Introduction to Real-Time Kernels

Time & Resource Management

2013-07-18

Jean J. Labrosse

CEO, **Micriμm**

Outline

- **The Tick ISR**
  - Time Delays
  - Timeouts
- **Soft Timers**
- **Resource sharing and Mutual Exclusion**
  - Priority Inversions
  - Priority Inheritance

The Tick ISR

- **Most kernels require a periodic interrupt source**
  - Through a hardware timer
    - Interrupt rate between 10 and 1,000 Hz
  - Could be from the power line
    - 50 or 60 Hz
  - The higher the tick rate, the higher the overhead
- **A Clock Tick is NOT mandatory**

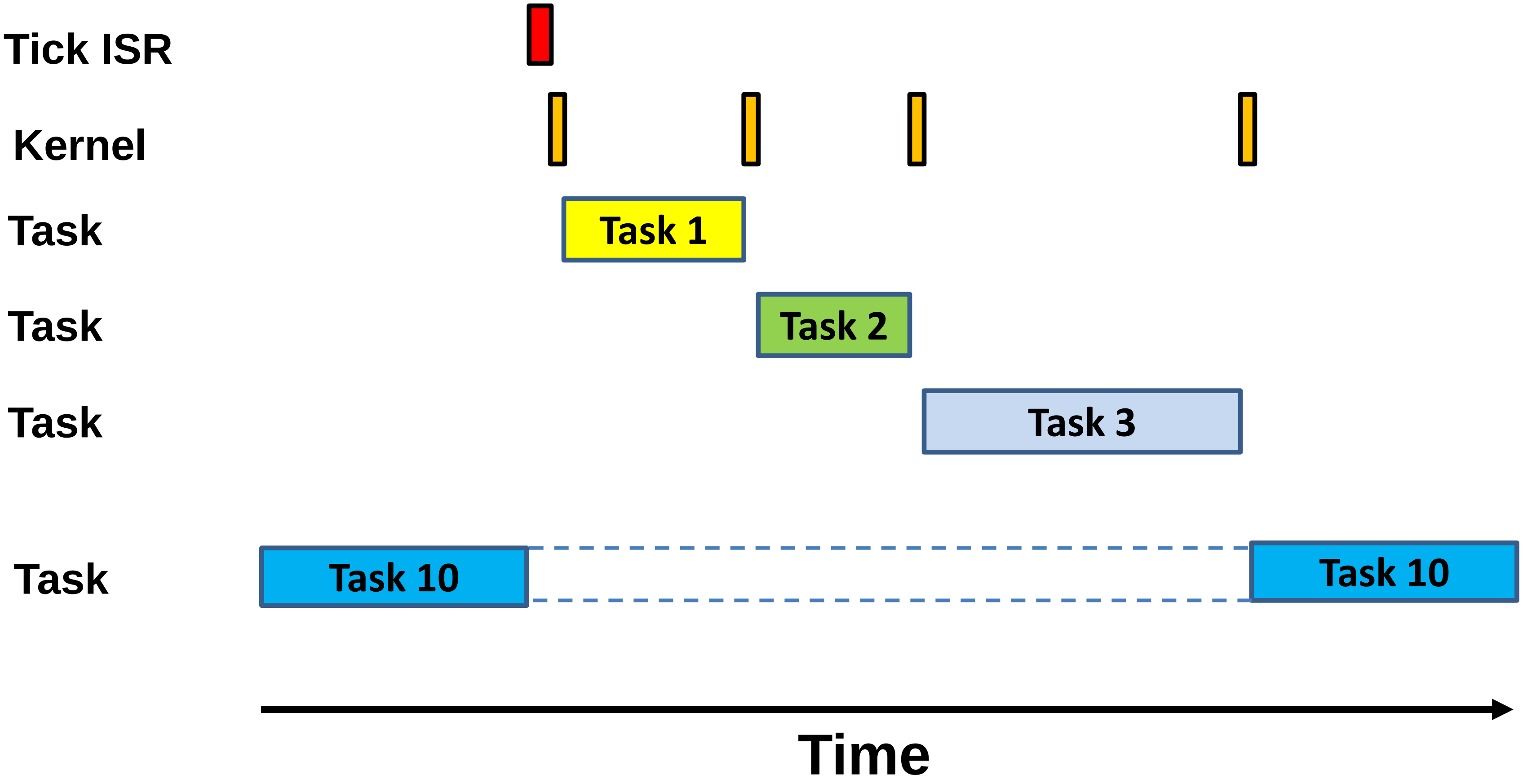
Why do kernels have a Tick?

- **To allow tasks to suspend execution based on time**
  - For example, scanning a keyboard

```
void  MyTask (void)
{
      while (1) {
            OSTimeDly(50);
            Scan keyboard;
      }
}
```

- **To provide timeouts while waiting for events**
  - Avoids waiting forever for events to occur
  - Eliminates deadlocks

Tick Wait List



Soft Timers

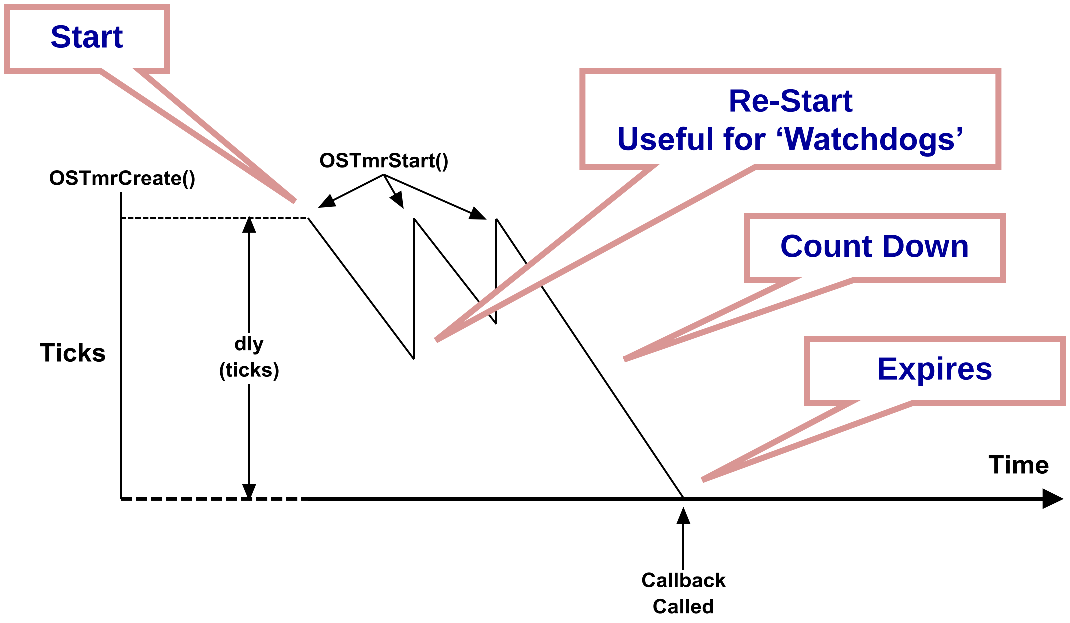
- **Most kernels provide ‘soft timers’**
  - Soft Timers are derived from a single interrupt source
  - ‘Callback’ function is called when timer expires

- **Useful for ‘watchdog’ type applications**

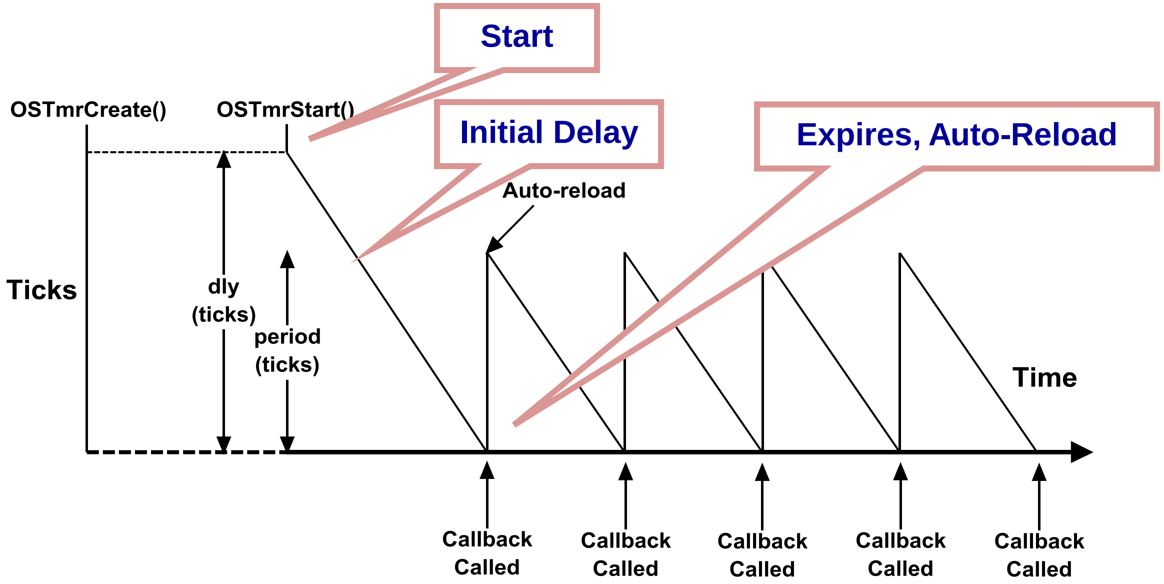
- **Kernel level task manages any number of timers**

- **Timers can be one-shot or periodic**
  - Can be started, re-started or stopped

One-Shot Timers



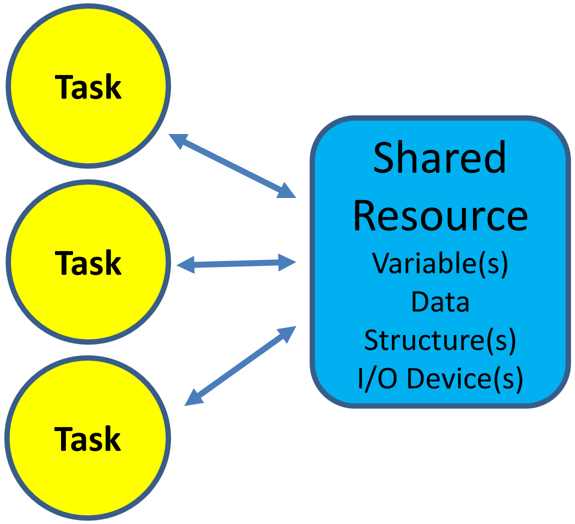
Periodic Timers



Resource Sharing

- **YOU MUST ensure that access to common resources is protected!**
  - A kernel only gives you mechanisms

- **You protect access to common resources by:**
  - Disabling/Enabling interrupts
  - Lock/Unlock
  - Semaphores
  - MUTEX (Mutual Exclusion Semaphores)



Resource Sharing
(Disabling and Enabling Interrupts)

- **When access to resource is done quickly**
  - Example:

```
rpm = 60.0 / time;
Disable interrupts;
Global RPM = rpm;
Enable interrupts;
```

- **Disable/Enable interrupts is the fastest way!**
  - Be careful with Floating-point!

Resource Sharing
(Locking and Unlocking the Scheduler)

- **‘Locking’ the scheduler prevents the scheduler from changing tasks**
  - Interrupts are still enabled
  - Can be used to access non-reentrant functions
  - Can be used to reduce priority inversion
  - Same effect as making the current task the Highest Priority Task
  - Defeats the purpose of having a kernel.
  - Pseudo code:

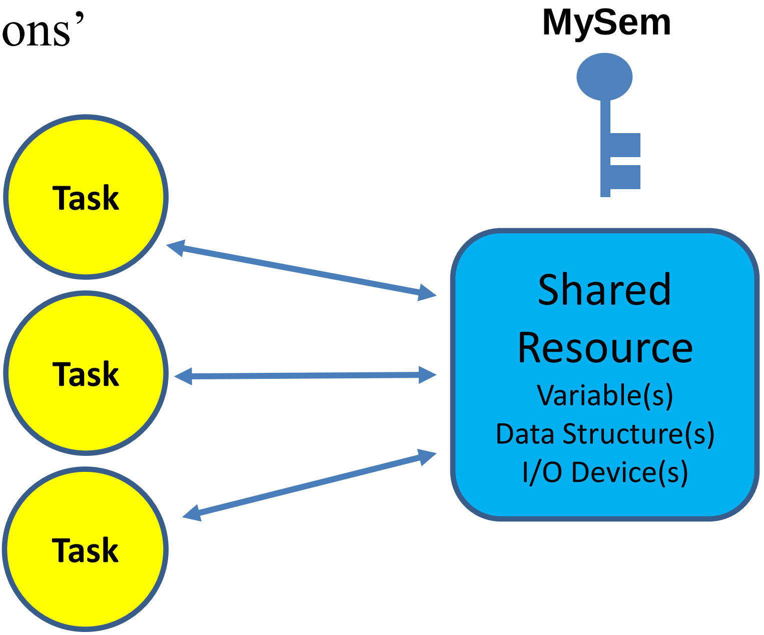
```
OS_SchedLock();
Code with scheduler disabled;
OS_SchedUnlock;
```

- **‘Unlocking’ invokes the scheduler to see if a High-Priority Task has been made ready while locked**

Resource Sharing
(Semaphores)

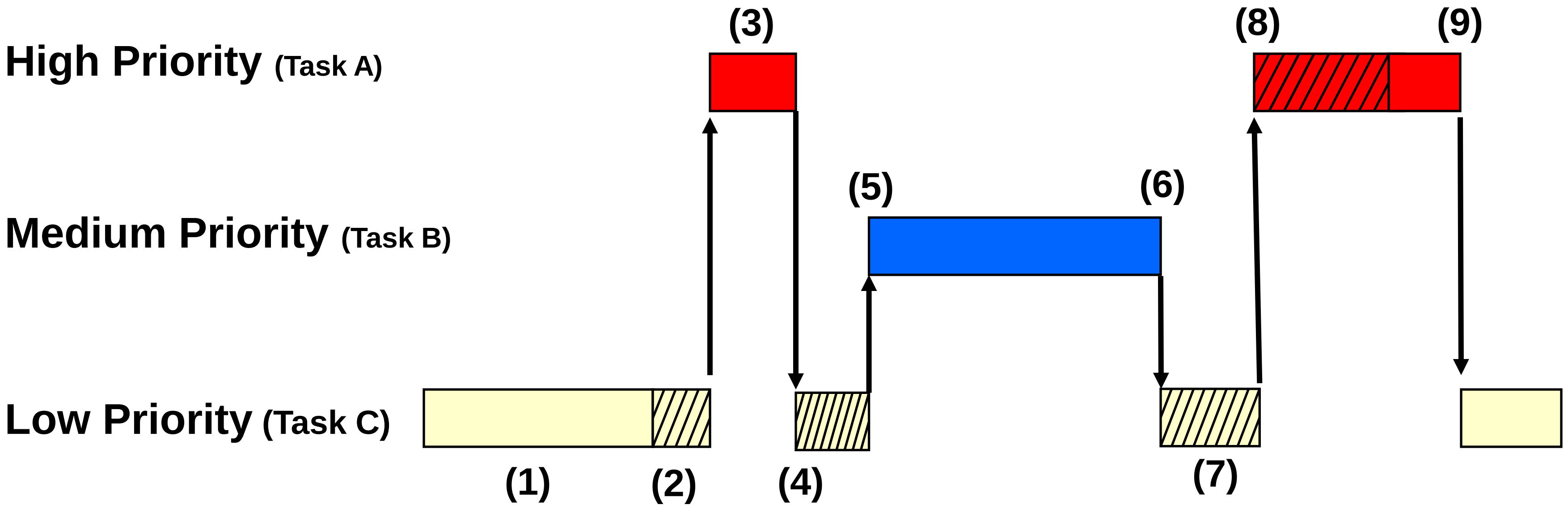
- **A semaphore is a kernel 'object'**
  - Your application needs to obtain the semaphore before it can proceed to access the resource
  - If the resource is used by another task, the caller is blocked
  - Semaphores are subject to 'priority inversions'

```
SemWait(&MySem);
Code can access resource;
SemRelease(&MySem);
```



Resource Sharing

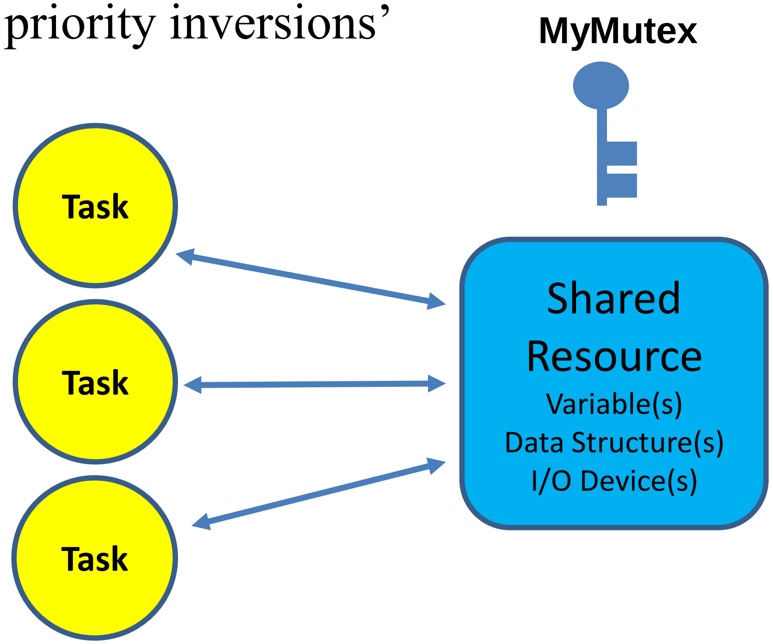
(Semaphores – Priority Inversions)



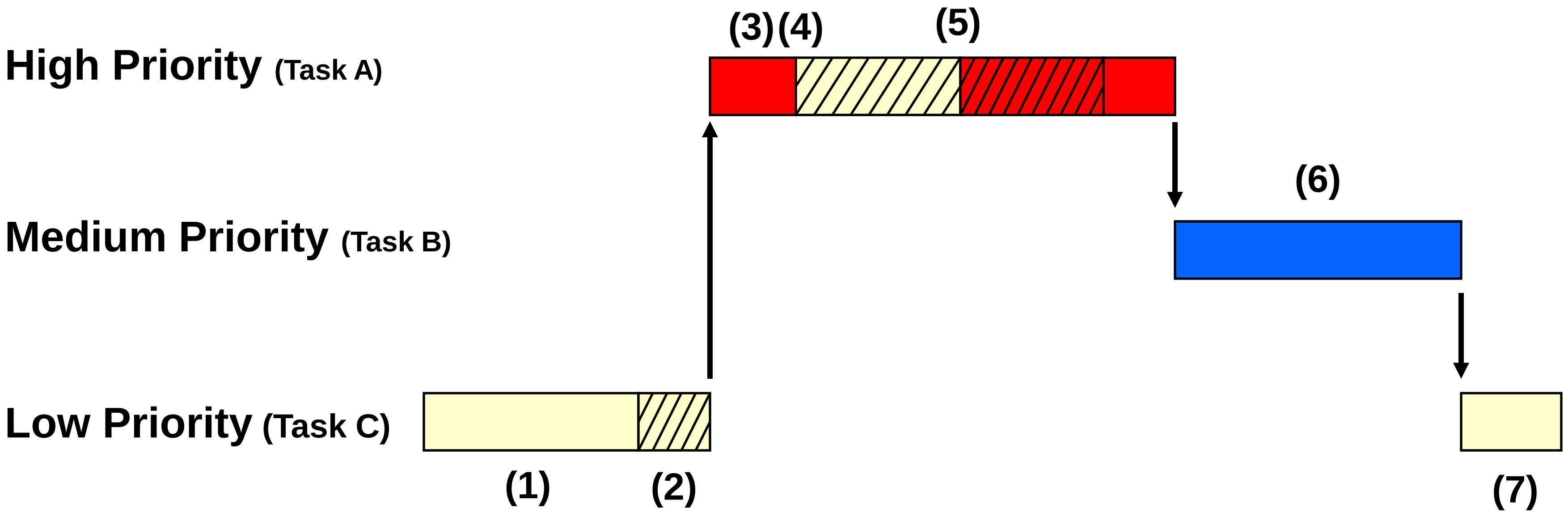
Resource Sharing
(Mutual Exclusion Semaphores - Mutex)

- **A Mutex is a kernel ‘object’**
  - Your application needs to obtain the mutex before it can proceed to access the resource
  - If the resource is used by another task, the caller is blocked
  - Mutexes protect your application against ‘priority inversions’

```
MutexWait(&MyMutex);
Code can access resource;
MutexRelease(&MyMutex);
```



Resource Sharing
(Mutual Exclusion Semaphores - Mutex)



Next Class

- **Signaling a Task**
  - Semaphores
  - Event Flags
- **Inter-task Communications**
- **Debugging kernel-based applications**
  - Debuggers
  - Kernel Aware Debuggers
  - Output Port
  - DAC output
  - Run-Time Kernel Awareness
  - Trace Tool
- **Summary**