uC/OS-II: Task Management

Embedded OS Implementation

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

Objectives

- To know (and trace the codes of) the services to:
  - create a task,
  - delete a task,
  - change the priority of a task,
  - suspend a task,
  - resume a task,
  - obtain information about a task.

Tasks

- A task could be periodic or aperiodic.

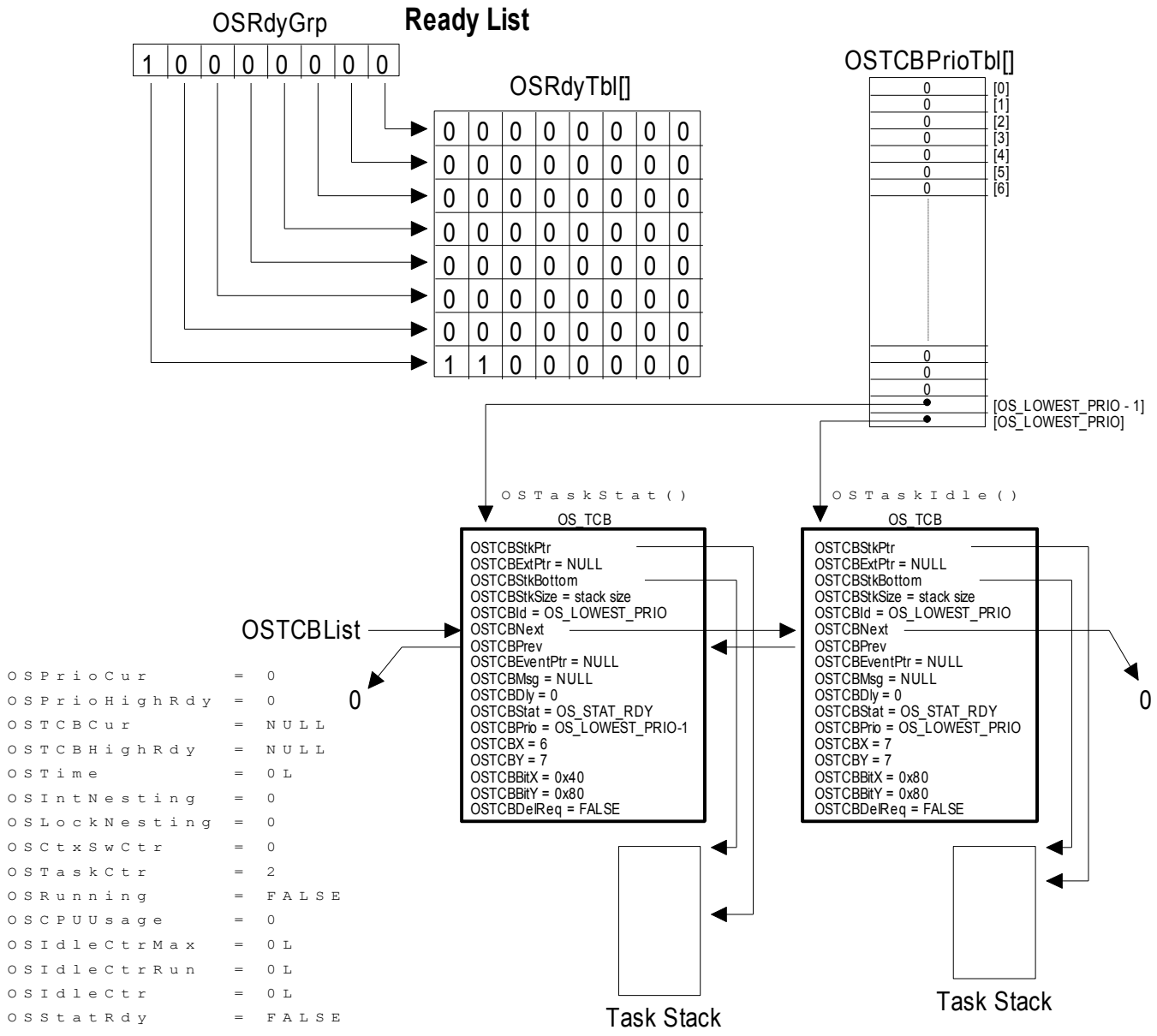
```
void YourTask (void *pdata)
{
  for (;;) {
     /* USER CODE */
     Call one of uC/OS-II's services:
     OSMboxPend();
     OSQPend();
     OSSemPend();
     OSTaskDel(OS_PRIO_SELF);
     OSTaskSuspend(OS_PRIO_SELF);
     OSTimeDly();
     OSTimeDlyHMSM();
     /* USER CODE */
     }
}
```

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

A task consists of periodically invoked jobs

Creating a Task

- You must create at least one task before multitasking is started.
  - Calling OSInit() and OSStatInit() will implicitly create 2 tasks (it must be done before OSStart()).
- An ISR can not create a task.
  - Due to the potential PEND operations in system services.
- Related data structures are created according to the given parameters.
  - A Task Control Block (TCB).
  - The stack of the created task.
  - The priority table.
- After a new task is created, the scheduler is called if multitasking is enabled.


OSRdyGrp
Ready List
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 [0]
0 [1]
0 [2]
0 [3]
0 [4]
0 [5]
0 [6]
0
0
0
[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
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
Task Stack
Task Stack

OSTaskCreate()

```
INT8U  OSTaskCreate (void (*task)(void *pd),
                     void *pdata, OS_STK *ptos, INT8U prio)
{
    OS_STK   *psp;
    INT8U     err;

    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {
        OSTCBPrioTbl[prio] = (OS_TCB *)1;

    OS_EXIT_CRITICAL();
    psp = (OS_STK *)OSTaskStkInit(task, pdata, ptos, 0);
```

Linux: first in first out

null

Occupying a priority table slot and re-enable interrupts immediately.

占位，TCB未建好，怕被抢走

Stack initialization is a hardware-dependant implementation. (Because of the growing direction)

OSTaskCreate()

```
    err = OS_TCBInit(prio, psp, (OS_STK *)0, 0, 0, (void *)0, 0);
    if (err == OS_NO_ERR) {
        OS_ENTER_CRITICAL();
        OSTaskCtr++;
        OS_EXIT_CRITICAL();
        if (OSRunning == TRUE) {
            OS_Sched();
        }
    } else {
        OS_ENTER_CRITICAL();
        OSTCBPrioTbl[prio] = (OS_TCB *)0;
        OS_EXIT_CRITICAL();
    }
    return (err);
}
OS_EXIT_CRITICAL();
return (OS_PRIO_EXIST);
}
```

Create a corresponding TCB and connect it with the priority table.

If the task is created with multitasking started, the scheduler is called.

```
void  OS_Sched (void)
{
#if OS_CRITICAL_METHOD == 3                                 /* Allocate storage for CPU status register     */
    OS_CPU_SR  cpu_sr;
#endif
    INT8U      y;


    OS_ENTER_CRITICAL();
    if ((OSIntNesting == 0) && (OSLockNesting == 0)) {      /* Sched. only if all ISRs done & not locked    */
        y             = OSUnMapTbl[OSRdyGrp];               /* Get pointer to HPT ready to run              */
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) {                   /* No Ctx Sw if current task is highest rdy     */
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;                                   /* Increment context switch counter             */
            OS_TASK_SW();                                   /* Perform a context switch                     */
        }
    }
    OS_EXIT_CRITICAL();
}
```

OSTaskCreateExt()

- **task**: a pointer-to-function points to the entry point of a task (note the syntax).
- **pdata**: a parameter passed to the task.
- **ptos**: a pointer points to the top-of-stack.
- **prio**: task priority.
- **id**: task id, for future extension.
- **pbos**: a pointer points to the bottom-of-stack.
- **stk_size**: the stack size in the number of elements (OS_STK bytes each)
- **pext**: an user-defined extension to the TCB.
- **opt**: the options specified to create the task.

OSTaskCreateExt()

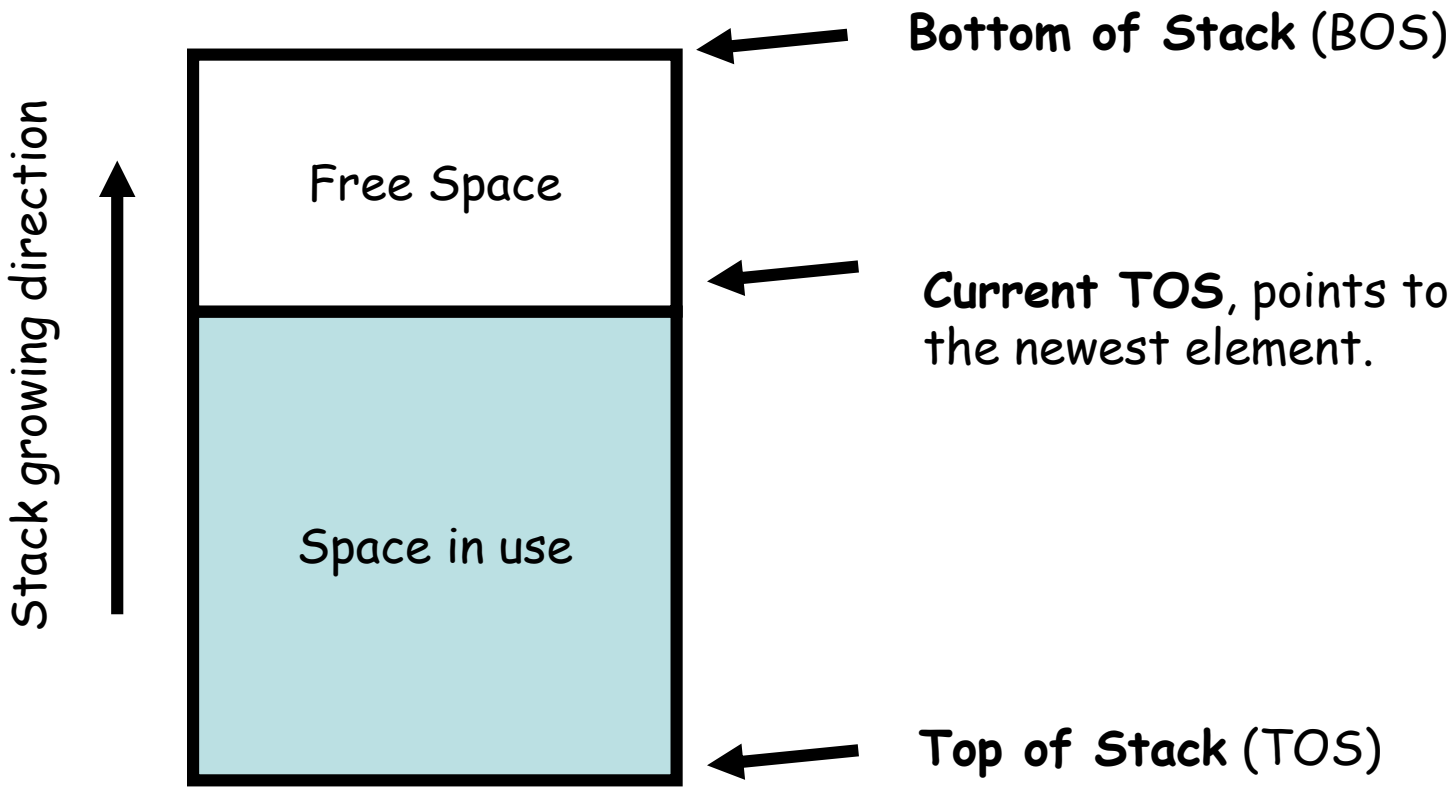
```
INT8U  OSTaskCreateExt (void   (*task)(void *pd),  void    *pdata,
                        OS_STK  *ptos, INT8U    prio, INT16U   id, OS_STK  *pbos,
                        INT32U   stk_size, void    *pext, INT16U   opt)
{
   OS_STK   *psp;
   INT8U     err;


   OS_ENTER_CRITICAL();
   if (OSTCBPrioTbl[prio] == (OS_TCB *)0) {
      OSTCBPrioTbl[prio] = (OS_TCB *)1;

      OS_EXIT_CRITICAL();

      if (((opt & OS_TASK_OPT_STK_CHK) != 0x0000) ||
         ((opt & OS_TASK_OPT_STK_CLR) != 0x0000)) {
         #if OS_STK_GROWTH == 1
         (void)memset(pbos, 0, stk_size * sizeof(OS_STK));
         #else
         (void)memset(ptos, 0, stk_size * sizeof(OS_STK));
         #endif
      }
```

The stack is required to be cleared

The stack grows toward low address, so the starting address is bos.

The stack grows toward high address, so the starting address is tos.

OSTaskCreateExt()



OSTaskCreateExt()

```
        psp = (OS_STK *)OSTaskStkInit(task, pdata, ptos, opt);
        err = OS_TCBInit(prio, psp, pbos, id, stk_size, pext, opt);
        if (err == OS_NO_ERR) {
            OS_ENTER_CRITICAL();
            OSTaskCtr++;
            OS_EXIT_CRITICAL();
            if (OSRunning == TRUE) {
                OS_Sched();
            }
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;
            OS_EXIT_CRITICAL();
        }
        return (err);
    }
    OS_EXIT_CRITICAL();
    return (OS_PRIO_EXIST);
}
```

OSTaskCreate:
(task, pdata, ptos, 0);
(prio, psp, (OS_STK *)0, 0, 0, (void *)0, 0);

OSTaskCreateExt:
(task, pdata, ptos, opt);
(prio, psp, pbos, id, stk_size, pext, opt);

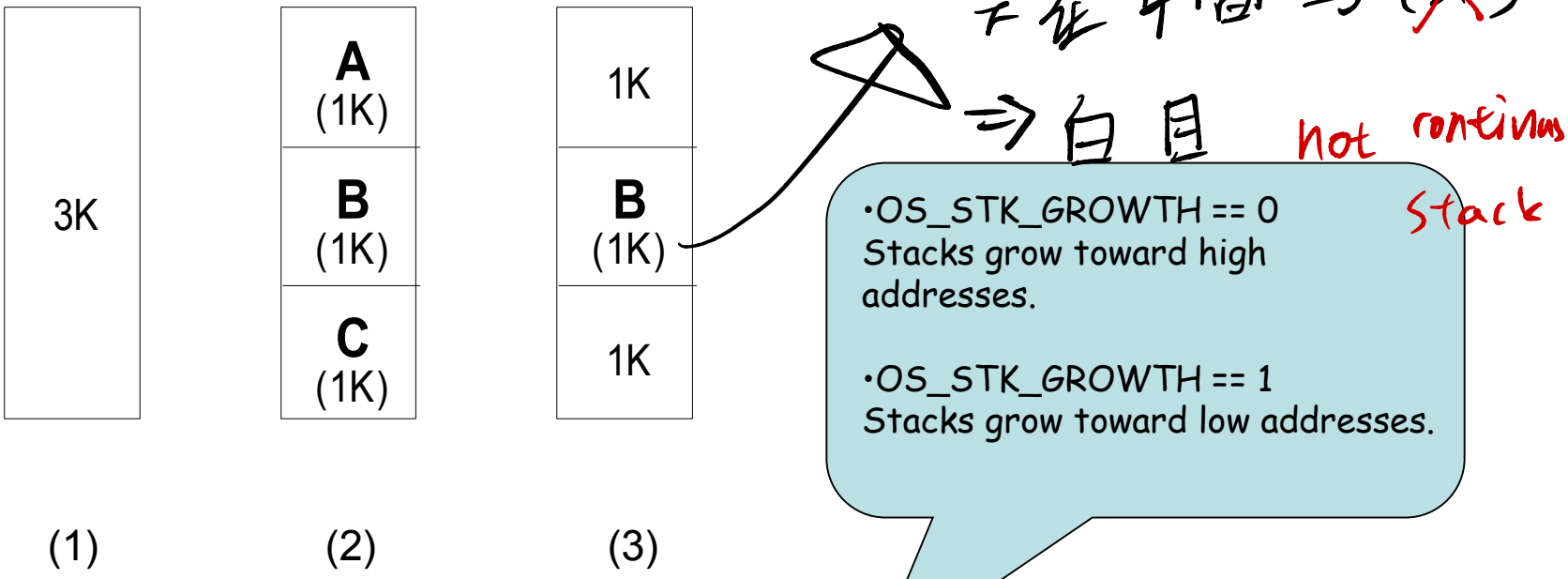
Task Stacks

- The stack must be a contiguous memory space.
- Stack grows either toward high memory address or low memory space.
- Operations over stacks might not be byte-wise.
  - The element size might not be 1 byte.
    - 16 bits under x86.
  - Defined by a macro OS_STK.
- Stack space can be statically declared variables or dynamically allocated space (malloc()).

Task Stacks

- Fragmentation might cause memory allocations (for stacks) failed.
- Stacks might grow in different directions under different processors.

Task Stacks



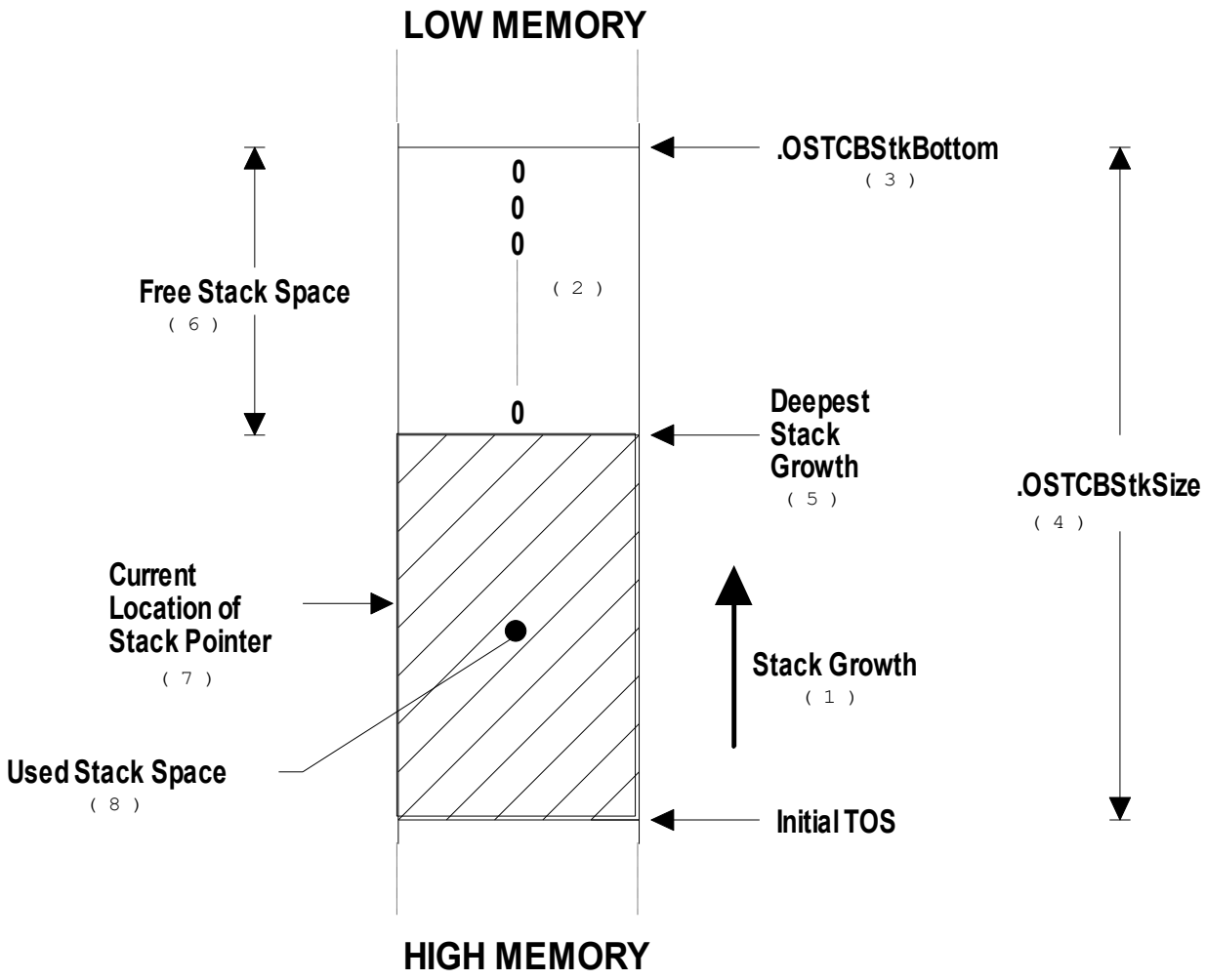

```
OS_STK  TaskStack[TASK_STACK_SIZE];
#if OS_STK_GROWTH == 0
   OSTaskCreate(task, pdata, &TaskStack[0], prio);
#else
   OSTaskCreate(task, pdata, &TaskStack[TASK_STACK_SIZE-1], prio);
#endif
```

Stack Checking

- Stack checking intends to determine the maximum run-time usage of stacks.

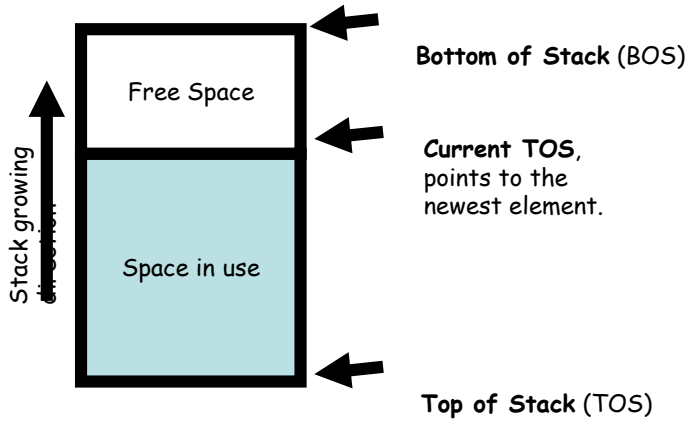
- To do stack checking:
  - Create your tasks by using OSTaskCreateExt() with options OS_TASK_OPT_STK_CHK + OS_TASK_OPT_STK_CLR and give the tasks reasonably large stacks.
  - Call OSTaskStkChk() to determine the stack usage of a certain task.
  - Reduce the stack size if possible, once you think you had run enough simulations.

Stack Checking



```
INT8U  OSTaskStkChk (INT8U prio, OS_STK_DATA *pdata)
{
   OS_TCB    *ptcb;
   OS_STK    *pchk;
   INT32U     free, size;

   pdata->OSFree = 0;
   pdata->OSUsed = 0;
   OS_ENTER_CRITICAL();
   if (prio == OS_PRIO_SELF) {
      prio = OSTCBCur->OSTCBPrio;
   }
   ptcb = OSTCBPrioTbl[prio];
   if (ptcb == (OS_TCB *)0) {
      OS_EXIT_CRITICAL();
      return (OS_TASK_NOT_EXIST);
   }
   if ((ptcb->OSTCBOpt & OS_TASK_OPT_STK_CHK) == 0) {
      OS_EXIT_CRITICAL();
      return (OS_TASK_OPT_ERR);
   }
```

```
    free = 0;
    size = ptcb->OSTCBStkSize;
    pchk = ptcb->OSTCBStkBottom;
    OS_EXIT_CRITICAL();
#if OS_STK_GROWTH == 1
    while (*pchk++ == (OS_STK)0) {
        free++;
    }
#else
    while (*pchk-- == (OS_STK)0) {
        free++;
    }
#endif
    pdata->OSFree = free * sizeof(OS_STK);
    pdata->OSUsed = (size - free) * sizeof(OS_STK);
    return (OS_NO_ERR);
}
```



For either stack growing direction...

Counting from BOS until a non-zero element is encountered.

Deleting a Task

- Deleting a task means that the data structures (e.g., TCB) corresponding to the task-to-delete would be removed from main-memory.
  - The code resides on ROM are still there.
- Deleting a task is slightly more complicated than creating it since every resources/objects held by the task must be returned to the operating system.

```
INT8U  OSTaskDel (INT8U prio)
{
    OS_EVENT     *pevent;
    OS_FLAG_NODE *pnode;
    OS_TCB       *ptcb;
    BOOLEAN       self;

    if (OSIntNesting > 0) {
        return (OS_TASK_DEL_ISR);
    }
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {
        prio = OSTCBCur->OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb != (OS_TCB *)0) {
        if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {
            OSRdyGrp &= ~ptcb->OSTCBBitY;
        }
        pevent = ptcb->OSTCBEventPtr;
        if (pevent != (OS_EVENT *)0) {
            if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
                pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
            }
        }
        pnode = ptcb->OSTCBFlagNode;
        if (pnode != (OS_FLAG_NODE *)0) {
            OS_FlagUnlink(pnode);
        }
```

We do not allow to delete a task within ISR's, because the ISR might currently interrupts that task.

Clear the corresponding bit of the task-to-delete in the ready list.

If the row are all 0's, then clear the RdyGrp bit also.

Remove the task from the event control block since we no longer wait for the event.

```
    ptcb->OSTCBDly  = 0;
    ptcb->OSTCBStat = OS_STAT_RDY;
    if (OSLockNesting < 255) {
        OSLockNesting++;
    }
    OS_EXIT_CRITICAL();
    OS_Dummy();
    OS_ENTER_CRITICAL();
    if (OSLockNesting > 0) {
        OSLockNesting--;
    }
    OSTaskDelHook(ptcb);
    OSTaskCtr--;
    OSTCBPrioTbl[prio] = (OS_TCB *)0;
    if (ptcb->OSTCBPrev == (OS_TCB *)0) {
        ptcb->OSTCBNext->OSTCBPrev = (OS_TCB *)0;
        OSTCBList                  = ptcb->OSTCBNext;
    } else {
        ptcb->OSTCBPrev->OSTCBNext = ptcb->OSTCBNext;
        ptcb->OSTCBNext->OSTCBPrev = ptcb->OSTCBPrev;
    }
    ptcb->OSTCBNext = OSTCBFreeList;
    OSTCBFreeList   = ptcb;
    OS_EXIT_CRITICAL();
    OS_Sched();
    return (OS_NO_ERR);
}
OS_EXIT_CRITICAL();
return (OS_TASK_DEL_ERR);
}
```

Prevent tickISR from making this task ready when interrupts are re-enabled later.

Prevent this task from being resumed since we are not in the ready list now (a "ready task can not be resumed).

Lock the scheduler.

Interrupts are re-enabled for a while (note that the scheduler is locked).
*What does OS_dummy() do?

↳ critical too long

⇒ 先 Lock
⇒ ISR ⇒ 服務其它人
⇒ unLock

Relink

Changing a Task's Priority

- When you create a new task, you assign the task a priority
- At run time, you can change this priority dynamically by calling OSTaskChangePrio
- Cannot change the priority of the idle task
- INT8U OSTaskChangePrio(INT8U oldPrio, INT8U newPrio)

Procedures

- Reserve the new priority by OSTCBPrioTbl[newprio] = (OS_TCB *) 1;
- Remove the task from the priority table
- Insert the task into new location of the priority table
- Change the OS_TCB of the task
- Call OSSched()

```
#if OS_TASK_CHANGE_PRIO_EN > 0
INT8U  OSTaskChangePrio (INT8U oldprio, INT8U newprio)
{
#if OS_EVENT_EN > 0
    OS_EVENT    *pevent;
#endif

    OS_TCB      *ptcb;
    INT8U        x, y, bitx, bity;

    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[newprio] != (OS_TCB *)0) {                    /* New priority must not already exist */
        OS_EXIT_CRITICAL();
        return (OS_PRIO_EXIST);
    } else {
        OSTCBPrioTbl[newprio] = (OS_TCB *)1;                       /* Reserve the entry to prevent others */
        OS_EXIT_CRITICAL();
        y    = newprio >> 3;                                       /* Precompute to reduce INT. latency   */
        bity = OSMapTbl[y];
        x    = newprio & 0x07;
        bitx = OSMapTbl[x];
        OS_ENTER_CRITICAL();
        if (oldprio == OS_PRIO_SELF) {                             /* See if changing self                */
            oldprio = OSTCBCur->OSTCBPrio;                         /* Yes, get priority                   */
        }
        ptcb = OSTCBPrioTbl[oldprio];
        if (ptcb != (OS_TCB *)0) {                                 /* Task to change must exist           */
            OSTCBPrioTbl[oldprio] = (OS_TCB *)0;                   /* Remove TCB from old priority        */
            if ((OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) != 0x00) {  /* If task is ready make it not */
                if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {
                    OSRdyGrp &= ~ptcb->OSTCBBitY;
                }
                OSRdyGrp    |= bity;                               /* Make new priority ready to run      */
                OSRdyTbl[y] |= bitx;
```

the task is ready to run

```
#if OS_EVENT_EN > 0
        } else {
            pevent = ptcb->OSTCBEventPtr;
            if (pevent != (OS_EVENT *)0) {                  /* Remove from event wait list  */
                if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
                    pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
                }
                pevent->OSEventGrp    |= bity;              /* Add new priority to wait list       */
                pevent->OSEventTbl[y] |= bitx;
            }
#endif
        }
        OSTCBPrioTbl[newprio] = ptcb;                       /* Place pointer to TCB @ new priority */
        ptcb->OSTCBPrio       = newprio;                    /* Set new task priority               */
        ptcb->OSTCBY          = y;
        ptcb->OSTCBX          = x;
        ptcb->OSTCBBitY       = bity;
        ptcb->OSTCBBitX       = bitx;
        OS_EXIT_CRITICAL();
        OS_Sched();                                         /* Run highest priority task ready     */
        return (OS_NO_ERR);
    } else {
        OSTCBPrioTbl[newprio] = (OS_TCB *)0;                /* Release the reserved prio.          */
        OS_EXIT_CRITICAL();
        return (OS_PRIO_ERR);                               /* Task to change didn't exist         */
    }
  }
}
#endif
```

Suspending a Task

- A suspended task can only resumed by calling the OSTaskResume() function call
- If a task being suspended is also waiting for time to expire, the suspension needs to be removed and the time needs to expire in order for the task ready to run.
- INT8U OSTaskSuspend(INT8U prio)
- INT8U OSTaskResume(INT8U prio)

```
INT8U  OSTaskSuspend (INT8U prio)
{
    BOOLEAN    self;
    OS_TCB    *ptcb;

    OS_ENTER_CRITICAL();
    /* See if suspending self*/
    if (prio == OS_PRIO_SELF) {
        prio = OSTCBCur->OSTCBPrio;
        self = TRUE;
    } else if (prio == OSTCBCur->OSTCBPrio) {
        self = TRUE;
    } else {
        self = FALSE;
    }
    ptcb = OSTCBPrioTbl[prio];
    /* Task to suspend must exist*/
    if (ptcb == (OS_TCB *)0) {
        OS_EXIT_CRITICAL();
        return (OS_TASK_SUSPEND_PRIO);
    }
```

```
 /* Make task not ready*/
    if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {
        OSRdyGrp &= ~ptcb->OSTCBBitY;
    }
    /* Status of task is 'SUSPENDED'*/
    ptcb->OSTCBStat |= OS_STAT_SUSPEND;
    OS_EXIT_CRITICAL();
    /* Context switch only if SELF*/
    if (self == TRUE) {
        OS_Sched();
    }
    return (OS_NO_ERR);
}
```

Resuming a Task

```
INT8U  OSTaskResume (INT8U prio) {
    OS_TCB    *ptcb;

    OS_ENTER_CRITICAL();
    ptcb = OSTCBPrioTbl[prio];
    /* Task to suspend must exist*/
    if (ptcb == (OS_TCB *)0) {
        OS_EXIT_CRITICAL();
        return (OS_TASK_RESUME_PRIO);
    }
```

```
 /* Task must be suspended   */
    if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) != 0x00) {
       /* Remove suspension        */
        if (((ptcb->OSTCBStat &= ~OS_STAT_SUSPEND) == OS_STAT_RDY) &&
             /* Must not be delayed      */
             (ptcb->OSTCBDly  == 0)) {
            /* Make task ready to run   */
            OSRdyGrp               |= ptcb->OSTCBBitY;
            OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
            OS_EXIT_CRITICAL();
            OS_Sched();
        } else {
            OS_EXIT_CRITICAL();
        }
        return (OS_NO_ERR);
    }
    OS_EXIT_CRITICAL();
    return (OS_TASK_NOT_SUSPENDED);
}
```

Getting Task Information

- OSTaskQuery return a copy of the contents of the desired task's OS_TCB
- To call OSTaskQuery, your application must allocate storage for an OS_TCB
- Only this function to SEE what a task is doing
  - don't modify the contains (OSTCBNext, OSTCBPrev)

```
INT8U  OSTaskQuery (INT8U prio, OS_TCB *pdata) {
    OS_TCB    *ptcb;

    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {
        prio = OSTCBCur->OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl[prio];
    /* Task to query must exist*/
    if (ptcb == (OS_TCB *)0) {
        OS_EXIT_CRITICAL();
        return (OS_PRIO_ERR);
    }
    /* Copy TCB into user storage area*/
    memcpy(pdata, ptcb, sizeof(OS_TCB));
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```