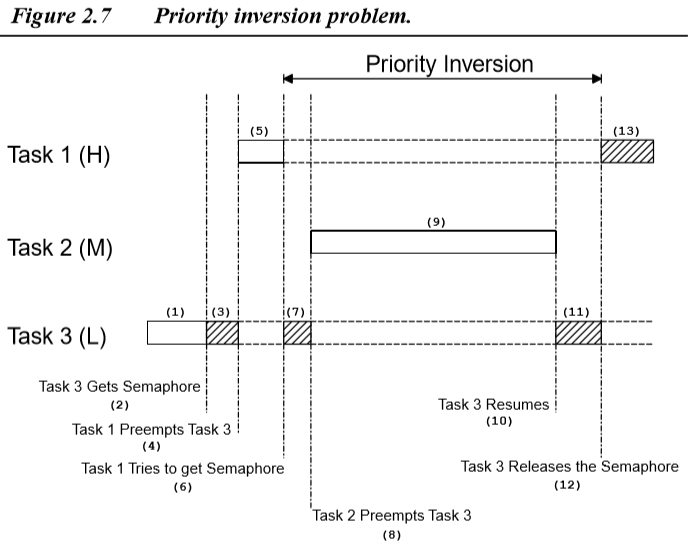
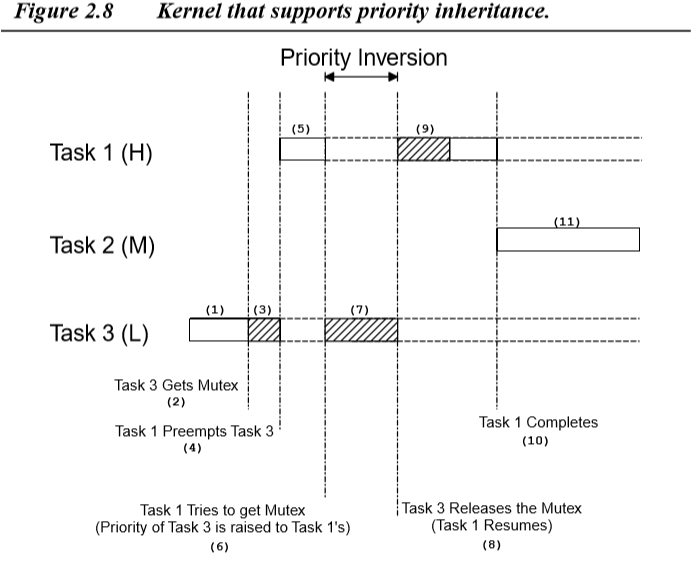
**CHAPTER 2**

* **Foreground/Background Systems (Super-loops): use to halt the processor and perform all of the processing in ISRs**
* Background (task level): An application consists of an infinite loop that calls modules (function) to perform the desired operations.
* Foreground (interrupt level): ISRs handle asynchronous events.
* Critical operations must be performed by ISRs to ensure that they are dealt with in a timely fashion => ISRs have tendency to take longer than normal.
* Task-level response: information for background module that an ISR makes available is not processed until the background routine gets its turn to execute. The longer it takes the background to execute, the longer the task-level response. The time of successive passes through a portion of the loop is nondeterministic.
* **Critical Sections of Code (Critical Region):** code that needs to be treated indivisibly. It must not be interrupted after the section of code starts executing. To make sure that, interrupts are disabled before the critical code is executed and enable after the critical code is finished.
* **Resources:** A resource is any entity used by a task. It can be I/O device, such as a display, a variable, a structure or an array.
* **Shared Resources:** A shared resource is a resource that can be used by more than 1 task. Each task should gain exclusive access to the shared resource to prevent data corruption (mutual exclusion).
* **Multitasking:** the process of scheduling and switching the CPU between several tasks. Multitasking is like foreground/background with multiple backgrounds.
* Maximizes the use of the CPU and provides for modular construction of applications.
* Allows the application programmer to mange complexity inherent in real-term applications
* **Task (thread):** 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 responsible for a portion of the problem. Each task is assigned a priority, its own set of CPU registers, and its own stack area.
* Task typically is an infinite loop that can be in any five states: dormant, ready, running, waiting or interrupted. A task is ready when it has control of the CPU, is waiting when it requires the occurrence of an event, is dormant when it resides in memory but has not been made available, is ready when it can execute but its priority is less than the currently task, is interrupted when interrupt has occurred and the CPU is in the process of servicing the interrupt
* **Context Switches (Task Switches):** When a multitasking kernel decides to run a different task, it saves the current task’s context (CPU register) in the current task’s context storage area. After 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.
* Context switching adds overhead to the application. The more registers a CPU has, the higher the overhead.
* **Kernel:** is a part of a multitasking system responsible for management of tasks and communication between tasks. The fundamental service provided by the kernel is context switching
* Allowing the application to be divided into multiple tasks that the kernel manages.
* Requires extra ROM and additional RAM for the data structures and each task require its own stack space
* **Scheduler (dispatcher):** part of the kernel responsible for the determining which task run next.
* **Non-Preemptive Kernels:** Allows each task to run until it voluntarily gives up control of the CPU. An interrupt preempts a task. Upon completion of the ISR, the ISR returns to the interrupted task. Task-level response is much better than with a foreground/background system but is still nondeterministic.
* The advantage of a non-preemptive kernel is that interrupt latency is typically low. At task level, non-preemptive kernels can also use non-reentrant functions.
* Another advantage of non-preemptive is the lesser need to guard shared data through the use of semaphores. Each task owns the CPU, and you don’t have to fear that a task will be preempted.
* The most important drawback of a non-preemptive is responsiveness. A higher priority task has been made ready to run might have to wait a long time to run because the current task must give up the CPU when it is ready to do so. You never really know when the highest priority task will get control of the CPU.
* **Preemptive Kernel:** always executes the highest priority task that is ready to run. An interrupt preempts a task. Upon completion of an ISR, the kernel resumes execution of the highest priority task ready to run. Task-level response is optimum and deterministic. (uC/OS-II is a preemptive kernel).
* With a preemptive kernel, execution of the highest priority task is deterministic: you can determine when it will get control of the CPU. Task-level is also minimized.
* Application code using a preemptive kernel should not use non-reentrant functions unless exclusive access to these functions is ensured through the use of mutual exclusion semaphores. Corruption of data can occur if the higher priority task preempts a lower priority task that is using the function.
* **Reentrant Functions:** can be used by more than 1 task without fear of data corruption, can be interrupted at any time and resumed at a later time without loss of data.
* Use local variables or protect data when global variables are used.
* **Round-Robin Scheduling (time slicing):** when 2 or more tasks have the same priority, the kernel allows 1 task to run for a predetermined time (quantum) and then selects another task. uC/OS-II does not currently support round-robin scheduling. Each task must have a unique priority in your application.
* The kernel gives control to the next task in line if: the current task has no work to do during its time slice of; the current task completes before the end of its time slice or the time slice ends.
* **Task Priorities:** A priority is assigned to each task. The more important the task, the higher the priority given to it.
* **Static Priorities:** Task priorities are static when the priority of each task does not change during the application’s execution; each task is given a fixed priority at compile time. All tasks and their timing constraints are known at compile time in a system where priorities are static.
* **Dynamic Priorities:** Task priorities are dynamic ì the priority of tasks can be changed during the application’s execution; each task can change its priority at run time. uC/OS provide this feature to avoid priority inversions.
* **Priority Inversions:** a problem in real-time systems and occurs mostly when user use a real-time kernel. A multitasking kernel should allow task priorities to change dynamically to help prevent priority inversion. 
* **Priority inheritance:** a kernel that changes the priority of a task automatically, avoid priority inversion. uC/OS-II provides this feature.



* **Assigning Task Priorities:** not a trivial undertaking because of the complex nature of real-time systems. Most real-time systems have a combination of soft and hard requirements. In a soft real-time system, tasks are performed as quickly as possible, but don’t have to finish in specific times. In hard real-time ones, tasks have to be performed not only correctly but also on time.
* **Rate Monotonic Scheduling (RMS):** established to assign task priorities based on how often tasks execute: tasks with the highest rate of execution are given the highest priority
* **Mutual Exclusion:** The easiest way for tasks to communicate with each other is through shared data structures. The most common methods of obtaining exclusive access to shared resources are: disabling interrupts, performing test and set operations, disabling scheduling, using semaphores.
* **disabling interrupts:** the easiest way to gain exclusive access to a shared resource. uC/OS-II uses this technique to access internal variables and data structures. This method is the only way that a task can share variables or data structures with an ISR.
* **test-and-set operations:** 2 functions access to a resource must check a global variable if it is 0. To prevent the other function from accessing the resource, however, the first function that gets the resource sets the variable to 1.
* **Disabling and enabling the scheduler:** if task is not sharing variables or data structures with ISR, you can disable and enable scheduling. While scheduler is locked, interrupts are enabled, and if an interrupt occurs while in the critical section, the ISR is executed immediately
* **Semaphores:** a protocol mechanism offered by most multitasking kernels. Semaphores are used to: control access to a shared resource; signal the occurrence of an event; allow 2 tasks to synchronize their activities. Only 3 operations can be performed on a semaphore: *initialize (create), wait (pend), signal (post).* The initial value of the semaphore must be provided when the semaphore is initialized. The waiting list of tasks is always initially empty. Semaphore are especially useful when tasks share I/O devices. A counting semaphore is used when a resource can be used by more than 1 task at the same time.
* **Deadlock (Deadly Embrace):** a situation in which 2 tasks are each unknowingly waiting for resources held by the other. The simplest way to avoid a deadlock is for tasks to: acquire all resources before proceeding; acquire the resources in the same order; release the resources in the reverse order.
* **Synchronization:** a task can be synchronized with an ISR by using a semaphore. When used as a synchronization mechanism, the semaphore is initialized to 0. Using a semaphore for this type of synchronization is called a unilateral rendezvous.
* If the kernel supports counting semaphores, the semaphore accumulates events that have not yet been processed
* **Bilateral rendezvous:** 2 tasks can synchronize their activities by using 2 semaphores. A bilateral rendezvous is similar to a unilateral rendezvous, except both tasks must synchronize with 1 another before proceeding. A bilateral rendezvous cannot be performed between a task and a ISR because ISR cannot wait on a semaphore.
* **Event Flags:** are used when a task needs to synchronize with the occurrence of multiple events
* **Disjunctive synchronization (logical OR):** the task can be synchronized when any of the events have occurred.
* **Conjunctive synchronization (logical AND):** a task can be synchronized when all events have occurred.
* Common events can be used to signal multiple tasks and typically grouped (8, 16, 32 events). A task is resumed when all the events it requires are satisfied.
* **Inter-task Communication:** is the information transfer from a task or an ISR to another task. There are 2 ways:
* Through global data: Task or ISR must have exclusive access to global variable. For ISR, the only way is to disable interrupt
* By sending message: Task can only communicate information to an ISR by using global variables. Task is not aware when a global variable is changed by an ISR so we have to use a message mailbox or a message queue.
* **Message Mailboxes (message exchange):** istypically a pointer-size variable. A task or an ISR can deposit a message into this mailbox, and one or more tasks can receive it through a kernel service. A waiting list is associated with each mailbox in case more than one task wants to receive messages through the mailbox. The kernel allows the task waiting for a message to specify a timeout. When a message is deposited into the mailbox, either the highest priority task waiting for the message is given the message (priority-based), or the first task to request a message is given the message.
* Mailbox services of Kernels: Initialize the contents of a mailbox; POST, PEND, and ACCEPT
* **Message queues:** is used to send one or more messages to a task. A message queue is basically an array of mailboxes. A task or an ISR can deposit a message (the pointer) into a message queue, one or more tasks can receive messages through a service provided by the kernel. A waiting list is associated with each message queue, in case more than one task is to receive messages through the queue. The kernel allows the task waiting for a message to specify a timeout. When a message is deposited into the queue, either the highest priority task, or the first task to wait for the message is given the message.
* Message queue service of kernels: Initialize the queue; POST, PEND, and ACCEPT
* **Interrupts:** is a hardware mechanism used to inform the CPU that an asynchronous event has occurred. When it is interrupt, CPU jumps into Interrupt Service Routine (ISR). ISR processes the event and returns to:

the background for a foreground/background system; the interrupted task for a non-preemptive kernel; the highest priority task ready to run for a preemptive kernel.

* **Interrupt Latency= Maximum amount of time interrupts are disabled + Time to start executing the first instruction in the ISR**
* All real-time systems disable interrupts to manipulate critical sections of code and reenable interrupts when the critical sections have been executed. The longer interrupts are disabled, the higher the interrupt latency.
* **Interrupt Response:** is defined as the time between the reception of the interrupt and the start of the user code that handles the interrupt.
* The interrupt response time accounts for all of the overhead involved in handling an interrupt.
* for a foreground/background and non-preemptive system: **The response time = Interrupt latency + Time to save the CPU’s context**
* for a preempt kernel: **The response time = Interrupt latency + Time to save the CPU’s context + Execution time of the kernel ISR entry function**
* **Interrupt Recovery:** the time required for the processor to return to the interrupted code or to a higher priority task, in the case of a preemptive kernel. Interrupt recovery in a foreground/background system simply involves restoring the processor’s context and returning to the interrupted task.
* for a foreground/background and non-preemptive system: **Interrupt recovery = Time to restore the CPU’s context + Time to execute the return from interrupt instruction**
* for a preempt kernel: **Interrupt recovery = Time to restore the CPU’s context + Time to execute the return from interrupt instruction + Time to execute the return from interrupt instruction**
* **Interrupt Latency, Response, and Recovery:** For a preemptive kernel, the exit function decides to return either to the interrupted task or to a higher priority task that the ISR has made ready to run
* **ISR Processing Time:** No absolute limits on the amount of time exist for an ISR. Signaling a task from an ISR (i.e., through a semaphore, a mailbox, or a queue) requires some processing time. If processing your interrupt requires less than the time required to signal a task, you should consider processing the interrupt in the ISR itself and possibly enabling interrupts to allow higher priority interrupts to be recognized and serviced.
* **Non-maskable Interrupts:** cannot be disabled, interrupt latency, response, and recovery are minimal. The NMI is generally reserved for drastic measures, such as saving important information during a power down
* **Interrupt Latency = Time to execute longest instruction + Time to start executing the NMI ISR**
* **Interrupt Response = Interrupt latency + Time to save the CPU’s context**
* **Interrupt Recovery = Time to restore the CPU’s context + Time to execute the return from interrupt instruction**
* When servicing an NMI, you cannot use kernel services to signal a task because NMIs cannot be disabled to access critical sections of code. NMIs can be disabled by adding external circuitry. You wouldn't want to disable interrupts to use kernel services, but you could use this feature to pass parameters to and from the ISR and a task.
* **Clock Tick:** a special interrupt that occurs periodically, can be viewed as the system’s heartbeat. The time between interrupts is application specific and is generally between 10 and 200ms. The clock tick interrupt allows a kernel to delay tasks for an integral number of clock ticks
* **Memory Requirements:** a multitasking system requires more code space (ROM) and data space (RAM) than a foreground/background system. The amount of extra ROM depends only on the size of the kernel, and the amount of RAM mostly depends on the number of tasks in your system.
* **The total code space = Application code size + Kernel code size**
* The stack size must account for the task requirements (local variables, function calls, etc.), for maximum interrupt nesting (saved registers, local storage in ISRs, etc.).
* **The total RAM required = Application code requirements + Data space (i.e., RAM) needed by the kernel itself + SUM (task stacks + MAX (ISR nesting))** (the kernel does not support a separate interrupt stack)
* **The total RAM required = Application code requirements + Data space (i.e., RAM) needed by the kernel + SUM (task stacks) + MAX (ISR nesting)** (the kernel supports a separate stack for interrupts)
* **Advantages and Disadvantages of Real-Time Kernels:**

Advantages:

* applications to be designed and expanded easily; functions can be added without requiring major changes to the software
* simplifies the design process by splitting the application code into separate tasks
* all time-critical events are handled as quickly and as efficiently as possible
* make better use of your resources by pro- viding you with valuable services, such as semaphores, mailboxes, queues, time delays, and timeouts

Disadvantages:

* The cost would preclude you from even considering an RTOS.
* The development cost to use an RTOS varies from 70 USD (US Dollars) to well over 30,000 USD. The RTOS vendor might also require royalties on a per-target-system basis.
* **Real-Time Systems Summary:** 