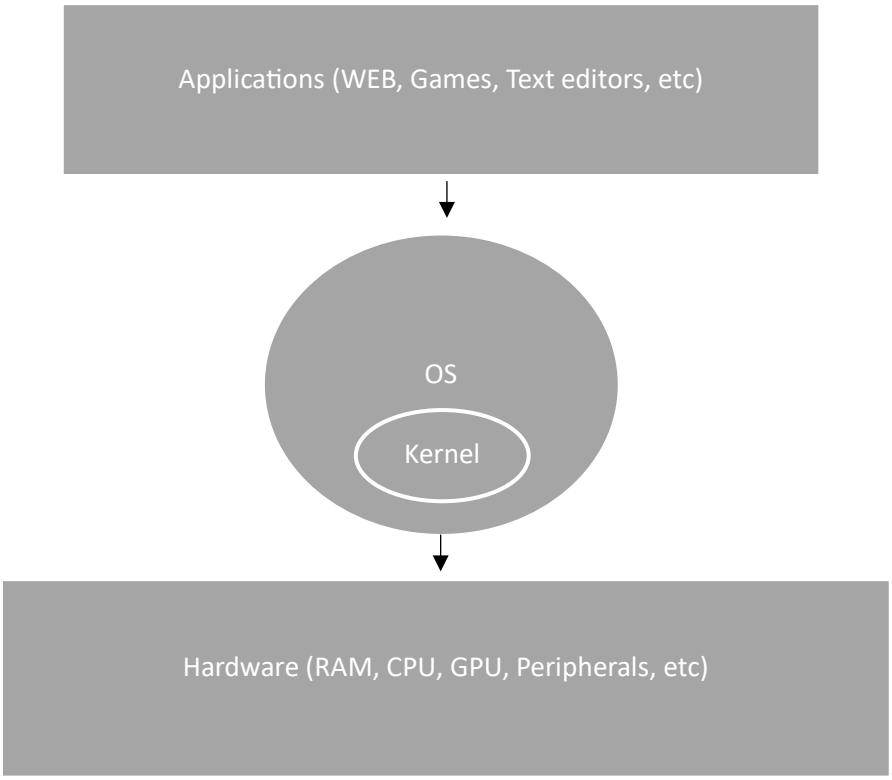
RTOS Concepts

What is an OS?

Has direct access to hardware.

- Manage hardware according to predefined rules and policies.
- Hide hardware complexity from the application perspective.
- Enable multitasking.



What is RTOS?

Real-time operating systems have similar functions as general-purpose OS (GPOS), like Linux, Microsoft Windows, but are designed to run applications with very precise timing. RTOS is designed for critical systems and for devices like **microcontrollers** that are timing-specific.

Hard vs soft RTOS:

Hard real-time:

- A system that can always meet the desired deadlines (100 percent), even under a worst-case system load. In hard real-time systems, missing a deadline, even a single time, can have fatal consequences. Hard real-time systems are used in cases where a particular task, usually involving life safety issues, needs to be performed within a particular time frame, otherwise a catastrophic event will occur.

Soft real-time:

- A system that can meet the desired deadlines on average. A soft real-time system will give reduced average latency but not a guaranteed maximum response time.
- Soft RTOS can miss a few deadlines (such as dropping a few frames in a video application) without failing the overall system.

Task and Multitasking:

Task:

- Task also called A thread.
- It's the basic block of an application written under RTOS.
- It's a simple program with an infinite loop.
- Task has its own stack, CPU registers, and priority.

Example:

```
void vTaskFunction( void *pvParameters)

{

for(;;)

{--Task application code here. -- }

}
```

Multitasking,

- Is the process of switching the CPU between several tasks.
- It maximizes the utilization of the CPU.
- Provide modular construction for our application, makes the
- application design is easier.

Polling and interruptions

Polling:

- We are checking an event through an infinite loop.
- checks all devices in a **round robin** fashion.
- The main drawback of this method the application needs to wait and check whether the new information has arrived, so, its a waste of time of the processor.
- Also, it may miss some events.

Interrupt

- An external or internal event that interrupts the processor to inform it that a device needs its service.
- When an event happens, the processor jumps to the Interrupt Service routine (ISR).
- ISR is a function executes once the related interrupt happens.

What is a Resource?

A Resource is an entity used by the task; it can be:

- I/O device, Printer, Keyboard, Display, Variable, Array, Structure, File

What is a Shared Resource?

A shared Resource is a resource that can be used by more than one task.

- Each task should have exclusive access to the shared resource to prevent data corruption.
- There are techniques to ensure exclusive access of the resource like mutual exclusion.

Critical Section of Code

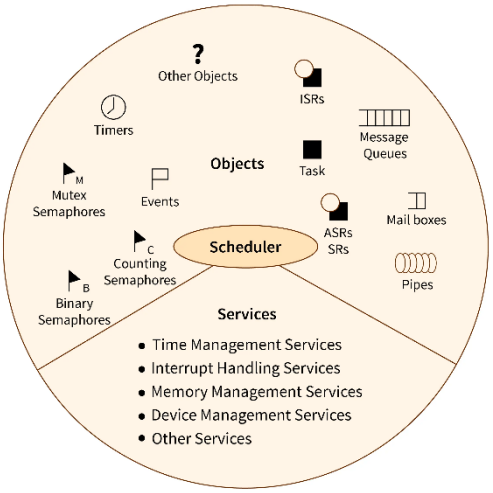
- Also called critical region.
- Is a section of code that shouldn't be interrupted.
- Most RTOS Systems enable us to disable the interrupt before
- this section then enables it again after that.
- There are also other methods to protect this critical section.

RTOS Kernel

Kernel is the core component of any OS.

kernel components:

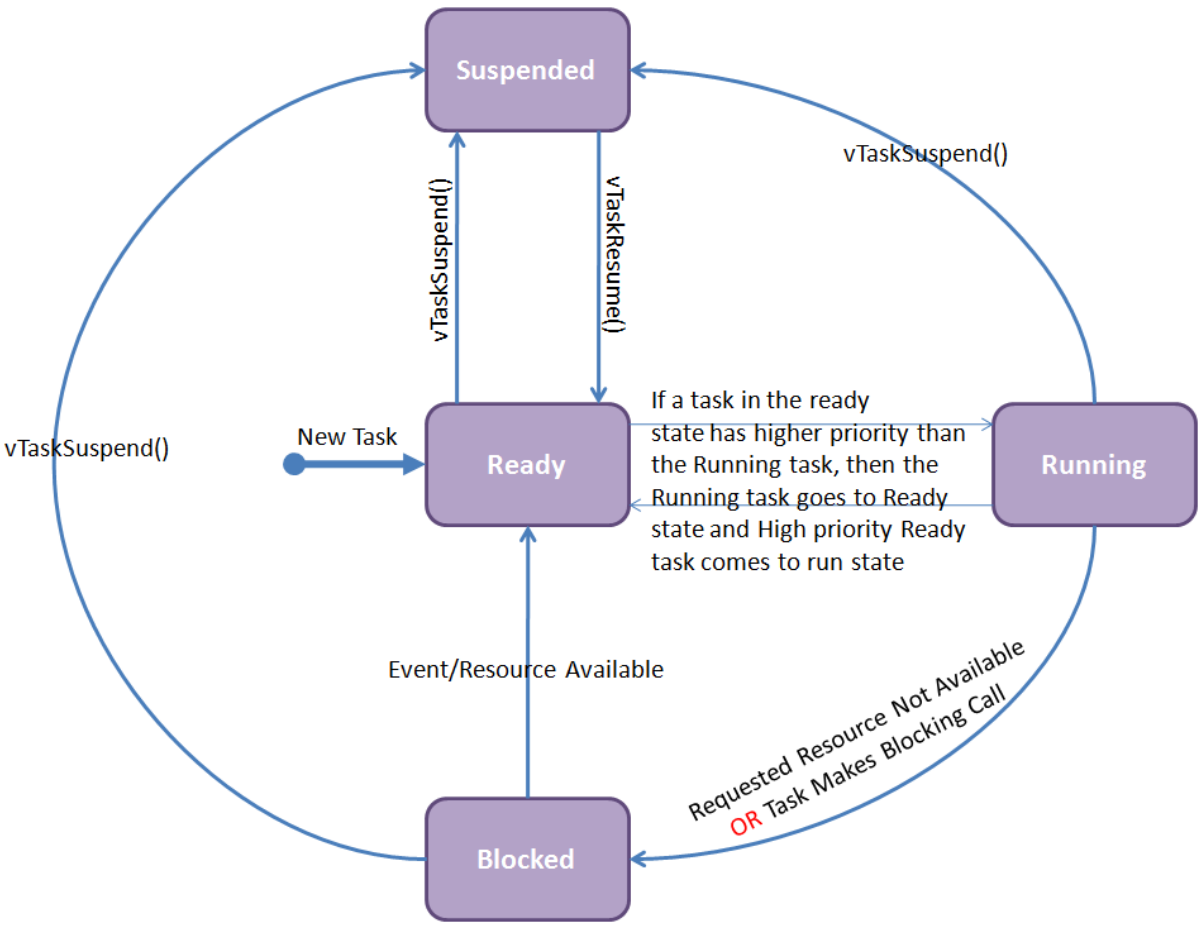
- **Scheduler** is a set of algorithms that determines which task executes when.
- **Objects** are special kernel constructs that help developers create applications for real-time embedded systems.
- **Services** are operations that the kernel performs on an object.



Priority Based Kernels

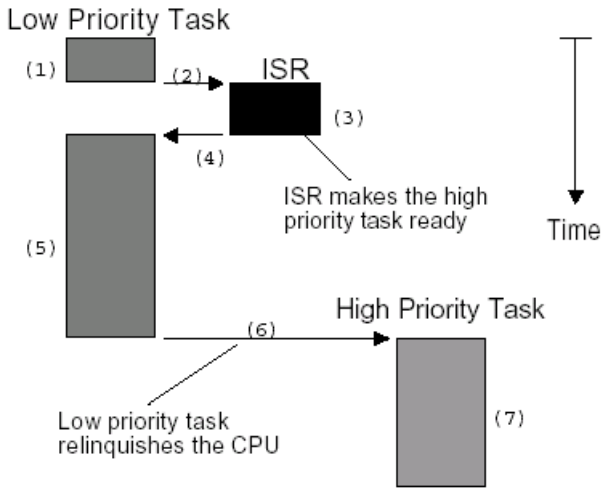
- Are Kernels that decide which task will run regarding its priority.
- Most real time kernels are priority based.
- Each task takes a priority based on its importance.
- The Highest priority task is always ready to run.
- There are two types of priority-based kernels: Non-Preemptive and Preemptive Kernel.

RTOS Task status



Non preemptive kernel:

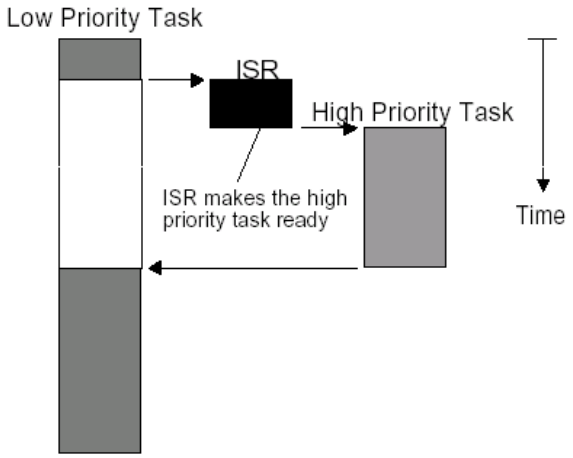
- An ISR (Interrupt Service Routine) makes higher priority task ready to run.
- The ISR always return to the interrupted task.
- The new higher priority task will run when the current task gives up the CPU.
- This minimizes the data corruption risks in multitasking as each task finish before the other begins.
- Task response time will equal to the time of the longest task.



Preemptive kernel:

Here the highest priority task ready to run always given the control to the CPU.

- ISR makes the highest priority task run.
- The kernel next run the highest priority task in the ready queue.
- Here the response time to the highest priority task is it's best.
- Corruption of data may happen for non-protected shared resources.



RTOS Scheduler

Scheduler is the core Component of any RTOS kernel.

- It’s a set of algorithms that determines which task executes when.
- It's keeping track on the status of each task and decides which to run.
- In Most RTOSs the developers are the ones who sets the priority of each task, regarding to this priority the scheduler will decide which task will run.
- The scheduler assumes that you knew what you were doing while setting tasks priority.
- A bad design for tasks priority, may leads to a high priority task hogs the processor for long time, this is called CPU starvation.
- It's keeping track on the status of each task and decides which to run.
- Scheduler has no control on tasks on the blocked status.
- If tasks are blocked the scheduler waits an event to unblock these tasks, like an external interrupt from pushing a button.
- If no events happened, surely, it's a bad design from your side.

What if two tasks have the same priority are ready?

- Some RTOSs make it illegal to set two tasks with the same priority, and here the kernel limits the number of tasks in an application to the number of priority levels.
- Others will time slice between the two tasks (Round robin).

- Some will run one task until it blocks the run of the other task.

System tasks

- The Tasks the system uses to run its internal operations.
- The RTOS system tasks have reserved priorities that we shouldn't use in our application tasks as it may affect the overall system performance or behaviour.

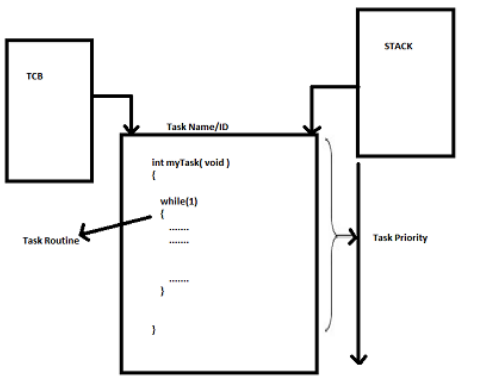
Examples of system tasks include:

- **initialization or startup task:** Initializes the system and creates and starts system tasks,
- **idle task**: uses up processor idle cycles when no other activity is present, set to the lowest priority, and executes in an endless loop, ensures the processor program counter is always valid when no other tasks are running, user can implement his own idle task for example: a power conservation code, such as system suspension, after a period of idle time.
- **logging task**: logs system messages.
- **exception-handling task**: handles exceptions.
- **debug agent task**: allows debugging with a host debugger.

Task object data

Each task has an associated:

- a name,
- a unique ID,
- a priority,
- a task control block (TCB),
- a stack,
- and a task routine,



Task Context

- Every task has its own context (its own Data).
- Every Task Created has its own data structure called Task Control Block (TCB).
- Task saves its data like tasks status, ID, priority, stack pointer, Pointer to function (task itself) ....., in its TCB.
- Task context also saved in its own stack and CPU registers.

Context switching between tasks

Context switching: Is how the processor switch between the context of one task to the another, so the system must:

- Save the state of the old process.
- Then load the saved state for the new process.
- The new process continues from where it left off just before the context switch.
- When the task is not running, its context is frozen within the TCB, to be restored the next time the task runs.
- **The Dispatcher**: Is the part of the scheduler that performs context switching.
- **Context Switch Time**: Is the time it takes for the scheduler to switch from one task to another.
- frequent context switching makes a performance overhead.

Shared data problem

The shared data problem occurs when several functions (or ISRs or tasks) share a variable. Shared data problem can arise in a system when another higher priority task finishes an operation and modifies the data or a variable before the completion of previous task operations.

Non reentrant functions

- Is a function that can' t be used between more than one task.
- Can't be interrupted or a data loss will happen.

Reentrant functions

Is a function that can be used between more than one task without fear of data corruption.

Can be interrupted at any time and resumed without loss of data.

Reentrant functions either:

- use local variables, or use protected global variables.
- Can't Call other non reentrant function.

Gray area of reentrancy

Some Functions and some operation on a shared data are processor and compiler dependent.

Example:

- **printf()** Function Is **reentrant** or non **reentrant**?

The answer is it's depended on the processor and on the compiler.

How to protect shared data

Using Mutual Exclusion access. Examples on Mutual Exclusion methods are:

- Disable and enable interrupts.
- Disable the Scheduler.
- Using Semaphores.

Disable and enable interrupts

Most of Systems have this technique.

- Disabling the interrupt for long time affect the response to your system which known by Interrupt Latency.
- **Interrupt Latency:** is the time taken by a system to respond to an interrupt.

So, disable the interrupt should be for as little time as possible.

This is the only way for a task to share a variable with ISR.

Disable and enable the scheduler

If we don't share data with any ISR, then it's better to disable and enable scheduling.

While the scheduler is locked, the interrupts is enabled, and if interrupt happen, the ISR is executed immediately.

As the scheduling is disabled, when the ISR finish, the kernel

will return to the interrupted task not the highest priority one.

Disabling the scheduler also is not the best solution.

Semaphores

- Is a kernel object that one or more threads of execution can acquire or release for the purposes of synchronization or mutual exclusion.
- A semaphore is like a key that enables a task to carry out some operation or to access a resource.
- When a task acquires the semaphore, no other task can access the resource that is protected by the semaphore.
- Other tasks acquire the semaphore will be suspended until the semaphore is released by its current owner.

Semaphores parameters and data structures

When a semaphore is created, the kernel assigns to it:

- An associated semaphore control block (SCB).
- A unique ID.
- A value (binary or a count) depending on its type.
- A task waiting list.

Semaphore Types

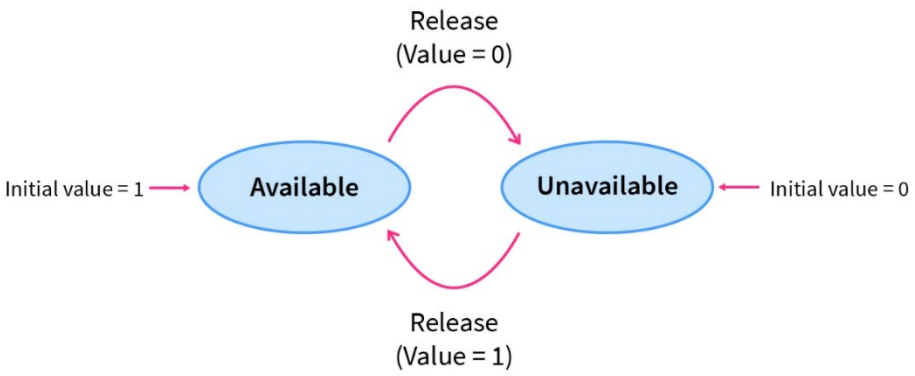
Binary Semaphore:

- Its value = 0, if it's not available.
- Its value = 1, if it's available.

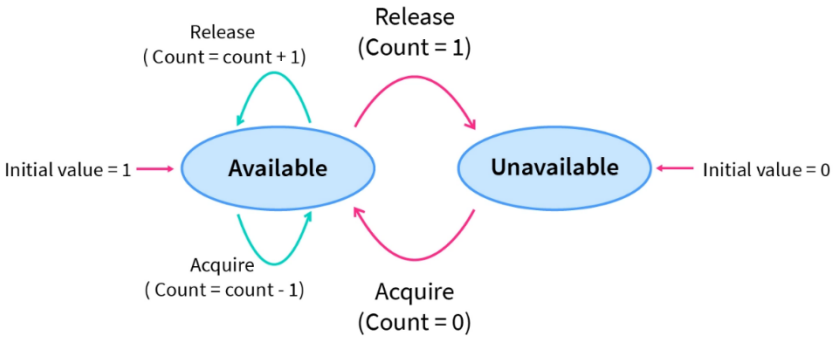
Counting Semaphore:

- Its value = 0, if it's not available.
- Its value > 0, if it's available.

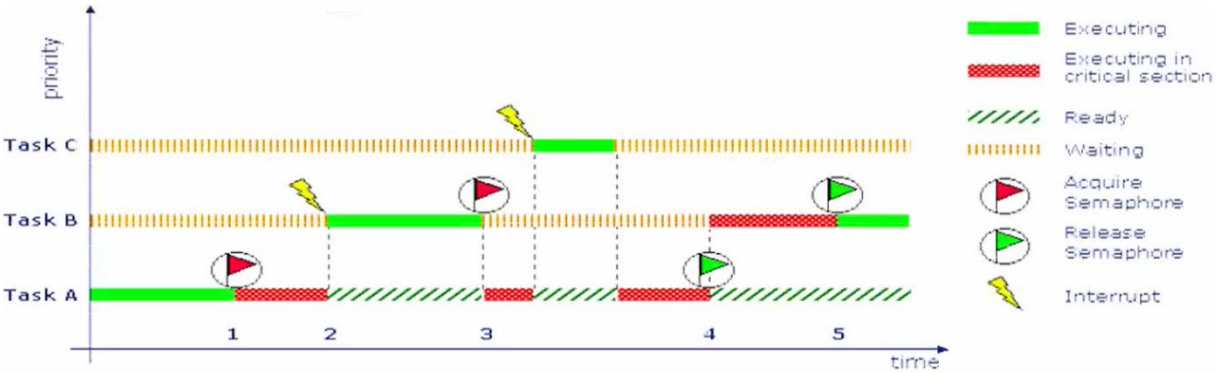
Binary semaphore:



Counting semaphore:



Mutual exclusion with binary semaphore:



Using semaphore to access shared data doesn't affect the interrupt latency.

If ISR or the current running task makes a higher priority task to run it run immediately.

Dead lock problem:

Also called Deadly Embrace.

When two tasks are waiting the resource held by the other.

Example:

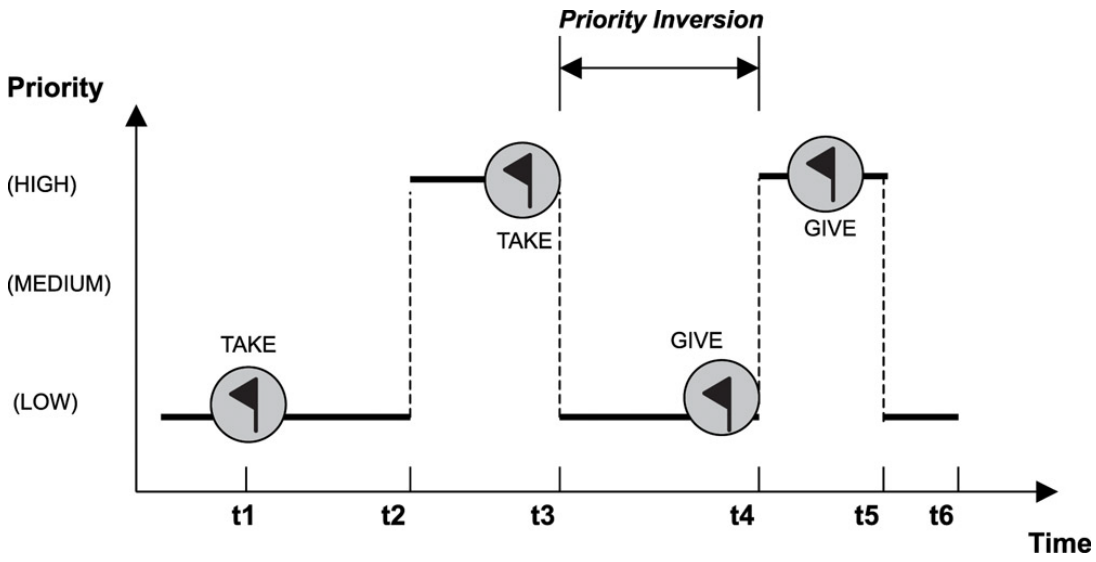
- Task1 has an exclusive access to Resourcel,
- And Task2 has an exclusive access to Resource2.
- If Task1 needs an exclusive access to Resource2,
- And Task2 needs an exclusive access to Resourcel.

- Both the tasks will be blocked, and a DeadLock happen.

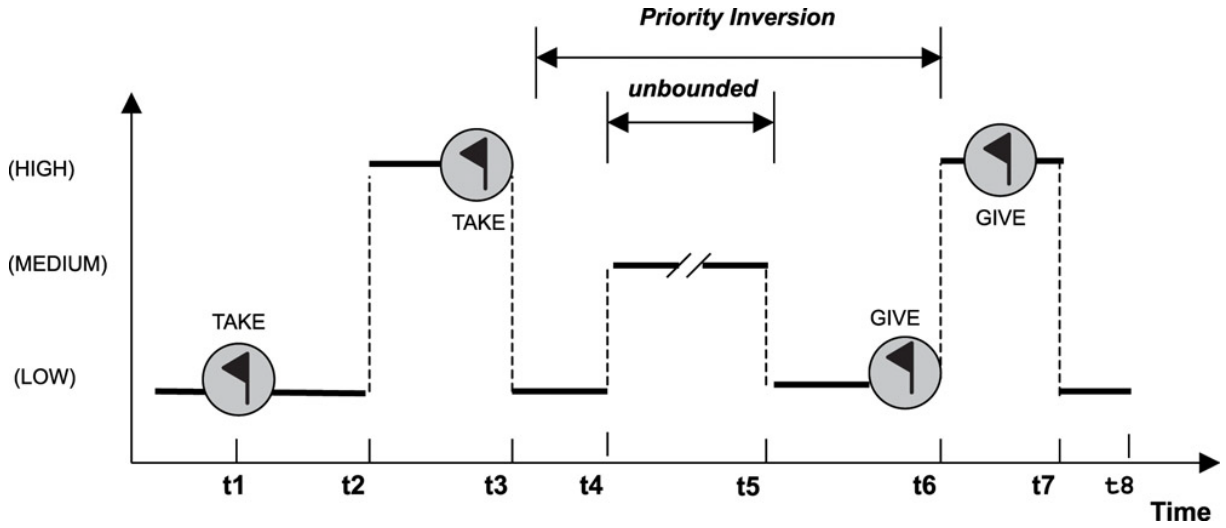
Avoid dead lock

- Through a timeout, If the resource is not available for a certain time, the task will resume executing.
- Good Design.

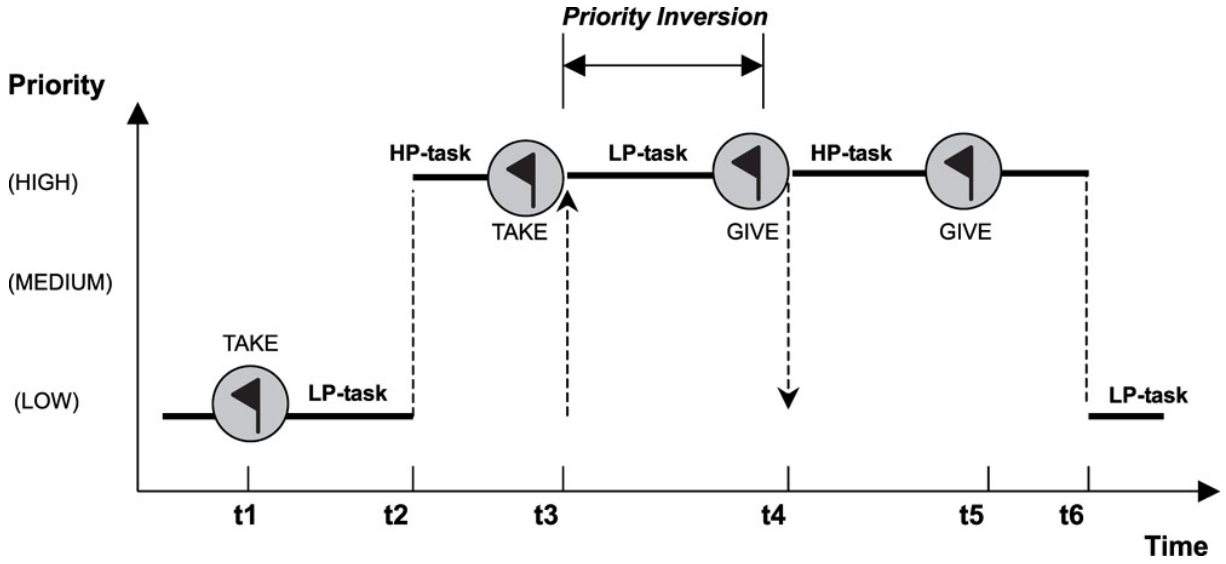
Priority inversion



Priority inversion problem (Extended priority inversion)



Extended Priority inversion solution (Priority Inheritance)



Race Condition

when two or more tasks have access to a shared resource and they try to edit it at the same time,

To prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time.

CPU Starvation

CPU starvation: occurs when higher priority tasks use all of the CPU execution time and lower priority tasks do not get to run.

In a preemptive multitasking environment, If higher priority tasks are not designed to block, CPU starvation can result.

Mutex

- Mutex is short for Mutual Exclusion.
- Mutex is a special type of binary semaphore used for controlling access to the shared resource.
- Mutexes include **a priority inheritance mechanism** to avoid extended priority inversion problem, this ensure the higher priority task is kept in the blocked state for the shortest time possible.
- Priority inheritance does not cure priority inversion! It just minimizes its effect in some situations.
- Hard real time applications should be designed such that priority inversion does not happen in the first place.

Mutex vs Binary Semaphore

Ownership:

- Mutex semaphore is "owned" by the task that takes it, so when a task locks (acquires) a mutex only it can unlock (release) it.
- Binary semaphore has no owner, can be unlocked by any task,

Usage:

- Mutex is a locking mechanism used to synchronize access to a resource.
- Semaphore is signalling mechanism ("I am done, you can carry on" kind of signal.

Determinism vs Responsiveness

- Determinism: determine the time that every task will be executed.
- Responsiveness: How your application be responsive to (External or Internal) events.
- **RTOS Increase responsiveness and decrease determinism.**

Why RTOS?

Multitasking:

- Allows you to break a complex problem into simpler Pieces.
- Focus on the development of each task rather than building a scheduler.
- Increase the Efficiency of using the CPU.

Services:

- Kernel services are provided for resource management, memory management, event handling, messaging, interrupt handling and more.

Structure:

- Structure design of your application.

Portability:

- Your application will run on any platform the RTOS runs on.
- You can build an application for multiple targets from the same codebase.

Security:

- Some RTOS offers some security features, like encryption support for various communication protocols memory protection unit (MPU) support.

Debugging:

- Some IDES (such as IAR Embedded Workbench) have plugins that show nice live data about your running process such as task CPU utilization and stack utilization.

Support:

- Some RTOSs provides you with technical support and gives you access to the expertise of the engineers who developed each module, and they can advise and support you on your project.

Why not RTOS?

Resources:

- RTOS will require extra resources.
- Processing overhead due to context switch complex algorithms.
- memory usage overhead due to OS source size.

Determinism:

- RTOS decreases determinism.

Design:

- Needs very careful design, and more developing skills.
- It is never easy to port an RTOS.

Debugging:

- complex debugging (due to race conditions on resources shared among tasks)

Cost:

- Lots of RTOS needs expensive license, so product cost Will increase.

Thank you!