Preemptive Multitasking

RTX Kernel is a **preemptive** multitasking operating system. If a task with a higher priority than currently running task becomes ready to run, it will **suspend** the current running task.

A preemptive task switch occurs when:

- the task scheduler is executed from a system **tick timer** interrupt function. Task scheduler process the **delays** of tasks. If a delay for a task with higher priority has expired, this task will continue to execute instead of currently running task.
- an **event** is set for a higher priority task by a currently running task or by an interrupt service routine. The currently running task will be suspended and the higher priority task continues to run.
- a token is returned to a **semaphore** and a higher priority task is waiting for one. The currently running task will be suspended and a semaphore waiting task will continue to run. The token may be returned by a currently running task or by an interrupt service routine.
- a **mutex** is released and a higher priority task is waiting for it. The currently running task will be suspended and a task waiting for mutex will continue to run.
- a message is posted to a **mailbox** and a higher priority task is waiting for one. The currently running task will be suspended and a message waiting task will continue to run. The message may be posted by a currently running task or by an interrupt service routine.
- a **mailbox** is **full**, and a higher priority task is waiting to post a message to a mailbox. As soon as currently running task or an interrupt service routine has popped out a message from a mailbox, a task waiting to post a message will continue to run.
- a **priority** of currently running task has reduced. If other task is ready to run and has a higher priority than currently running task, this task is suspended immediately and higher priority task resumes it's execution.

Take a look at the following example. Task **job1** has a higher priority than task **job2**. When **job1** is started, it creates task **job2** and enters **os_evt_wait_or()** function. Here it is suspended and execution continues with task **job2**. As soon as **job2** has set an event flag for **job1**, it is suspended and task **job1** is resumed. Task **job1** increments counter cnt1 and suspends again calling os_evt_wait_or() function. Task **job2** is resumed, increments counter cnt2 and sets an event flag for **job1**. This process is repeated indefinitely.

```
#include <RTL.h>

OS_TID tsk1,tsk2;
int    cnt1,cnt2;

void job1 (void) __task;
void job2 (void) __task;

void job1 (void) __task  {
   os_tsk_prio (2);
   os_tsk_create (job2, 1);
   while (1) {
      os_evt_wait_or (0x0001, 0xffff);
      cnt1++;
   }
}

void job2 (void) __task  {
   while (1) {
      os_evt_set (0x0001, job1);
      cnt2++;
   }
}

void main (void) {
   os_sys_init (job1);
   while (1);
}
```