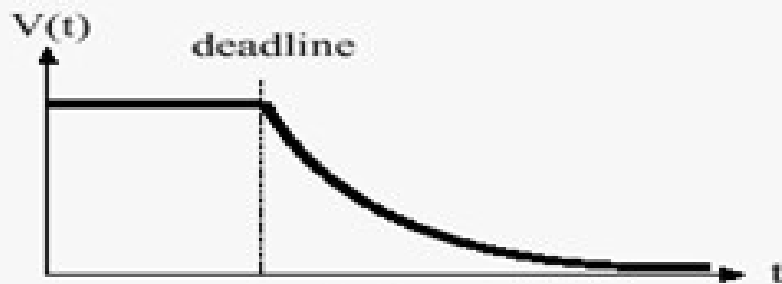


# Real Time Systems

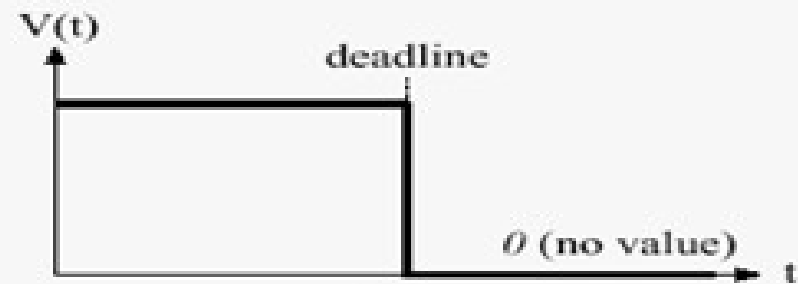
**Real-Time Systems**

Jane W. S. Liu, University of Illinois at Urbana-Champaign

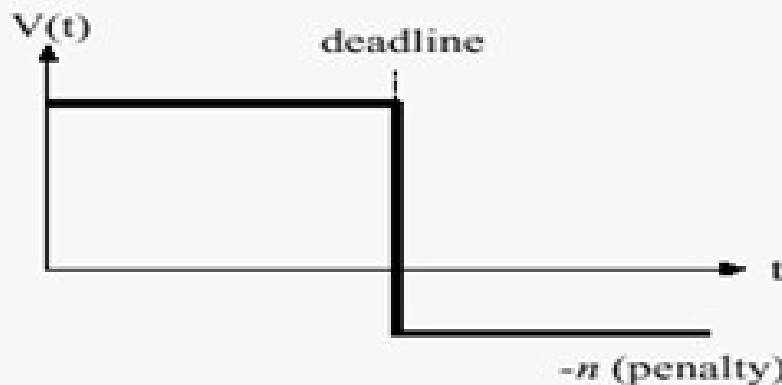
# Real Time Types



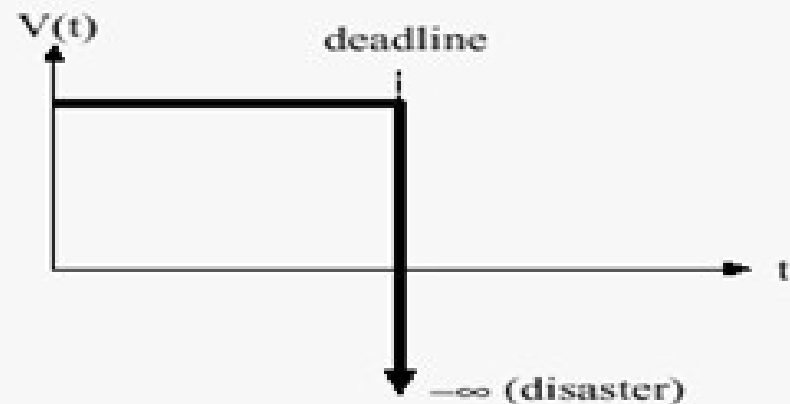
(a) Soft



(b) Firm



(c) Hard essential



(d) Hard critical

Fig. 1. Deadlines represented with value functions.

# Real-Time Is Not Fair

- Main goal of an RTOS scheduler is to meet task deadlines
- If you have five homework assignments and only one is due in half an hour, you work on that one first
- Fairness does not help you meet deadlines

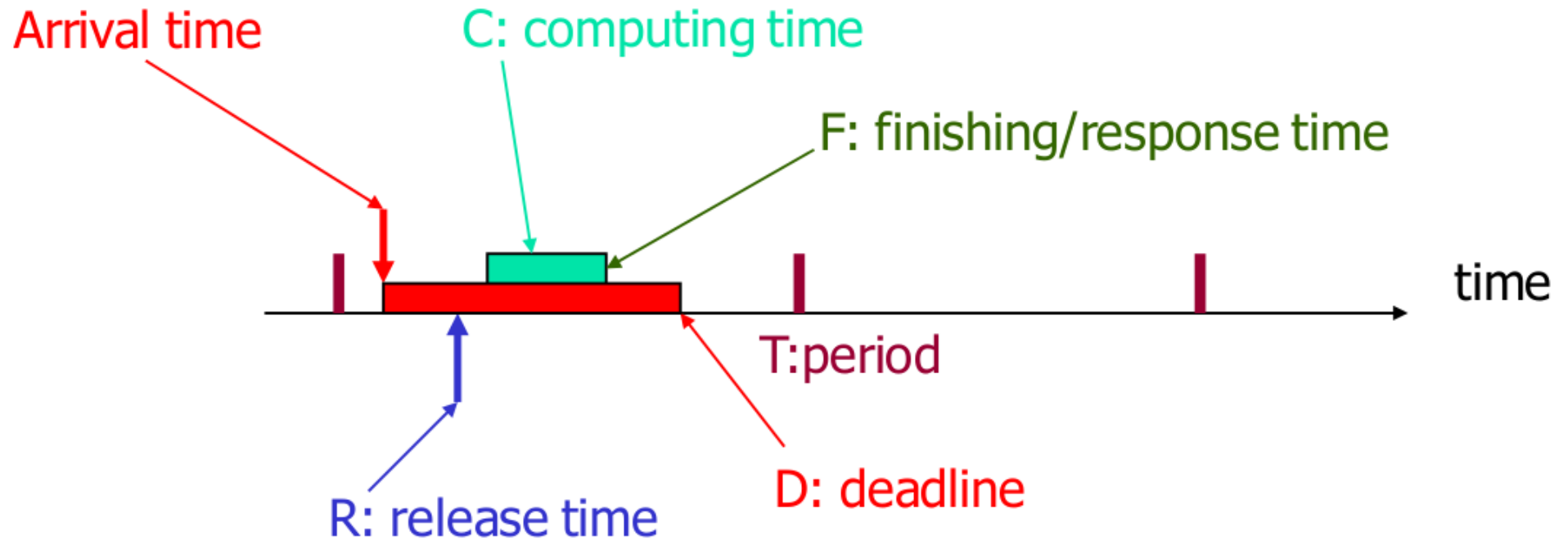
# Real time is not necessarily fast

- Fast means low average latency
- Real time needs predictable worst case performance
- One needs to mathematically prove that all deadlines will be met even in the worst case
  - For uniprocessor system and periodic tasks, mathematical proofs were given in 1970-1990. Now one has to perform some schedulability tests, and if the tests pass, the deadlines are guaranteed to be met.
  - Linux real time patch is designed to be real time, but is not real time, as the code base is too complex for giving mathematical guarantees
  - It has a number of changes to improve reliability/predictability

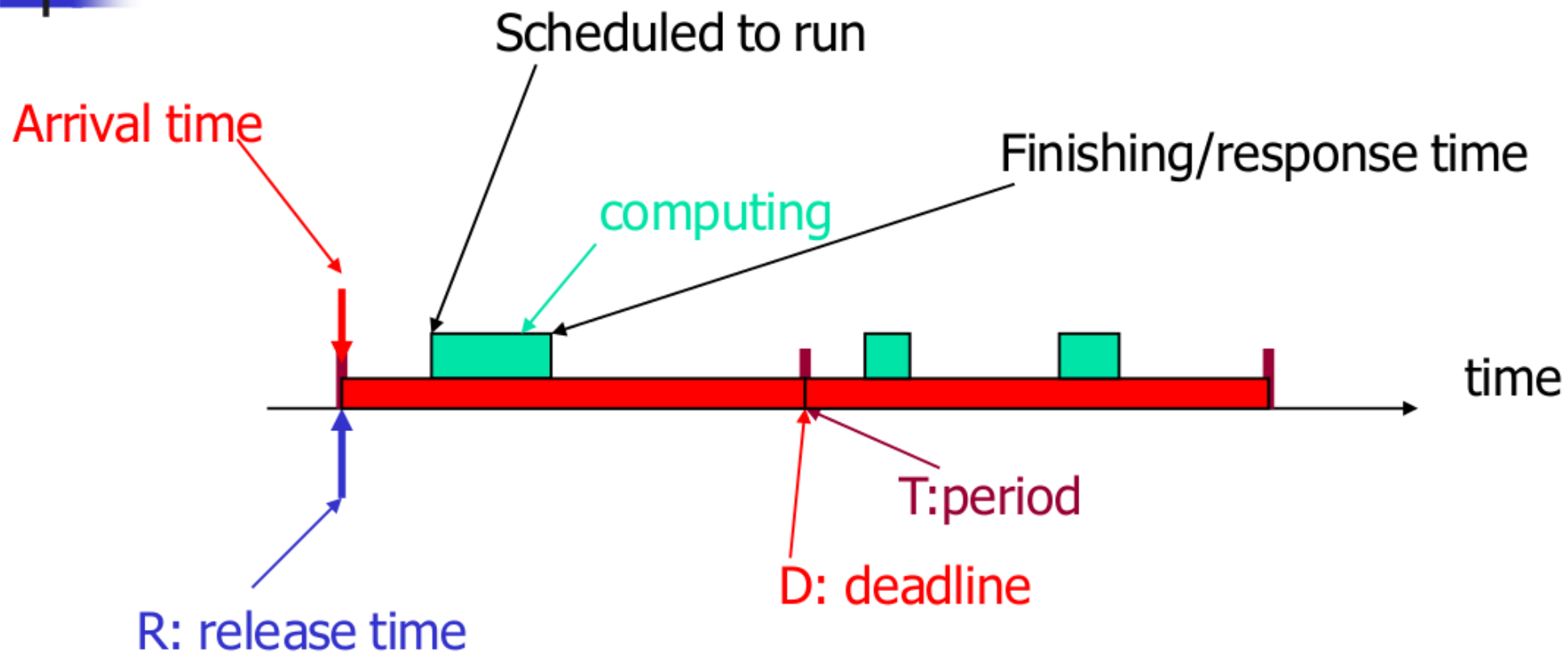
# Other real time system classifications

- Based on where it runs
  - **Uniprocessor**
  - Multicore
  - Distributed
- Based on time characteristics of tasks
  - **Periodic**
  - Aperiodic
  - Sporadic
- Based on when scheduling decisions are taken
  - Clock based
  - Event based
  - Hybrid

# Periodic tasks



# Periodic tasks (the simplified case)





# Periodic task model

---

- A task =  $(C, T)$ 
  - $C$ : worst case execution time/computing time ( $C \leq T$ !)
  - $T$ : period ( $D = T$ )
  
- A task set:  $(C_i, T_i)$ 
  - All tasks are independent
  - The periods of tasks start at 0 simultaneously



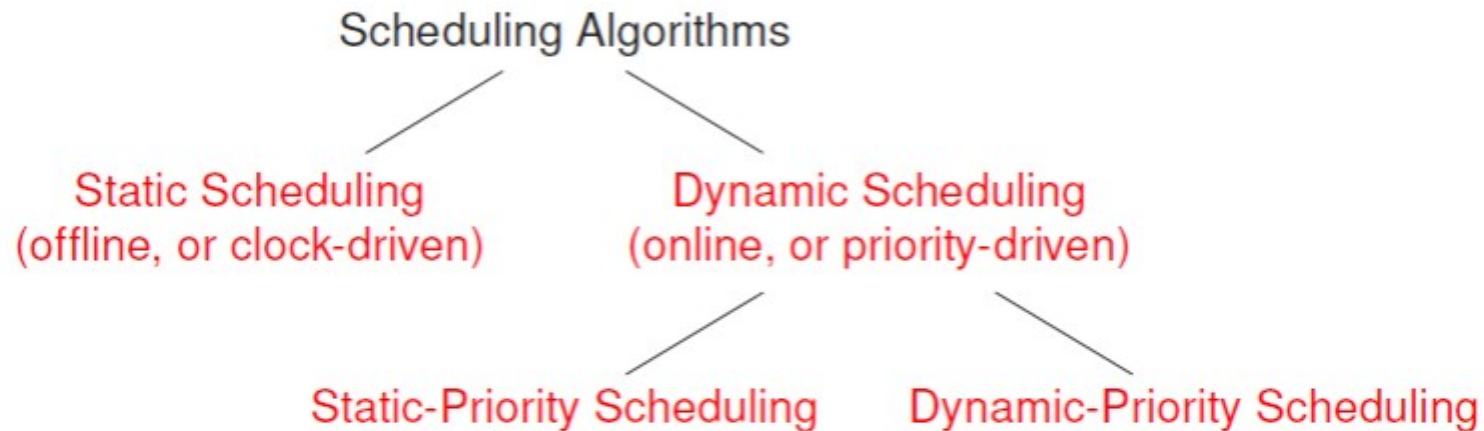


## CPU utilization

---

- $C/T$  is the CPU utilization of a task
- $U = \sum(C_i/T_i)$  is the CPU utilization of a task set
- Note that the CPU utilization is a measure on how busy the processor could be during the **shortest repeating cycle**:  
 $T_1 * T_2 * \dots * T_n$ 
  - $U > 1$  (overload): some task will fail to meet its deadline no matter what algorithms you use!
  - $U \leq 1$ : it will depend on the scheduling algorithms
    - If  $U = 1$  and the CPU is kept busy (non idle algorithms e.g. EDF), all deadlines will be met

# Scheduling Algorithms



- Preemptive vs. Non-preemptive
- Guarantee-Based vs. Best-Effort
- Optimal vs. Non-optimal



## Static cyclic scheduling

---

- Shortest repeating cycle = least common multiple (LCM)
- Within the cycle, it is possible to construct a static schedule i.e. a time table
- Schedule task instances according to the time table within each cycle



## Example: the Car Controller

---

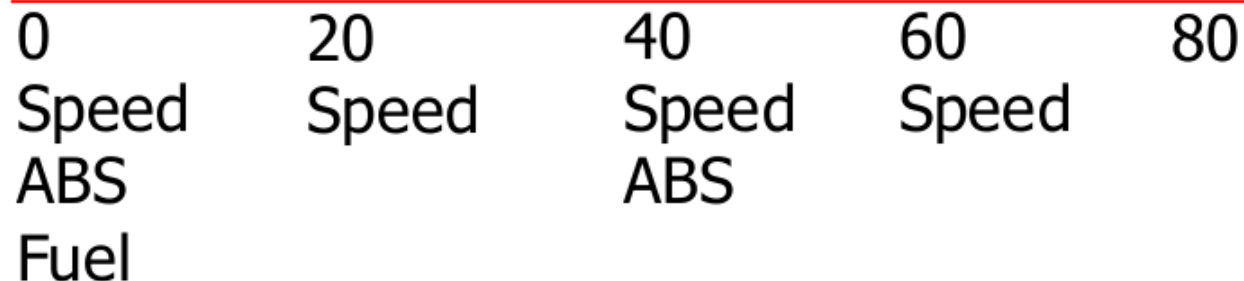
Activities of a car control system. Let

1.  $C$  = worst case execution time
  2.  $T$  = (sampling) period
  3.  $D$  = deadline
- Speed measurement:  $C=4\text{ms}$ ,  $T=20\text{ms}$ ,  $D=20\text{ms}$
  - ABS control:  $C=10\text{ms}$ ,  $T=40\text{ms}$ ,  $D=40\text{ms}$
  - Fuel injection:  $C=40\text{ms}$ ,  $T=80\text{ms}$ ,  $D=80\text{ms}$
  - Other software with soft deadlines e.g audio, air condition etc



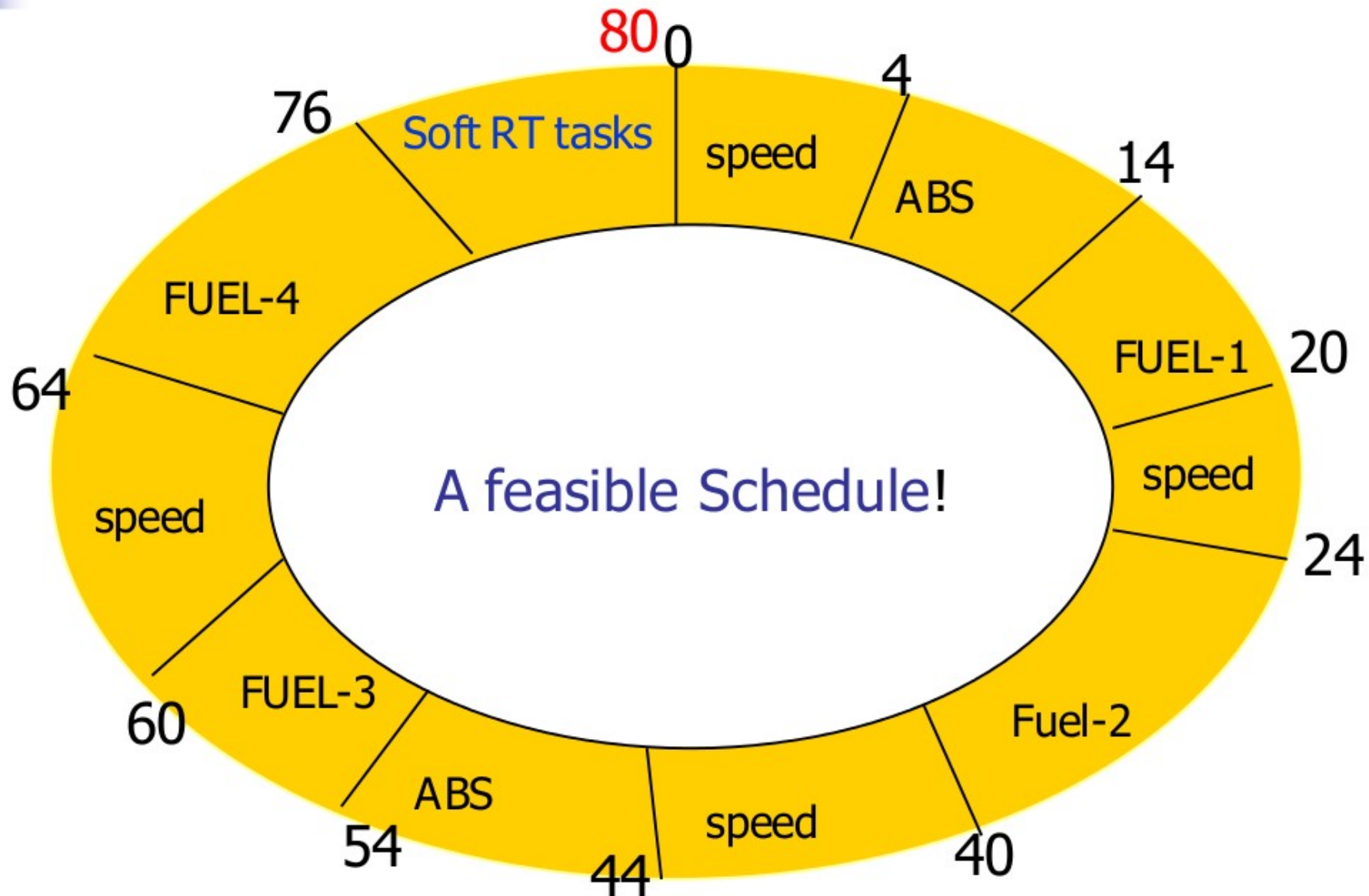
## The car controller: static cyclic scheduling

- The shortest repeating cycle = 80ms
- All task instances within the cycle:



- Try any method to schedule the tasks

# The car controller: time table constructed with EDF





## Static cyclic scheduling: + and –

---

- Deterministic: predictable (+)
- Easy to implement (+)
- Inflexible (-)
  - Difficult to modify, e.g adding another task
  - Difficult to handle external events
- The table can be huge (-)
  - Huge memory-usage
  - Difficult to construct the time table



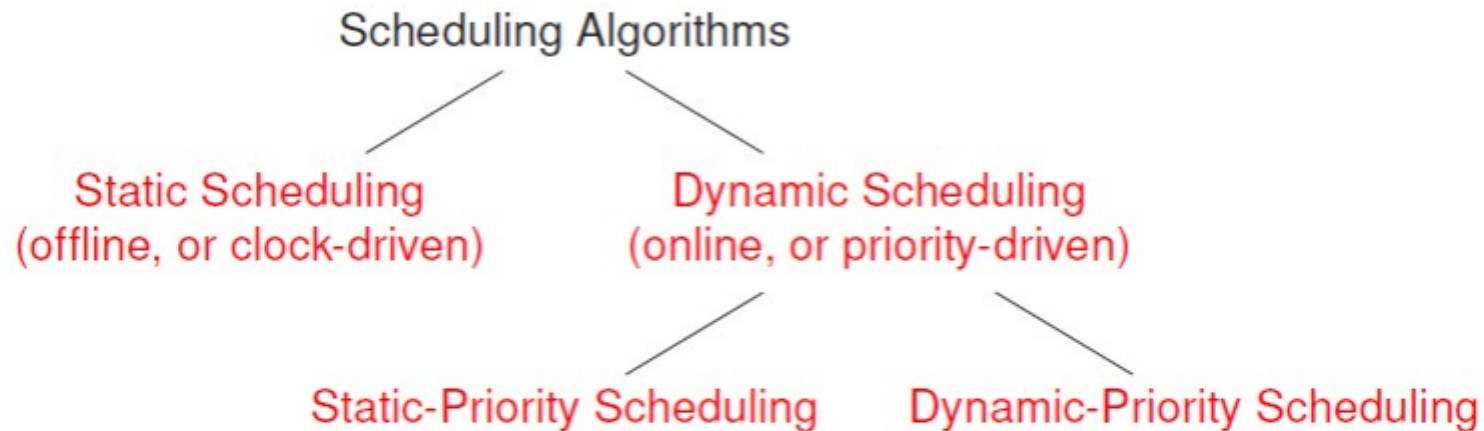
## Example: shortest repeating cycle

---

- OBS: The LCM determines the size of the time table
  - LCM = 50ms for tasks with periods: 5ms, 10ms and 25ms
  - LCM =  $7 * 13 * 23 = 2093$  ms for tasks with periods: 7ms, 13ms and 23ms (very much bigger)
- So if possible, manipulate the periods so that they are multiples of each other
  - Easier to find a feasible schedule and
  - Reduce the size of the static schedule, thus less memory usage



# Scheduling Algorithms



- Preemptive vs. Non-preemptive
- Guarantee-Based vs. Best-Effort
- Optimal vs. Non-optimal

# Fixed versus Dynamic Priority Algorithms

---

- A fixed-priority algorithm assigns the same priority to all the jobs in each task.
- A dynamic-priority algorithm assigns different priorities to the individual jobs in each task.
- Most real-time scheduling algorithms of practical interests assign job-level fixed priorities.

# Fixed-Priority Algorithms

---

- Rate-monotonic (RM) algorithm: It assigns priorities to tasks based on their periods: the shorter the period, the higher the priority. Hence, the higher the rate, the higher the priority.
- Deadline-monotonic (DM) algorithm: It assigns priorities to jobs according to their relative deadlines: the shorter the relative deadline, the higher the priority.

# EDF Algorithm

---

- Earliest-Deadline-First (EDF) algorithm assigns priorities to jobs according to their deadlines. The earlier the deadline, the higher the priority.
- This algorithm is optimal when used to schedule jobs on a processor as long as preemption is allowed and jobs do not contend for resources.
- Definition of “optimal”: can produce a feasible schedule of a set of jobs with arbitrary release times and deadlines on a processor if a feasible schedule exists.



# LST Algorithm

---

- Least-Slack-Time-First (LST) algorithm, a.k.a. Minimum-Laxity-First (MLF) algorithm, assigns priorities to jobs based on their slacks: the smaller the slack, the higher the priority.
- At any time  $t$ , the slack (or laxity) of a job with deadline at  $d$  is equal to  $d - t$  minus the time required to complete the remaining portion of the job.
- EDF and LST algorithms are optimal only when preemption is allowed.

# Schedulability Tests



# Earliest Deadline First (EDF)

---

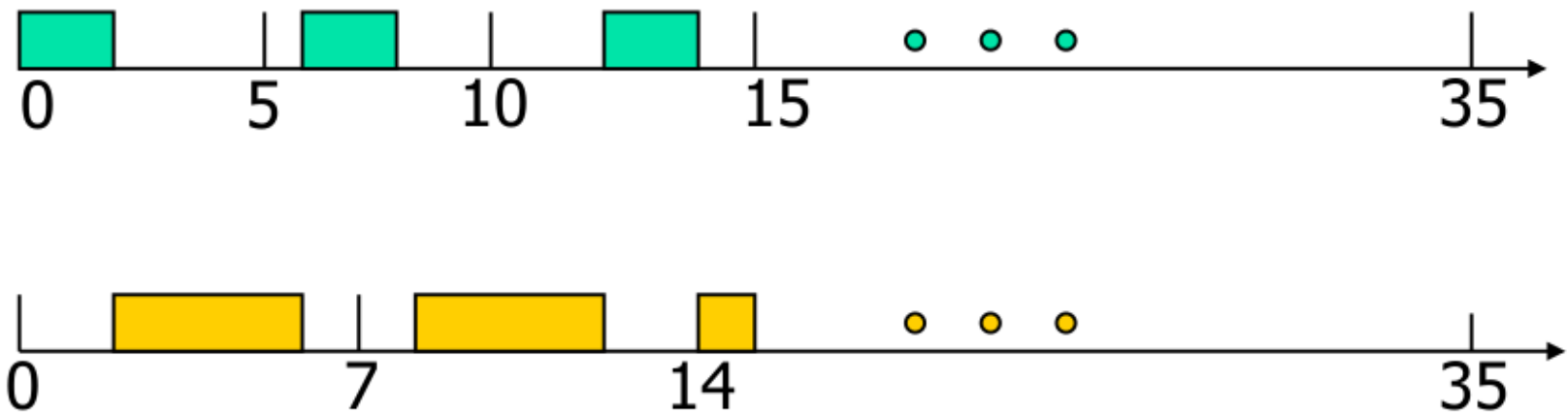
- Task model
  - a set of independent periodic tasks (not necessarily the simplified task model)
- EDF:
  - Whenever a new task arrive, sort the ready queue so that the task closest to the end of its period assigned the highest priority
  - Preempt the running task if it is not placed in the first of the queue in the last sorting
- **FACT 1:** EDF is optimal
  - EDF can schedule the task set if any one else can
- **FACT 2** (Scedulability test):
  - $\sum C_i/T_i \leq 1$  iff the task set is schedulable



## Example

---

- Task set:  $\{(2,5),(4,7)\}$
- $U = 2/5 + 4/7 = 34/35 \sim 0.97$  (schedulable!)








## EDF: + and -

Data structure space and insertion times are too high for typical embedded real time platforms and applications

- Note that this is just the simple EDF algorithm; it works for all types of tasks: periodic or non periodic
    - It is simple and works nicely in theory (+)
    - Simple schedulability test:  $U \leq 1$  (+)
    - Optimal (+)
    - Best CPU utilization (+)
  - **Difficult to implement** in practice. It is not very often adopted due to the dynamic priority-assignment (expensive to sort the ready queue on-line), which has nothing to do with the periods of tasks. Note that Any task could get the highest priority (-)
  - **Non stable**: if any task instance fails to meet its deadline, the system is not predictable, any instance of any task may fail (-)
- 



## Rate Monotonic Scheduling: task model

---

Assume a set of periodic tasks:  $(C_i, T_i)$

- $D_i = T_i$
- Tasks are always released at the start of their periods
- Tasks are independent



## RMS: fixed/static-priority scheduling

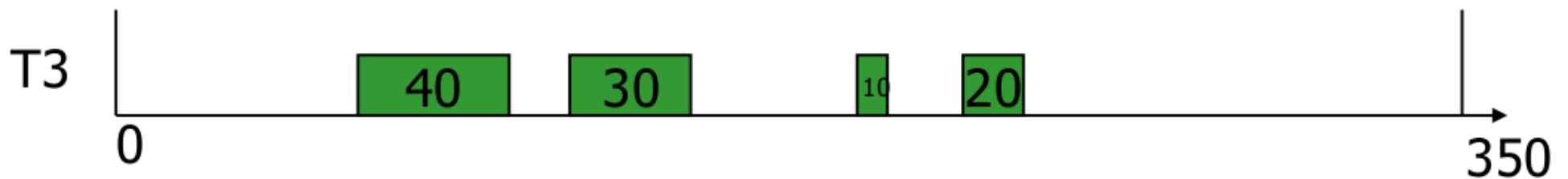
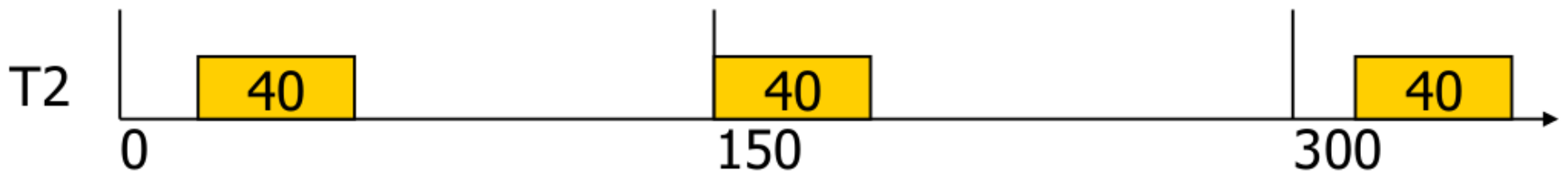
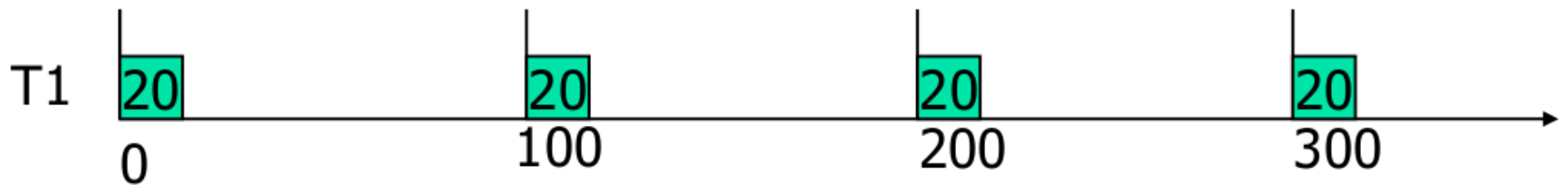
---

- Rate Monotonic Fixed-Priority Assignment:
  - Tasks with smaller periods get higher priorities
- Run-Time Scheduling:
  - Preemptive highest priority first
- **FACT:** RMS is optimal in the sense:
  - If a task set is schedulable with any **fixed-priority** scheduling algorithm, it is also schedulable with RMS

# Example

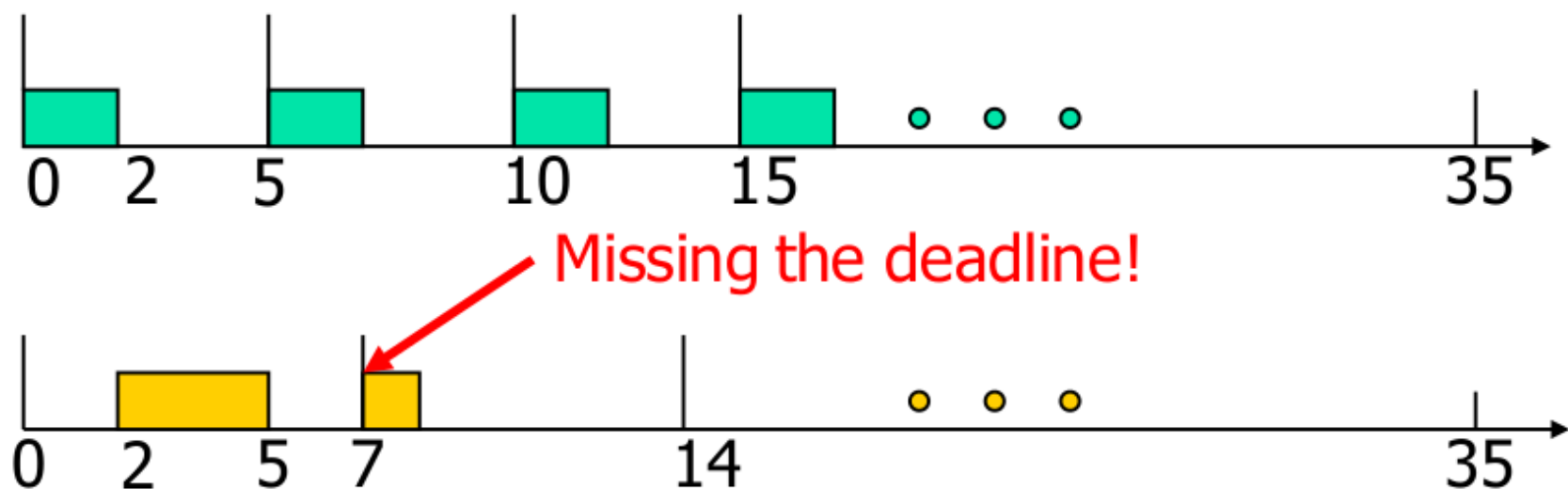
$\{(20,100),(40,150),(100,350)\}$

$\text{Pr}(T1)=1, \text{Pr}(T2)=2, \text{Pr}(T3)=3$



# Example

- Task set:  $T1=(2,5)$ ,  $T2=(4,7)$
- $U = 2/5 + 4/7 = 34/35 \sim 0.97$  (schedulable?)
- RMS priority assignment:  $Pr(T1)=1$ ,  $Pr(T2)=2$





## The famous **Utilization Bound test** (UB test) [by Liu and Layland, 1973: a classic result]

---

- Assume a set of  $n$  independent tasks:
  - $S = \{(C_1, T_1)(C_2, T_2) \dots (C_n, T_n)\}$  and  $U = \sum C_i/T_i$
- **FACT:** if  $U \leq n \cdot (2^{1/n} - 1)$ , then  $S$  is schedulable by RMS
- Note that the bound depends only on the size of the task set



## Example: Utilization bounds

---

$B(1)=1.0$	$B(4)=0.756$	$B(7)=0.728$
$B(2)=0.828$	$B(5)=0.743$	$B(8)=0.724$
$B(3)=0.779$	$B(6)=0.734$	$U(\infty)=0.693$

Note that  $U(\infty)=0.693$  !



## Example: applying UB Test

	C	T (D=T)	C/T
Task 1	20	100	0.200
Task 2	40	150	0.267
Task 3	100	350	0.286

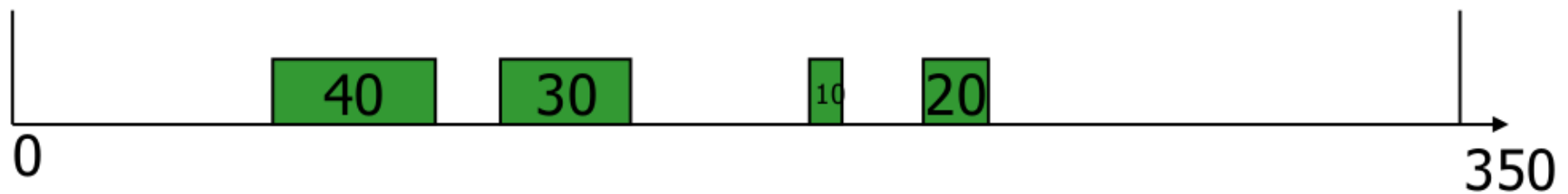
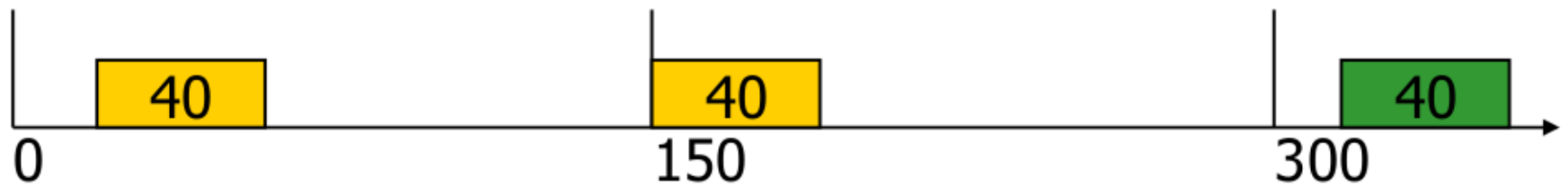
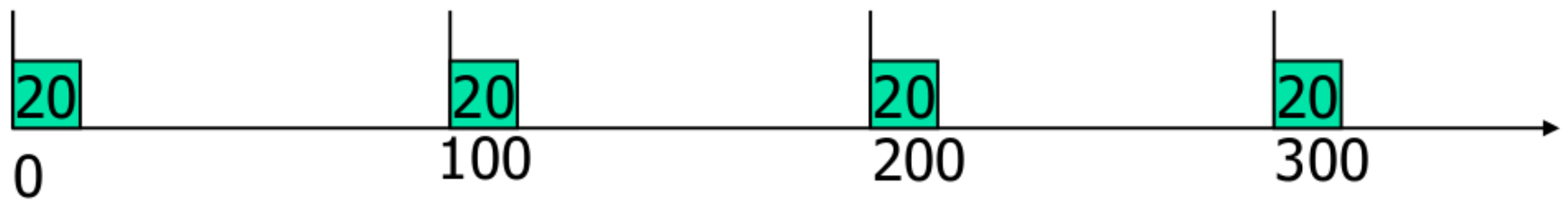
Total utilization:  $U=0.2+0.267+0.286=0.753 < B(3)=0.779!$

The task set is schedulable



# Example: RM Scheduling

$\{(20,100),(40,150),(100,350)\}$





## UB test is only **sufficient, not necessary!**

---


- Let  $U = \sum C_i/T_i$  and  $B(n) = n \cdot (2^{1/n} - 1)$
- Three possible outcomes:
  - $0 \leq U \leq B(n)$ : **schedulable**
  - $B(n) < U \leq 1$ : **no conclusion**
  - $1 < U$ : **overload**
- Thus, the test may be too conservative
- (exact test will be given later)



## Example: UB test is sufficient, not necessary

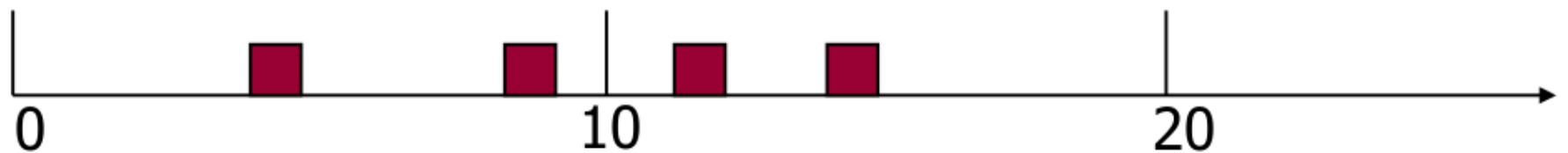
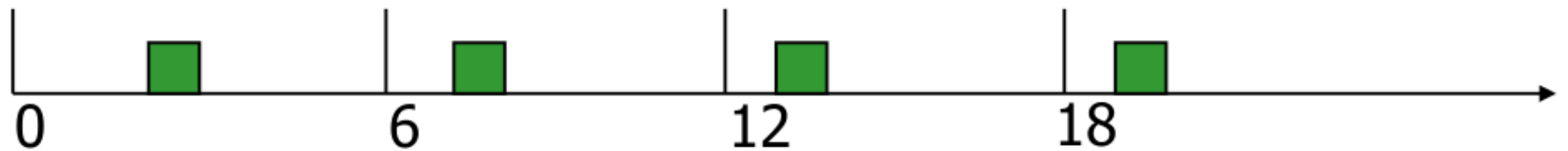
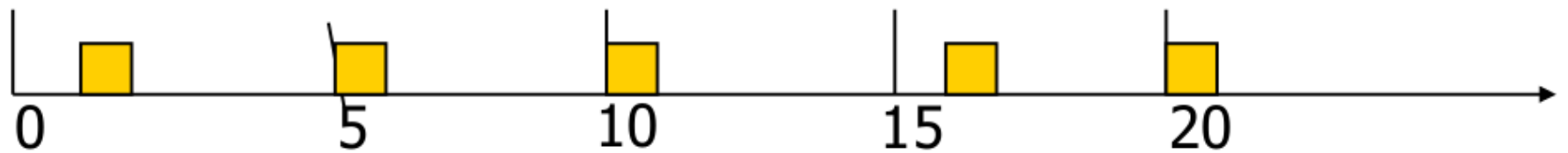
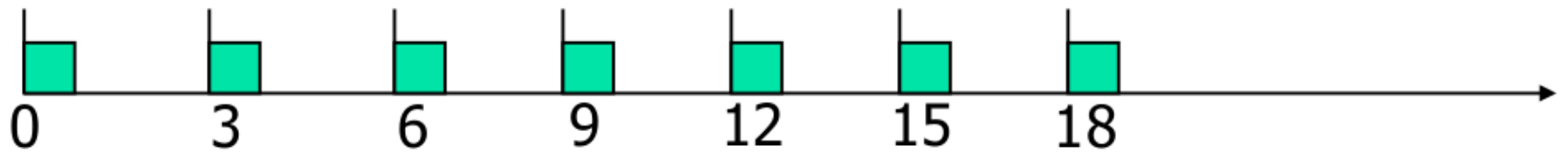
---

- Assume a task set:  $\{(1,3),(1,5),(1,6),(2,10)\}$
- CPU utilization  $U = 1/3 + 1/5 + 1/6 + 2/10 = 0.899$
- The utilization bound  $B(4) = 0.756$
- The task set fails in the UB test due to  $U > B(4)$
- Question: is the task set schedulable?
- Answer: YES



$\{(1,3),(1,5),(1,6),(2,10)\}$

Response times?  
Worst case? First period?  
Why?



This is only for the first periods! But we will see that this is enough to tell that the task set is schedulable.



## RMS: Summary

---

- Task model:
  - periodic, independent,  $D=T$ , and a task =  $(C_i, T_i)$
- Fixed-priority assignment:
  - smaller periods = higher priorities
- Run time scheduling: Preemptive HPF
- Sufficient schedulability test:  $U \leq n \cdot (2^{1/n} - 1)$
- Precise/exact schedulability test exists



## RMS: + and -

---

- Simple to understand (and remember!) (+)
- Easy to implement (static/fixed priority assignment)(+)
- **Stable**: though some of the lower priority tasks fail to meet deadlines, others may meet deadlines (+)
  
- "lower" CPU utilization (-)
- Requires  $D=T$  (-)
- Only deal with independent tasks (-)
- **Non-precise schedulability analysis** (-)
- But these are not really disadvantages; they can be fixed (+++)
  - We can solve all these problems except "lower" utilization



## Critical instant: an important observation

---

- Note that in our examples, we have assumed that all tasks are released at the same time: this is to consider the critical instant (the worst case scenario)
  - If tasks meet the first deadlines (the first periods), they will do so in the future (why?)
- **Critical instant** of a task is the time at which the release of the task will yield the largest response time. It occurs when the task is released simultaneously with higher priority tasks
- Note that the start of a task period is not necessarily the same as any of the other periods: but the delay between two releases should be equal to the constant period (otherwise we have jitters)



## Sufficient and necessary schedulability analysis

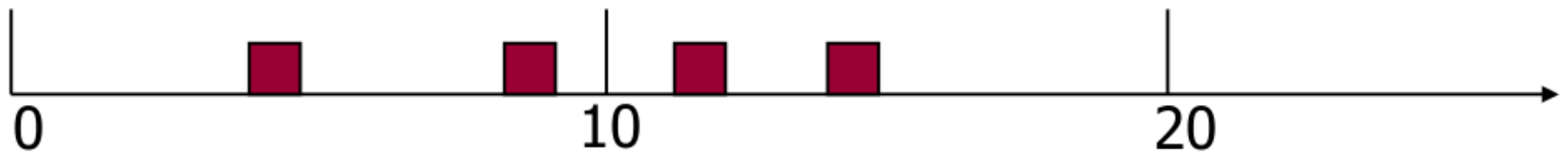
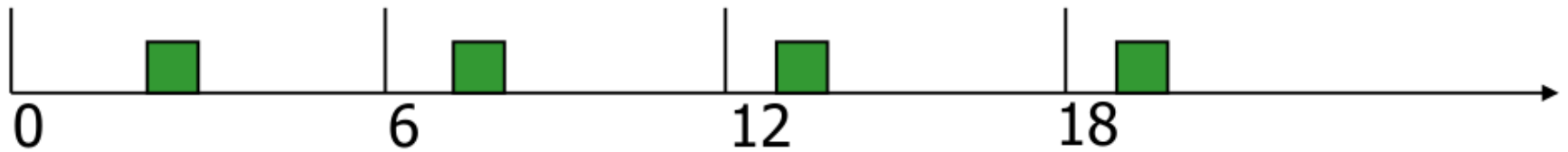
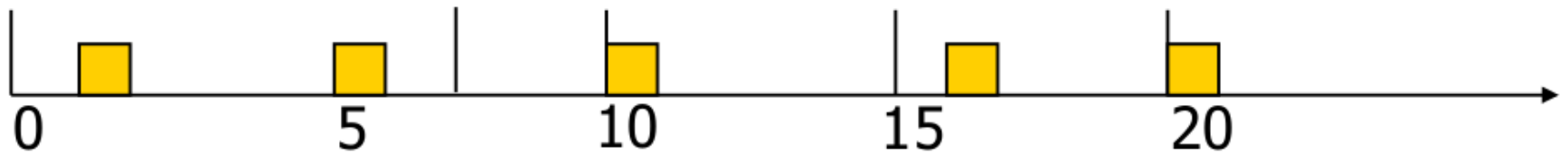
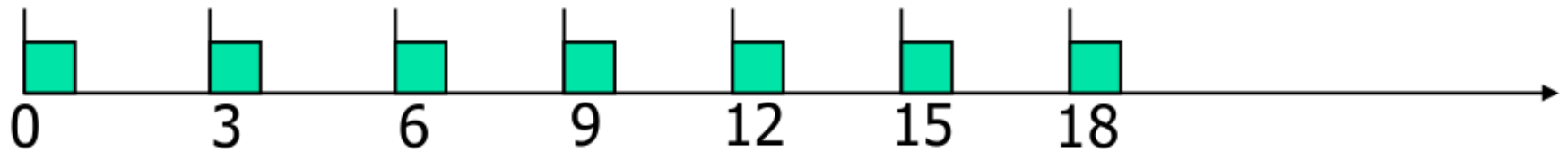
- **Simple ideas** [Mathai Joseph and Paritosh Pandya, 1986]:
  - **Critical instant**: the worst case response time for all tasks is given when all tasks are released at the same time
  - Calculate the worst case response time  $R$  for each task with deadline  $D$ . If  $R \leq D$ , the task is schedulable/feasible. Repeat the same check for all tasks
  - If all tasks pass the test, the task set is schedulable
  - If some tasks pass the test, they will meet their deadlines even the other don't (**stable and predictable**)
- **Question:**
  - how to calculate the worst case response times?



# Worst case response time calculation: example

$\{(1,3),(1,5),(1,6),(2,10)\}$

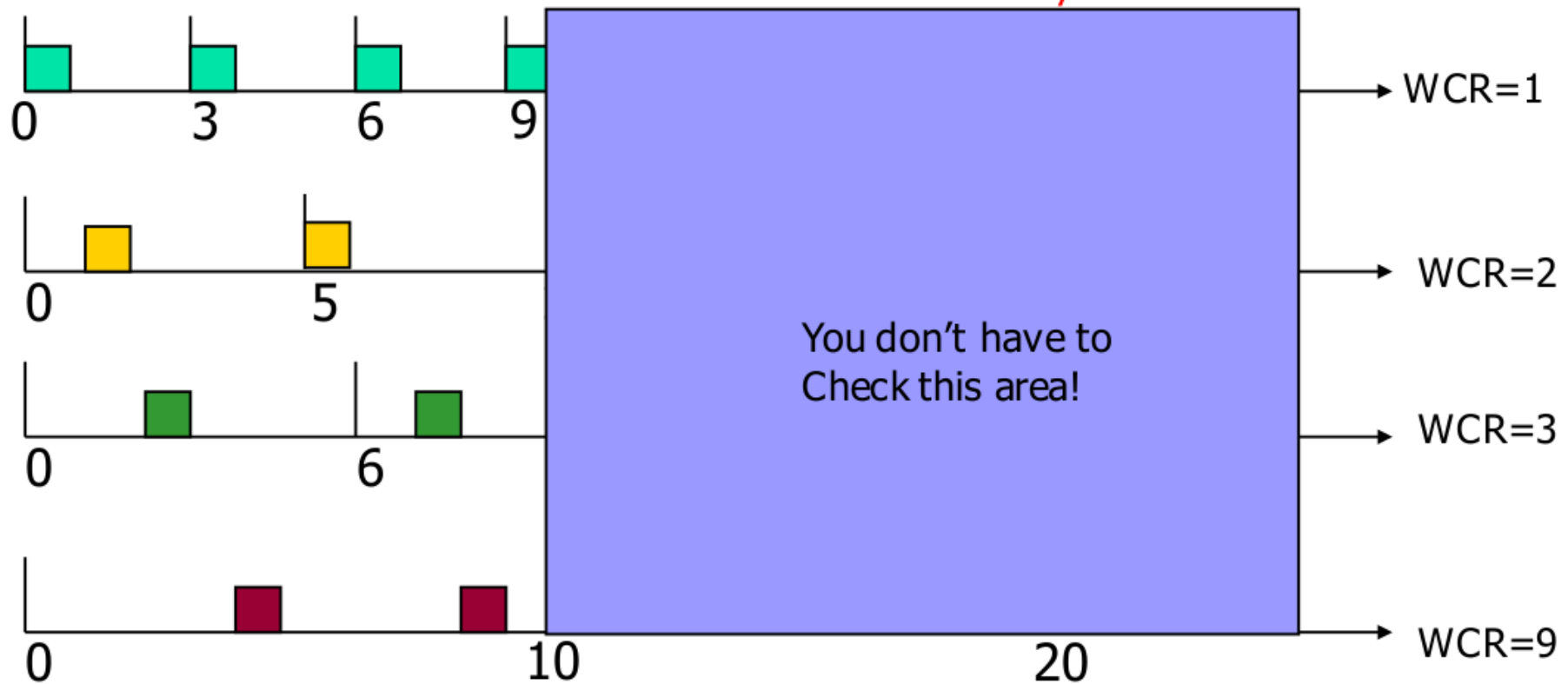
Response times?  
Worst case? First period?  
Why?



# Worst case response time calculation: example

$\{(1,3),(1,5),(1,6),(2,10)\}$

Response times?  
Worst case? First period?  
Why?





# Calculation of worst case response times

[Mathai Joseph and Paritosh Pandya, 1986]

---

- Let  $R_i$  stand for the response time for task  $i$ . Then
$$R_i = C_i + \sum_j I(i,j)$$
  - $C_i$  is the computing time
  - $I(i,j)$  is the so-called **interference** of task  $j$  to  $i$
  - $I(i,j) = 0$  if task  $i$  has higher priority than  $j$
- $I(i,j) = \lceil R_i/T_j \rceil * C_j$  if task  $i$  has lower priority than  $j$ 
  - $\lceil x \rceil$  denotes the least integer larger than  $x$
  - E.g  $\lceil 3.2 \rceil = 4, \lceil 3 \rceil = 3, \lceil 1.9 \rceil = 2$
- $R_i = C_i + \sum_{j \in \text{HP}(i)} \lceil R_i/T_j \rceil * C_j$



## Intuition on the equation

---

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

- $\lceil R_i / T_j \rceil$  is the number of instances of task  $j$  during  $R_j$
- $\lceil R_i / T_j \rceil * C_j$  is the time needed to execute all instances of task  $j$  released within  $R_j$
- $\sum_{j \in \text{HP}(i)} \lceil R_i / T_j \rceil * C_j$  is the time needed to execute instances of tasks with higher priorities than task  $i$ , released during  $R_j$
- $R_j$  is the sum of the time required for executing task instances with higher priorities than task  $j$  and its own computing time



## Equation solving and schedulability analysis

---

- We need to solve the equation:

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

- This can be done by numerical methods to compute the fixed point of the equation e.g. By iteration: let
  - $R_i^0 = C_i + \sum_{j \in \text{HP}(i)} C_j = C_1 + C_2 + \dots + C_i$  (the first guess)
  - $R_i^{k+1} = C_i + \sum_{j \in \text{HP}(i)} \lceil R_i^k / T_j \rceil * C_j$  (the (k+1)th guess)
- The iteration stops when either
  - $R_i^{m+1} > T_i$  or  $\rightarrow$  non schedulable
  - $R_i^m < T_i$  and  $R_i^{m+1} = R_i^m \rightarrow$  schedulable
- This is the so called **Precise test**



## Combine UB and Precise tests

---

- **Order tasks** according to their priorities (periods)
- Use UB test as far as you can until you **find the first non-schedulable task**
- Calculate response time for the task and **all the tasks with lower priority**



## Example

---

	C	T	C/T
Task 1	40	100	0.400
Task 2	40	150	0.267
Task 3	100	350	0.286

Total utilization:  $U=0.4+0.267+0.286=0.953 > B(3)=0.779!$

UB test is inclusive: we need Precise test

but we do have  $U(T1)+U(T2)=0.4+0.267=0.667 < U(2)=0.828$

so we need to calculate R3 only!



## Calculate response time for task 3

---

- $R3^0 = C1 + C2 + C3 = 180$
- $R3^1 = C3 + \lceil R3^0/T1 \rceil * C1 + \lceil R3^0/T2 \rceil * C2$   
 $= 100 + \lceil 180/100 \rceil * 40 + \lceil 180/150 \rceil * 40$   
 $= 100 + 2 * 40 + 2 * 40 = 260$
- $R3^2 = C3 + \lceil R3^1/T1 \rceil * C1 + \lceil R3^1/T2 \rceil * C2$   
 $= 100 + \lceil 260/100 \rceil * 40 + \lceil 260/150 \rceil * 40 = 300$
- $R3^3 = C3 + \lceil R3^2/T1 \rceil * C1 + \lceil R3^2/T2 \rceil * C2$   
 $= 100 + \lceil 300/100 \rceil * 40 + \lceil 300/150 \rceil * 40 = 300$  (done)

Task 3 is schedulable and so are the others!





## Example (combine UB test and precise test)

- Consider the same task set:  $\{(1,3),(1,5),(1,6),(3,10)\}$
- CPU utilization  $U = 1/3 + 1/5 + 1/6 + 3/10 = 0.899 > B(4) = 0.756$ 
  - Fail the UB test!
- But  $U(3) = 1/3 + 1/5 + 1/6 = 0.699 < B(3) = 0.779$ 
  - This means that the first 3 tasks are schedulable
- **Question:** is task 4 set schedulable?
  - $R_4^0 = C_1 + C_2 + C_3 + C_4 = 6$
  - $R_4^1 = C_4 + \lceil R_4^0/T_1 \rceil * C_1 + \lceil R_4^0/T_2 \rceil * C_2 + \lceil R_4^0/T_3 \rceil * C_3$   
 $= 3 + \lceil 6/3 \rceil * 1 + \lceil 6/5 \rceil * 1 + \lceil 6/6 \rceil * 1 = 8$
  - $R_4^2 = C_4 + \lceil R_4^1/T_1 \rceil * C_1 + \lceil R_4^1/T_2 \rceil * C_2 + \lceil R_4^1/T_3 \rceil * C_3$   
 $= 3 + \lceil 8/3 \rceil * 1 + \lceil 8/5 \rceil * 1 + \lceil 8/6 \rceil * 1$   
 $= 3 + 3 + 2 + 2$   
 $= 10$
  - $R_4^3 = C_4 + \lceil R_4^2/T_1 \rceil * C_1 + \lceil R_4^2/T_2 \rceil * C_2 + \lceil R_4^2/T_3 \rceil * C_3$   
 $= 3 + 4 + 2 + 2 = 11$  (task 4 is non schedulable!)



## Summary: Three ways to check schedulability

---

1. UB test (simple but conservative)
2. Response time calculation (precise test)
3. Construct a schedule for the first periods
  - assume the first instances arrive at time 0 (critical instant)
  - draw the schedule for the first periods
  - if all tasks are finished before the end of the first periods, schedulable, otherwise NO



# Deadline Monotonic Scheduling (DMS)

[Leung et al, 1982]

---

- **Task model**: the same as for RMS but  $D_i \leq T_i$
- **Priority-Assignment**: tasks with shorter deadline are assigned higher priorities
- **Run-time scheduling**: preemptive HPF
  
- **FACTS**:
  - DMS is optimal
  - RMS is a special case of DMS
- DMS is often referred as Rate Monotonic Scheduling for historical reasons and they are so similar



## DMS: Schedulability analysis

---

- UB test (sufficient):

$\sum C_i/D_i \leq n \cdot (2^{1/n} - 1)$  implies schedulable by DMS

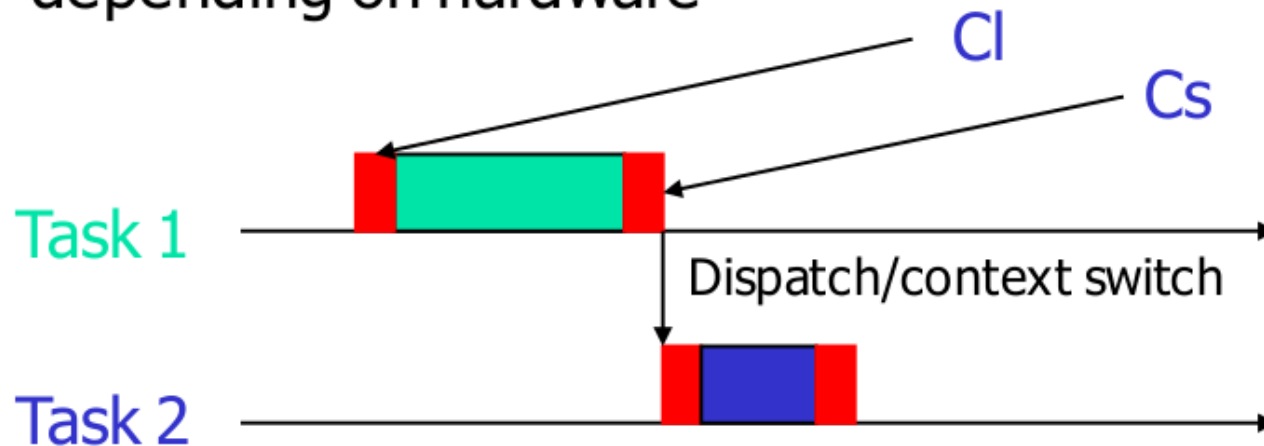
- Precise test (exactly the same as for RMS):

Response time calculation:  $R_i = C_i + \sum_{j \in \text{HP}(i)} \lceil R_i/T_j \rceil \cdot C_j$

- $R_i^0 = C_i + \sum_{j \in \text{HP}(i)} C_j = C_1 + C_2 + \dots + C_i \rightarrow$  the first guess
- $R_i^{k+1} = C_i + \sum_{j \in \text{HP}(i)} \lceil R_i^k/T_j \rceil \cdot C_j \rightarrow$  the (k+1)th guess
- The iteration stops when either
  - $R_i^{m+1} > D_i$  or  $\rightarrow$  non schedulable
  - $R_i^m < D_i$  and  $R_i^{m+1} = R_i^m \rightarrow$  schedulable

# Handling context switch overheads in schedulability analysis

- Assume that
  - $C_l$  is the extra time required to load the context for a new task (load contents of registers etc from TCB)
  - $C_s$  is the extra time required to save the context for a current task (save contents of registers etc to TCB)
  - Note that in most cases,  $C_l = C_s$ , which is a parameter depending on hardware





## Handling context switch overheads ?

---

- Thus, the **real computing time** for a task should be  $C_i' = C_i + C_l + C_s$
- The schedulability analysis techniques we studied so far are applicable if we use the new computing time  $C'$ .
  - Unfortunately this is not right



## Handling context switch

---

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

(each preemption  $\rightarrow$  2 context switches)

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

# Handling interrupts: problem and example

Task 0 is the interrupt handler with highest priority

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







## Handling interrupts: solution

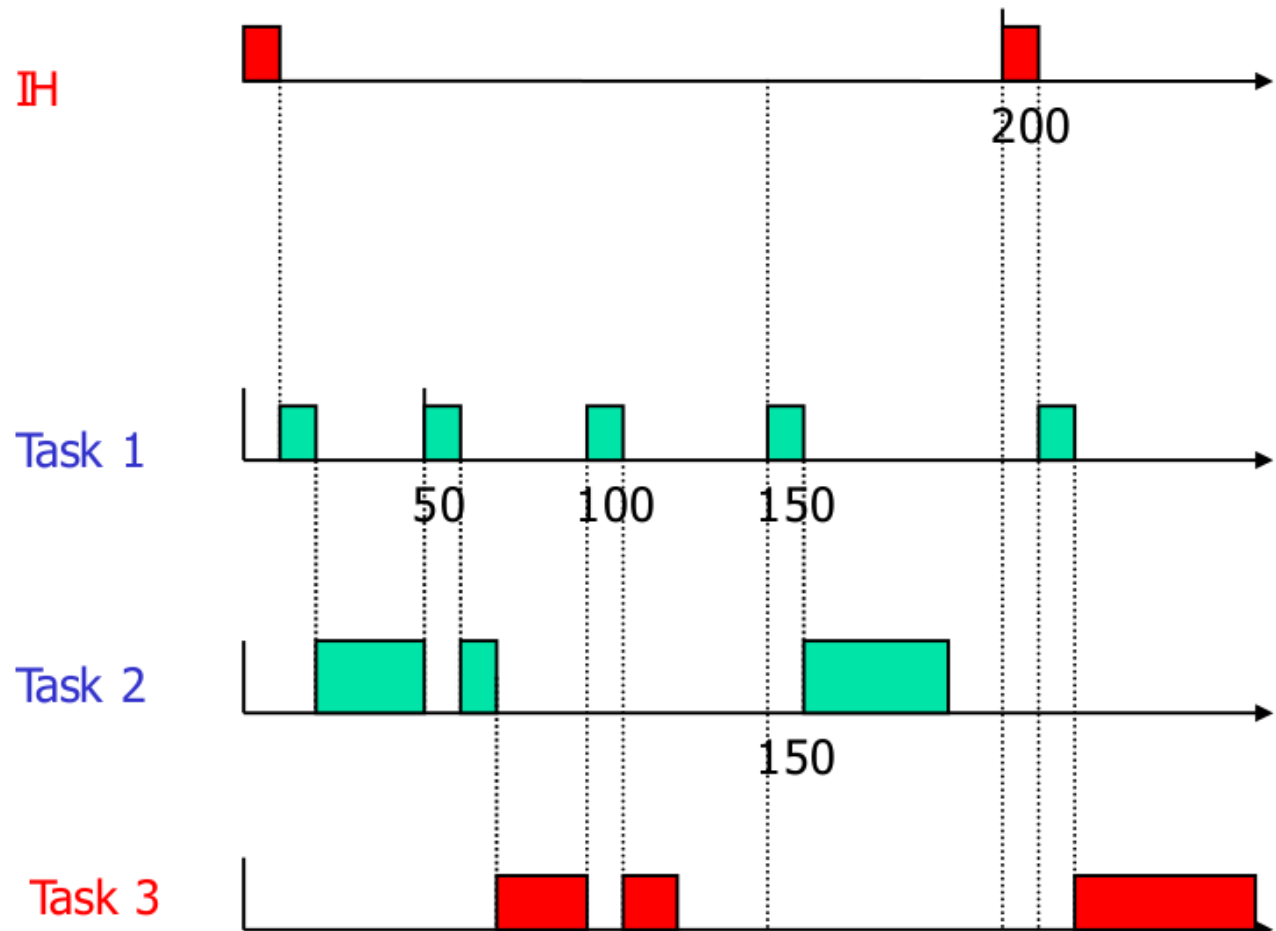
---

- **Whenever possible:** move code from the interrupt handler to a special application task with the same rate as the interrupt handler to make the interrupt handler (with high priority) as shorter as possible
- Interrupt processing can be inconsistent with RM priority assignment, and therefore can effect schedulability of task set (previous example)
  - Interrupt handler runs with high priority despites its period
  - Interrupt processing may delay tasks with shorter periods (deadlines)
  - **how to calculate the worst case response time ?**

# Handling interrupts: example

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

---

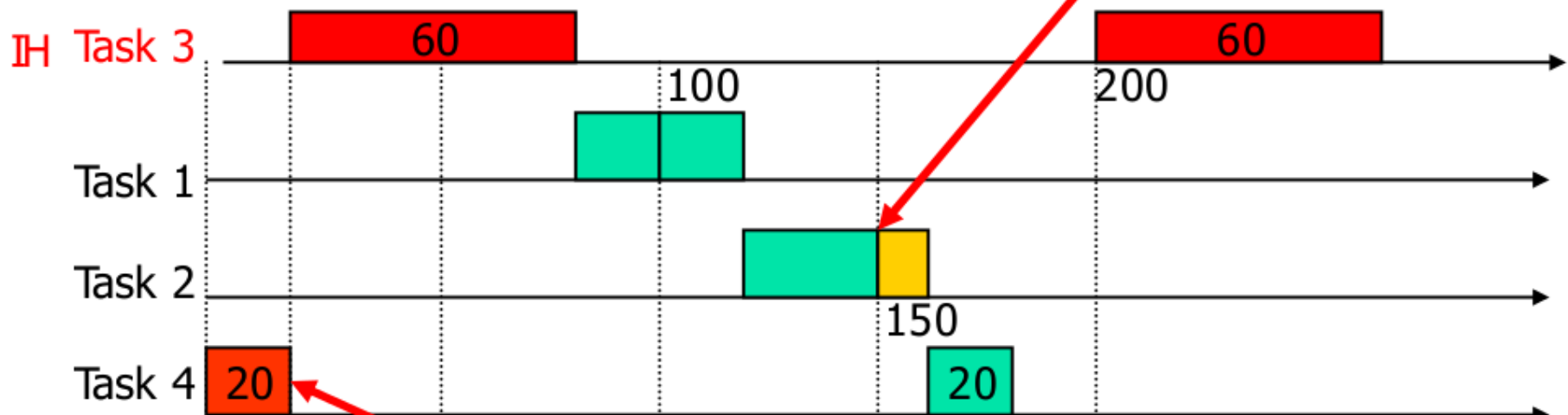
- So far, we have assumed that all tasks are preemptive regions of code. This not always the case e.g code for context switch though it may be short, and the short part of the interrupt handler as we considered before
  - Some section of a task is non preemptive
- In general, we may assume an extra parameter **B** in the task model, which is the computing time for the non preemptive section of a task.
  - $B_i$  = computing time of non preemptive section of task  $i$

# Handling non preemptive sections: Problem and Example


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

Missing deadline 150



Non preemptive/non interruptible section of 20



## Handling non-preemptive sections: Response time calculation

---

- The equation for response time calculation:

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

- Where  $B_i$  is the longest time that task  $i$  can be blocked by lower-priority tasks with non preemptive section
  - Note that a task preempts only one task with lower priority within each period



So now, we have an equation:

---

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



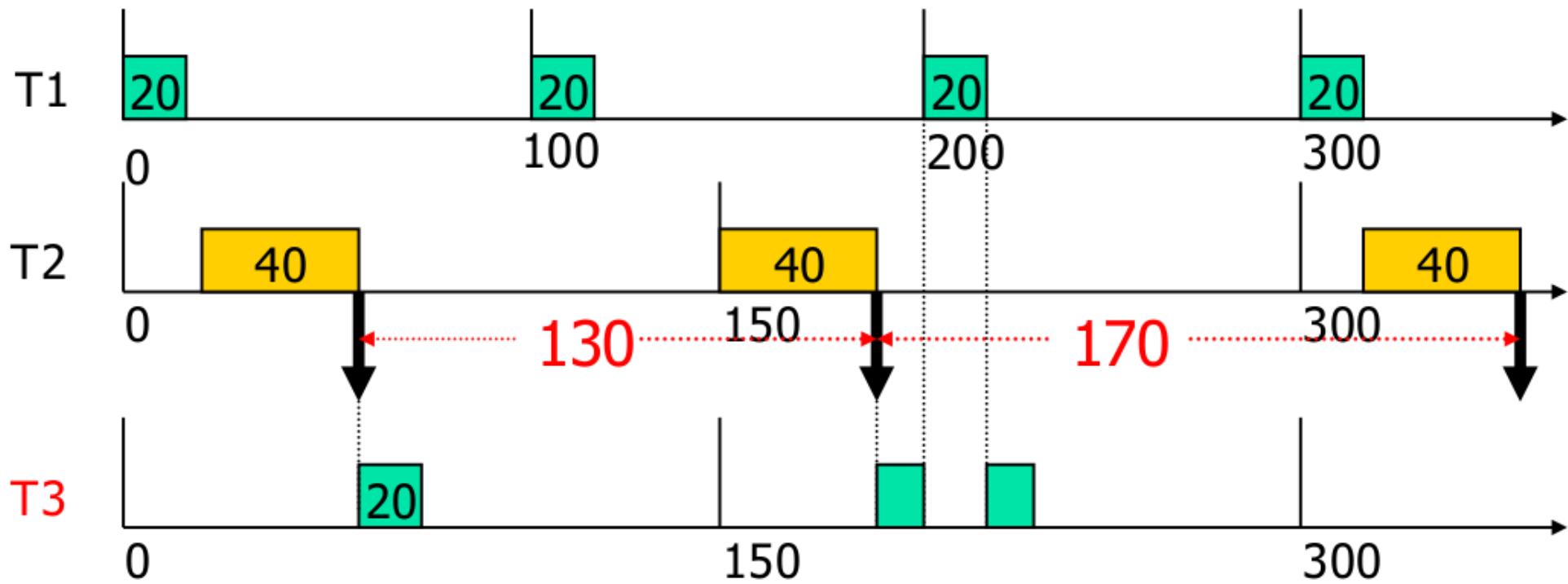
## The Jitter Problem

---

- So far, we have assumed that tasks are released at a constant rate (at the start of a constant period)
- This is true in practice and a realistic assumption
- However, there are situations where the period or rather the release time may 'jitter' or change a little, but the jitter is bounded with some constant  $J$
- The jitter may cause some task missing deadline

# Jitter: Example

$\{(20,100),(40,150),(20, T3)\}$



T3 is activated by T2 when it finishes within each period  
Note that because the response time for T2 is not a constant,  
the period between two instances of T3 is not a constant: 170, 130





## Jitter: Definition

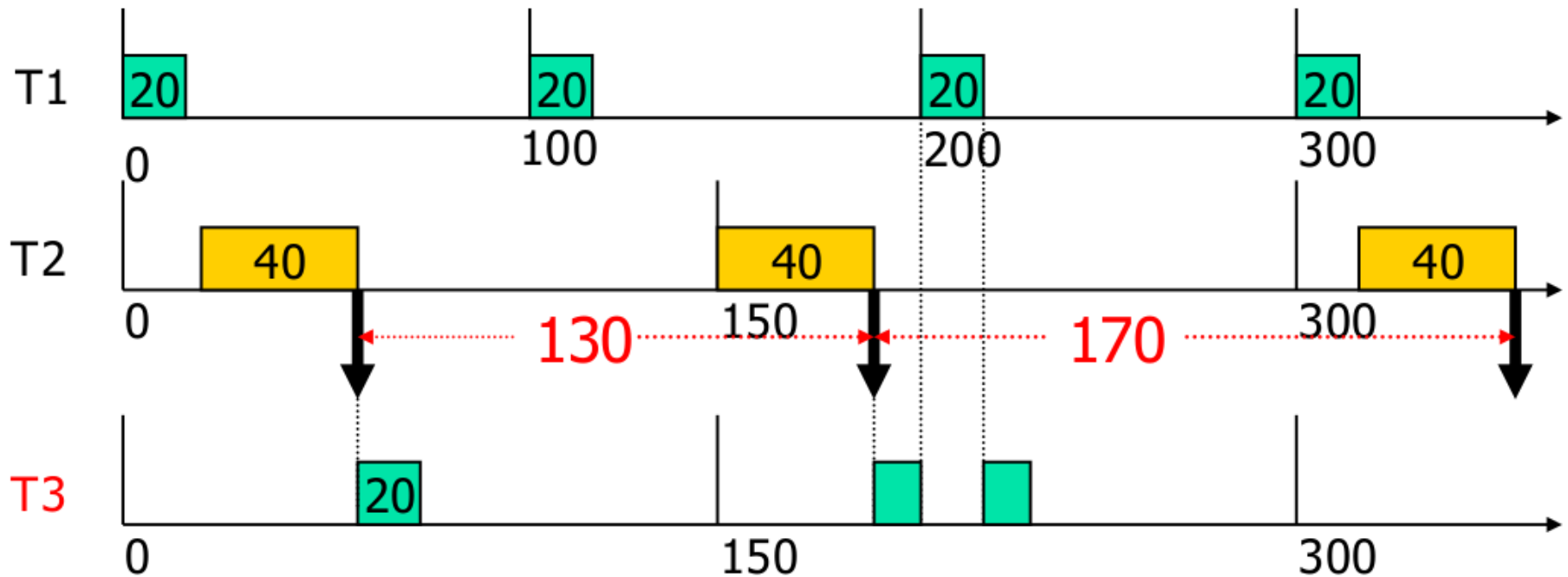
---

- $J(\text{biggest}) = \text{maximal delay from period-start}$
- $J(\text{smallest}) = \text{minimal delay from period-start}$
- $\text{Jitter} = J(\text{biggest}) - J(\text{smallest})$
  
- Jitter = the maximal length of the interval in which a task may be released **non-deterministically**
  
- **If  $J(\text{biggest}) = J(\text{smallest})$ , then NO JITTER** and therefore no influence on the other tasks with lower priorities

# Jitter: Example

$\{(20,100),(40,150),(20, T3)\}$

$\text{Pr}(T1)=1, \text{Pr}(T2)=2, \text{Pr}(T3)=3$



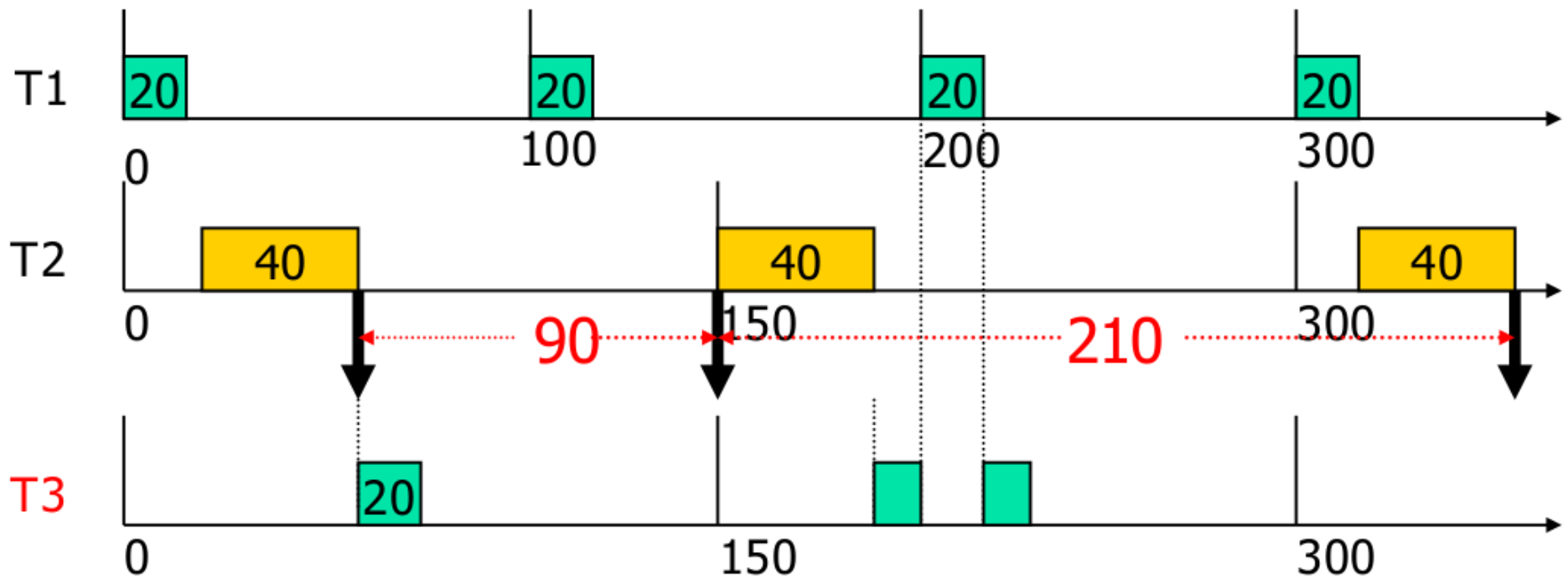
T3 is activated by T2 by the end of each instance

$J(\text{biggest}) = R2(\text{worst case}), J(\text{smallest}) = R2(\text{best case})$

Jitter =  $J(\text{biggest}) - J(\text{smallest}) = 60 - 40 = 20$

# Jitter: Example

$\{(20,100),(40,150),(20, T3)\}$



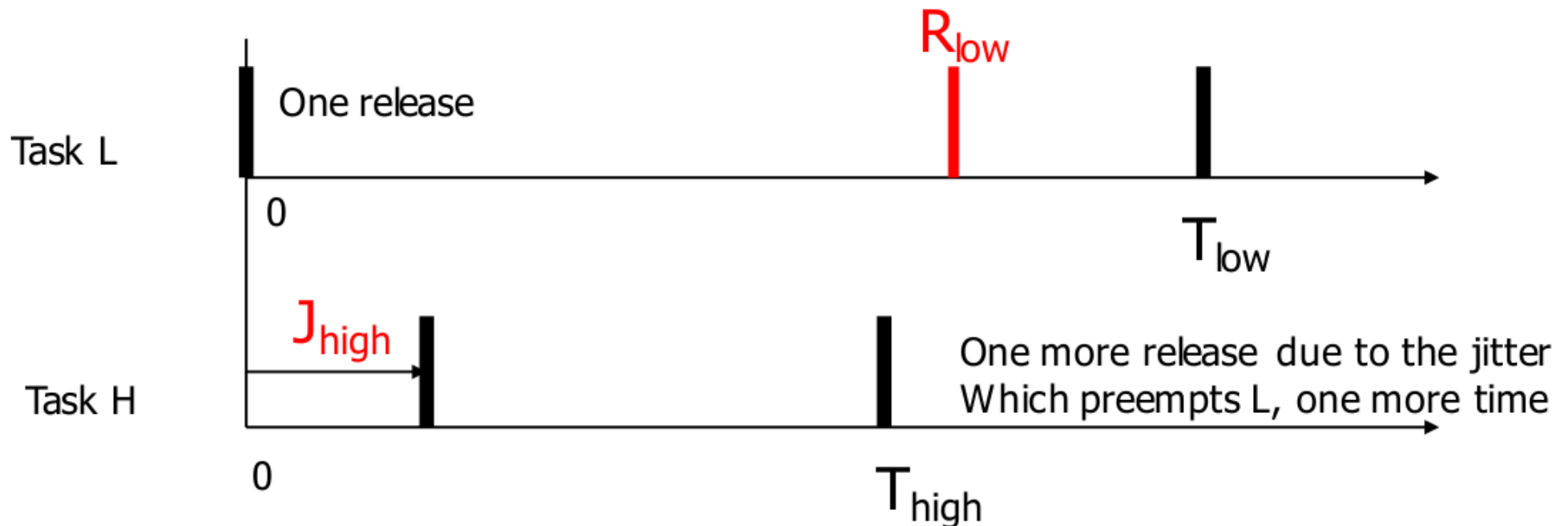
T3 is activated by T2 at any time during its execution of an instance

$J(\text{biggest}) = R2(\text{worst case})$ ,  $J(\text{smallest}) = R2(\text{best case}) - C2$

Jitter =  $J(\text{biggest}) - J(\text{smallest}) = 60 - 0 = 60$

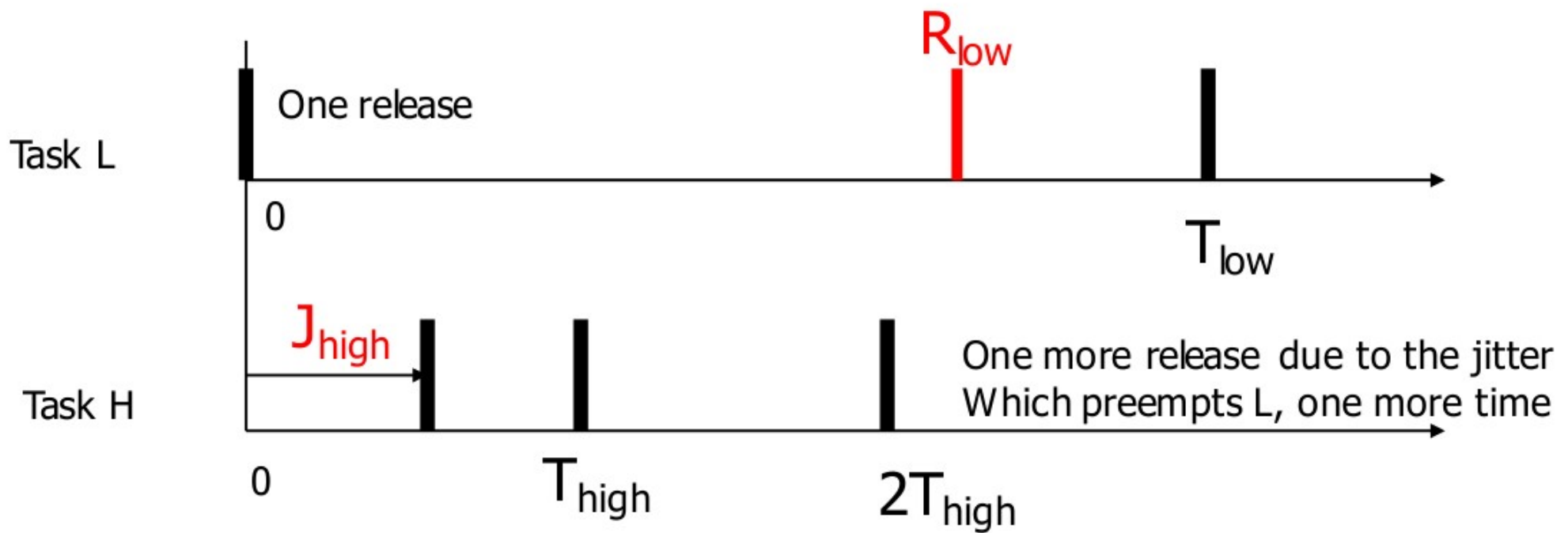
# The number of preemptions due to Jitter

Task L will be preempted **at least 2 times** if  $R_{low} > T_{high} - J_{high}$





Task L will be preempted **at least 3 times** if  $R_{low} > 2T_{high} - J_{high}$





## The number of preemptions/blocking when jitters occur

---

- Task L will be preempted **at least 2** times if  $R_{low} > T_{high} - J_{high}$
- Task L will be preempted **at least 3** times if  $R_{low} > 2 * T_{high} - J_{high}$
- ...
- Task L will be preempted **at least n** times if
$$R_{low} > (n-1) * T_{high} - J_{high}$$
- Thus  $(R_{low} + J_{high}) / T_{high} > n-1$
- the largest **n** satisfying the condition is given by
$$n = \lceil (R_{low} + J_{high}) / T_{high} \rceil$$



## Handling Jitters in schedulability analysis

---

- $R_i = C_i + \sum_{j \in HP(i)} \text{"number of preemptions"} * C_j$ 
  - $R_i^* = R_i + J_i(\text{biggest})$
- if  $R_i^* < D_i$ , task  $i$  is schedulable otherwise no



## Handling Jitters in schedulability analysis

---

- $R_i = C_i + \sum_{j \in \text{HP}(i)} \lceil (R_i + J_j) / T_j \rceil * C_j$ 
  - $R_i^* = R_i + J_i(\text{biggest})$

why  $R_i + J_i(\text{biggest})$  ?

- if  $R_i^* < D_i$ , task  $i$  is schedulable, otherwise no





Now, we have an equation:

---

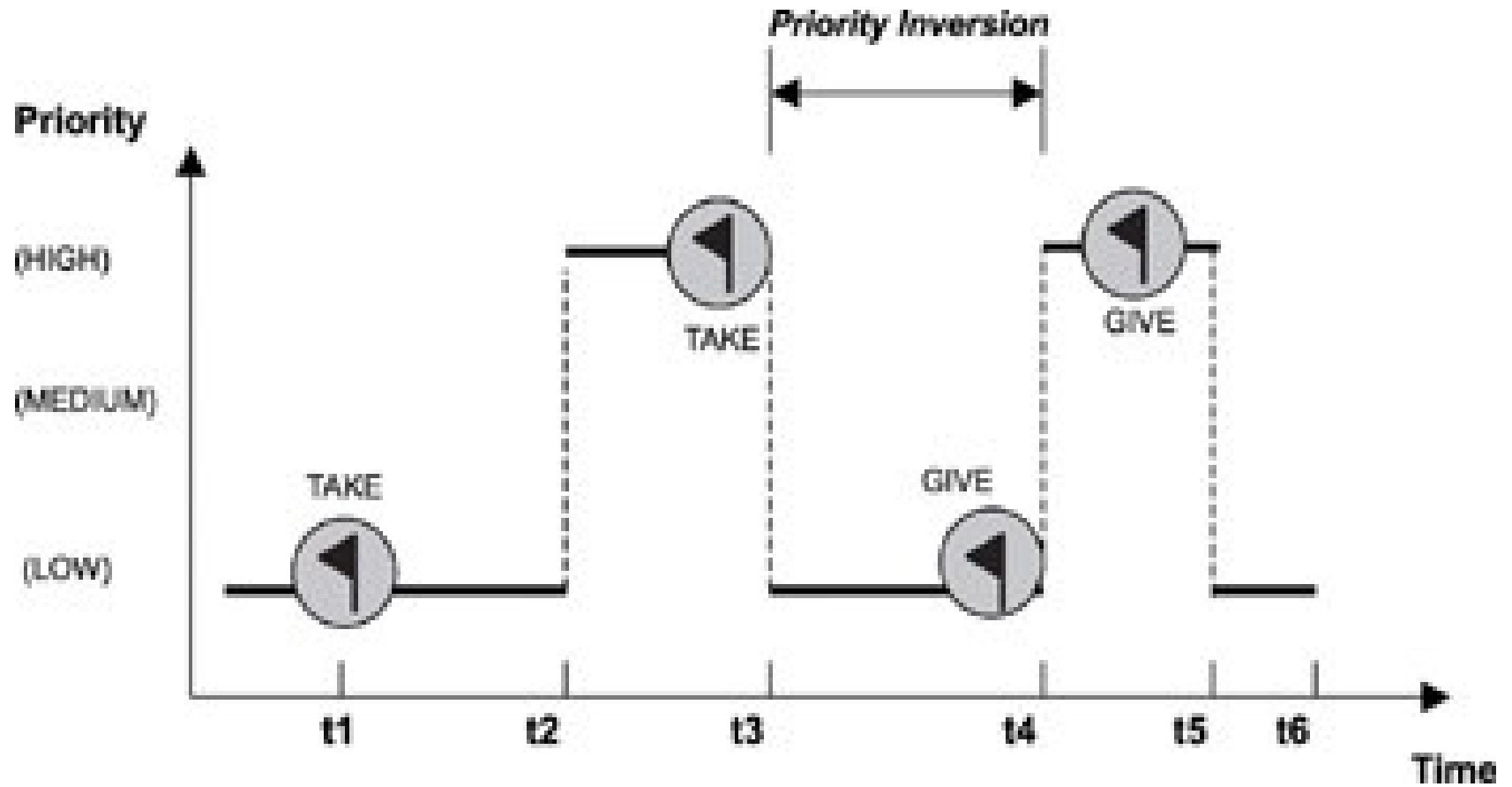
$$R_i = C_i + 2Ccs + B_i + \sum_{j \in HP(i)} \lceil (R_i + J_j) / T_j \rceil * (C_j + 4Ccs)$$

The response time for **task i**

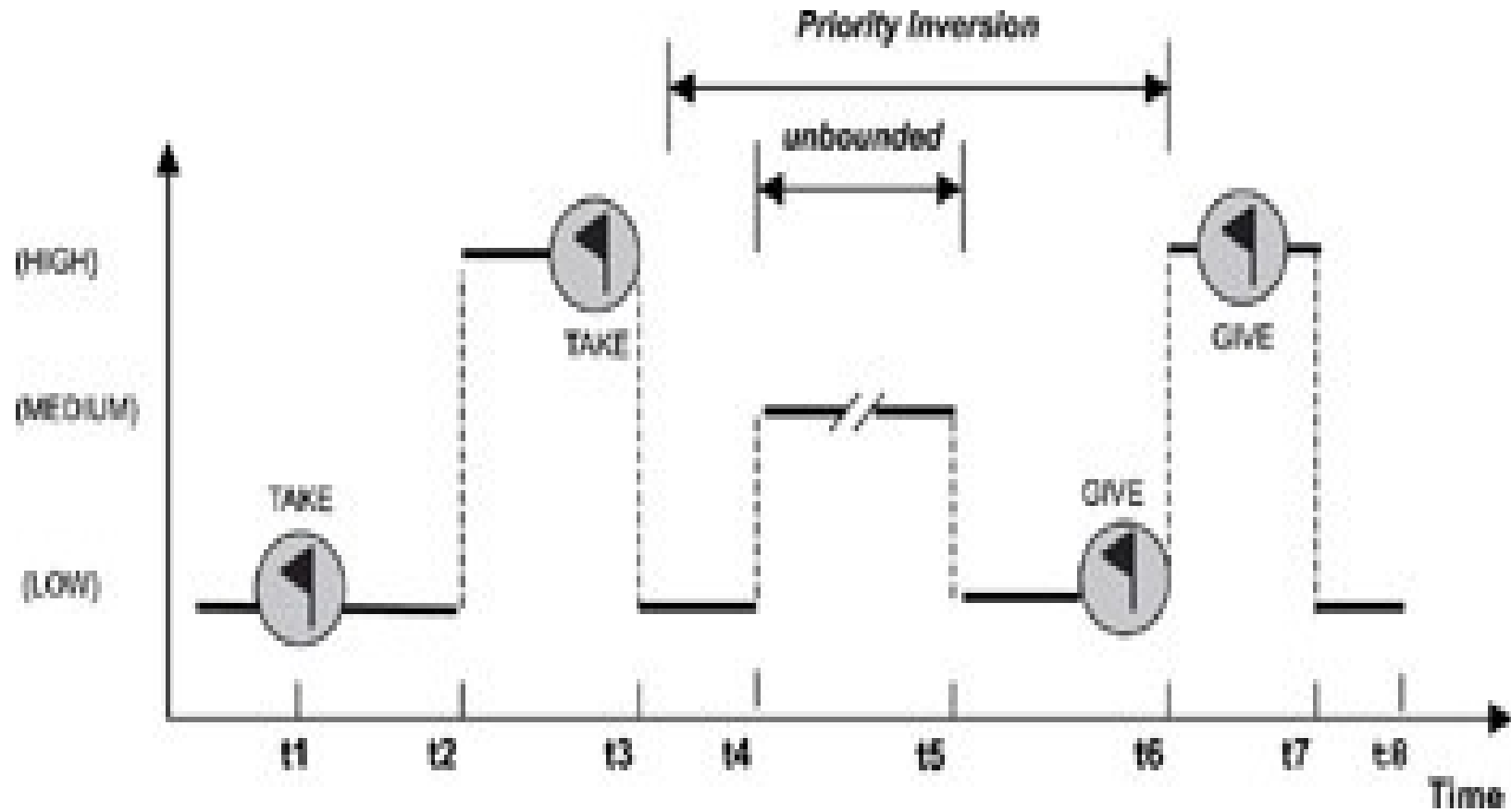
$$R_i^* = R_i + J_i(\text{biggest})$$

$J_i(\text{biggest})$  is the "biggest jitter" for task i

# Bounded Priority Inversion

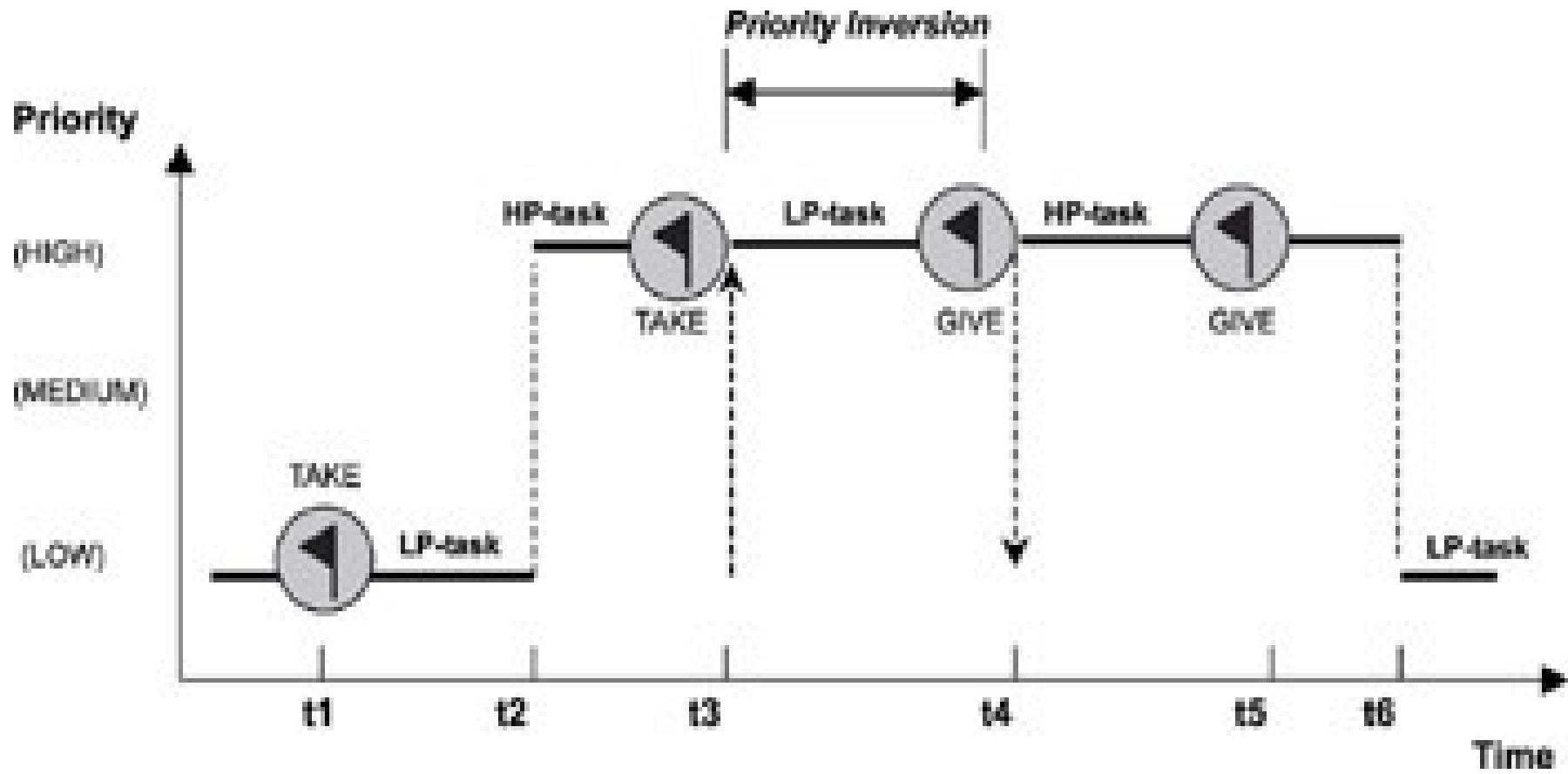


# Unbounded Priority Inversion

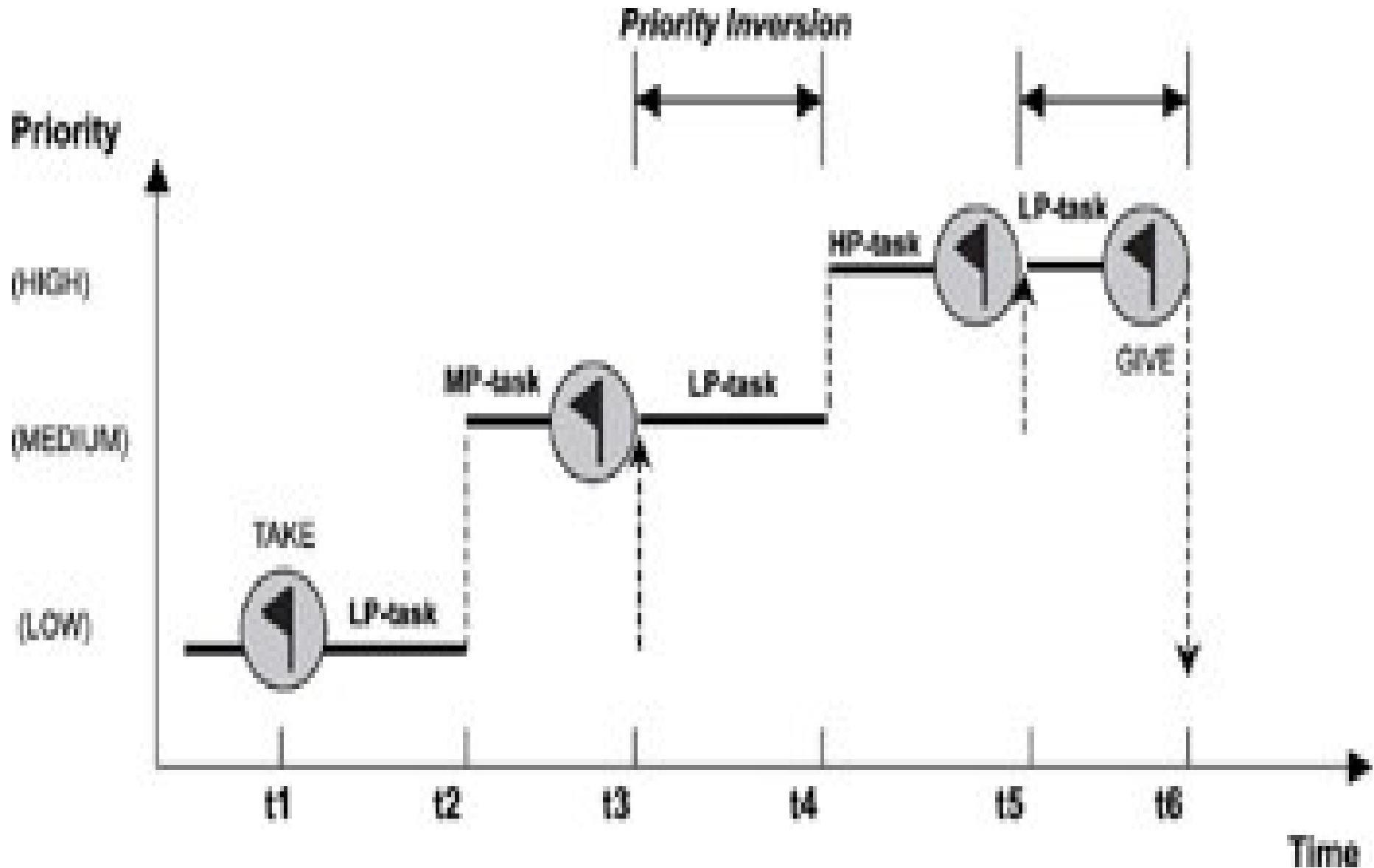


# Priority Inheritance

Rule #	Description
1	If R is in use, T is blocked.
2	If R is free, R is allocated to T.
3	When a task of a higher priority requests the same resource, T's execution priority is raised to the requesting task's priority level.
4	The task returns to its previous priority when it releases R.



# Transitive Priority Inheritance

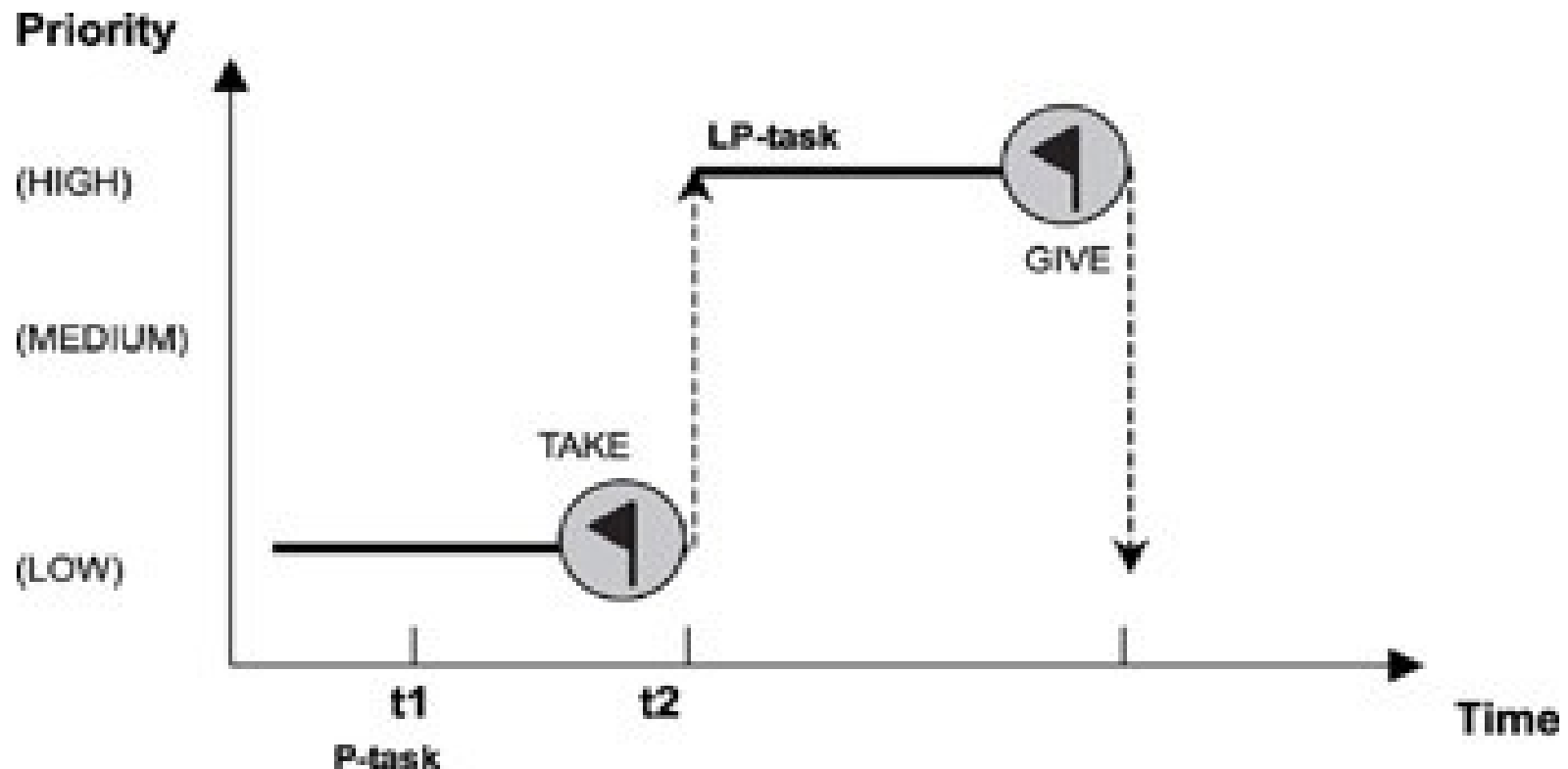


# Ceiling Priority Protocol

- In the ceiling priority protocol, the priority of every task is known, as are the resources required by every task.
- For a given resource, the priority ceiling is the highest priority of all possible tasks that might require the resource.
- For example, if a resource R is required by four tasks (T1 of priority 4, T2 of priority 9, T3 of priority 10, and T4 of priority 8), the priority ceiling of R is 10, which is the highest priority of the four tasks.
- With the ceiling priority protocol, the task inherits the priority ceiling of the resource as soon as the task acquires the resource even when no other higher priority tasks contend for the same resource.
- This rule implies that all critical sections from every sharing task have the same criticality level. The idea is to finish the critical section as soon as possible to avoid possible conflicts.

# Ceiling Priority Protocol Rules

Rule #	Description
1	If R is in use, T is blocked.
2	If R is free, R is allocated to T. T's execution priority is raised to the priority ceiling of R if that is higher. At any given time, T's execution priority equals the highest priority ceiling of all its held resources.
3	T's priority is assigned the next-highest priority ceiling of another resource when the resource with the highest priority ceiling is released.
4	The task returns to its assigned priority after it has released all resources.



# Next week classes

- Commercial and free RTOS
  - Hard real time guarantees
- Linux Real Time patch
  - No guarantee, but design for more predictability
- Minor 2 (due on Oct 11):
  - Build raspbian from source (CFS scheduler)
  - Apply BFS patch
  - Apply Preempt RT patch
  - Run same benchmark with three different schedulers and report average and worst case latencies