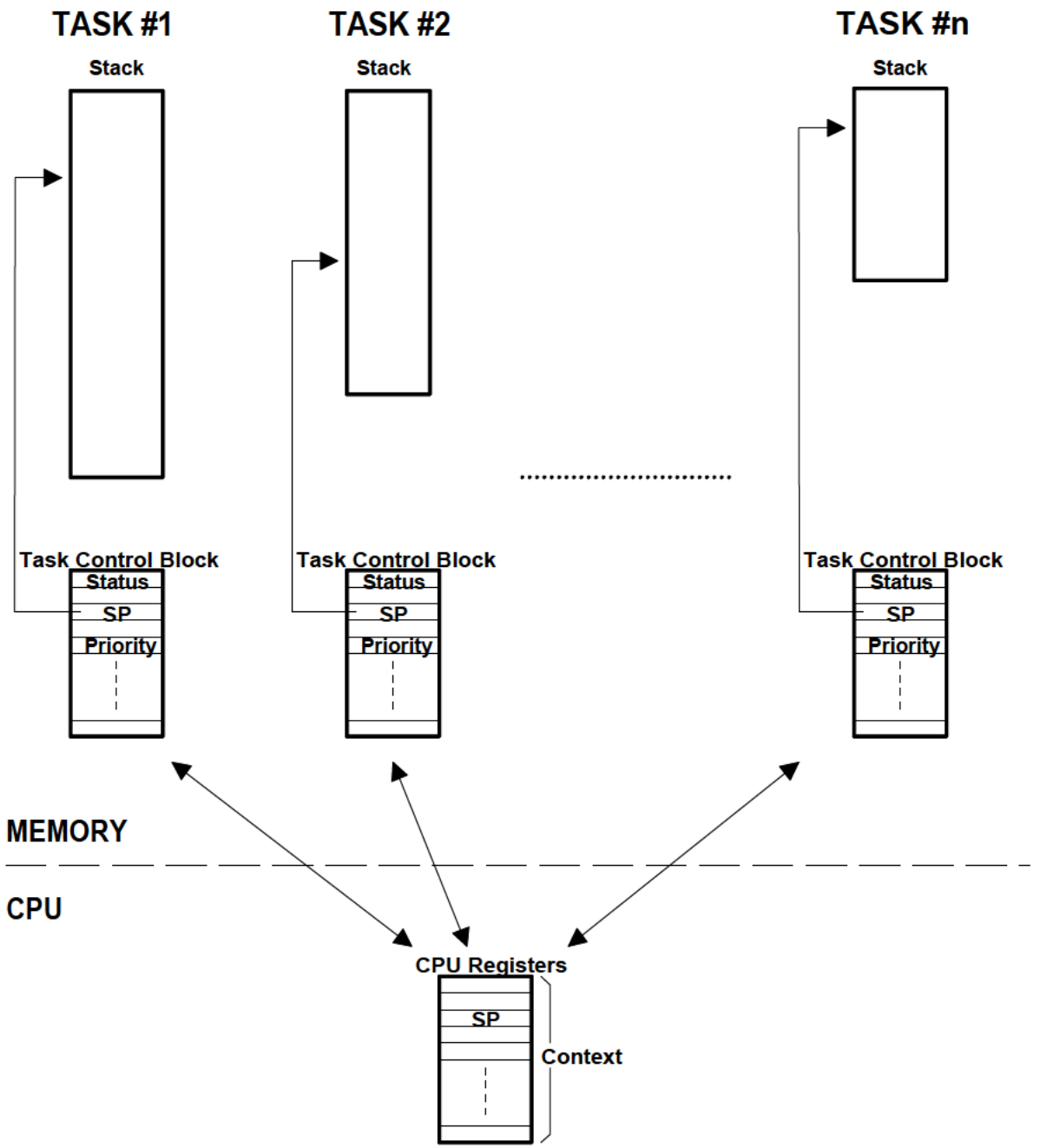
- The kernel is responsible to:
  - The management of tasks
  - Inter-task communication
- The kernel imposes additional overheads on task execution
  - Kernel services take time
    - Semaphores, message queues, mailboxes, timing controls, and etc…
  - Kernel resident in RAM and/or ROM

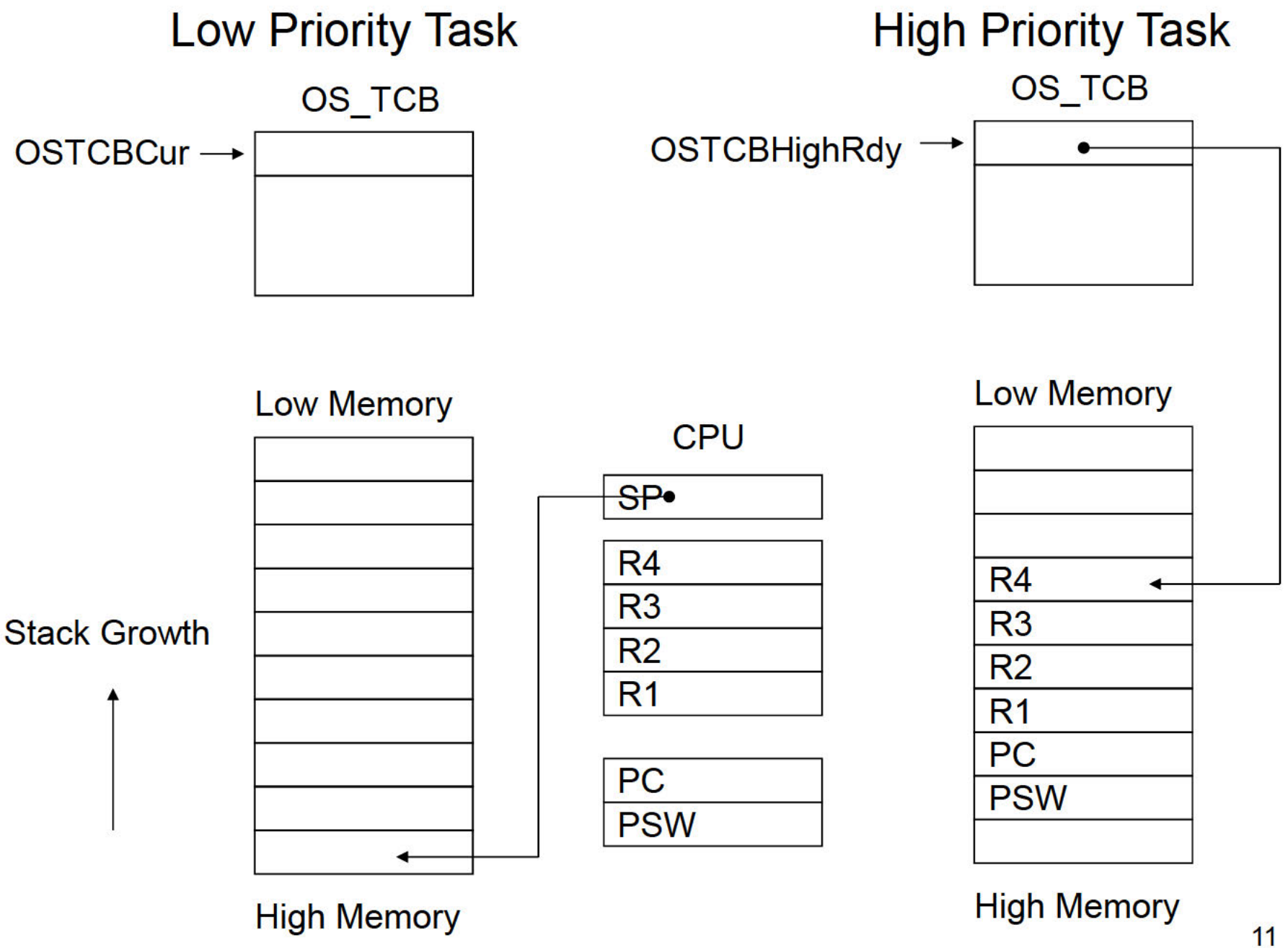
Context Switch (1/2)

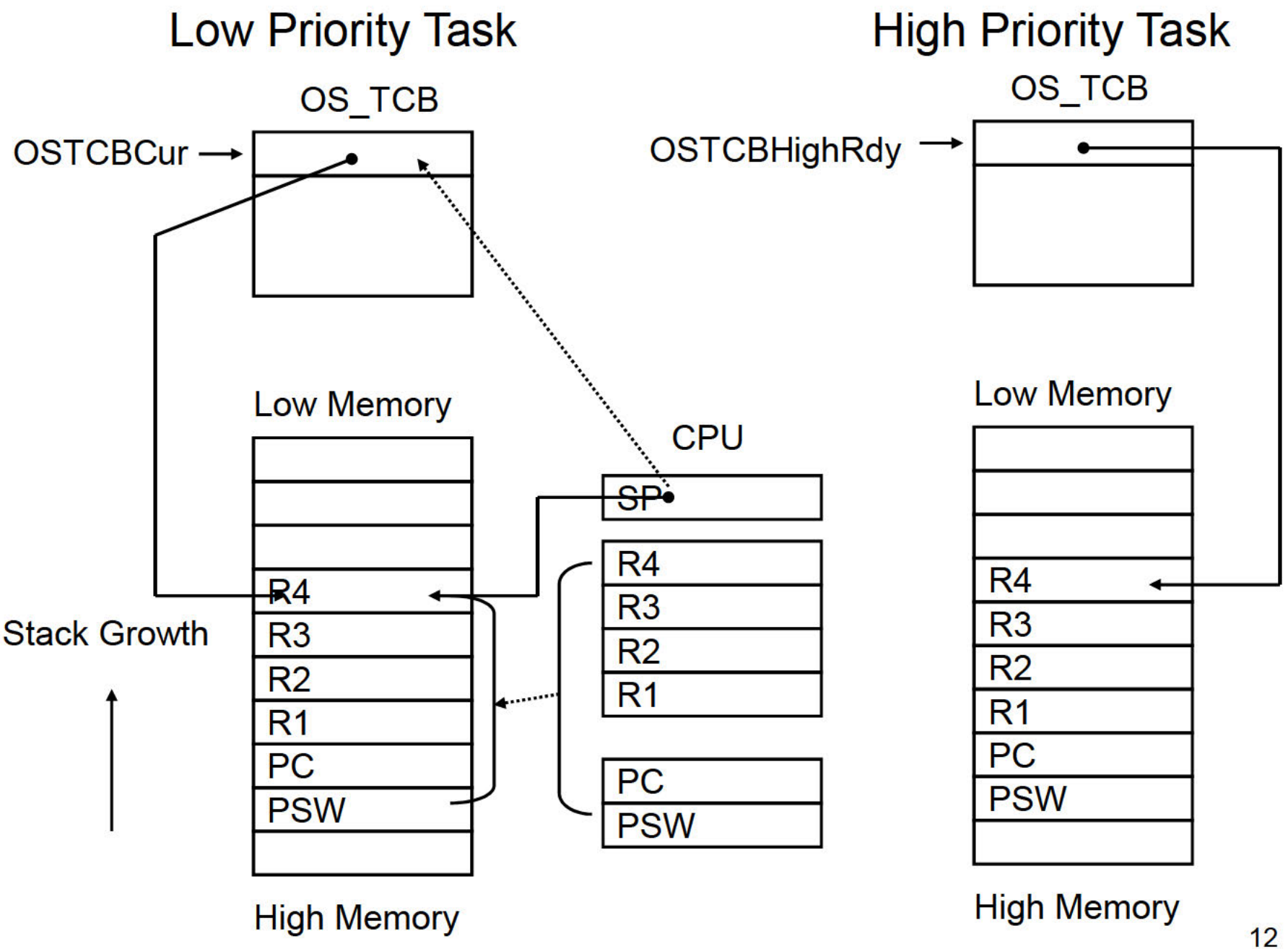
- It occurs when the scheduler decides to switch to a different task
- The scheduler must save the context of the current task and then restore the context of the task-to-run.
  - The context is of a priority, the contents of the registers, the pointers to its stack, and the related housekeeping data

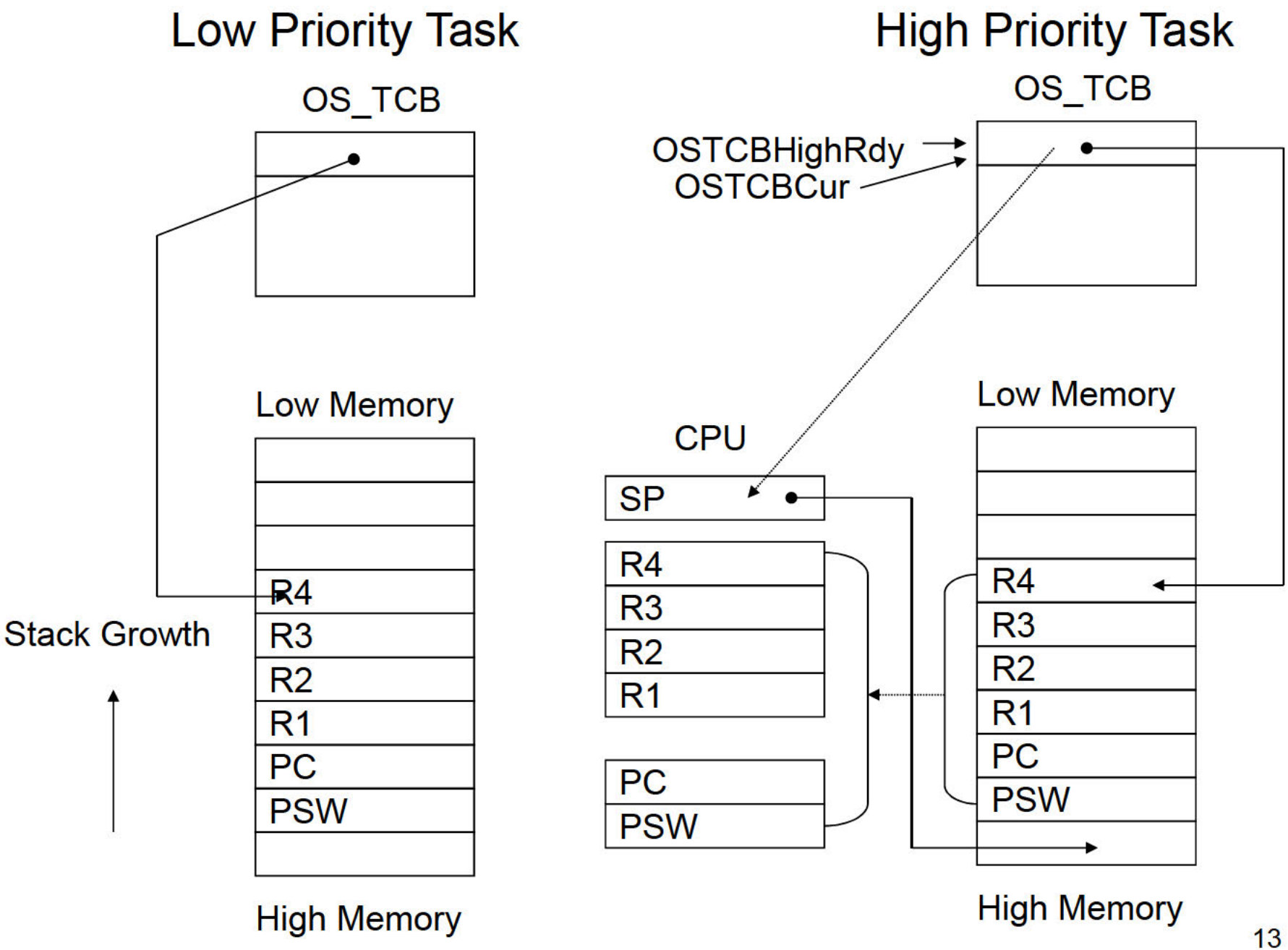
Context Switch (2/2)

- Context switches impose overheads on the execution of tasks
  - Intensive context switches should be prevented because modern processors have deep pipelines and large register files
- For a real-time operating system, we must know how much time it takes to perform a context switch
  - The overheads of context switch are accounted as overheads of high-priority tasks


TASK #1
TASK #2
TASK #n
Stack
Stack
Stack
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
MEMORY
CPU
CPU Registers
SP
Context


Low Priority Task
OS_TCB
OSTCBCur
High Priority Task
OS_TCB
OSTCBHighRdy
Low Memory
CPU
SP
R4
R3
R2
R1
PC
PSW
Stack Growth
High Memory
Low Memory
R4
R3
R2
R1
PC
PSW
High Memory


Low Priority Task
OS_TCB
OSTCBCur
Low Memory
CPU
SP
R4
R3
R2
R1
PC
PSW
Stack Growth
High Memory
High Priority Task
OS_TCB
OSTCBHighRdy
Low Memory
R4
R3
R2
R1
PC
PSW
High Memory


Low Priority Task
OS_TCB
Low Memory
R4
R3
R2
R1
PC
PSW
Stack Growth
High Memory
High Priority Task
OS_TCB
OSTCBHighRdy
OSTCBCur
CPU
SP
R4
R3
R2
R1
PC
PSW
Low Memory
R4
R3
R2
R1
PC
PSW
High Memory

Schedulers

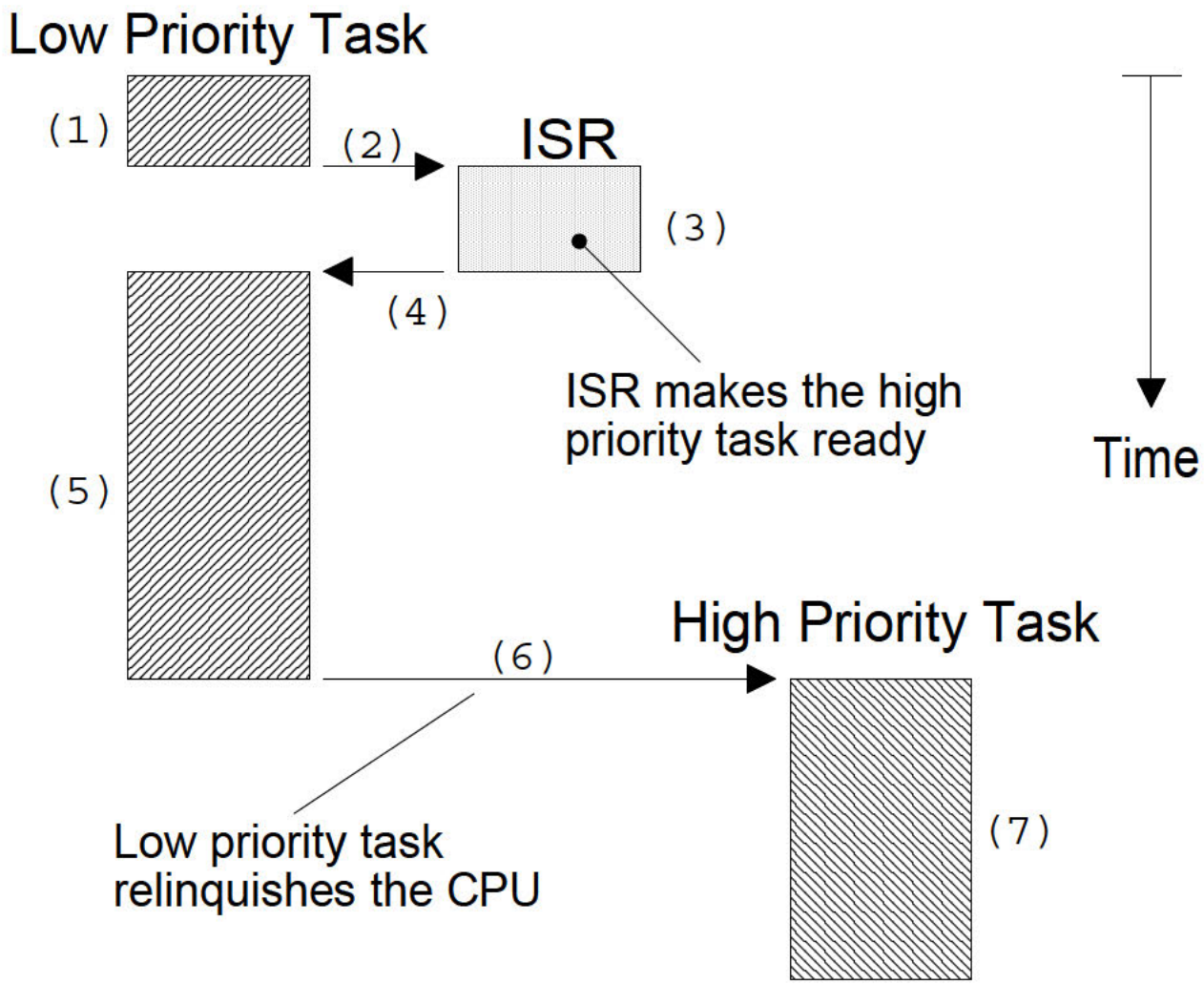
- A scheduler is a part of a kernel. It is responsible to choose the next task-to-run
  - Preemptive scheduling or non-preemptive scheduling
  - Priority-driven or deadline-driven
- For priority-driven scheduling, the highest-priority ready task always gains the control of the CPU

Non-Preemptive Kernels (1/2)

- Context switches occur only when tasks explicitly give up control of the CPU
  - High-priority tasks gain control of the CPU
  - This procedure must be done frequently to improve the responsiveness
- Events are still handled in ISR's
  - ISR's always return to the interrupted task

Non-Preemptive Kernels (2/2)

- Free from task ←→ task race conditions
  - Non-reentrant codes can be used without protections
  - In some cases, synchronizations are still needed
- Pros: simple and robust
- Cons: Not very responsive. There might be lengthy priority inversions


Low Priority Task
(1)
(2)
ISR
(3)
(4)
ISR makes the high
priority task ready
Time
(5)
High Priority Task
(6)
Low priority task
relinquishes the CPU
(7)

(1) A task is executing but gets interrupted.

(2) If interrupts are enabled, the CPU vectors (i.e. jumps) to the ISR.

(3) The ISR handles the event and makes a higher priority task ready-to-run.

(4) Upon completion of the ISR, a *Return From Interrupt* instruction is executed and the CPU returns to the interrupted task.

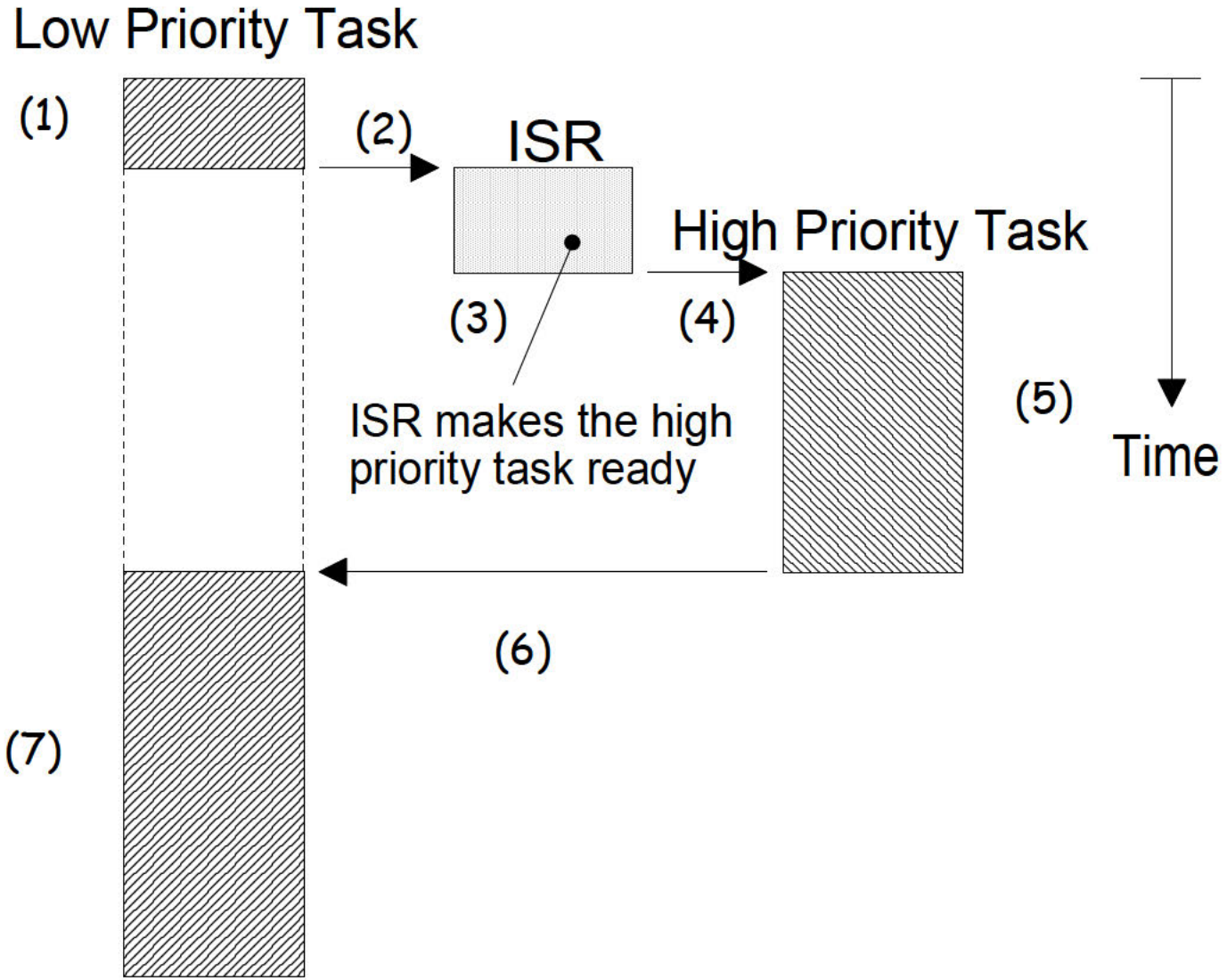
(5) The task code resumes at the instruction following the interrupted instruction.

(6) When the task code completes, it calls a service provided by the kernel to relinquish the CPU to another task.

(7) The new higher priority task then executes to handle the event signaled by the ISR.

Preemptive Kernels

- The benefit of a preemptive kernel is more responsive
  - uC/OS-2 (and most RTOS) is preemptive
  - A high-priority task gains control of the CPU instantly when it is ready (if no resource-locking is done)
- ISR might not return to the interrupted task
  - It might return a high-priority task which is ready
- IPC is needed to protect resources


Low Priority Task
(1)
(2)
ISR
High Priority Task
(3)
(4)
(5)
Time
ISR makes the high
priority task ready
(6)
(7)

(1) A task is executing but interrupted.

(2) If interrupts are enabled, the CPU vectors (jumps) to the ISR.

(3) The ISR handles the event and makes a higher priority task ready to run. Upon completion of the ISR, a service provided by the kernel is invoked. (i.e., a function that the kernel provides is called).

(4)

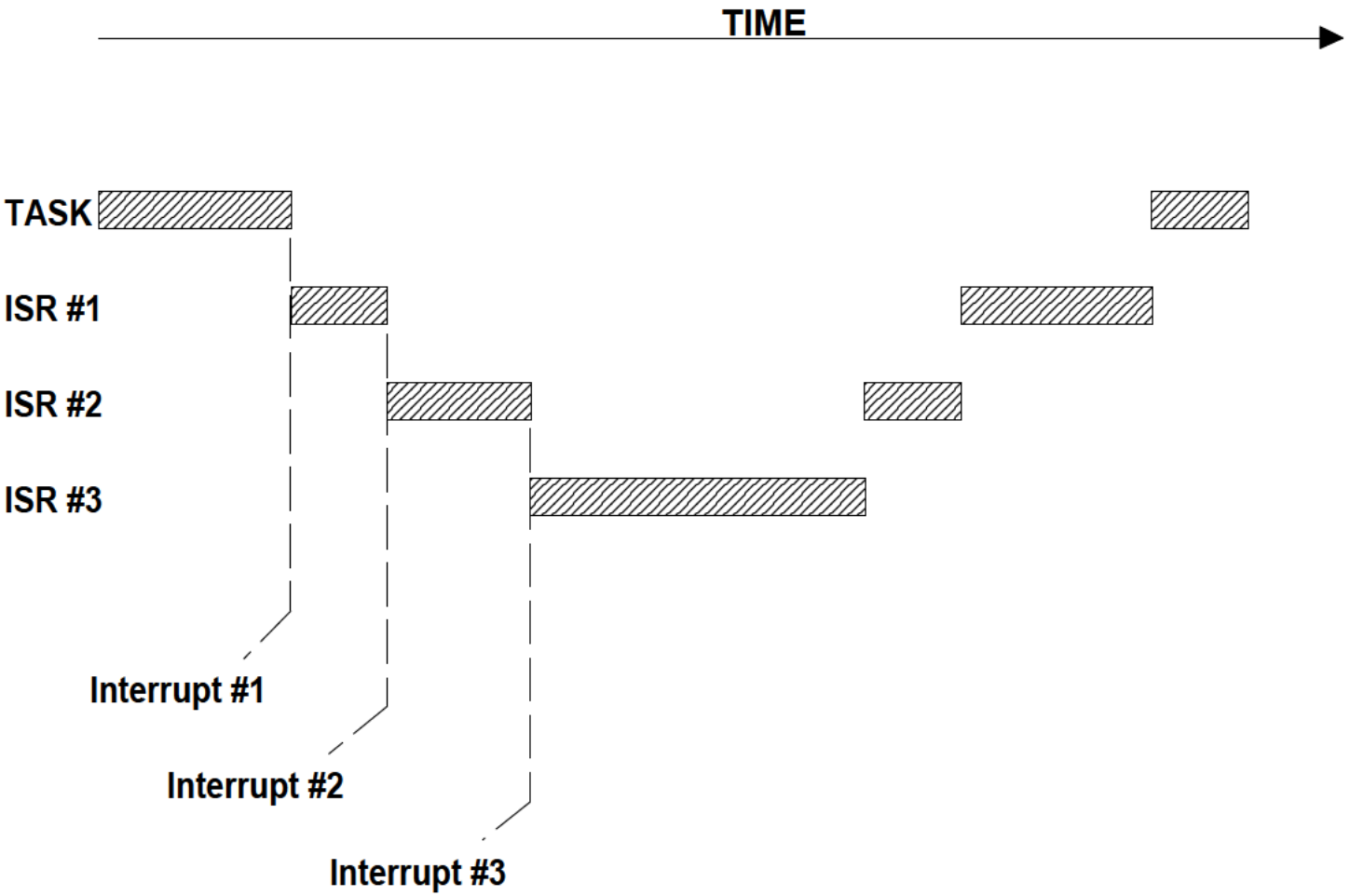
(5) This function knows that a more important task has been made ready to run, and thus, instead of returning to the interrupted task, the kernel performs a context switch and executes the code of the more important task. When the more important task is done, another function that the kernel provides is called to put the task to sleep waiting for the event (i.e., the ISR) to occur.

(6)

(7) The kernel then sees that a lower priority task needs to execute, and another context switch is done to resume execution of the interrupted task.

Interrupts

- A hardware event to inform the CPU of an asynchronous event
  - clock tick (triggering scheduling), I/O events, hardware errors.
- The context of the current task is saved and the corresponding interrupt service routine (ISR) is invoked
- The ISR processes the event, and upon completion of the ISR, the program returns to
  - The background for a foreground/background system
  - The interrupted task for a non-preemptive kernel
  - The highest priority task ready to run for a preemptive kernel


TIME
TASK
ISR #1
ISR #2
ISR #3
Interrupt #1
Interrupt #2
Interrupt #3

Interrupts under uC/OS-2

- uC/OS-2 requires an ISR written in assembly, if your compiler does not support in-line assembly.

(1) and (4)→for possible cxt switch

```
YourISR:
   Save all CPU registers;                                  (1)
   Call OSIntEnter() or, increment OSIntNesting directly;   (2)
   If(OSIntNesting == 1)                                    (3)
        OSTCBCur->OSTCBStkPtr = SP;                         (4)
   Clear the interrupting device;                           (5)
   Re-enable interrupts (optional);                         (6)
   Execute user code to service ISR;                        (7)
   Call OSIntExit();                                        (8)
   Restore all CPU registers;                               (9)
   Execute a return from interrupt instruction;             (10)
```

Interrupts under uC/OS-2

(1) In an ISR, uC/OS-2 requires that all CPU registers are saved onto the interrupted task's stack

(2) Increase the interrupt-nesting counter counter

(4) If it is the first interrupt-nesting level, we immediately save the stack pointer to OSTCBCur.

- We do this because a context-switch might occur

Interrupts under uC/OS-2

(8) Call OSIntExit(), which checks if we are in the inner-level of nested interrupts. If not, the scheduler is called
  - A potential context-switch might occur
  - Interrupt-nesting counter is decremented

(9) On the return to this point, there might be several high-priority tasks ran by the CPU
  - Since uC/OS-2 is a preemptive kernel

(10) The CPU registers are restored from the stack and the control is returned to the interrupted instruction

Interrupts under uC/OS-2

```
void OSIntExit (void)
{
   OS_ENTER_CRITICAL();
   if ((--OSIntNesting | OSLockNesting) == 0) {
      OSIntExitY    = OSUnMapTbl[OSRdyGrp];
      OSPrioHighRdy = (INT8U)((OSIntExitY << 3) +
                      OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
      if (OSPrioHighRdy != OSPrioCur) {
         OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
         OSCtxSwCtr++;
         OSIntCtxSw();
      }
   }
   OS_EXIT_CRITICAL();
}
```

If scheduler is not locked and no interrupt nesting

If there is another high-priority task ready

A context switch is performed.

Note that OSIntCtxSw() is called instead of calling OS_TASK_SW() because the ISR already saves the CPU registers onto the stack.

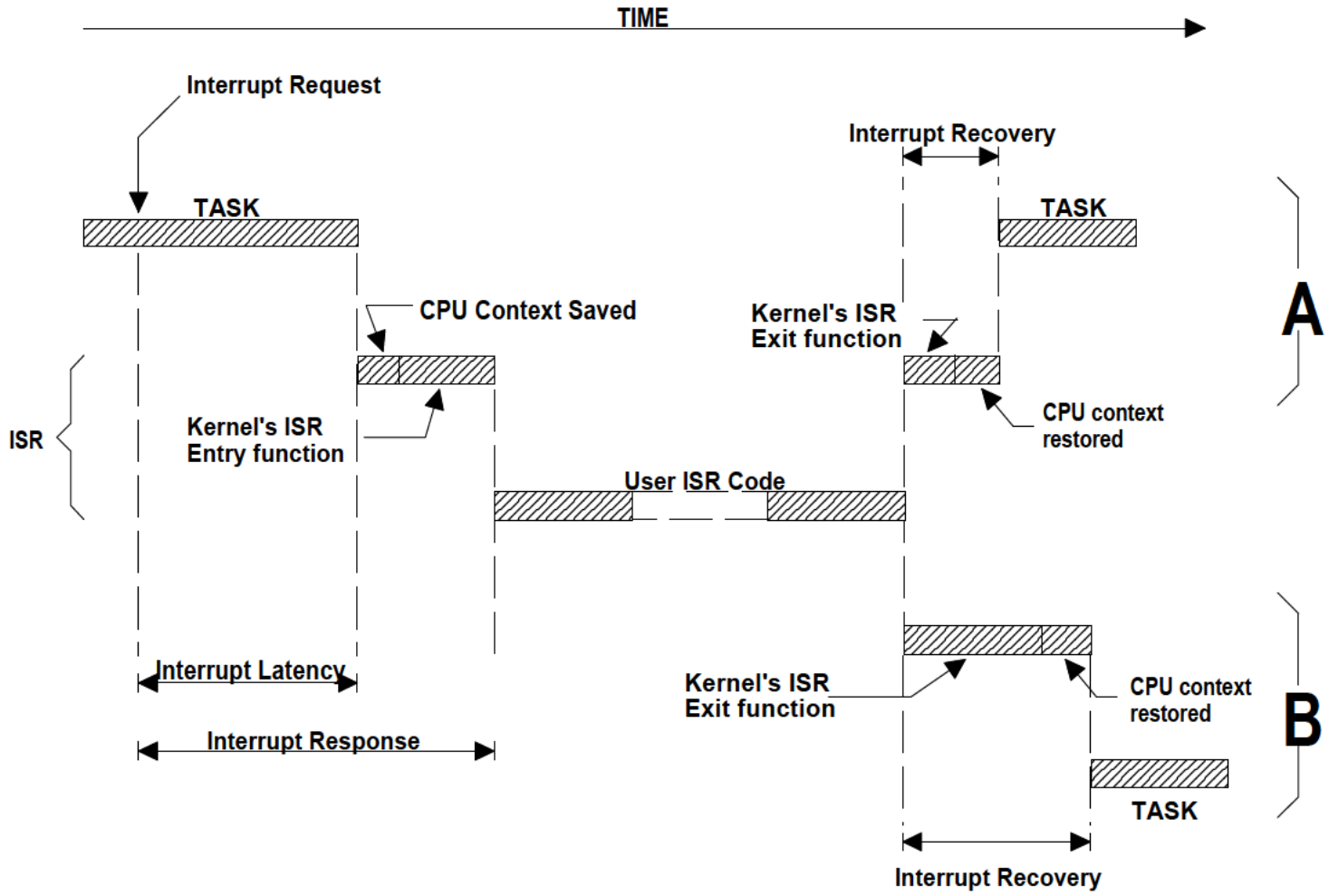
```
void OSIntEnter (void)
{
   OS_ENTER_CRITICAL();
   OSIntNesting++;
   OS_EXIT_CRITICAL();
}
```

ISR Processing Time

- ISRs should be as short as possible

- In most cases, an ISR should
  - Recognize the interrupt
  - Obtain data or status from the interrupting device
  - Pass data from devices to a task for follow-ups

Interrupt latency, response, and recovery (Preemptive kernel)



Clock Tick

- Periodic hardware event (interrupt) generated by a timer
- The heart-beat of a system
- The higher the tick rate is,
  - the better the responsiveness is
  - the higher the overhead is

Clock Tick

- A time source is needed to keep track of time delays and timeouts
- You must enable ticker interrupts after multitasking is started
- Clock ticks are serviced by calling OSTimeTick() from a clock tick ISR
- Clock tick ISR is always a port (of uC/OS-2) of a CPU
  - Since we have to access CPU registers in the tick ISR

Clock Tick

```
void OSTickISR(void)
{
    Save processor registers;
    Call OSIntEnter() or increment OSIntNesting;
    If(OSIntNesting == 1)
            OSTCBCur->OSTCBStkPtr = SP;
    Call OSTimeTick();
    Clear interrupting device;
    Re-enable interrupts (optional);
    Call OSIntExit();
    Restore processor registers;
    Execute a return from interrupt instruction;
}
```

Clock Tick

```
void  OSTimeTick (void)
{
    OS_TCB    *ptcb;


    OSTimeTickHook();

    if (OSRunning == TRUE) {
        ptcb = OSTCBList;
        while (ptcb->OSTCBPrio != OS_IDLE_PRIO) {
            OS_ENTER_CRITICAL();
            if (ptcb->OSTCBDly != 0) {
                if (--ptcb->OSTCBDly == 0) {
                    if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == OS_STAT_RDY) {
                        OSRdyGrp               |= ptcb->OSTCBBitY;
                        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                    } else {
                        ptcb->OSTCBDly = 1;
                    }
                }
            }
            ptcb = ptcb->OSTCBNext;
            OS_EXIT_CRITICAL();
        }
    }
}
```

For all TCB's

Decrement delay-counter if needed

If the delay-counter reaches zero, make the task ready. Or, the task remains waiting.

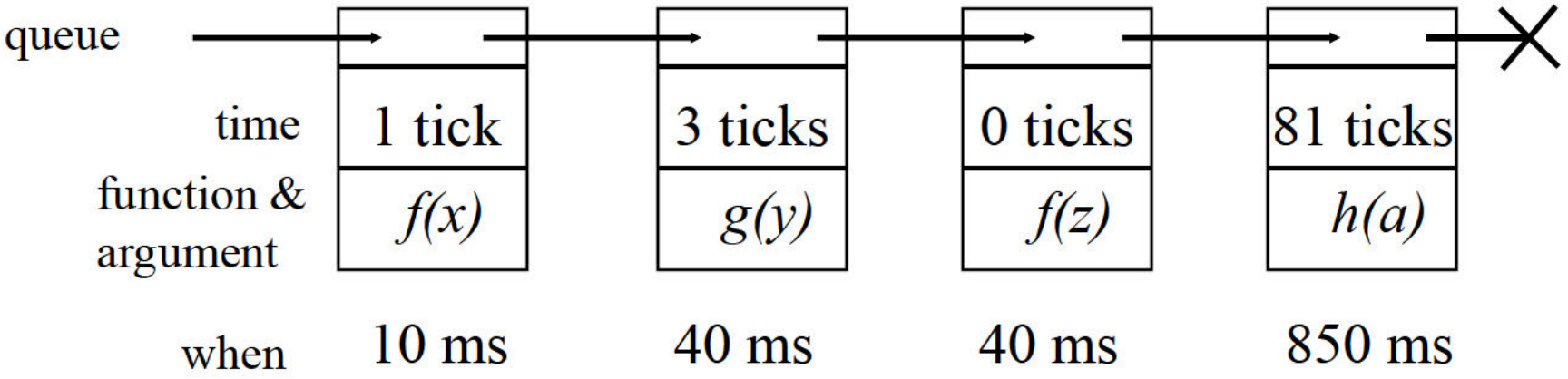
Clock Tick

- OSTimeTick() is a hardware-independent routine to service the tick ISR.

- A callout-list is more efficient on the decrementing process of OSTCBDly.
  - Constant time to determine if a task should be made ready.
  - Linear time to put a task in the list.
  - Compare it with the approach of µC/OS-II?

Callout Queue

- Sorted in time order (soonest first)
- Delta queue



Clock Tick

- You can also move a bunch of code in the tick ISR to a user task:

```
void OSTickISR(void)
{
    Save processor registers;
    Call OSIntEnter() or increment OSIntNesting;
    If(OSIntNesting == 1)
    OSTCBCur->OSTCBStkPtr = SP;

    Post a 'dummy' message (e.g. (void *)1)
      to the tick mailbox;

    Call OSIntExit();
    Restore processor registers;
    Execute a return from interrupt instruction;
}
```

Post a message

```
void TickTask (void *pdata)
{
    pdata = pdata;
    for (;;) {
        OSMboxPend(...);
        OSTimeTick();
        OS_Sched();
    }
}
```

Do the rest of the work

Memory Requirements

- Many embedded systems have tight memory-space budget
- RAM requirements can be significantly reduced if
  - Stack size of every task can differently be specified
  - A separate stack is used to handle ISR's. (uC/OS-2 doesn't, DOS does)
- RAM requirement = application requirement + kernel requirement + SUM(task stacks + MAX(ISR nesting))
- RAM requirement = application requirement + kernel requirement + SUM(task stacks) + MAX(ISR nesting)
  - If a separate stack is prepared for ISR's.

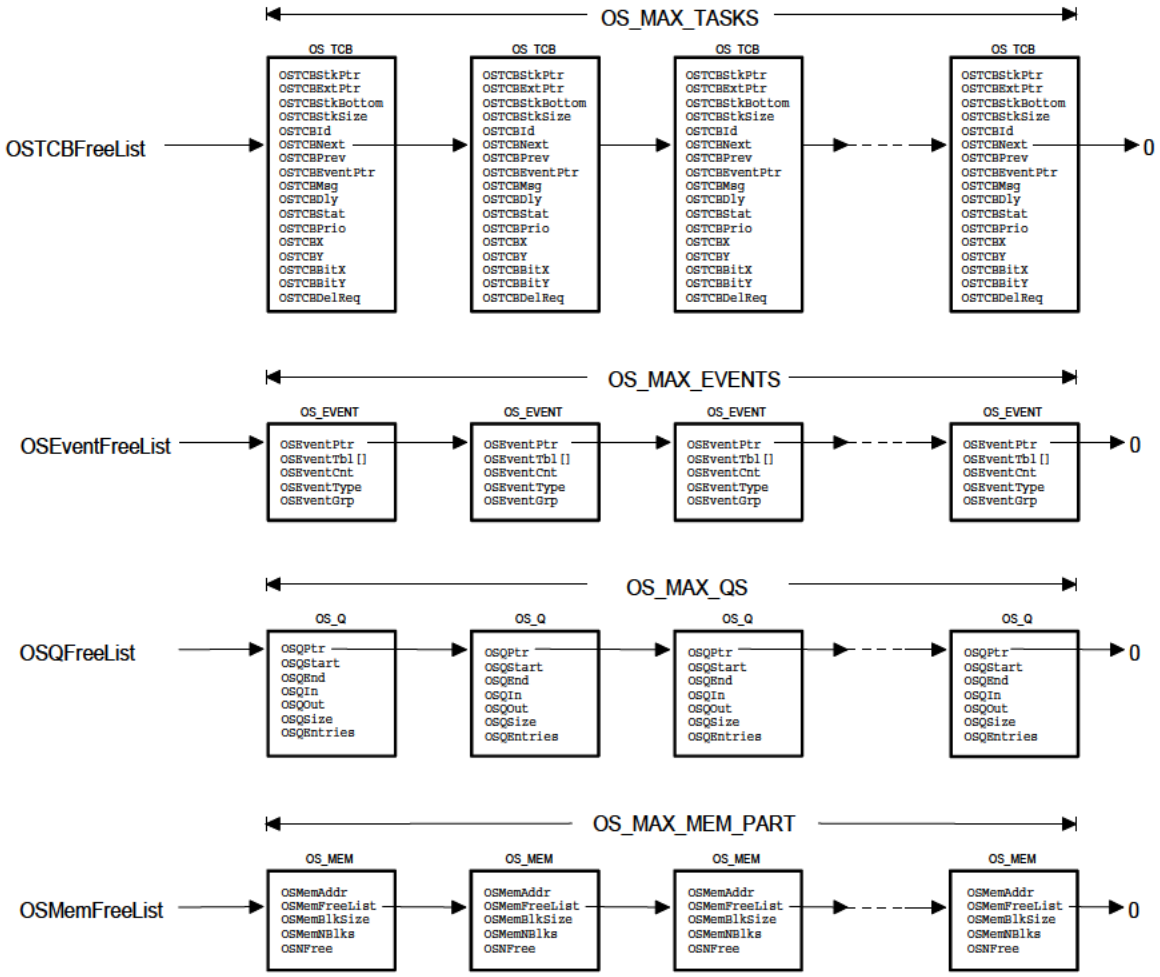
Memory Requirements

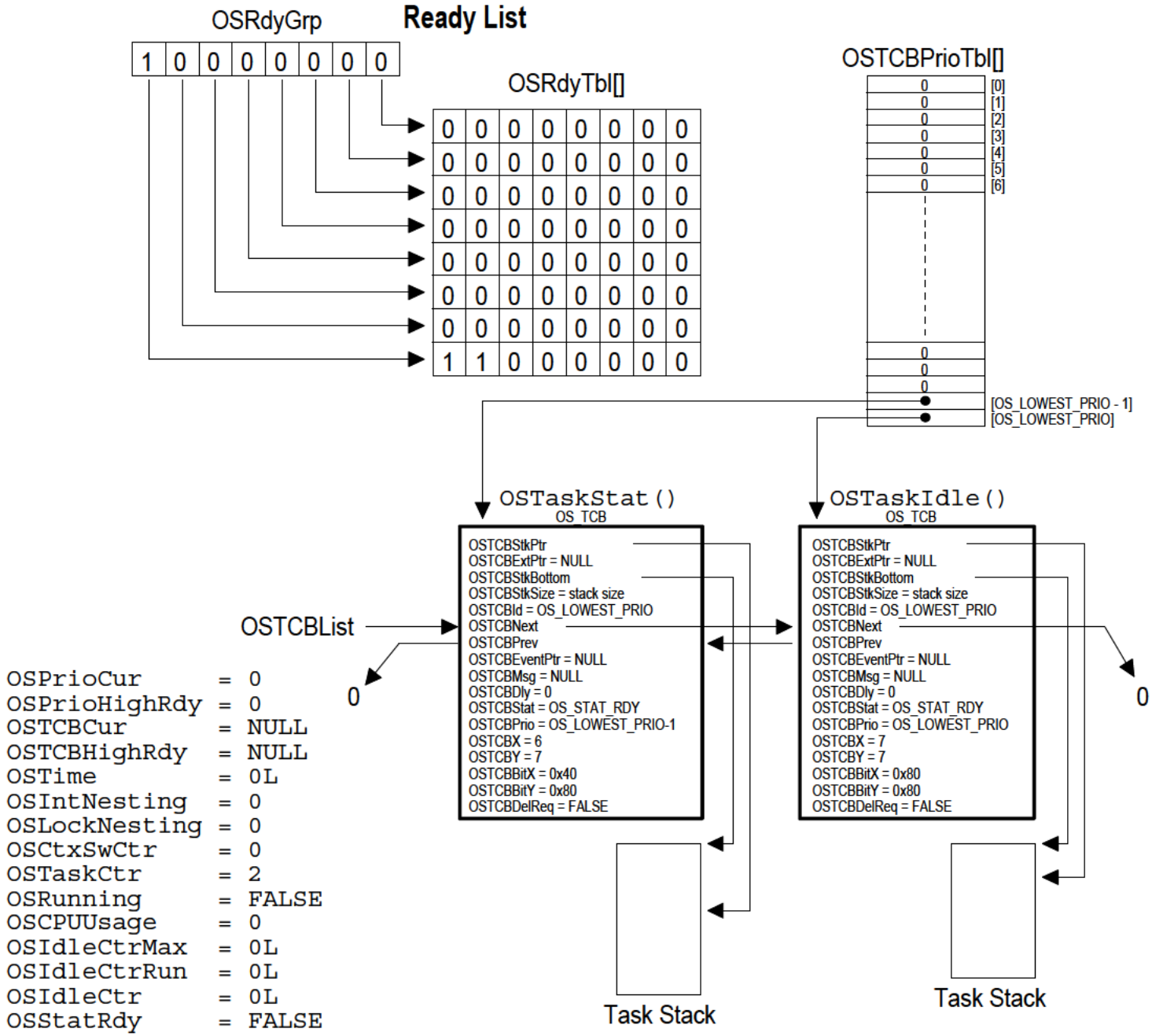
- We must be careful on the usages of tasks' stacks:
  - Large arrays and structures as local variables
  - Recursive function call
  - ISR nesting
  - Function calls with many arguments

Starting of µC/OS-II

- OSInit() initializes data structures for µC/OS-II and creates OS_TaskIdle().
- OSStart() pops the CPU registers of the highest-priority ready task and then executes a return from interrupt instruction.
  - It never returns to the caller of OSStart() (i.e., main()).

uC/OS-2 Initialization





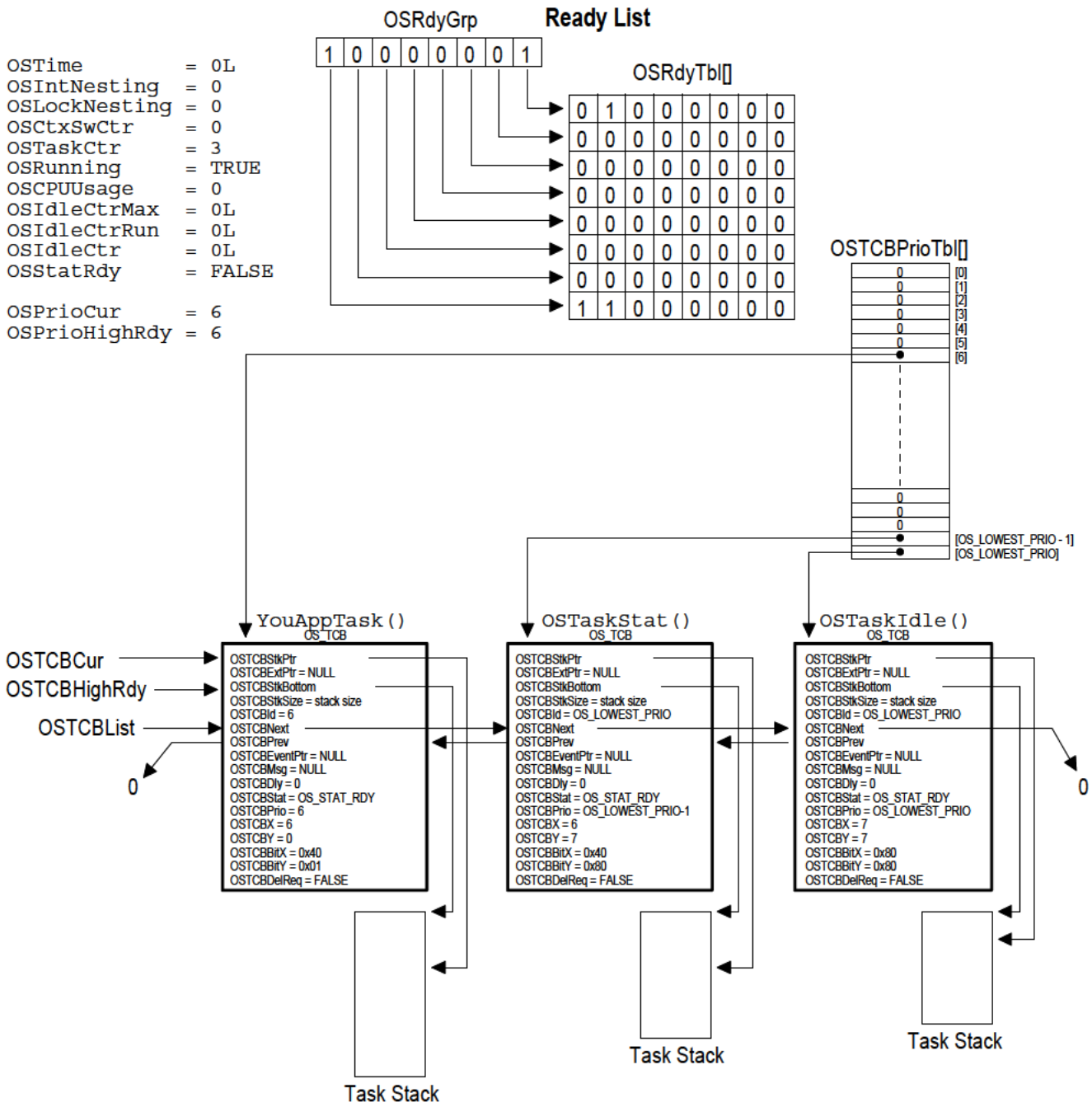

```
OSPrioCur     = 0
OSPrioHighRdy = 0
OSTCBCur      = NULL
OSTCBHighRdy  = NULL
OSTime        = 0L
OSIntNesting  = 0
OSLockNesting = 0
OSCtxSwCtr    = 0
OSTaskCtr     = 2
OSRunning     = FALSE
OSCPUUsage    = 0
OSIdleCtrMax  = 0L
OSIdleCtrRun  = 0L
OSIdleCtr     = 0L
OSStatRdy     = FALSE
```

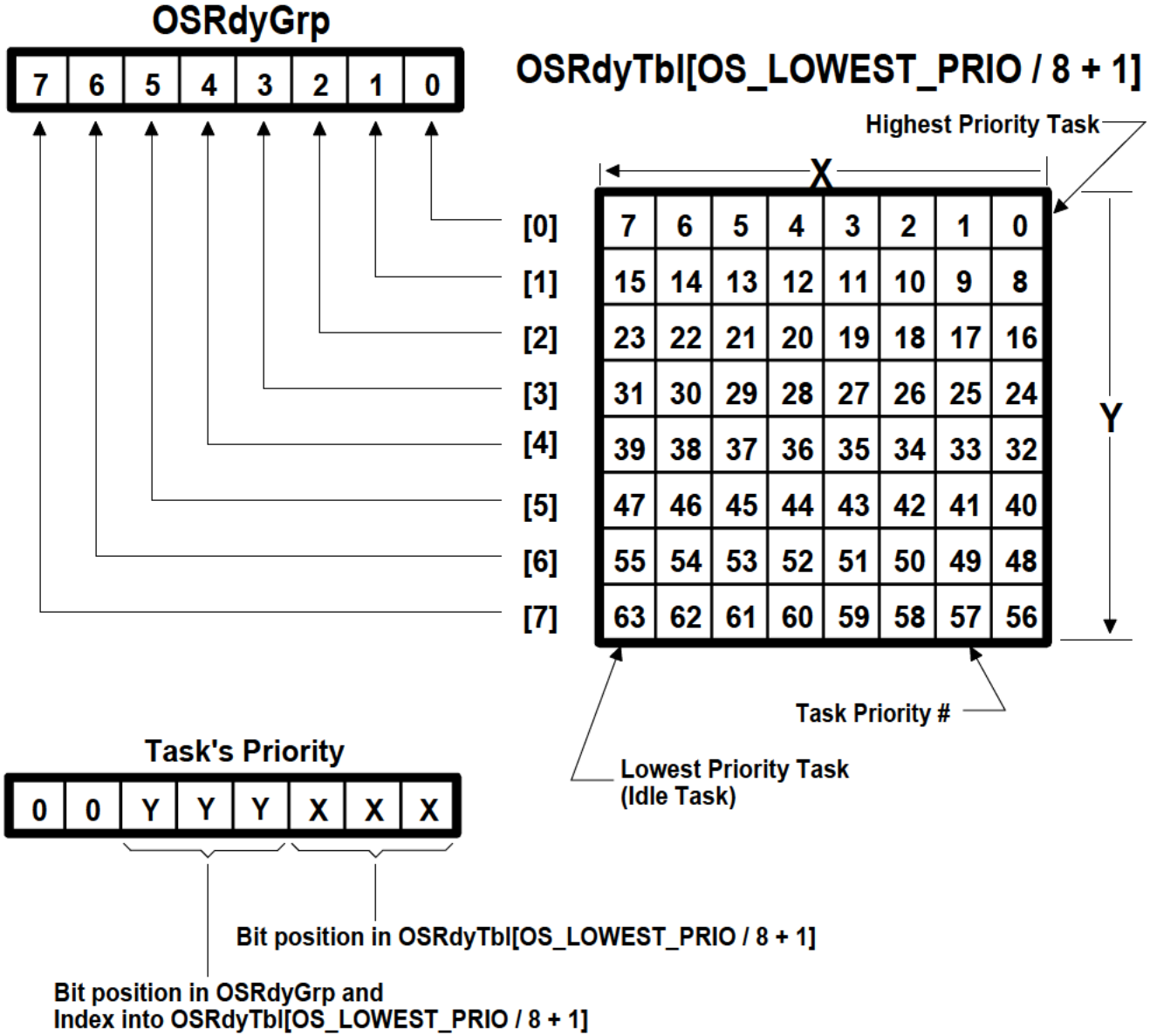
Starting uC/OS-2

```
void main (void)
{
    OSInit();          /* Initialize uC/OS-II                    */
    .
    Create at least 1 task using either OSTaskCreate() or OSTaskCreateExt();
    .
    OSStart();         /* Start multitasking!  OSStart() will not return */
}
```

```
void OSStart (void)
{
    INT8U y;
    INT8U x;
    if (OSRunning == FALSE) {
        y             = OSUnMapTbl[OSRdyGrp];
        x             = OSUnMapTbl[OSRdyTbl[y]];
        OSPrioHighRdy = (INT8U)((y << 3) + x);
        OSPrioCur     = OSPrioHighRdy;
        OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
        OSTCBCur      = OSTCBHighRdy;
        OSStartHighRdy();
    }
}
```

Start the highest-priority ready task


OSTime = 0L
OSIntNesting = 0
OSLockNesting = 0
OSCtxSwCtr = 0
OSTaskCtr = 3
OSRunning = TRUE
OSCPUUsage = 0
OSIdleCtrMax = 0L
OSIdleCtrRun = 0L
OSIdleCtr = 0L
OSStatRdy = FALSE
OSPrioCur = 6
OSPrioHighRdy = 6
OSRdyGrp
Ready List
OSRdyTbl[]
OSTCBPrioTbl[]
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[OS_LOWEST_PRIO - 1]
[OS_LOWEST_PRIO]
YouAppTask()
OSTaskStat()
OSTaskIdle()
OS_TCB
OSTCBCur
OSTCBHighRdy
OSTCBList
OSTCBStkPtr
OSTCBExtPtr = NULL
OSTCBStkBottom
OSTCBStkSize = stack size
OSTCBId = 6
OSTCBId = OS_LOWEST_PRIO
OSTCBNext
OSTCBPrev
OSTCBEventPtr = NULL
OSTCBMsg = NULL
OSTCBDly = 0
OSTCBStat = OS_STAT_RDY
OSTCBPrio = 6
OSTCBPrio = OS_LOWEST_PRIO-1
OSTCBPrio = OS_LOWEST_PRIO
OSTCBX = 6
OSTCBX = 7
OSTCBY = 0
OSTCBY = 7
OSTCBBitX = 0x40
OSTCBBitX = 0x80
OSTCBBitY = 0x01
OSTCBBitY = 0x80
OSTCBDelReq = FALSE
0
Task Stack
Task Stack
Task Stack


OSRdyGrp
7 6 5 4 3 2 1 0
OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]
Highest Priority Task
X
Y
[0] 7 6 5 4 3 2 1 0
[1] 15 14 13 12 11 10 9 8
[2] 23 22 21 20 19 18 17 16
[3] 31 30 29 28 27 26 25 24
[4] 39 38 37 36 35 34 33 32
[5] 47 46 45 44 43 42 41 40
[6] 55 54 53 52 51 50 49 48
[7] 63 62 61 60 59 58 57 56
Task Priority #
Lowest Priority Task
(Idle Task)
Task's Priority
0 0 Y Y Y X X X
Bit position in OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]
Bit position in OSRdyGrp and
Index into OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]

OSMapTbl

Index	Bit mask (Binary)
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

Bit 0 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[0]** is 1.
Bit 1 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[1]** is 1.
Bit 2 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[2]** is 1.
Bit 3 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[3]** is 1.
Bit 4 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[4]** is 1.
Bit 5 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[5]** is 1.
Bit 6 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[6]** is 1.
Bit 7 in **OSRdyGrp** is 1 when any bit in **OSRdyTbl[7]** is 1.

- Make a task ready to run:

```
OSRdyGrp               |= OSMapTbl[prio >> 3];
OSRdyTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

- Remove a task from the ready list:

```
if ((OSRdyTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0)
    OSRdyGrp &= ~OSMapTbl[prio >> 3];
```

What does this code do?

```
INT8U  const  OSUnMapTbl[] = {
   0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x00 to 0x0F                    */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x10 to 0x1F                   */
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x20 to 0x2F                    */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x30 to 0x3F                    */
   6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x40 to 0x4F                    */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x50 to 0x5F                    */
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x60 to 0x6F                    */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x70 to 0x7F                    */
   7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x80 to 0x8F                    */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x90 to 0x9F                    */
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xA0 to 0xAF                    */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xB0 to 0xBF                    */
   6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xC0 to 0xCF                    */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xD0 to 0xDF                    */
   5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xE0 to 0xEF                    */
   4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0     /* 0xF0 to 0xFF                    */
};
```

- Finding the highest-priority task ready to run:

```
y    = OSUnMapTbl[OSRdyGrp];
x    = OSUnMapTbl[OSRdyTbl[y]];
prio = (y << 3) + x;
```

This matrix is used to locate the first LSB which is '1', by given a value.

For example, if 00110010 is given, then '1' is returned.

Summary

- The study of OS concept, we learn something:
  - What a task is?
  - What a interrupt is?
  - How the scheduler works, especially on detailed operations done for context switching
  - How interrupts are serviced in uC/OS-II
  - The initialization and starting of µC/OS-II

See you next class!