*u*C/OS-II – Real-Time Kernel

Tei-Wei Kuo

ktw@csie.ntu.edu.tw

Dept. of Computer Science and Information Engineering

National Taiwan University

• Slides come from my class notes, slides from graduates and students in my lab (Li-Pin Chang and Shi-Wu Lo), and slides contributed by Labrosse, the author of MicroC/OS-II.

Contents

- Introduction
- Kernel Structure

Introduction

- Different ports from the official *u*C/OS-II Web site at http://www.uCOS-II.com.
- Neither freeware nor open source code.
- *u*C/OS-II is certified in an avionics product by FAA in July 2000.
- Text Book:
  - Jean J. Labresse, “MicroC/OS-II: The Real-Time Kernel,” CMP Book, ISBN: 1-57820-103-9

Introduction

- *u*C/OS-II
  - Micro-Controller Operating Systems, Version 2
  - A very small real-time kernel.
    - Memory footprint is about 20KB for a fully functional kernel.
    - Source code is about 5,500 lines, mostly in ANSI C.
    - It’s source is open but not free for commercial usages.

Introduction

- *u*C/OS-II
  - Preemptible priority-driven real-time scheduling.
    - 64 priority levels (max 64 tasks)
    - 8 reserved for *u*C/OS-II
    - Each task is an infinite loop.
  - Deterministic execution times for most *u*C/OS-II functions and services.
  - Nested interrupts could go up to 256 levels.

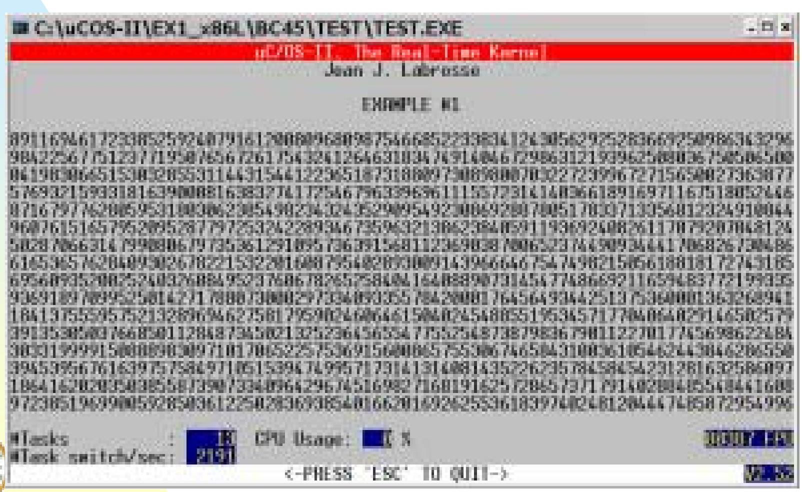
Introduction

- *u*C/OS-II
  - Supports of various 8-bit to 64-bit platforms: x86, 68x, MIPS, 8051, etc
  - Easy for development: Borland C++ compiler and DOS (optional).
- However, *u*C/OS-II still lacks of the following features:
  - Resource synchronization protocols.
  - Sporadic task support.
  - Soft-real-time support.

Introduction

- Getting started with *u*C/OS-II!
  - See how a *u*C/OS-II program looks like.
  - Learn how to write a skeleton program for *u*C/OS-II.
    - How to initialize *u*C/OS-II?
    - How to create real-time tasks?
    - How to use inter-task communication mechanisms?
    - How to catch system events?

Example 1: Multitasking



Example 1 : Multitasking

- 13 tasks run concurrently.
  - 2 internal tasks:
    - The idle task and the statistic task.
  - 11 user tasks:
    - 10 tasks randomly print numbers onto the screen.
- Focus: System initialization and task creation.

Example 1: Multitasking

- Files
  - The main program (test.c)
  - The big include file (includes.h)
  - The configuration of *u*C/OS-II (os_cfg.h) for each application
- Tools needed:
  - Borland C++ compiler (V3.1+)

Example 1

```
#include "includes.h"

/*
*********************************************************************************
    ********************************
*                                               CONSTANTS
*********************************************************************************
    ********************************
*/

#define  TASK_STK_SIZE                512       /* Size of each task's
    stacks (# of WORDs)             */
#define  N_TASKS                       10       /* Number of identical
    tasks                           */

/*
*********************************************************************************
    ********************************
*                                               VARIABLES
*********************************************************************************
    ********************************
*/

OS_STK        TaskStk[N_TASKS][TASK_STK_SIZE];        /* Tasks stacks
    */
OS_STK        TaskStartStk[TASK_STK_SIZE];
char          TaskData[N_TASKS];                      /* Parameters to
    pass to each task                */
OS_EVENT     *RandomSem;
```

A semaphore
(explain later)

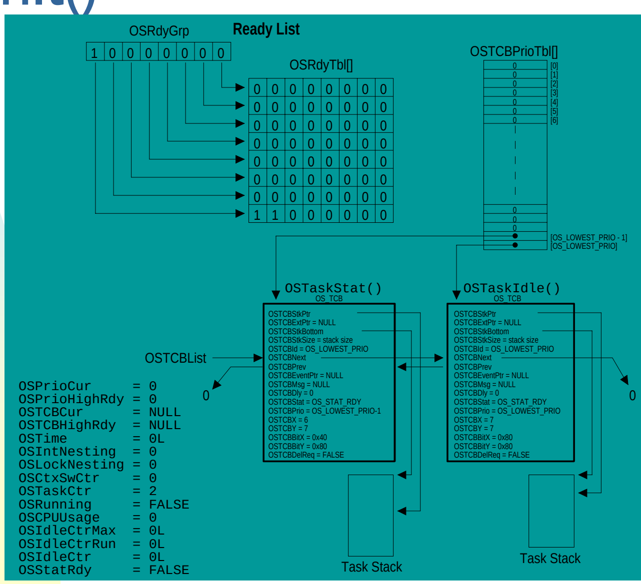
Main()

```
void main (void)
{
    PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);
   (1)
    OSInit();
   (2)
    PC_DOSSaveReturn();
   (3)
    PC_VectSet(uCOS, OSCtxSw);
   (4)
    RandomSem = OSSemCreate(1);
   (5)
    OSTaskCreate(TaskStart,
   (6)
                 (void *)0,
                 (void *)&TaskStartStk[TASK_STK_SIZE-
   1],
                 0);
    OSStart();
   (7)
}
```

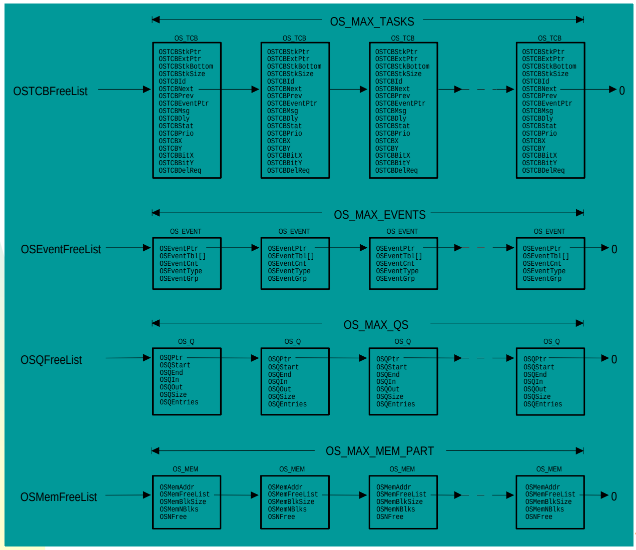
Main()

- OSinit():
  - internal structures of uC/OS-2.
    - Task ready list.
    - Priority table.
    - Task control blocks (TCB).
    - Free pool.
  - Create housekeeping tasks.
    - The idle task.
    - The statistics task.

OSinit()



OSinit()



*

Main()

- PC_DOSSaveReturn()
  - Save the current status of DOS for the future restoration.
    - Interrupt vectors and the RTC tick rate.
  - Set a global returning point by calling setjump().
    - *u*C/OS-II can come back here when it terminates.
    - PC_DOSReturn()

PC_DOSSaveReturn()

```
void PC_DOSSaveReturn (void)
{
    PC_ExitFlag  = FALSE;                                          (1)
    OSTickDOSCtr =     8;                                          (2)
    PC_TickISR   = PC_VectGet(VECT_TICK);                          (3)

    OS_ENTER_CRITICAL();
    PC_VectSet(VECT_DOS_CHAIN, PC_TickISR);                        (4)
    OS_EXIT_CRITICAL();

    setjmp(PC_JumpBuf);                                            (5)
    if (PC_ExitFlag == TRUE) {
        OS_ENTER_CRITICAL();
        PC_SetTickRate(18);                                        (6)
        PC_VectSet(VECT_TICK, PC_TickISR);                         (7)
        OS_EXIT_CRITICAL();
        PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);          (8)
        exit(0);                                                   (9)
    }
}
```

*

Main()

- PC_VectSet(uCOS,OSCtxSw)
  - Install the context switch handler.
  - Interrupt 0x08 under 80x86 family.
    - Invoked by INT instruction.
- OSStart()
  - Start multitasking of uC/OS-2.
  - It never returns to main().
  - *u*C/OS-II is terminated if PC_DOSReturn() is called. *

Main()

- OSSemCreate()
  - Create a semaphore for resource synchronization.
    - To protect non-reentrant codes.
  - The created semaphore becomes a mutual exclusive mechanism if "1" is given as the initial value.
  - In this example, a semaphore is created to protect the standard C library "random()".

Main()

- OSTaskCreate()
  - Create tasks with the given arguments.
  - Tasks become "ready" after they are created.
- Task
  - An active entity which could do some computations.
  - Priority, CPU registers, stack, text, housekeeping status.
- The *u*C/OS-II picks up the highest-priority task to run on context-switching.
  - Tightly coupled with RTC ISR.

OSTaskCreate()

- OSTaskCreate(
  TaskStart,
  (void *)0,
  &TaskStartStk[TASK_STK_SIZE-1],
   0
  );

Entry point of the task (a pointer to function)

User-specified data

Top of Stack

Priority (0=hightest)

TaskStart()

```
void  TaskStart (void *pdata)
{
#if OS_CRITICAL_METHOD == 3                          /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr;
#endif
    char       s[100];
    INT16S     key;

    pdata = pdata;                                   /* Prevent compiler warning                 */

    TaskStartDispInit();                             /* Initialize the display                   */

    OS_ENTER_CRITICAL();
    PC_VectSet(0x08, OSTickISR);                     /* Install uC/OS-II's clock tick ISR        */
    PC_SetTickRate(OS_TICKS_PER_SEC);                /* Reprogram tick rate                      */
    OS_EXIT_CRITICAL();

    OSStatInit();                                    /* Initialize uC/OS-II's statistics         */

    TaskStartCreateTasks();                          /* Create all the application tasks         */

    for (;;) {
        TaskStartDisp();                             /* Update the display                       */


        if (PC_GetKey(&key) == TRUE) {               /* See if key has been pressed              */
            if (key == 0x1B) {                       /* Yes, see if it's the ESCAPE key          */
                PC_DOSReturn();                      /* Return to DOS                            */
            }
        }

        OSCtxSwCtr = 0;                              /* Clear context switch counter             */
        OSTimeDlyHMSM(0, 0, 1, 0);                   /* Wait one second                          */
    }
}
```

Change the ticking rate

TaskStart()

- OS_ENTER_CRITICAL()/OS_EXIT_CRITICAL()
  - Enable/disable most interrupts.
  - An alternative way to accomplish mutual exclusion.
    - No rescheduling is possible during the disabling of interrupts. (different from semaphores)
  - Processor specific.
    - CLI/STI (x86 real mode)
    - Interrupt descriptors (x86 protected mode)

TaskStartCreateTasks()

Entry point of the created task

```
static  void  TaskStartCreateTasks (void)
{
    INT8U  i;

    for (i = 0; i < N_TASKS; i++) {

        TaskData[i] = '0' + i;

        OSTaskCreate(
        Task,
        (void *)&TaskData[i],
        &TaskStk[i][TASK_STK_SIZE - 1],
        i + 1);
    }
}
```

Argument: character to print

Stack

Priority

Task()

```
void  Task (void *pdata)
{
    INT8U  x;
    INT8U  y;
    INT8U  err;


    for (;;) {
        OSSemPend(RandomSem, 0, &err);/* Acquire semaphore to perform random numbers
    */
        x = random(80);               /* Find X position where task number will appear
    */
        y = random(16);               /* Find Y position where task number will appear
    */
        OSSemPost(RandomSem);         /* Release semaphore
    */
                                      /* Display the task number on the screen
    */
        PC_DispChar(x, y + 5, *(char *)pdata, DISP_FGND_BLACK + DISP_BGND_LIGHT_GRAY);
        OSTimeDly(1);                 /* Delay 1 clock tick
    */
    }
}
```

Semaphore operations.

Semaphores

- A semaphore consists of a wait list and an integer counter.
  - OSSemPend():
    - Counter--;
    - If the value of the semaphore <0, then the task is blocked and moved to the wait list immediately.
    - A time-out value can be specified.
  - OSSemPost():
    - Counter++;
    - If the value of the semaphore >= 0, then a task in the wait list is removed from the wait list.
      - Reschedule if needed.

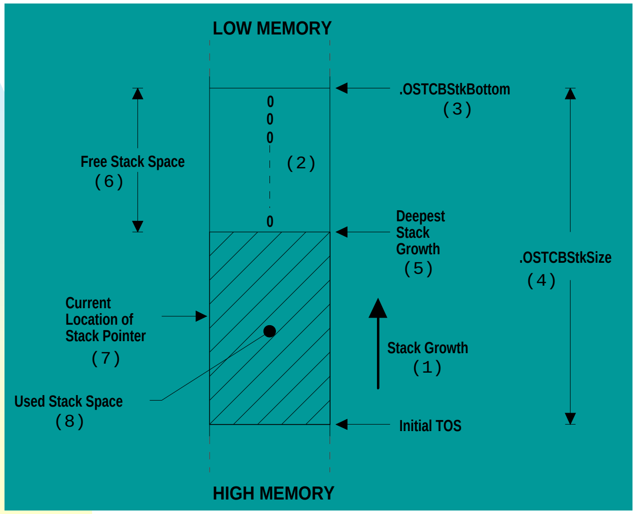
Example 1: Multitasking

- Summary:
  - *u*C/OS-II is initialized and started by calling OSInit() and OSStart(), respectively.
  - Before *u*C/OS-II is started,
    - The DOS status is saved by calling PC_DOSSaveReturn().
    - A context switch handler is installed by calling PC_VectSet().
    - User tasks must be created first!
  - Shared resources can be protected by semaphores.
    - OSSemPend(),OSSemPost().

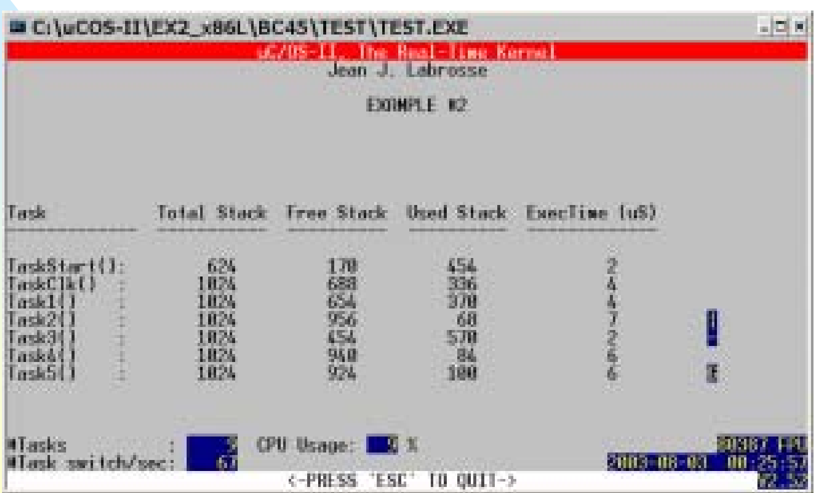
Example 2: Stack Checking

- Five tasks do jobs on message sending/receiving, char-displaying with wheel turning, and char-printing.
  - More task creation options
    - Better judgment on stack sizes
  - Stack usage of each task
    - Different stack sizes for tasks
  - Emulation of floating point operations
    - 80386 or lower-end CPU's
  - Communication through mailbox
    - Only the pointer is passed.

The Stack Usage of a Task



Example 2: Stack Checking



```
#define         TASK_STK_SIZE     512             /* Size of each task's stacks (# of WORDs)
    */

#define         TASK_START_ID       0             /* Application tasks IDs
    */
#define         TASK_CLK_ID         1
#define         TASK_1_ID           2
#define         TASK_2_ID           3
#define         TASK_3_ID           4
#define         TASK_4_ID           5
#define         TASK_5_ID           6

#define         TASK_START_PRIO    10             /* Application tasks priorities
    */
#define         TASK_CLK_PRIO      11
#define         TASK_1_PRIO        12
#define         TASK_2_PRIO        13
#define         TASK_3_PRIO        14
#define         TASK_4_PRIO        15
#define         TASK_5_PRIO        16

OS_STK       TaskStartStk[TASK_STK_SIZE];         /* Startup    task stack
    */
OS_STK       TaskClkStk[TASK_STK_SIZE];           /* Clock      task stack
    */
OS_STK       Task1Stk[TASK_STK_SIZE];             /* Task #1    task stack
    */
OS_STK       Task2Stk[TASK_STK_SIZE];             /* Task #2    task stack
    */
OS_STK       Task3Stk[TASK_STK_SIZE];             /* Task #3    task stack
    */
OS_STK       Task4Stk[TASK_STK_SIZE];             /* Task #4    task stack
    */
OS_STK       Task5Stk[TASK_STK_SIZE];             /* Task #5    task stack
    */

OS_EVENT     *AckMbox;                            /* Message mailboxes for Tasks #4 and #5
    */
OS_EVENT     *TxMbox;
```

2 Mailboxes

Main()

```
void main (void)
{
    OS_STK *ptos;
    OS_STK *pbos;
    INT32U  size;

    PC_DispClrScr(DISP_FGND_WHITE);                /* Clear the screen          */
    OSInit();                                      /* Initialize uC/OS-II       */
    PC_DOSSaveReturn();                            /* Save environment to return to
    DOS        */
    PC_VectSet(uCOS, OSCtxSw);                     /* Install uC/OS-II's context
    switch vector */
    PC_ElapsedInit();                              /* Initialized elapsed time
    measurement     */
    ptos        = &TaskStartStk[TASK_STK_SIZE - 1]; /* TaskStart() will use Floating-
    Point      */
    pbos        = &TaskStartStk[0];
    size        = TASK_STK_SIZE;
    OSTaskStkInit_FPE_x86(&ptos, &pbos, &size);
    OSTaskCreateExt(TaskStart,
                   (void *)0,
                   ptos,
                   TASK_START_PRIO,
                   TASK_START_ID,
                   pbos,
                   size,
                   (void *)0,
                   OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

    OSStart();                                     /* Start multitasking    */
  }
```

TaskStart()

```
void  TaskStart (void *pdata)
{
#if OS_CRITICAL_METHOD == 3                          /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr;
#endif
    INT16S     key;


    pdata = pdata;                                   /* Prevent compiler warning                 */

    TaskStartDispInit();                             /* Setup the display                        */

    OS_ENTER_CRITICAL();                             /* Install uC/OS-II's clock tick ISR        */
    PC_VectSet(0x08, OSTickISR);
    PC_SetTickRate(OS_TICKS_PER_SEC);                /* Reprogram tick rate                      */
    OS_EXIT_CRITICAL();

    OSStatInit();                                    /* Initialize uC/OS-II's statistics         */

    AckMbox = OSMboxCreate((void *)0);               /* Create 2 message mailboxes               */
    TxMbox  = OSMboxCreate((void *)0);

    TaskStartCreateTasks();                          /* Create all other tasks                   */

    for (;;) {
        TaskStartDisp();                             /* Update the display                       */

        if (PC_GetKey(&key)) {                       /* See if key has been pressed              */
            if (key == 0x1B) {                       /* Yes, see if it's the ESCAPE key          */
                PC_DOSReturn();                      /* Yes, return to DOS                       */
            }
        }

        OSCtxSwCtr = 0;                              /* Clear context switch counter             */
        OSTimeDly(OS_TICKS_PER_SEC);                 /* Wait one second                          */
    }
}
```

Create 2 mailboxes

The dummy loop wait for 'ESC'

Timer drifting

Task1()

```
void  Task1 (void *pdata)
{
    INT8U       err;
    OS_STK_DATA data;                       /* Storage for task stack data    */
    INT16U      time;                       /* Execution time (in uS)  */
    INT8U       i;
    char        s[80];


    pdata = pdata;
    for (;;) {
        for (i = 0; i < 7; i++) {
            PC_ElapsedStart();
            err  = OSTaskStkChk(TASK_START_PRIO + i, &data);
            time = PC_ElapsedStop();
            if (err == OS_NO_ERR) {
                sprintf(s, "%4ld        %4ld        %4ld        %6d",
                        data.OSFree + data.OSUsed,
                        data.OSFree,
                        data.OSUsed,
                        time);
                PC_DispStr(19, 12 + i, s, DISP_FGND_BLACK +
                    DISP_BGND_LIGHT_GRAY);
            }
        }
        OSTimeDlyHMSM(0, 0, 0, 100);            /* Delay for 100 ms */
    }
}
```

Extra overheads on measurement

Task1: total 1024 Free 654 Used 370

Task2() & Task3()

```
void  Task2 (void *data)
{
    data = data;
    for (;;) {
        PC_DispChar(70, 15, '|',  DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(10);
        PC_DispChar(70, 15, '/',  DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(10);
        PC_DispChar(70, 15, '-',  DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(10);
        PC_DispChar(70, 15, '\\', DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(10);
    }
}
```

Timer drifting

```
void  Task3 (void *data)
{
    char    dummy[500];
    INT16U  i;


    data = data;
    for (i = 0; i < 499; i++) {        /* Use up the stack with 'junk'
 */
        dummy[i] = '?';
    }
    for (;;) {
        PC_DispChar(70, 16, '|',  DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(20);
        PC_DispChar(70, 16, '\\', DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(20);
        PC_DispChar(70, 16, '-',  DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(20);
        PC_DispChar(70, 16, '/',  DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDly(20);
    }
}
```


Task4() and Task5()

```
void  Task4 (void *data)
{
    char   txmsg;
    INT8U  err;


    data  = data;
    txmsg = 'A';
    for (;;) {
        OSMboxPost(TxMbox, (void *)&txmsg);      /* Send message to Task #5   */
        OSMboxPend(AckMbox, 0, &err);            /* Wait for acknowledgement from Task #5 */
        txmsg++;                                 /* Next message to send      */
        if (txmsg == 'Z') {
            txmsg = 'A';                         /* Start new series of messages */
        }
    }
}

void  Task5 (void *data)
{
    char  *rxmsg;
    INT8U  err;


    data = data;
    for (;;) {
        rxmsg = (char *)OSMboxPend(TxMbox, 0, &err);          /* Wait for message from Task #4 */
        PC_DispChar(70, 18, *rxmsg, DISP_FGND_YELLOW + DISP_BGND_BLUE);
        OSTimeDlyHMSM(0, 0, 1, 0);                            /* Wait 1 second                 */
        OSMboxPost(AckMbox, (void *)1);                       /* Acknowledge reception of msg  */
    }
}
```

Mail Box

- A mailbox is for data exchanging between tasks.
  - A mailbox consists of a data pointer and a wait-list.
- OSMboxPend():
  - The message in the mailbox is retrieved.
  - If the mailbox is empty, the task is immediately **blocked** and moved to the wait-list.
  - A time-out value can be specified.
- OSMboxPost():
  - A message is posted in the mailbox.
  - If there is already a message in the mailbox, then an error is returned (not overwritten).
  - If tasks are waiting for a message from the mailbox, then the task with the highest priority is removed from the wait-list and scheduled to run.

OSTaskStkInit_FPE_x86()

- OSTaskStkInit_FPE_x86(&ptos, &pbos, &size)
  - Pass the original top address, the original bottom address, and the size of the stack.
  - On the return, arguments are modified, and some stack space are reserved for the floating point library.

OSCreateTaskExt()

```
OSTaskCreateExt(
  TaskStart,
  (void *)0,
  ptos,
  TASK_START_PRIO,
  TASK_START_ID,
  pbos,
  size,
  (void *)0,
  OS_TASK_OPT_STK_CHK |
  OS_TASK_OPT_STK_CLR
  );
```


OSTaskStkCheck()

- Check for any stack overflow
  - bos < (tos – stack length)
  - Local variables, arguments for procedure calls, and temporary storage for ISR’s.
  - *u*C/OS-II can check for any stack overflow for the creation of tasks and when OSTaskStkCheck() is called.
  - *u*C/OS-II does not automatically check for the status of stacks.

Example2: Stack Checking

- Summary:
  - Local variable, function calls, and ISR's will utilize the stack space of user tasks.
    - ISR will use the stack of the interrupted task.
  - If floating-point operations are needed, then some stack space should be reserved.
  - Mailboxes can be used to synchronize the work of tasks.

Example 3: Extension of *u*C/OS-II

- A Pointer to from the TCB of each task to a user-provided data structure
  - Passing user-specified data structures on task creations or have application-specific usage.
- Message queues
  - More than one potiners
- Demonstration on how to use OS hooks to receive/process desired event from the *u*C/OS-II

Example 3: Extension of *u*C/OS-II


```
#define         TASK_STK_SIZE     512               /* Size of each task's stacks (# of WORDs)       */

#define         TASK_START_ID       0               /* Application tasks                             */
#define         TASK_CLK_ID         1
#define         TASK_1_ID           2
#define         TASK_2_ID           3
#define         TASK_3_ID           4
#define         TASK_4_ID           5
#define         TASK_5_ID           6

#define         TASK_START_PRIO    10               /* Application tasks priorities                  */
#define         TASK_CLK_PRIO      11
#define         TASK_1_PRIO        12
#define         TASK_2_PRIO        13
#define         TASK_3_PRIO        14
#define         TASK_4_PRIO        15
#define         TASK_5_PRIO        16

#define         MSG_QUEUE_SIZE     20               /* Size of message queue used in example         */

typedef struct {
    char    TaskName[30];
    INT16U  TaskCtr;
    INT16U  TaskExecTime;
    INT32U  TaskTotExecTime;
} TASK_USER_DATA;


OS_STK          TaskStartStk[TASK_STK_SIZE];        /* Startup    task stack                         */
OS_STK          TaskClkStk[TASK_STK_SIZE];          /* Clock      task stack                         */
OS_STK          Task1Stk[TASK_STK_SIZE];            /* Task #1    task stack                         */
OS_STK          Task2Stk[TASK_STK_SIZE];            /* Task #2    task stack                         */
OS_STK          Task3Stk[TASK_STK_SIZE];            /* Task #3    task stack                         */
OS_STK          Task4Stk[TASK_STK_SIZE];            /* Task #4    task stack                         */
OS_STK          Task5Stk[TASK_STK_SIZE];            /* Task #5    task stack                         */

TASK_USER_DATA  TaskUserData[7];

OS_EVENT       *MsgQueue;                           /* Message queue pointer                         */
void           *MsgQueueTbl[20];                    /* Storage for messages                          */
```

User-defined data structure to pass to tasks

Message queue and an array of event

```
void  Task1 (void *pdata)
{
   char  *msg;
   INT8U  err;


   pdata = pdata;
   for (;;) {
      msg = (char *)OSQPend(MsgQueue, 0, &err);
      PC_DispStr(70, 13, msg, DISP_FGND_YELLOW + DISP_BGND_BLUE);
      OSTimeDlyHMSM(0, 0, 0, 100);
   }
}


void  Task2 (void *pdata)
{
   char  msg[20];


   pdata = pdata;
   strcpy(&msg[0], "Task 2");
   for (;;) {
      OSQPost(MsgQueue, (void *)&msg[0]);
      OSTimeDlyHMSM(0, 0, 0, 500);
   }
}
```

Task 2, 3, 4 are functionally identical.

Message Queues

- A message queue consists of an array of elements and a wait-list.
- Different from a mailbox, a message queue can hold many data elements (in a FIFO basis).
- As same as mailboxes, there can be multiple tasks pend/post to a message queue.
- OSQPost(): a message is appended to the queue. The highest-priority pending task (in the wait-list) receives the message and is scheduled to run, if any.
- OSQPend(): a message is removed from the array of elements. If no message can be retrieved, the task is moved to the wait-list and becomes blocked.

Hooks

- A hook function will be called by *u*C/OS-II when the corresponding event occurs.
  - Event handlers could be in user programs.
  - For example, OSTaskSwHook () is called every time when context switch occurs.
- The hooks are specified in the compiling time in *u*C/OS-II:
  - *u*C/OS-II is an embedded OS.
    - OS_CFG.H (OS_CPU_HOOKS_EN = 0)
  - Many OS's can register and un-register hooks.

User Customizable Hooks for ***u*C/OS-II**

void OSInitHookBegin (void)
void OSInitHookEnd (void)
void OSTaskCreateHook (OS_TCB *ptcb)
void OSTaskDelHook (OS_TCB *ptcb)
void OSTaskIdleHook (void)
void OSTaskStatHook (void)
void OSTaskSwHook (void)
void OSTCBInitHook (OS_TCB *ptcb)
void OSTimeTickHook (void)

```
void  OSTaskStatHook (void)
{
    char    s[80];
    INT8U   i;
    INT32U  total;
    INT8U   pct;


    total = 0L;                                     /* Totalize TOT. EXEC. TIME for each task */
    for (i = 0; i < 7; i++) {
        total += TaskUserData[i].TaskTotExecTime;
        DispTaskStat(i);                            /* Display task data                      */
    }
    if (total > 0) {
        for (i = 0; i < 7; i++) {                   /* Derive percentage of each task         */
            pct = 100 * TaskUserData[i].TaskTotExecTime / total;
            sprintf(s, "%3d %%", pct);
            PC_DispStr(62, i + 11, s, DISP_FGND_BLACK + DISP_BGND_LIGHT_GRAY);
        }
    }
    if (total > 1000000000L) {                      /* Reset total time counters at 1 billion */
        for (i = 0; i < 7; i++) {
            TaskUserData[i].TaskTotExecTime = 0L;
        }
    }
}

void  OSTaskSwHook (void)
{
    INT16U          time;
    TASK_USER_DATA  *puser;


    time  = PC_ElapsedStop();                   /* This task is done                          */
    PC_ElapsedStart();                          /* Start for next task                        */
    puser = OSTCBCur->OSTCBExtPtr;              /* Point to used data                         */
    if (puser != (TASK_USER_DATA *)0) {
        puser->TaskCtr++;                       /* Increment task counter                     */
        puser->TaskExecTime     = time;         /* Update the task's execution time           */
        puser->TaskTotExecTime += time;         /* Update the task's total execution time     */
    }
}
```

Elapsed time for the current task

OSTCBCur → TCB of the current task
OSTCBHighRdy → TCB of the new task

Example 3: Extension of *u*C/OS-II

- Summary:
  - Message queues can be used to synchronize among tasks.
    - Multiple messages can be held in a queue.
    - Multiple tasks can “pend”/“post” to message queues simultaneously.
  - Hooks can be used to do some user-specific computations on certain OS events occurs.
    - They are specified in the compiling time.
- A Pointer to from the TCB of each task to a user-provided data structure

Introduction

- Getting Started with *u*C/OS-II:
  - How to write a dummy *u*C/OS-II program?
  - How the control flows among procedures?
  - How tasks are created?
  - How tasks are synchronized by semaphore, mailbox, and message queues?
  - How the space of a stack is utilized?
  - How to capture system events?

Contents

- Introduction
- Kernel Structure

Objectives

- To understand what a task is.
- To learn how *u*C/OS-II manages tasks.
- To know how an interrupt service routine (ISR) works.
- To learn how to determine the percentage of CPU that your application is using.

The *u*C/OS-II 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

Software

Hardware

CPU

Timer

Source Availability

- Download the “Ports” of *u*C/OS-II from the web site http://www.ucos-II.com/
  - Processor-independent and dependent code sections (for Intel 80x86) are contained in the companion CD-ROM of the textbook

Critical Sections

- A *critical section* is a portion of code that is not safe from race conditions because of the use of shared resources.
- They can be protected by interrupt disabling/enabling interrupts or semaphores.
  - The use of semaphores often imposes a more significant amount of overheads.
  - A RTOS often use interrupts disabling/ enabling to protect critical sections.
- Once interrupts are disabled, neither context switches nor any other ISR’s can occur.

Critical Sections

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

- Interrupt latency is vital to an RTOS!
  - Interrupts should be disabled as short as possible to improve the responsiveness.
  - It must be accounted as a blocking time in the schedulability analysis.
- Interrupt disabling must be used carefully:
  - E.g., if OSTimeDly() is called with interrupt disabled, the machine might hang!

Critical Sections

- The states of the processor must be carefully maintained across multiple calls of OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL().
- There are three implementations in *u*C/OS-II:
  - Interrupt enabling/disabling instructions.
  - Interrupt status save/restore onto/from stacks.
  - Processor Status Word (PSW) save/restore onto/from memory variables.
- Interrupt enabling/disabling can be done by various way:
  - In-line assembly.
  - Compiler extension for specific processors.

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”.
  - Interrupt statuses might not be consistent across kernel services/function calls!!

```
{
    .
    disable_interrupt();
    a_kernel_service();
    .
    .
}
```

```
{
    .
    disable_interrupt();
    critical section
    enable_interrupt();
    .
}
```

Interrupts are now implicitly re-enabled!

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”)
```

Some compilers might not be smart enough to adjust the stack pointer after the processing of in-line assembly.

Critical Sections

- OS_CRITICAL_METHOD=3
- The compiler and processor allow the PSW to be saved/restored to/from a memory variable.

```
void foo(arguments)
{
    OS_CPU_SR cpu_sr;
    .
    cpu_sr = get_processor_psw();
    disable_interrupts();
    .
    /* critical section */
    .
    set_processor_psw(cpu_sr);
    .
}
```

OS_ENTER_CRITICAL()

OS_EXIT_CRITICAL()

Tasks

- A task is an active entity which could do some computations.
- Under real-time *u*C/OS-II systems, a 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 the next event/period, so that other tasks can run.

Tasks

- *u*C/OS-II can have up to 64 priorities.
  - Each task must be associated with an unique priority.
  - 63 and 62 are reserved (idle, stat).
- An insufficient number of priority might damage the schedulability of a real-time scheduler.
  - The number of schedulable task would be reduced.
    - Because there is no distinction among the tasks with the same priority.
    - For example, under RMS, tasks have different periods but are assigned with the same priority.
    - It is possible that all other tasks with the same priority are always issued before a particular task.
  - Fortunately, most 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 it is done.

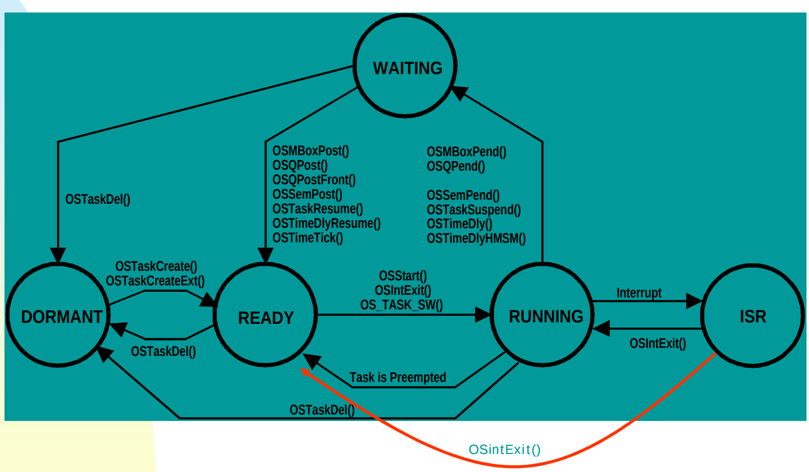
```
void YourTask (void *pdata)
{
   /* USER CODE */
   OSTaskDel(OS_PRIO_SELF);
}
```

The priority of the current task

Task States

- Dormant: Procedures residing on RAM/ROM is not an task unless you call OSTaskCreate() to execute them.
  - No tasks correspond to the codes yet!
- 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 scheduled to run on the CPU .
  - There must be only one running task.
  - The task running might be preempted and become ready.
- 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



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 the execution of 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).

Task States

- The running task could be preempted by an ISR 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, e.g., timer expiration).
- 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 main-memory-resident data structure used to maintain the state of a task, especially when it is preempted.
- 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, the 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;
```

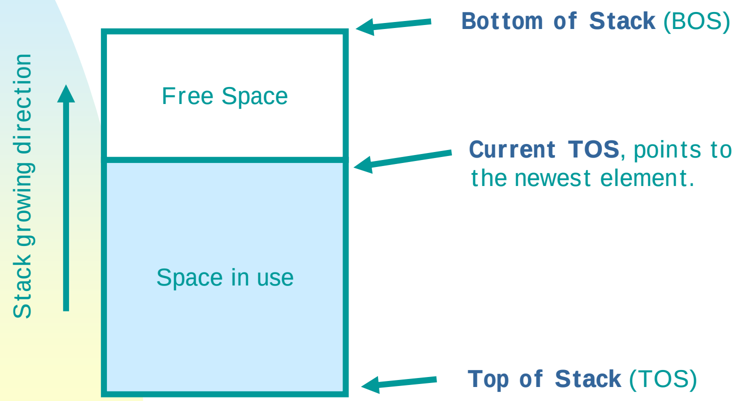
Task Control Blocks (TCB)

- .OSTCBStkPtr contains a pointer to the current TOS for the task.
  - It is the first entry of TCB so that it can be accessed directly from 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
  - The pointer is ordinarily cleared in the hook OSTaskDelHook().

Task Control Blocks (TCB)

- .OSTCBStkBottom is a pointer to the bottom of the task's stack.
- .OSTCBStkSize holds the size of the stack in the number of elements, instead of bytes.
  - The element size is a macro OS_STK.
  - The total stack size is OSTCBStkSize*OS_STK bytes
  - .OSTCBStkBottom and .OSTCBStkSize are used to check up stacks (if OSTaskCreateExt( ) is invoked).

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 enabled for the task .
  - OS_TASK_OPT_STK_CLR: indicates that the stack needs to be cleared when the task is created.
  - OS_TASK_OPT_SAVE_FP: Tell 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**: hold an identifier for the task.
- **.OSTCBNext** and **.OSTCBPrev** are used to doubly link OS_TCB's

Task Control Blocks (TCB)

- **.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 info regarding 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 wait 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)

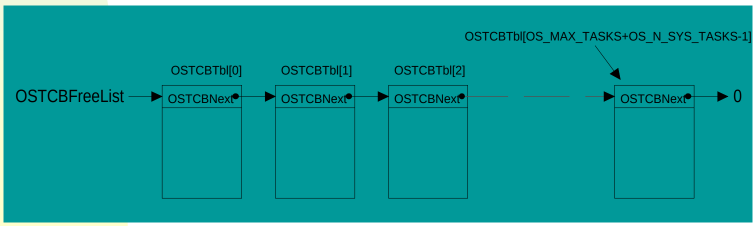
- **.OSTCBX .OSTCBY .OSTCBBitX** and **.OSTCBBitY**
  - They are used to accelerate the process of making a task ready to run or make a task wait for an event.

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

- **.OSTCBDelReq** is a boolean used to indicate whether or not a task requests that the current task to be deleted.
- OS_MAX_TASKS is specified in OS_CFG.H
  - # OS_TCB's allocated by *u*C/OS-**II**
- **OSTCBTbl**[ ] : where all OS_TCB's are placed.
- When *u*C/OS-II is initialized, all OS_TCB's in the table are linked in a singly linked list of free OS_TCB's.

Task Control Blocks (TCB)

- When a task is created, the OS_TCB pointed to by OSTCBFreeList is assigned to the task, and OSTCBFreeList is adjusted to point to the next OS_TCB in the chain.
- When a task is deleted, its OS_TCB is returned to the list of free OS_TCB.
- 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.

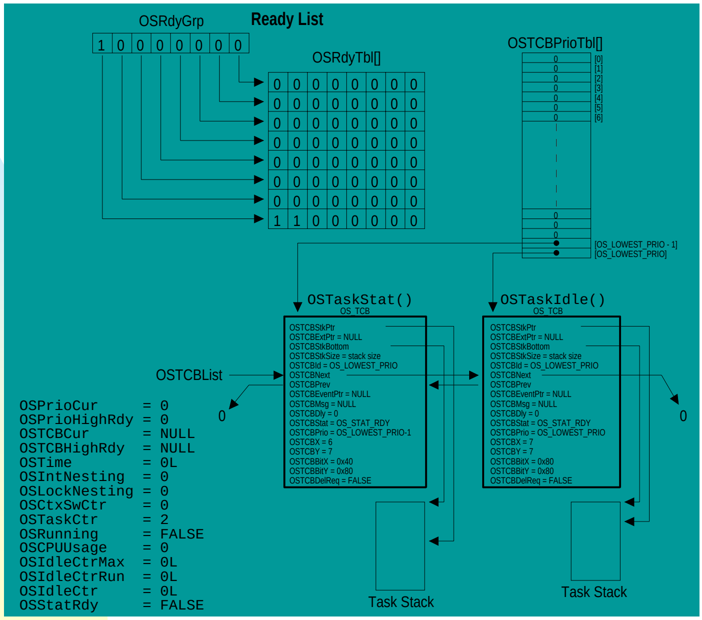
Priority table

TCB list

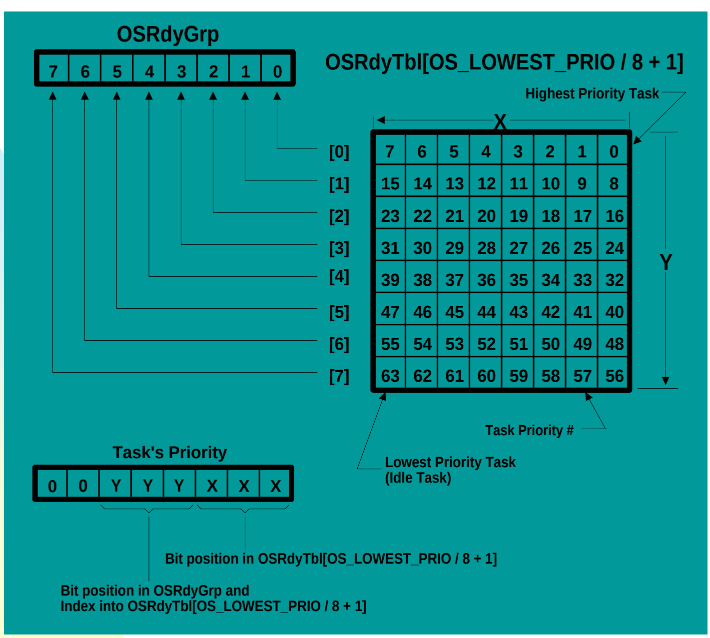
Ready list

Ready List

- Ready list is a special bitmap to reflect which task is currently in the ready state.
  - Each task is identified by its unique priority in the bitmap.
- A primary design consideration of the ready list is how to efficiently locate the highest-priority ready task.
  - The designer could trade some ROM space for an improved performance.
- If a linear list is adopted, it takes O(n) to locate the highest-priority ready task.
  - It takes O(log n) if a heap is adopted.
  - Under the design of ready list of *u*C/OS-II, it takes only O(1).
    - Note that the space consumption is much more than other approaches, and it also depends on the bus width.



Ready List
OSRdyGrp
OSRdyTbl[]
OSTCBPrioTbl[]
[OS_LOWEST_PRIO - 1]
[OS_LOWEST_PRIO]
OSTaskStat()
OSTaskIdle()
OS_TCB
OSTCBList
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



OSRdyGrp
OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]
Highest Priority Task
X
Y
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?

Coding Style

The author writes:

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

How about this:

```
char x,y,mask;

    x = prio & 0x07;
    y = prio >> 3;
    mask = ~(OSMapTbl[x]);          // a mask for bit clearing
    if((OSRdyTbl[x] &= mask) == 0) // clear the task's bit
    {                        // the group bit should be cleared too
        mask = ~(OSMapTbl[y]);      // another bit mask…
        OSRdyGrp &= mask;           // clear the group bit
    }
```

Coding Style

```
mov     al,byte ptr [bp-17]
mov     ah,0
and     ax,7
lea     dx,word ptr [bp-8]
add     ax,dx
mov     bx,ax
mov     al,byte ptr ss:[bx]
not     al
mov     dl,byte ptr [bp-17]
mov     dh,0
sar     dx,3
lea     bx,word ptr [bp-16]
add     dx,bx
mov     bx,dx
and     byte ptr ss:[bx],al
mov     al,byte ptr ss:[bx]
or      al,al
jne     short @1@86
mov     al,byte ptr [bp-17]
mov     ah,0
sar     ax,3
lea     dx,word ptr [bp-8]
add     ax,dx
mov     bx,ax
mov     al,byte ptr ss:[bx]
not     al
and     byte ptr [bp-18],al
```

```
mov     al,byte ptr [bp-17]
and     al,7
mov     byte ptr [bp-19],al
mov     al,byte ptr [bp-17]
mov     ah,0
sar     ax,3
mov     byte ptr [bp-20],al
mov     al,byte ptr [bp-19]
mov     ah,0
lea     dx,word ptr [bp-8]
add     ax,dx
mov     bx,ax
mov     al,byte ptr ss:[bx]
not     al
mov     cl,al
mov     al,byte ptr [bp-19]
mov     ah,0
lea     dx,word ptr [bp-16]
add     ax,dx
mov     bx,ax
and     byte ptr ss:[bx],cl
mov     al,byte ptr ss:[bx]
or      al,al
jne     short @1@142
mov     al,byte ptr [bp-20]
mov     ah,0
lea     dx,word ptr [bp-8]
add     ax,dx
mov     bx,ax
mov     al,byte ptr ss:[bx]
not     al
mov     cl,al
and     byte ptr [bp-18],cl
```



```
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.

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.
  - The difference is the saving/restoration of PSW (or CPU flags).
- *u*C/OS-II 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.

Task Scheduling

```
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 done if the scheduler is locked or an ISR is currently serviced.

(2) Find the highest-priority ready task.

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

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

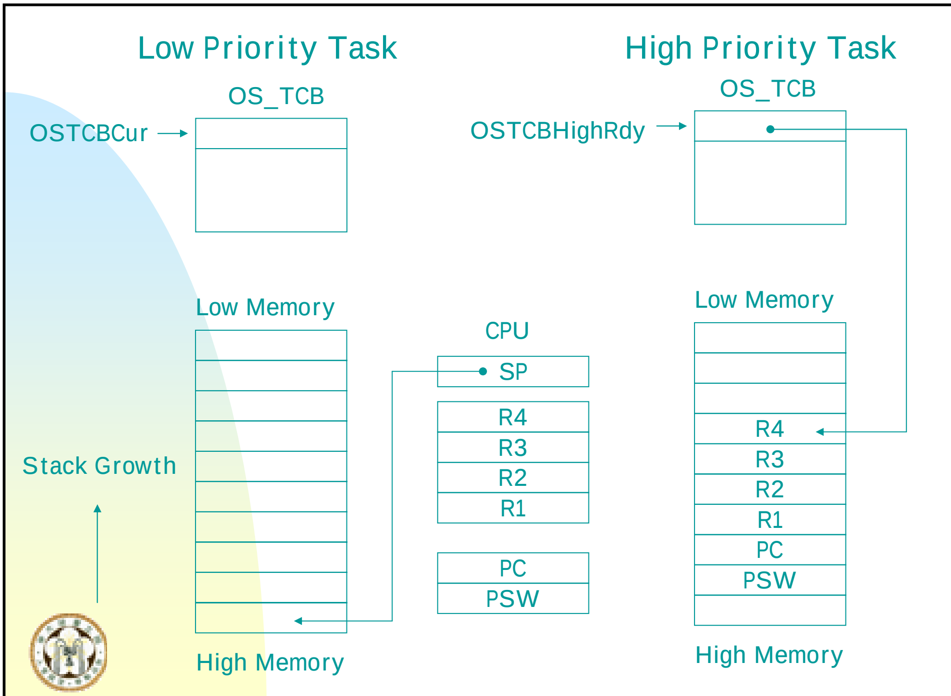
* All rights reserved, Tei-Wei Kuo, National Taiwan University, 2003.

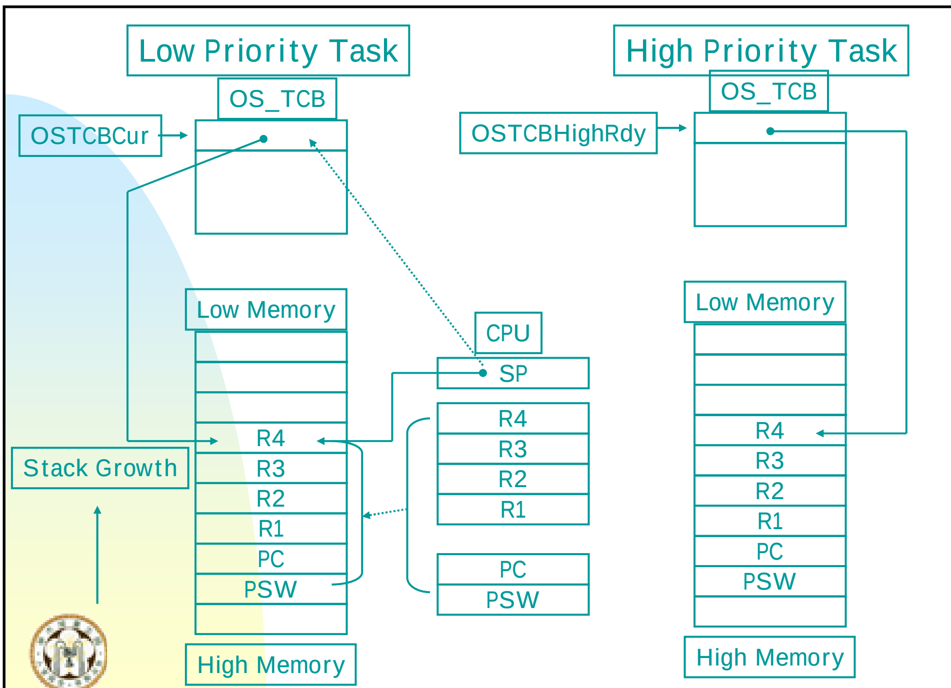
Task Scheduling

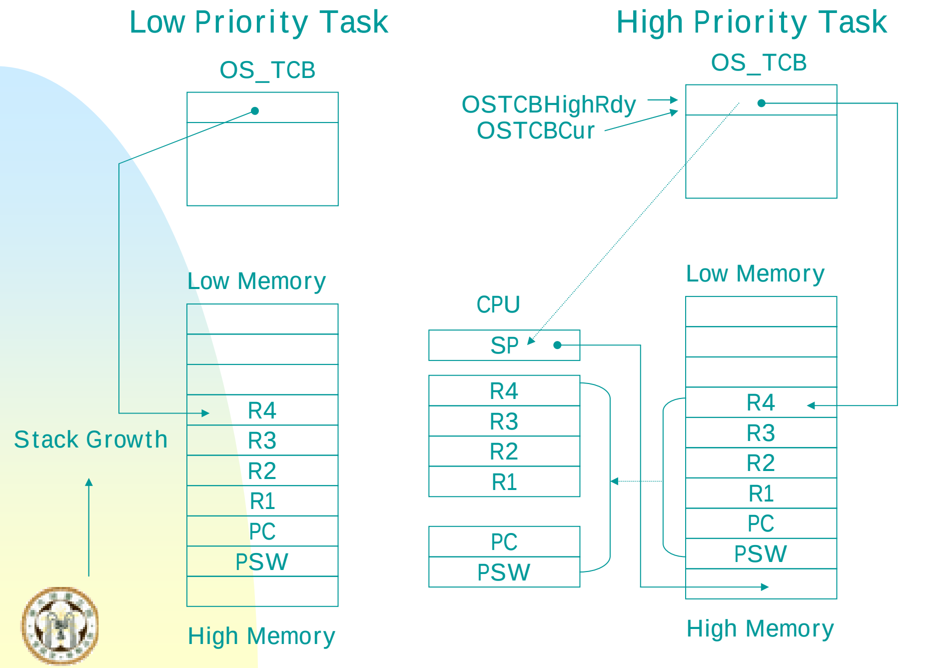
- A context switch must save all CPU registers and PSW of the preempted task onto its stack, and then restore the CPU registers and PSW of the highest-priority ready task from its stack.
- Task-level scheduling will emulate that as if preemption/scheduling is done in an ISR.
  - OS_TASK_SW() will trigger a software interrupt.
  - The interrupt is directed to the context switch handler OSCtxSw(), which is installed when *u*C/OS-II is initialized.
- Interrupts are disabled during the locating of the highest-priority ready task to prevent another ISR's from making some tasks ready.

Task-Level Context Switch

- By default, context switches are handled at the interrupt-level. Therefore task-level scheduling will invoke a software interrupt to emulate that effect:
  - Hardware-dependent! Porting must be done here.


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



Locking and Unlocking the Scheduler

- OSSchedLock() prevents high-priority ready tasks from being scheduled to run while interrupts are still recognized.
  - OSSchedLock() and OSSchedUnlock() must be used in pairs.
  - 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. Otherwise, the system will be locked up.
- Sometimes we disable scheduling (but with interrupts still recognized) because we hope to avoid lengthy interrupt latencies without introducing race conditions.
- OSLockNesting keeps track of the number of OSSchedLock() has been called.

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();
        }
    }
}
```

Idle Task

- The idle task is always the lowest-priority task and can not be deleted or suspended by user-tasks.
- To reduce power dissipation, you can issue a HALT-like instruction in the idle task.
  - Suspend services in OSTaskIdleHook()!!

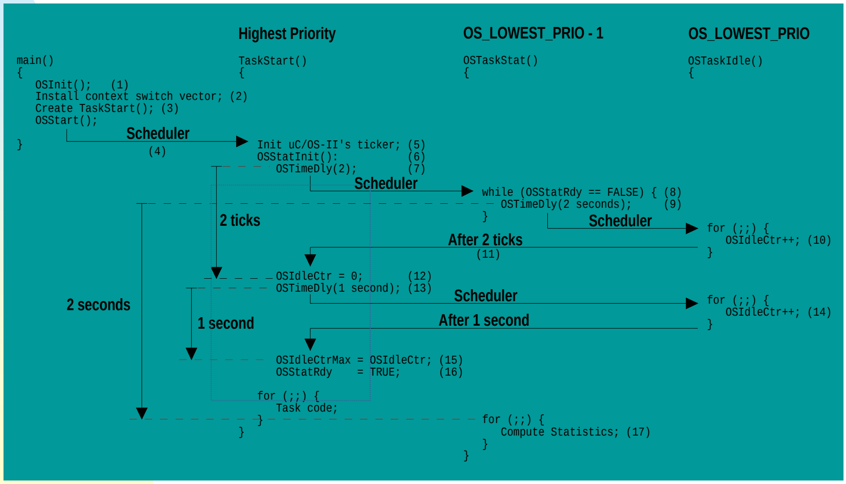
```
void  OS_TaskIdle (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
   OS_CPU_SR  cpu_sr;
#endif


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


Statistics Task

- It is created by *u*C/OS-II. It executes every second to compute the percentage of the CPU usage.
  - It is with the OS_LOWEST_PRIO – 1 priority.
- OSStatInit() must be called before OSStart() is called.

```
void main (void)
{
    OSInit();                 /* Initialize uC/OS-II                  (1)*/
    /* Install uC/OS-II's context switch vector                          */
    /* Create your startup task (for sake of discussion, TaskStart())  (2)*/
    OSStart();                /* Start multitasking                   (3)*/
}
void TaskStart (void *pdata)
{
    /* Install and initialize µC/OS-II’s ticker                       (4)*/
    OSStatInit();             /* Initialize statistics task           (5)*/
    /* Create your application task(s)                                   */
    for (;;) {
        /* Code for TaskStart() goes here!                               */
    }
}
```

Statistics Task



Statistics Task

(7) TaskStart: Delay for 2 ticks→ transfer CPU to the statistics task to do some initialization.

(9) OS_TaskStat: Delay for 2 seconds→ Yield the CPU to the task TaskStart and the idle task.

(13) TaskStart: Delay for 1 second→ Let the idle task count OSIdleCtr for 1 second.

(15) TaskStart: When the timer expires in (13), OSIdleCtr contains the value of OSIdleCtr can be reached in 1 second.

Notes:

- Since OSStatinit() assume that the idle task will count the OSOdleCtr at the full CPU speed, you must not install an idle hook before calling OSStatInit()!!!
- After the statistics task is initialized, it is OK to install a CPU idle hook and perform some power-conserving operations! Note that the idle task consumes the CPU power just for the purpose of being idle.

Statistics Task

- With the invocation of OSStatInit(), we have known how large the value of the idle counter can reach in 1 second (OSIdleCtrMax).
- The percentage of the CPU usage can be calculated by the actual idle counter and the OSIdleCtrMax.

$$OSCPUUsage_{(\%)} = 100 \times \left(1 - \frac{OSIdleCtr}{OSIdleCtrMax}\right)$$

This term is always 0 under an integer operation

$$OSCPUUsage_{(\%)} = \left(100 - \frac{100 \times OSIdleCtr}{OSIdleCtrMax}\right)$$

$$OSCPUUsage_{(\%)} = \left(100 - \frac{OSIdleCtr}{\left(\frac{OSIdleCtrMax}{100}\right)}\right)$$

This term might overflow under fast processors! (42,949,672)

Statistics Task

```
#if OS_TASK_STAT_EN > 0
void  OS_TaskStat (void *pdata)
{
#if OS_CRITICAL_METHOD == 3
   OS_CPU_SR  cpu_sr;
#endif
   INT32U     run;
   INT32U     max;
   INT8S      usage;


   pdata = pdata;
   while (OSStatRdy == FALSE) {
      OSTimeDly(2 * OS_TICKS_PER_SEC);
   }
   max = OSIdleCtrMax / 100L;
   for (;;) {
      OS_ENTER_CRITICAL();
      OSIdleCtrRun = OSIdleCtr;
      run          = OSIdleCtr;
      OSIdleCtr    = 0L;
      OS_EXIT_CRITICAL();
      if (max > 0L) {
         usage = (INT8S)(100L - run / max);
         if (usage >= 0) {
            OSCPUUsage = usage;
         } else {
            OSCPUUsage = 0;
         }
      } else {
         OSCPUUsage = 0;
         max        = OSIdleCtrMax / 100L;
      }
      OSTaskStatHook();
      OSTimeDly(OS_TICKS_PER_SEC);
   }
}
```

Interrupts under *u*C/OS-II

- *u*C/OS-II requires an ISR being written in assembly if your compiler does not support in-line assembly!

An ISR Template:

```
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 *u*C/OS-II

(1) In an ISR, *u*C/OS-II requires that all CPU registers are saved onto the stack of the interrupted task.
- For processors like Motorola 68030_, a different stack is used for ISR.
- For such a case, the stack pointer of the interrupted task can be obtained from OSTCBCur (offset 0).

(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.

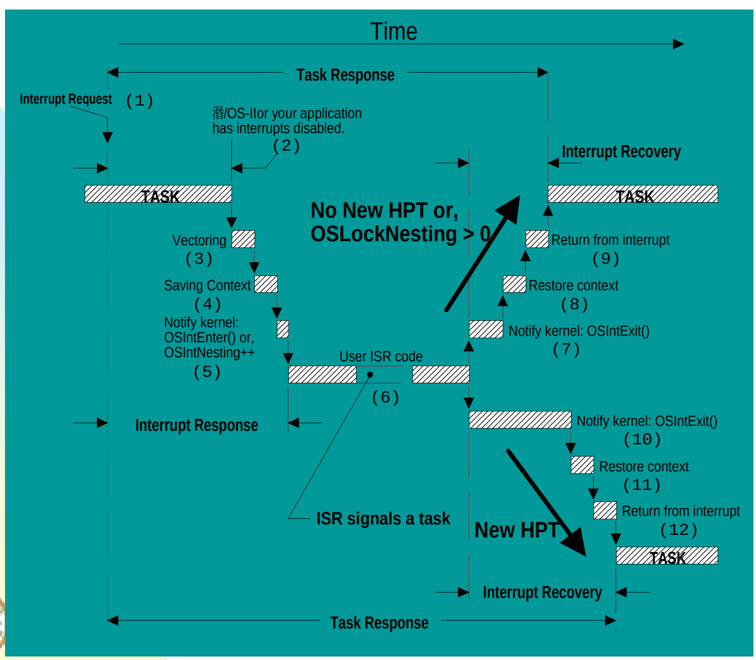
Interrupts under *u*C/OS-II

(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.
- The Interrupt-nesting counter is decremented.

(9) On the return from this point, there might be several high-priority tasks since *u*C/OS-II is a preemptive kernel.

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



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 no scheduler locking and no interrupt nesting

If there is another high-priority task ready

A context switch is executed.

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

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


Clock Tick

- A time source is needed to keep track of time delays and timeouts.
- You must enable ticker interrupts after multitasking is started.
  - In the TaskStart() task of the examples.
  - Do not do this before OSStart().
- Clock ticks are serviced by calling OSTimeTick() from a 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 *u*C/OS-II?

Clock Tick

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

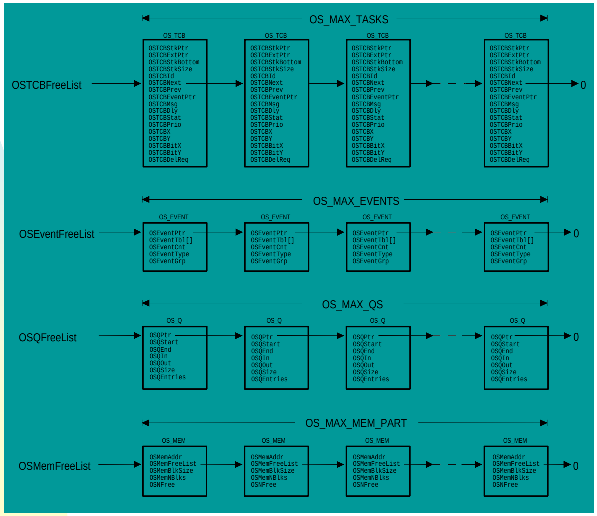
```
void OSTickISR(void)
{
    Save processor registers;
    Call OSIntEnter() or increment OSIntNesting;
    If(OSIntNesting == 1)
    OSTCBCur->OSTCBStkPtr = SP;

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

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

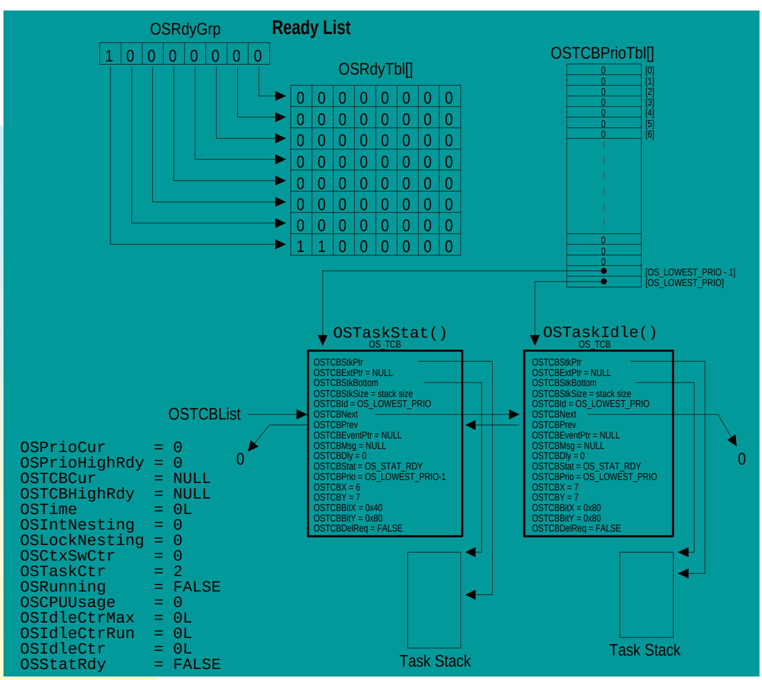
Post a message

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

Do the rest of the job!

*u*C/OS-II Initialization





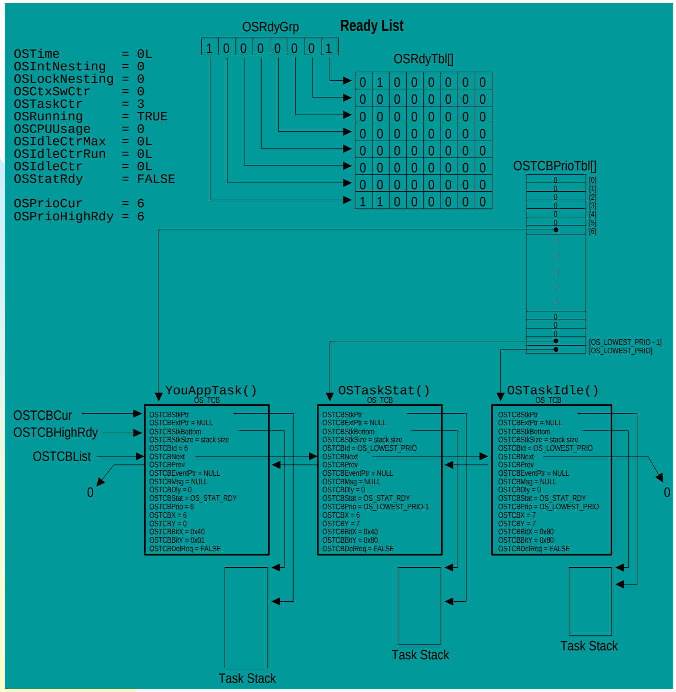
Starting of *u*C/OS-II

- OSInit() initializes data structures for *u*C/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()).

Starting of *u*C/OS-II

```
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



Summary

- The study of the *u*C/OS-II kernel structure, we learn something:
  - What a task is? How *u*C/OS-II manages a task and related data structures.
  - How the scheduler works, especially on detailed operations done for context switching.
  - The responsibility of the idle task and the statistics task! How they works?
  - How interrupts are serviced in *u*C/OS-II.
  - The initialization and starting of *u*C/OS-II.