

uC/OS-II Part 2: Real-Time System Concepts

Prof. Li-Pin Chang
Embedded Software and Storage Lab.
National Chiao-Tung University

Objectives

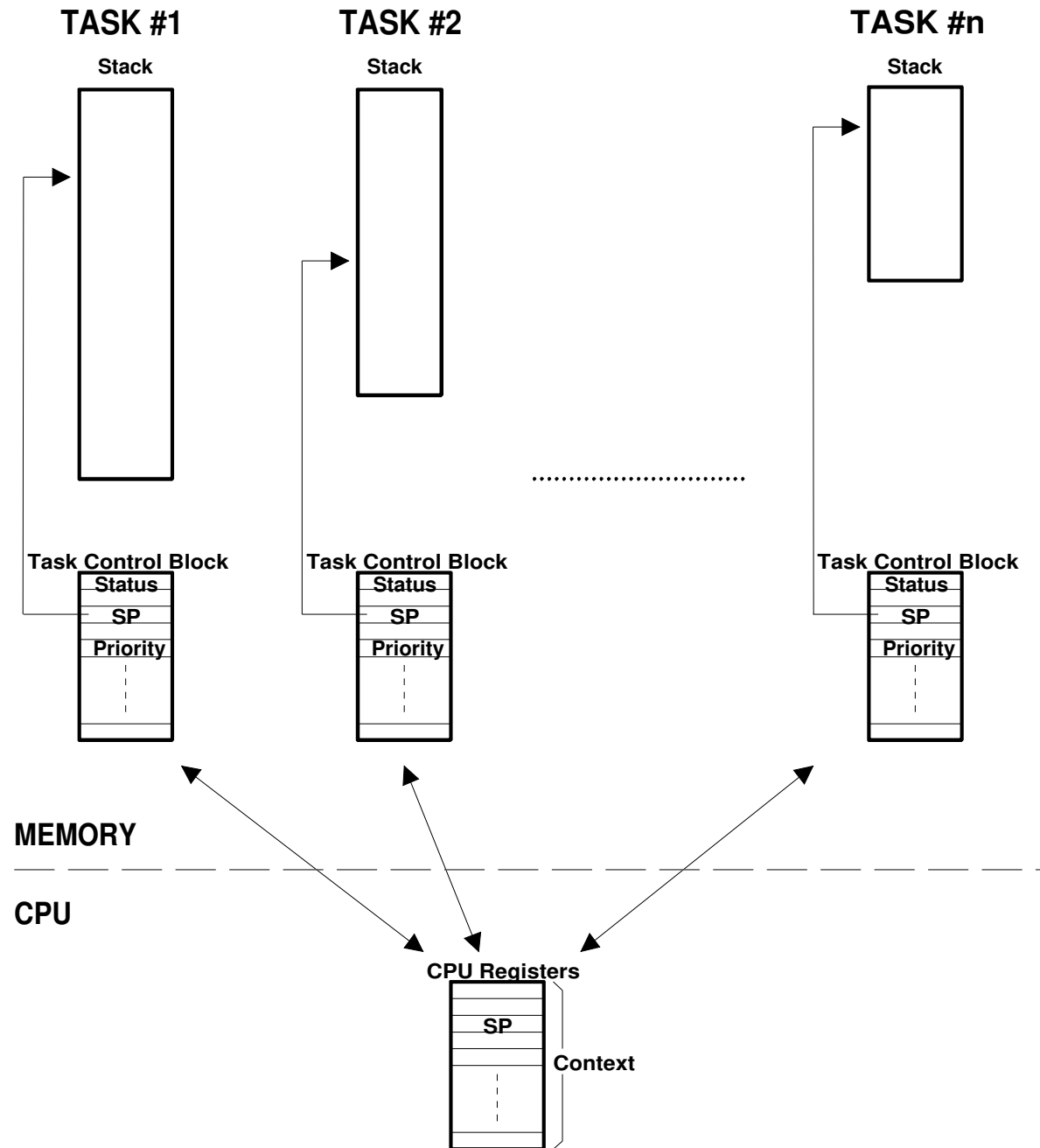
- To understand
 - Priority-driven scheduling
 - Interrupt handling
 - Context switch

Multitasking

- The scheduler switches the attention of CPU among several tasks
 - Tasks are logically concurrent
 - The scheduler determines how much CPU share each task receives

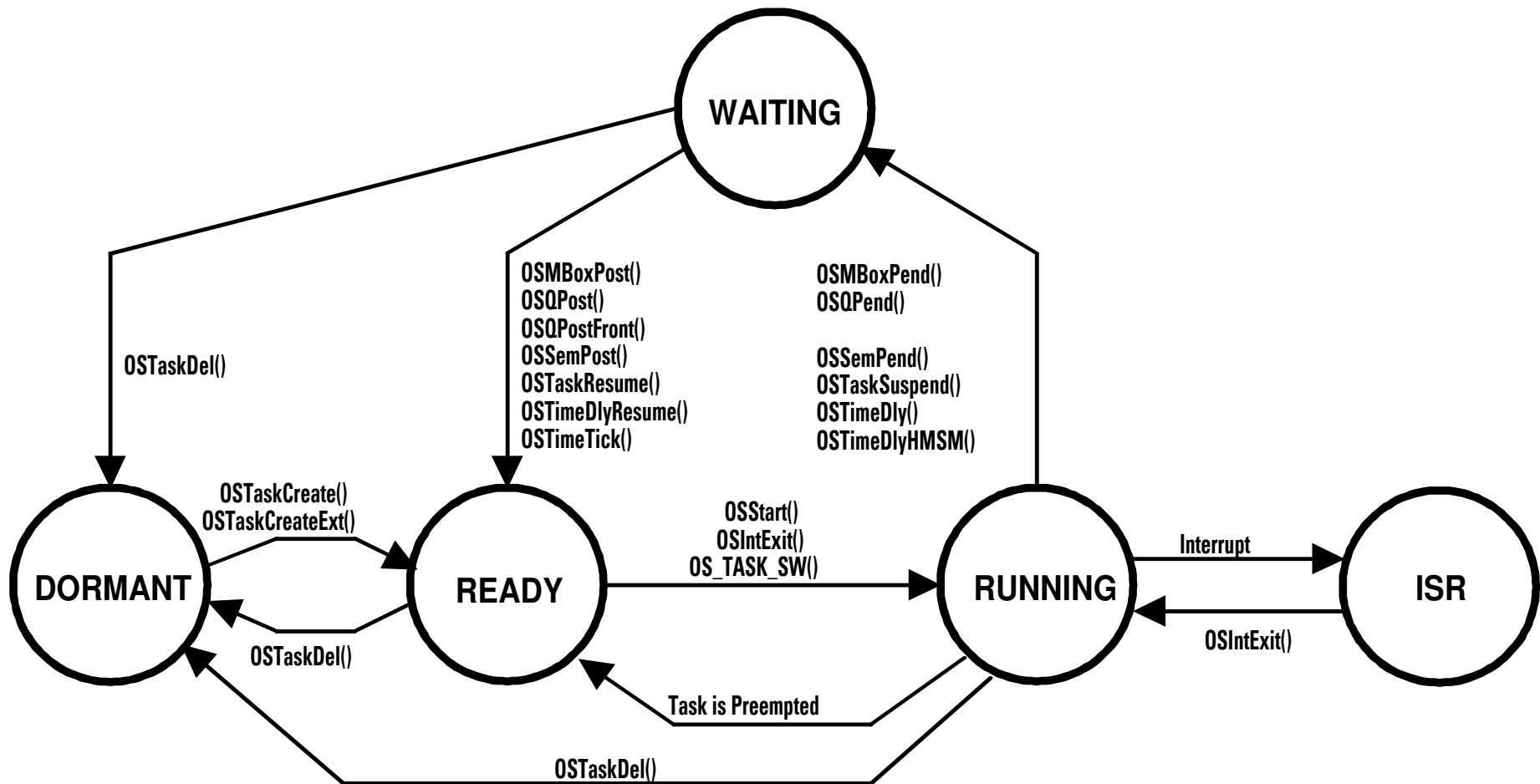
Task

- Tasks, threads, processes are used interchangeably
- From the point of view of OS, a task is comprised of
 - a priority
 - a set of registers
 - its own stack
 - some housekeeping information (e.g., TCB)



Task State

- The structure of a real-time periodic task is a big, infinite loop
- uC/OS-2 defines 5 task states :
 - Dormant, ready, running, waiting, and interrupted.



Context Switch (1/2)

- It occurs when the scheduler switches the CPU from the current task to another task
- The scheduler must save the context of the current task and then restore the context of the task-to-run

Context Switch (2/2)

- Context switches add extra time overheads
 - Intensive context switches should be prevented because modern processors have deep pipelines and large register files
- The overhead of a context switch is part of the specification of an RTOS
 - This overhead is accounted to
 - The preempting tasks
 - The blocked tasks

Kernels

- The kernel is responsible for
 - Task management
 - Inter-task communication
- The kernel adds additional time/space overheads
 - Kernel services take time
 - Semaphores, message queues, mailboxes, timing controls, and etc...
 - Kernel resident in RAM and/or ROM

US

Service	Interrupts Disabled			Minimum			Maximum		
	I	C	確	I	C	確	I	C	確
Miscellaneous									
OSInit()	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OSSchedLock()	4	34	1.0	7	87	2.6	7	87	2.6
OSSchedUnlock()	57	567	17.2	13	130	3.9	73	782	23.7
OSStart()	0	0	0.0	35	278	8.4	35	278	8.4
OSStatInit()	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OSVersion()	0	0	0.0	2	19	0.6	2	19	0.6
Interrupt Management									
OSIntEnter()	4	42	1.3	4	42	1.3	4	42	1.3
OSIntExit()	56	558	16.9	27	207	6.3	57	574	17.4
OSTickISR()	30	310	9.4	948	10803	327.4	2304	20620	624.8
Message Mailboxes									
OSMboxAccept()	15	161	4.9	13	257	7.8	13	257	7.8
OSMboxCreate()	15	148	4.5	115	939	28.5	115	939	28.5
OSMboxPend()	68	567	17.2	28	317	9.6	184	1912	57.9
OSMboxPost()	84	747	22.6	24	305	9.2	152	1484	45.0
OSMboxQuery()	120	988	29.9	128	1257	38.1	128	1257	38.1
Memory Partition Management									
OSMemCreate()	21	181	5.5	72	766	23.2	72	766	23.2
OSMemGet()	19	247	7.5	18	172	5.2	33	350	10.6
OSMemPut()	23	282	8.5	12	161	4.9	29	321	9.7
OSMemQuery()	40	400	12.1	45	450	13.6	45	450	13.6
Message Queues									
OSQAccept()	34	387	11.7	25	269	8.2	44	479	14.5
OSQCreate()	14	150	4.5	154	1291	39.1	154	1291	39.1
OSQFlush()	18	202	6.1	25	253	7.7	25	253	7.7
OSQPend()	64	620	18.8	45	495	15.0	186	1938	58.7
OSQPost()	98	873	26.5	51	547	16.6	155	1493	45.2
OSQPostFront()	87	788	23.9	44	412	12.5	153	1483	44.9
OSQQuery()	128	1100	33.3	137	1171	35.5	137	1171	35.5
Semaphore Management									
OSSemAccept()	10	113	3.4	16	161	4.9	16	161	4.9
OSSemCreate()	14	140	4.2	98	768	23.3	98	768	23.3
OSSemPend()	58	567	17.2	17	184	5.6	164	1690	51.2
OSSemPost()	87	776	23.5	18	198	6.0	151	1469	44.5
OSSemQuery()	110	882	26.7	116	931	28.2	116	931	28.2

Table 9.3, Execution times of μ C/OS-II services on 33 MHz 80186.

Schedulers

- A scheduler is a part of a kernel, it is responsible for choosing the next task-to-run
 - Preemptive scheduling or non-preemptive scheduling
 - Priority-driven or deadline-driven
- For priority-driven scheduling, the highest-priority ready task always gains the control of the CPU

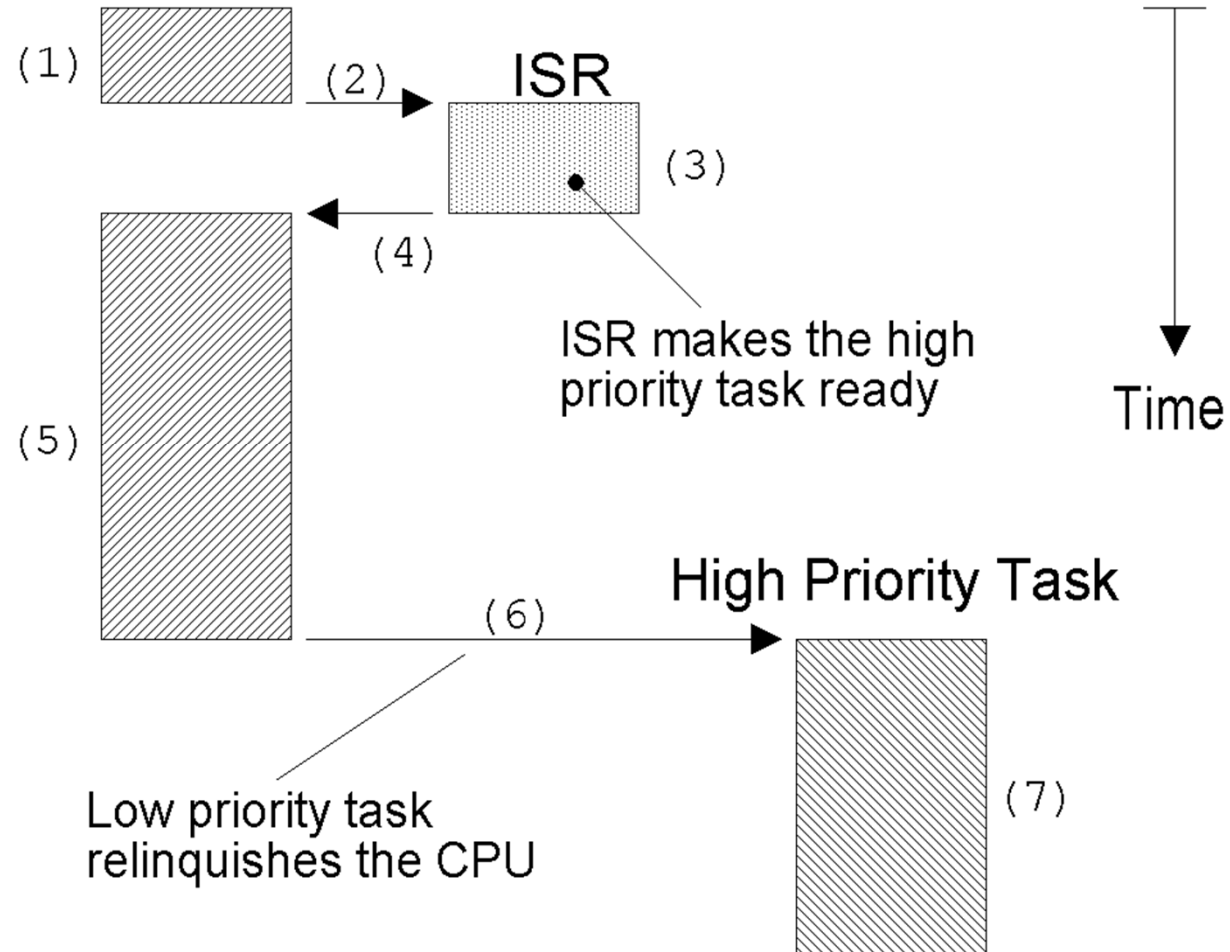
Non-Preemptive Kernels (1/2)

- Context switches occur only when a task explicitly surrenders the CPU
 - This procedure must be done frequently to improve the responsiveness
- Interrupts are handled in ISRs, and ISRs always return to the interrupted task
 - Interrupts can make task ready but introduce no preemption

Non-Preemptive Kernels (2/2)

- Free from task-to-task race
 - No protection of critical sections is required
- There is still race between task and ISR
 - Can be solved by interrupt disabling or deferred tasks
- Pros: simple and robust
- Cons: low response poor schedulability

Low Priority Task

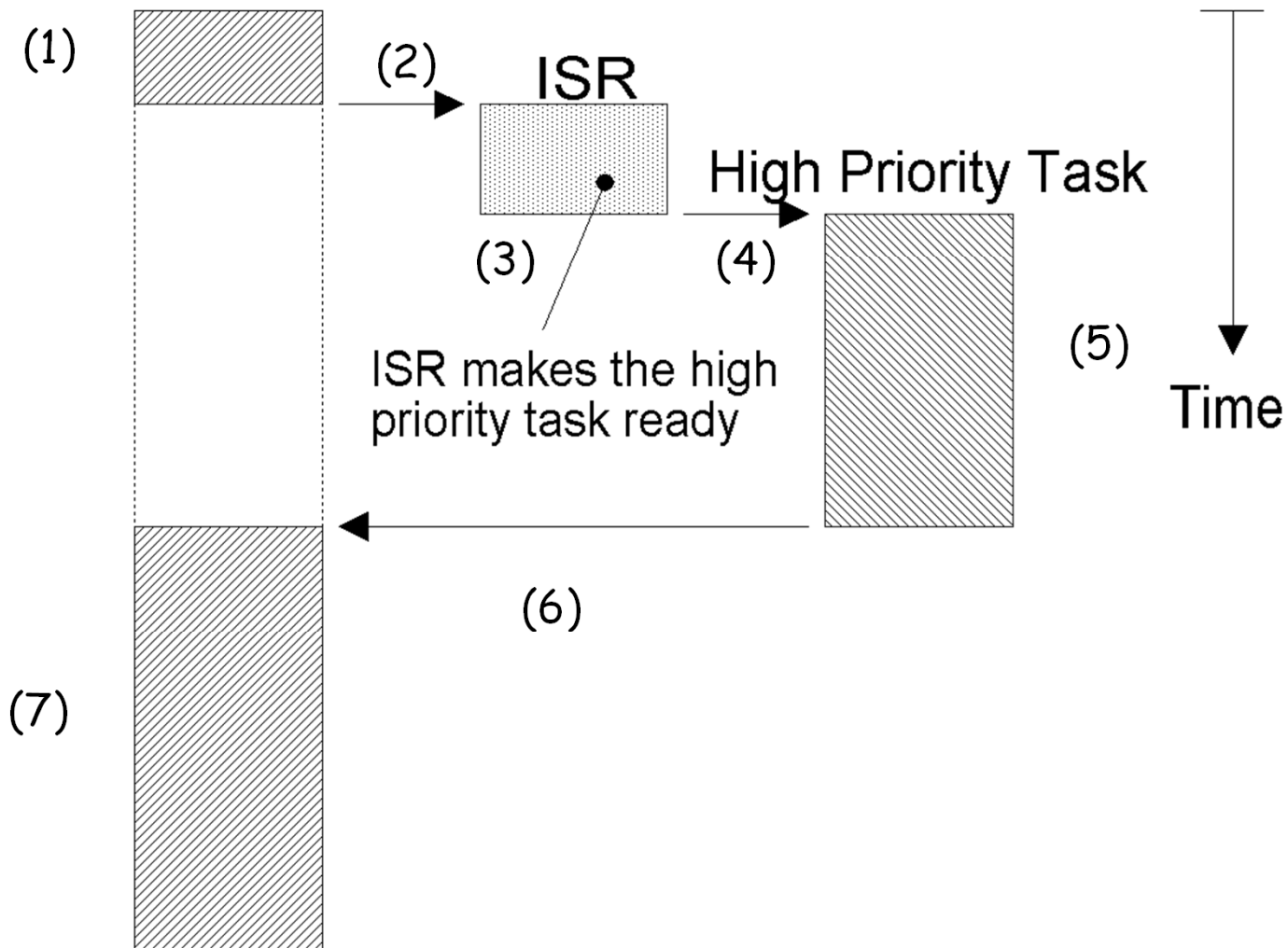


- (1) A task is executing but gets interrupted.
- (2) If interrupts are enabled, the CPU vectors (i.e. jumps) to the ISR.
- (3) The ISR handles the event and makes a higher priority task ready-to-run.
- (4) Upon completion of the ISR, a *Return From Interrupt* instruction is executed and the CPU returns to the interrupted task.
- (5) The task code resumes at the instruction following the interrupted instruction.
- (6) When the task code completes, it calls a service provided by the kernel to relinquish the CPU to another task.
- (7) The new higher priority task then executes to handle the event signaled by the ISR.

Preemptive Kernels

- A preemptive kernel is more responsive
 - uC/OS-2 (and most RTOSes) is preemptive
 - A ready, high-priority task gains control of the CPU instantly
- ISR may not return to the interrupted task
 - It may return to a ready, high-priority task
- Both task-task race and task-ISR races exist

Low Priority Task

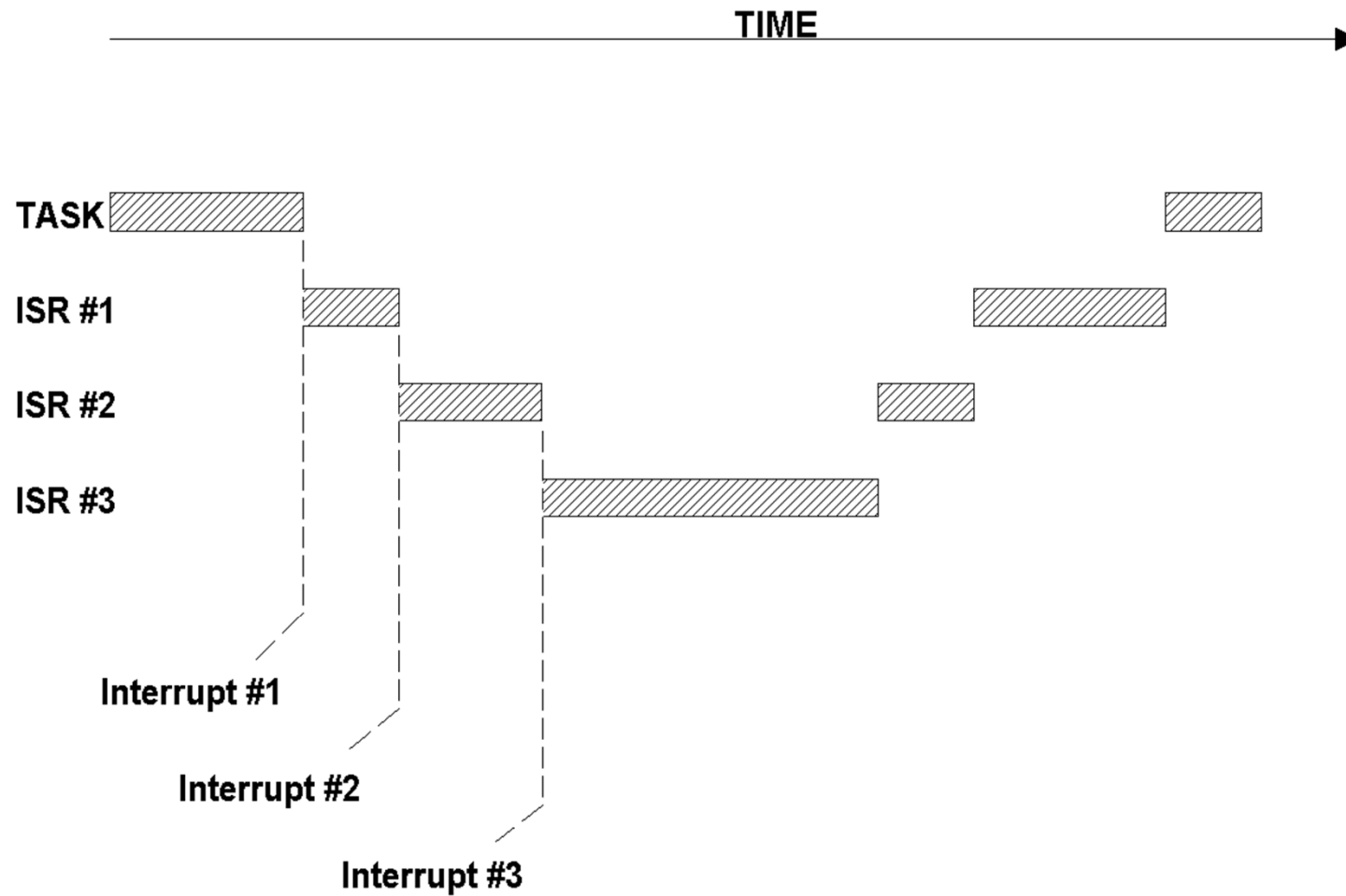


- (1) A task is executing but interrupted.
- (2) If interrupts are enabled, the CPU vectors (jumps) to the ISR.
- (3) The ISR handles the event and makes a higher priority task ready to run. Upon completion of the ISR, a service provided by the kernel is invoked. (i.e., a function that the kernel provides is called).
- (4)
- (5) This function knows that a more important task has been made ready to run, and thus, instead of returning to the interrupted task, the kernel performs a context switch and executes the code of the more important task. When the more important task is done, another function that the kernel provides is called to put the task to sleep waiting for the event (i.e., the ISR) to occur.
- (6)
- (7) The kernel then sees that a lower priority task needs to execute, and another context switch is done to resume execution of the interrupted task.

Interrupts

- A hardware event to inform the CPU of an asynchronous event
 - clock tick, I/O events, and hardware errors
- The context of the current task is saved and the corresponding service routine (ISR) is invoked
- The ISR processes the event, and upon completion of the ISR, it returns to
 - The interrupted task in a non-preemptive kernel
 - The highest-priority ready task in a preemptive kernel

Nested Interrupts



Interrupt Latency

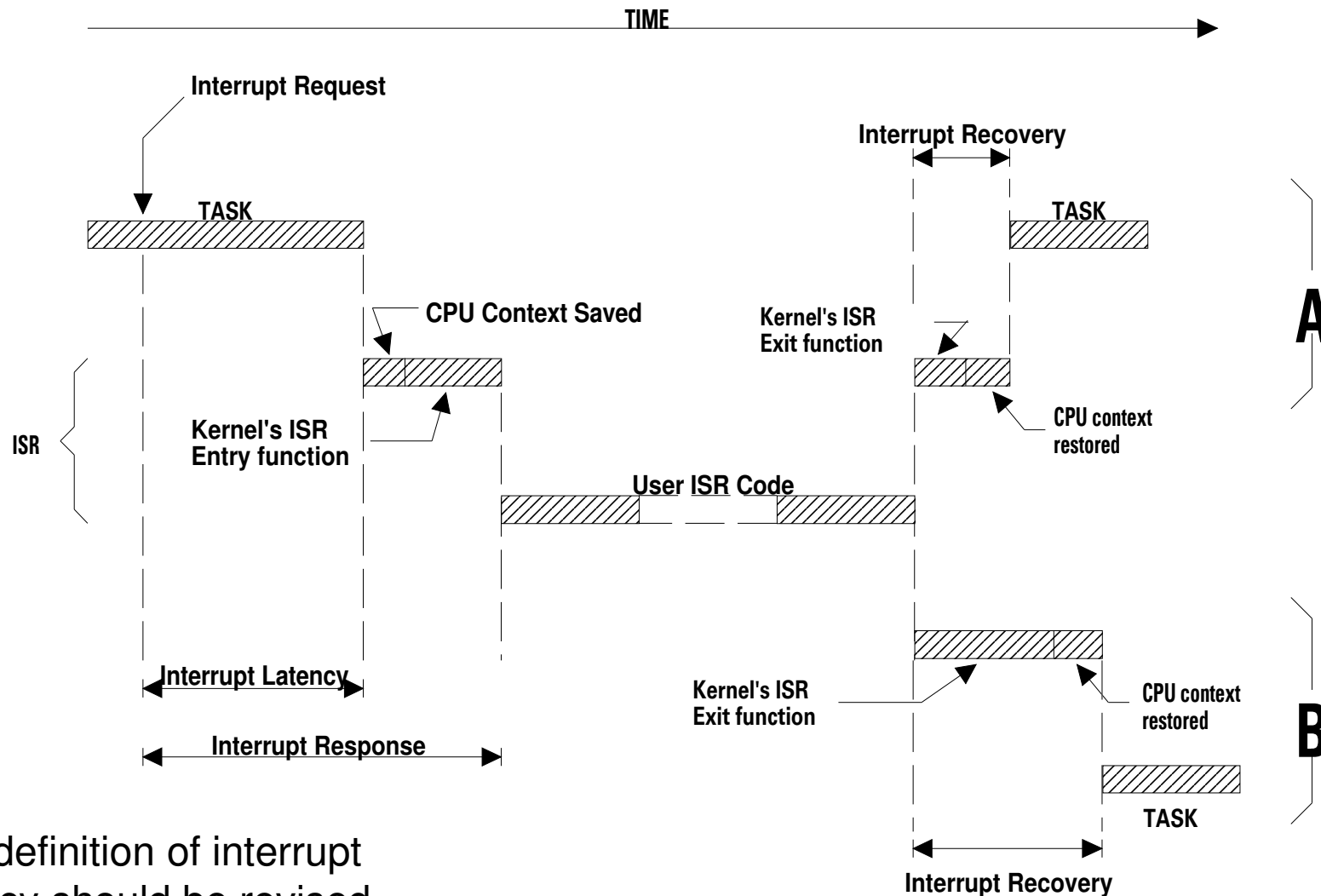
- The longer the interrupts are disabled, the higher the interrupt latency would be

interrupt latency =
the time length of interrupt disabling +
Time to start executing the first instruction in the ISR

ISR Processing Time

- In most cases, an ISR should
 - Save the context of the current task
 - Recognize the interrupt
 - Obtain data or status from the interrupting device
 - Resume task execution
- ISRs should be as short as possible
 - ISRs add asynchronous, extra time overhead
 - Don't do a big job in ISR, make a worker task ready for follow-ups

Interrupt latency, response, and recovery (Preemptive kernel)



The definition of interrupt latency should be revised...

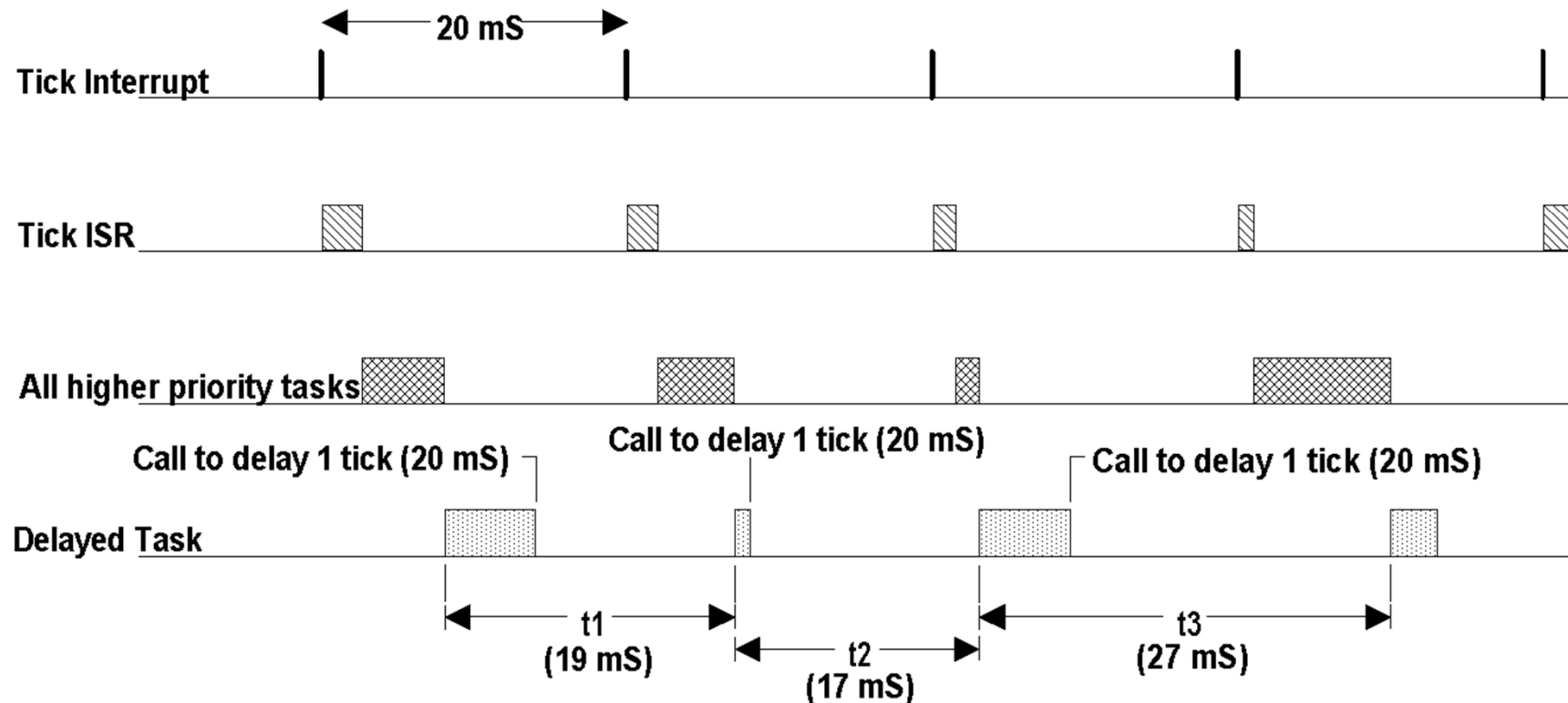
Clock Tick

- Periodic hardware event (interrupt) generated by a timer
- The heart-beat of a system
- The higher the tick rate is,
 - the better the responsiveness is
 - the higher the cxtsw frequency is

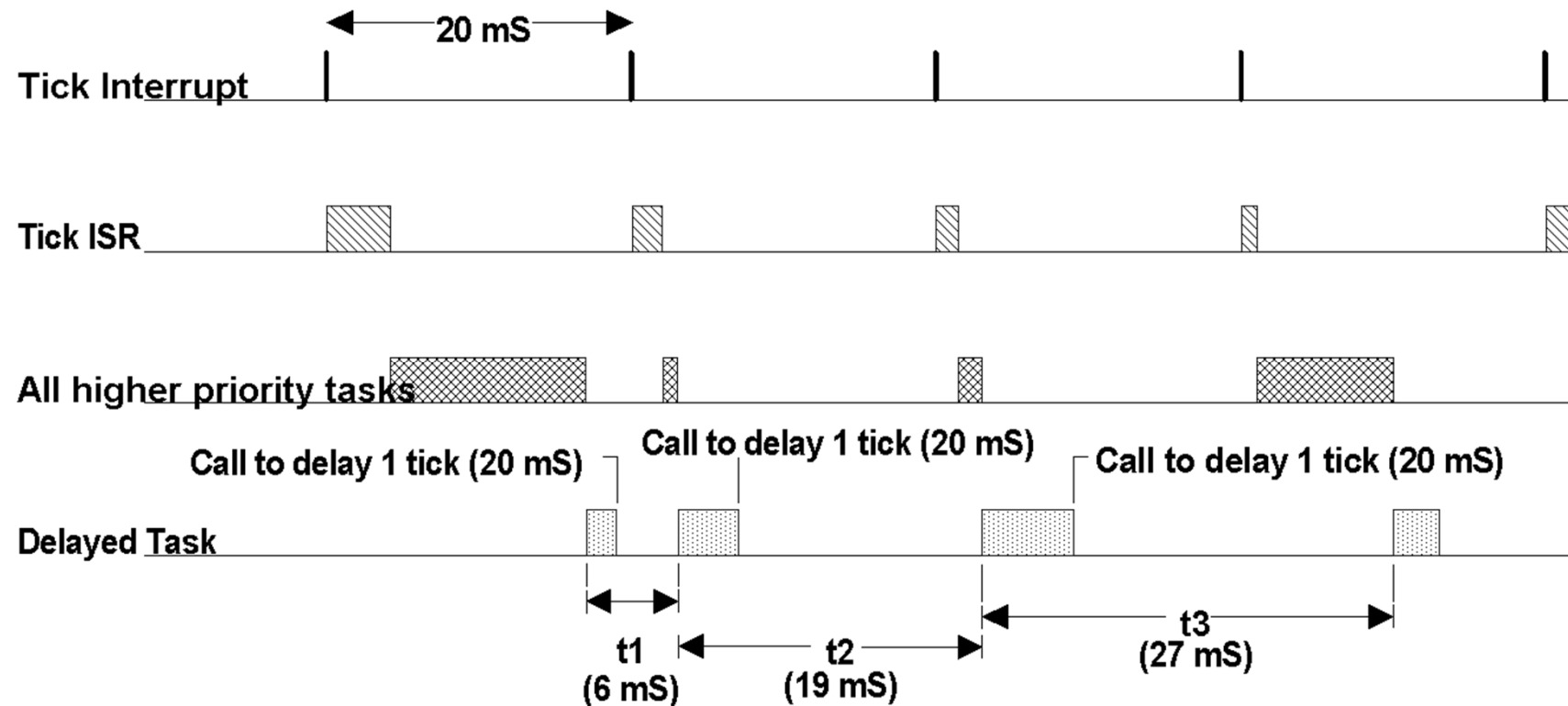
OSTimeDly()

- Delay for a specified tick count
- The actual “sleep time” may never be as accurate as specified
- Several sources of errors
 - The ready task must wait other HPTs
 - ISR overhead
 - Tick count is discrete

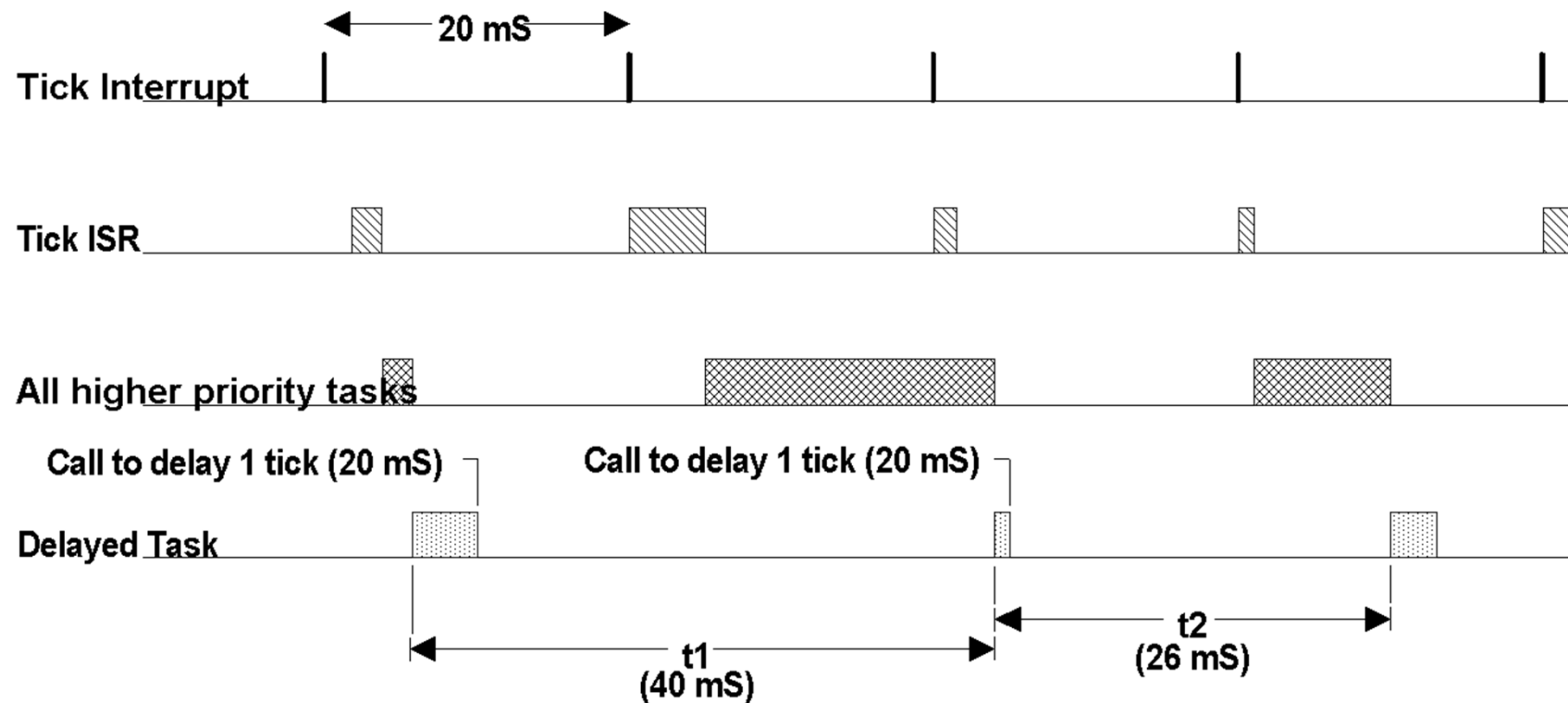
The difference between period and inter-execution time



- A task delaying itself for one tick
- Higher priority tasks and ISRs execute prior to the task, which needs to delay for 1 tick



- The execution times of all higher priority tasks and ISRs are slightly less than 1 tick
- As a result, if you need to delay at least one tick, you must specify one extra tick



- The execution times of all higher priority tasks and ISRs are more than 1 clock tick.
- The task that tries to delay for 1 tick actually executes two ticks later and violates its deadline.

- Delaying $t+1$ ticks guarantees sufficient time of delaying t ticks ($\geq t$)
- This does not affect the scheduling theory, which uses task arrival times (and execution durations) to determine system feasibility

Memory Requirements

- RAM usage = code + data + stack
- Code: the size of the executable binary
 - Can be determined at compile time
- Data + stack:
 - Global variables (including free lists)
 - Can be determined at compile time
 - Task stacks, used by ISRs and kernel code
 - Can be (partially) determined by offline analysis
- Total RAM requirement =
application requirement +
kernel requirement +
SUM(each task(stack + MAX(ISR nesting)))

Memory Requirements

- We must be careful about stack usage:
 - Large arrays and structures as local variables
 - Recursive function calls
 - ISR nesting
 - Function calls with many arguments

Summary

- In RTOS, as long as events occurs, context switches are conducted with respect to task priorities
- The accounting of interrupt latency
- The accounting of the timing overheads due to clock granularities