Real-time Operating System

Kevin Wen-Hung Huang
LS 12, TU Dortmund

Embedded Systems vs RTOS

Embedded Systems

Automobiles

Medical

Entertainment

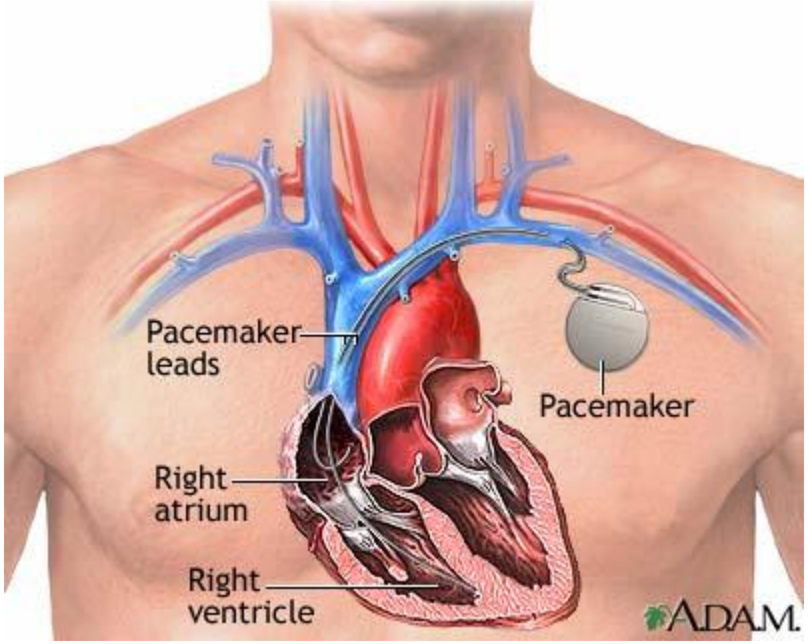
Handheld

Airplanes

Military

Real-Time Operating System (RTOS)

- An operating system intended to server real-time application requests
- Specified time constrains
- Applications
  - Automotive systems
  - Avionics
  - Pacemaker

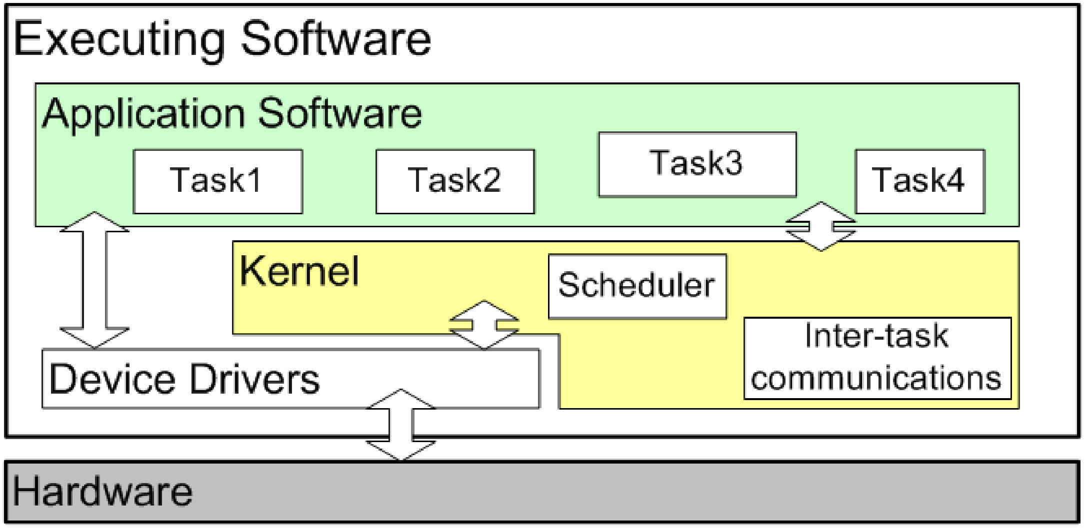


Stanford Cardiac Arrhythmia Service

Overview of FreeRTOS

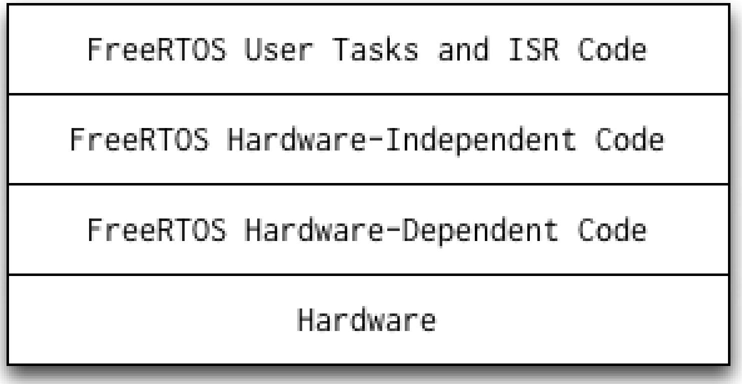
FreeRTOS

- A real-time operating system (RTOS)
  - Relatively small application
  - Various architectures support
- Three main areas
  - Tasks
  - Communications
  - The hardware wrapper



Hardware Considerations

- Hardware-dependent layer
  - Talk to the chip architecture you choose
- FreeRTOS ships with all the hardware-independent
  - ARM7, ARM Cortex-M3, various PICs, Silicon Labs 8051, etc.



FreeRTOS User Tasks and ISR Code

FreeRTOS Hardware-Independent Code

FreeRTOS Hardware-Dependent Code

Hardware

Ref: http://aosabook.org/en/freertos.html

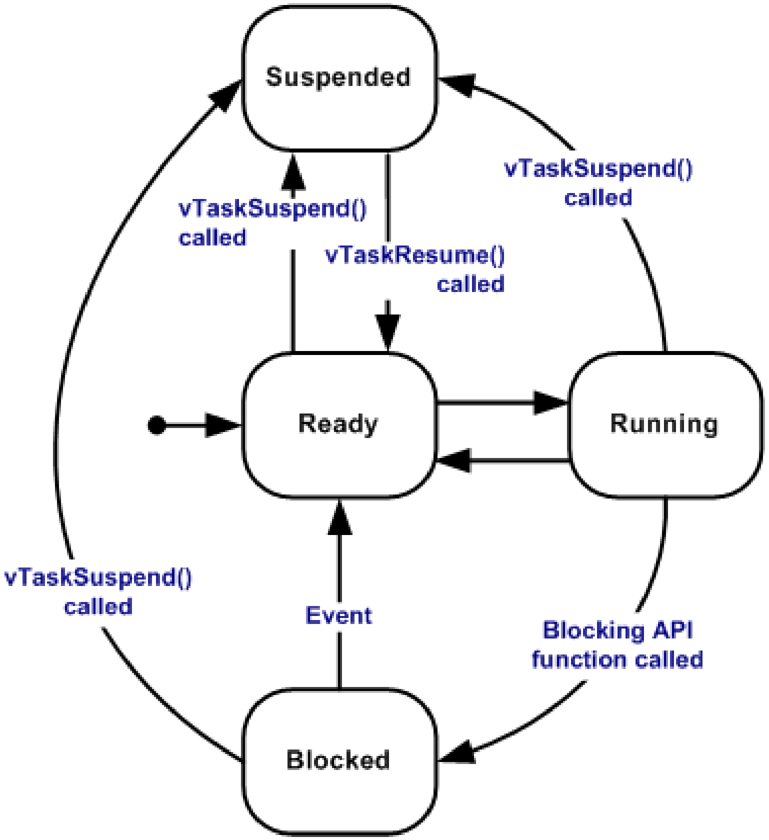
Variables and Functions Naming Conventions

- Variables prefix
  - c: char
  - s: short
  - l : long
  - x: portBASE_TYPE and any others
  - u: unsigned
  - p: pointer
    - Combinations are possible
- Function prefix
  - By the returning data type
  - v: void
- Macros everywhere
  - pdTRUE is 1, pdFALSE is 0
  - pdPASS is 1, pdFAIL is 0

Tasks
in RTOS

Tasks

- Running
  - When a task is actually executing
- Ready
  - A task that is able to execute is not currently running due to its lower priority
- Blocked
  - Waiting for either a temporal or external event
  - Always has a timeout
- Suspended
  - Via vTaskSuspend() and vTaskResume()
  - No timeout period allowed



Tasks in FreeRTOS

- pvTaskCode
    - A function that performs the computation of the task
- pcName
    - Name of the task used for debugging
- usStackDepth
    - Stack size of the process (task)
- pvParameters
    - Parameters passed to the process (task)
- uxPriority
    - Priority level
- xTaskHandle
    - Handler when c

```
portBASE_TYPE xTaskCreate(
                    pdTASK_CODE pvTaskCode,
                    const char * const pcName,
                    unsigned short usStackDepth,
                    void *pvParameters,
                    unsigned portBASE_TYPE uxPriority,
                    xTaskHandle *pvCreatedTask
);
```

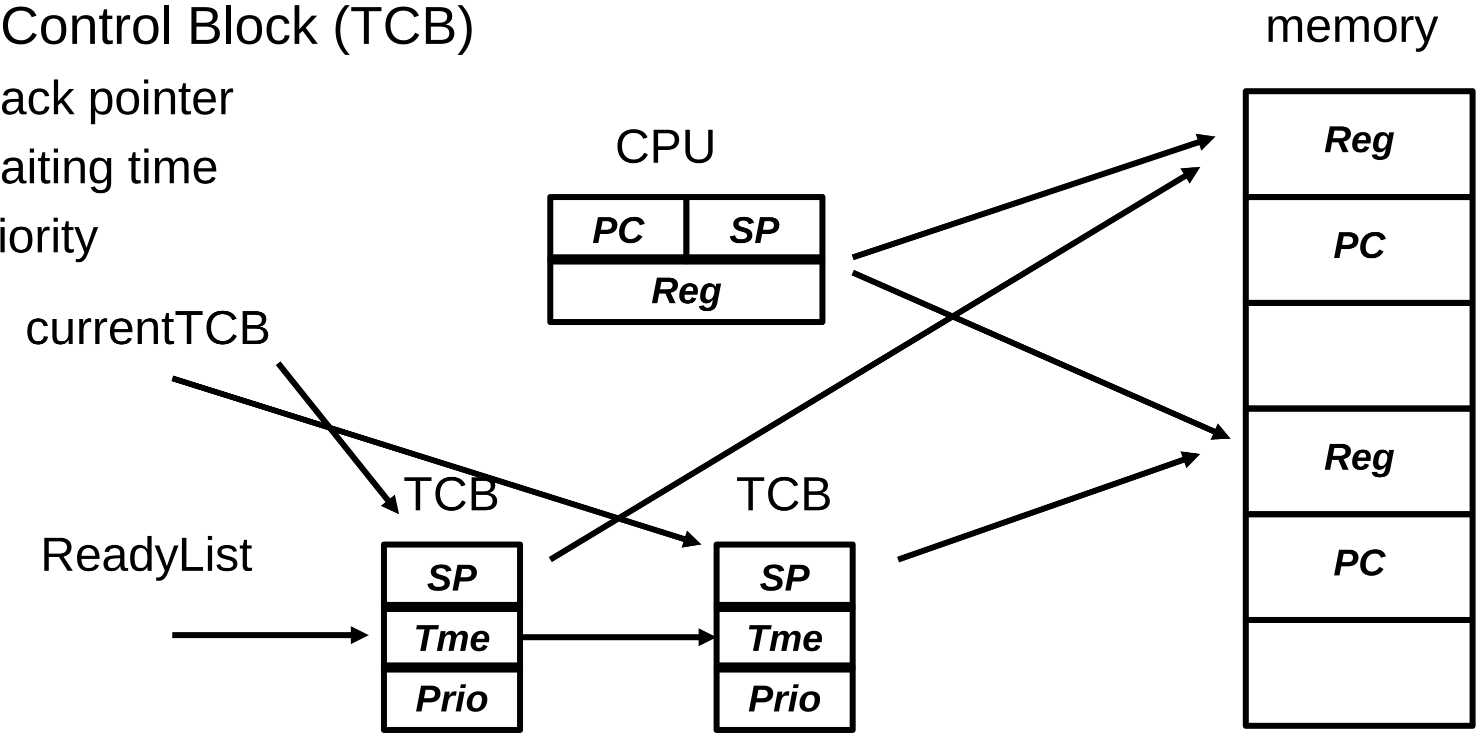
Tasks in FreeRTOS

```
/* Task to be created. */
void vTaskCode( void * pvParameters )
{
  for( ;; )
  {
      /* Task code goes here. */
  }
}
```

```
/* Function that creates a task. */
void vOtherFunction( void )
{
static unsigned char ucParameterToPass;
TaskHandle_t xHandle = NULL;

xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskIDLE_PRIORITY,
               &xHandle );
  configASSERT( xHandle );

  /* Use the handle to delete the task. */
  if( xHandle != NULL )
  {
      vTaskDelete( xHandle );
  }
 }
```

Task Control Block & Task Executions

- Central Processing Unit (CPU)
  - Program Counter (PC)
  - Stack Pointer (SP)
  - Registers
- Task Control Block (TCB)
  - Stack pointer
  - Waiting time
  - priority

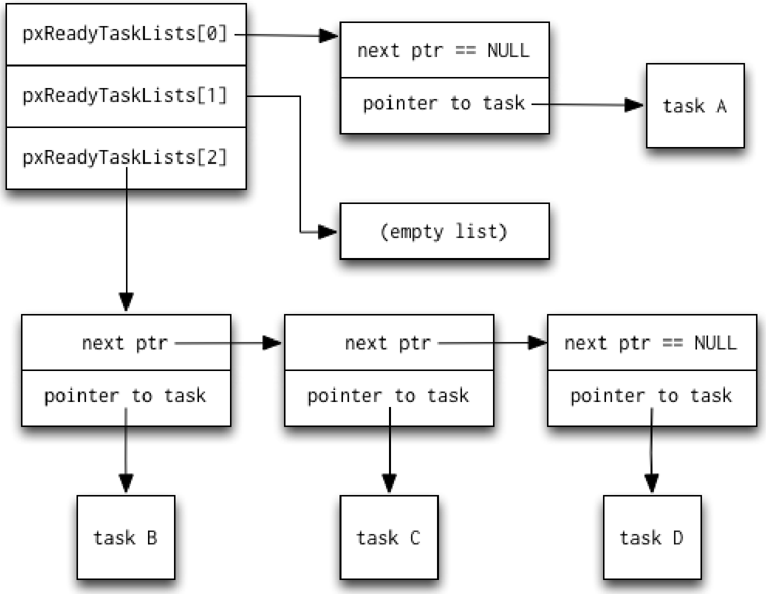


Task Control Block in FreeRTOS

```
typedef struct tskTaskControlBlock
{
    volatile portSTACK_TYPE *pxTopOfStack;      /*< Points to the location of the last item placed on the tasks stack.  THIS MUST BE THE FIRST MEMBER OF THE STRUCT. */

    xListItem               xGenericListItem;   /*< List item used to place the TCB in ready and blocked queues. */
    xListItem               xEventListItem;     /*< List item used to place the TCB in event lists. */
    unsigned portBASE_TYPE  uxPriority;         /*< The priority of the task where 0 is the lowest priority. */
    portSTACK_TYPE          *pxStack;           /*< Points to the start of the stack. */

    #if ( portSTACK_GROWTH > 0 )
        portSTACK_TYPE *pxEndOfStack;           /*< Used for stack overflow checking on architectures where the stack grows up from low memory. */
    #endif

    #if ( configUSE_MUTEXES == 1 )
        unsigned portBASE_TYPE uxBasePriority;  /*< The priority last assigned to the task - used by the priority inheritance mechanism. */
    #endif

} tskTCB;
```

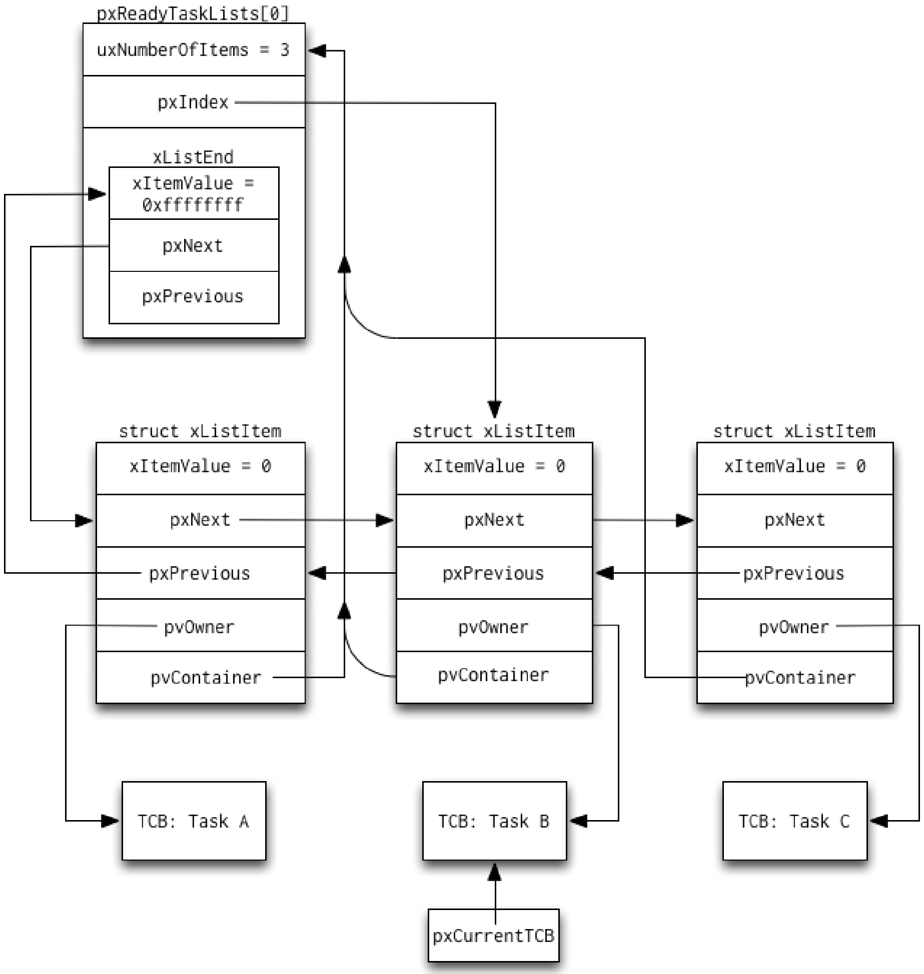
Task Priority & Ready List

- User-assigned priority
  - **configMAX_PRIORITIES**
- An array of task lists
  - **static xList pxReadyTasksLists[configMAX_PRIORITIES]; /*< *Prioritised ready tasks.* */**



Ref: http://aosabook.org/en/freertos.html

Overview of Lists



Ref: http://aosabook.org/en/freertos.html

Lists in FreeRTOS

```
struct xLIST_ITEM
{
    portTickType xItemValue;                /*< The value being listed.
In most cases this is used to sort the list in descending order. */
    volatile struct xLIST_ITEM * pxNext;    /*< Pointer to the
next xListItem in the list. */
    volatile struct xLIST_ITEM * pxPrevious;/*< Pointer to the
previous xListItem in the list. */
    void * pvOwner;                         /*< Pointer to the object
(normally a TCB) that contains the list item.  There is therefore a
two way link between the object containing the list item and the
list item itself. */
    void * pvContainer;                     /*< Pointer to the list in which
this list item is placed (if any). */
};
```

Scheduling in RTOS

Context Switch

- What is a context switch
  - The computing process of **storing** and **restoring** state of a CPU
  - Not for free
- When to switch
  - Multitasking
  - Interrupt handling
  - User and kernel mode change

Scheduling in FreeRTOS

- A ready queue maintains the TCB pointers of the tasks that are ready to be executed.
- The scheduler then selects the highest-priority job (task instance) in the ready queue for execution
- Fixed-priority scheduling
  - All the task instances of the task will then use the same priority for executing
  - If there are multiple task instances in the ready queue with the same priority, they share the processor and FreeRTOS uses a shared scheme to run these tasks

Heartbeat of FreeRTOS

- System periodic tick
  - Millisecond range
- vTaskSwitchContext
  - Selects Highest-priority ready task
  - Puts it in pxCurrentTCB

```
 /* Find the highest priority queue that contains ready tasks. */

while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopReadyPriority ] )
 ) )
   {
       --uxTopReadyPriority;
   }

/* listGET_OWNER_OF_NEXT_ENTRY walks through the list, so the tasks of
the same priority get an equal share of the processor time. */
   listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB,
&( pxReadyTasksLists[ uxTopReadyPriority ] ) );
```

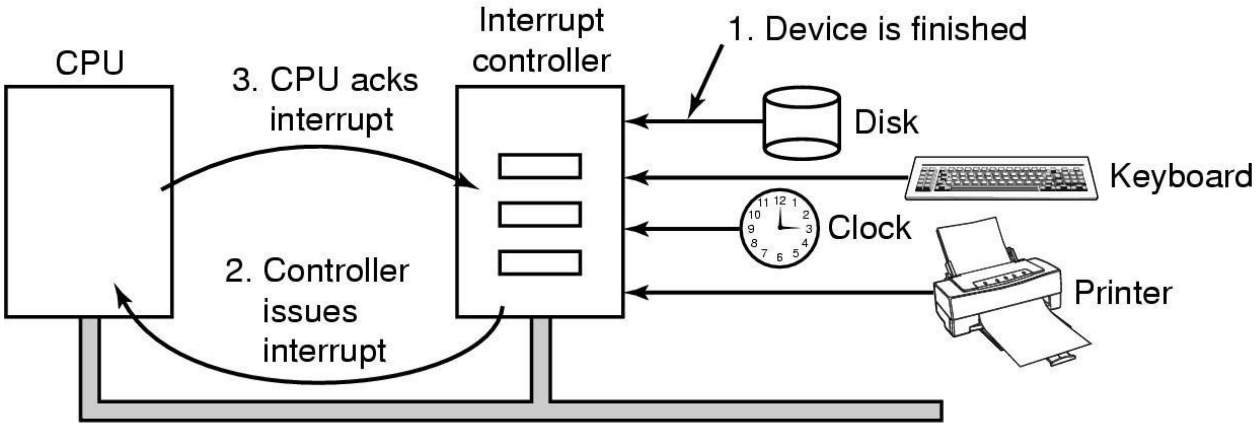
Communication & Synchronization in RTOS

Interrupt Handling in RTOS

- The needs of interrupt handling
  - Help peripherals “talk” to microprocessors
  - These devices occasionally need CPU service
    - We can’t predict when
- External events typically occur on a macroscopic timescale
  - we want to keep the CPU busy between events
- Three types:
  - Software interrupts
  - Hardware interrupts
  - Exceptions
    - Occur in response to error state in the processor or during debugging (trace, breakpoint, etc.)

Possible Solutions

- Polling
  - Constantly testing a port to see if data is available.
  - Inefficient, as it requires CPU for busy-looping
- Interrupt
  - an external hardware/software event that causes the CPU to interrupt the current instruction sequence
    - Interrupt Service Routine (ISR)
  - More efficient, as the CPU can continue while it is waiting for I/O

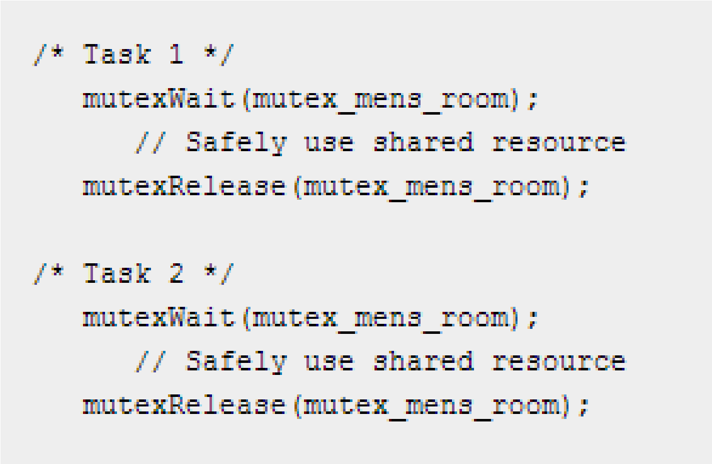


What to Notice for Interrupt Handling in RTOS

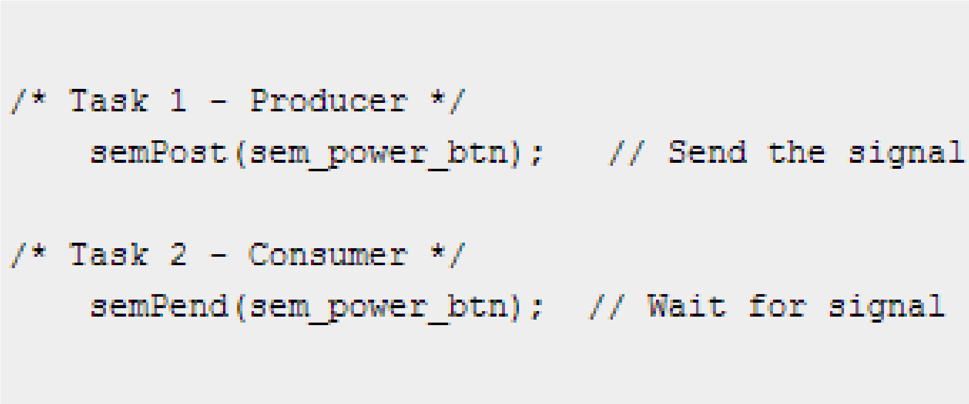
- General
  - The interrupt handler should be fast, efficient, and predictable
  - The execution time of an interrupt handler should be bounded
  - It is normally desirable to keep each ISR as short as possible
- FreeRTOS:
  - No specific event processing strategy on the application designer
  - Feature provision for simple implementation

Mutexes & Semaphores

- Mutexes (lock)
  - a key and a locker
  - critical sections
- Semaphores (toilet)
  - persons and rooms
  - Producer & consumer



```
/* Task 1 */
   mutexWait(mutex_mens_room);
      // Safely use shared resource
   mutexRelease(mutex_mens_room);

/* Task 2 */
   mutexWait(mutex_mens_room);
      // Safely use shared resource
   mutexRelease(mutex_mens_room);
```



```
/* Task 1 - Producer */
    semPost(sem_power_btn);   // Send the signal

/* Task 2 - Consumer */
    semPend(sem_power_btn);  // Wait for signal
```

Ref: http://www.barrgroup.com/Embedded-Systems/How-To/RTOS-Mutex-Semaphore#endnote1

Critical Sections

- A **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.
- Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use.
  - Race condition
  - No preemptive allowed

```
// Global data declaration and initialization
int GlobalData;
```

```
int LocalData;

// Thread 1 code
if (GlobalData!= 0) {

    LocalData = GlobalData;
}
```

```
// Thread 2 code
if (SomeCondition != FALSE)
{
    GlobalData = 0;
}
```

Which one can be used with multiple calls from different tasks?

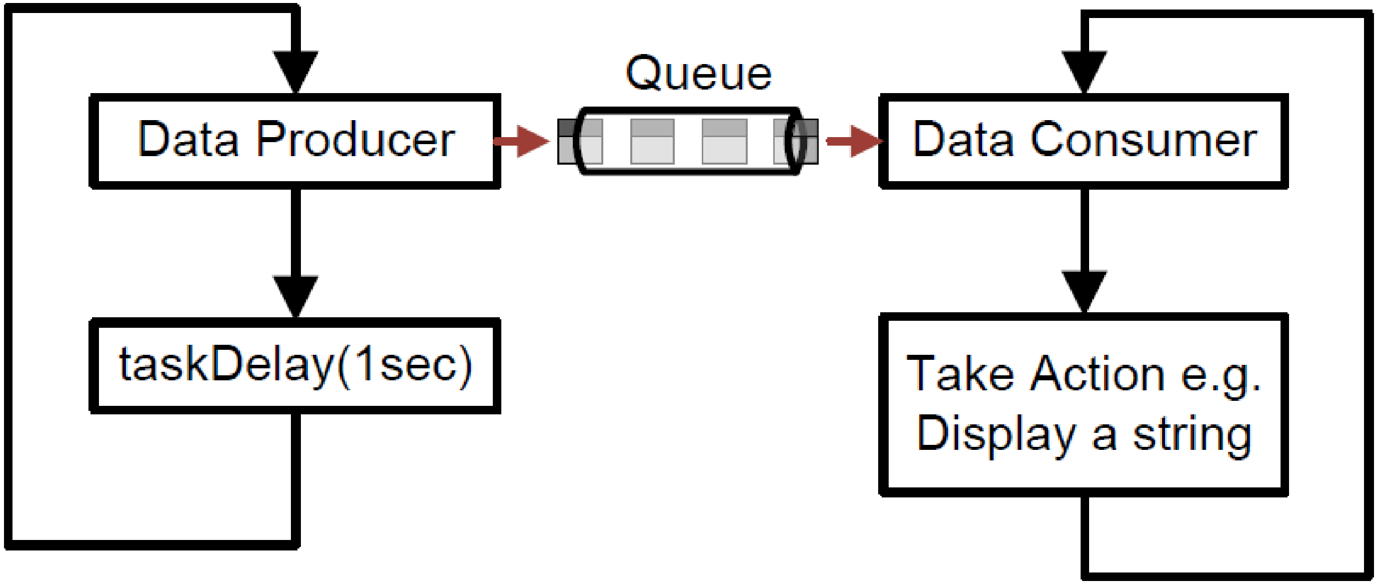
```
long addOneHundered(long lVar1)
{
   long lVar2;
   lVar2 = lVar1+100;

   return lVar2;
}
```

```
long addOneHundered(long lVar1)
{
   static long lVar2;
   lVar2 = lVar1+100;

   return lVar2;
}
```

Critical Sections in FreeRTOS

```
void vPortEnterCritical( void )
{
    vPortDisableInterrupts();
    uxCriticalNesting++;
}
void vPortExitCritical( void )
{
    /* Check for unmatched exits. */
    if ( uxCriticalNesting > 0 )
    {
        uxCriticalNesting--;
    }
    /* If we have reached 0 then re-enable the interrupts. */
    if( uxCriticalNesting == 0 )
    {
        /* Have we missed ticks? This is the equivalent of pending an interrupt. */
        vPortEnableInterrupts();
    }
}
```

Queue


Queue
Data Producer
taskDelay(1sec)
Data Consumer
Take Action e.g. Display a string

Semaphores in FreeRTOS

- Do not store any actual data
  - Only care how many entries are currently occupied

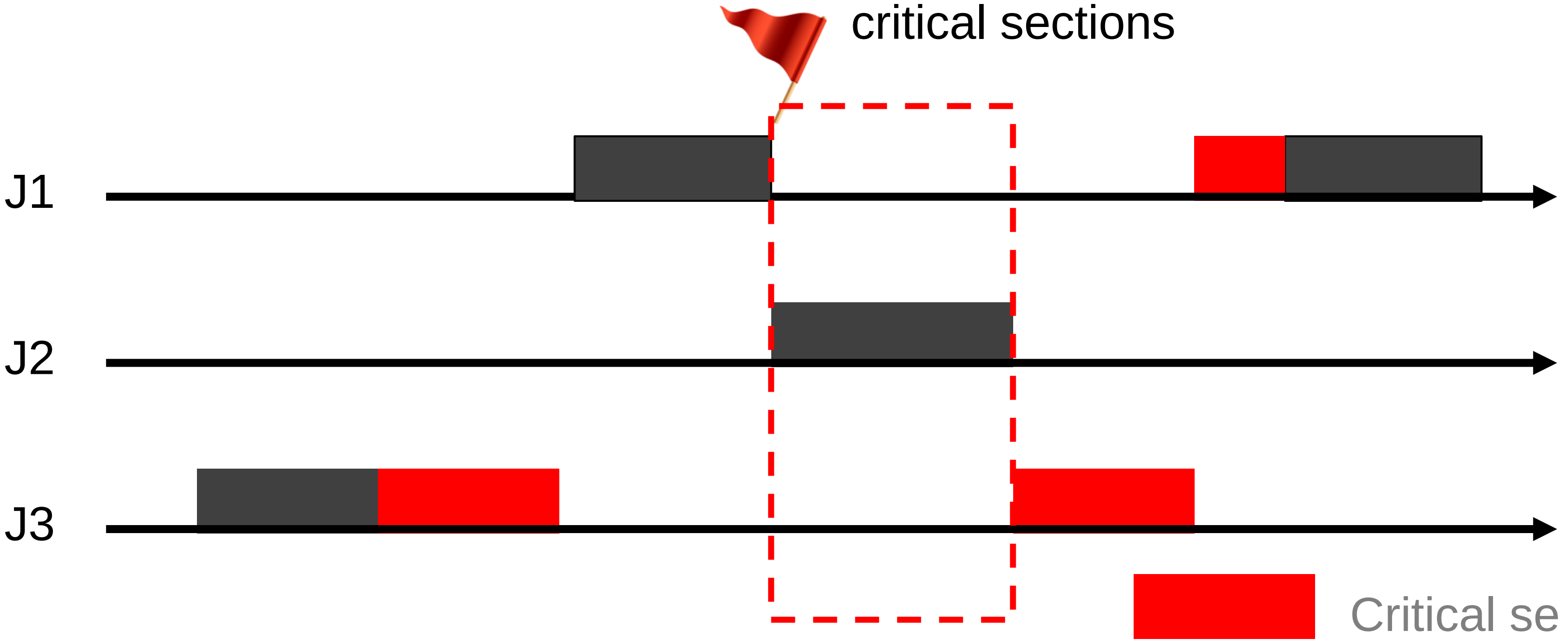
```
#define vSemaphoreCreateBinary( xSemaphore ) {  xSemaphore =
xQueueCreate( ( unsigned portBASE_TYPE ) 1,
semSEMAPHORE_QUEUE_ITEM_LENGTH );
if( xSemaphore != NULL ) {   xSemaphoreGive( xSemaphore );     }
}
```

```
#define xSemaphoreTake( xSemaphore, xBlockTime )
xQueueGenericReceive( ( xQueueHandle ) xSemaphore, NULL,
xBlockTime, pdFALSE )
```

```
#define xSemaphoreGive( xSemaphore )
xQueueGenericSend( ( xQueueHandle ) xSemaphore, NULL,
semGIVE_BLOCK_TIME, queueSEND_TO_BACK )
```

Priority Inversion (Recap)

- A medium-priority task preempts a lower-priority task using a shared resource on which the higher-priority task is pending.



J1 intends to access critical sections

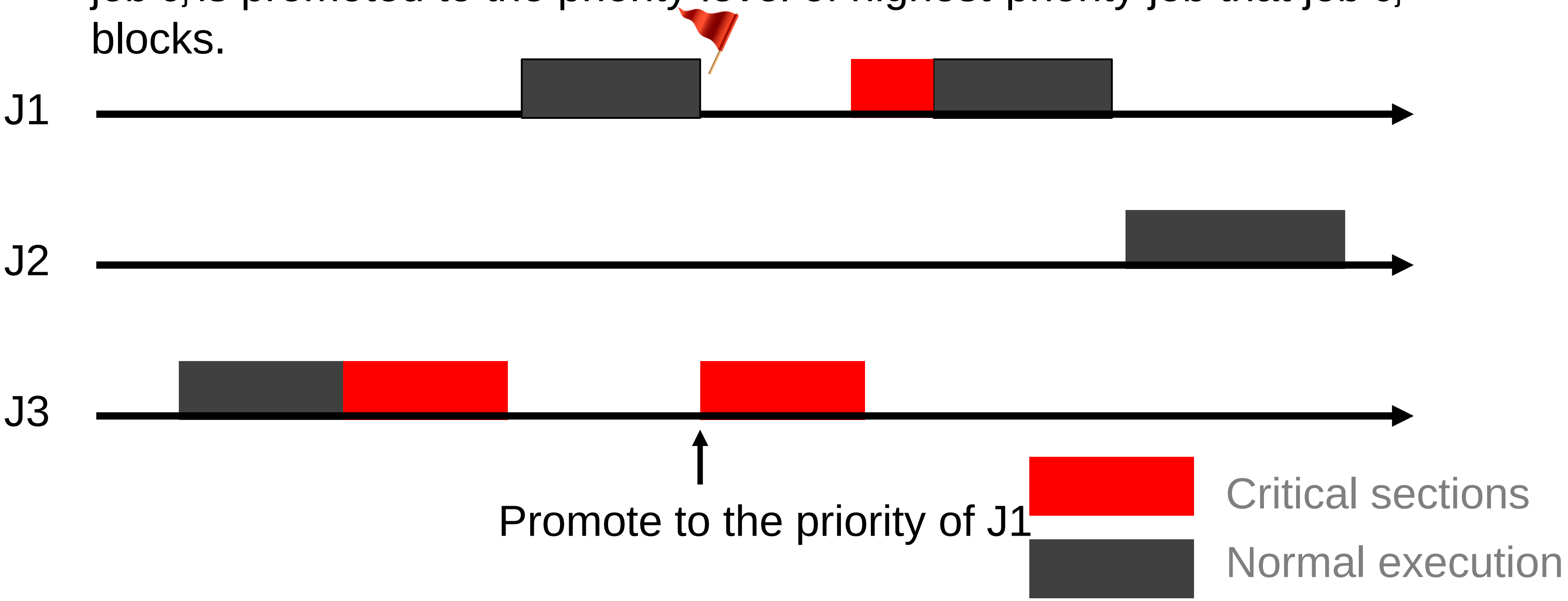
Priority inversion between J1 and J2

🟥 Critical sections

⬛ Normal execution

Workaround - PIP

- Disallow preemption
  - Simple
  - Unnecessary blockings occur
- Priority Inheritance Protocol (PIP)
  - When a lower-priority job J_j blocks a higher-priority job, the priority of job J_j is promoted to the priority level of highest-priority job that job J_j blocks.



PIP in FreeRTOS

```
void vTaskPriorityInherit( xTaskHandle * const pxMutexHolder )
    {
    tskTCB * const pxTCB = ( tskTCB * ) pxMutexHolder;

        if( pxTCB->uxPriority < pxCurrentTCB->uxPriority )
        {
            /* Adjust the mutex holder state to account for its new priority. */
            listSET_LIST_ITEM_VALUE( &( pxTCB->xEventListItem ), configMAX_PRIORITIES - ( portTickType )
pxCurrentTCB->uxPriority );

            /* If the task being modified is in the ready state it will need to
            be moved in to a new list. */
            if( listIS_CONTAINED_WITHIN( &( pxReadyTasksLists[ pxTCB->uxPriority ] ), &( pxTCB-
>xGenericListItem ) ) )
            {
                vListRemove( &( pxTCB->xGenericListItem ) );

                /* Inherit the priority before being moved into the new list. */
                pxTCB->uxPriority = pxCurrentTCB->uxPriority;
                prvAddTaskToReadyQueue( pxTCB );
            }
            else
            {
                /* Just inherit the priority. */
                pxTCB->uxPriority = pxCurrentTCB->uxPriority;
            }
        }
    }
```

Questions?

Ref: http://www.coach-em.com/