COE718: Embedded Systems Design

Lecture 9:
Real-Time Scheduling Cont'd

D.W. Lewis, "Fundamentals of Embedded Software", Chapter 10

Anita Tino

What We Have Covered So Far

- Rate Monotonic Scheduling (RMS)
- Earliest Deadline First (EDF)
- Priority Inversion

- Why do we need these scheduling algorithms?

Deadline Monotonic Scheduling (DMS)

- Task model the same as RMS, however Di <= Ti
  - i.e. The deadline may not necessarily be (ideally) the same as the period
- Priorities assigned the same way as RMS – Shorter deadline, higher priority
- For historical reasons, DMS is often referred to as RMS (very similar)

Deadline Monotonic Scheduling (DMS)

- Schedulability test

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{1/N} - 1)$$

- C = computation time
  - Worst Case Execution Time (WCET)
- T = min time b/w process releases
- N = number of processes

An example for DMS…

Ways to Check Schedulability

1. Schedulability Test
   - Sufficient, but inconclusive
2. Draw the schedule out for the first set of periods given in process set

Ways to Check Schedulability

1. Schedulability Test
   - Sufficient, but inconclusive
2. Draw the schedule out for the first set of periods given in process set
3. Response Time calculations

Response Time Calculations

Method:

- In order of priority, calculate Task i's worst-case response time, R_i using:

$$R_i = C_i + I_i$$

- Where I is the interference from higher priority tasks
- Also note that $R_i \leq D_i$

Response Time Calculations

- Interference consists of:

$$\text{Number of Releases} = \left\lceil \frac{R_i}{T_j} \right\rceil \qquad \text{Total interference} = \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- Therefore, response time for a task set:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- hp(i) is the set of tasks with a higher priority than task i (then task which you're evaluating)
- For the highest priority, *R* is then simply *C*

Response Time Calculations

- Therefore, response time for a task set:

$$R_i = C_i + \boxed{\sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j}$$

All tasks that keep preempting current task i

- hp(i) is the set of tasks with a higher priority than task i (then task which you're evaluating)
- Ri = the previous response time of task i
- Tj and Cj = period and computation of the other tasks (that keep preempting this task i)
  - Which adds to the response time (Ri) of this task

Response Time Calculations

- We solve Response time by forming a recurrence relationship:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

- Therefore once we plugin the values and obtain $w_i^0, w_i^1, w_i^2, \ldots, w_i^n, \ldots$, we will come to a solution when $w_i^n = w_i^{n+1}$
  - Indicates that the response time has settled in value (monotonically increasing)

An example of Response Time Calculations…

Response Time Algorithm

for i in 1..N loop -- for each process in turn
  n := 0
  $w_i^n := C_i$
loop
    calculate new w_i^{n+1}
    if $w_i^{n+1} = w_i^n$ then
      $R_i = w_i^n$
      exit value found
    end if
    if $w_i^{n+1} > T_i$ then
      exit value not found
    end if
    n := n + 1
  end loop
end loop

Going back: Priority Inversion

- Problematic scenario in scheduling
- When a high priority task is preempted (indirectly) by a lower priority task
  - Indirectly = Example – when a higher priority requires a shared resource which a lower task is executing on (must relinquish)
- Although not intended, this inverts the priorities of these two tasks
- Lab - Priority inheritance

Priority Ceiling Protocols

- Synchronization protocol to:
  - ensure mutual exclusive resources access
  - prevent deadlock
  - reduce blocking time
    - Especially transitive blocking (i.e. Chain blocking)
  - Ensure a high priority process can only be blocked at most once by a lower priority during its execution

Priority Ceiling Protocols

- 2 different protocols we will look at:
  1. OCPP: Original Ceiling Priority Protocol
  2. ICPP: Immediate Ceiling Priority Protocol

OCPP

- Each process has a static default priority assigned
- Each resource defines its static (ceiling) value
  - This value defines the *maximum static priority* a process may have to use the resource

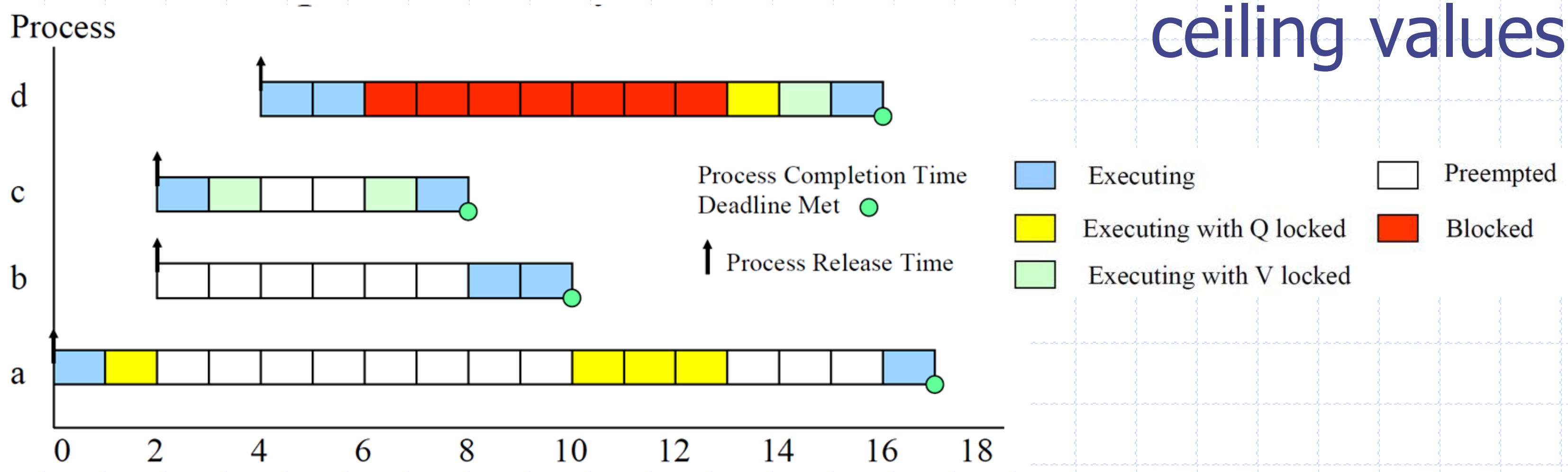
OCPP

- Each process also has a dynamic priority
  - Assigned at runtime
  - *Max*(**static** priority, any *process* priority it has to **inherit** due to blocking)
- Locking a resource:
  - Can be gained by a process only if its dynamic priority is higher than the ceiling value of any currently locked resource
    - Excluding any resource that it has already locked itself

OCPP

- Example from last class

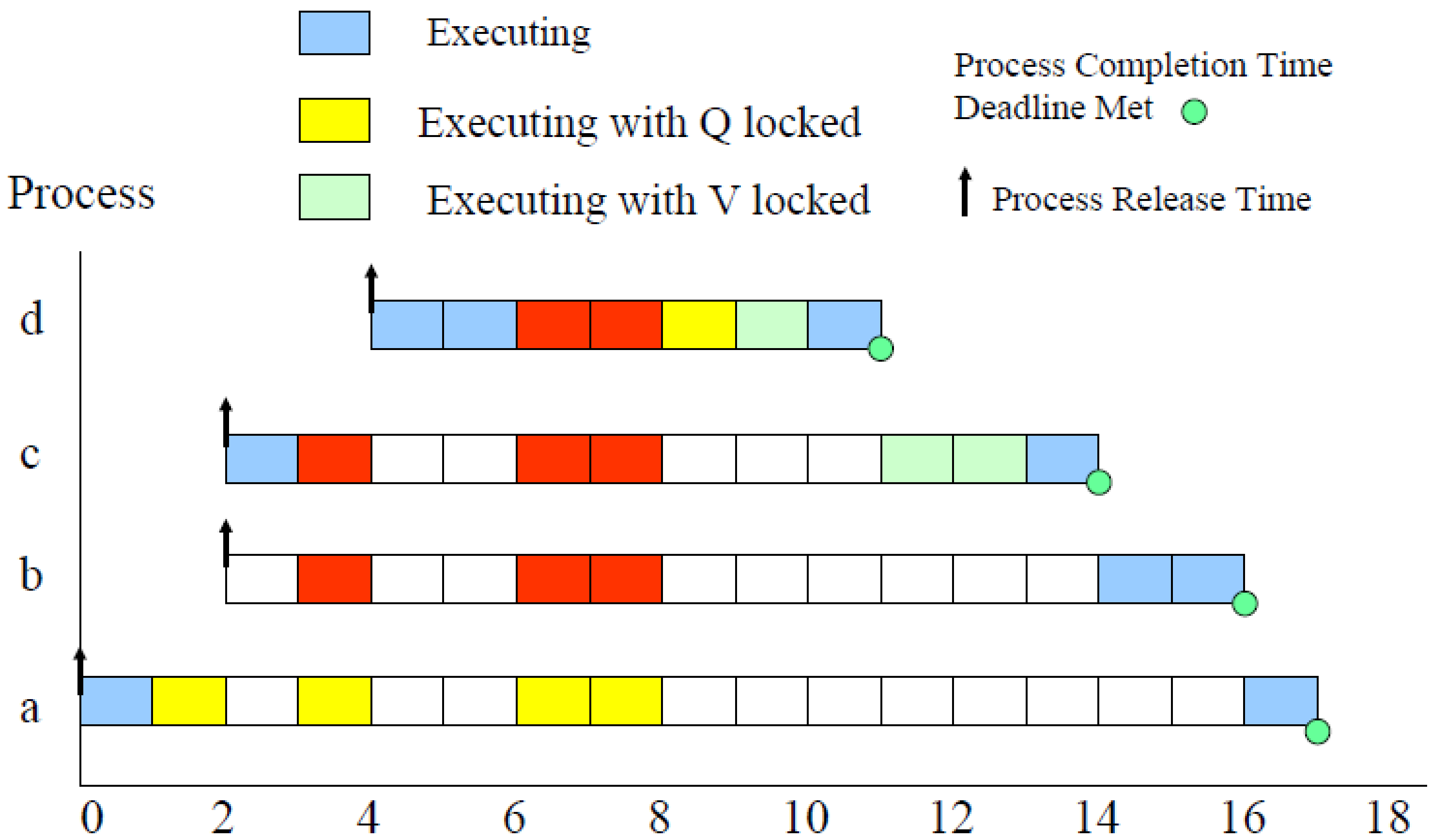
Process	Priority	Execution Sequence	Release Time
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4



- Now we need:
  - Dynamic priority
  - Resource ceiling values

OCPP

Process	Priority	Execution Sequence	Release Time
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4

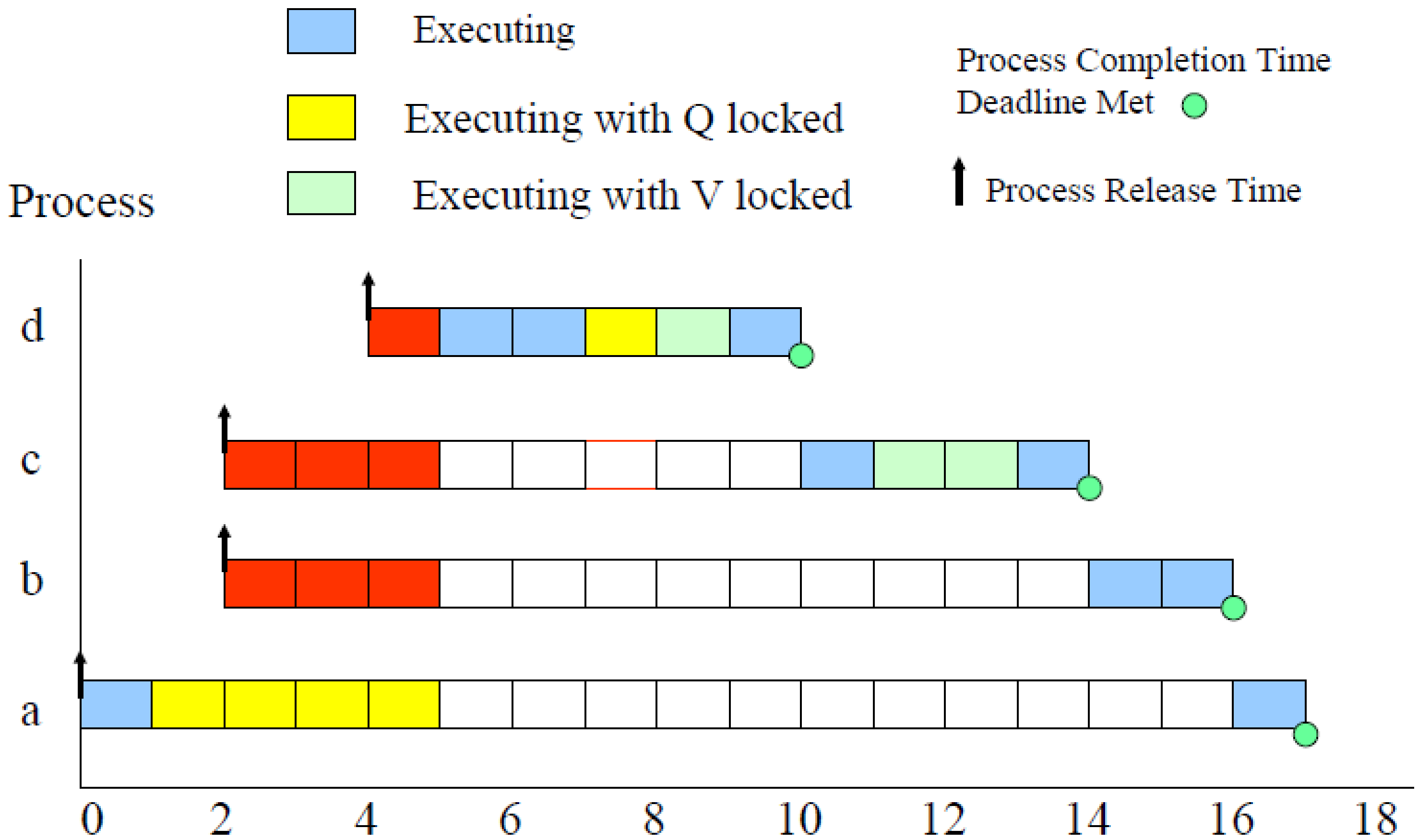


ICPP

- Each process has a static default priority
- Each resource has a static ceiling value
- Process has a dynamic priority that is:
  - *Max*(own priority, **ceiling value** of any **resource** it has locked)
  - Once the process starts executing, all the resources it requires must be free
  - If not, then the process executing on the required resource either has the same or higher priority (& this process must wait)

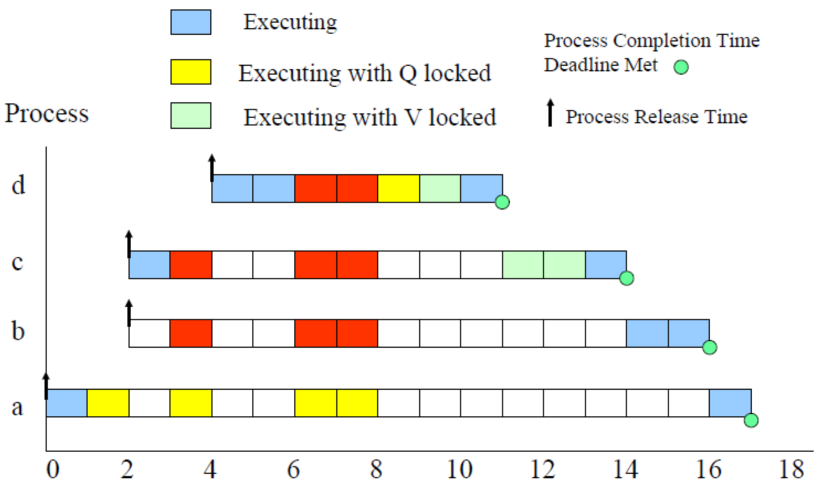
ICPP

Process	Priority	Execution Sequence	Release Time
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4



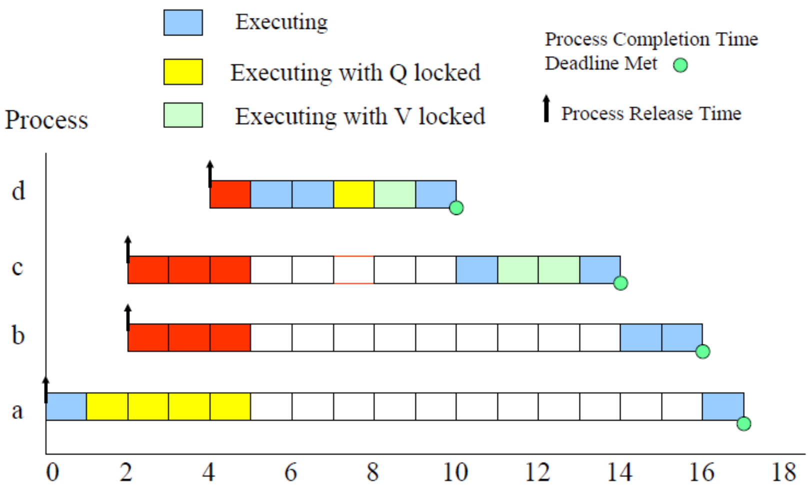
ICPP

Process	Priority	Execution Sequence	Release Time
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4



← OCPP

ICPP →



OCPP versus ICPP

ICPP

- Easier to implement
- Less context switches

- Requires more priority movements (happens with all resource usages)

OCPP

- Harder (must monitor various processes and their blocking effects on other processes)

- Changes priority only if an actual block occurs during execution

Going back to Response Time...

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

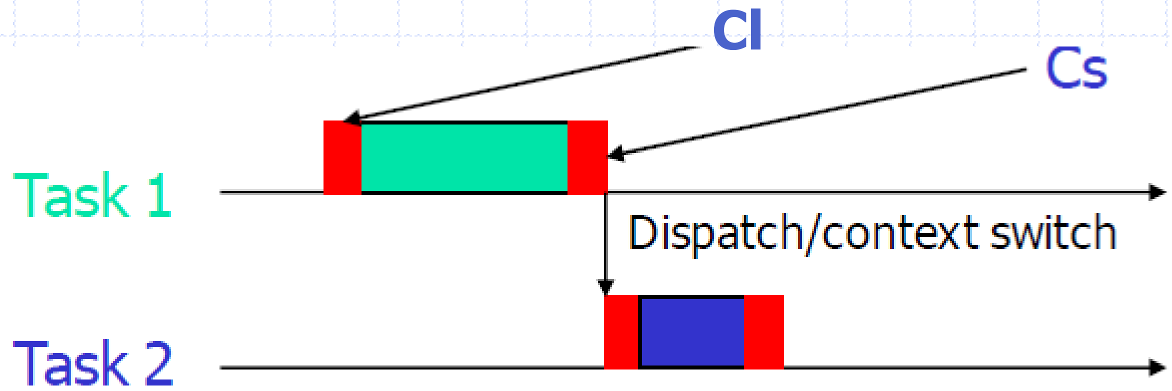
- What are we not taking into consideration in this calculation/methodology?

Going back to Response Time…

- What are we not taking into consideration in this calculation/methodology?
  - More Blocking! (other task critical sections of code (non-preemptive) etc)
  - Co-operative scheduling between the tasks (i.e. Raising priorities, process interactions, interrupts, context switch latencies etc)
  - Release jitter, offsets etc
  - Arbitrary deadlines
  - Optimal priority assignments
  - Fault tolerance

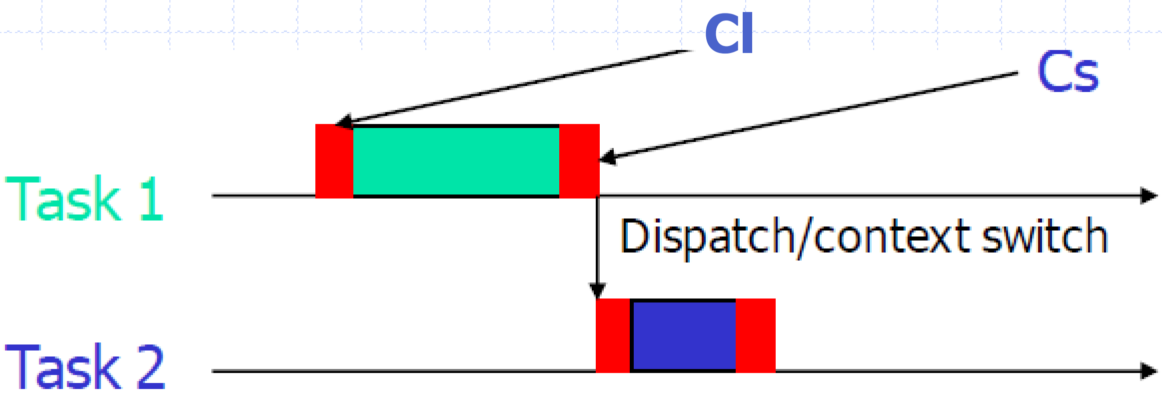
$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Handling Context Switches



Cl – extra time required to load the context of the new task
Cs – extra time required to save the context of the current task

Handling Context Switches



$$R_i = \boxed{C_i} + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \boxed{C_j}$$

$R_i = C_i + 2C_{cs} + \sum_{j \in HP(i)} \lceil R_i/T_j \rceil * C_j$

$+ \sum_{j \in HP(i)} \lceil R_i/T_j \rceil * 4C_{cs}$

(each preemption →2 context switches)

$= C_i + 2C_{cs} + \sum_{j \in HP(i)} \lceil R_i/T_j \rceil * (C_j + 4C_{cs})$

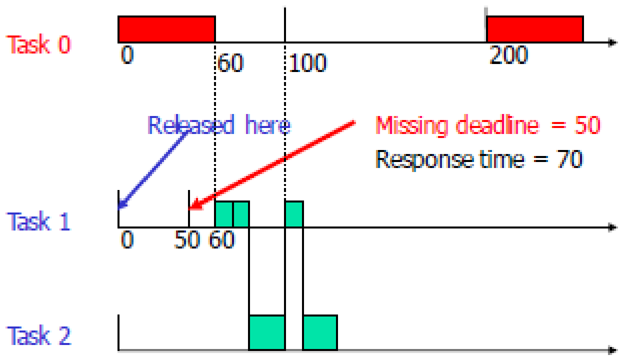
-Will affect the computation time, i.e. Every computation time must add 2 context switch delays

-And all PREEMPTED response times must account for its own context switch, plus the context switch of the other tasks!

Handling Interrupts

Task 0 is the interrupt handler with highest priority

	C	T=D
IH, task0	60	200
Task 1	10	50
Task 2	40	250



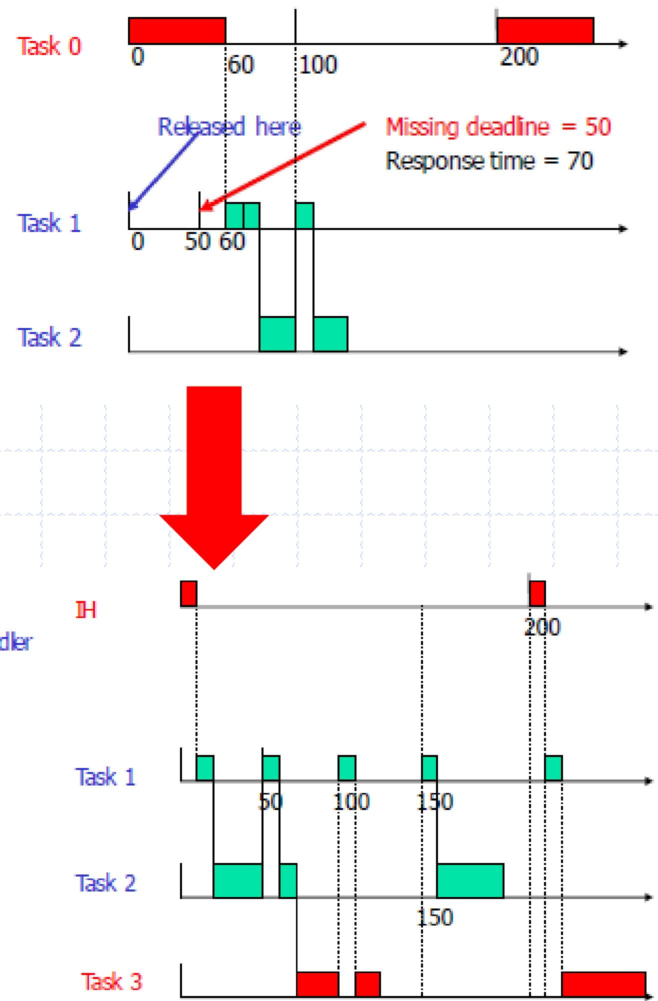
-Interrupt handling can be inconsistent, therefore can affect schedulability of a task set, delay other task periods, especially with shorter periods

-**Whenever possible**: move code from interrupt handler to a “special” task with the same rate as the interrupt handler (assuming periodic interrupt)

Handling Interrupts

Task 0 is the interrupt handler with highest priority

	C	T=D
IH, task0	60	200
Task 1	10	50
Task 2	40	250



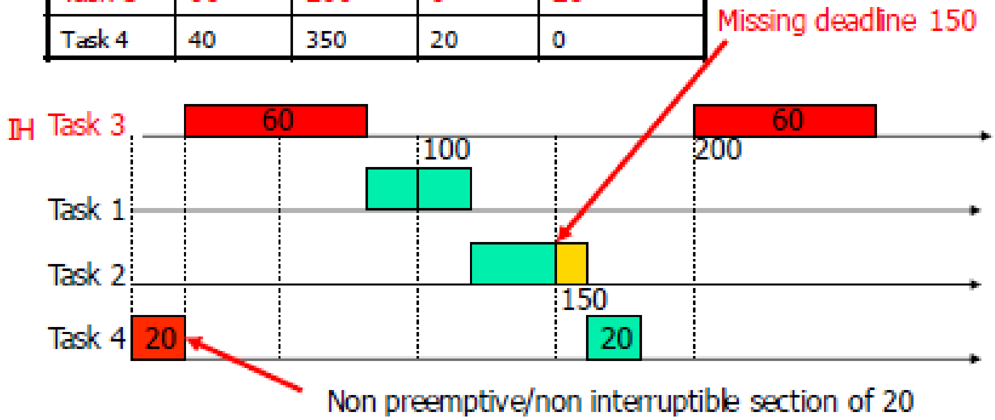
Task 0 is the interrupt handler with highest priority

	C	T=D
IH	10	200
Task 1	10	50
Task 2	40	150
Task 3	50	200

Handling Non-Preemptive Sections

Task 3 is an interrupt handler with highest priority
Task 4 has a non preemptive section of 20 sec

	C	T=D	blocking	blocked
Task 1	20	100	0	20
Task 2	40	150	0	20
Task 3	60	200	0	20
Task 4	40	350	20	0



Handling Non-Preemptive Sections

$$Ri = Bi + Ci + \sum_{j \in HP(i)} \lceil Ri/Tj \rceil * Cj$$

- Not always the case that all regions of code are preemptive
- Bi is the longest time that task i can be blocked by a lower-priority's non-preemptive section

$$Ri = Bi + Ci + 2Ccs + \sum_{j \in HP(i)} \lceil Ri/Tj \rceil * (Cj + 4*Ccs)$$

Word Exercises in RT Scheduling

- Consider 3 processes: P, Q, and S. P has a period of 100ms, in which it requires 30ms of processing. The corresponding values for Q and S are (1,6) and (5, 25) respectively. Assume that P is the most important process in the system, followed by Q and then S

1. What is the behaviour of the scheduler if priority is based on importance?
2. What is the process utilization of P, Q, S?
3. How should the processes be scheduled so that all deadlines are met?

Sitara AM335x Dev Board