uC/OS-II: Kernel Structure

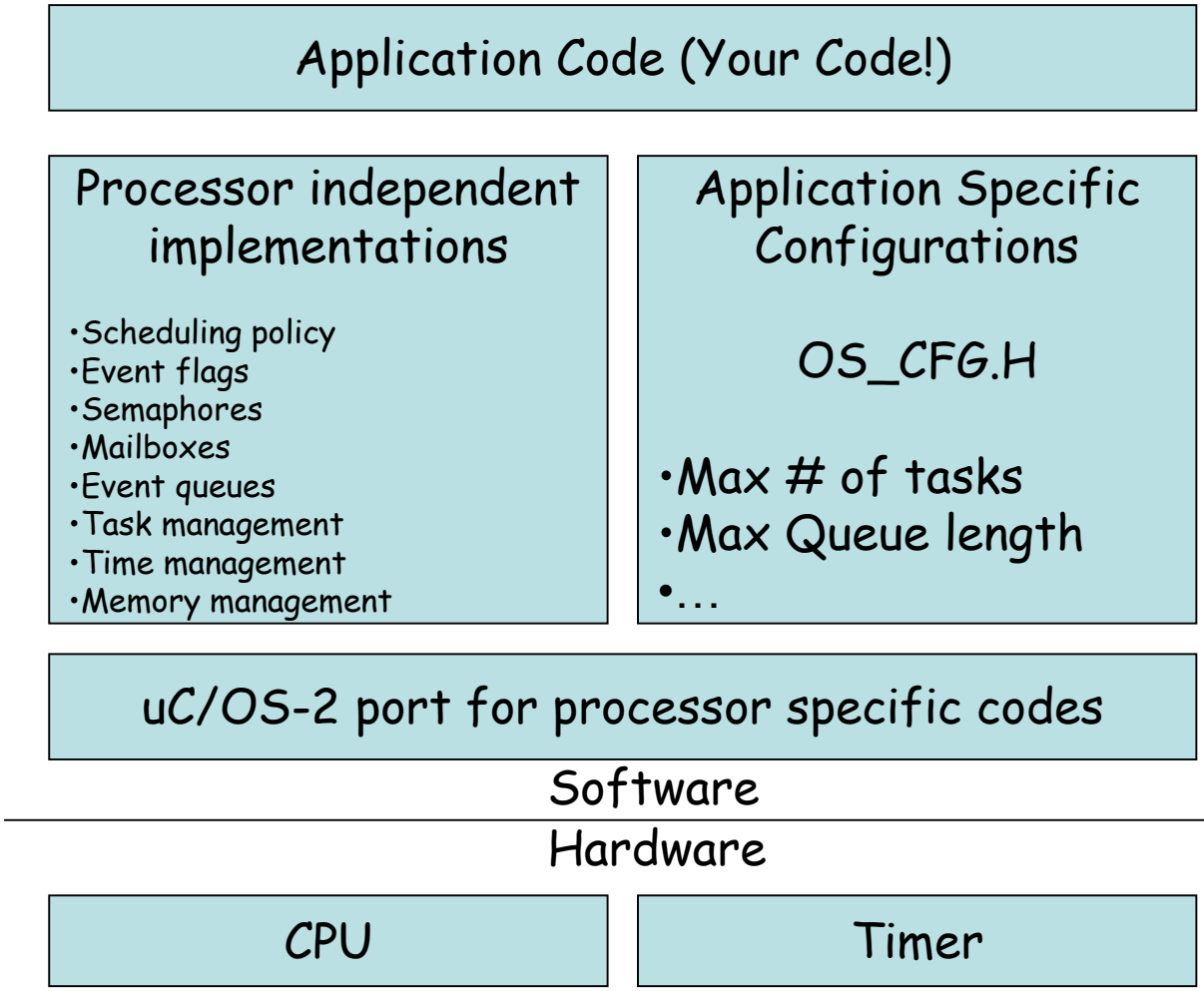
Embedded OS Implementation

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

Objectives

- To understand what a task is
- To learn how uC/OS-2 manages tasks

The uC/OS-2 File Structure



Tasks

- A task is an active entity which could do some computations
- 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 must associate with an unique priority
  - 63 and 62 are reserved (idle, stat)

- Insufficient number of priority will damage the schedulability of a real-time scheduler
  - Tie-breaking for tasks having the same priority is an issue
  - Fortunately, many embedded systems have a limited number of tasks to run

Tasks

- A task is created by OSTaskCreate() or OSTaskCreateExt()
- The priority of a task can be changed by OSTaskChangePrio()
- A task could delete itself when done.

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

The priority of the current task

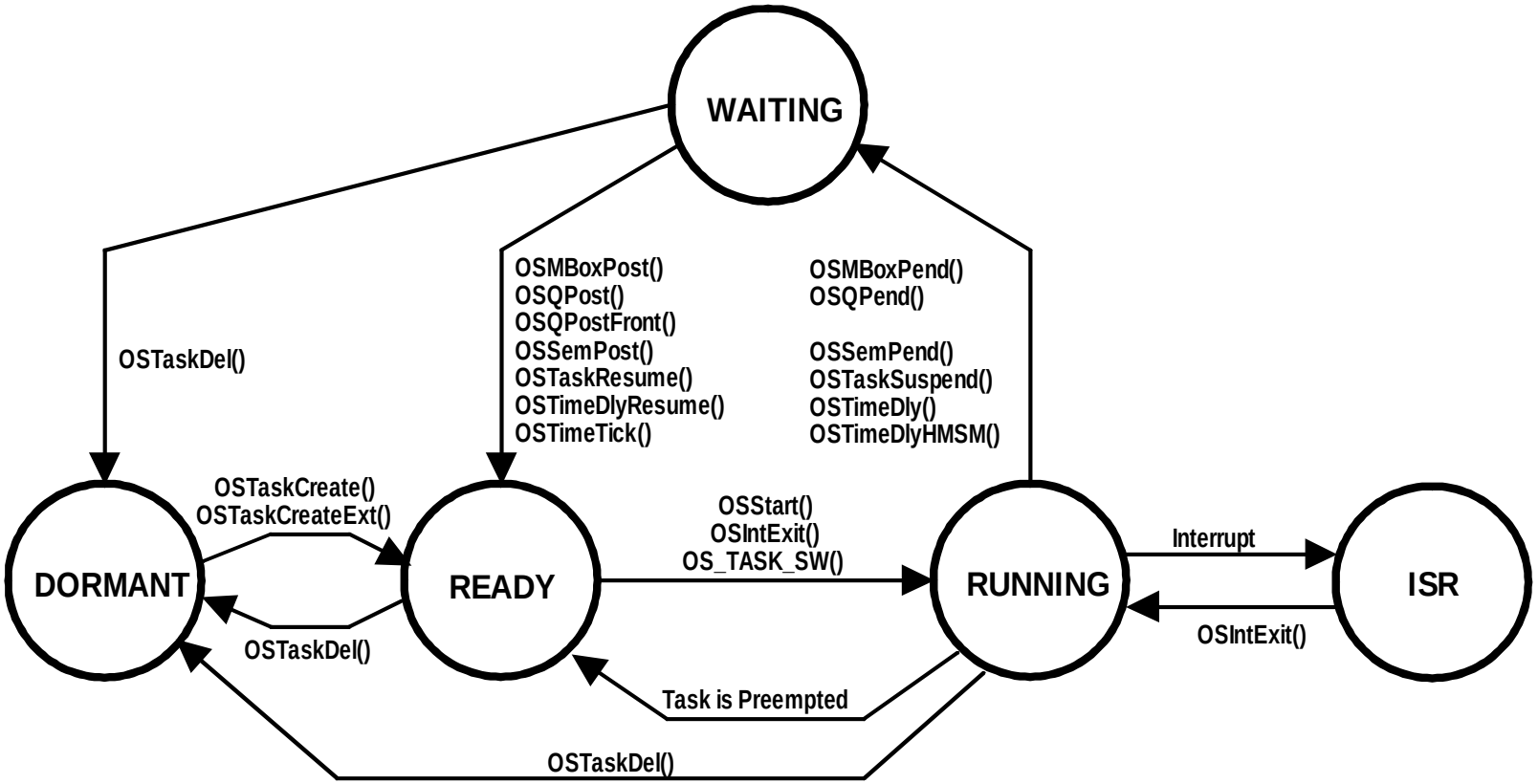
Task States

- Dormant: Procedures residing on RAM/ROM is not a task yet unless you call OSTaskCreate() to create one to execute them
- Ready: A task is neither delayed nor waiting for any event to occur
  - A task is ready once it is created
- 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 task is waiting for certain events to occur
  - Timer expiration, signaling of semaphores, messages in mailboxes, and etc
- ISR: A task is preempted by an interrupt
  - The stack of the interrupted task is utilized 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 Control Blocks (TCB)

- A TCB is a RAM-resident per-task data structure
- Each task is associated with a TCB
  - All valid TCB's are doubly linked
  - Free TCB's are linked in a free list
- The contents of a TCB is saved/restored when a context-switch occurs
  - Task priority, delay counter, event to wait, location of the stack
  - *CPU registers are stored in the stack rather than in the 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)

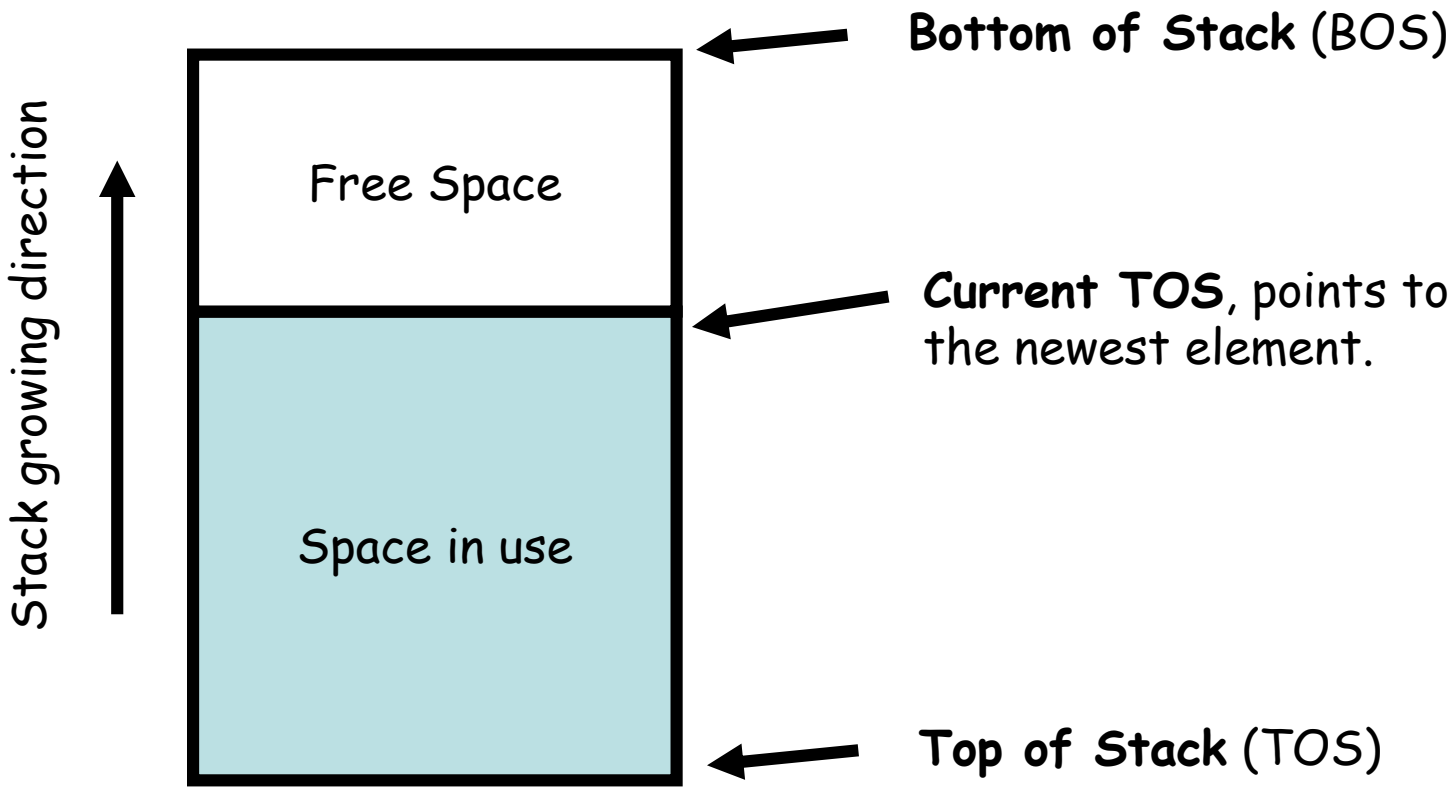
- **.OSTCBStkPtr** points to the current TOS for the task
  - It is the first entry of TCB so that it can be accessed directly via assembly language (offset=0)
- **.OSTCBExtPtr** is a pointer to a user-definable task control block extension.
  - Set OS_TASK_CREATE_EXT_EN to 1.
  - The pointer is set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

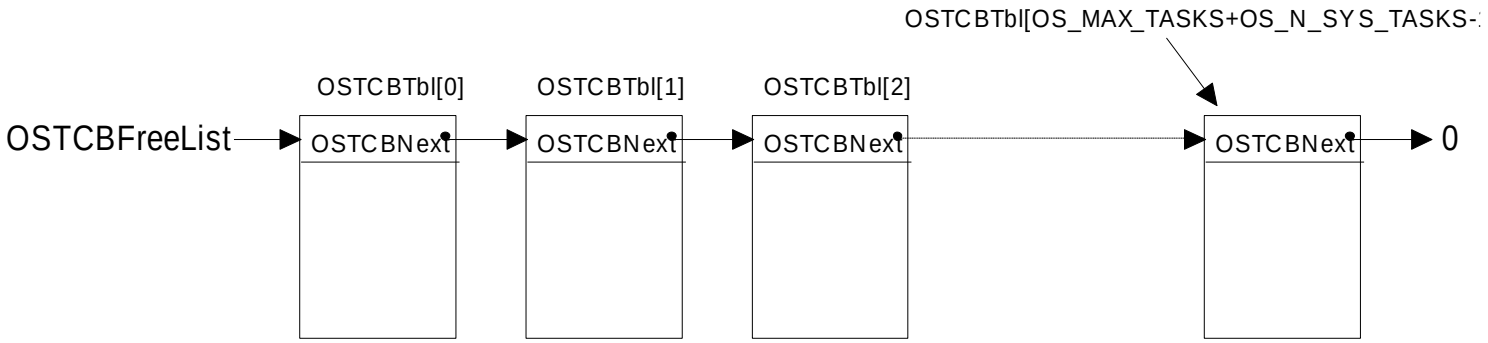
```
OSTCBY  = priority >> 3;
OSTCBBitY       = OSMapTbl[priority >> 3];
OSTCBX  = priority & 0x07;
OSTCBBitX       = OSMapTbl[priority & 0x07];
```

- **.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**[ ] : where all OS_TCBs are placed.
- 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

Critical Sections

- A critical section is a portion of code that is not safe from race conditions
- They can be protected by interrupt disabling/enabling interrupts or semaphores
  - However, the use of semaphores imposes much more overheads than enabling/disabling interrupts
  - A RTOS kernel itself mostly use interrupts disabling/enabling to protect critical sections
- Once interrupts are disabled, neither context switches nor any other ISR's can occur

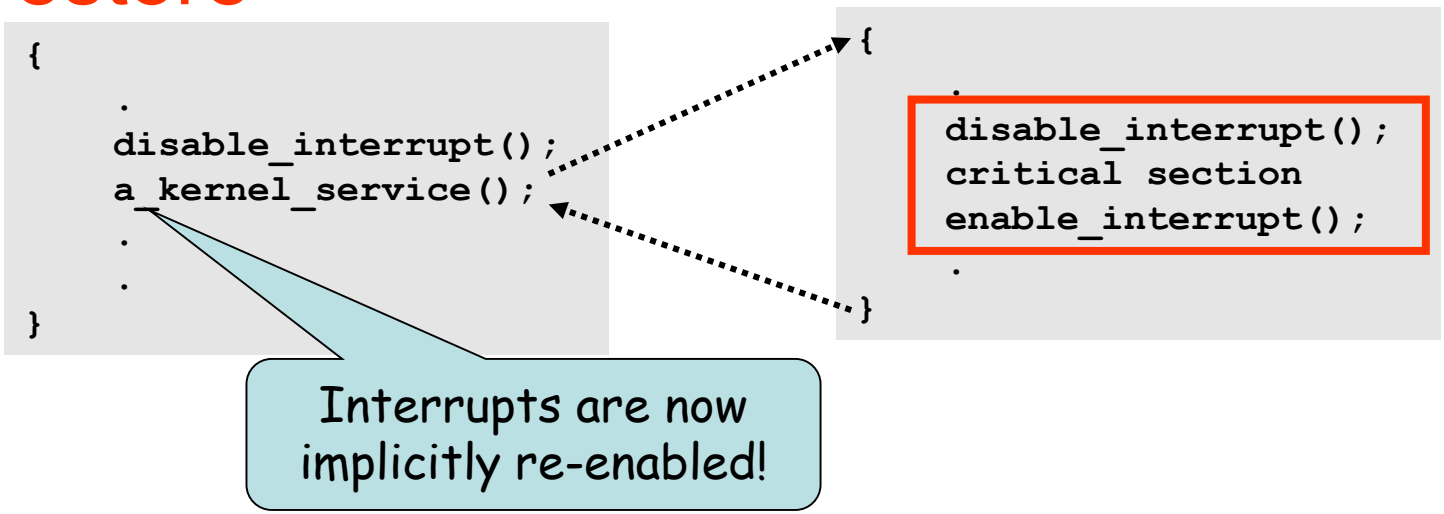
Critical Sections

- The states of the processor must be carefully maintained across multiple calls of OS_ENTER_CRITICAL() / OS_EXIT_CRITICAL()

- There are three possible implementations for the maintenance of process states:
  - Interrupt enabling/disabling instructions
  - Interrupt status save/restore onto/from stacks
  - Processor Status Word (PSW) save/restore onto/from memory variables

Critical Sections

- OS_CRITICAL_METHOD=1
- Interrupt enabling/disabling instructions.
- The simplest way, however, this approach does not have the sense of "save" and "restore"



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");
```

Task Scheduling

- The scheduler always schedules the highest-priority ready task to run
- Task-level scheduling and ISR-level scheduling are done by OS_Sched() and OSIntExit(), respectively
- uC/OS-2 scheduling time is a predictable amount of time, i.e., a constant time
  - For example, the design of the ready list intends to achieve this objective

```
void OSSched (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”

```
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

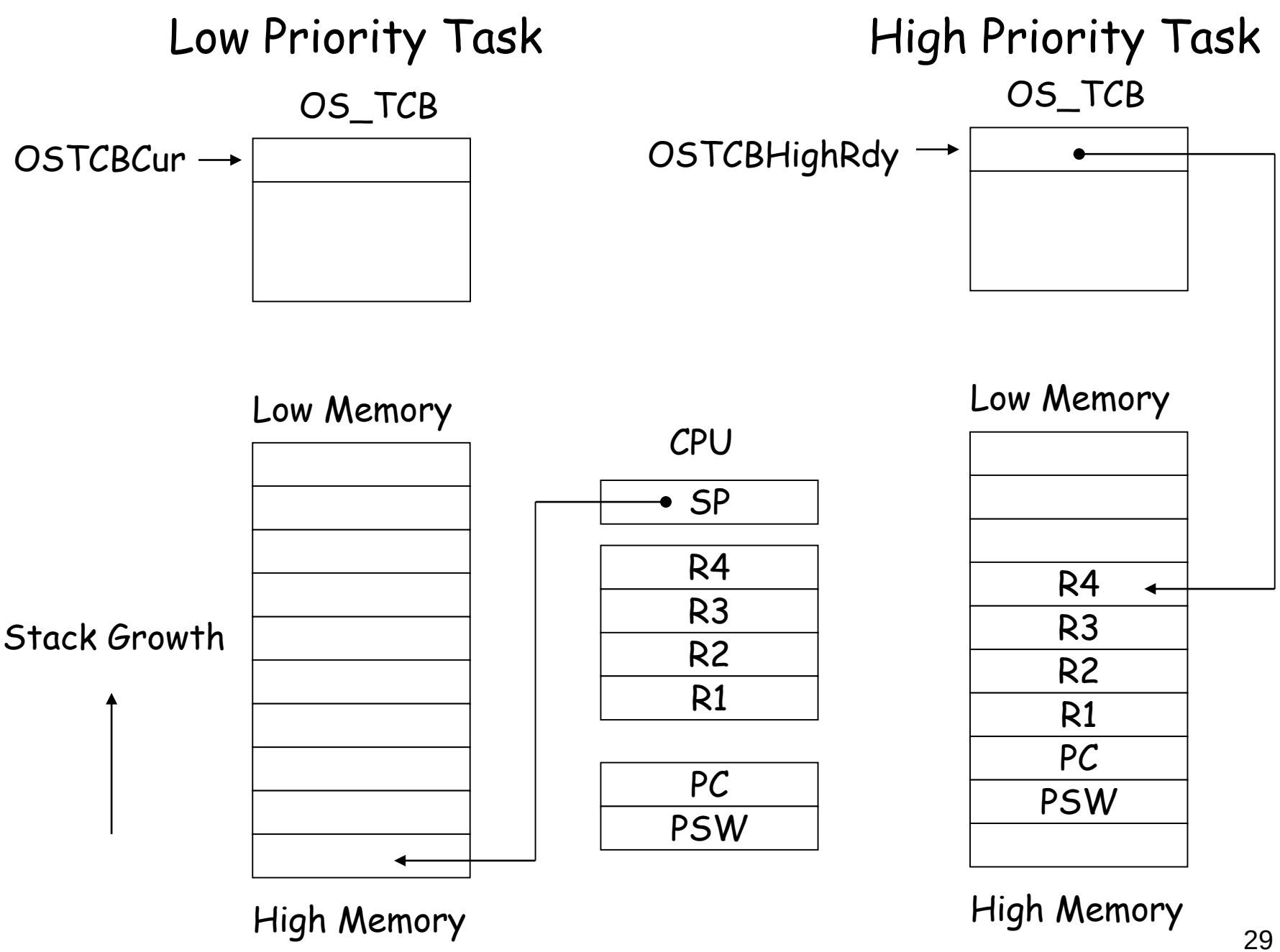
A context switch is performed.

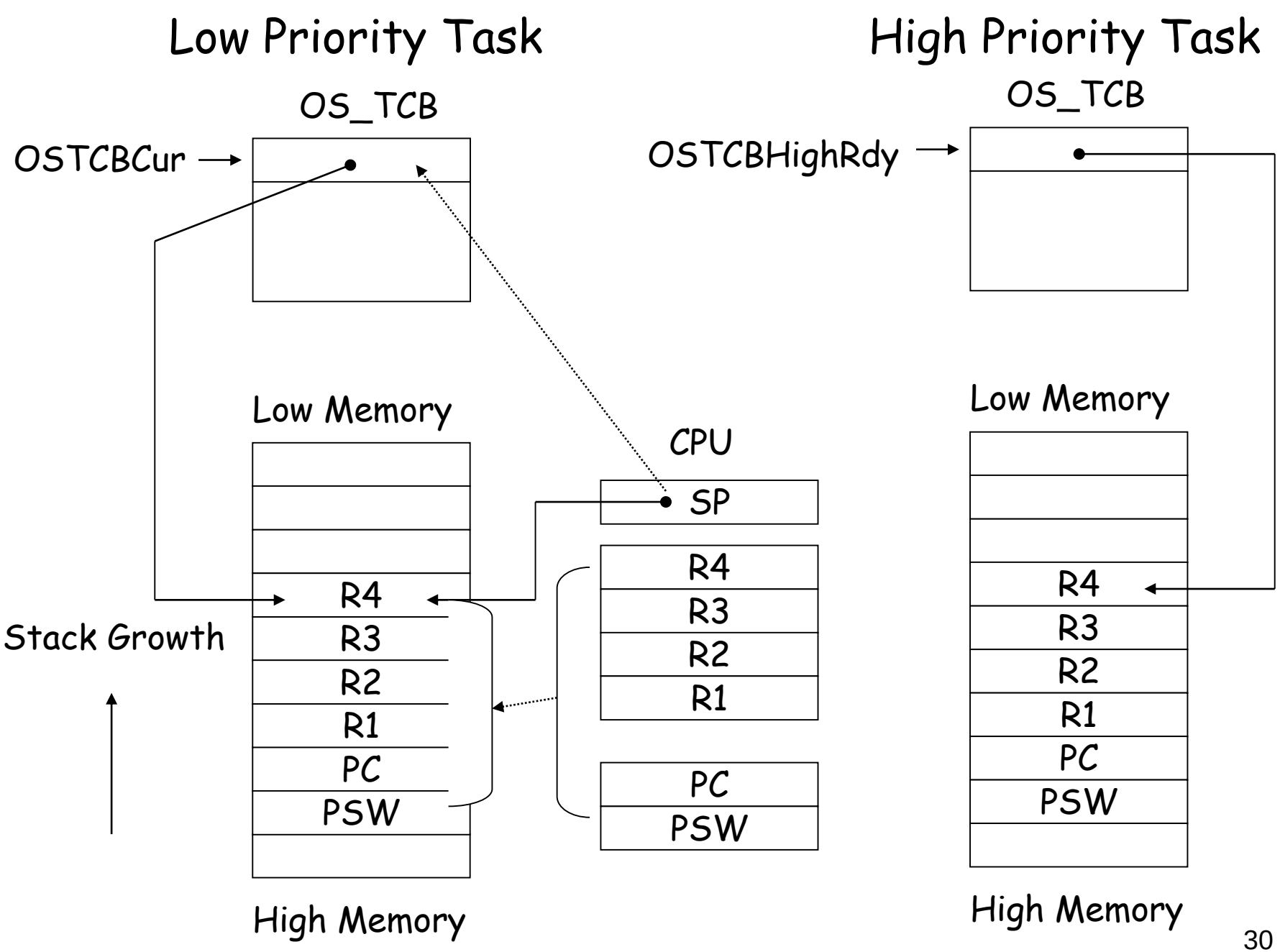
Note that OSIntCtxSw() is called instead of calling OS_TASK_SW() because the ISR already saves the CPU registers onto the stack.

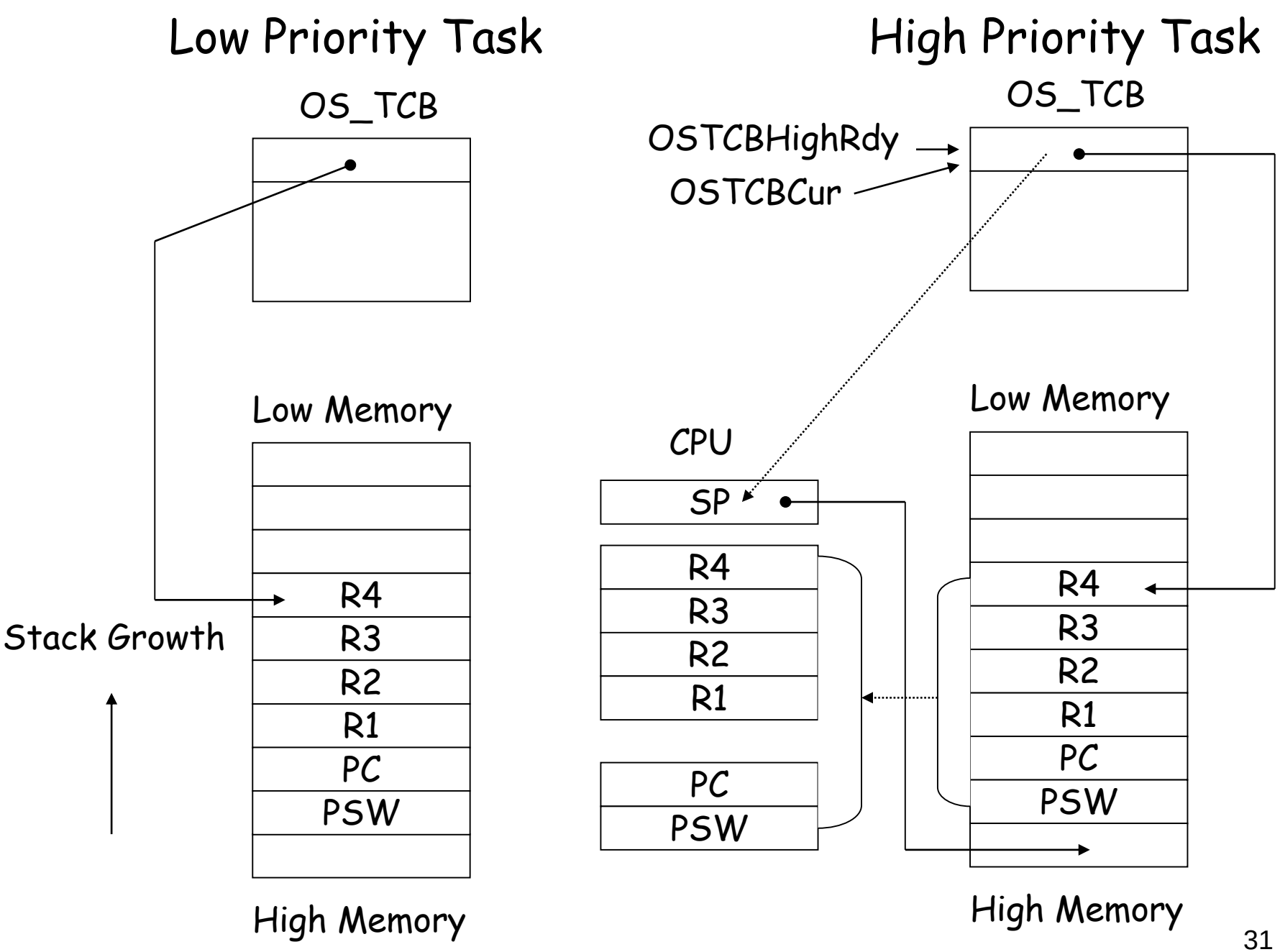
```
void OSIntEnter (void)
{
    OS_ENTER_CRITICAL();
    OSIntNesting++;
    OS_EXIT_CRITICAL();
}
```

Task Level Context Switch

- By default, context switches are handled at interrupt-level, therefore task-level scheduling will invoke a software interrupt to simulate that
  - Hardware dependent, porting must be done


Low Priority Task
High Priority Task
OS_TCB
OS_TCB
OSTCBCur
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
OS_TCB
OSTCBCur
OSTCBHighRdy
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


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

Locking and Unlocking the Scheduler

- OSSchedLock() prevent high-priority ready tasks from being scheduled to run while interrupts are still recognized
- OSSchedLock() and OSSchedUnlock() are used in pairs
- OSLockNesting keeps track of the number of OSSchedLock() has been called

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 will be locked up

- To lock the scheduler is to prevent from race conditions while interrupts can still be 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();
        }
    }
}
```

The Idle Task

- The idle task is always the lowest-priority task and can not be deleted or suspended
- To conserve power dissipation, you can issue a HALT instruction in the idle task

```
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();
   }
}
```

Summary

- In this class, you should learn:
  - What a task is, how uC/OS-2 manages a task, and related data structures
  - How the scheduler works in uC/OS-2
  - The responsibility of the idle task