uC/OS-II Part 3: uC/OS-II: Kernel Structure

Prof. Li-Pin Chang
Embedded Software and Storage Lab@NYCU

Objectives

- To understand what a task is
- To learn how uC/OS-2 manages tasks
  - Essential data structures
  - Context switch
- To know how an ISR works

The uC/OS-2 File Structure

Application Code (Your Code!)

Processor independent implementations

- Scheduling policy
- Event flags
- Semaphores
- Mailboxes
- Event queues
- Task management
- Time management
- Memory management

Application Specific Configurations

OS_CFG.H

- Max # of tasks
- Max Queue length
- …

uC/OS-2 port for processor specific codes

Tick ISR
Cxtsw

Software

Hardware

CPU

Timer

Critical Section Concept

Critical Sections

- A critical section is a piece of code that is not safe from race condition; also known as non-entrant code
- Use semaphores or mutex locks to protect critical sections
  - A good approach to task-task race in user code, but too heavy-duty for kernel critical sections
  - Kernel critical sections are often short and context switch is not desirable (or not allowed)

Critical Sections

- Task-task race in kernel code
  – Critical sections in kernels are usually short, and semaphores/mutexes are too expensive
- Task-ISR race
  – ISR cannot call blocking calls (e.g., OSSemPend), the reasons why are:
  – Potential deadlock, as the interrupted task itself maybe part of the handling of the current interrupt
  – Unexpected long delay on the interrupted task

Critical Sections

- By disabling interrupts, task preemption is masked
- Suitable to
  - Kernel code; critical sections in kernel are short
  - Task-ISR race; because ISR cannot use blocking methods of critical section (e.g., a semaphore)
- Notice: this does not work in multiprocessor systems; use spinlocks instead

Critical Sections

- The interrupt latency is part of the specification of an RTOS
  - Interrupt disabling should be as short as possible to avoid response degradation
- Interrupt disabling must be used carefully:
  - E.g., if OSTimeDly() is called with interrupt disabled, the machine may hang as the tick interrupt is blocked
  - A basic rule: do not call system services when interrupt is disabled (or in an ISR)

```
{
    .
    OS_ENTER_CRITICAL();
    .      /* Critical Section */
    OS_EXIT_CRITICAL();
    .
}
```

Critical Sections

- The states of the processor must be carefully maintained in nested calls of OS_ENTER_CRITICAL() / OS_EXIT_CRITICAL()
- There are different implementations for the maintenance of process states:
  - Interrupt enabling/disabling instructions
  - Interrupt status save/restore onto/from stacks

Critical Sections

- OS_CRITICAL_METHOD=2
- Processor Status Word (PSW) can be saved/restored onto/from stacks
  – PSW's of nested interrupt enable/disable operations can be exactly recorded in stacks

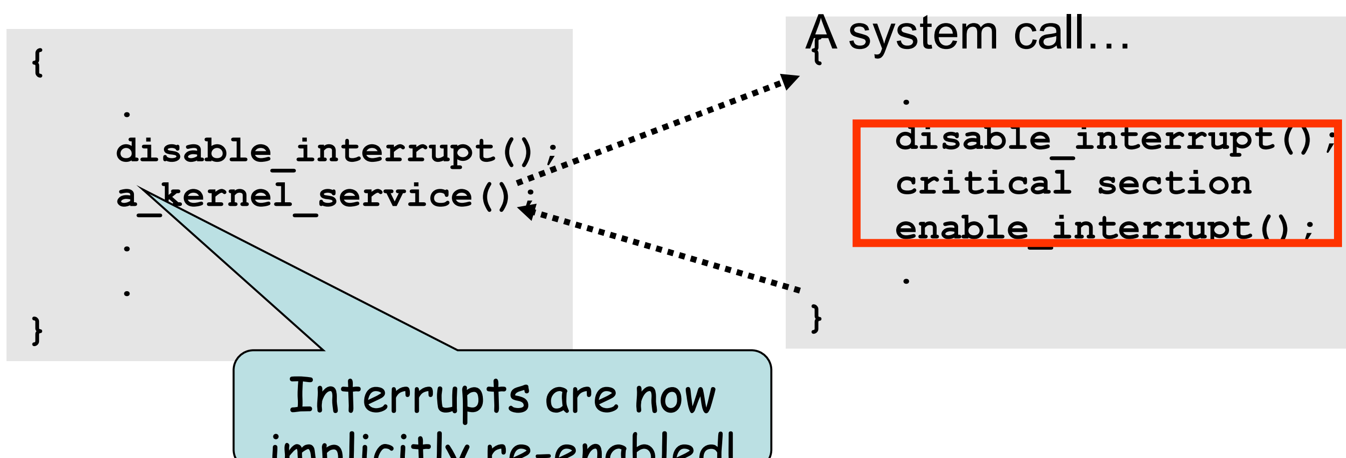
```
#define OS_ENTER_CRITICAL() \
         asm("PUSH    PSW");
         asm("DI");


#define OS_EXIT_CRITICAL() \
         asm("POP     PSW");
```

← x86 port uses this method

Critical Sections

- Why use method 2?
- OS_CRITICAL_METHOD=1
  - does not involve the task stack
- Interrupt is immediately enabled once OS_CRITICAL_EXIT() is called, not the outermost OS_CRITICAL_EXIT()



Task Structure

Tasks

- A task is an active entity that conducts computation
- In real-time systems, a periodic task is typically an infinite loop

```
void YourTask (void *pdata)                              (1)
{
   for (;;) {                                            (2)
      /* USER CODE */
      Call one of uC/OS-II's services:
      OSMboxPend();
      OSQPend();
      OSSemPend();
      OSTaskDel(OS_PRIO_SELF);
      OSTaskSuspend(OS_PRIO_SELF);
      OSTimeDly();
      OSTimeDlyHMSM();
      /* USER CODE */
   }
}
```

Delay itself for next event/period, so that other tasks can run.

Tasks

- uC/OS-2 can have up to 64 priorities
  - Each task has an unique priority
  - Priorities 63 and 62 are reserved (idle, stat)
- Insufficient number of priority will damage the schedulability of a real-time scheduler
  - Fortunately, # of tasks in embedded systems is usually not large so unique task priorities are possible

Tasks

- A task is created by OSTaskCreate() or OSTaskCreateExt()
- A task can change its priority using OSTaskChangePrio()
- A task can delete itself when done

```
void YourTask (void *pdata)
{
   /* USER CODE */
   OSTaskDel(OS_PRIO_SELF);
}
```

The priority of the current task

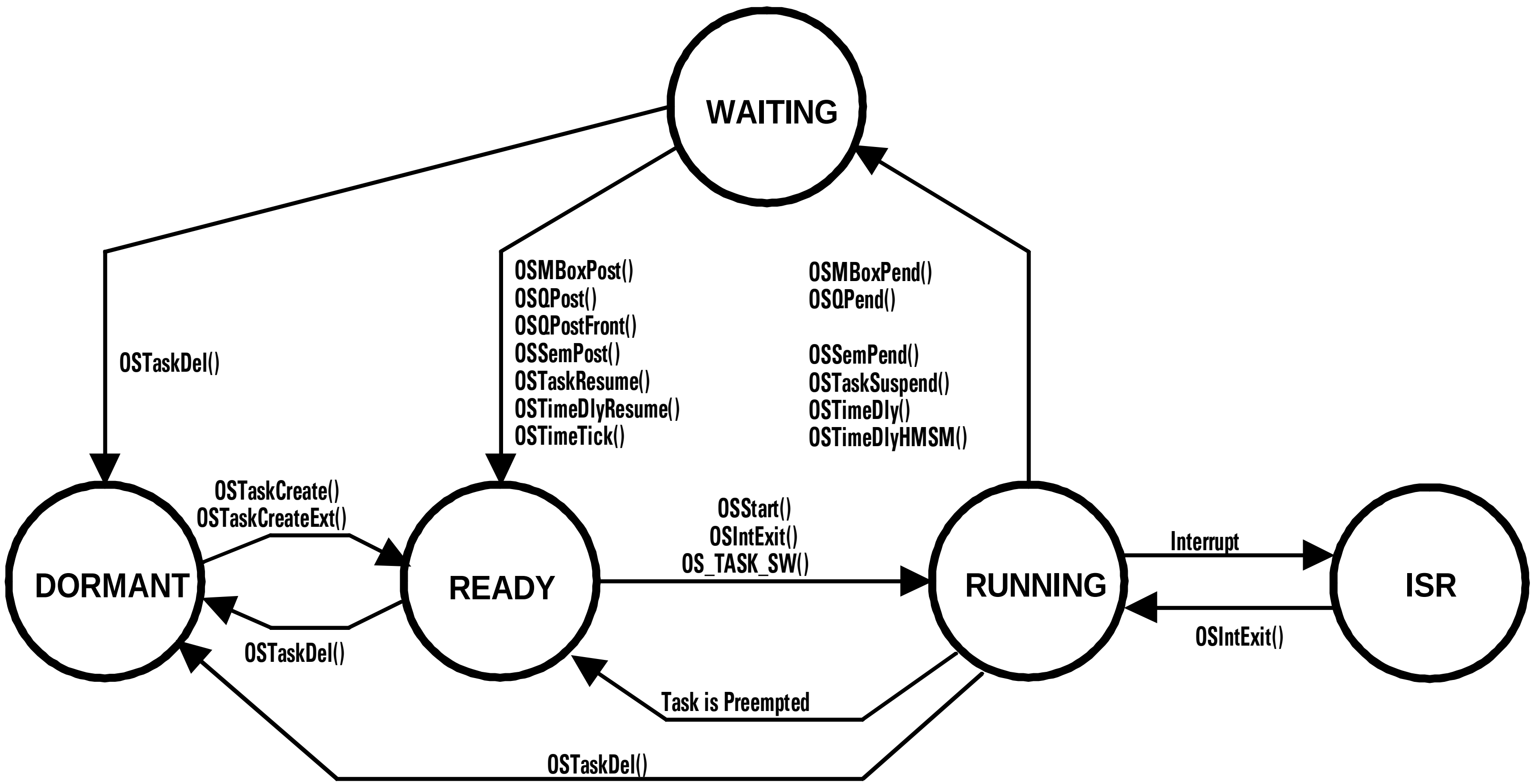
Task States

- Dormant: Procedures residing in RAM/ROM is not a task yet unless you call OSTaskCreate() to create one to execute them
- Ready: A ready task can be scheduled to run on the CPU. It must not be waiting
  - A new task is a ready one
- Running: A ready task is running on the CPU
  - There must be only one running task
  - The task running might be preempted and then become ready

Task States

- Waiting: A waiting task is waiting for some events to occur
  - Timer expiration, signaling of semaphores, messages in mailboxes, etc.
- ISR: A task in this state has been interrupted and an ISR is in execution
  - The task stack is being used by the ISR

Task States



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()

Task States

- A task can delay itself by calling OSTimeDly() or OSTimeDlyHMSM().
  - The task is placed in the waiting state.
  - The task will be made ready by OSTimeTick().
    - It is the clock ISR, you don’t have to call it explicitly from your code.
- A task can wait for an event by OSFlagPend(), OSSemPend(), OSMboxPend(), or OSQPend().
  - The task remains waiting until the occurrence of the desired event. (or timeout)
- The running task is always preempted by ISR’s, unless interrupts are disabled.
  - ISR’s could make one or more tasks ready by signaling events.
  - On the return of an ISR, the scheduler will check if rescheduling is needed.
- Once new tasks become ready, the next highest priority ready task is scheduled to run (due to occurrences of events, timer expirations).
- If no task is running and all tasks are not in the ready state, the idle task executes.

Task States

- A task can delay itself by calling OSTimeDly() or OSTimeDlyHMSM().
  - The task is placed in the waiting state
  - The task will later be ready when the timer expires
    - The clock ISR OSTimeTick() decrements the timer
- A task can wait for an event by OSFlagPend(), OSSemPend(), OSMboxPend(), or OSQPend()
  - The task remains waiting until the desired event is signaled (or timeout)

Task Control Blocks (TCB)

- A TCB is a per-task data structure
  - In-use TCBs are in the TCB list
  - Free TCB’s are in a free list
- TCBs are updated during context switches
  - Task priority, delay counter, event to wait, stack pointer
  - CPU registers are saved to the stack, not TCB

```
typedef struct os_tcb {
    OS_STK         *OSTCBStkPtr;
#if OS_TASK_CREATE_EXT_EN
    void           *OSTCBExtPtr;
    OS_STK         *OSTCBStkBottom;
    INT32U          OSTCBStkSize;
    INT16U          OSTCBOpt;
    INT16U          OSTCBId;
#endif
    struct os_tcb *OSTCBNext;
    struct os_tcb *OSTCBPrev;
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
    OS_EVENT       *OSTCBEventPtr;
#endif
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    void           *OSTCBMsg;
#endif

    INT16U          OSTCBDly;
    INT8U           OSTCBStat;
    INT8U           OSTCBPrio;
    INT8U           OSTCBX;
    INT8U           OSTCBY;
    INT8U           OSTCBBitX;
    INT8U           OSTCBBitY;
#if OS_TASK_DEL_EN
    BOOLEAN         OSTCBDelReq;
#endif
} OS_TCB;
```

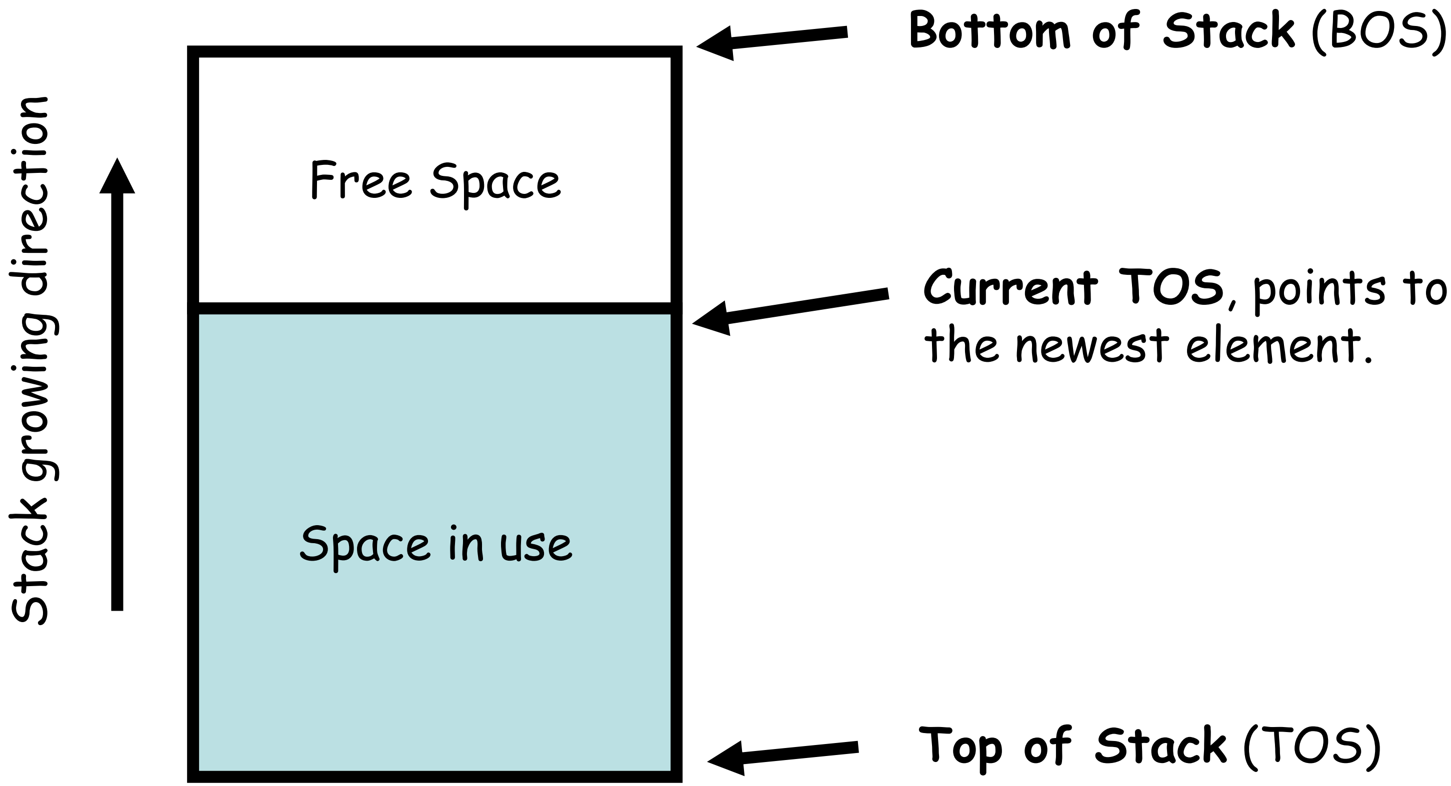
Task Control Blocks (TCB)

- **.OSTCBStkPtr** points to the current TOS for the task
  - It is the first entry of TCB so that it can be accessed directly via assembly language (offset=0)
- **.OSTCBExtPtr** is a pointer to a user-definable task control block extension.
  - Set OS_TASK_CREATE_EXT_EN to 1.
  - The pointer is set when OSTaskCreateExt( ) is called

Task Control Blocks (TCB)

- **.OSTCBStkBottom** is a pointer to the bottom of the task's stack
- **.OSTCBStkSize** holds the size of the stack in number of elements instead of bytes
  - The element size is the macro OS_STK (16 bits in x86)
  - Total stack size is OSTCBStkSize*OS_STK bytes
  - .OSTCBStkBottom and .OSTCBStkSize are used to check stack

Task Control Blocks (TCB)



Task Control Blocks (TCB)

- **.OSTCBOpt** holds “options” that can be passed to OSTaskCreateExt( )
  - OS_TASK_OPT_STK_CHK: stack checking is enable for the task being created.
  - OS_TASK_OPT_STK_CLR: indicates that the stack needs to be cleared when the task is created.
  - OS_TASK_OPT_SAVE_FP: tells OSTaskCreateExt( ) that the task will be doing floating-point computations. Floating point processor’s registers must be saved to the stack on context-switches.
- **.OSTCBId**: holds an identifier for the task.
- **.OSTCBNext** and **.OSTCBPrev** are used to double link OS_TCBs
- **.OSTCBEVEventPtr** is pointer to an event control block.
- **.OSTCBMsg** is a pointer to a message that is sent to a task.
- **.OSTCBFlagNode** is a pointer to a flagnode.
- **.OSTCBFlagsRdy** maintains which event flags make the task ready.
- **.OSTCBDly** is used when:
  - a task needs to be delayed for a certain number of clock ticks, or
  - a task needs to pend for an event to occur with a timeout.
- **.OSTCBStat** contains the state of the task. (0 is ready to run)
- **.OSTCBPrio** contains the task priority.

Task Control Blocks (TCB)

- **.OSTCBX .OSTCBY .OSTCBBitX** and **.OSTCBBitY**
  - They are used to accelerate the process of making a task ready to run or make a task wait for an event.

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

- **.OSTCBDelReq** is boolean used to indicate whether or not a task request that the current task to be deleted.

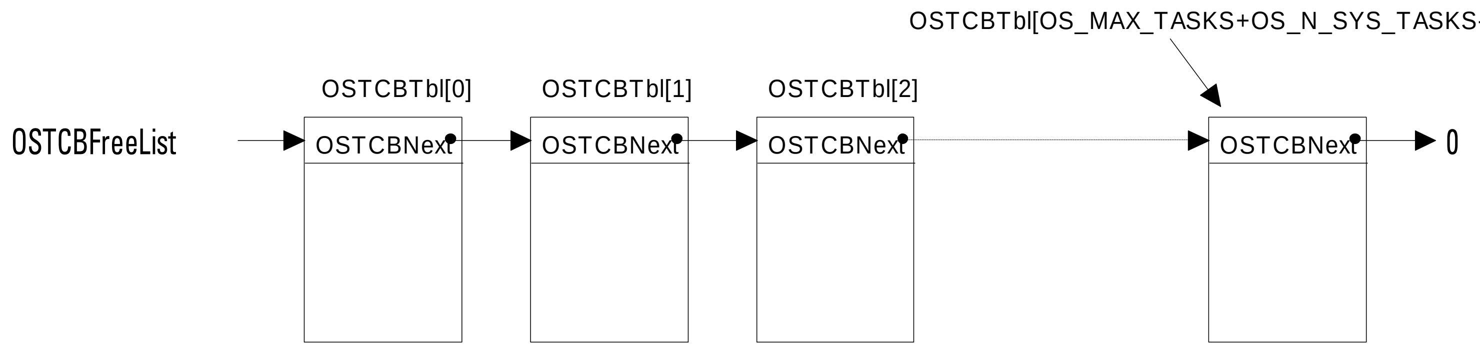
- OS_MAX_TASKS is specified in OS_CFG.H
  - # OS_TCBs allocated by µC/OS-II

- **OSTCBTbl**[ ] : an array holding all OS_TCBs

- When µC/OS-II is initialized, all OS_TCBs in the table are linked in a singly linked list of free OS_TCBs

Task Control Blocks (TCB)

- A task receives/frees its OS_TCB from/to the free list
- An OS_TCB is initialized by the function OS_TCBInit(), which is called by OSTaskCreate().



```
INT8U  OS_TCBInit (INT8U prio, OS_STK *ptos, OS_STK *pbos, INT16U id, INT32U stk_size, void *pext, INT16U
opt)
{
#if OS_CRITICAL_METHOD == 3                                /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr;
#endif
    OS_TCB    *ptcb;


    OS_ENTER_CRITICAL();
    ptcb = OSTCBFreeList;                                  /* Get a free TCB from the free TCB list    */
    if (ptcb != (OS_TCB *)0) {
        OSTCBFreeList        = ptcb->OSTCBNext;            /* Update pointer to free TCB list          */
        OS_EXIT_CRITICAL();
        ptcb->OSTCBStkPtr    = ptos;                       /* Load Stack pointer in TCB                */
        ptcb->OSTCBPrio      = (INT8U)prio;                /* Load task priority into TCB              */
        ptcb->OSTCBStat      = OS_STAT_RDY;                /* Task is ready to run                     */
        ptcb->OSTCBDly       = 0;                          /* Task is not delayed                      */

#if OS_TASK_CREATE_EXT_EN > 0
        ptcb->OSTCBExtPtr    = pext;                       /* Store pointer to TCB extension           */
        ptcb->OSTCBStkSize   = stk_size;                   /* Store stack size                         */
        ptcb->OSTCBStkBottom = pbos;                       /* Store pointer to bottom of stack         */
        ptcb->OSTCBOpt       = opt;                        /* Store task options                       */
        ptcb->OSTCBId        = id;                         /* Store task ID                            */
#else
        pext                 = pext;                       /* Prevent compiler warning if not used     */
        stk_size             = stk_size;
        pbos                 = pbos;
        opt                  = opt;
        id                   = id;
#endif

#if OS_TASK_DEL_EN > 0
        ptcb->OSTCBDelReq    = OS_NO_ERR;
#endif

        ptcb->OSTCBY         = prio >> 3;                  /* Pre-compute X, Y, BitX and BitY          */
        ptcb->OSTCBBitY      = OSMapTbl[ptcb->OSTCBY];
        ptcb->OSTCBX         = prio & 0x07;
        ptcb->OSTCBBitX      = OSMapTbl[ptcb->OSTCBX];
```

Get a free TCB from the free list

```
#if OS_EVENT_EN > 0
        ptcb->OSTCBEventPtr   = (OS_EVENT *)0;                /* Task is not pending on an event          */
#endif

#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0) && (OS_TASK_DEL_EN > 0)
        ptcb->OSTCBFlagNode   = (OS_FLAG_NODE *)0;            /* Task is not pending on an event flag     */
#endif

#if (OS_MBOX_EN > 0) || ((OS_Q_EN > 0) && (OS_MAX_QS > 0))
        ptcb->OSTCBMsg        = (void *)0;                    /* No message received                      */
#endif

#if OS_VERSION >= 204
        OSTCBInitHook(ptcb);
#endif

        OSTaskCreateHook(ptcb);                               /* Call user defined hook                   */

        OS_ENTER_CRITICAL();
        OSTCBPrioTbl[prio] = ptcb;
        ptcb->OSTCBNext    = OSTCBList;                       /* Link into TCB chain                      */
        ptcb->OSTCBPrev    = (OS_TCB *)0;
        if (OSTCBList != (OS_TCB *)0) {
            OSTCBList->OSTCBPrev = ptcb;
        }
        OSTCBList              = ptcb;
        OSRdyGrp              |= ptcb->OSTCBBitY;             /* Make task ready to run                   */
        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        OS_EXIT_CRITICAL();
        return (OS_NO_ERR);
    }
    OS_EXIT_CRITICAL();
    return (OS_NO_MORE_TCB);
}
```

User-defined hook is called here.

Priority table

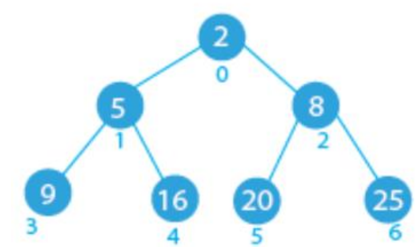
TCB list

Ready list

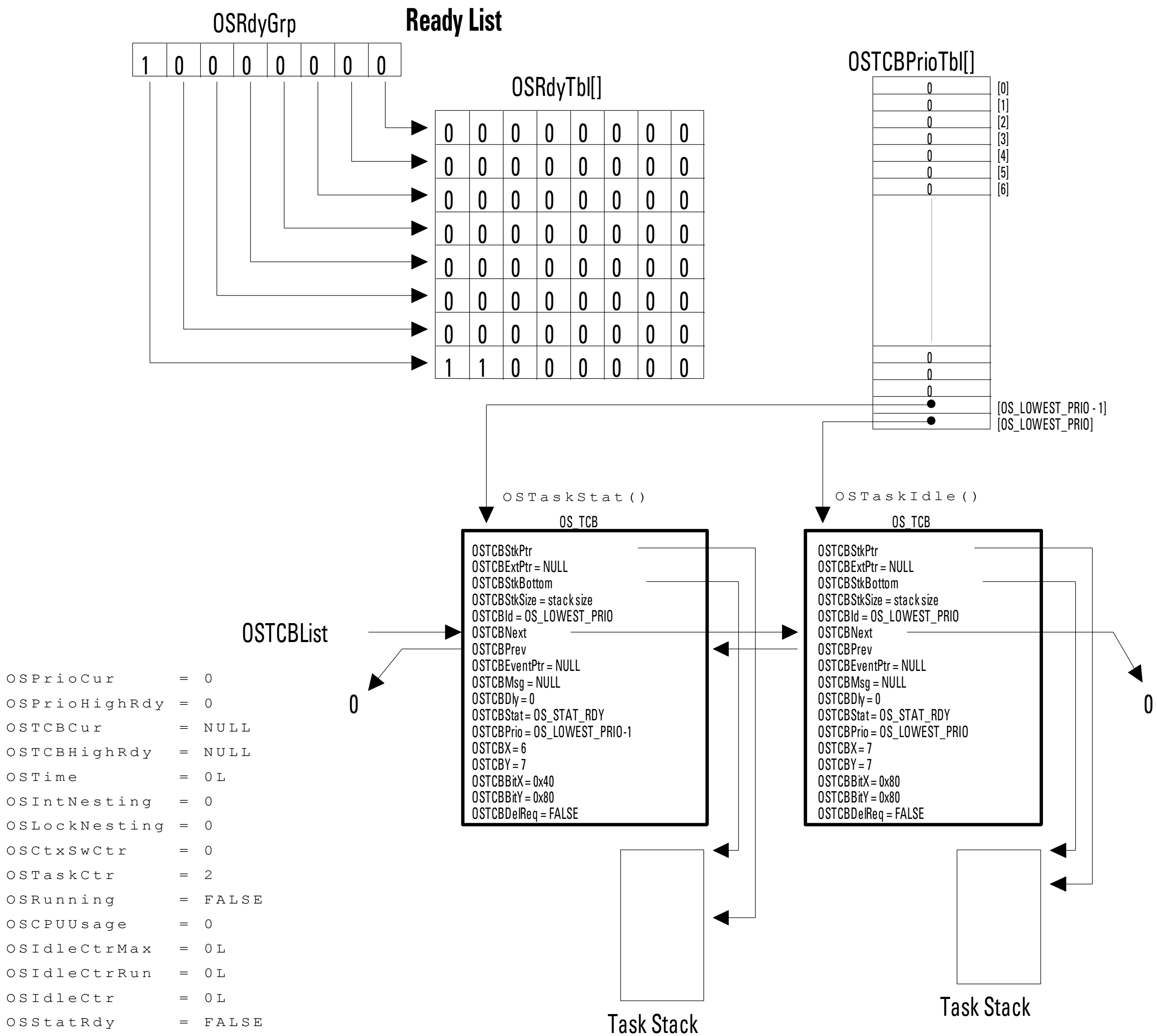
Context Switch and Scheduling

Ready List

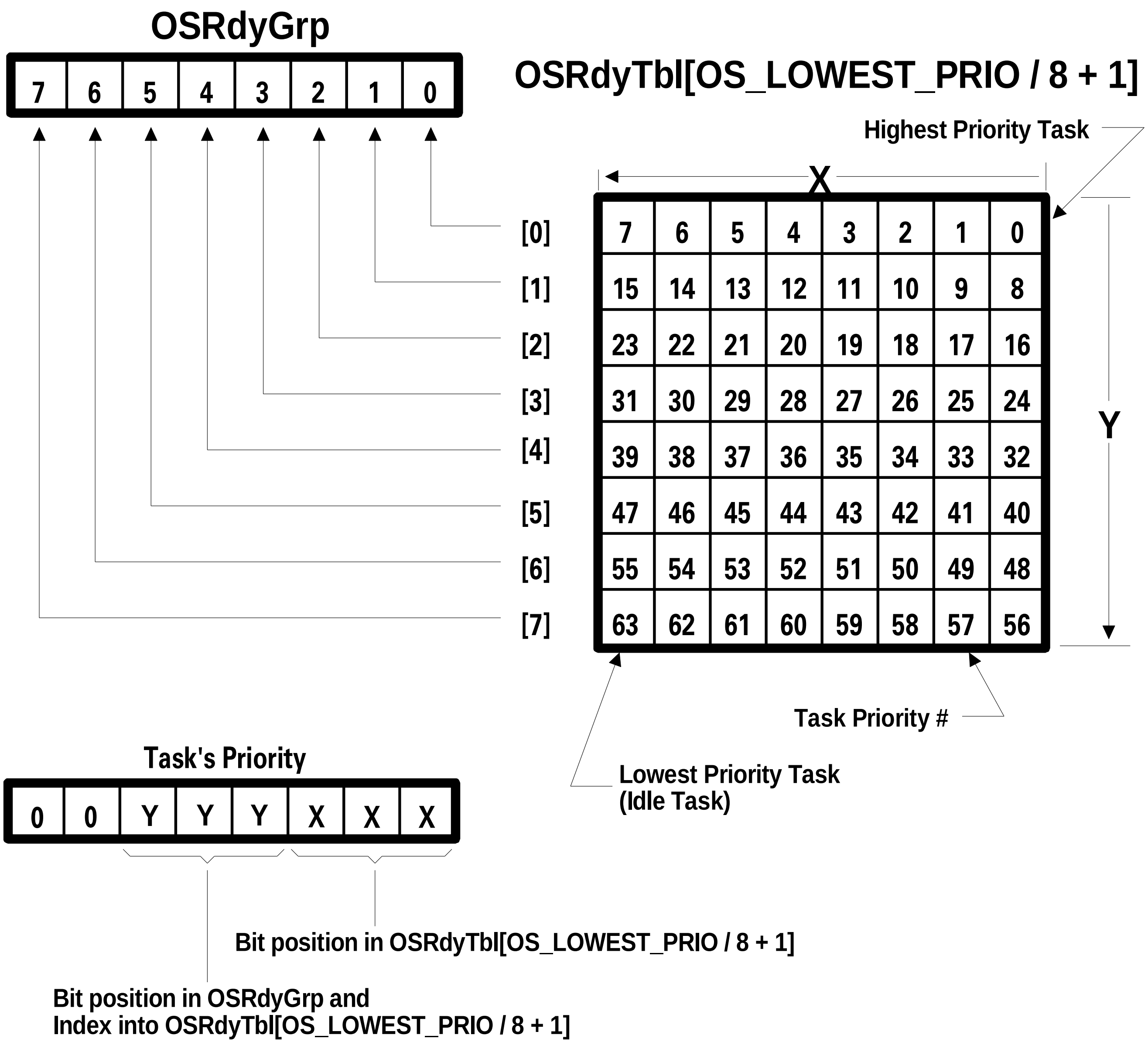
- Ready list is actually a bitmap that indicates which tasks are ready
- Design options
  - A linear list takes a O(n) time to locate the highest-priority ready task
  - A max heap takes a O(log n) time to find and delete the max item
  - It takes only O(1) using the bitmap



Min Heap



Ready List
OSRdyGrp
1 0 0 0 0 0 0 0
OSRdyTbl[]
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
OSTCBPrioTbl[]
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[OS_LOWEST_PRIO - 1]
[OS_LOWEST_PRIO]
OSTaskStat()
OS_TCB
OSTCBStkPtr
OSTCBExtPtr = NULL
OSTCBStkBottom
OSTCBStkSize = stack size
OSTCBId = OS_LOWEST_PRIO
OSTCBNext
OSTCBPrev
OSTCBEventPtr = NULL
OSTCBMsg = NULL
OSTCBDly = 0
OSTCBStat = OS_STAT_RDY
OSTCBPrio = OS_LOWEST_PRIO-1
OSTCBX = 6
OSTCBY = 7
OSTCBBitX = 0x40
OSTCBBitY = 0x80
OSTCBDelReq = FALSE
OSTaskIdle()
OS_TCB
OSTCBStkPtr
OSTCBExtPtr = NULL
OSTCBStkBottom
OSTCBStkSize = stack size
OSTCBId = OS_LOWEST_PRIO
OSTCBNext
OSTCBPrev
OSTCBEventPtr = NULL
OSTCBMsg = NULL
OSTCBDly = 0
OSTCBStat = OS_STAT_RDY
OSTCBPrio = OS_LOWEST_PRIO
OSTCBX = 7
OSTCBY = 7
OSTCBBitX = 0x80
OSTCBBitY = 0x80
OSTCBDelReq = FALSE
OSTCBList
0
0
Task Stack
Task Stack
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


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:

```
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     */
};
```

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.

•Finding the highest-priority task ready to run:

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

Task Scheduling

- The scheduler always schedules the highest-priority ready task to run
- Task-level scheduling is done through OS_Sched()
  - When the running task gives up the CPU
- ISR-level scheduling is done during OSIntExit()
  - When the running task is preempted

```
void OS_Sched (void)
{
    INT8U y;
    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) {                          (1)
        y             = OSUnMapTbl[OSRdyGrp];                           (2)
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);   (2)
        if (OSPrioHighRdy != OSPrioCur) {                               (3)
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];                 (4)
            OSCtxSwCtr++;                                               (5)
            OS_TASK_SW();                                               (6)
        }
    }
    OS_EXIT_CRITICAL();
}
```

(1) Rescheduling will not be performed if the scheduler is locked or some interrupt is currently serviced (why?).

(2) Find the highest-priority ready task.

(3) If it is not the current task, then

(4) ~(6) Perform a context-switch.

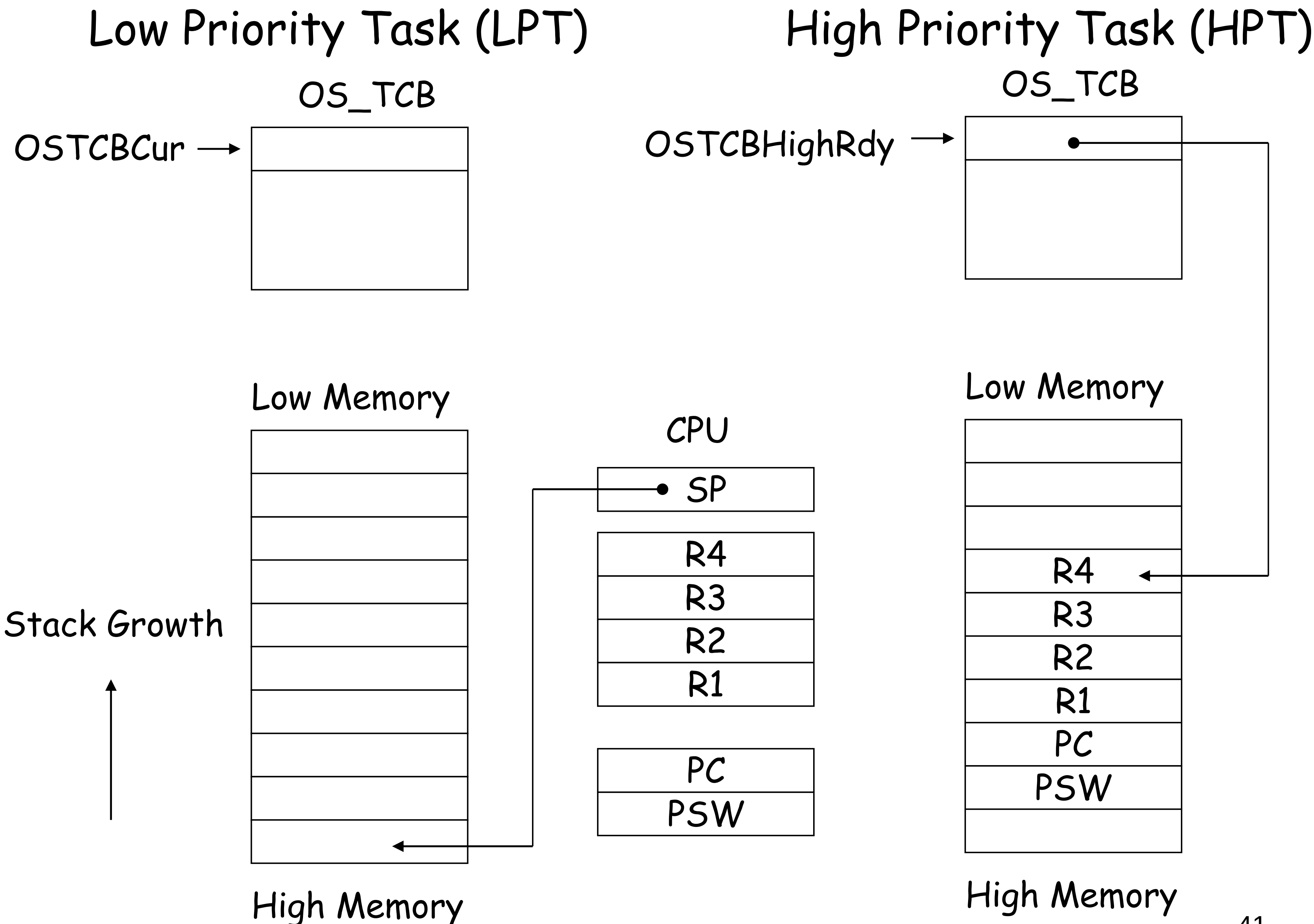
OS_TASK_SW() is a macro: “asm int 0x80” that generates a software interrupt

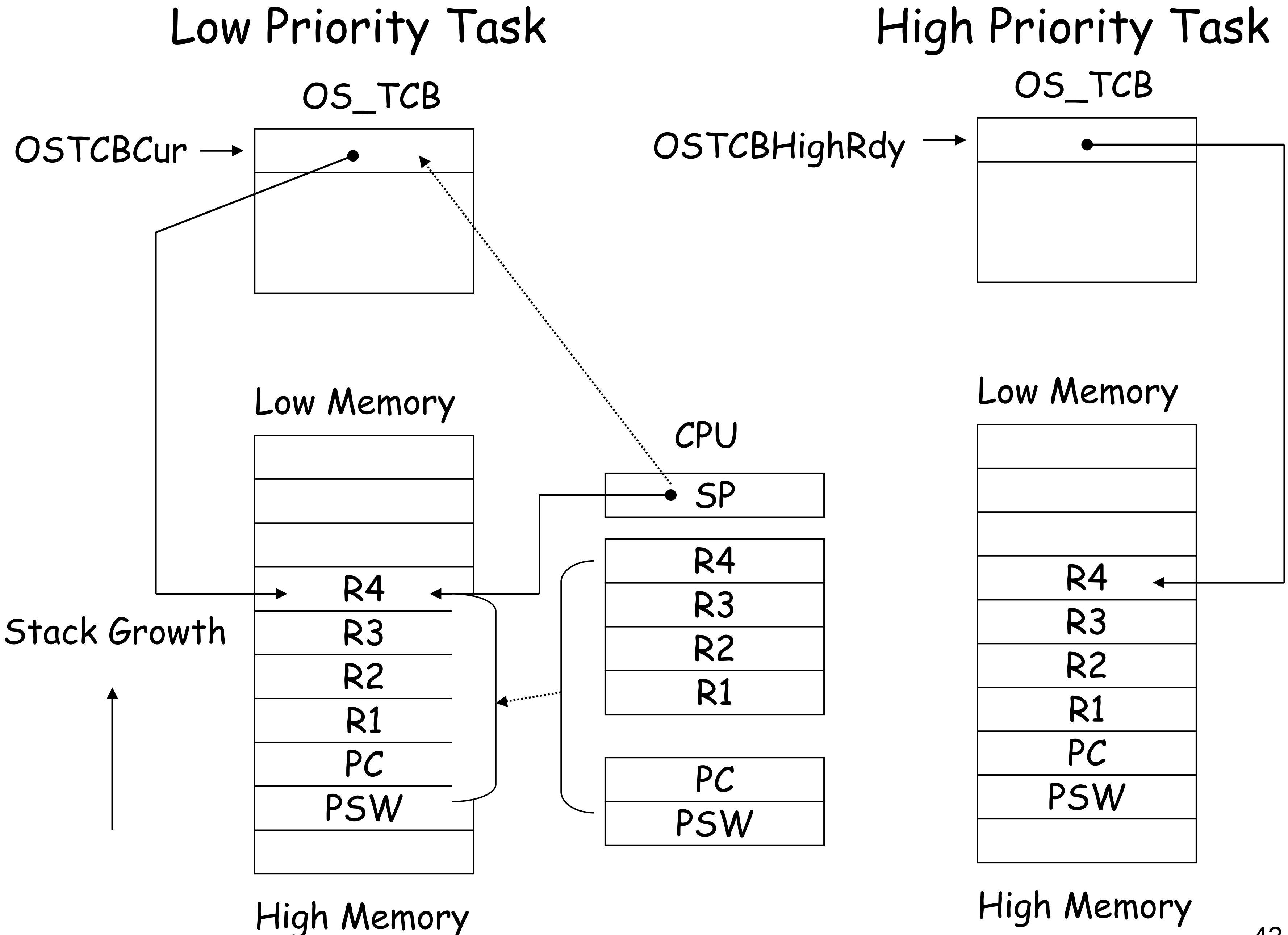
Task Scheduling

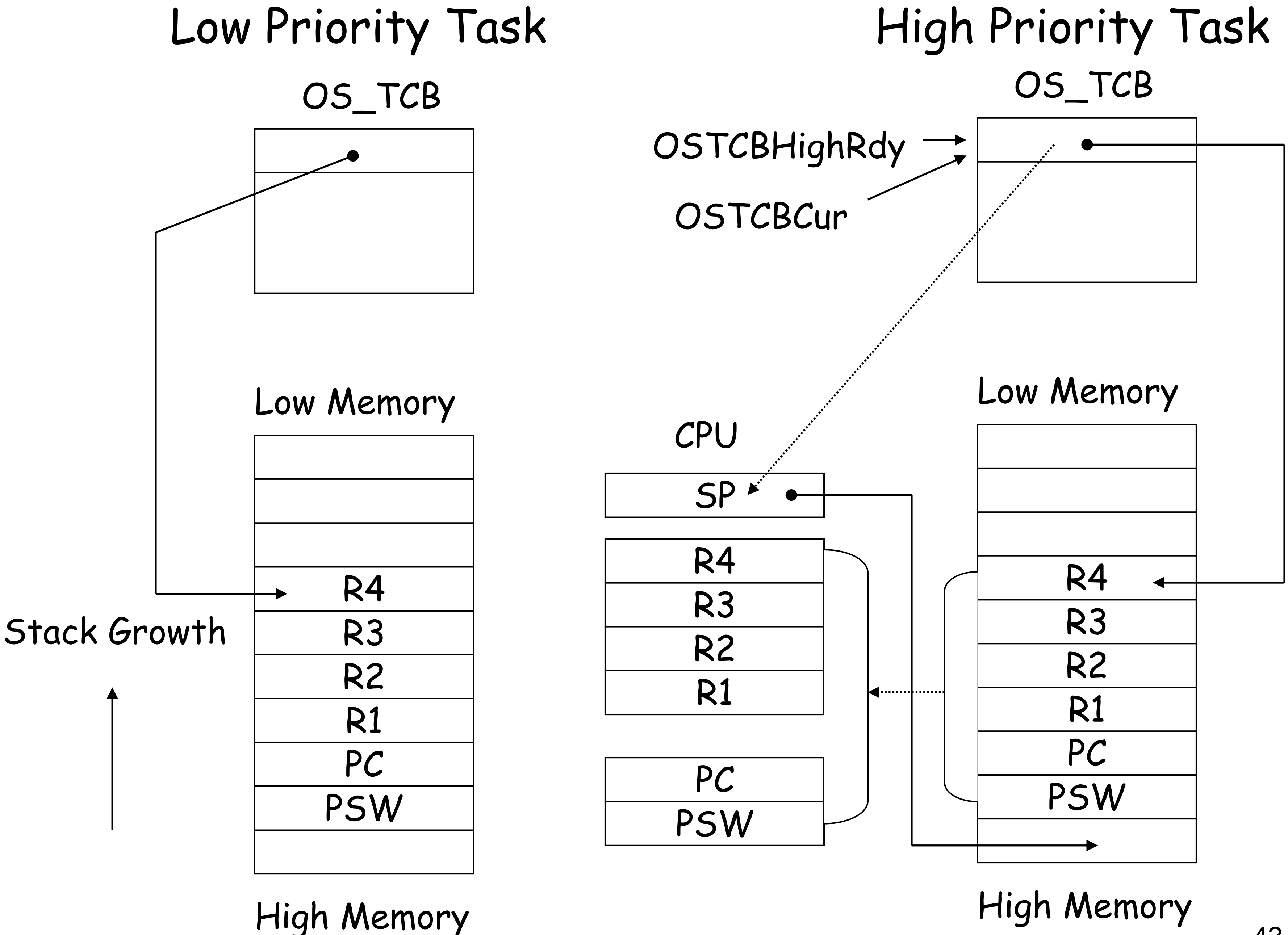
- Context switching must save all CPU registers and PSW of the LPT to its stack, and then restore the CPU registers and PSW of the HPT from its stack
- Written in assembly
  - For efficiency
  - For direct access to registers and stack

Task-Level Context Switch

- Strictly speaking, context switches always happen on the way out of ISRs
  - When leaving the clock tick ISR
  - When leaving the cxtsw ISR
- How to perform cxtsw when a task voluntarily gives up the CPU (task-level cxtsw)?
  - There is no “interrupt” at this time, so generate one!
  - “INT 80h” in x86


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


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


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

Interrupt Handling

Interrupts under uC/OS-2

- ISRs in uC/OS-2 are written in assembly
- Check: how does the task stack looks like at Step 7?

(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) Upon entry of an ISR, all CPU registers must be saved in the interrupted task's stack

- As the execution of the ISR may alter the registers

(2) Increase the interrupt-nesting 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

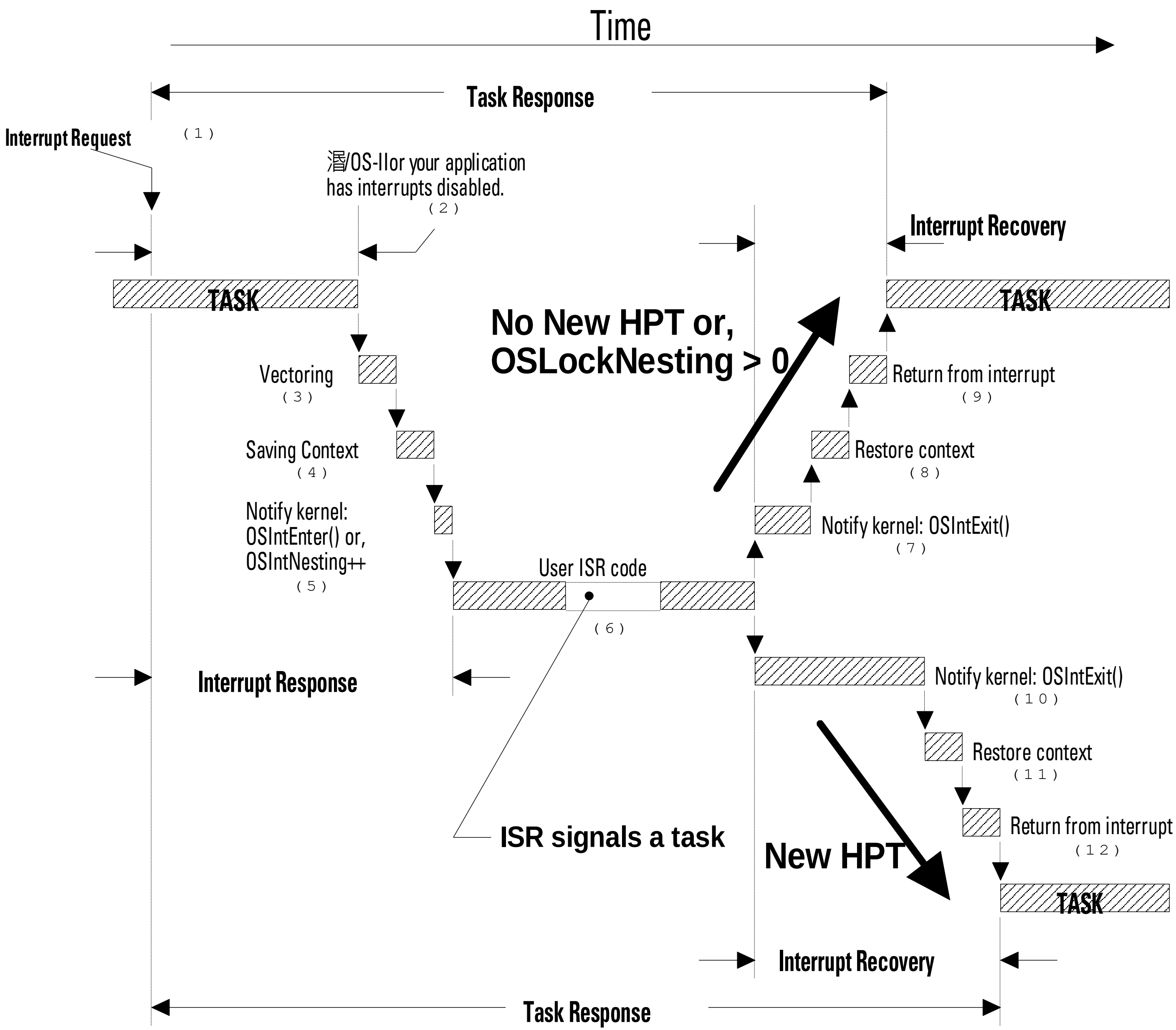
(8) Call OSIntExit(), which checks if we are in the inner-level of nested interrupts. If we are at the outmost level ISR, the scheduler is called

- Decrementing the Interrupt-nesting counter
- A potential context-switch might occur

(9) On the return to this point, several high-priority tasks may have been run by the CPU

- If OSIntExit() performs a context switch

(10) The CPU registers are restored from the stack and CPU execution returns to the interrupted instruction (of a task)


Time
Task Response
Interrupt Request
(1)
滑/OS-IIor your application
has interrupts disabled.
(2)
Interrupt Recovery
TASK
No New HPT or,
OSLockNesting > 0
TASK
Vectoring
(3)
Return from interrupt
(9)
Saving Context
(4)
Restore context
(8)
Notify kernel:
OSIntEnter() or,
OSIntNesting++
(5)
Notify kernel: OSIntExit()
(7)
User ISR code
(6)
Interrupt Response
Notify kernel: OSIntExit()
(10)
Restore context
(11)
ISR signals a task
Return from interrupt
(12)
New HPT
TASK
Interrupt Recovery
Task Response

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 we are already in an ISR

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

Interrupt-Level Task Scheduling

- OSIntExit() checks if a higher-priority task becomes ready
  - If so, perform context switch
- Task-level vs. Interrupt-level cxtsw
  - Task-level cxtsw actually “emulates” a interrupt-level cxtsw via a software interrupt

Clock Tick

- A timer is needed to keep track of time delays and timeouts
- You must install uC/OS-2 tick ISR after OSStart()
  - Do this in the startup task
- Tick ISR calls OSTimeTick()
- Clock tick ISR is also written in assembly

Clock Tick ISR Pseudo Code

```
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;
}
```

```
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.

Clock Tick

- OSTimeTick() is hardware independent
  - Called by OSTickISR, which is hardware dependent

- Linearly visiting all TCBs and decrementing delay
  - O(n) to progress 1 unit of time
  - O(1) to insert a new sleeping task
- Alternative: delta list (AKA timer list)
  - O(1) to progress 1 unit of time
  - O(n) to insert a new sleeping task

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

Locking and Unlocking the Scheduler

- OSSchedLock() prevent high-priority ready tasks from preempting the current task
  - Becoming non-preemptible scheduling
  - Interrupts are still recognized and processed
- OSSchedLock() and OSSchedUnlock() are used in pairs
- OSLockNesting keeps track of the number of OSSchedLock() has been called (how? why?)

Locking and Unlocking the Scheduler

- After calling OSSchedLock(), you must not call kernel services which might cause context switch, such as OSFlagPend(), OSMboxPend(), OSMutexPend(), OSQPend(), OSSemPend(), OSTaskSuspend(), OSTimeDly, OSTimeDlyHMSM() until OSLockNesting == 0. Or the system may be locked up
- To lock the scheduler is to prevent from task-task race conditions while interrupts are still handled

OSSchedLock()

```
void  OSSchedLock (void)
{
#if OS_CRITICAL_METHOD == 3       /* Allocate storage for CPU status register  */
    OS_CPU_SR  cpu_sr;
#endif


    if (OSRunning == TRUE) {      /* Make sure multitasking is running            */
        OS_ENTER_CRITICAL();
        if (OSLockNesting < 255) {/* Prevent OSLockNesting from wrapping back to 0*/
            OSLockNesting++;      /* Increment lock nesting level                 */
        }
        OS_EXIT_CRITICAL();
    }
}
```

OSSchedUnlock()

```
void  OSSchedUnlock (void)
{
#if OS_CRITICAL_METHOD == 3                  /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr;
#endif


    if (OSRunning == TRUE) {                /* Make sure multitasking is running    */
        OS_ENTER_CRITICAL();
        if (OSLockNesting > 0) {            /* Do not decrement if already 0        */
            OSLockNesting--;                /* Decrement lock nesting level         */
            if ((OSLockNesting == 0) &&
              (OSIntNesting == 0)) {        /* See if sched. enabled and not an ISR */
                OS_EXIT_CRITICAL();
                OS_Sched();                 /* See if a HPT is ready                */
            } else {
                OS_EXIT_CRITICAL();
            }
        } else {
            OS_EXIT_CRITICAL();
        }
    }
}
```

Recap: Race Avoidance

- OS_ENTER_CRITICAL/OS_EXIT_CRITICAL
  - neither interrupt nor preemption
  - For short critical sections (kernel code)
- OSSchedLock()/OSSchedUlock()
  - Preemption is prohibited but interrupts are handled
  - All tasks become non-preemptible
- OSSemPend()/OSSemPost()
  - Both preemption and interrupt are allowed
  - Only pending/posting tasks are affected

Recap: Interrupt Handling

- Do's
  - Make ISR as short as possible
  - Defer long job to a worker thread
- Don'ts
  - Call a system service with interrupt disabled
  - Call a system service with scheduler locked
  - Call a blocking call from an ISR

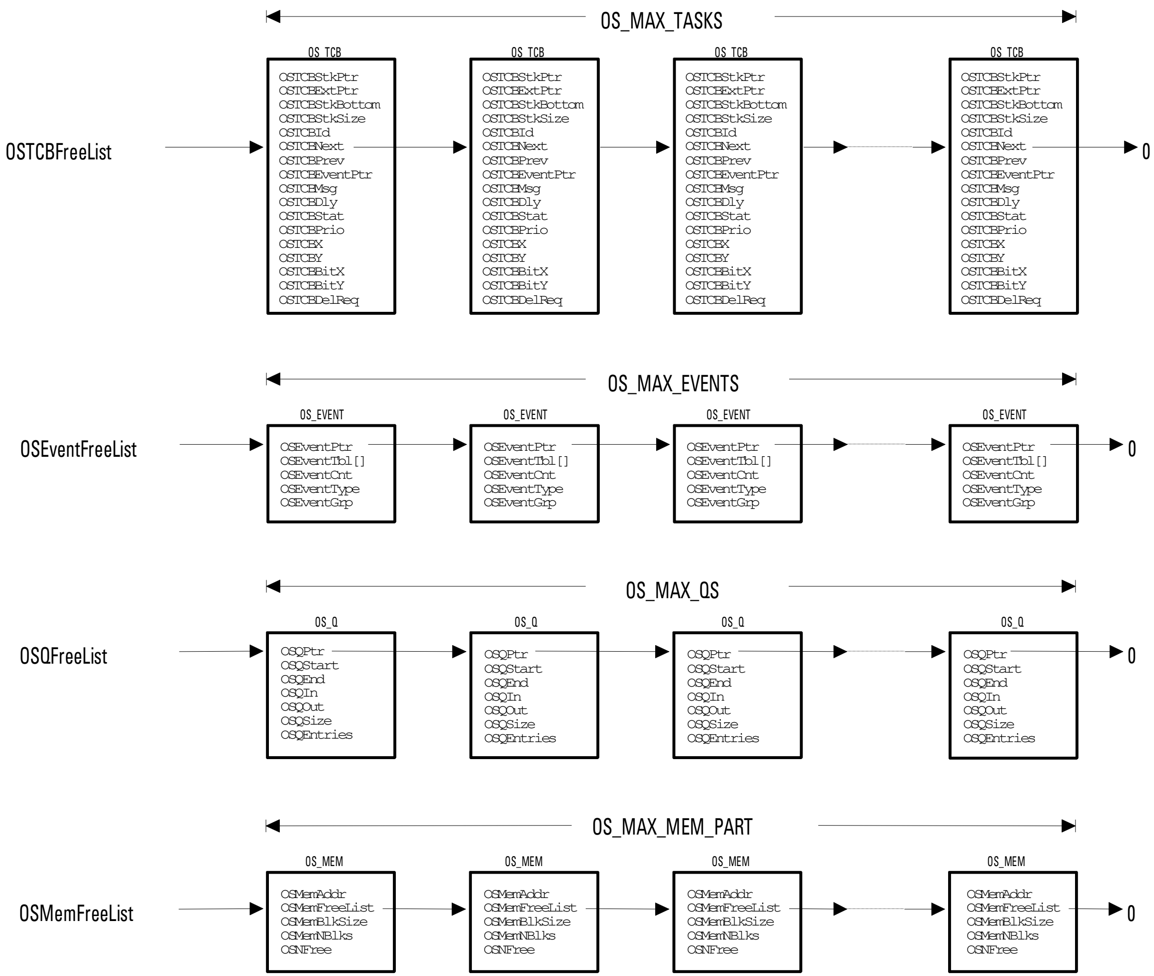
The Idle Task

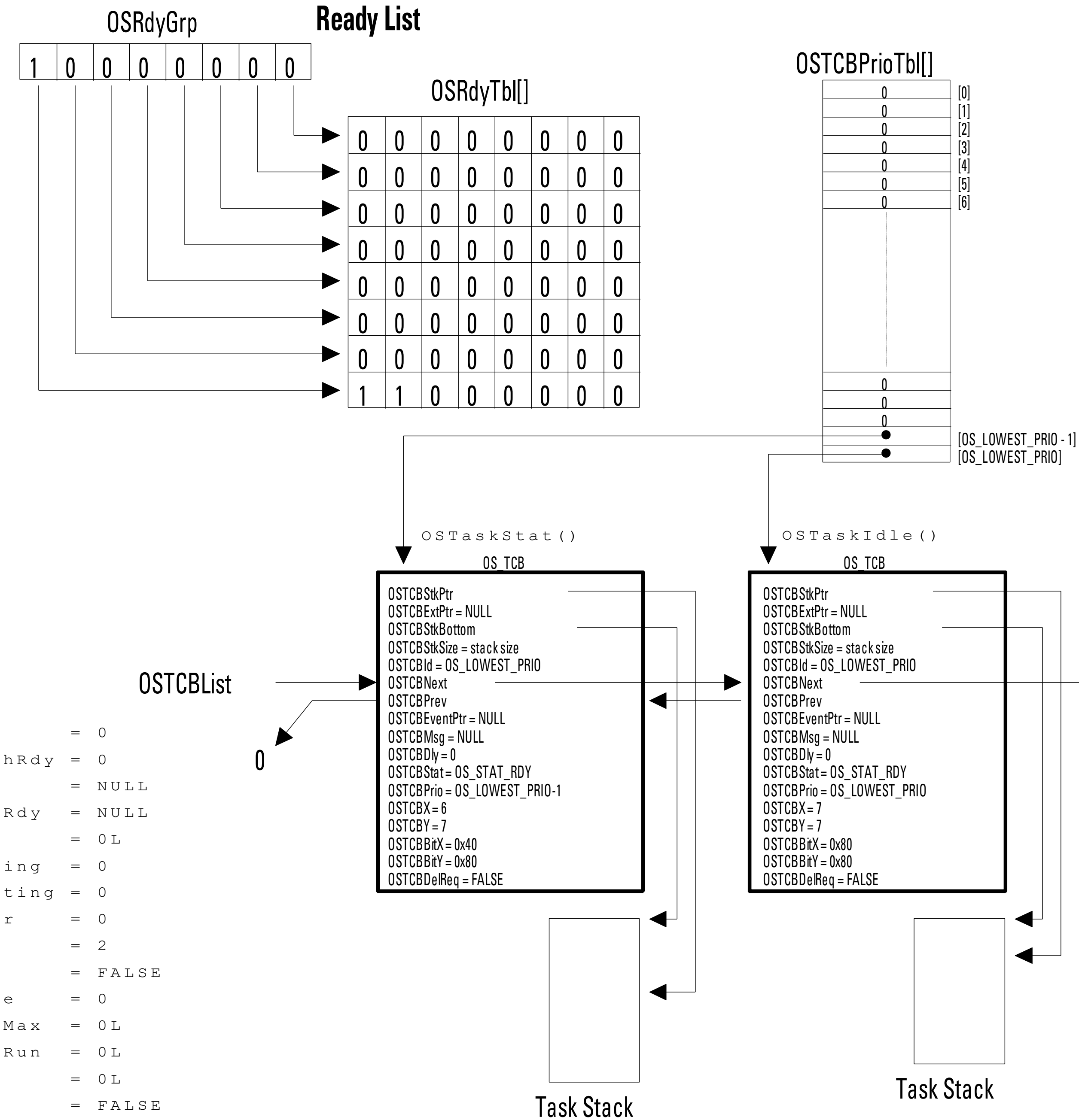
- The idle task is the lowest-priority task and can not be deleted or suspended
- Do not call delay or suspend services in OSTaskIdleHook()!!

```
void  OS_TaskIdle (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
   OS_CPU_SR  cpu_sr;
#endif


   pdata = pdata;
   for (;;) {
      OS_ENTER_CRITICAL();
      OSIdleCtr++;
      OS_EXIT_CRITICAL();
      OSTaskIdleHook();
   }
}
```

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

- OSInit() initializes the data structures for uC/OS-2 and creates OS_TaskIdle()

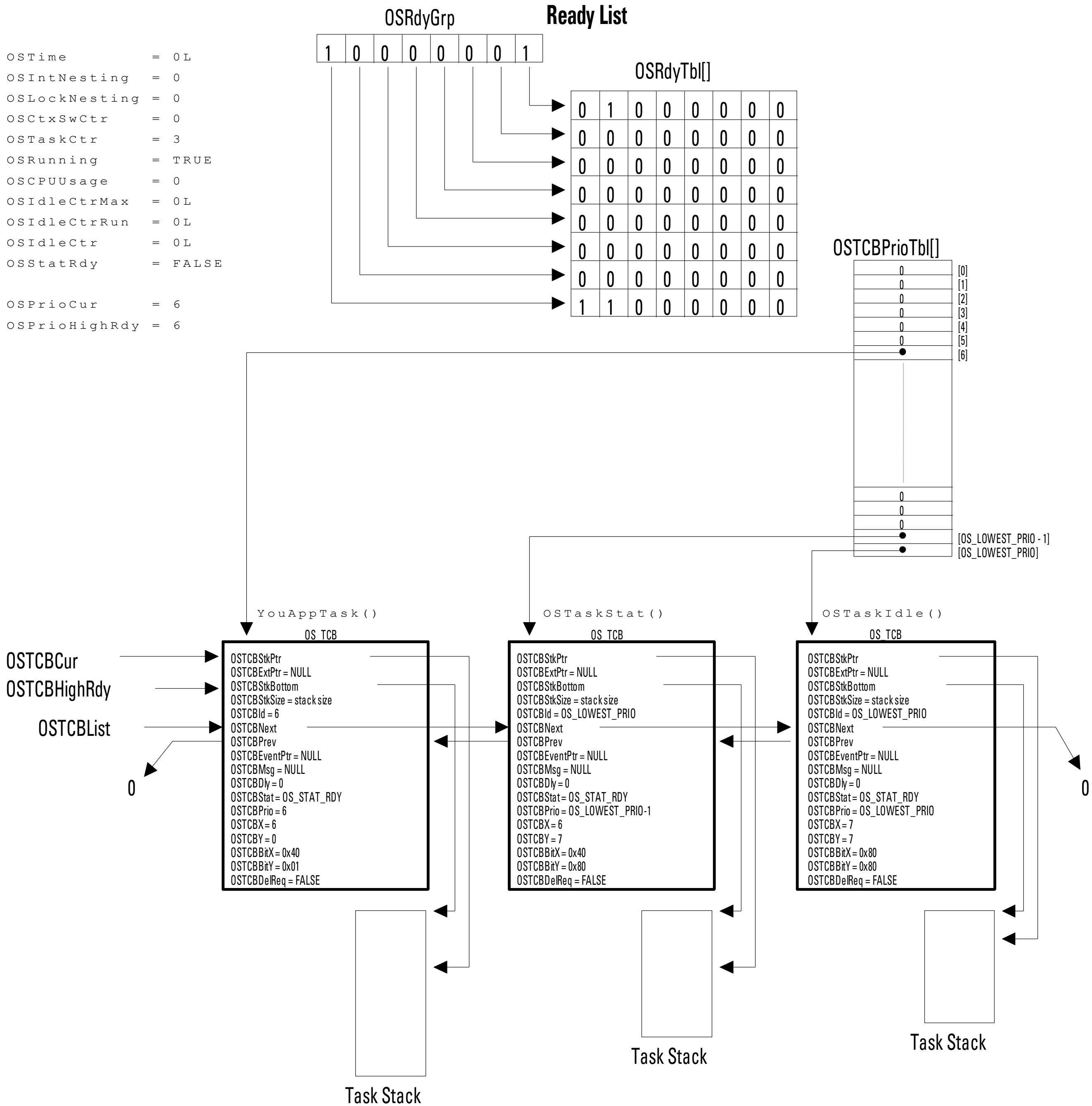
- OSStart() pops the CPU registers of the highest-priority ready task and then executes a return from interrupt instruction (IRET)
  - It never returns to the caller of OSStart() (i.e., main())
  - IRET: Actually no task is currently interrupted. New tasks are created as if they were just being interrupted.

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
1 0 0 0 0 0 0 1
OSRdyTbl[]
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0
OSTCBPrioTbl[]
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[OS_LOWEST_PRIO - 1]
[OS_LOWEST_PRIO]
YouAppTask()
OSTaskStat()
OSTaskIdle()
OS_TCB
OSTCBCur
OSTCBHighRdy
OSTCBList
0
OSTCBStkPtr
OSTCBExtPtr = NULL
OSTCBStkBottom
OSTCBStkSize = stack size
OSTCBId = 6
OSTCBNext
OSTCBPrev
OSTCBEventPtr = NULL
OSTCBMsg = NULL
OSTCBDly = 0
OSTCBStat = OS_STAT_RDY
OSTCBPrio = 6
OSTCBX = 6
OSTCBY = 0
OSTCBBitX = 0x40
OSTCBBitY = 0x01
OSTCBDelReq = FALSE
OS_TCB
OSTCBStkPtr
OSTCBExtPtr = NULL
OSTCBStkBottom
OSTCBStkSize = stack size
OSTCBId = OS_LOWEST_PRIO
OSTCBNext
OSTCBPrev
OSTCBEventPtr = NULL
OSTCBMsg = NULL
OSTCBDly = 0
OSTCBStat = OS_STAT_RDY
OSTCBPrio = OS_LOWEST_PRIO-1
OSTCBX = 6
OSTCBY = 7
OSTCBBitX = 0x40
OSTCBBitY = 0x80
OSTCBDelReq = FALSE
OS_TCB
OSTCBStkPtr
OSTCBExtPtr = NULL
OSTCBStkBottom
OSTCBStkSize = stack size
OSTCBId = OS_LOWEST_PRIO
OSTCBNext
OSTCBPrev
OSTCBEventPtr = NULL
OSTCBMsg = NULL
OSTCBDly = 0
OSTCBStat = OS_STAT_RDY
OSTCBPrio = OS_LOWEST_PRIO
OSTCBX = 7
OSTCBY = 7
OSTCBBitX = 0x80
OSTCBBitY = 0x80
OSTCBDelReq = FALSE
0
Task Stack
Task Stack
Task Stack

Quick Review

- There are several places at which the scheduling decision is made. What are they?

Summary

- In this chapter, you should learn:
  - What a task is, how uC/OS-2 manages a task, and related data structures
  - How the scheduler works, and the detailed operations done in context switches
  - The responsibility of the idle task
  - How interrupts are serviced in uC/OS-2
  - The initializing and starting of uC/OS-2