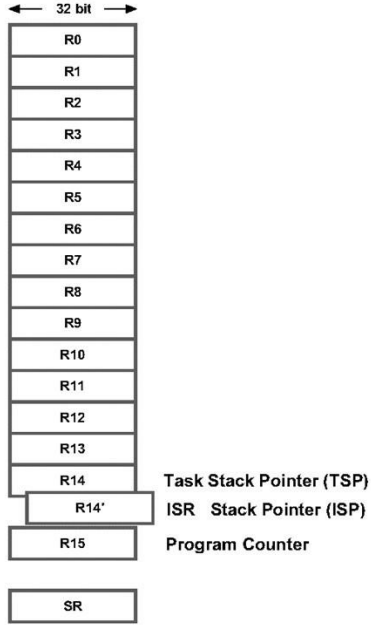
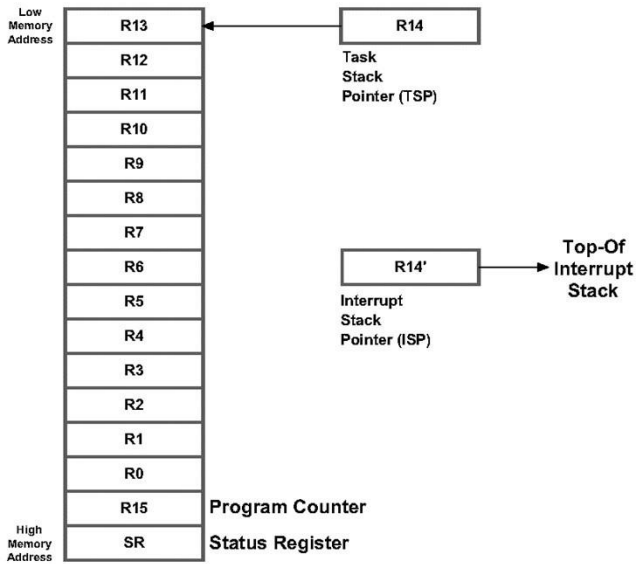
Context Switching

- *Task Context:* typically consists of the CPU registers utilized by the task while executing.
- *Context Switch:* when the OS saves the current task's context into the current task's stack and restores the context of the new task and resumes execution of that task.
- Context switching adds overhead
- The context switch code is part of a processor's *port*
  - A port is the code needed to adapt µC/OS-III to the desired processor



Stack Frame for a Ready Task

- The stack frame for a ready task is pre-initialized by OS in a similar way as an interrupt just occurred
- Task stack pointer …
- Interrupt stack pointer …

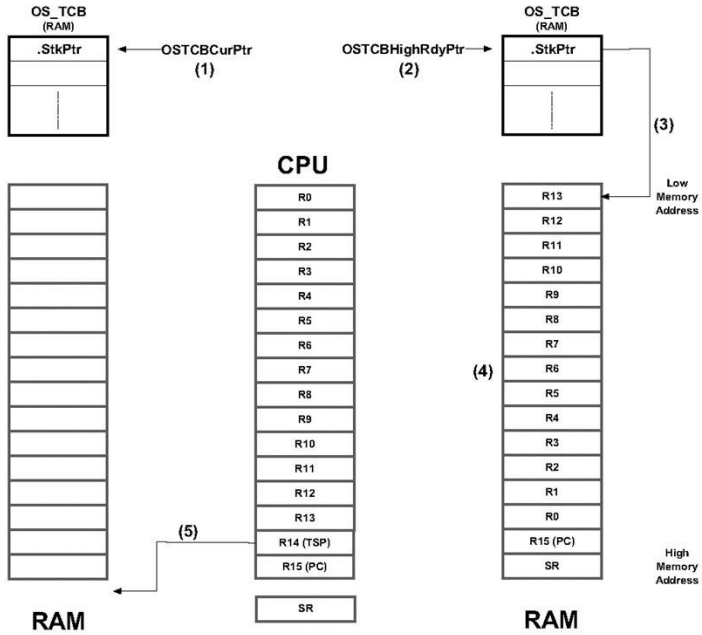


Types of Context Switch

1. Task level context switch
   - This type of context switch is performed by the kernel by calling OSCtxSw() which is invoked by the macro OS_Task_SW()
   - OSCtxSw() function is called when a new high priority task needs to execute
2. ISR context switch
   - This type of context switch is implemented by the kernel by OSIntCtxSw() function.

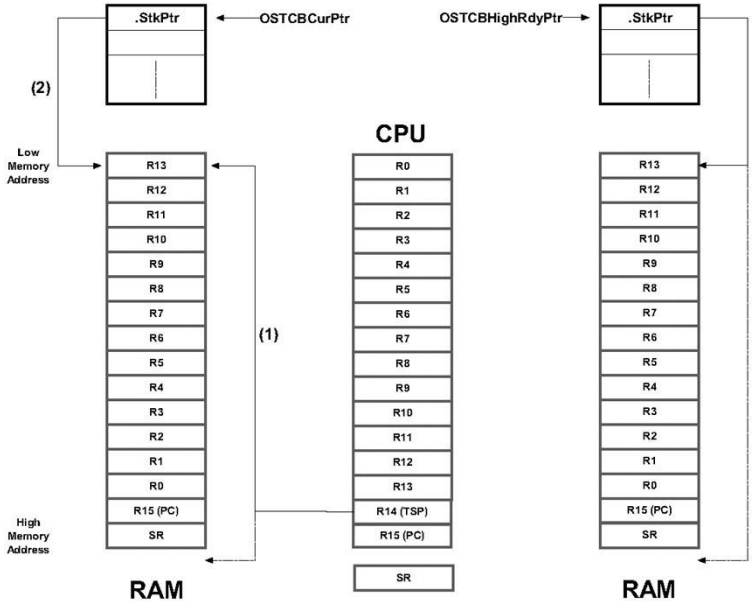
Task Level Context Switch Explained

- The task level scheduler OSSched() calls OSCtxSw()
- Prior to calling OSCtxSw() function the variable and data structures involved look as in the diagram
- (1) …
- (2) …
- (3) …
- (4) …
- (5) …



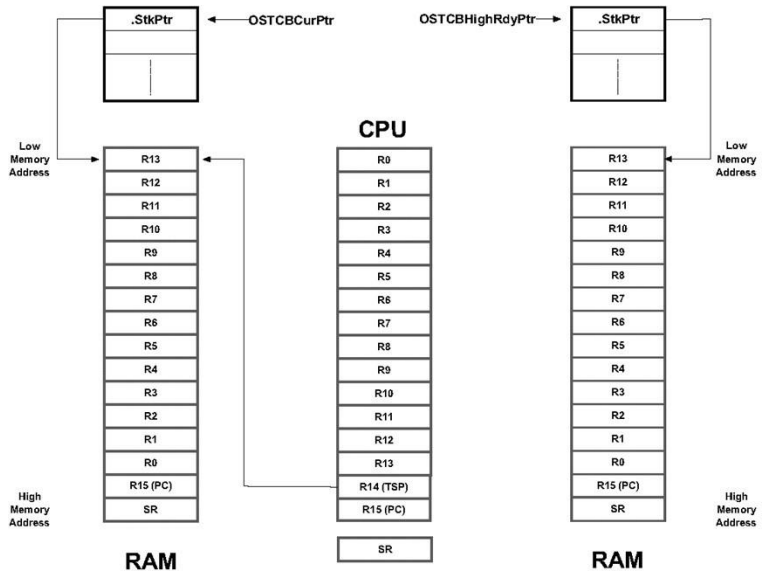
Executing the Task Level Context Switch

- (1) OSCtxSw() begins by saving the SR and PC of the current task onto the current task's stack
- (2) …
- (3) …
- (4) …



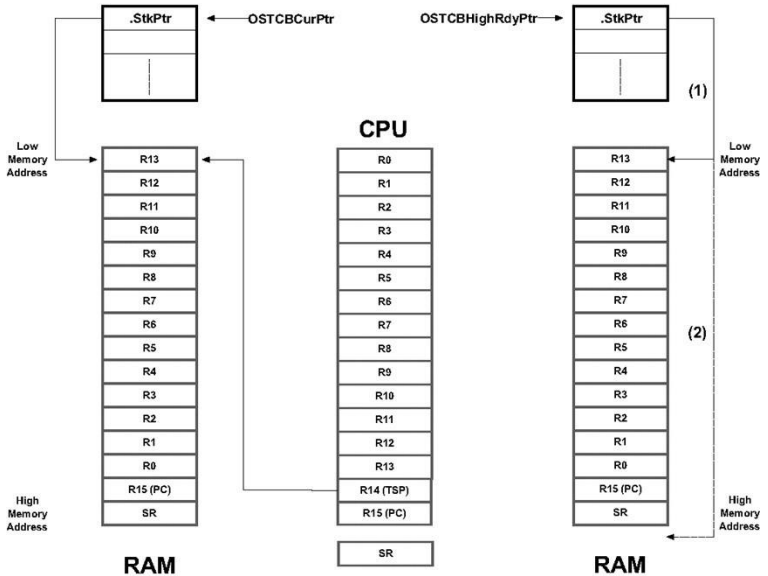
ISR Level Context Switch Explained

- OSIntCtxSw() (see os_cpu_a.asm) is called when the ISR level scheduler (OSIntExit()) determines that a new high priority task is ready to execute
- Prior to calling OSIntCtxSw() the data structures is like in the diagram
- At interrupt, the CPU registers of the current task are automatically saved onto the task stack



Operations Performed by OSIntCtxSw()

- The second half o the context switch is done by OSIntCtxSw()
- (1) …
- (2) …



Interrupt Management in uC/OS-III

- An *interrupt* is a hardware mechanism used to inform the CPU that an asynchronous event occurred.
- When interrupts are recognized by CPU, some specific hardware operations take place internally to CPU
  - The CPU saves part (or all) of its context and jumps to a special routine called ISR
- The ISR processes the event, and – upon completion of the ISR – the program either returns to the interrupted task, or the highest priority task, if the ISR made a higher priority task ready-to-run.
- Microprocessors allow interrupts to be masked, disabled or enabled, etc.
- However, in real-time environments, interrupts should be disabled as little as possible

Important Time Specifications of a Real-Time Kernel

- Interrupt disable time
  - Is the maximum amount of time that interrupts are disabled by the kernel – all real-time systems disable interrupts to manipulate critical sections of code
- Interrupt response
  - Is defined as the time between the receiving an interrupt signal and the start of the user code that handles the interrupt
- Interrupt recovery
  - Is defined as the time required for the processor to return to the interrupted code or to a higher priority task
- Task latency
  - Is defined as the time it takes from the time interrupt occurs to the time task level code resumes

Handling CPU Interrupts

- In most cases, an *interrupt controller* captures all of the different interrupts presented to the processor
- CPUs interrupts models:
  - All interrupts *vector* to a single interrupt handler.
  - Each interrupt *vectors* directly to an interrupt handler

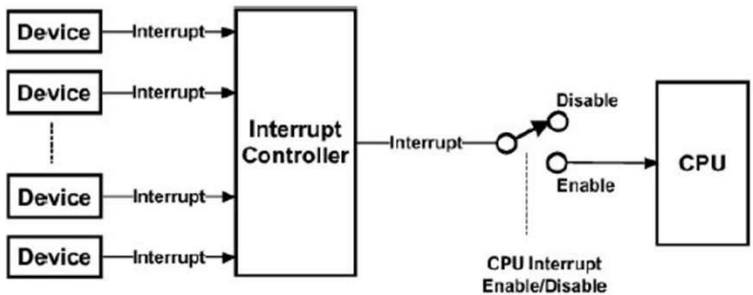


Figure - Interrupt controllers

Typical uC/OS-III Interrupt Service Routine

Case 1: ISR signals or sends a message to a task

- (1) to (6) … prologue …
- (8) … further processing …
- (9) … must call this OS function …
- (9) to (11) … epilogue …

```
MyISR:                                                          (1)
    Disable all interrupts;                                     (2)
    Save the CPU registers;                                     (3)
    OSIntNestingCtr++;                                          (4)
    if (OSIntNestingCtr == 1) {                                 (5)
        OSTCBCurPtr->StkPtr = Current task's CPU stack pointer register value;
    }
    Clear interrupting device;                                  (6)
    Re-enable interrupts (optional);                            (7)
    Call user ISR;                                              (8)
    OSIntExit();                                                (9)
    Restore the CPU registers;                                  (10)
    Return from interrupt;                                      (11)
```

(8) … as a general rule, keep the ISRs short. The ISR should signal or send a message to a task and that task will perform the actual interrupt service.

The ISR will call one of the following functions:
OSSemPost(), OSTaskSemPost(), OSFlagPost(), OSQPost() or OSTaskQPost().

Fast Interrupt Service Routine

- Case 2: the ISR does not signal a task but performs all the work quickly in the ISR, then consider writing the ISR as a "Non Kernel-Aware Interrupt Service Routine,"

```
MyShortISR:                                                    (1)
    Save enough registers as needed by the ISR;                (2)
    Clear interrupting device;                                 (3)
    DO NOT re-enable interrupts;                               (4)
    Call user ISR;                                             (5)
    Restore the saved CPU registers;                           (6)
    Return from interrupt;                                     (7)
```

(4) Do not re-enable interrupts at this point since another interrupt could make µC/OS-III calls, forcing a context switch to a higher-priority task

(5) Now you can take care of the interrupting device in assembly language or call a C function, if necessary.

Processor with Multiple Interrupt Priorities

- There are some processors that actually supports multiple interrupt levels as shown

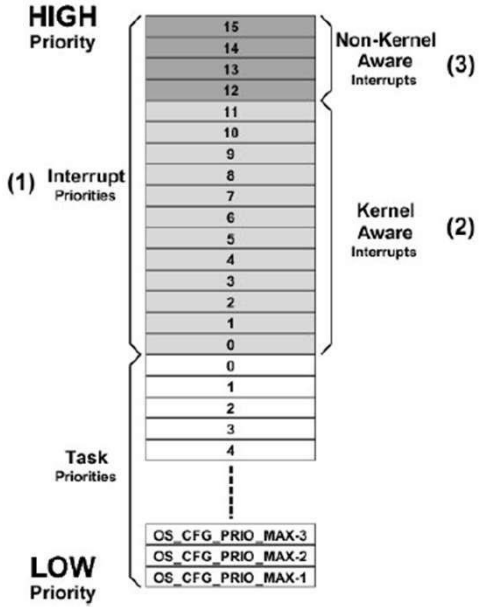


Figure - Kernel Aware and Non-Kernel Aware Interrupts

Listing 9-3 shows how to implement non-kernel aware ISRs when the processor supports multiple interrupt priorities.

```
MyNonKernelAwareISR:                                        (1)
            Save enough registers as needed by the ISR;     (2)
            Clear interrupting device;                      (3)
            Enable higher priority interrupts (optional);   (4)
            Call user ISR;                                  (5)
            Restore the saved CPU registers;                (6)
            Return from interrupt;                          (7)
```

Listing - Non-Kernel Aware ISRs for Processors with Multiple Priority Levels

It's important to note that since µC/OS-III cannot disable the non-kernel aware interrupts, interrupt latency for these interrupts is very short.

All Interrupts Vector to a Common Location

Even though an interrupt controller is present in most designs, some CPUs still vector to a common interrupt handler, and the ISR queries the interrupt controller to determine the source of the interrupt.

```
An interrupt occurs;                                    (1)
 The CPU vectors to a common location;                  (2)
 The ISR code performs the "ISR prologue"               (3)
 The C handler performs the following:                  (4)
     while (there are still interrupts to process) {    (5)
         Get vector address from interrupt controller;
         Call interrupt handler;
     }
 The "ISR epilogue" is executed;                        (6)
```

Listing - Single interrupt vector for all interrupts

Every Interrupt Vectors to a Unique Location

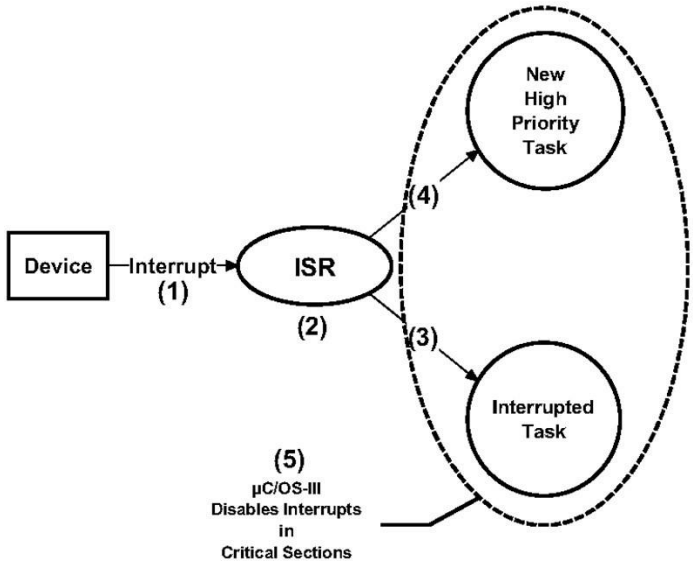
- If the interrupt controller vectors directly to the appropriate interrupt handler, each of the ISRs that are actually used in the application should be written in assembly language and follow the typical uc/OS-III Interrupt Service Routine guidelines (See MyISR structure).

uC/OS-III Event Posting from ISR

- uC/OS-III handles event posting from interrupts in two ways:
  - Direct post method
  - Deferred post method
- From an application and ISR code point of view these methods are completely transparent
- The programmer needs only change the configuration value OS_CFG_ISR_POST_DEFERRED_EN to switch between the two methods

Direct and Deferred Post Methods for Event Posting

- (1) …; (2) …; (3) …;(4) …
- (5) In the Direct Post Method, µC/OS-III must protect critical sections by disabling interrupts as some of these critical sections can be accessed by ISRs
- In the differed post method, instead of disabling interrupts to access critical sections, uC/OS-III locks the scheduler …



Characterizing the Interrupt Times for the uC/OS-III Services

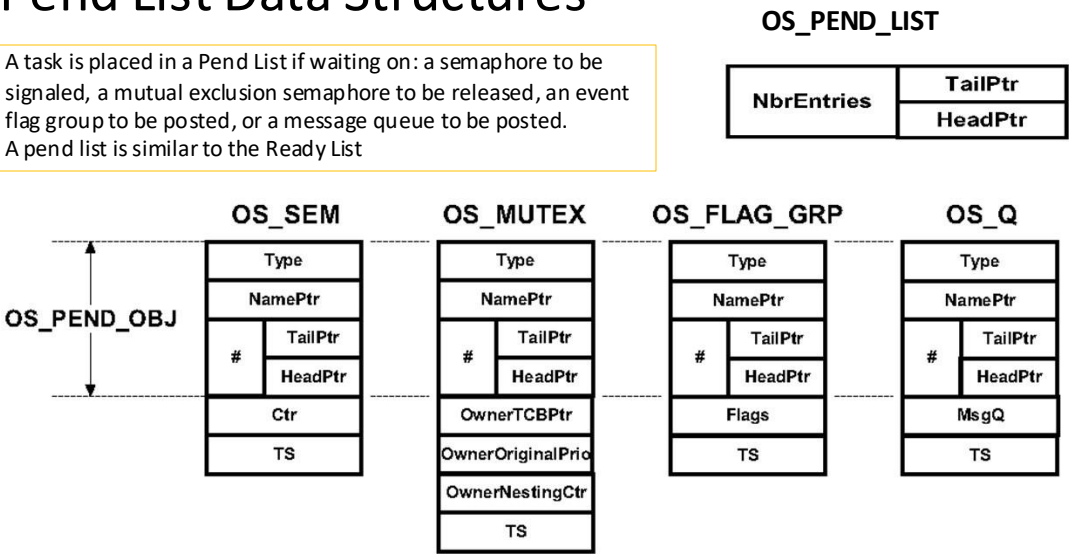
- µC/OS-III disables interrupts while accessing critical sections
- You can determine the interrupt latency, interrupt response, interrupt recovery, and task latency by adding the execution times of the code involved for each:
  - Interrupt Latency = Maximum interrupt disable time;
  - Interrupt Response = Interrupt latency + Vectoring to the interrupt handler + ISR prologue;
  - Interrupt Recovery = Handling of the interrupting device + Posting a signal or a message to a task + OSIntExit() + OSIntCtxSw();
  - Task Latency = Interrupt response + Interrupt recovery + Time scheduler is locked;
- The post times calls can be measured by utilizing OS_TS_GET()

Pend List Data Structure for uC/OS-III

- A pend list is similar to the Ready List, but it keeps track of tasks waiting for an object to be posted.
  - In addition, the pend list is sorted by priority; the highest priority task waiting on the object is placed at the head of the list, …
- A pend list is a data structure of type OS_PEND_LIST, which consists of three fields
- Each kernel object using a pend list contains the same three fields at the beginning of the kernel object that we called an OS_PEND_OBJ.
  - Notice that the first field is always a “Type” …

Pend List Data Structures

A task is placed in a Pend List if waiting on: a semaphore to be signaled, a mutual exclusion semaphore to be released, an event flag group to be posted, or a message queue to be posted.
A pend list is similar to the Ready List



There are particular OS internal functions that manipulate entries in a pend list

Function	Description
`OS_PendListChangePrio()`	Change the priority of a task in a pend list
`OS_PendListInit()`	Initialize a pend list
`OS_PendListInsertPrio()`	Insert a task in priority order in the pend list
`OS_PendListRemove()`	Remove a task from a pend list

Table - Pend List access functions

Time Management

- RTOS kernels generally require the provision of a periodic interrupt to keep track of time delays and timeouts.
  - This periodic time source is called a clock tick and for uC/OS-III should occur between 10 and 1000 times per second, or Hertz (see OS_CFG_TICK_RATE_HZ in os_cfg_app.h)
- The actual frequency of the clock tick depends on the desired tick resolution of the application.
  - Note: a high frequency of the ticker will give higher overhead.

uC/OS-III Time Management Services

Function Name	Operation
OSTimeDly()	Delay execution of a task for “n” ticks
OSTimeDlyHMSM()	Delay a task for a user specified time in HH:MM:SS.mmm
OSTimeDlyResume()	Resume a delayed task
OSTimeGet()	Obtain the current value of the tick counter
OSTimeSet()	Set the tick counter to a new value
OSTimeTick()	Signal the occurrence of a clock tick

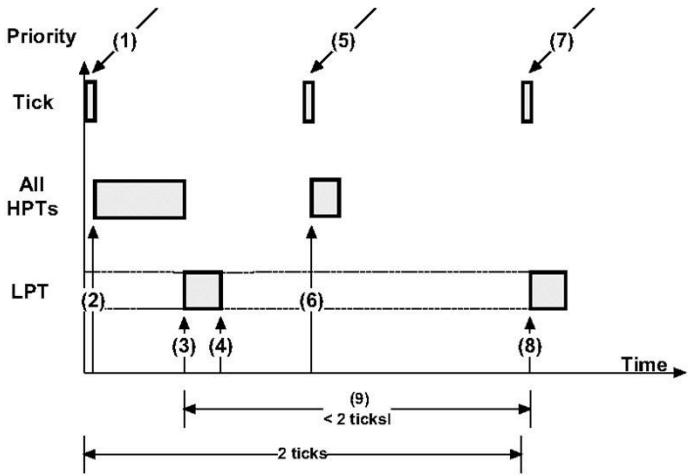
Using OSTimeDly()

- This function allows 3 modes of time delay: relative, periodic, and absolute
- OSTimeDly() -- relative usage
  - (1) …
  - (2) …
  - (3) …
  - (4) …
- Problem: the delay is not accurate

```
void  MyTask (void *p_arg)
{
    OS_ERR  err;
    :
    :
    while (DEF_ON) {
        :
        :
        OSTimeDly(2,                          (1)
                  OS_OPT_TIME_DLY,            (2)
                  &err);                      (3)
        /* Check "err" */                     (4)
        :
        :
    }
}
```

OSTimeDly() Relative, Time Delay Analysis

- Assume that low priority task (LPT) calls to delay for 2 ticks
- (1): tick interrupt comes
- (2): The execution time of HPTs is unknown and can vary
- (3): LPT executes
- (4): LPT requests time delay
- (9): Due to execution time of the HPTs, the time delay is not exactly two ticks, as requested.



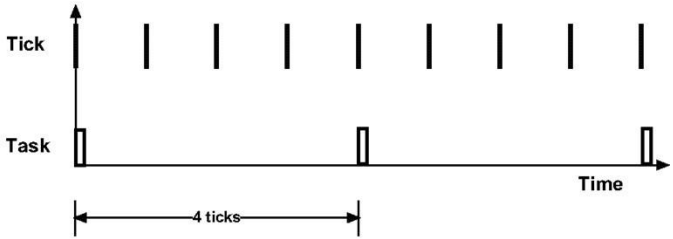
OSTimeDly() with Periodic Usage

(1) … OSTimeDly() …

(2) … periodic option …

This option allows delaying the task until the tick counter reaches a certain periodic match value

- This, ensures that the spacing in time is always the same as it is not subject to CPU load variations




```
void  MyTask (void *p_arg)
{
    OS_ERR   err;
    :
    :
    while (DEF_ON) {
        OSTimeDly(4,                          (1)
                  OS_OPT_TIME_PERIODIC,       (2)
                  &err);
        /* Check "err" */                     (3)
        :
        :
    }
}
```

OSTimeDly() with Absolute Usage

- Absolute usage: the "dly" parameter corresponds to the desired value of OSTickCtr you want to reach.
- OSTickCtr value in this case is normally counted from the power up time

- Summary of the "opt" values based on OSTickCtr values

Value of "opt"	Task wakes up when
OS_OPT_TIME_DLY	OSTickCtr + dly
OS_OPT_TIME_PERIODIC	OSTCBCurPtr->TickCtrPrev + dly
OS_OPT_TIME_MATCH	dly

Homework: Time Delay Functions

1. Review the following time delay functions:
   a. OSTimeDlyHMSM()
   b. OSTimeDlyResume()
   c. OSTimeSet()
   d. OSTimeGet()
   e. OSTimeTick()
2. Present a scenario of LPT, MPT, and HPT tasks using at least 2 tasks where the delayed tick value of the task varies with the system load.

Timer Management

- Timer services are found in os_tmr.c file
- Timer services, not time delay services, are enabled when setting OS_CFG_TMR_EN to 1 in os_cfg.h
- The resolution of the timer is set by the configuration constant: OS_CFG_TMR_TASK_RATE_HZ, which is expressed in Hertz (Hz).
- Timers are down counters that perform an action when the counter reaches to zero
  - The action is performed by a callback function which is a user-declared function.
- The callback can be used to turn a light on/off, start a motor, etc.
  - But we should not make blocking calls within a callback function (i.e., call OSTimeDly(), OSTimeDlyHMSM(), OS???Pend(), or anything that causes the timer task to block or be deleted).

Timer Services

Function Name	Operation
`OSTmrCreate()`	Create and specify the operating mode of the timer.
`OSTmrDel()`	Delete a timer.
`OSTmrRemainGet()`	Obtain the remaining time left before the timer expires.
`OSTmrStart()`	Start (or restart) a timer.
`OSTmrStateGet()`	Obtain the current state of a timer.
`OSTmrStop()`	Stop the countdown process of a timer.

Working with Timers

- A timer needs to be created; it can be started (or restarted) and stopped as often as is necessary.
- Timers operate in one of three modes: One-shot, Periodic (no initial delay), and Periodic (with initial delay).

```
void  OSTmrCreate (OS_TMR              *p_tmr,          /* Pointer to timer     */
                   CPU_CHAR            *p_name,         /* Name of timer, ASCII */
                   OS_TICK              dly,            /* Initial delay        */
                   OS_TICK              period,         /* Repeat period        */
                   OS_OPT               opt,            /* Options              */
                   OS_TMR_CALLBACK_PTR  p_callback,     /* Fnct to call at 0    */
                   void                *p_callback_arg, /* Arg. to callback     */
                   OS_ERR              *p_err)
```

One-Shot Timers

- A one-shot timer will countdown from its initial value, call the callback function when it reaches zero, and stop
- Once completed, the timer needs to be restarted by calling OSTmrStart(), to be activated

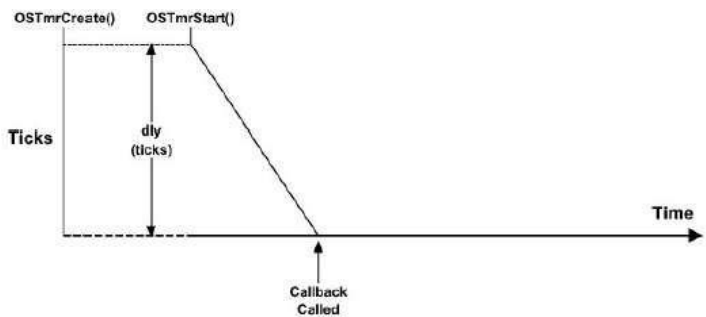
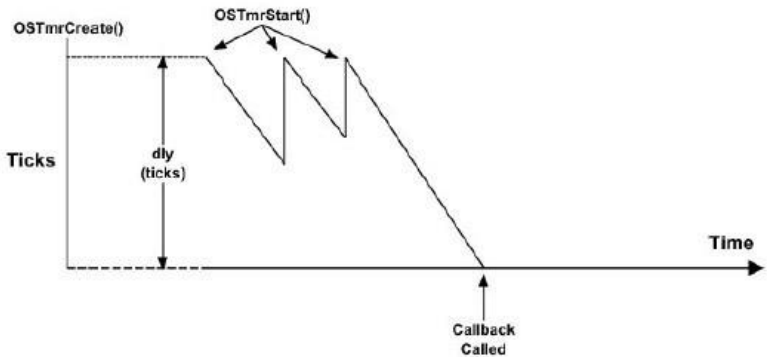


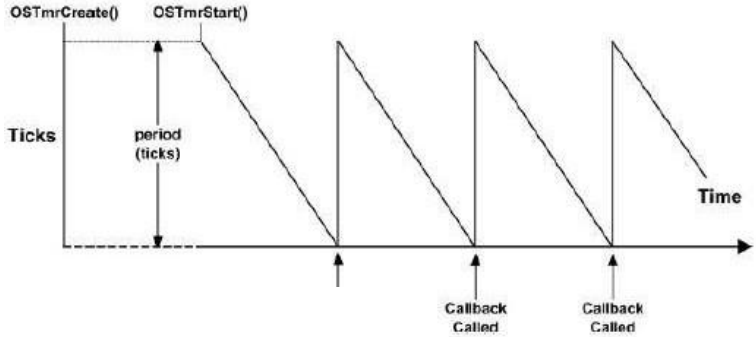
Figure - One Shot Timers (dly > 0, period == 0)

Application of One-Time Shot Timer

- A one-shot timer can be retriggered by calling OSTmrStart() before the timer reaches zero.
- This feature can be used to implement watchdogs and similar safeguards.



Periodic With and Without Initial Delay



Timers (dly == 0, period > 0)

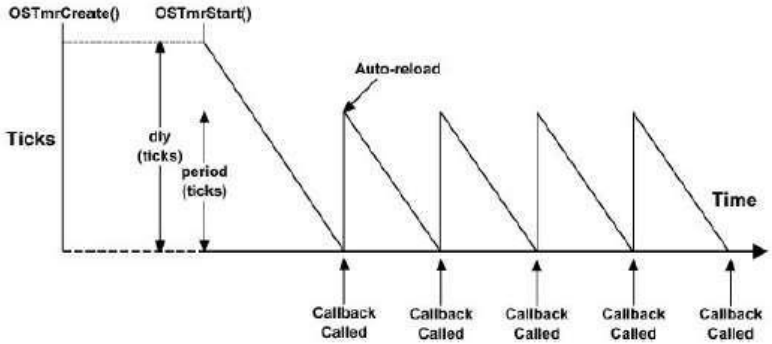


Figure - Periodic Timers (dly > 0, period > 0)

Timer Internals and Timer Task

- See reference manual

Homework: Timers

1. Study the functionality of one-shot timers and periodic timers as described in the user manual
2. Write an application that uses a timer to read an ADC from a sensor and store the data in a queue. You should use more than one task, so your application is a real-time application. For example, you could use the sensor data in a controller task or a monitoring task, etc.
3. Think of an application where you could use the timer as a safeguard

Resource Management in uC/OS-III

- uC/OS-III provides services to manage shared resources
- A shared resource is typically a variable (static or global), a data structure, table (in RAM), or registers in an I/O device
- When protecting a shared resource within a uC/OSIII application is preferred to use mutual exclusion semaphores
- It is important to ensure that each task has exclusive access to the shared data to avoid contention and data corruption

- See Example: Time of the day in the uC/OS-III documentation (pp. 201)

Example of Shared Resource Requirement

Scenario:
1) TimeofDay() task was preempted by a HPT, after setting the Minutes = 0, due to some interrupt occurring …
2) The HPT wants to know the current time from the time-of-day module.
3) Since the Hours were not incremented prior to the interrupt, the HPT will read the time incorrectly.

Solution:
The code that updates variables for TimeOfDay() task must treat all variables indivisibly (or atomically) when preemtion is possible

```
void  TimeOfDay (void *p_arg)
{
    OS_ERR  err;


    (void)&p_arg;
    while (DEF_ON) {
        OSTimeDlyHMSM(0,
                      0,
                      1,
                      0,
                      OS_OPT_TIME_HMSM_STRICT,
                      &err);
        /* Examine "err" to make sure the call was successful */
        Seconds++;
        if (Seconds > 59) {
            Seconds = 0;
            Minutes++;
            if (Minutes > 59) {
                Minutes = 0;
                Hours++;
                if (Hours > 23) {
                    Hours = 0;
                }
            }
        }
    }
}
```

Common Methods to Protect Shared Resources

- The most common methods of obtaining exclusive access to shared resources and to create critical sections are:
  - disabling interrupts
  - disabling the scheduler
  - using semaphores
  - using mutual exclusion semaphores (a.k.a. a mutex)

Resource Sharing Methods

Resource Sharing Method	When should you use?
Disable/Enable Interrupts	When access to shared resource is very quick (reading from or writing to few variables) and access is faster than µC/OS-III's interrupt disable time. It is highly recommended to not use this method as it impacts interrupt latency.
Locking/Unlocking the Scheduler	When access time to the shared resource is longer than µC/OS-III's interrupt disable time, but shorter than µC/OS-III's scheduler lock time. Locking the scheduler has the same effect as making the task that locks the scheduler the highest-priority task. It is recommended not to use this method since it defeats the purpose of using µC/OS-III. However, it is a better method than disabling interrupts, as it does not impact interrupt latency.
Semaphores	When all tasks that need to access a shared resource do not have deadlines. This is because semaphores may cause unbounded priority inversions (described later). However, semaphore services are slightly faster (in execution time) than mutual-exclusion semaphores.
Mutual Exclusion Semaphores	This is the preferred method for accessing shared resources, especially if the tasks that need to access a shared resource have deadlines. µC/OS-III's mutual exclusion semaphores have a built-in priority inheritance mechanism, which avoids unbounded priority inversions. However, mutual exclusion semaphore services are slightly slower (in execution time) than semaphores since the priority of the owner may need to be changed, which requires CPU processing.

Resource Sharing without Objects Usage

Disable/Enable interrupts method will use uC/CPU services …

(1)…
(2)…
(3)…
(4)…

```
void YourFunction (void)
{
    CPU_SR_ALLOC();                 (1)


    CPU_CRITICAL_ENTER();           (2)
    Access the resource;            (3)
    CPU_CRITICAL_EXIT();            (4)
}
```

Lock/Unlock scheduler method …

(1)…
(2)…
(3)…
(4)…

```
void YourFunction (void)
{
    OS_ERR  err();                  (1)


    OSSchedLock(&err);              (2)
    Access the resource;            (3)
    OSSchedUnlock(&err);            (4)
}
```

Semaphores

- Semaphore is a kernel object defined by OS_SEM data type
- An application can have any number of semaphores
- Types of semaphores:
  - Binary: takes two values, 0 or 1
  - Counting: takes values between 0 and 255; 65,535; or 4,294,967,295; depending on the data length used …
- uC/OS-III keeps track of semaphore's value and all tasks waiting for the semaphore's availability
- ISRs are not allowed to use semaphores for sharing resources

Semaphore Services

- When semaphores are used for sharing resources, every semaphore function must be called from a task and never from an ISR.
- The same limitation does not apply when using semaphores for signaling.

Function Name	Operation
`OSSemCreate()`	Create a semaphore.
`OSSemDel()`	Delete a semaphore.
`OSSemPend()`	Wait on a semaphore.
`OSSemPendAbort()`	Abort the wait on a semaphore.
`OSSemPost()`	Release or signal a semaphore.
`OSSemSet()`	Force the semaphore count to a desired value.

Binary Semaphore

- A semaphore must be created
- A task that wants to acquire a resource must perform a pend (or wait) operation on the semaphore
- A task releases a semaphore by performing a post (or signal) operation on the semaphore

```
OS_SEM  MySem;                                  (1)


void  main (void)
{
    OS_ERR  err;
    :
    :
    OSInit(&err);
    :
    OSSemCreate(&MySem,                         (2)
                "My Semaphore",                 (3)
                1,                              (4)
                &err);                          (5)
    /* Check "err" */
    :
    /* Create task(s) */
    :
    OSStart(&err);
    (void)err;
}
```

Using a Semaphore to Access a Shared Resource

- (1) …
- (2) … timeout specified in clock ticks …
- (3) … specifies how to wait …
  - Blocking …
  - Non-blocking …
- (4) … time stamp post …
- …

```
void  Task1 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;


    while (DEF_ON) {
        :
        OSSemPend(&MySem,                                   (1)
                  0,                                        (2)
                  OS_OPT_PEND_BLOCKING,                     (3)
                  &ts,                                      (4)
                  &err);                                    (5)
        switch (err) {
            case OS_ERR_NONE:
                 Access Shared Resource;                    (6)
                 OSSemPost(&MySem,                          (7)
                           OS_OPT_POST_1,                   (8)
                           &err);                           (9)
                 /* Check "err" */
                 break;

            case OS_ERR_PEND_ABORT:
                 /* The pend was aborted by another task     */
                 break;

            case OS_ERR_OBJ_DEL:
                 /* The semaphore was deleted                */
                 break;

            default:
                 /* Other errors                             */
```

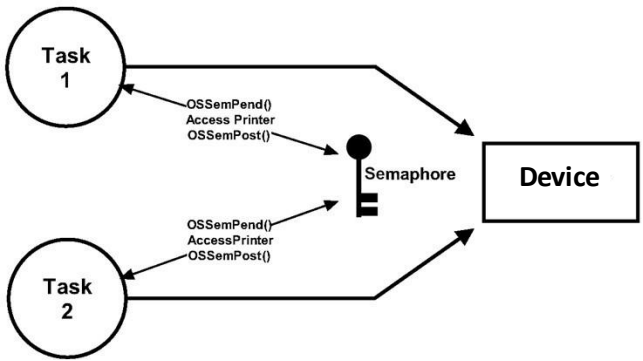
Using a Semaphore from Multiple Tasks to Access a Shared Resource

- Another task, must use the same procedure as Task1
- (1) …
- …

```
void  Task2 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;


    while (DEF_ON) {
        :
        OSSemPend(&MySem,                                    (1)
                  0,
                  OS_OPT_PEND_BLOCKING,
                  &ts,
                  &err);
        switch (err) {
            case OS_ERR_NONE:
                 Access Shared Resource;
                 OSSemPost(&MySem,
                           OS_OPT_POST_1,
                           &err);
                 /* Check "err" */
                 break;

            case OS_ERR_PEND_ABORT:
                 /* The pend was aborted by another task     */
                 break;

            case OS_ERR_OBJ_DEL:
                 /* The semaphore was deleted                */
                 break;

            default:
                 /* Other errors                             */
        }
```

Semaphore Used for Tasks to Share I/O Devices

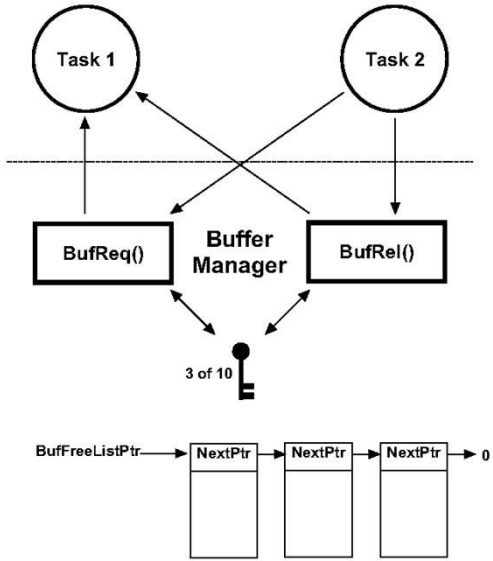
- Rule: to access the device each task must first obtain the device's key (semaphore)
- Each task must know about the existence of the semaphore to access the resource



Counting Semaphores Example

Counting semaphore is used when elements of a resource can be used by more than one task at the same time. Example, used in the management of a buffer pool.

```
BUF  *BufReq (void)
{
    BUF  *ptr;


    Wait on semaphore;
    ptr = OSMemGet(...) ;                    /* Get a buffer      */
    return (ptr);
}


void  BufRel (BUF *ptr)
{
    OSMemPut(..., (void *)ptr, ...);         /* Return the buffer */
    Signal semaphore;
}
```



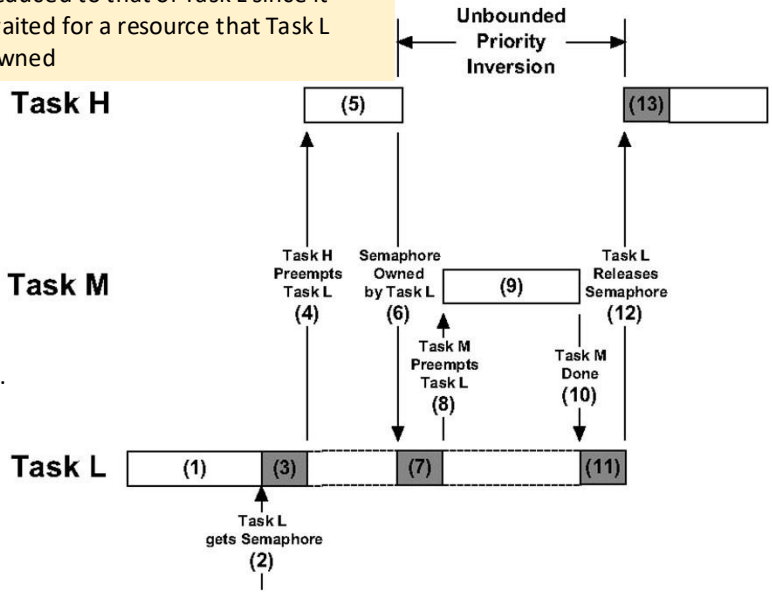
Semaphore Internals

- See reference for semaphore data structures and pending and posting

Priority Inversion

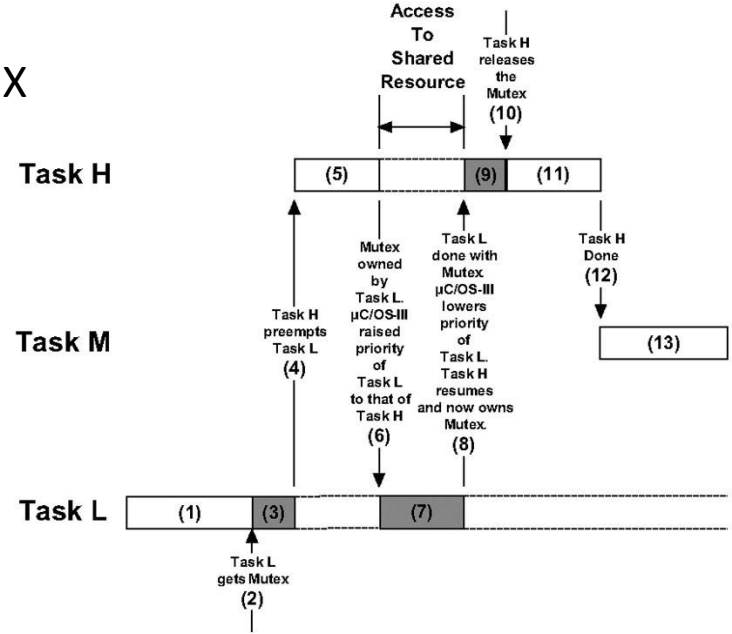
- What is priority inversion?
- Follow the 3 tasks example
- (1) … task L executes …
- (2) … task L gets sem S …
- (3) …
- (4) …
- (5) … task H wants sem S …
- (7) …
- (8) … task M starts executing …
- (9) …
- (10) …
- (11) … task L releases sem S …
- (12) …

The priority of Task H has been reduced to that of Task L since it waited for a resource that Task L owned



Mutual Exclusion Semaphore: MUTEX

- MUTEX is a special type of binary semaphore that eliminates the unbounded priority inversion
- …
- (6) …
- (7) …
- (8) …
- (9) …
- (10) …
- …



MUTEX Functions

- A mutex is a kernel object defined by the OS_MUTEX data type
- Mutex functionality is based on the full-priority inheritance protocol implemented by uC/OSIII
- ISRs cannot use a MUTEX

Function Name	Operation
`OSMutexCreate()`	Create a mutex.
`OSMutexDel()`	Delete a mutex.
`OSMutexPend()`	Wait on a mutex.
`OSMutexPendAbort()`	Abort the wait on a mutex.
`OSMutexPost()`	Release a mutex.

Usage of MUTEXes

```
OS_MUTEX          MyMutex;
SOME_STRUCT       MySharedResource;


void  MyTask (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;

    :
    while (DEF_ON) {
        OSMutexPend((OS_MUTEX *)&MyMutex,                          (1)
                    (OS_TICK   )0,
                    (OS_OPT    )OS_OPT_PEND_BLOCKING,
                    (CPU_TS    *)&ts,
                    (OS_ERR    *)&err);
        /* Check "err"                                             */  (2)
        /* Acquire shared resource if no error                     */
        MyLibFunction();                                           (3)
        OSMutexPost((OS_MUTEX *)&MyMutex,                          (7)
                    (OS_OPT    )OS_OPT_POST_NONE,
                    (OS_ERR    *)&err);
        /* Check "err"                                             */
    }
}
```

```
void  MyLibFunction (void)
{
    OS_ERR  err;
    CPU_TS  ts;

    OSMutexPend((OS_MUTEX *)&MyMutex,                              (4)
                (OS_TICK   )0,
                (OS_OPT    )OS_OPT_PEND_BLOCKING,
                (CPU_TS    *)&ts,
                (OS_ERR    *)&err);
    /* Check "err"                                                 */
    /* Access shared resource if no error                          */  (5)
    OSMutexPost((OS_MUTEX *)&MyMutex,                              (6)
                (OS_OPT    )OS_OPT_POST_NONE,
                (OS_ERR    *)&err);
    /* Check "err"                                                 */
}
```

MUTEX Internals

- A mutex is a kernel object defined by the OS_MUTEX data type, which is derived from the structure os_mutex (see os.h) as shown

```
typedef  struct  os_mutex  OS_MUTEX;                    (1)

struct  os_mutex {
    OS_OBJ_TYPE          Type;                          (2)
    CPU_CHAR            *NamePtr;                       (3)
    OS_PEND_LIST         PendList;                      (4)
    OS_MUTEX            *MutexGrpNextPtr;               (5)
    OS_TCB              *OwnerTCBPtr;                   (6)
    OS_NESTING_CTR       OwnerNestingCtr;               (7)
    CPU_TS               TS;                            (8)
};
```

Creating a MUTEX

- A mutual exclusion semaphore (mutex) must be created before it can be used by an application

```
OS_MUTEX  MyMutex;                                  (1)


void  MyTask (void *p_arg)
{
    OS_ERR  err;
    :
    :
    OSMutexCreate(&MyMutex,                         (2)
                  "My Mutex",                       (3)
                  &err);                            (4)
    /* Check "err" */
    :
    :
}
```

215

Pending on a Mutex

```
OS_MUTEX  MyMutex;


          void MyTask (void *p_arg)
          {
              OS_ERR  err;
              CPU_TS  ts;
              :
              while (DEF_ON) {
                  :
                  OSMutexPend(&MyMutex,             /* (1) Pointer to mutex
*/
                              10,                   /*     Wait up until this time for the
mutex      */
                              OS_OPT_PEND_BLOCKING, /*     Option(s)
*/
                              &ts,                  /*     Timestamp of when mutex was released
*/
                              &err);                /*     Pointer to Error returned
*/
                  :
                  /* Check "err"                       (2)
*/
                  :
                  OSMutexPost(&MyMutex,             /* (3) Pointer to mutex
*/
                              OS_OPT_POST_NONE,
                              &err);                /*     Pointer to Error returned
*/
```

Deadlock

A deadlock, is a situation in which two tasks are each unknowingly waiting for resources held by the other

```
void T1 (void *p_arg)
{
    while (DEF_ON) {
        Wait for event to occur;          (1)
        Acquire M1;                       (2)
        Access  R1;                       (3)
        :
        :
        \--------  Interrupt!             (4)
        :
        :                                 (8)
        Acquire M2;                       (9)
        Access  R2;
    }
}
```

```
void  T2 (void *p_arg)
{
    while (DEF_ON) {
        Wait for event to occur;          (5)
        Acquire M2;                       (6)
        Access  R2;
        :
        :
        Acquire M1;                       (7)
        Access  R1;
    }
}
```

Scenario of deadlock:

1. T1 executes before T2; T2 has a higher priority
2. T1 deadlocks at (9) waiting for M2 and T2 deadlocks at (7) waiting for M1

Techniques Used to Avoid Deadlocks

1. Each task should acquire all resources before proceeding, and in the same order
2. Always acquire resources in the same order
3. Use timeouts on pend calls – this method doesn't really solve the deadlock, it just breaks it

Homework

- Study all examples
- Study the priority inversion scenario and understand the difference between using semaphores and Mutex