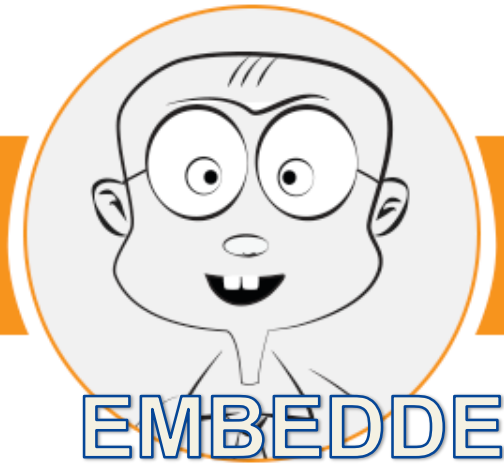


EMBEDDED SYSTEM COURSE

LECTURE 9: INTRODUCE TO BASIC REAL-TIME APPLICATIONS AND RTOS

Learning Goals

- Introduce about the real-time and its application.
- Introduce about most important parts in the real-time operating system including kernel, tasks, processes, scheduler, non-preemptive/preemptive kernel.
- Introduce basic concepts on how to synchronize among tasks in RTOS.

Table of contents

Table of contents

Real-Time Applications

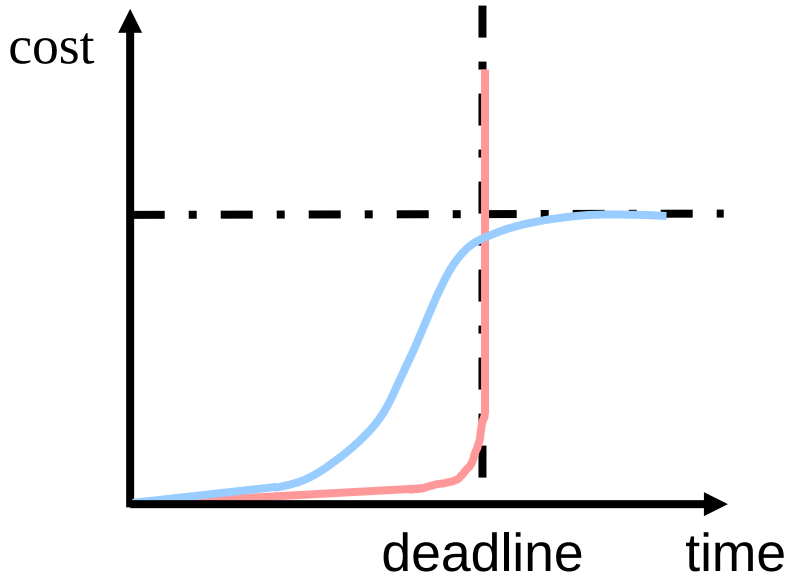
- Process control:
  - Food processing
  - Chemical plants
- Automotive:
  - Engine controls
  - Anti-lock braking systems
- Office automation:
  - FAX machines
  - Copiers
- Computer peripherals:
  - Printers
  - Terminals
  - Scanners
  - Modems
- Robots
- Aerospace:
  - Flight management systems
  - Weapons systems
  - Jet engine controls
- Domestic:
  - Microwave ovens
  - Dishwashers
  - Washing machines
  - Thermostats

What is Real-Time

- If a task must be completed within a given time, it is said to be a real-time task

- The correct result of a real time system depends on:
  - a correct answer/reaction to a stimuli
  - a point of time, when the result will be delivered

- Type of real time systems
  - hard real time
  - soft real time



Features of real-time systems

- Most real-time systems do not provide the features found in a standard desktop system.
- Reasons include
  - ✓ Real-time systems are typically single-purpose.
  - ✓ Real-time systems often do not require interfacing with a user.
  - ✓ Features found in a desktop PC require more substantial hardware than what is typically available in a real-time system.

Hard Real-Time System

- Timing is critical and deadline cannot be missed
  - ✓ If the failure to meet the deadline is considered to be a fatal fault
- Examples:
  - ✓ Nuclear reactors
  - ✓ Flight controller
  - ✓ Air bag deployment
  - ✓ Anti-lock brakes

Soft Real-Time System

- A miss of timing constraints is undesirable. However, a few misses do not serious harm
  - ✓ The timing requirements are often specified in probability terms
  - ✓ The time constraints are guaranteed on a statistical basis
- Examples:
  - ✓ Multimedia Streaming
  - ✓ Electronic games
  - ✓ Quality-of-Service (QoS) guarantees
  - ✓ Automatic teller machine (ATM)
    - If the ATM takes 30 seconds longer than the ideal, the user still won't walk away

Real Time Operating Systems

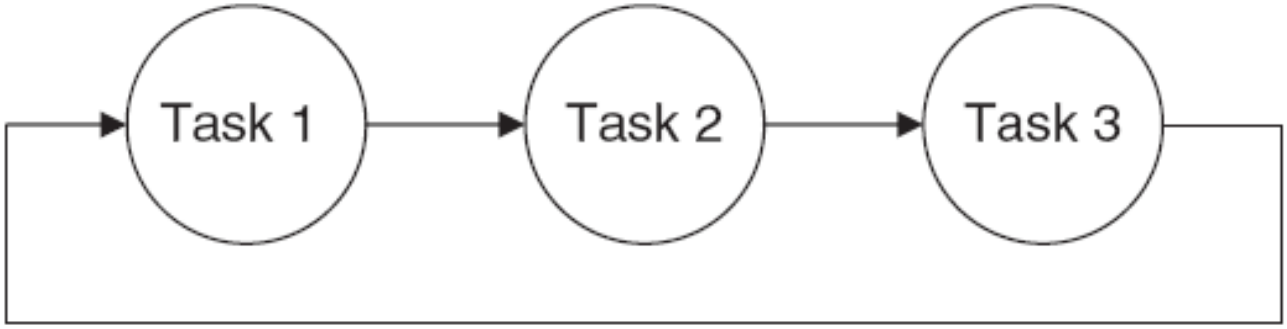
- A real-time operating system manages the time of a microprocessor or microcontroller.
  - **Features of an RTOS:**
    - Allows multi-tasking
    - Scheduling of the tasks with priorities
    - Synchronization of the resource access
    - Inter-task communication
    - Time predictable
    - Interrupt handling

Why we need
the real-time operating system (RTOS) ?

Table of contents

Real-Time application by Using State Machines

- Normally, we can design and implement a embedded application/ by using a simple state machines
- State machines are simple constructs used to perform several activities, usually in a sequence.



State machine implementation

Real-Time application by Using State Machines

```
for(;;)
{
        state = 1;
        switch (state)
        {
                CASE 1:
                        implement TASK 1
                        state++;
                        break;
                CASE 2:
                        implement TASK 2
                        state++;
                        break;
                CASE 3:
                        implement TASK 3
                        state = 1;
                        break;
        }
        Delay_ms(n);
}
```

State machine implementation in C

Real-Time application by Using State Machines

```
for(;;)
{
        state = 1;
        switch (state)
        {
                CASE 1:
                        implement TASK 1
                        state = 2;
                        break;
                CASE 2:
                        implement TASK 2
                        state = 3;
                        break;
                CASE 3:
                        implement TASK 3
                        state = 1;
                        break;
        }
        Delay_ms(n);
}
```

Selecting the next state from the current state

Real-Time application by Using State Machines

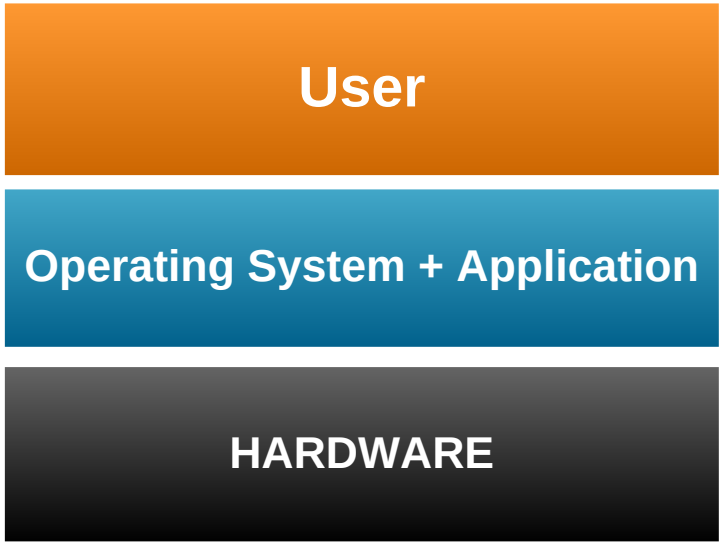
- State machines, although easy to implement, are primitive and have limited application. They can only be used in systems which are not truly responsive, where the task activities are well-defined and the tasks are not prioritized.
- Moreover, some tasks may be more important than others. We may want some tasks to run whenever they become eligible.

This kind of implementation of tasks
requires a sophisticated system like Embedded OS – RTOS.

Embedded Operating Systems

- Fusion of the application and the OS to one unit

- Characteristics
  - Resource management
    - Primary internal resources
  - Less overhead
  - Code of the OS and the application mostly reside in ROM



Real-Time Operating System – RTOS

- An RTOS will typically use specialized scheduling algorithms in order to provide the real-time developer with the tools necessary to produce deterministic behavior in the final system.

- An RTOS is valued more for how quickly and/or predictably it can respond to a particular event than for the amount of work it can perform over a given period of time.

- Key factors in an RTOS are therefore a minimal interrupt latency and a minimal thread switching latency.

Why use an RTOS?

- Plan to use drivers that are available with an RTOS
- Would like to spend your time developing application code and not creating or maintaining a scheduling system
- Multi-thread support with synchronization
- Portability of application code to other CPUs
- Resource handling
- Add new features without affecting higher priority functions
- Support for upper layer protocols such as:
  - ❑ TCP/IP, USB, Flash Systems, Web Servers,
  - ❑ CAN protocols, Embedded GUI, SSL, SNMP

RTOS Design Philosophies

- Two basic designs exist:
  - Event-driven (priority scheduling) designs switch tasks only when an event of higher priority needs service, called pre-emptive priority.
  - Time-sharing designs switch tasks on a clock interrupt, and on events, called round robin (Cooperative scheduling). Time-sharing designs switch tasks more often than is strictly needed, but give smoother, more deterministic multitasking, giving the illusion that a process or user has sole use of a machine.
- Newer RTOSes almost invariably implement time-sharing scheduling with priority driven pre-emptive scheduling.

RTOS Kernel

• The kernel is the part of a multitasking system responsible for the management of tasks (that is, for managing the CPU's time) and communication between tasks.

• The fundamental service provided by the kernel is context switching.

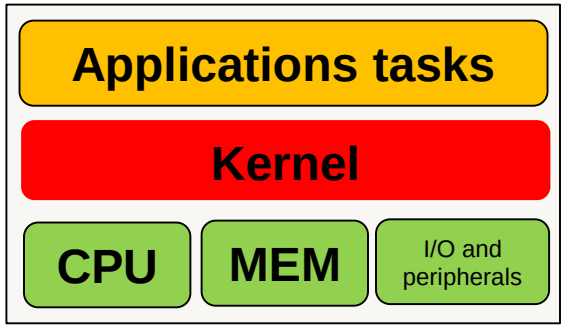
• The use of a real-time kernel will generally simplify the design of systems by allowing the application to be divided into multiple tasks managed by the kernel.

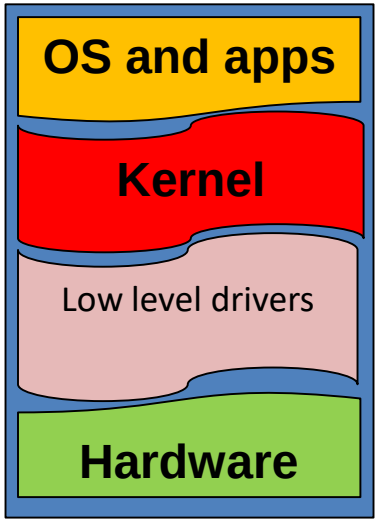
• A kernel will add overhead to your system because it requires extra memory
(code space), additional RAM for the kernel data structures but most importantly, each task requires its own stack space which has a tendency to eat up RAM quite quickly.

• A kernel will also consume CPU time (typically between 2 and 5%).

•A kernel can allow you to make better use of your CPU by providing you with indispensible services such as semaphore management, mailboxes, queues, time delays, etc.

• Once you design a system using a real-time kernel, you will not want to go back to a foreground/background system





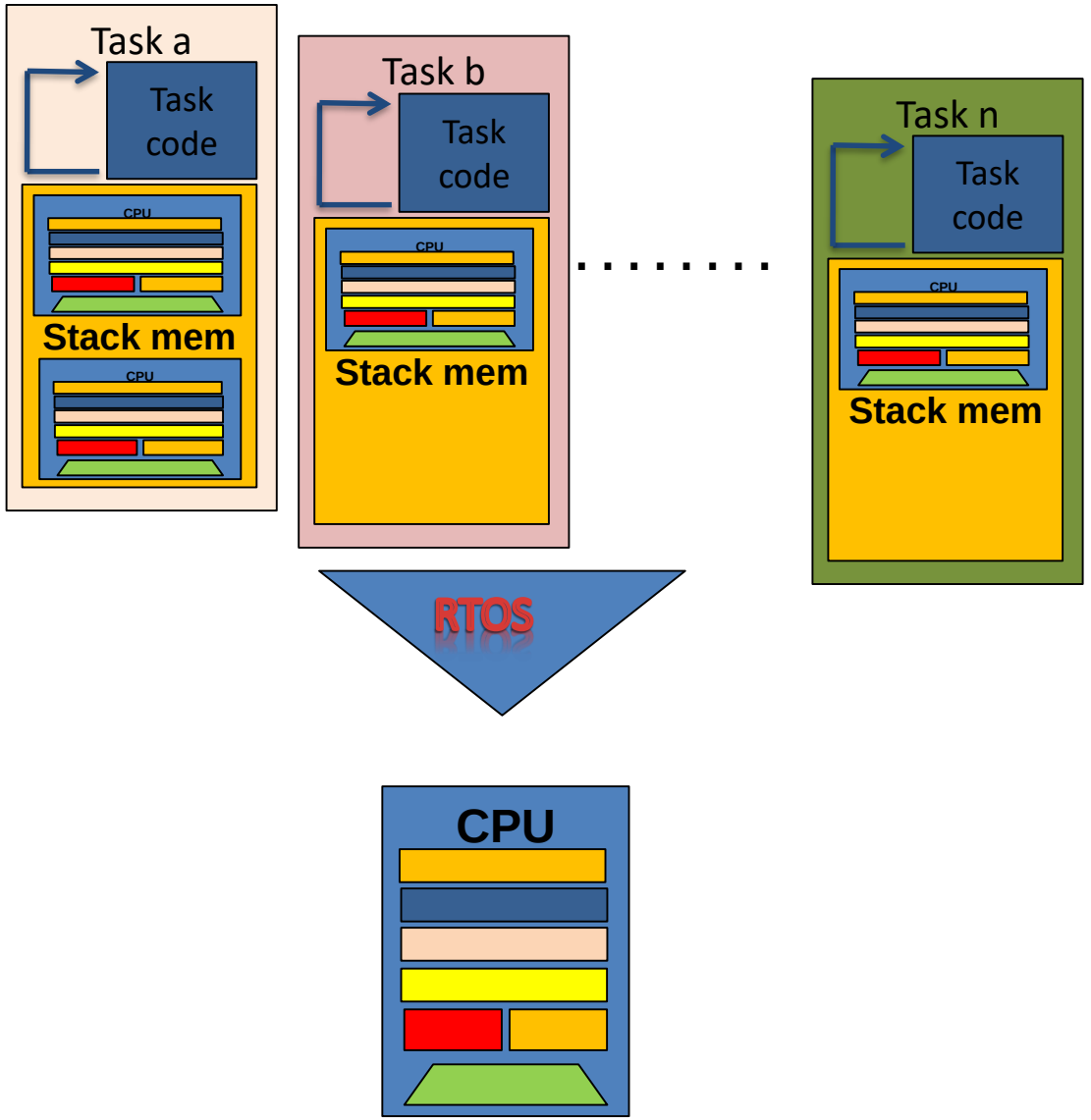
RTOS Task and Processes

A task *is a simple program that thinks it has the CPU all to itself.*

The design process for a real-time application involves splitting the work to be done into tasks which are responsible for a portion of the problem.

Each task is assigned a priority, its own set of CPU registers, and its own stack area

Each task typically is an infinite loop that can be in any one of the allowed states



RTOS Task and Processes (cont.)

• A priority is assigned to each task. The more important the task, the higher the priority given to it.

• **Static Priorities**

Task priorities are said to be static when the priority of each task does not change during the application's execution. Each task is thus given a fixed priority at compile time. All the tasks and their timing constraints are known at compile time in a system where priorities are static.

• **Dynamic Priorities**

Task priorities are said to be dynamic if the priority of tasks can be changed during the application's execution; each task can change its priority at run-time. This is a desirable feature to have in a real-time kernel to avoid priority inversions (see later).

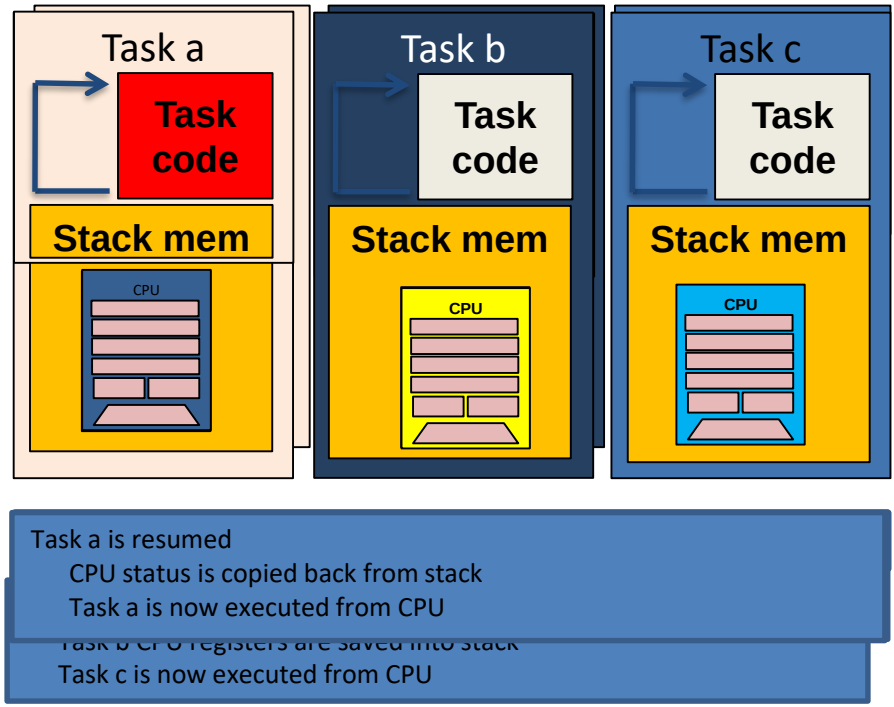
• Assigning task priorities is not a trivial undertaking because of the complex nature of real-time systems.

• In most systems, not all tasks are considered critical.

• Non-critical tasks should obviously be given low priorities.
• Most real-time systems have a combination of SOFT and HARD requirements.
• In a SOFT real-time system, tasks are performed by the system as quickly as possible, but they don't have to finish by specific times.
• In HARD real-time systems, tasks have to be performed not only correctly but on time.

RTOS Context Switching

- The context switching is some time so-called task switching
- When a multitasking kernel decides to run a different task, it simply saves the current task's *context (CPU registers) in* its stack.
- Once this operation is performed, the new task‘s context is restored from its storage area and then resumes execution of the new task's code.
- This process is called a *context switch or a task switch.*
- *Context switching adds overhead to the application. The more registers a CPU has, the* higher the overhead. The time required to perform a context switch is determined by how many registers have to be saved and restored by the CPU.



RTOS Scheduler

- The scheduler, also called the *dispatcher, is the part of the kernel responsible for determining which task will run next.*
- Most real-time kernels are priority based. Each task is assigned a priority based on its importance. The priority for each task is application specific.
- In a priority-based kernel, control of the CPU will always be given to the highest priority task ready-to-run.
- When the highest-priority task gets the CPU, however, is determined by the type of kernel used.
- There are two types of priority-based kernels: *non-preemptive and preemptive.*

RTOS Scheduler (cont.)

- In typical designs, a task has 4 states:
- 1) running,
- 2) ready,
- 3) waiting,
- 4) dormant

The *DORMANT state corresponds to a task which resides in program space* but has not been made available to RTOS. A task is made available to RTOS by calling its Create function.
When a task is created, it is made *READY to run. Tasks may be* created before multitasking starts or dynamically by a running task. A task can return itself or another task to the dormant state by calling Delete function.

- Most tasks are waiting, most of the time. Only one task per CPU is running. In simpler systems, the ready list is usually short, two or three tasks at most.

- The real key is designing the scheduler. Usually the data structure of the ready list in the scheduler is designed to minimize the worst-case length of time spent in the scheduler's critical section, during which preemption is inhibited, and, in some cases, all interrupts are disabled. But, the choice of data structure depends also on the maximum number of tasks that can be on the ready list.

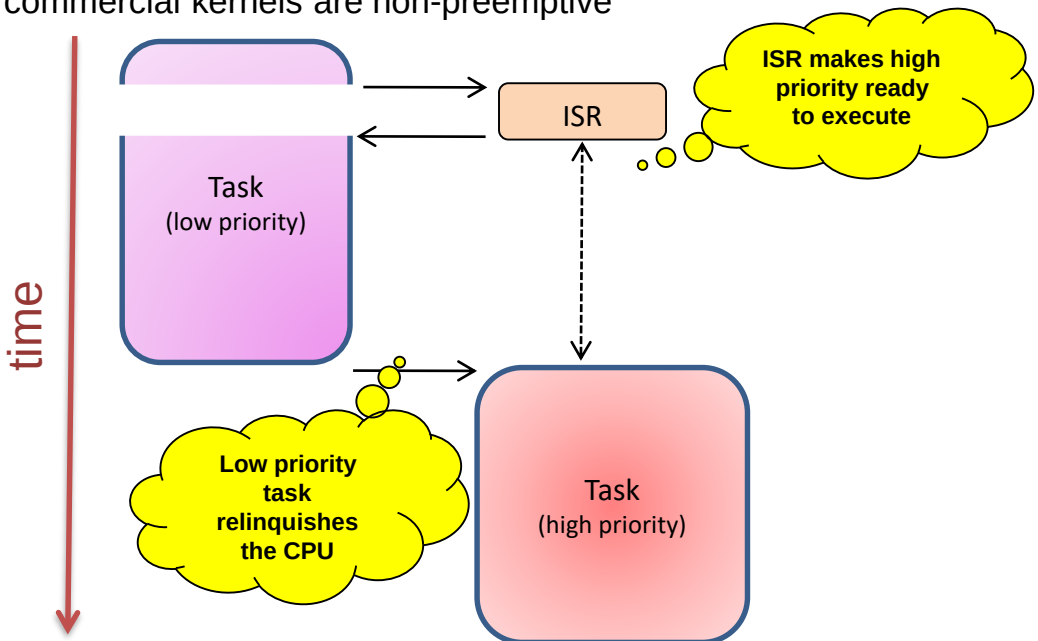
- The critical response time, sometimes called the fly back time, is the time it takes to queue a new ready task and restore the state of the highest priority task. In a well-designed RTOS, readying a new task will take 3-20 instructions per ready queue entry, and restoration of the highest-priority ready task will take 5-30 instructions.

RTOS Scheduler (cont.)

- **Round-robin (Cooperative Scheduling)** is one of the simplest scheduling algorithms for processes in an operating system, which assigns time slices to each process in equal portions and in circular order, handling all processes without priority. Round-robin scheduling is both simple and easy to implement, and starvation-free.
- When two or more tasks have the same priority, the kernel will allow one task to run for a predetermined amount of time, called a *quantum, and then selects another task. This is also called time slicing. The kernel gives control to the* next task in line if:
- 1) the current task doesn't have any work to do during its time slice or
- 2) the current task completes before the end of its time slice.

RTOS Non-Preemptive Kernel

- Non-preemptive kernels require that each task does something to explicitly give up control of the CPU. To maintain the illusion of concurrency, this process must be done frequently. Non-preemptive scheduling is also called *cooperative multitasking; tasks cooperate with each other to share the CPU.*
- A non-preemptive kernel allows each task to run until it voluntarily gives up control of the CPU. An interrupt will preempt a task. Upon completion of the ISR, the ISR will return to the interrupted task.
- The new higher priority task will gain control of the CPU only when the current task gives up the CPU.
- One of the advantages of a non-preemptive kernel is that interrupt latency is typically low. At the task level, non-preemptive kernels can also use non-reentrant functions (see later).
- The most important drawback of a non-preemptive kernel is responsiveness. A higher priority task that has been made ready to run may have to wait a long time to run, because the current task must give up the CPU when it is ready to do so. As with background execution in foreground/background systems, task-level response time in a non-preemptive kernel is non-deterministic; you never really know when the highest priority task will get control of the CPU. It is up to your application to relinquish control of the CPU.
- Task-level response is much better than with a foreground/background system but is still non-deterministic. Very few commercial kernels are non-preemptive



RTOS Preemptive Kernel

- A preemptive kernel is used when system responsiveness is important. Because of this most commercial real-time kernels are preemptive.
- The highest priority task ready to run is always given control of the CPU.
- When a task makes a higher priority task ready to run, the current task is preempted (suspended) and the higher priority task is immediately given control of the CPU. If an ISR makes a higher priority task ready, when the ISR completes, the interrupted task is suspended and the new higher priority task is resumed.

•Upon completion of an ISR, the kernel will resume execution to the highest priority task ready to run (not the interrupted task).
• Task-level response is optimum and deterministic.

•Application code using a preemptive kernel should not make use of non-reentrant functions unless exclusive access to these functions is ensured through the use of mutual exclusion semaphores, because both a low priority task and a high priority task can make use of a common function. Corruption of data may occur if the higher priority task preempts a lower priority task that is making use of the function.
• A reentrant function is a function that can be used by more than one task without fear of data corruption. A reentrant function can be interrupted at any time and resumed at a later time without loss of data. Reentrant functions either use local variables (i.e., CPU registers or variables on the stack) or protect data when global variables are used.

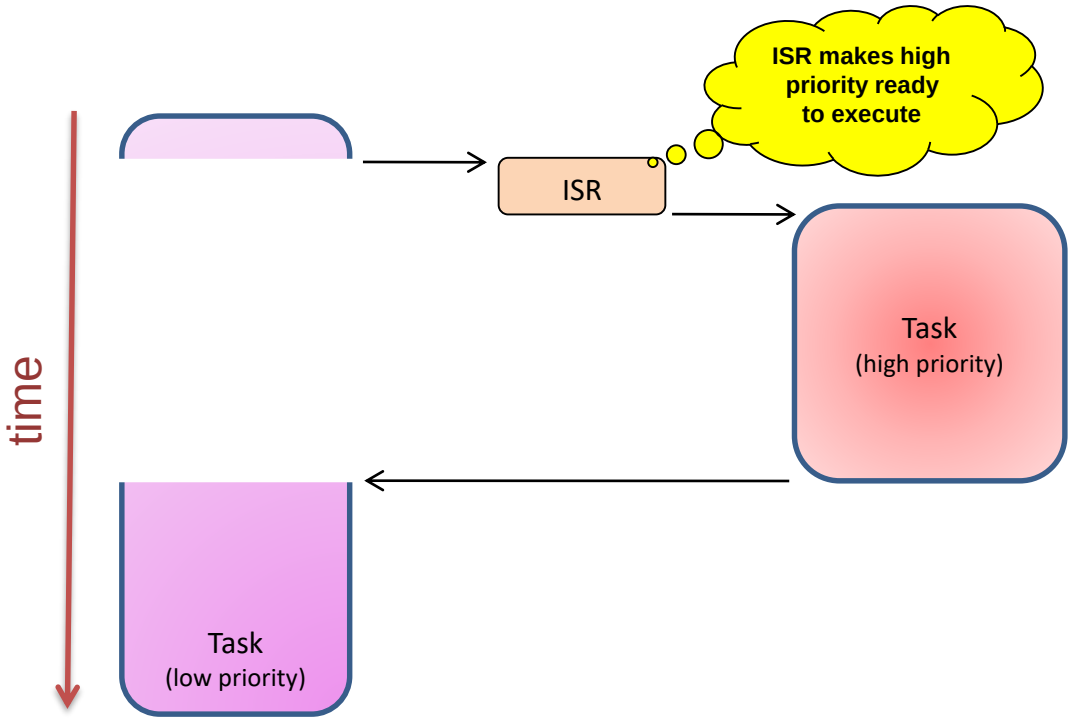


Table of contents

Inter-task Communication and Resource Sharing

- Multitasking systems must manage sharing data and hardware resources among multiple tasks. The easiest way for tasks to communicate with each other is through shared data structures.
- It is usually "unsafe" for two tasks to access the same specific data or hardware resource simultaneously. ("Unsafe" means the results are inconsistent or unpredictable, particularly when one task is in the midst of changing a data collection. The view by another task is best done either before any change begins, or after changes are completely finished.)

- There are few common approaches to resolve this problem:
  - Semaphores
  - Messages
  - Message Queues
  - Message Mailbox

Semaphores

A semaphore is a protocol mechanism offered by most multitasking kernels. Semaphores are used to:

a) control access to a shared resource (mutual exclusion);
b) signal the occurrence of an event;
c) allow two tasks to synchronize their activities.

A semaphore is a key that your code acquires in order to continue execution.
If the semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner.

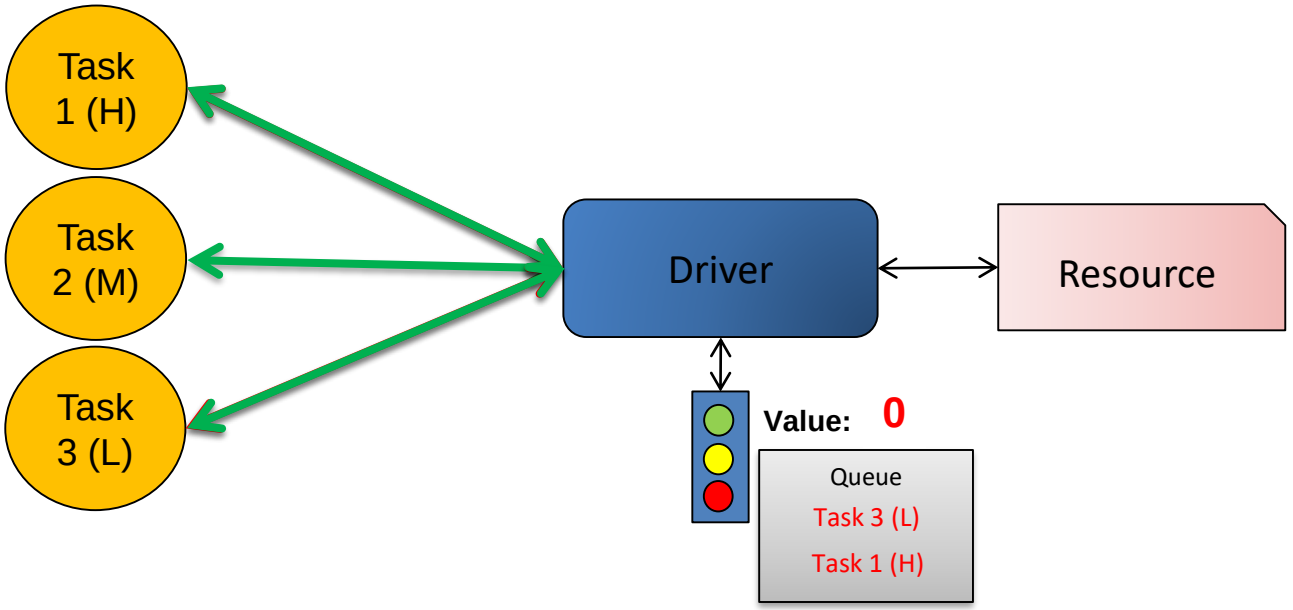
There are two types of semaphores:

- Binary semaphores (mutex) can only take two values: 0 (locked) or 1 (unlocked)
- Counting semaphore allows values between 0 and 255, 65535 or 4294967295, depending on whether the semaphore mechanism is implemented using 8, 16 or 32 bits, respectively.

Along with the semaphore's value, the kernel also needs to keep track of tasks waiting for the semaphore's availability.
Problems with semaphore based designs are well known: priority inversion and deadlocks.

Semaphore example

- The initial value of the semaphore must be provided when the semaphore is initialized. The waiting list of tasks is always initially empty.



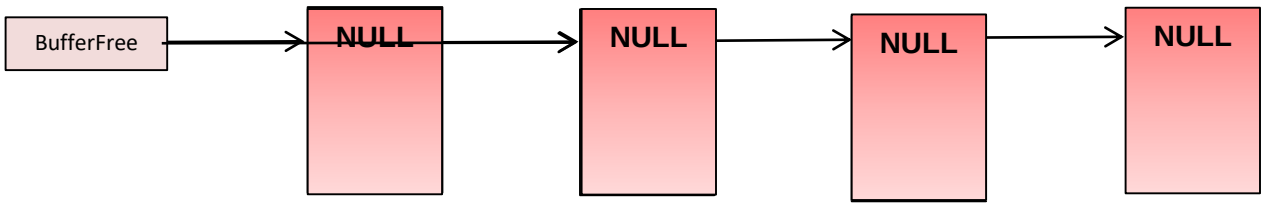
Task 3 releases the resource. Because queue is empty value is incremented ng list. The

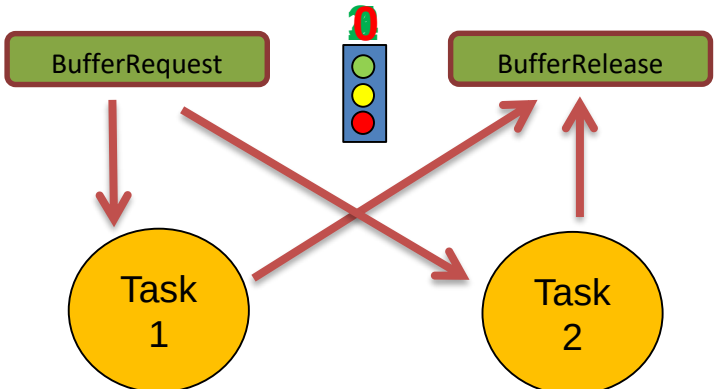
Counting semaphore example

A task will obtain a buffer from the resource pool by calling BufferRequest()
A task will release a buffer back to the pool, after using it, by calling BufferRelease()
A task would obtain a buffer from the resource pool by calling BufferRequest()
A task will release a buffer back to the pool, after using it, by calling BufferRelease()
management

```
BUFFER *BufferRequest(void)
{
    BUFFER *ptr;
    Get a semaphore;
    Disable interrupts;
    ptr = BufferFree;
    BufferFree = ptr->BufferNext;
    Enable interrupts;
    return (ptr);
}
```

```
void BufferRelease(BUF *ptr)
{
    Disable interrupts;
    ptr->BufferNext = BufferFreeList;
    BufferFreeList = ptr;
    Enable interrupts;
    Release semaphore;
}
```





Semaphores are often overused

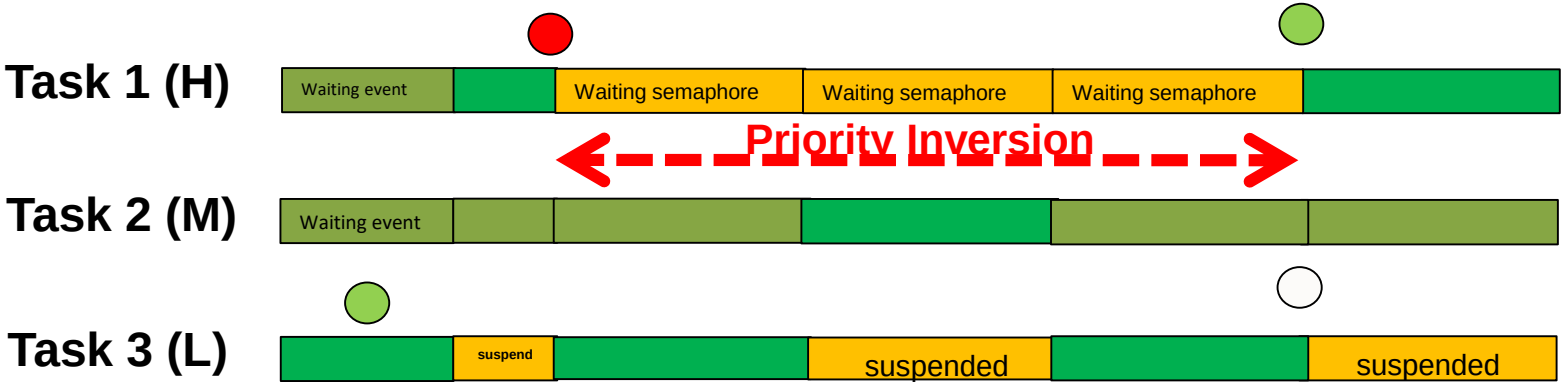
The use of a semaphore to access a simple shared variable is overkill in most situations.

The overhead involved in acquiring and releasing the semaphore can consume valuable time. You can do the job just as efficiently by disabling and enabling interrupts

Priority Inversion

Priority inversion is a problem in real-time systems and occurs mostly when you use a real-time kernel.

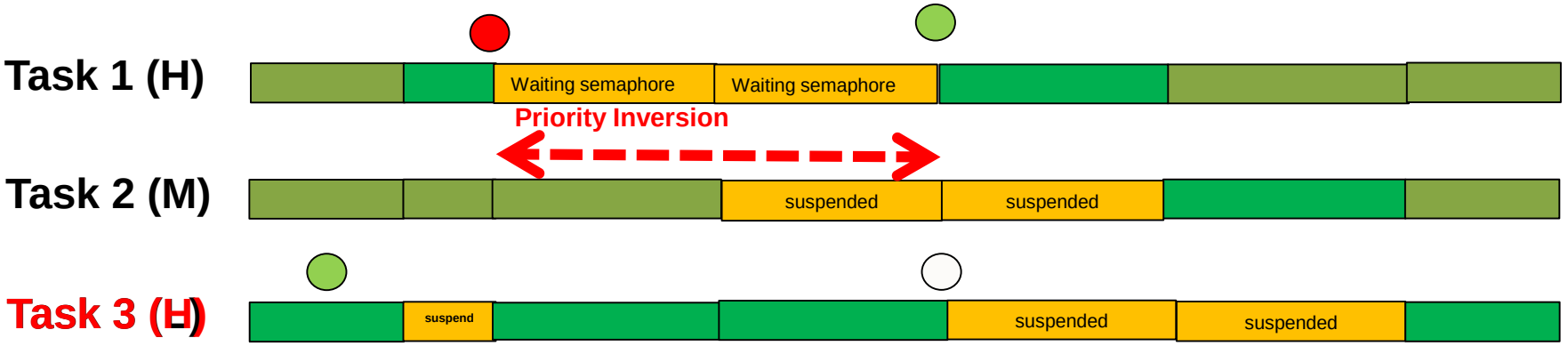
In **priority inversion**, a high priority task waits because a low priority task has a semaphore. A typical solution is to have the task that has a semaphore run at (inherit) the priority of the highest waiting task. But this simplistic approach fails when there are multiple levels of waiting (A waits for a binary semaphore locked by B, which waits for a binary semaphore locked by C). Handling multiple levels of inheritance without introducing instability in cycles is not straightforward.



Because Task 1 has the semaphore the Kernel switch back to Task 3

Priority Inversion

You can correct this situation by raising the priority of Task 3 (above the priority of the other tasks competing for the resource) for the time Task3 is accessing the resource and restore the original priority level when the task is finished.



Task 3 ends and Task 2 is now executed. When Task 2 ends Task 3 is executed

Task 3 priority is raised at the priority of Task 1

Deadlock

A deadlock, also called a *deadly embrace, is a situation in which two tasks are each unknowingly waiting for resources* held by each other. If task T1 has exclusive access to resource R1 and task T2 has exclusive access to resource R2, then if T1 needs exclusive access to R2 and T2 needs exclusive access to R1, neither task can continue.

The simplest way to avoid a deadlock is for tasks to:

a) acquire all resources before proceeding,
b) acquire the resources in the same order, and
c) release the resources in the reverse order.

Most kernels allow you to specify a timeout when acquiring a semaphore. This feature allows a deadlock to be broken.

If the semaphore is not available within a certain amount of time, the task requesting the resource will resume execution.

Some form of error code must be returned to the task to notify it that a timeout has occurred. A return error code prevents the task from thinking it has obtained to the resource.

Deadlocks generally occur in large multitasking
systems and are not generally encountered in embedded systems.

Task Synchronization

A task can be synchronized with an ISR, or another task when no data is being exchanged, by using a semaphore

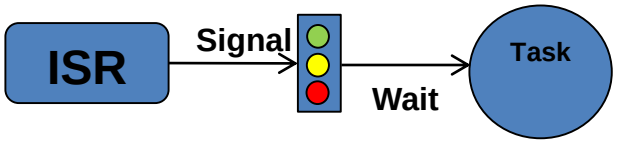
Note that, in this case, the semaphore is drawn as a flag, to indicate that it is used to signal the occurrence of an event (rather than to ensure mutual exclusion, in which case it would be drawn as a key).

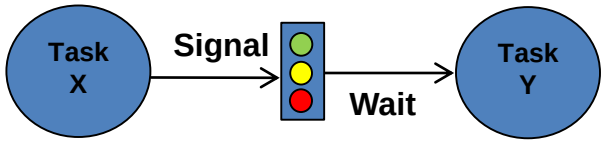
When used as a synchronization mechanism, the semaphore is initialized to 0.

A task initiates an I/O operation and then waits for the semaphore.

When the I/O operation is complete, an ISR (or another task) signals the semaphore and the task is resumed.

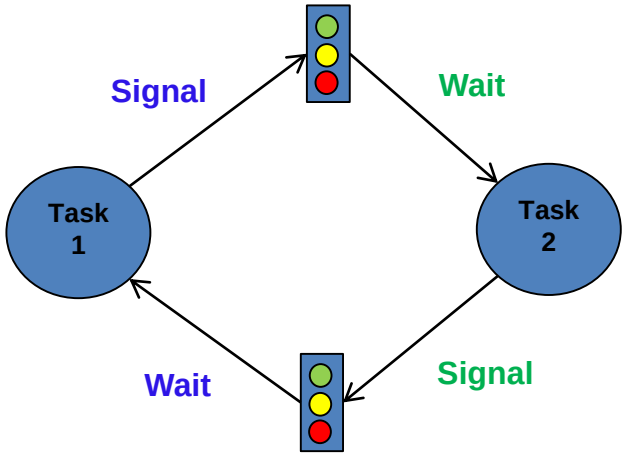
If the kernel supports counting semaphores, the semaphore would accumulate events that have not yet been processed.





Task Synchronization

Two tasks can synchronize their activities by using two semaphores

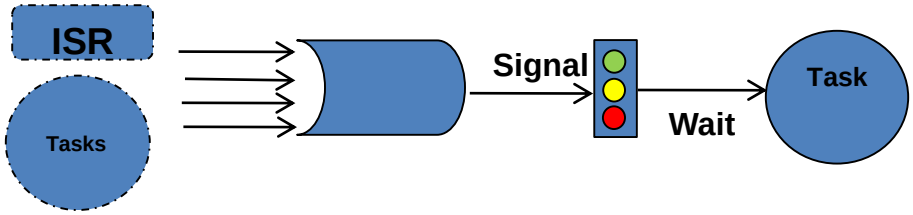



```
Task1()
{
        while  (true)
        {
                operations;
              signal task #2;
                wait for signal from task #2;
                continue operation;
        }
}
```

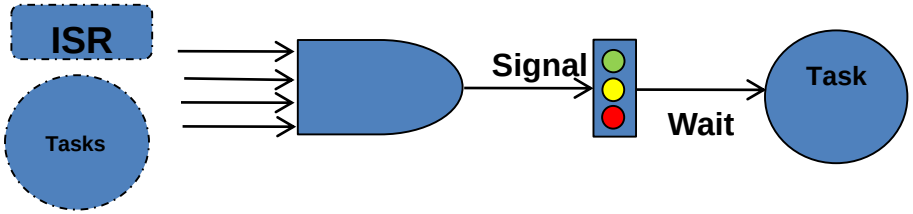
```
Task2()
{
        while  (true)
        {
                operations;
              signal task #1;
                wait for signal from task #1;
                continue operation;
        }
}
```

Event Flags

Event flags are used when a task needs to synchronize with the occurrence of multiple events.
The task can be synchronized when any of the events have occurred.
This is called *disjunctive synchronization (logical OR).*



A task can also be synchronized when all events have occurred. This is called *conjunctive synchronization (logical AND).*



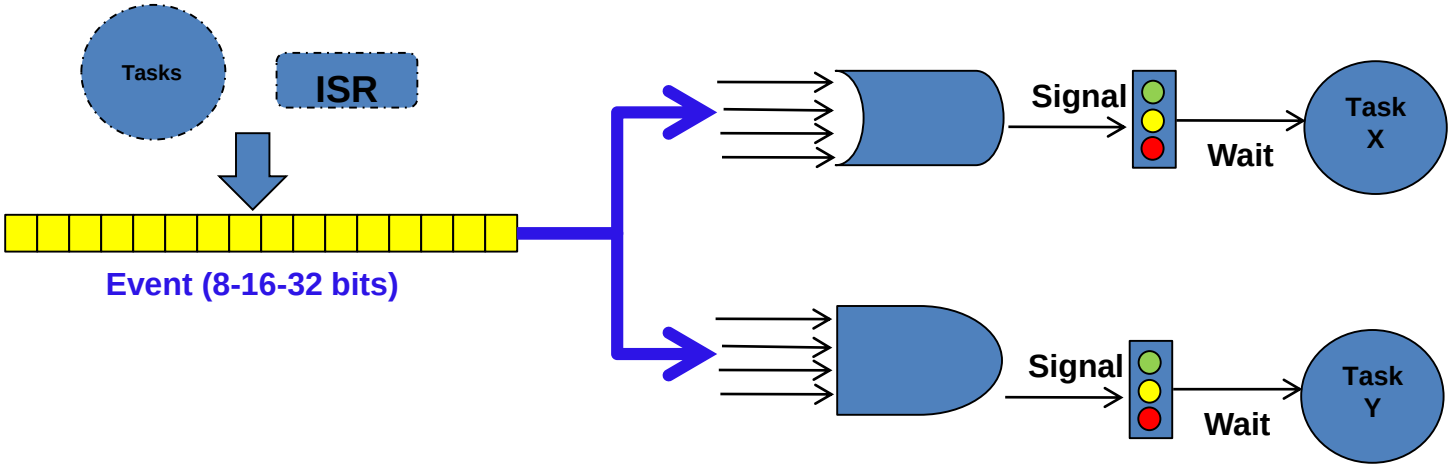
Kernels supporting event flags offer services to SET event flags, CLEAR event flags, and WAIT for event flags (conjunctively or disjunctively).

Event Flags

Depending on the kernel, a group consists of 8, 16 or 32 events (mostly 32-bits, though).

Tasks and ISRs can set or clear any event in a group.

A task is resumed when all the events it requires are satisfied. The evaluation of which task will be resumed is performed when a new set of events occurs.



Message Mailboxes

Messages can be sent to a task through kernel services.

A Message Mailbox, also called a message exchange, is typically a pointer size variable. Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into this mailbox.

Similarly, one or more tasks can receive messages through a service provided by the kernel.
Both the sending task and receiving task will agree as to what the pointer is actually pointing to.

A waiting list is associated with each mailbox in case more than one task desires to receive messages through the mailbox.

A task desiring to receive a message from an empty mailbox will be suspended and placed on the waiting list until a message is received.

Typically, the kernel will allow the task waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting task is made ready-to-run and an error code (indicating that a timeout has occurred) is returned to it.

When a message is deposited into the mailbox, either the highest priority task waiting for the message is given the message (called *priority-based) or the first task to request a* message is given the message (called *First-In-First-Out, or FIFO).*

Message Mailboxes

Kernel services are typically provided to:

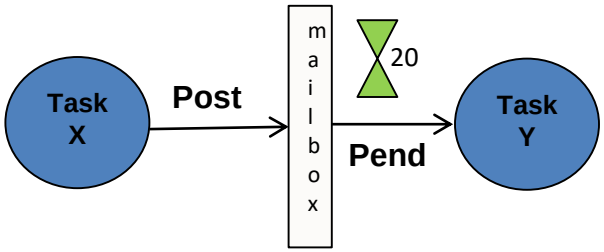
a) Initialize the contents of a mailbox. The mailbox may or may not initially contain a message.

b) Deposit a message into the mailbox (POST).

c) Wait for a message to be deposited into the mailbox (PEND).

d) Get a message from a mailbox, if one is present, but not suspend the caller if the mailbox is empty (ACCEPT). If the mailbox contains a message, the message is extracted from the mailbox. A return code is used to notify the caller about the outcome of the call.

Message mailboxes can also be used to simulate binary semaphores. A message in the mailbox indicates that the resource is available while an empty mailbox indicates that the resource is already in use by another task.



Message Queues

A message queue is used to send one or more messages to a task. A message queue is basically an array of mailboxes.

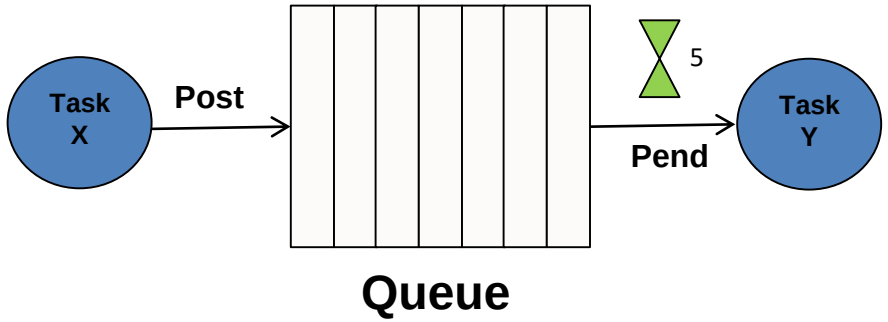
Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into a message queue.

Similarly, one or more tasks can receive messages through a service provided by the kernel. Both the sending task and receiving task will agree as to what the pointer is actually pointing to.

Generally, the first message inserted in the queue will be the first message extracted from the queue (FIFO).

As with the mailbox, a waiting list is associated with each message queue in case more than one task is to receive messages through the queue.
A task desiring to receive a message from an empty queue will be suspended and placed on the waiting list until a message is received. Typically, the kernel will allow the task waiting for a message to specify a timeout.
If a message is not received before the timeout expires, the requesting task is made ready-to-run and an error code (indicating a timeout occurred) is returned to it.

When a message is deposited into the queue, either the highest priority task or the first task to wait for the message will be given the message.



Interrupt responses

Interrupt Latency

Maximum amount of time interrupts are disabled

+

Time to start executing the first instruction in the ISR

***Interrupt Response for* foreground/background system and Non-preemptive kernel**

Interrupt latency

+

Time to save the CPU's context

***Interrupt Response for* Preemptive kernel**

Interrupt latency

+

Time to save the CPU's context

+

Execution time of the kernel ISR entry function

Interrupt Recovery

***Interrupt Recovery for* foreground/background system and Non-preemptive kernel**

Time to restore the CPU's context

+

Time to execute the return from interrupt instruction

***Interrupt Recovery for* Preemptive kernel**

Time to determine if a higher priority task is ready

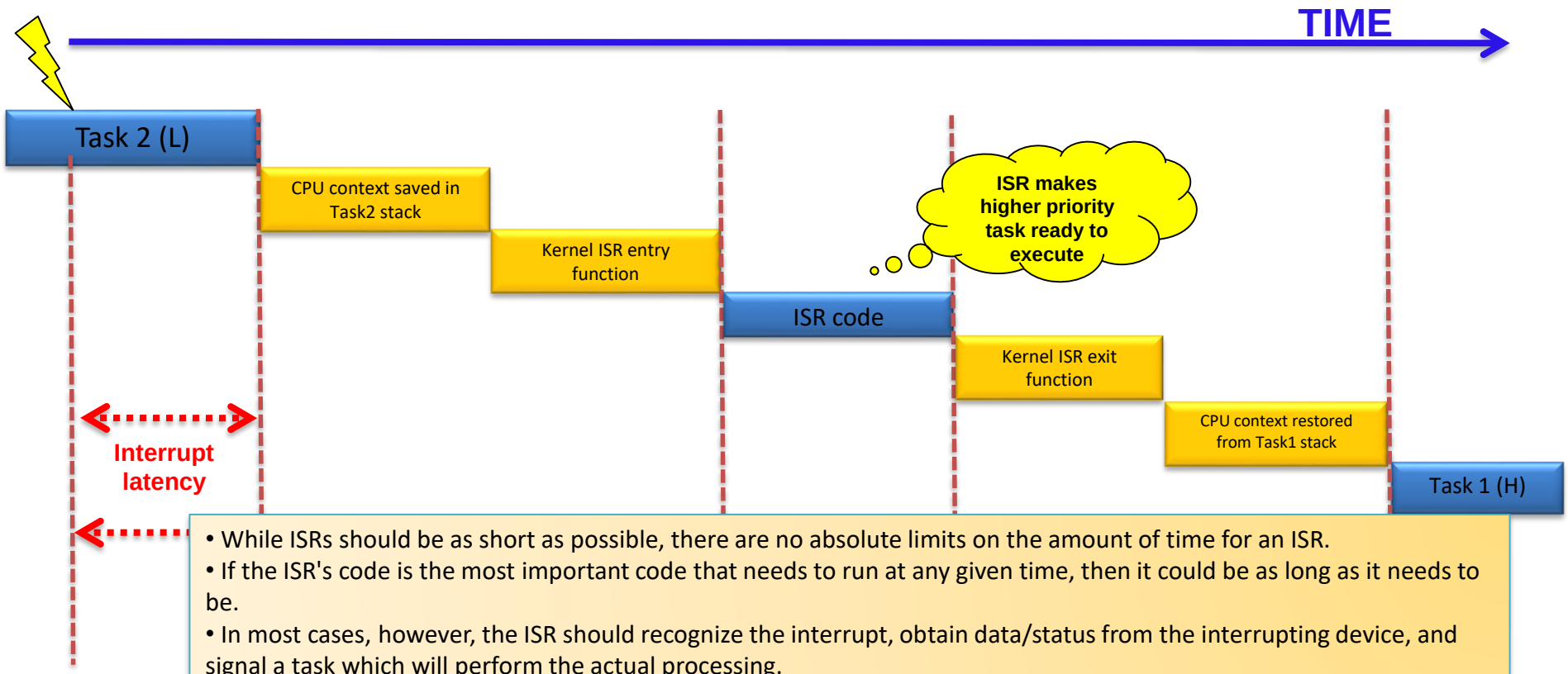
+

Time to restore the CPU's context of the highest priority task

+

Time to execute the return from interrupt instruction

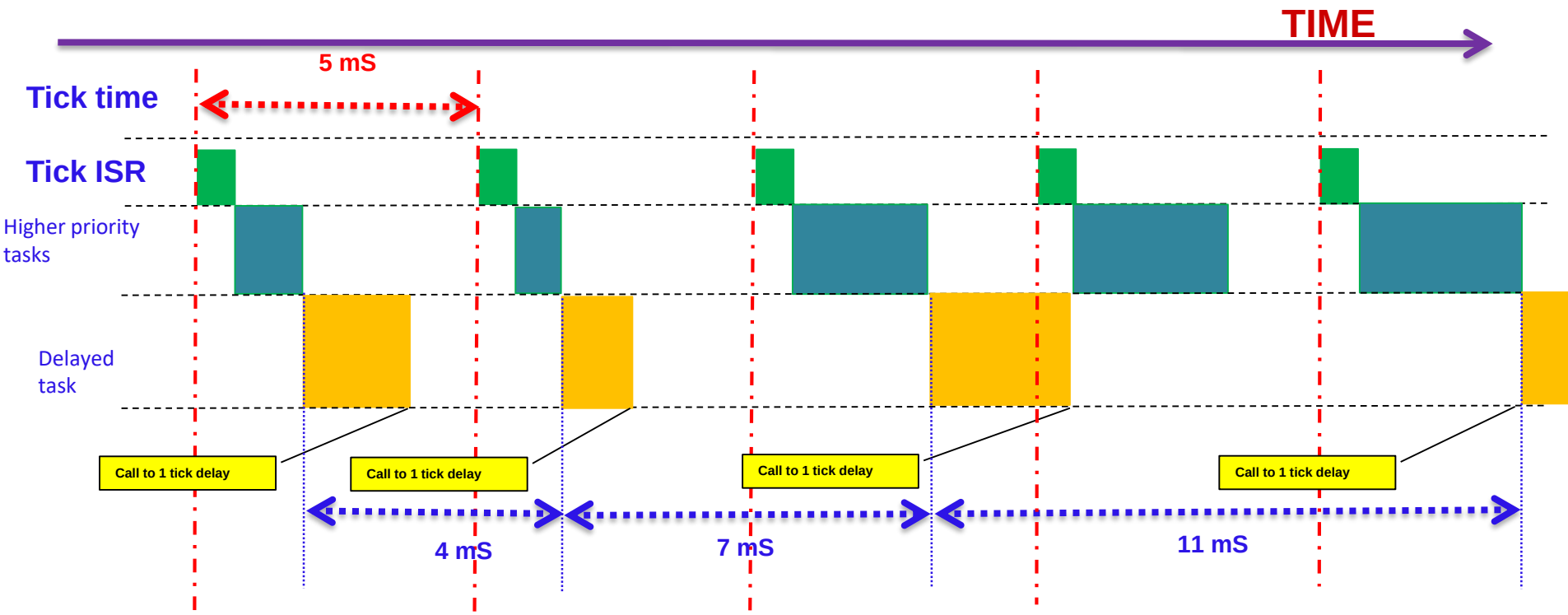
Interrupt latency, response, and recovery on preemptive kernel



- While ISRs should be as short as possible, there are no absolute limits on the amount of time for an ISR.
- If the ISR's code is the most important code that needs to run at any given time, then it could be as long as it needs to be.
- In most cases, however, the ISR should recognize the interrupt, obtain data/status from the interrupting device, and signal a task which will perform the actual processing.
- You should also consider whether the overhead involved in signaling a task is more than the processing of the interrupt. Signaling a task from an ISR (i.e. through a semaphore, a mailbox, or a queue) requires some processing time. If processing of your interrupt requires less than the time required to signal a task, you should consider processing the interrupt in the ISR itself and possibly enable interrupts to allow higher priority interrupts to be recognized and serviced.

Clock Tick - the system's heartbeat

A *clock tick is a special interrupt that occurs periodically. The* time between interrupts is application specific and is generally between 1 and 20 mS.
The clock tick interrupt allows a kernel to delay tasks for an integral number of clock ticks and to provide timeouts when tasks are waiting for events to occur.
The faster the tick rate, the higher the overhead imposed on the system.
All kernels allow tasks to be delayed for a certain number of clock ticks.
The resolution of delayed tasks is 1 clock tick, however, this does not mean that its accuracy is 1 clock tick.
This will thus cause the execution of the task to *jitter.*



Summary

- Real-time application is targeted on application with deadline must be predicted
- Real-time Operating System – RTOS is an Embedded OS designed for Real-time application usage
- There are few concepts related to RTOS: Tasks/Processes, non-preemptive and preemptive kernel
- There are multiple ways to synchronize and inter-process communication in RTOS
  - Semaphore
  - Event
  - Message Queue
  - Message Mailbox

Question & Answer

Thanks for your attention !

Copyright

- This course including **Lecture Presentations**, **Quiz**, **Mock Project**, **Syllabus**, **Assignments**, **Answers** are copyright by FPT Software Corporation.
- This course also uses some information from external sources and non-confidential training document from Freescale, those materials comply with the original source licenses.