# *~~~MicroC/OS-II Scheduling and Context Switching~~~*

MicroC/OS-II uses the preemptive priority-based scheduling algorithm. Each task is assigned a priority value and the task with the highest priority that is ready to run is selected for execution. Using this algorithm ensures that higher-priority task preempts lower-priority task. Here the priority-based algorithm is performed by using the priority-based data structures. The main data structure that is used here is the **ReadyList**. The index of the ReadyList represents its priority value. **OSRdyGrp** represents the Ready Group or the ready list itself. It uses a bitmap to represent the readiness of tasks at the reach priority level. Each bit in the bitmap corresponds to a specific priority level and a set bit indicates that there is at least one ready task at that priority level. The task level scheduling in uC/OS-II is performed by **OS_Sched()**. It is important to note that the ISR scheduling is done by **ISIntExit()**.

```
void  OS_Sched (void)
{
#if OS_CRITICAL_METHOD == 3u                           /* Allocate storage for CPU status register     */
    OS_CPU_SR  cpu_sr = 0u;
#endif



    OS_ENTER_CRITICAL();
    if (OSIntNesting == 0u) {                          /* Schedule only if all ISRs done and ...       */
        if (OSLockNesting == 0u) {                     /* ... scheduler is not locked                  */
            OS_SchedNew();
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            if (OSPrioHighRdy != OSPrioCur) {          /* No Ctx Sw if current task is highest rdy     */
#if OS_TASK_PROFILE_EN > 0u
                OSTCBHighRdy->OSTCBCtxSwCtr++;         /* Inc. # of context switches to this task      */
#endif
                OSCtxSwCtr++;                          /* Increment context switch counter             */

#if OS_TASK_CREATE_EXT_EN > 0u
#if defined(OS_TLS_TBL_SIZE) && (OS_TLS_TBL_SIZE > 0u)
                OS_TLS_TaskSw();
#endif
#endif

                OS_TASK_SW();                          /* Perform a context switch                     */
            }
        }
    }
    OS_EXIT_CRITICAL();
}
```

Let us understand what is happening during the Scheduling.

**OS_Sched()** determines the priority of the highest priority task that is in ready to run state. " After the highest priority task has been found, OS_Sched() verifies that the highest priority task is not the current task. Verification is done to avoid an unnecessary context switch, which would be time-consuming." Context switching involves saving the processor registers on stack of the task that is being suspended and restoring the registers of the highest priority task from the stack. The value 3u indicates that a global variable is used to store the CPU status register during critical sections.

**OS_ENTER_CRITICAL** disables the interrupts and ensures the code written next to it executes without interruption. Next, it checks if there are no nested interrupts currently being processed and also makes sure that the scheduler is called only when all interrupts have been handled. Since context switching is frequent, the condition **OSLockNesting** checks that the scheduler is not locked. If the schedule is locked, the OS prevents context switches. The **OS_SchedNew()** is then called. It determines the highest priority task that is ready to run.

**OS_SchedNew()**

```
static  void  OS_SchedNew (void)
{
#if OS_LOWEST_PRIO <= 63u                        /* See if we support up to 64 tasks                   */
    INT8U   y;


    y             = OSUnMapTbl[OSRdyGrp];
    OSPrioHighRdy = (INT8U)((y << 3u) + OSUnMapTbl[OSRdyTbl[y]]);
#else                                            /* We support up to 256 tasks                         */
    INT8U     y;
    OS_PRIO  *ptbl;


    if ((OSRdyGrp & 0xFFu) != 0u) {
        y = OSUnMapTbl[OSRdyGrp & 0xFFu];
    } else {
        y = OSUnMapTbl[(OS_PRIO)(OSRdyGrp >> 8u) & 0xFFu] + 8u;
    }
    ptbl = &OSRdyTbl[y];
    if ((*ptbl & 0xFFu) != 0u) {
        OSPrioHighRdy = (INT8U)((y << 4u) + OSUnMapTbl[(*ptbl & 0xFFu)]);
    } else {
        OSPrioHighRdy = (INT8U)((y << 4u) + OSUnMapTbl[(OS_PRIO)(*ptbl >> 8u) & 0xFFu] + 8u);
    }
#endif
}
```

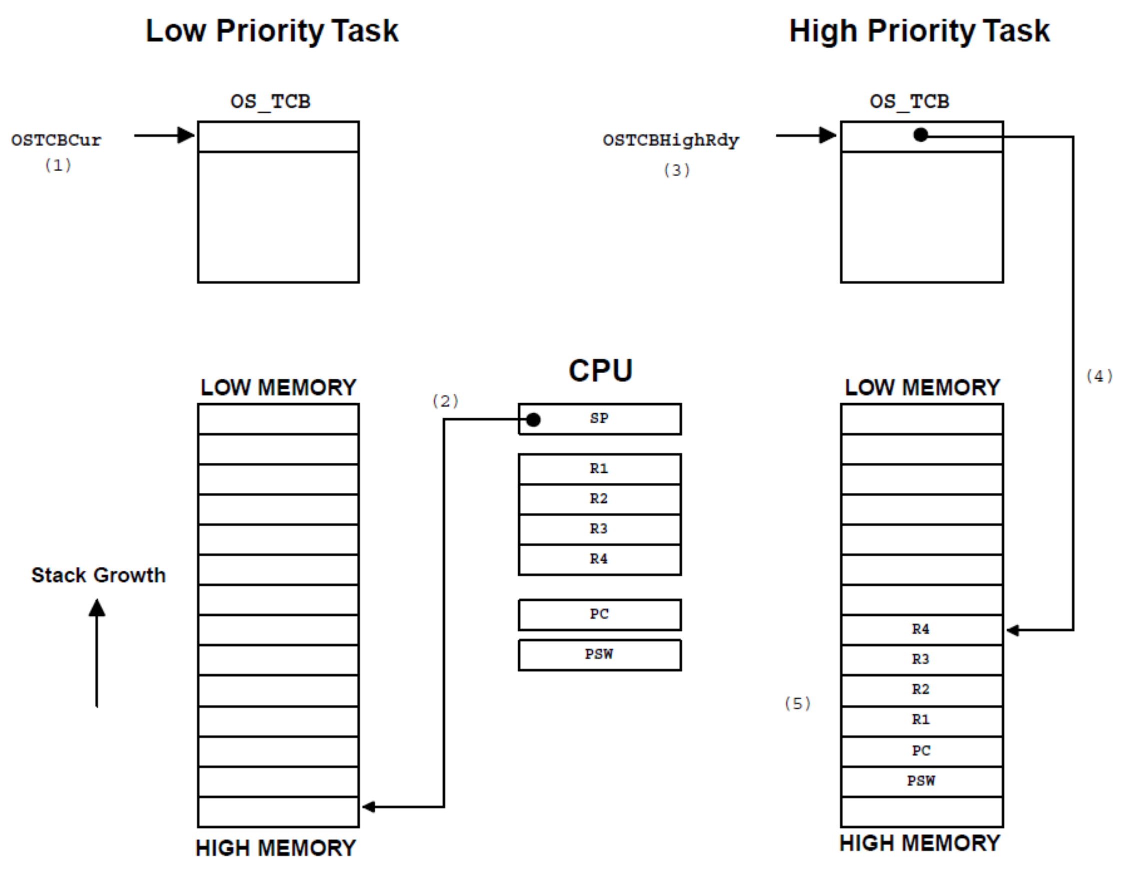
If the number of priorities is up to 63, check that the lower 8 bits of **OSRdyGrp** is not equal to zero. Then, from the **OSUnMapTbl** array, extract those bits and stored in the variable ‘y’. It is used to determine the specific mapping for a subset of bits in **OSRdyGrp** that is used for priority assignment. The **OSPrioHighRdy** is calculated by shifting ‘y’ left by 3 and adding the value to the corresponding value in **OSRdyTbl**.

Then, If the lower 8 bits of the value pointed to by **ptbl** are non-zero, **OSPrioHighRdy** is calculated by shifting y left by 4 and adding the value of the **OSUnMapTbl** entry corresponding to the lower 8 bits of ***ptbl**. Otherwise, **OSPrioHighRdy** is calculated by shifting y left by 4, adding the value of the **OSUnMapTbl** entry corresponding to the upper 8 bits of ***ptbl** shifted right by 8, and incremented by 8.

After the above operation, i.e. determining which task has the highest priority, the highest priority ready task is assigned to **OSTCBHighRdy**. This pointer points to the TCB of the task with the highest priority. Then, it is checked that if the highest priority task has the same priority as the priority of the current running task, if the priority is the same, no context switch is needed. To keep the track of number of times the context switch operation, the counter **OSTCBCtxSwCtr** is incremented once after every context switch. To perform a context switch **OS_TASK_SW()** is called. **OS_TASK_SW** is defined to **OSCtxSw**() in uC/OS-II.

Following is an illustration of what happens when **OS_TASK_SW()** is called:

*Figure 3.6 µC/OS-II structures when* `OS_TASK_SW()` *is called.*



Let us walk through the **OSCtxSw**() function and understand what is happening in there.

```
void  OSCtxSw (void)
{
    OS_TASK_STK   *p_stk;
    OS_TASK_STK   *p_stk_new;
#if (OS_MSG_TRACE > 0u)
    OS_TCB        *p_tcb_cur;
    OS_TCB        *p_tcb_new;
#endif
    CPU_SR_ALLOC();


#if (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_STATUS_LOCAL)
    cpu_sr = 0u;
#endif

#if (OS_MSG_TRACE > 0u)
    p_tcb_cur = OSTCBCur;
    p_tcb_new = OSTCBHighRdy;
#endif


    p_stk = (OS_TASK_STK *)OSTCBCur->OSTCBStkPtr;
    OSTaskSwHook();

    OSTCBCur  = OSTCBHighRdy;
    OSPrioCur = OSPrioHighRdy;

    if (p_stk->TaskState == STATE_RUNNING) {
        p_stk->TaskState  = STATE_SUSPENDED;
    }
    p_stk_new = (OS_TASK_STK *)OSTCBHighRdy->OSTCBStkPtr;
    switch (p_stk_new->TaskState) {
        case STATE_CREATED:                                     /* TaskState updated
to STATE_RUNNING once thread runs.     */
             ResumeThread(p_stk_new->ThreadHandle);
                                                                /* Wait while task is
created and until it is ready to run. */
             SignalObjectAndWait(p_stk_new->SignalPtr, p_stk_new->InitSignalPtr, INFINITE,
FALSE);
             break;


        case STATE_SUSPENDED:
             p_stk_new->TaskState = STATE_RUNNING;
             SetEvent(p_stk_new->SignalPtr);
             break;


        case STATE_INTERRUPTED:
             p_stk_new->TaskState = STATE_RUNNING;
             ResumeThread(p_stk_new->ThreadHandle);
             break;


#if (OS_MSG_TRACE > 0u)
        case STATE_NONE:
             OS_Printf("[OSCtxSw] Error: Invalid state STATE_NONE\nCur    Task[%3.1d] Thread
ID %5.0d: '%s'\nNew    Task[%3.1d] Thread ID %5.0d: '%s'\n\n",
                       p_tcb_cur->OSTCBPrio,
                       p_stk->ThreadID,
                       p_tcb_cur->OSTCBTaskName,
                       p_tcb_new->OSTCBPrio,
                       p_stk_new->ThreadID,
                       p_tcb_new->OSTCBTaskName);
             return;


        case STATE_RUNNING:
             OS_Printf("[OSCtxSw] Error: Invalid state STATE_RUNNING\nCur    Task[%3.1d]
Thread ID %5.0d: '%s'\nNew    Task[%3.1d] Thread ID %5.0d: '%s'\n\n",
                       p_tcb_cur->OSTCBPrio,
                       p_stk->ThreadID,
                       p_tcb_cur->OSTCBTaskName,
                       p_tcb_new->OSTCBPrio,
                       p_stk_new->ThreadID,
                       p_tcb_new->OSTCBTaskName);
             return;


        case STATE_TERMINATING:
             OS_Printf("[OSCtxSw] Error: Invalid state STATE_TERMINATING\nCur    Task[%3.1d]
Thread ID %5.0d: '%s'\nNew    Task[%3.1d] Thread ID %5.0d: '%s'\n\n",
                       p_tcb_cur->OSTCBPrio,
                       p_stk->ThreadID,
                       p_tcb_cur->OSTCBTaskName,
                       p_tcb_new->OSTCBPrio,
                       p_stk_new->ThreadID,
                       p_tcb_new->OSTCBTaskName);
             return;


        case STATE_TERMINATED:
             OS_Printf("[OSCtxSw] Error: Invalid state STATE_TERMINATED\nCur    Task[%3.1d]
Thread ID %5.0d: '%s'\nNew    Task[%3.1d] Thread ID %5.0d: '%s'\n\n",
                       p_tcb_cur->OSTCBPrio,
                       p_stk->ThreadID,
                       p_tcb_cur->OSTCBTaskName,
                       p_tcb_new->OSTCBPrio,
                       p_stk_new->ThreadID,
                       p_tcb_new->OSTCBTaskName);
             return;


#endif
        default:
             return;
    }


    if (p_stk->Terminate == DEF_TRUE) {
        OSTaskTerminate(p_stk);

        CPU_CRITICAL_EXIT();

        ExitThread(0u);                                                 /* ExitThread() never
returns.                              */
        return;
    }
    CPU_CRITICAL_EXIT();
    WaitForSingleObject(p_stk->SignalPtr, INFINITE);
    CPU_CRITICAL_ENTER();
}
```

The responsibility of the Context Switcher function is to

“

1) Save processor tegisters then,
2) Save current task’s stack pointer into the current task’s OS_TCB,
3) Call OSTaskSwHook(),
4) Set OSTCBCur = OSPrioHighRdy,
5) Set OSPrioCur = OSPrioHighRd,
6) And finally, switch to the highest priority task. “

The **CPU_CFG_CRITICAL_METHOD** defines the methods in which the critical section is configured. When **CPU_CRITICAL_METHOD_STATUS_LOCAL** is defined then, it Save/Restore the interrupt status to local variable. Here **cpu_sr** is being used which is defined in **CPU_SR_ALLOC**() function that needs to called before using any CPU register.

**p_stk** is the pointer that points to the current stack pointer. The current task control block **OSTCBCur** stack pointer is assigned to a local pointer, to store the current tasks stack pointer. It determines the new highest-priority task **OSTCBHighRdy**, **OSPrioHighRdy** and updates the current task. The function **OSTaskSwHook**() is called just before the actual task switching. It is used to add user custom code just before the context switching, so that, when the task is switched, the user code is also updated into the next task. Ok, but why do I even need to insert code during the context switch?

Consider you have a system that has multiple sensors, each sensor is defined with a specific task, these tasks are scheduled by the OS. When a task switch occurs, I could add the code to ensure the appropriate sensor is activated/deactivated during the currents running task. This also ensures that the sensor/device is synchronized with the execution of the needed task.

Next, if the current task state is running, change the state to suspended. **OSTCBHighRdy** represents the TCB of the next highest priority ready to run TCB and retrieve the stack pointer of the next task. Depending on the next tasks state, the following cases are considered; **STATE_CREATED**, **STATE_SUSPENDED**, **STATE_INTERRUPTED**.

If the case is **STATE_CREATED**, it indicated, the task is created but not yet in run state. It updates the taskState to **STATE_RUNNING** by calling **ResumeThread**() function to allow it to execute. The function **SignalObjectAndWait**() is called to ensure the newly created task is in ready to run state, this indicates a synchronization that a new task is completed its initialization before proceeding further. **ResumeThread**(), **SignalObjectAndWait**() are windows API’s.

If the case is **STATE_SUSPENDED**, then change the state of the task to **STATE_RUNNING** and set an event by calling the API **SetEvent**() to indicate that the task is ready to be scheduled for execution.

If the case is **STATE_INTERRUPTED**, change the tasks state to running and continue its execution from where it was interrupted by calling the **ResumeThread** and passing it the **ThreadHandle** of the task.

Return once done.

If the current task Terminate is set to **DEF_TRUE**, it indicates that the task needs to be terminated. It would be set true when **OSTaskDelHook**() is called.

Once the Context Switch is performed, exit the critical section by calling **OS_EXIT_CRITICAL**() that would also enable the interrupts.

Visualizing Saving and resuming during context switching:

*Figure 3.7 Saving the current task's context.*

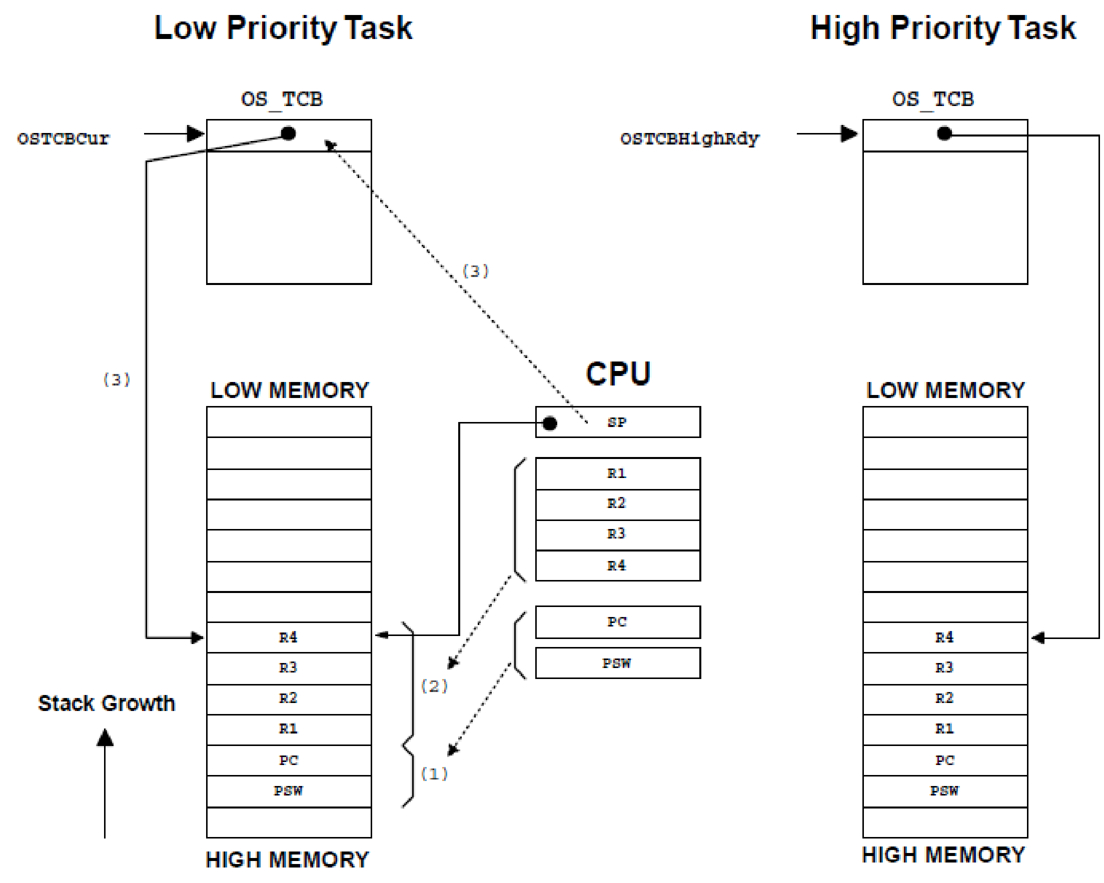
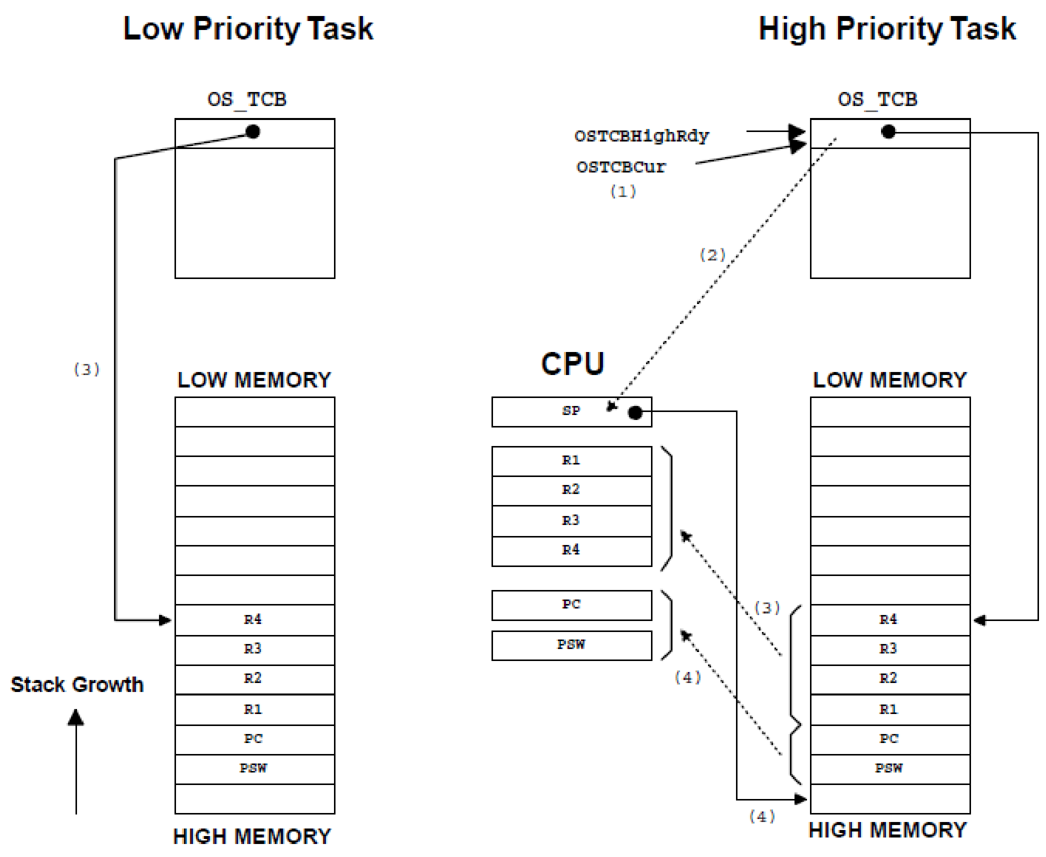


*Figure 3.8 Resuming the current task.*



Img Src: MicroC/OS-II The Real-Time Kernel, Second Edition, Jean J. Labrosse.

For the final overview of scheduling in MicroC/OS-II, **OS_Sched**() acts as an entry point for scheduling, **OS_SchedNew**() determines the highest priority task and the context switch is performed by **OS_TASK_SW**() while **OSTaskSwHook**() is used for custom code during a context switch.

**References**: MicroC/OS-II The Real-Time Kernel, Second Edition, Jean J. Labrosse.

**Article Written By**: Yashwanth Naidu Tikkisetty

Follow **#oswithyash** to get updates related to Embedded Systems, Space and Technology.