1 Real-Time Systems Concepts

Contents – 1/3

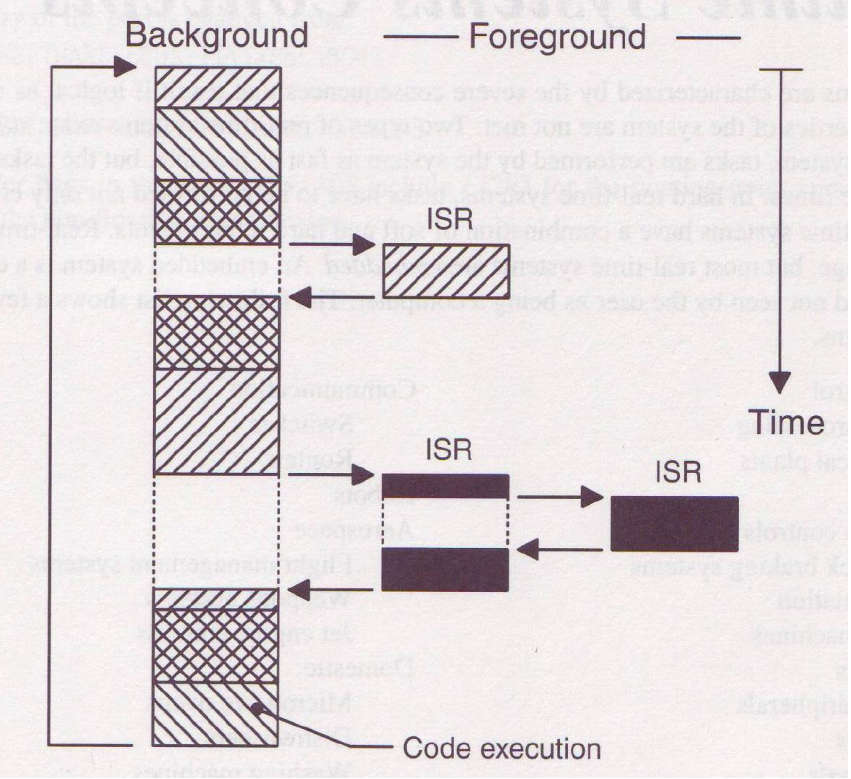
Contents – 2/3

Contents – 3/3

- Event Flags
- Intertask Communication
- Message Mailboxes
- Message Queues
- Interrupts
- Interrupt Latency, Response, and Recovery
- Clock Tick
- Memory Requirements
- Advantages and Disadvantages of Real-Time Kernels

Background/Foreground Systems – 1/2

- Called Super-Loops
- Background (task level) – many functions
- Foreground (interrupt level) – ISR handles asynchronous evens
- Small system of low complexity – most high-volume microcontroller-based applications (microwave oven, telephones, toys …)



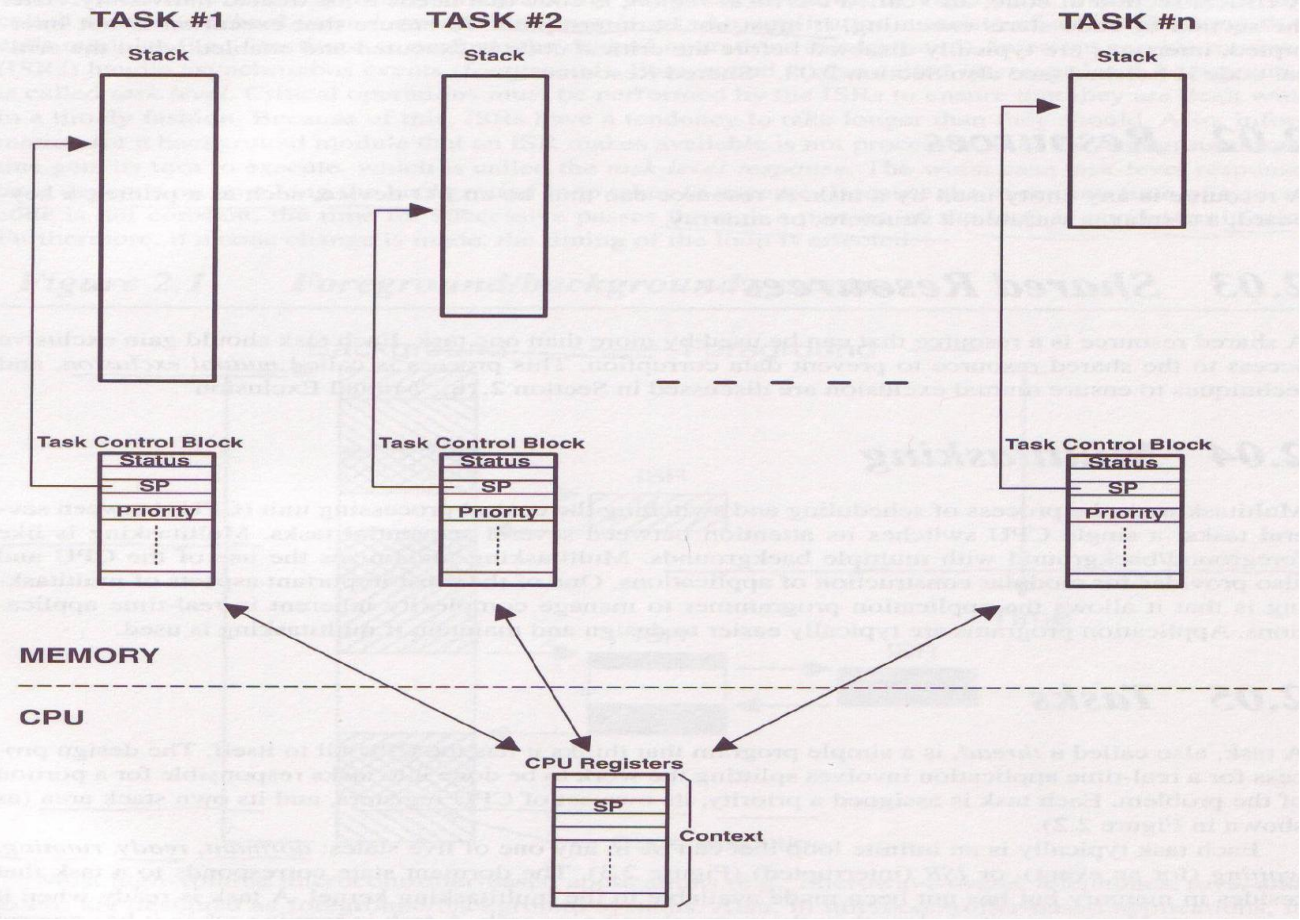
Background/Foreground Systems – 2/2

- Advantages
  - Low cost (no kernel and no royalty)
  - Small memory only for applications
- Disadvantages
  - Background is nondeterministic
  - All task have same priorities
  - Hard to program (all services should be programmed)

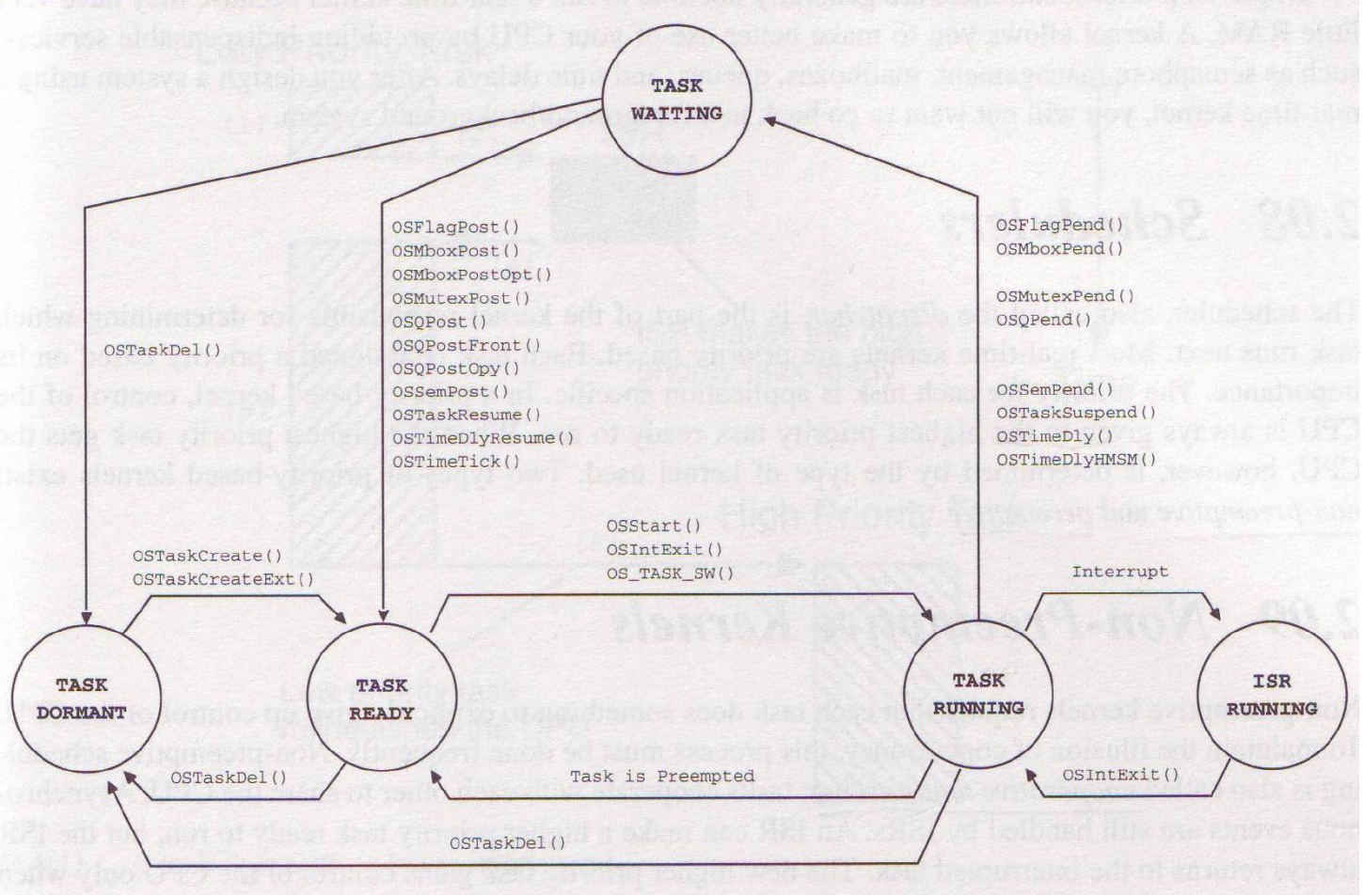
Useful Terms

- Critical section of codes – must not be interrupted
- Resources – processor, memory, devices, variables, programs
- Shared resources – used by more than one task
- Multitasking – switching CPU between several tasks
- Tasks – task thinks it has the CPU all to itself

Context Switches



Task States

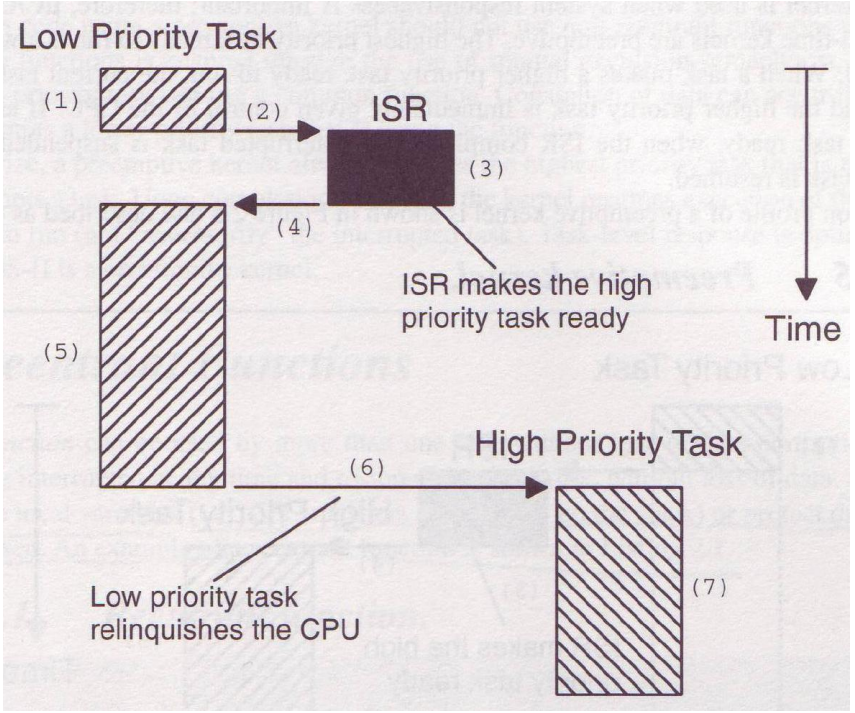

TASK WAITING
OSTaskDel()
OSFlagPost()
OSMboxPost()
OSMboxPostOpt()
OSMutexPost()
OSQPost()
OSQPostFront()
OSQPostOpy()
OSSemPost()
OSTaskResume()
OSTimeDlyResume()
OSTimeTick()
OSFlagPend()
OSMboxPend()
OSMutexPend()
OSQPend()
OSSemPend()
OSTaskSuspend()
OSTimeDly()
OSTimeDlyHMSM()
OSTaskCreate()
OSTaskCreateExt()
OSStart()
OSIntExit()
OS_TASK_SW()
Interrupt
TASK DORMANT
TASK READY
TASK RUNNING
ISR RUNNING
OSTaskDel()
Task is Preempted
OSIntExit()
OSTaskDel()

Kernels and Schedulers

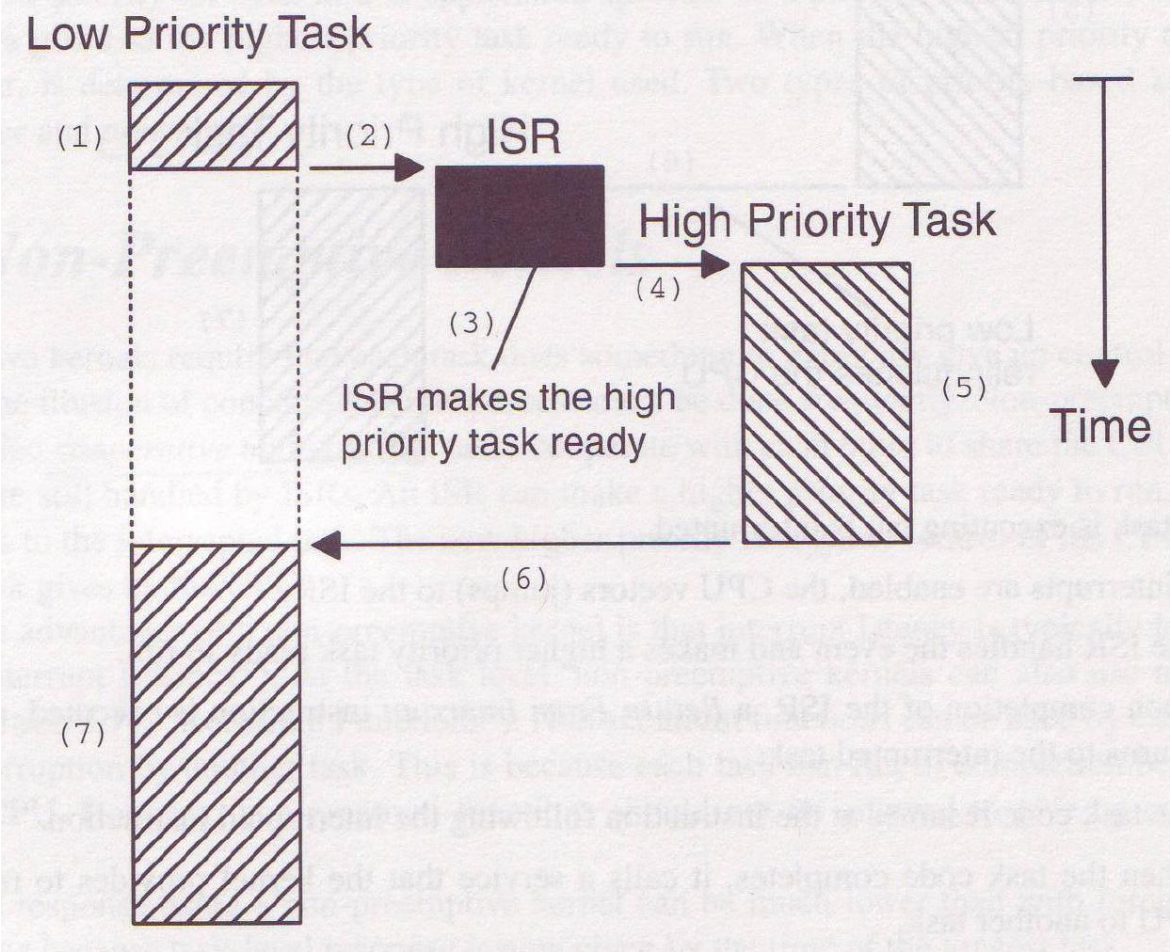
- Kernels
  - Management of tasks
  - Context Switching
  - Simplifies the design of system
  - Between 2 and 5% of CPU time
- Schedulers
  - A part of kernel
  - Called the dispatcher
  - Determines which task runs next

Non-Preemptive Kernels

- Cooperative multitasking
- Advantages
  - Low interrupt latency
  - Non reentrant function
  - Task level response time is lower than F/B systems
- Disadvantages
  - Task with high priority should wait a long time
  - Task level response time is nondeterministic



Preemptive Kernels



Reentrant Functions

- A reentrant function can be used by more than one task without fear of data corruption
- A reentrant function can be interrupted at any time and resume at later time without loss of data
- A reentrant code uses local variables (on stacks) or protected data when global variables are used

Reentrant Functions - Examples

Listing 2.1 Reentrant function.

```
void strcpy(char *dest, char *src)
{
    while (*dest++ = *src++) {
        ;
    }
    *dest = NUL;
}
```

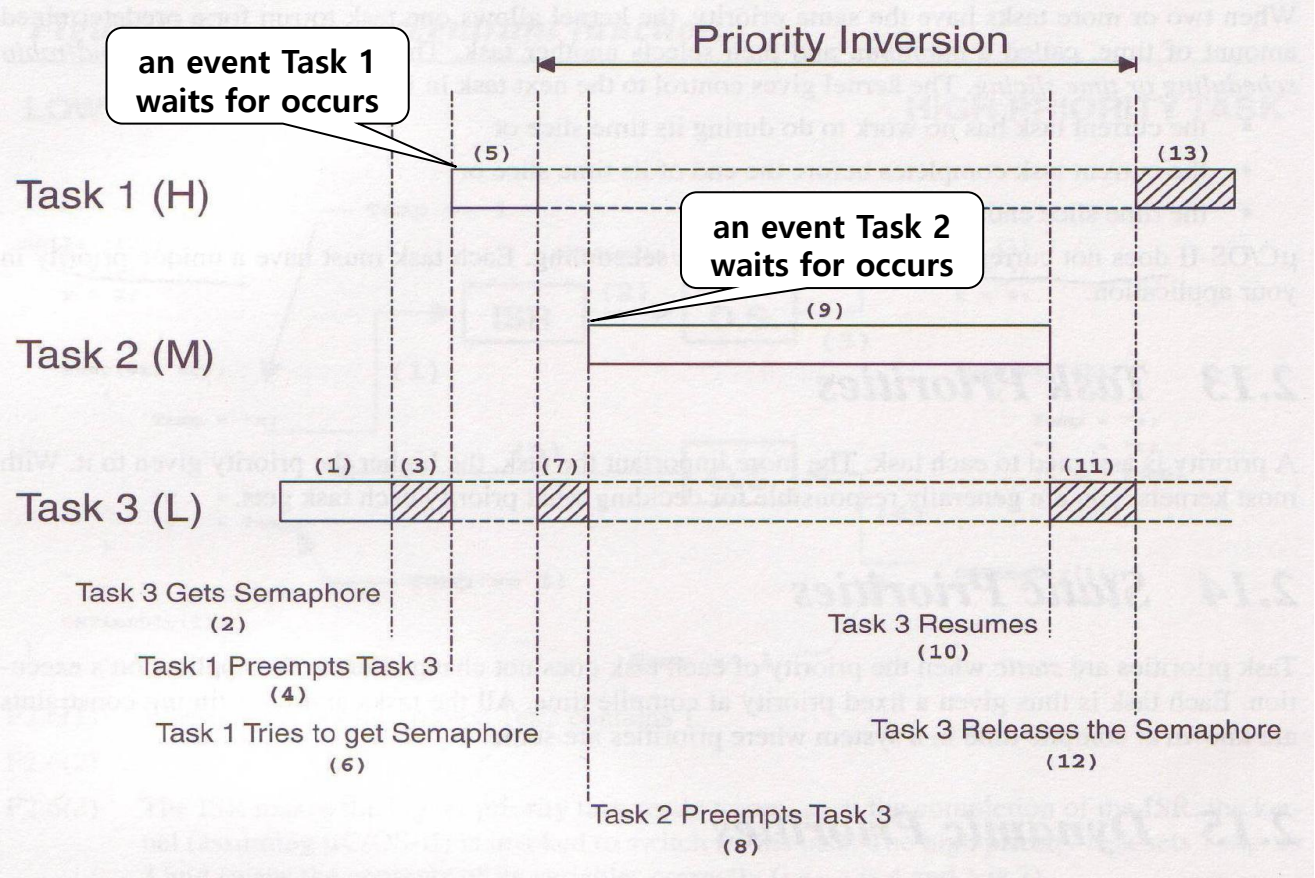
Listing 2.2 Non-reentrant function.

```
int Temp;

void swap(int *x, int *y)
{
    Temp = *x;
    *x   = *y;
    *y   = Temp;
}
```

Useful Terms

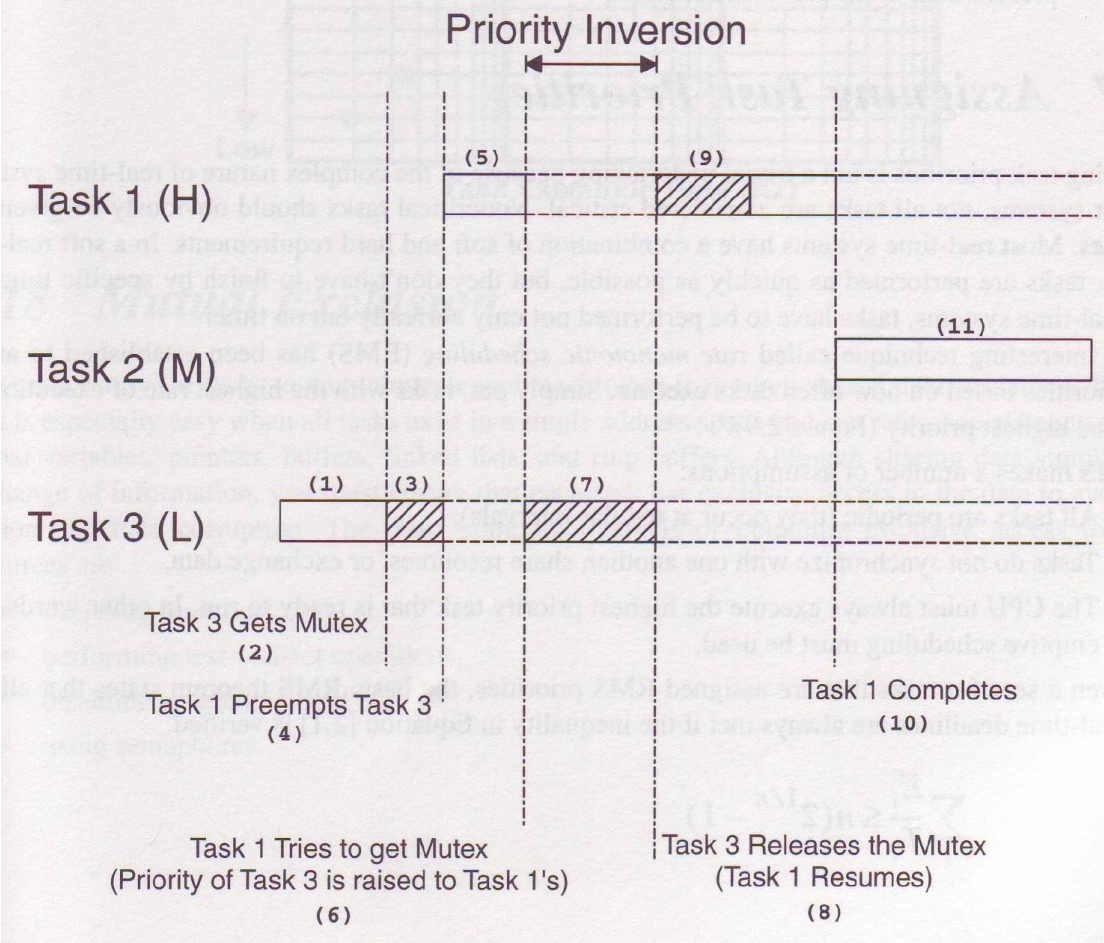
- Round-Robin Scheduling – allows one task to run for a predetermined amount of time, called a quantum, and then selects another task (uC/OS-II does not support it)
- Task Priorities
  - Static priorities
  - Dynamic priorities

Priority Inversion



우선순위 낮은 task가 가진 자원을 우선순위 높은 task가 요구하는 경우
우선순위 높은 task가 매우 오래 기다려야하는 현상

Priority Inheritance



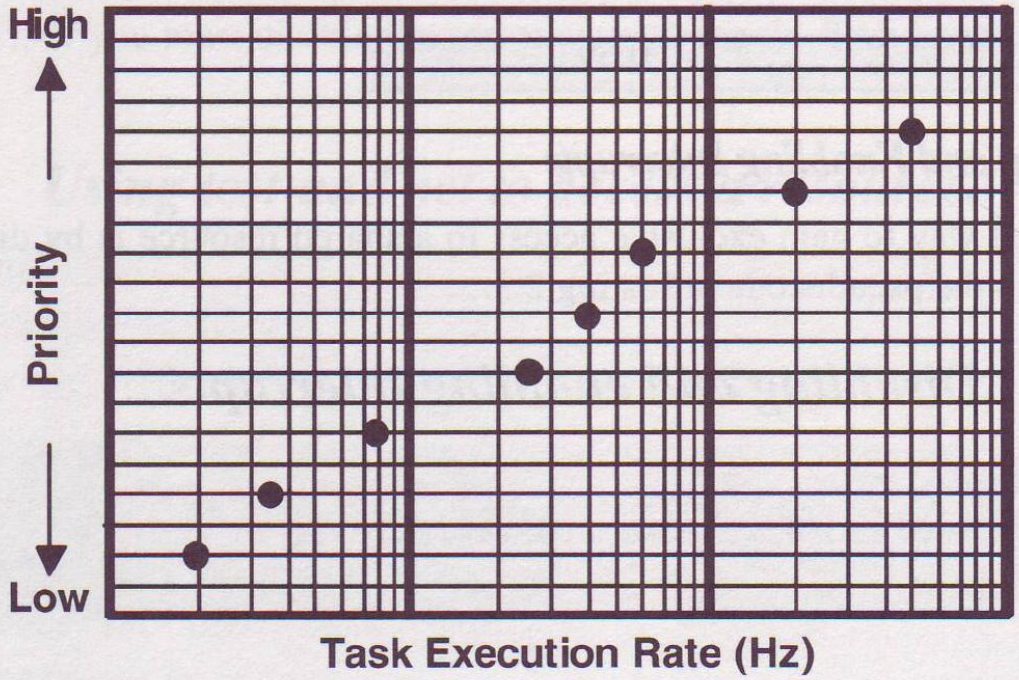
priority inversion을 해결하기 위하여 우선순위 낮은 task의 우선순위를 우선순위 높은 task의 우선 순위로 높이는 방법

Rate Monotonic Scheduling (RMS)

- RMS assumptions
  - All tasks with periods (run at regular intervals)
  - Tasks do not synchronize with one another, share resources, or exchange data
  - The CPU executes the highest priority task that is ready to run
- A feasible schedule that will always meet deadlines exists if the following inequality is verified
  - $U = \Sigma_i(Ei/Ti) \leq n(2^{1/n}-1)$ where n is the number of tasks, E is execution time and T is period
  - If n is infinite, the value of $n(2^{1/n}-1)$ is 0.693
  - If $E_1=1$, $E_2=2$, $E_3=2$, $T_1=8$, $T_2=5$, $T_3=10$, then U=0.725 and $3*(2^{1/3}-1)=0.779$. So a feasible schedule exists
- RMS says the highest rate task has the highest priority
  - T_5 in the example above

Assigning Task Priorities

Figure 2.9 *Assigning task priorities based on task execution rate.*



Mutual Exclusion

- Disabling interrupts
- Disabling scheduling
- Using semaphores

Disabling Interrupts

Listing 2.4 Using µC/OS-II macros to disable and enable interrupts.

```
void Function (void)
{
    OS_ENTER_CRITICAL();
    .
    .    /* You can access shared data in here */
    .
    OS_EXIT_CRITICAL();
}
```

be careful not to disable interrupt too long

Disabling Scheduling

Listing 2.6 Accessing shared data by disabling and enabling scheduling.

```
void Function (void)
{
    OSSchedLock();
    .
    .    /* You can access shared data in here (interrupts are recognized) */
    .
    OSSchedUnlock();
}
```

similar to non-preemptive kernel when ISR returns

Using Semaphores

- Semaphore is used to
  - Control access to a shared resources (mutual exclusion)
  - Signal the occurrence of an event
  - Allows two tasks to synchronize their activities

Using Semaphores - Example

Listing 2.7 Accessing shared data by obtaining a semaphore.

```
OS_EVENT *SharedDataSem;

void Function (void)
{
    INT8U err;

    OSSemPend(SharedDataSem, 0, &err);
    .
    .    /* You can access shared data in here (interrupts are recognized) */
    .
    OSSemPost(SharedDataSem);
}
```

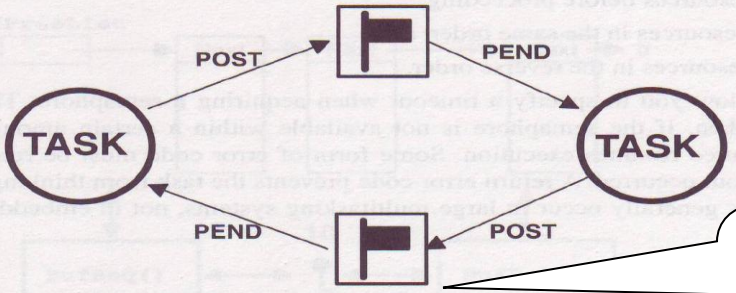
the highest priority task waiting for the semaphore gets the semaphore in uC/OS-II

Deadlock

- Two tasks are waiting for resources held by the other
- Most kernel allows you to specify a timeout when acquiring a semaphore – This feature allow a deadlock to be broken
- Deadlocks generally occur in large multitasking system not in embedded systems

Synchronization

Figure 2.14 Tasks synchronizing their activities.



two tasks can synchronize using two semaphores

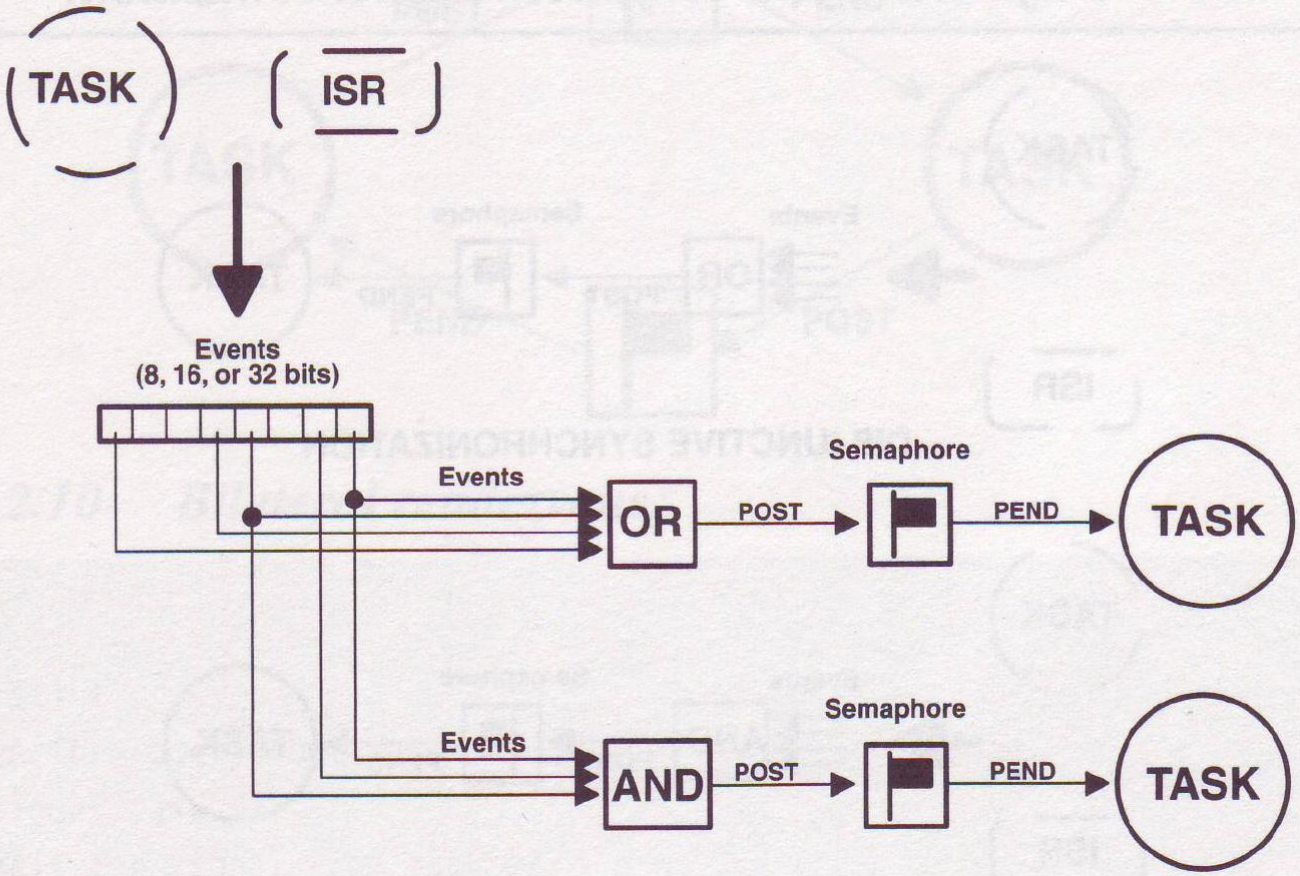
```
void Task1(void *pdata)
{
  INT8U err;
  for (;;) {
    OSSemPend(Sem1, 0, &err);              /* 1로 초기화 */
    /* 여기서 Task1의 주어진 일을 수행한다. */
    OSSemPost(Sem2);
  }
}

void Task2(void *pdata)
{
  INT8U err;
  for (;;) {
    OSSemPend(Sem2, 0, &err);              /* 0으로 초기화 */
    /* 여기서 Task2의 주어진 일을 수행한다. */
    OSSemPost(Sem1);
 }
}
```

Event Flags

- Event flags are used when a task need to synchronize with the occurrence of multiple events – conjunction or disjunction
- uC/OS-II offers services to SET event flags, CLEAR event flags, and wait for event flags (conjunctively or disjunctively)

Event Flags - Figure

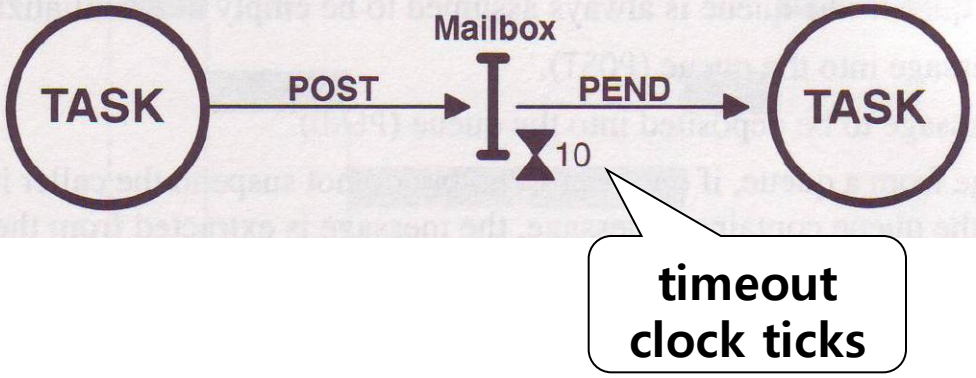
Figure 2.16 Event flags.



Message Mailboxes

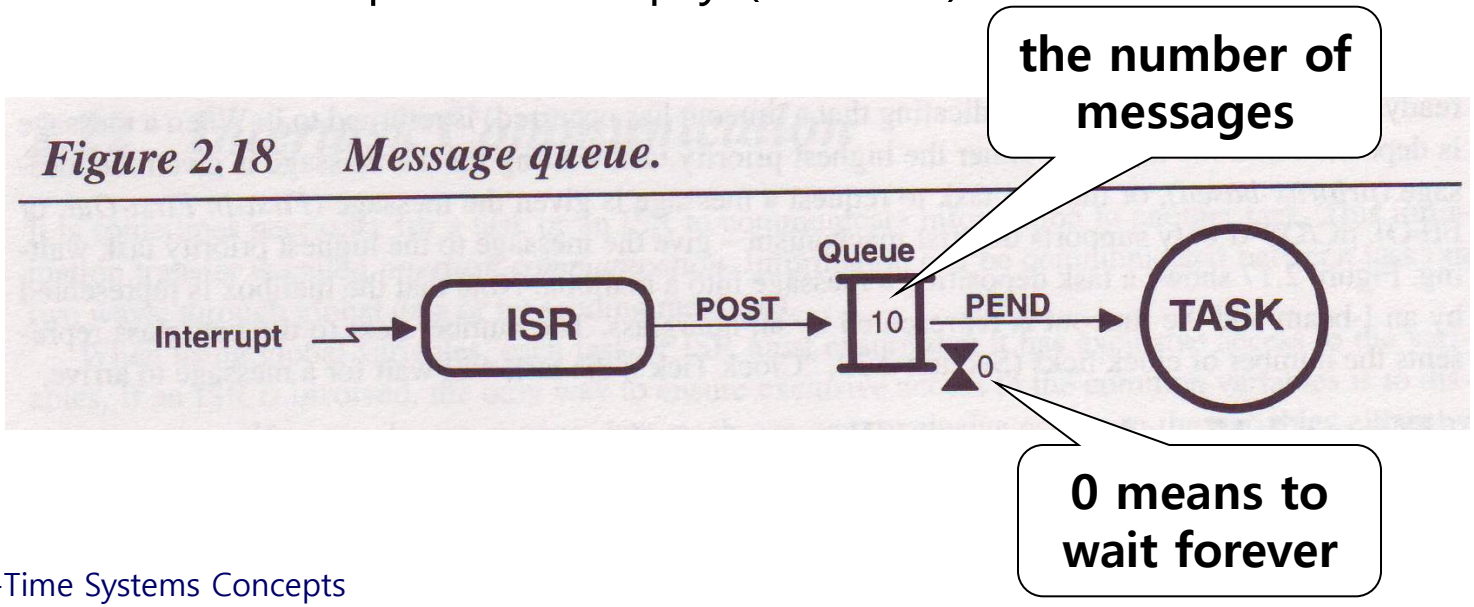
- Kernel supports
  - Initialize the contents of a mailbox
  - Deposit a message into the mailbox (POST)
  - Wait for a message to be deposited into the mailbox (PEND)
  - Get a message if one is present, but do not suspend the caller if the mailbox is empty (ACCEPT)

Figure 2.17 Message mailbox.



Message Queues

- Kernel supports
  - Initialize the queue (FIFO or LIFO)
  - Deposit a message into the queue (POST)
  - Wait for a message to be deposited into the queue (PEND)
  - Get a message if one is present, but do not suspend the caller if the queue is empty (ACCEPT)

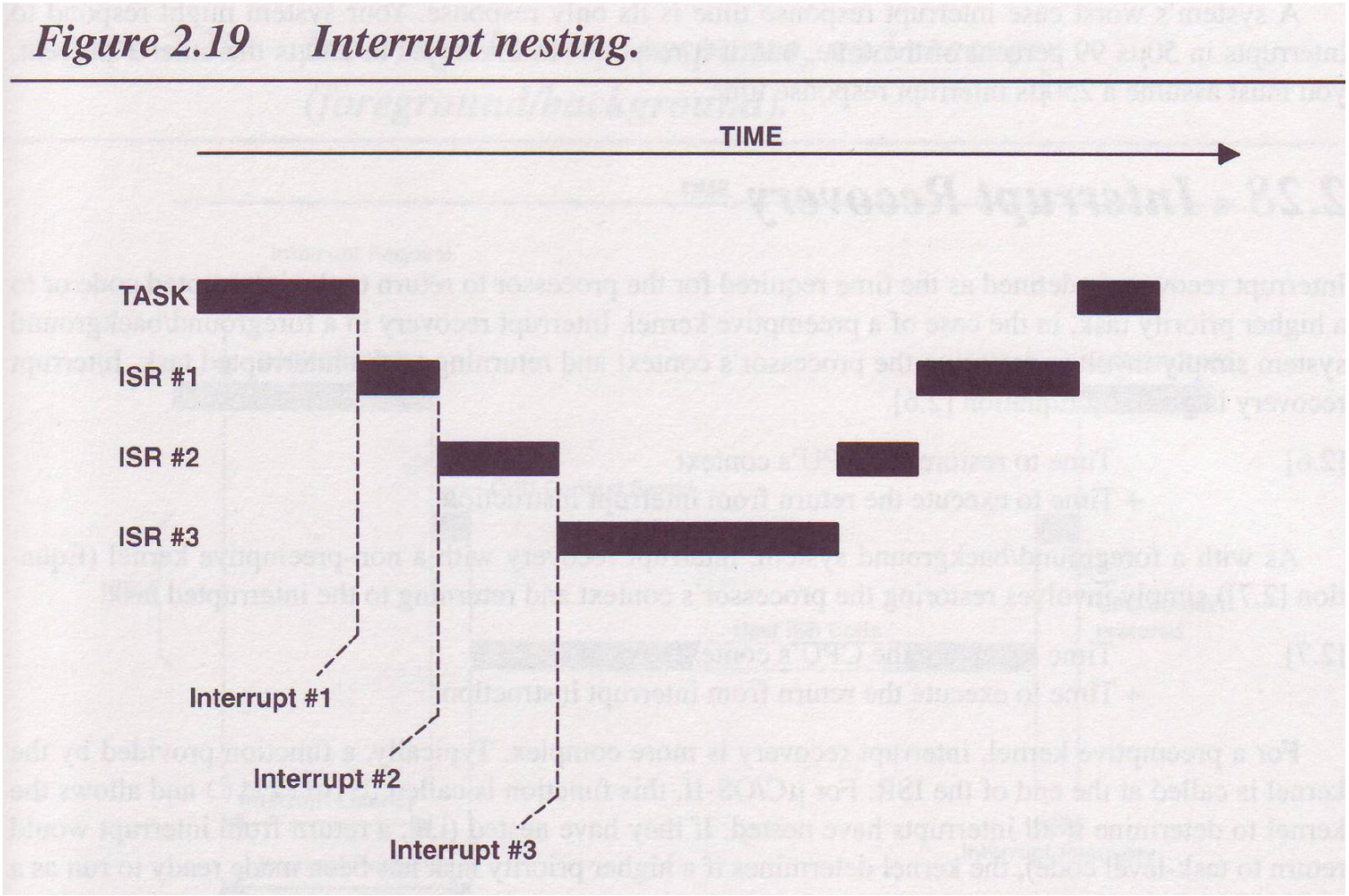


Interrupts

- An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred
- When an interrupt is recognized, the CPU save its context and jumps to a special subroutine, called an interrupt service routine (ISR)
- The ISR process the event, and, upon completion of the IRS, the program returns to the highest priority task ready to run
- Processor generally allows interrupts to be nested

Interrupt Nesting

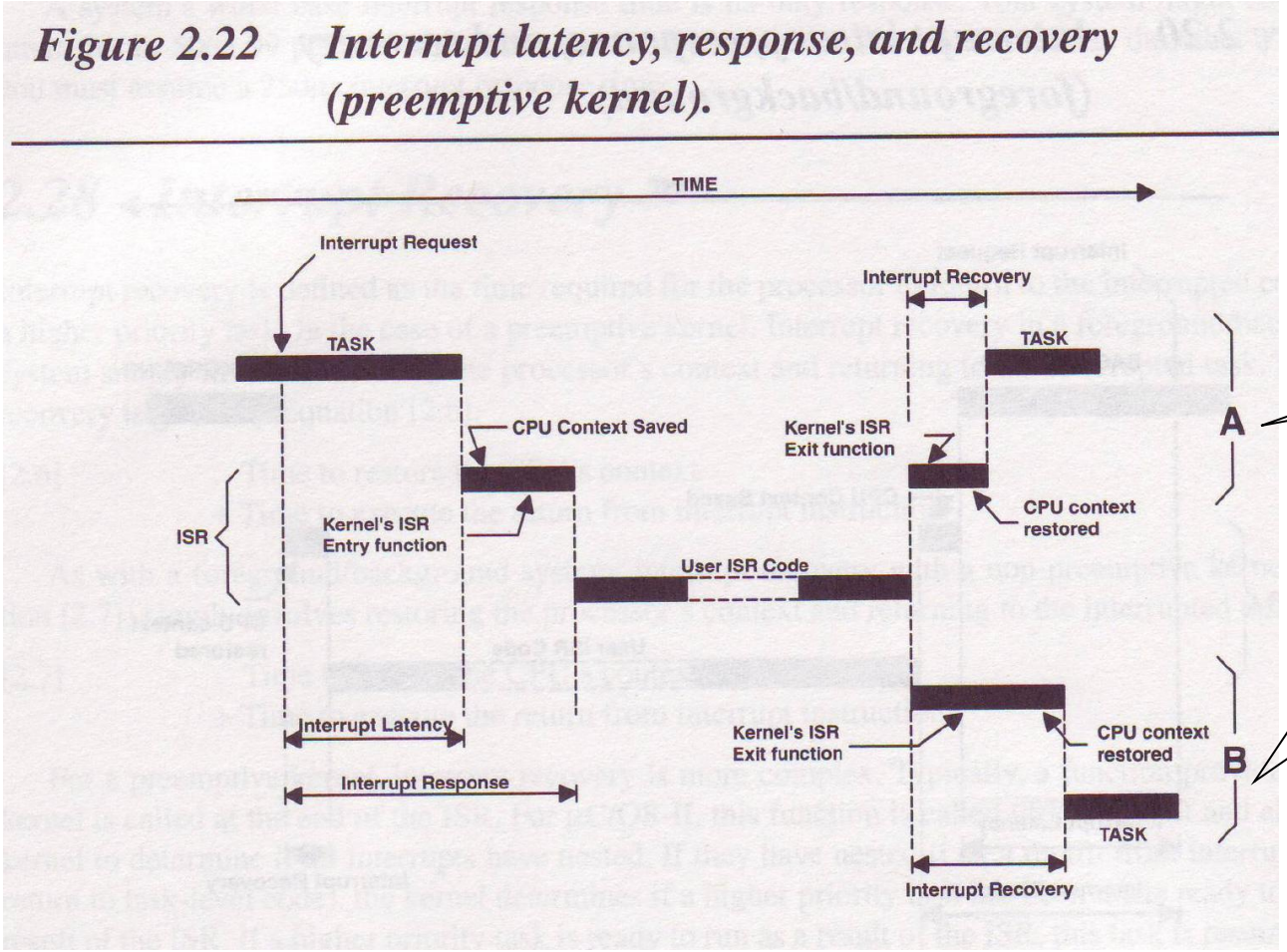
Figure 2.19 Interrupt nesting.



Interrupt Latency, Response, and Recovery

- Interrupt latency = max amount of time interrupts are disabled + time to start executing the first instruction in the ISR
- Interrupt response = Interrupt latency + time to save the CPU's context + execution time of the kernel ISR entry function
- Interrupt recovery = 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 - Figure



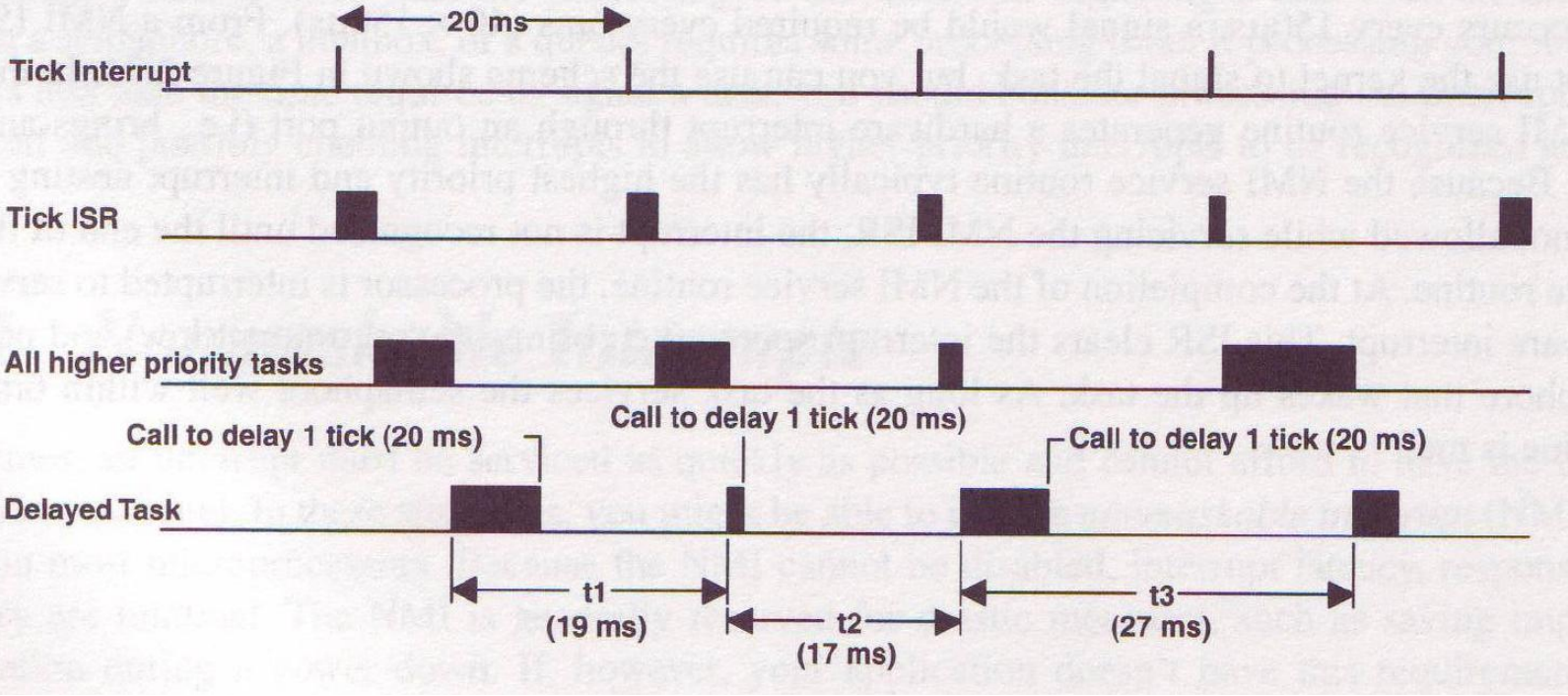
Clock Tick

- All kernels allow tasks to be delayed for a certain number of clock ticks
- The resolution of delayed tasks is one tick; however, this does not mean that its accuracy is one clock tick
- In SE3208 port, clock tick Hertz is defined as

```
#define OS_TICKS_PER_SEC 100
```

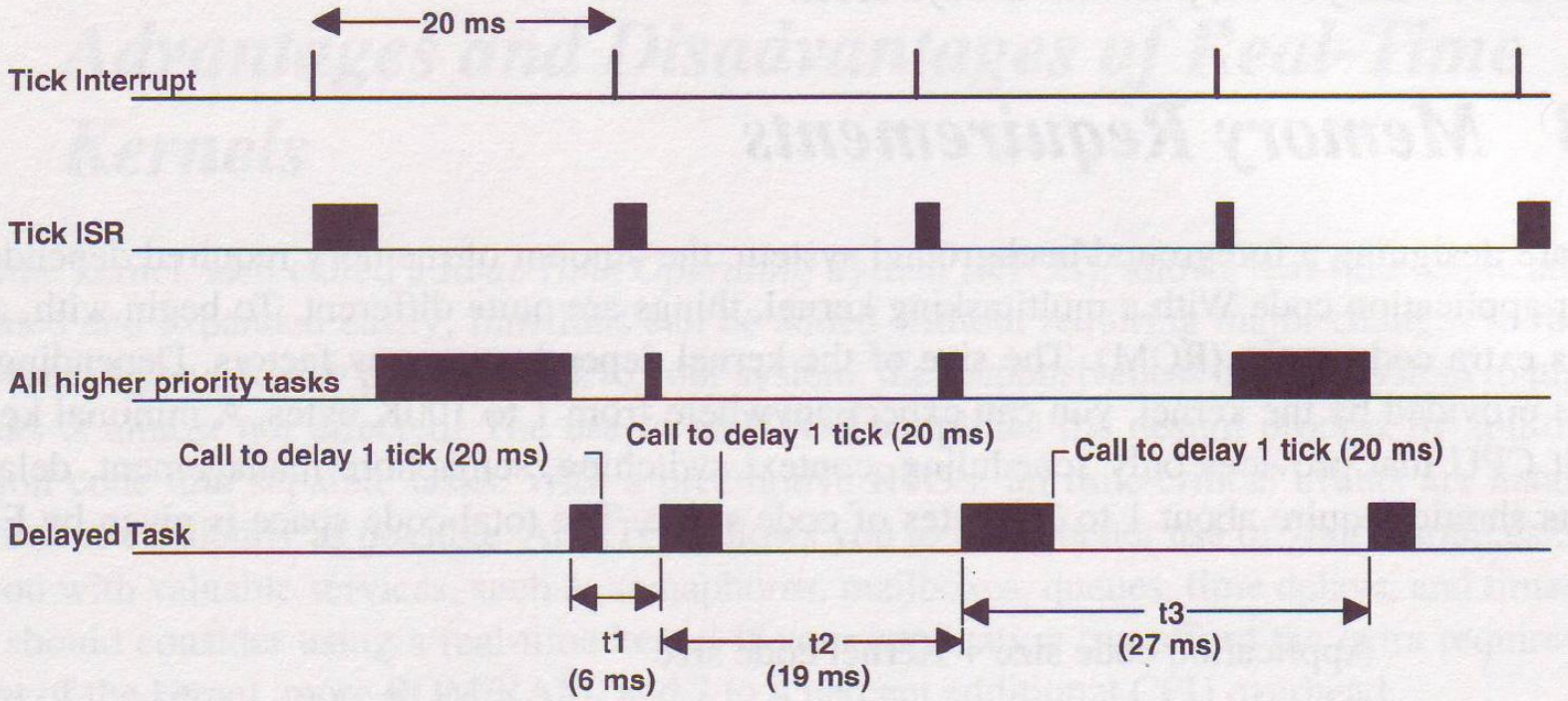
Delaying a Task for One Tick – Case 1

Figure 2.25 Delaying a task for one tick (Case 1).

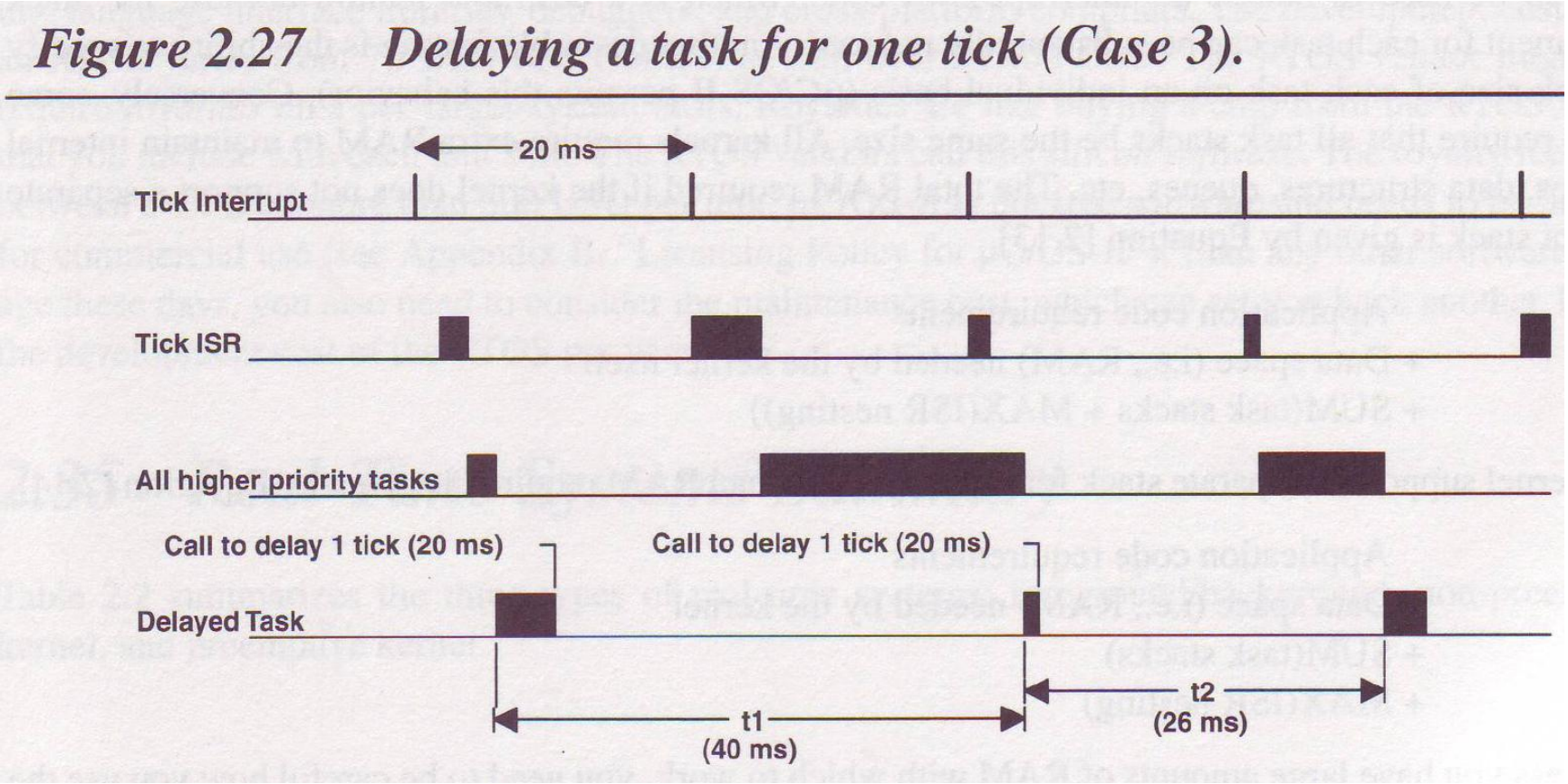


Delaying a Task for One Tick – Case 2

Figure 2.26 Delaying a task for one tick (Case 2).



Delaying a Task for One Tick – Case 3



Memory Requirements

- Total code size (ROM) = code size for kernel + code size for tasks
- Total data size (RAM) = data size for kernel + data size for tasks + stack size for tasks + stack size for ISR

Advantages and Disadvantages of Real-Time Kernels

- Advantages
  - Simplify the design process
  - All time-critical events are handled as quickly and as efficiently
  - Provide many services: semaphores, message mailboxes, message queues, …
- Disadvantages
  - Need extra ROM/RAM
  - Need 2 to 4% additional CPU overhead
  - Need license cost