ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Real-Time Operating Systems M

11. Real-Time: Periodic Task Scheduling

Notice

The course material includes slides downloaded from:

http://codex.cs.yale.edu/avi/os-book/

(slides by Silberschatz, Galvin, and Gagne, associated with Operating System Concepts, 9th Edition, Wiley, 2013)

and

http://retis.sssup.it/~giorgio/rts-MECS.html

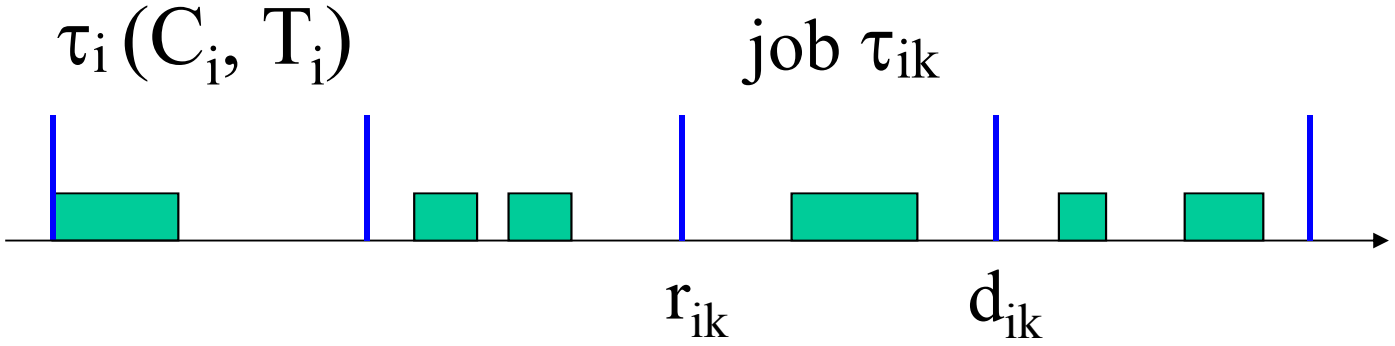
(slides by Buttazzo, associated with Hard Real-Time Computing Systems, 3rd Edition, Springer, 2011)

which has been edited to suit the needs of this course.

The slides are authorized for personal use only.

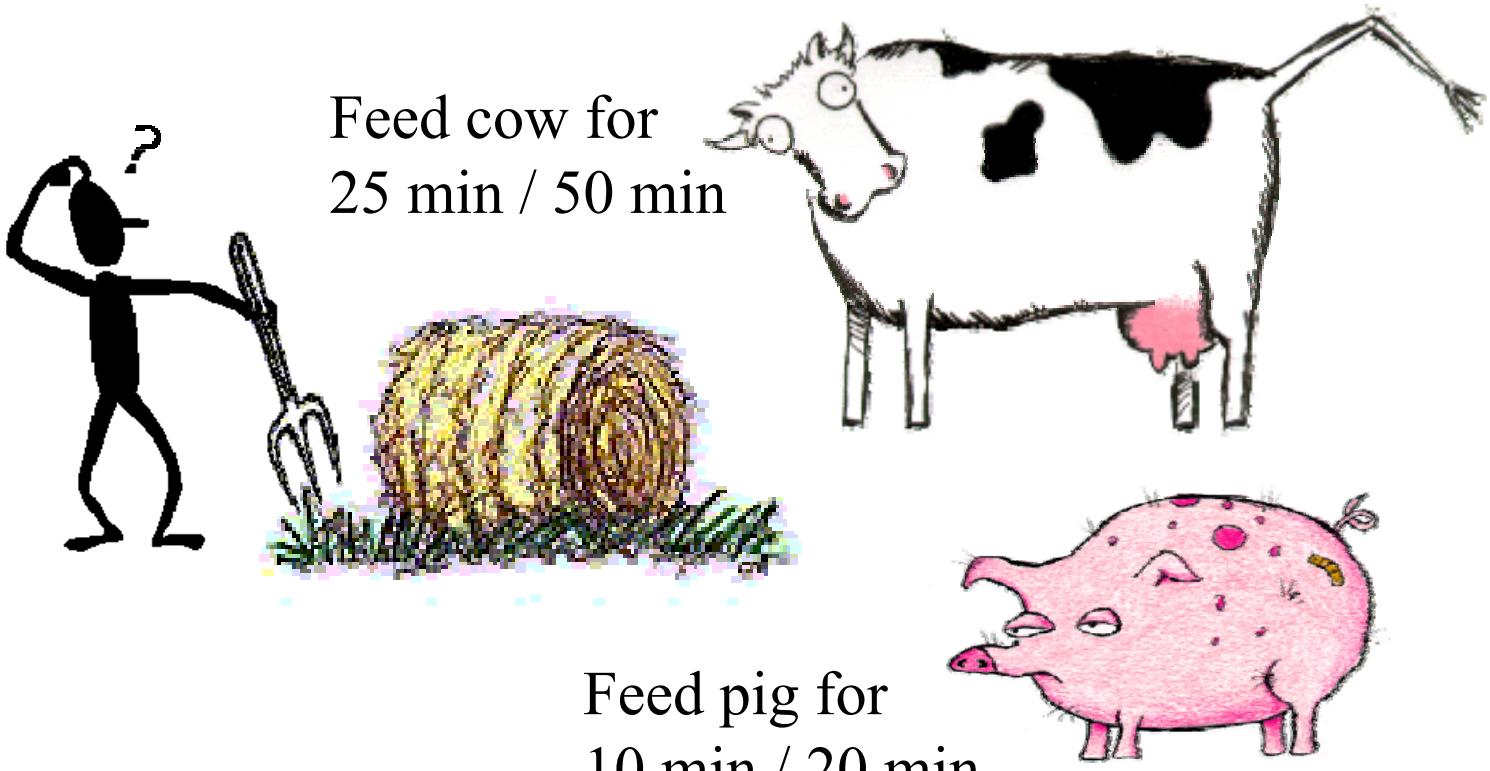
Any other use, redistribution, and any for profit sale of the slides (in any form) requires the consent of the copyright owners.

Problem Formulation



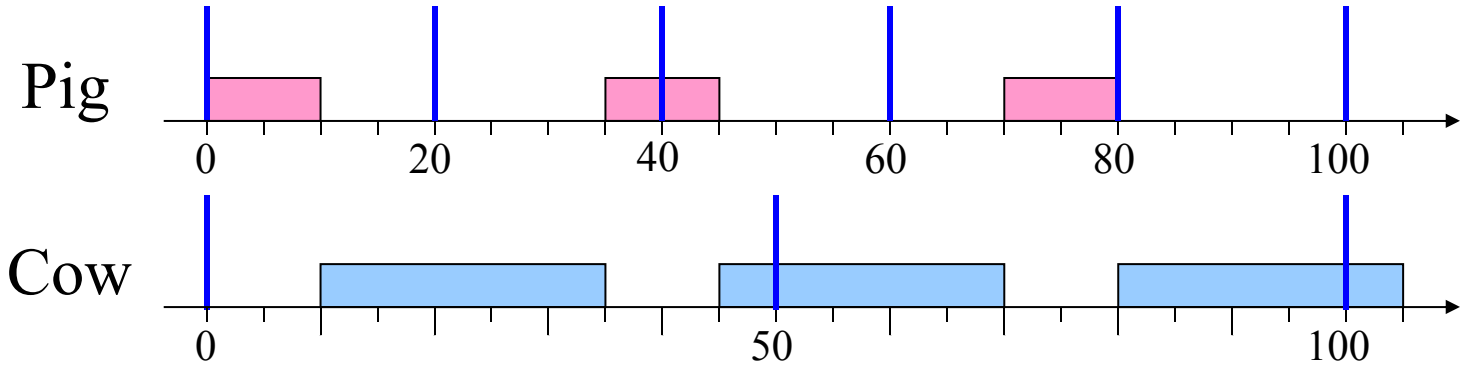
- For **each** periodic task, **guarantee** that:
  - Each job τ_{ik} is **activated** at $r_{ij} = (k-1)T_i$
  - Each job τ_{ik} **completes** within $d_{ik} = r_{ij} + D_i$

A Farm Scheduling Problem



First Algorithm

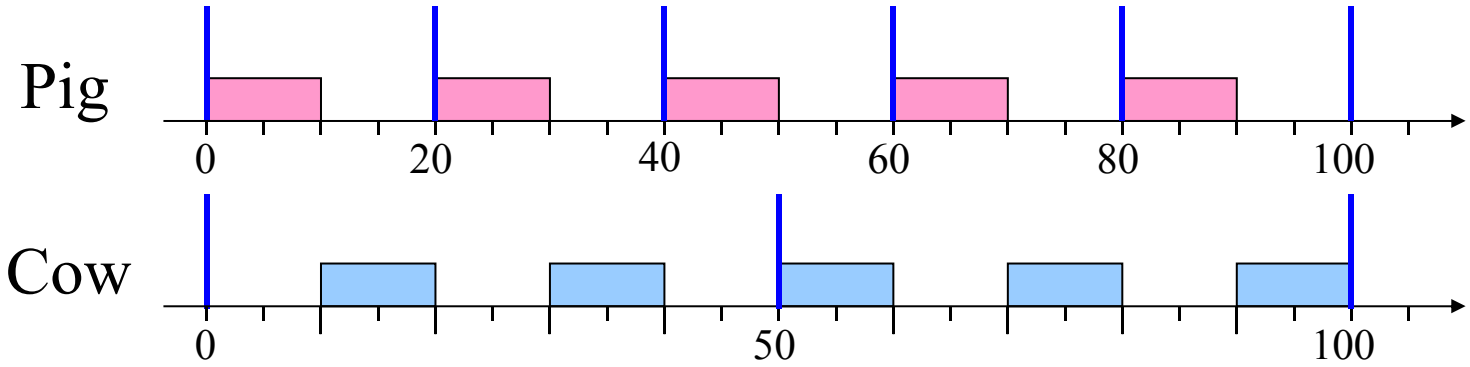
"Alternate pig with cow"



- Evaluation:
  - Pig gets hungry
  - Cow gets fat

Second Algorithm

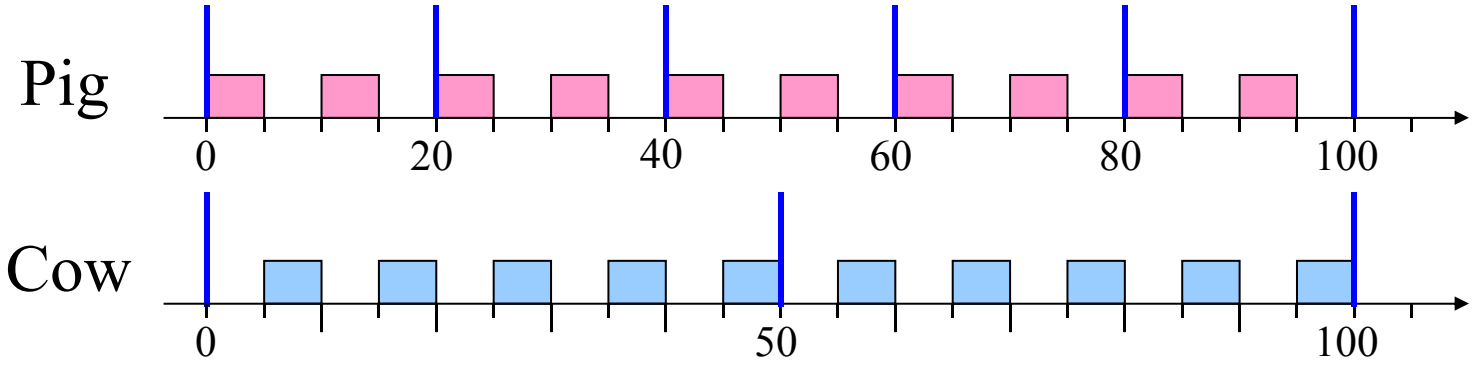
"Feed pig and cow 10 min each"



- Evaluation:
  - Pig is OK
  - Cow is not happy

Third Algorithm

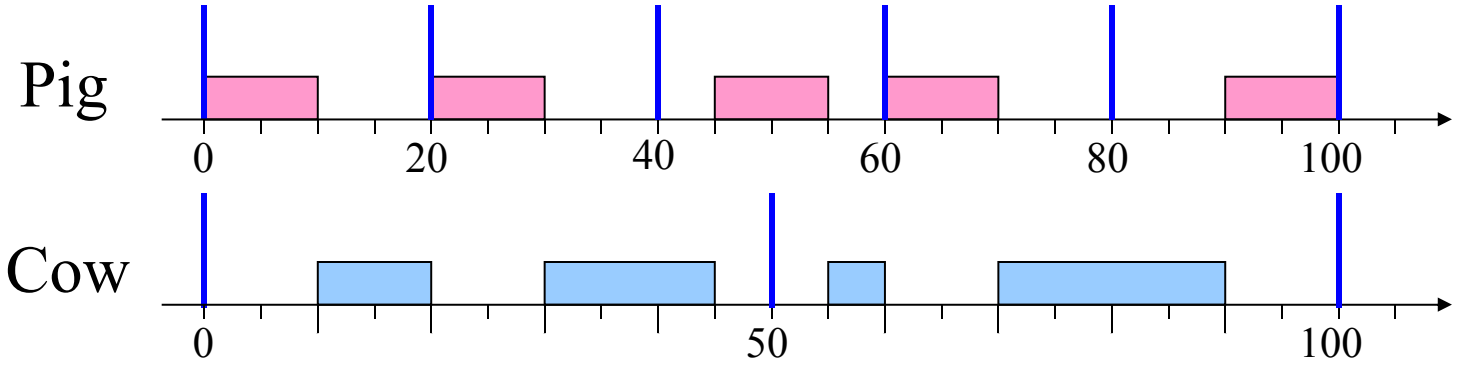
"Feed pig and cow 5 min each"



- Evaluation:
  - Pig is OK, Cow is OK
  - Farmer is tired

Optimal Algorithm

"Feed the most starving animal"

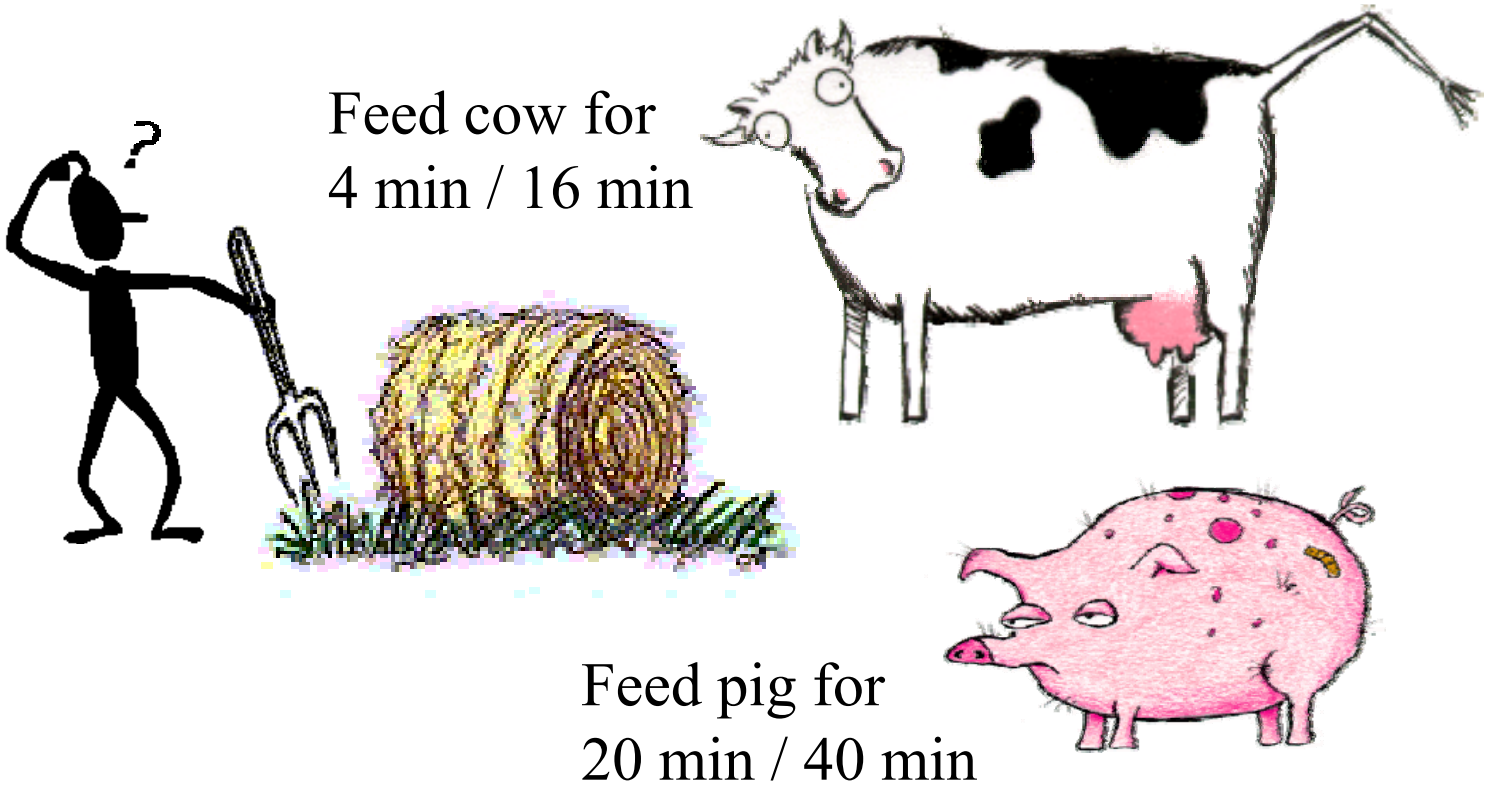


- Evaluation:
  - Everybody is happy

What Do We Learn?

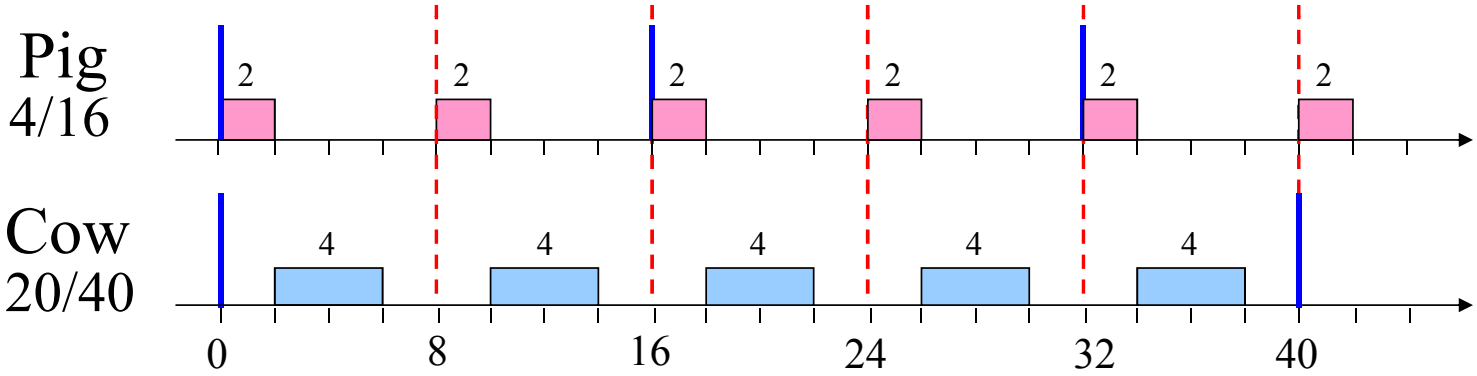
- Reducing the execution time window, we get closer to a feasible solution
- The time is split proportionally between the animals
  - In the example, each animal required food for 50% of the time
  - How can we generalize the solution if the animals require different fractions of time?

A New Scheduling Problem



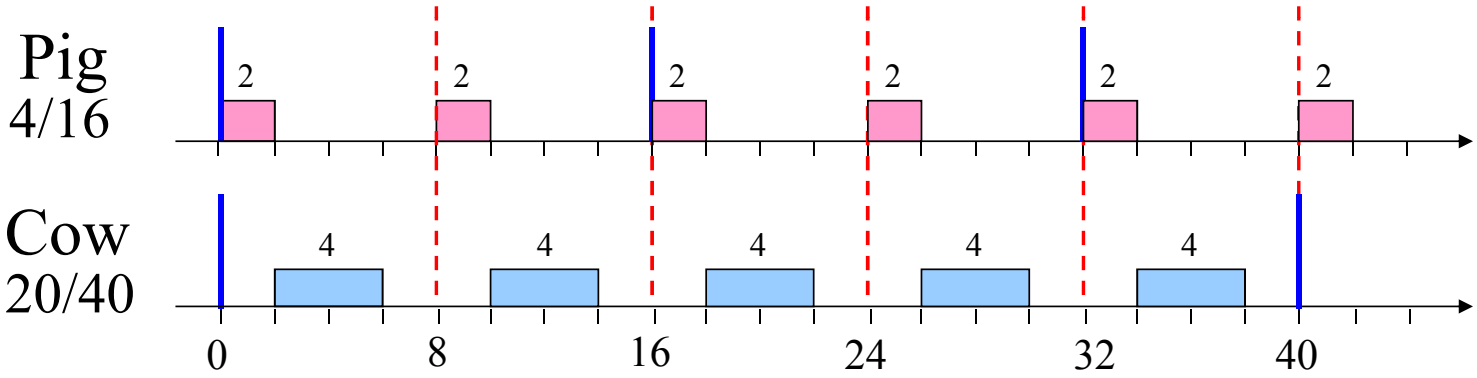
Proportional Share

- **Proportional Share** algorithm:
  1. Divide the timeline into slots of equal length
  2. Within each slot serve each task for a time proportional to its utilization:
     - Cow utilization factor = 4/16 = ¼
     - Pig utilization factor = 20/40 = ½



Proportional Share

- **In general**:
  - Let: U_i = required feeding fraction
  - $\Delta = GCD(T_1, T_2) = 8$
    - Execute each task for $\delta_i = U_i\Delta$ in each slot Δ



- Note: $U_i\Delta$ ensures C_i in T_i, in fact: $\delta(T_i/\Delta)=C_i$
- **Feasibility test**: $\Sigma\delta_i \leq \Delta$, i.e., $\Sigma U_i \leq 1$

Timeline Scheduling (Cyclic Executive)

- It has been used for 30 years in military systems, navigation, and monitoring systems.
  - Examples:
    - Air traffic control
    - Space Shuttle
    - Boeing 777

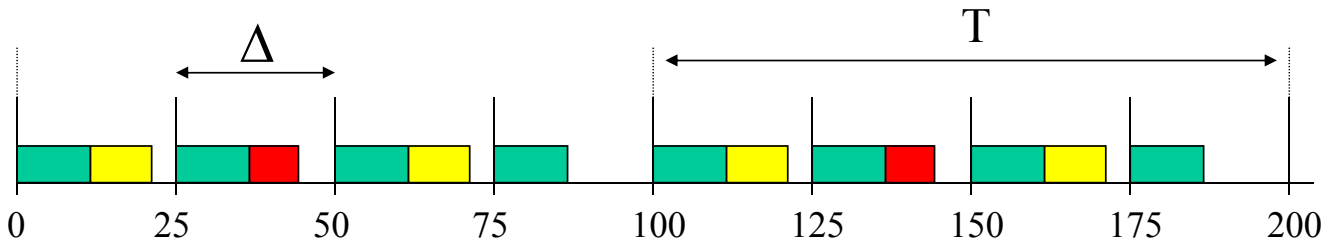
- Idea:
  - Divide time axis in slots of equal length
  - Design static scheduling (by hand)
    - Allocate each task in a slot, so as to meet the desired request rate
  - Activate execution of each slot by a timer

Example

task	f	T
A	40 Hz	25 ms
B	20 Hz	50 ms
C	10 Hz	100 ms

$\Delta = \text{GCD}$ (minor cycle)

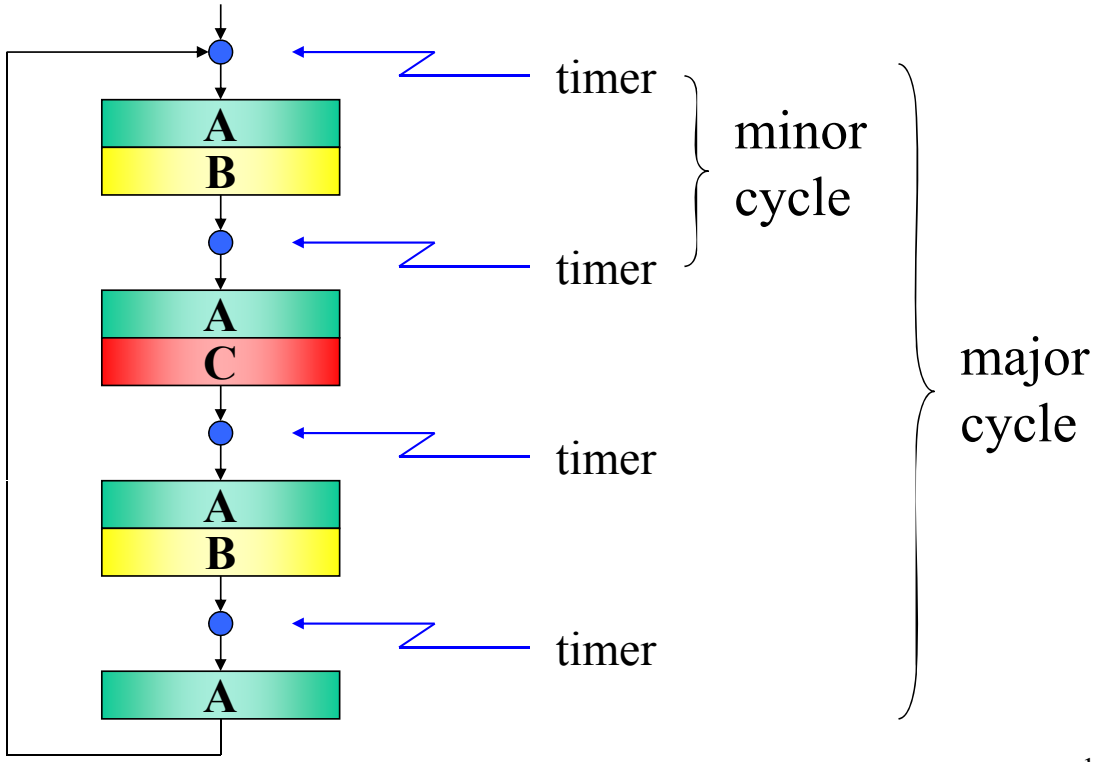
$T = \text{lcm}$ (major cycle)



- Guarantee is very simple (within each minor cycle):
  - $C_A + C_B \leq \Delta = 25$ ms
  - $C_A + C_c \leq \Delta = 25$ ms

Cyclic Executive: Implementation

- The task sequence is **not** decided by a scheduling algorithm in the kernel, but it is triggered by calls made by the main program (no context switches)



Cyclic Executive: PROs and CONs

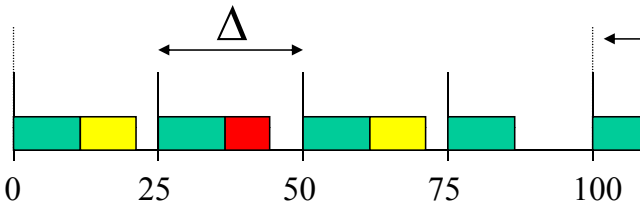
- Advantages: **lightweightness**, **regularity**
  - Simple implementation (does not require RTOS)
  - Low run-time overhead
  - Allows jitter control
- Disadvantages: **rigidity**
  - Fragile during overloads
  - Difficult to expand the schedule
  - Difficult to handle aperiodic activities

Problems During Overloads

- What do we do during task overruns?
  - Let the task continue
    - May have **domino effect** on all the other tasks (**timeline break**)
  - Abort the task
    - The system can remain in inconsistent states

Expandibility

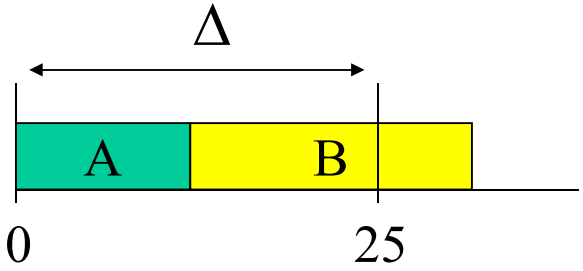
task	f	T
A	40 Hz	25 ms
B	20 Hz	50 ms
C	10 Hz	100 ms



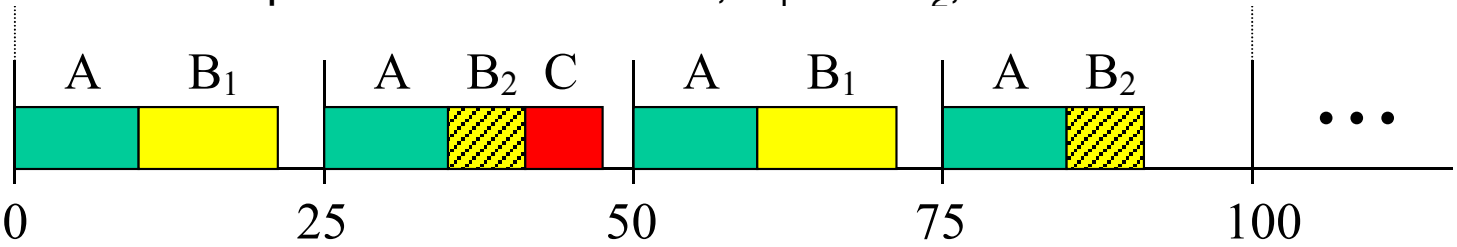
(previous schedule)

- If one or more tasks need to be upgraded, we may have to re-design the whole schedule again

- Example:
  - **Situation**: B is updated but $C_A+C_B > \Delta$

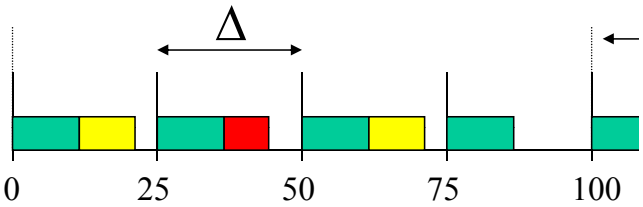


  - **Action**: split B in two subtasks, B_1 and B_2, and **re-build the schedule**



Expandibility

task	f	T
A	40 Hz	25 ms
B	20 Hz	50 ms
C	10 Hz	100 ms



(previous schedule)

- If the frequency of some task is changed, the impact can even be more significant:

- Example:
  - **Situation**: B's cycle changes from 50 ms to 40 ms

task	T	T
A	25 ms	25 ms
B	50 ms	**40 ms**
C	100 ms	100 ms
	before	**after**

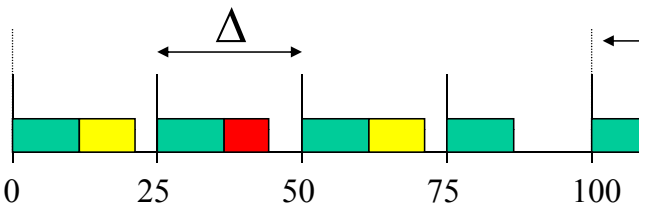
minor cycle:	$\Delta = 25$	$\Delta = 5$
major cycle:	$T = 100$	$T = 200$

(40 sync. per cycle!)

  - **Action**: **re-build the schedule** using different major/minor cycle length

Expandibility

task	f	T
A	40 Hz	25 ms
B	20 Hz	50 ms
C	10 Hz	100 ms



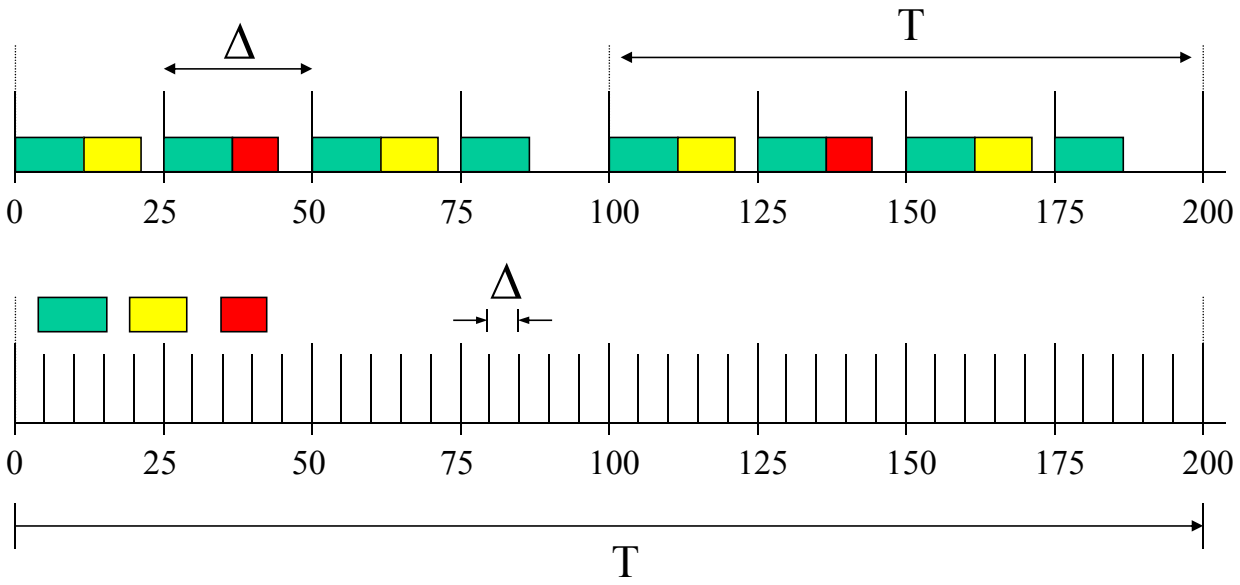
(previous schedule)

task	T	T
A	25 ms	25 ms
B	50 ms	**40 ms**
C	100 ms	100 ms
	before	**after**

minor cycle: $\Delta = 25$ $\Delta = 5$

major cycle: $T = 100$ $T = 200$

40 sync. per cycle!

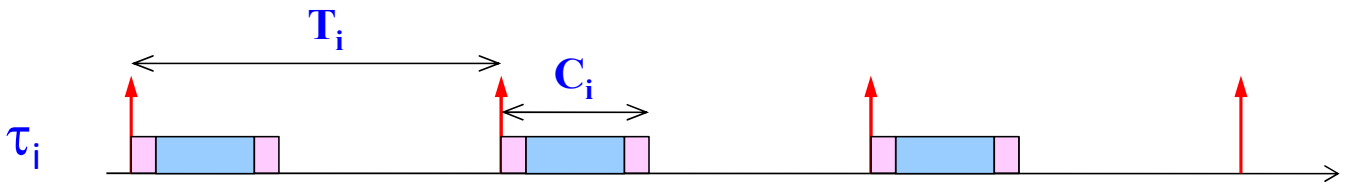


Priority Scheduling

- Idea:
  - Assign each task a **priority based on** its **timing constraints**
  - Verify the **feasibility** of the schedule using **analysis** techniques
  - Execute tasks on a **priority-based kernel**
- Priorities could be **static or dynamic**
- Examples:
  - **RM**: assign **fixed priority** to tasks, proportional to task rate
  - **EDF**: **at all times**, assign top priority to job with earliest absolute deadline

Assumptions

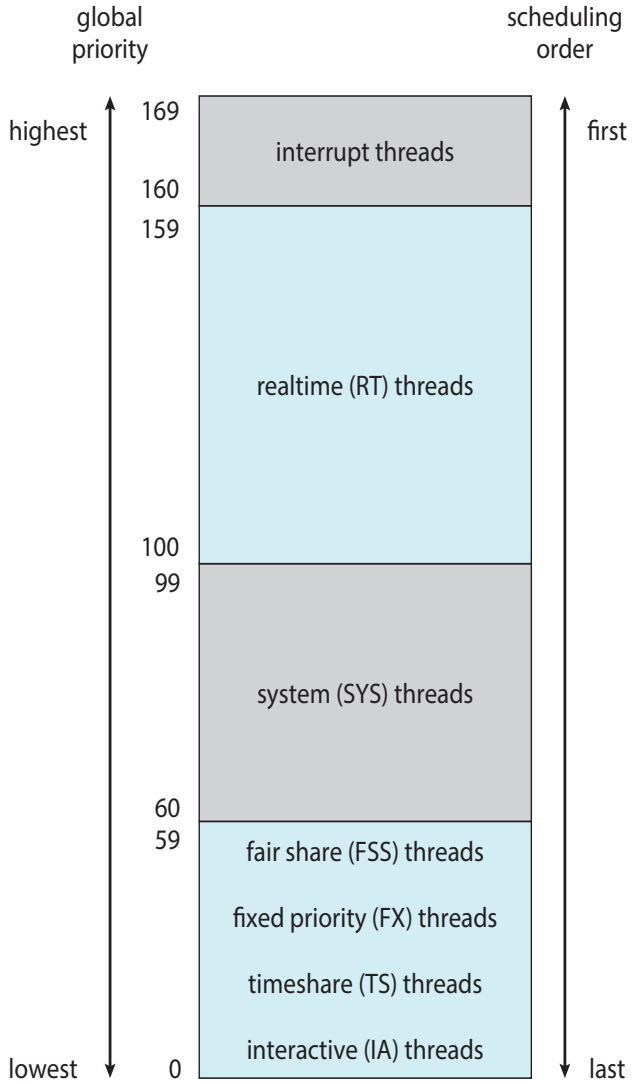
- The instances of a periodic task are regularly activated at a **constant rate** with **period** T_i.
  - All tasks are released as soon as they arrive.
- All instances of a periodic task have:
  - the **same worst-case execution time** C_i
  - the **same relative deadline** $D_i = T_i$
- Independent tasks (**no precedence relations**, **no resource constraints**)
- No task can suspend itself (trap)
- **Negligible kernel overheads**



Static Priority Scheduling

How to assign priorities?

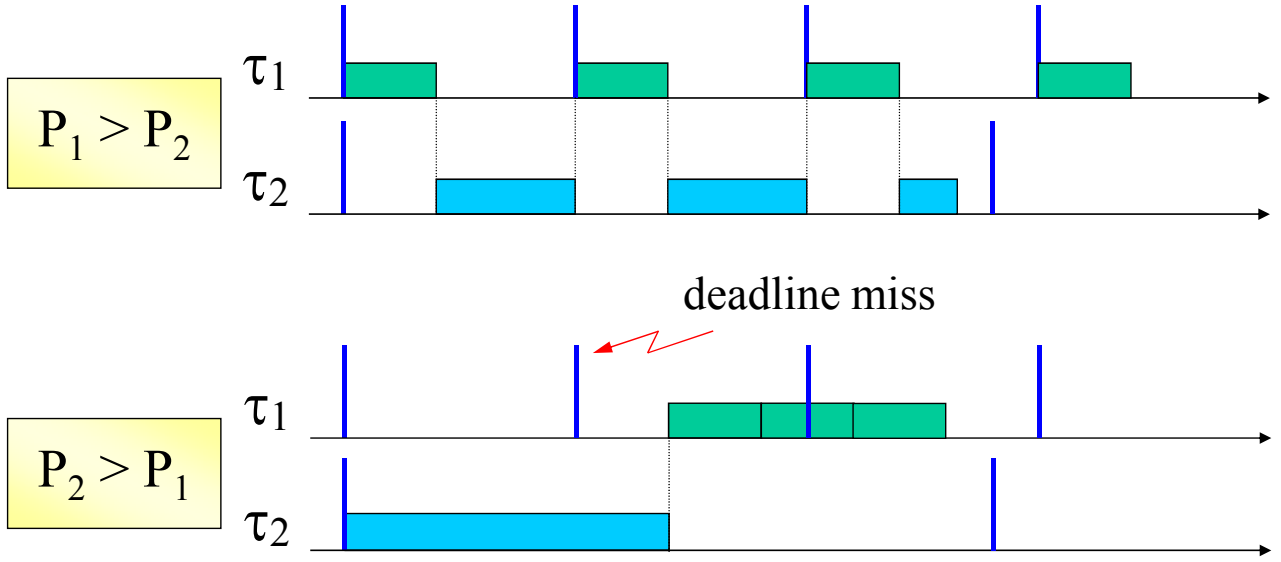
- Typically, task priorities are assigned based on the tasks' relative importance
  - Example: Solaris scheduling
- However, different assignments can lead to different **utilization bounds**



Priority ≠ Importance

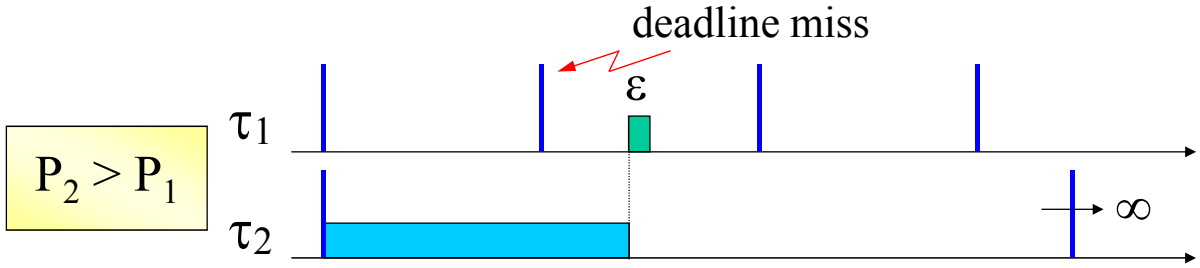
- If τ_2 is more important than τ_1 and is assigned a higher priority…

  …the schedule may not be feasilble...



Priority ≠ Importance

…while the utilization upper bound can be arbitrarily small.

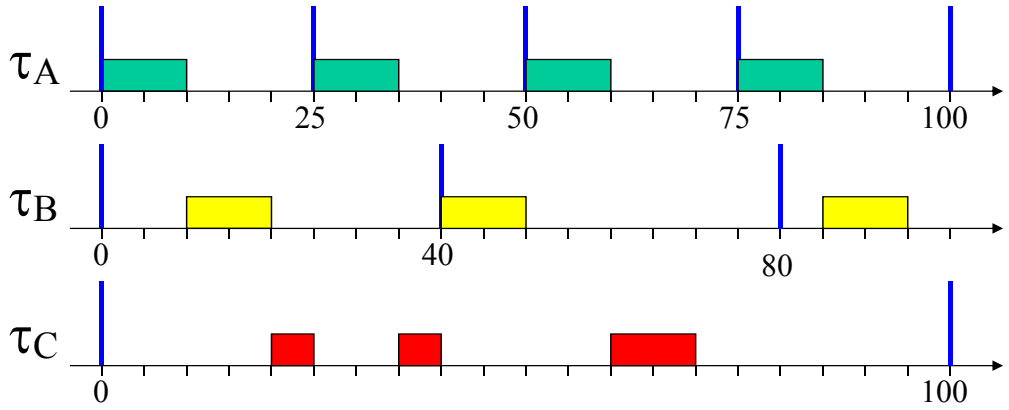


- $U = \varepsilon/C_1 + C_2/\infty \rightarrow 0$
- An application can be unfeasible even if the processor is almost empty!

Rate Monotonic (RM)

"Assign each task a fixed priority proportional to its request rate"

(Liu & Layland '73)



Rate Monotonic is Optimal

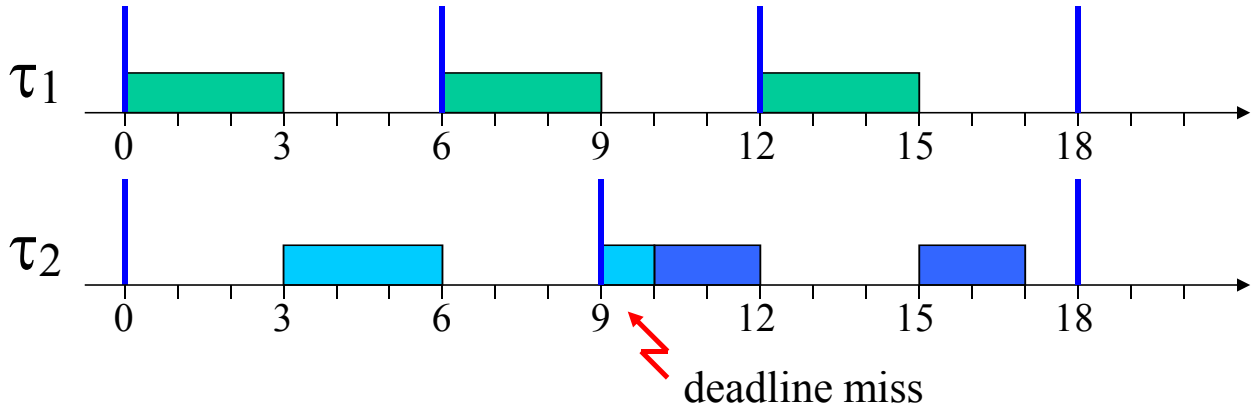
- **RM is optimal among all fixed priority algorithms (if $D_i = T_i$)**

- RM optimality (in the sense of feasibility):
  - If there exists a fixed priority assignment which leads to a feasilbe schedule for Γ, then the RM assignment is feasible for Γ.
  - If Γ is not schedulable by RM, then it cannot be scheduled by any fixed priority assignment.

An RM-Unfeasible Schedule

$D_i = T_i$

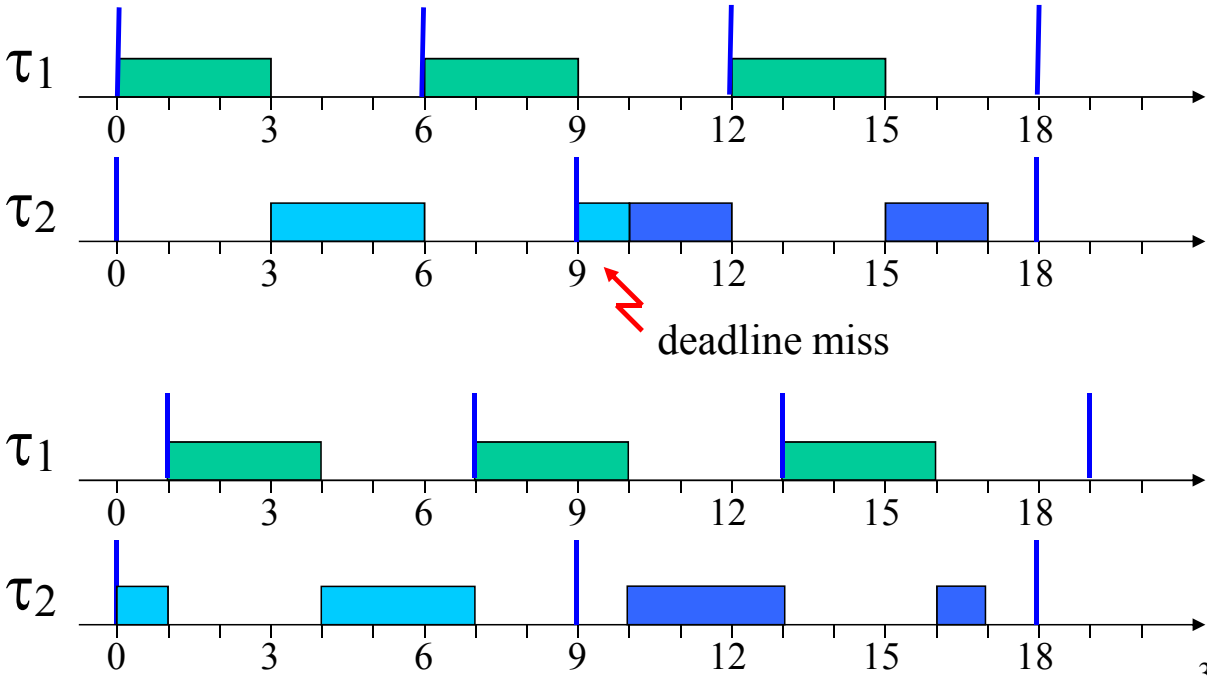
$$U_p = \frac{3}{6} + \frac{4}{9} = 0.94$$



Identifying the Worst Case

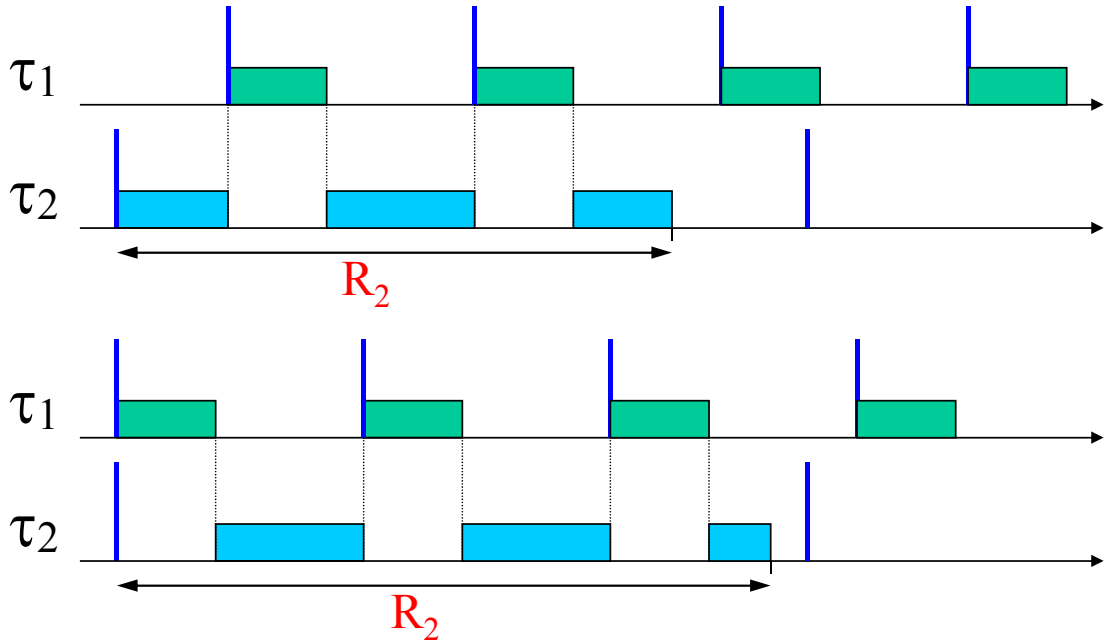
- Feasibility may depend on the initial activations (phases):

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.94$$



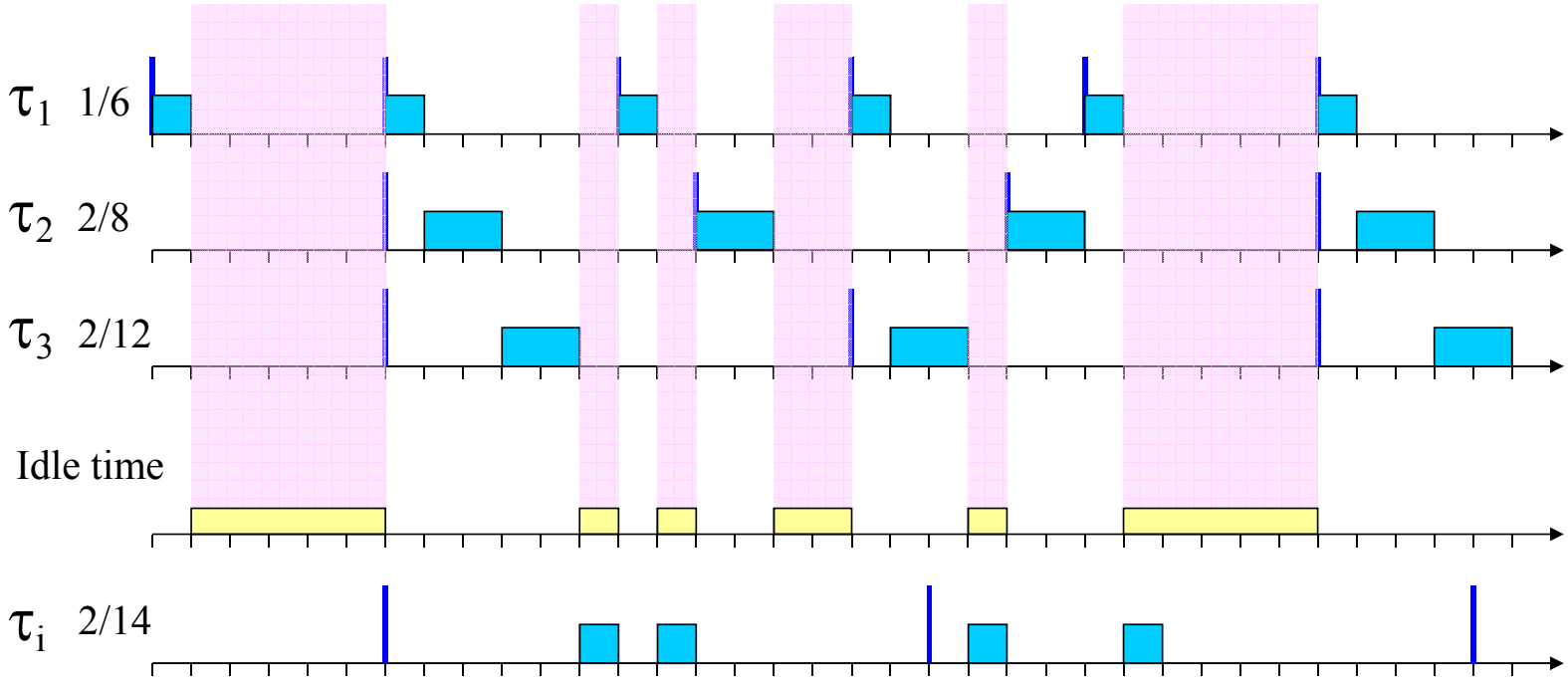
Critical Instant

- The **longest response time** occurs when a task arrives together with all higher priority tasks



Critical Instant

- For **independent preemptive tasks** under **fixed priorities**, the **critical instant** of τ_i occurs when it arrives together with all higher priority tasks.



How can we verify feasibility?

- Each task uses the processor for a fraction of time:

$$U_i = \frac{C_i}{T_i}$$

- Hence the total **processor utilization** is:

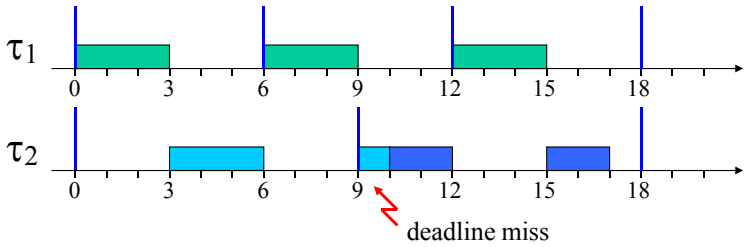
$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

- U_p is a measure of the **processor load**

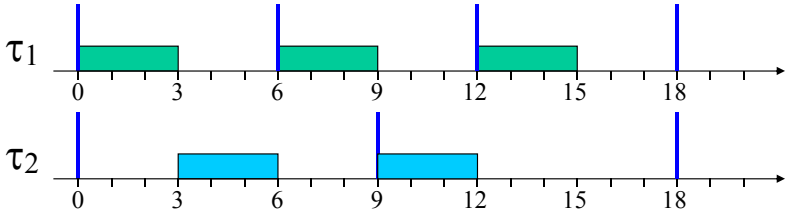
A Necessary Condition

- If $U_p > 1$ the processor is **overloaded** hence the task set cannot be schedulable
- However, there are cases where:
  - $U_p < 1$
  - but the task set is **not schedulable by RM**

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$



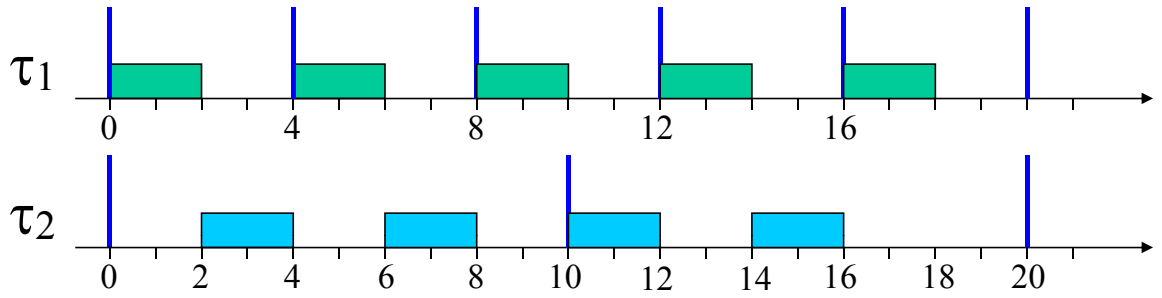
$$U_p = \frac{3}{6} + \frac{3}{9} = 0.833$$



- **Utilization upper bound**: if C_1 or C_2 is increased, τ_2 will miss its deadline!

…and a Different Upper Bound…

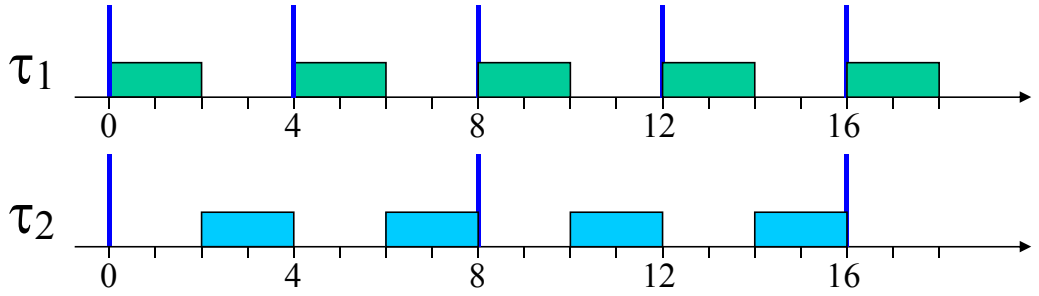
$$U_{ub} = \frac{2}{4} + \frac{4}{10} = 0.9$$



- The upper bound U_{ub} depends on the specific task set.

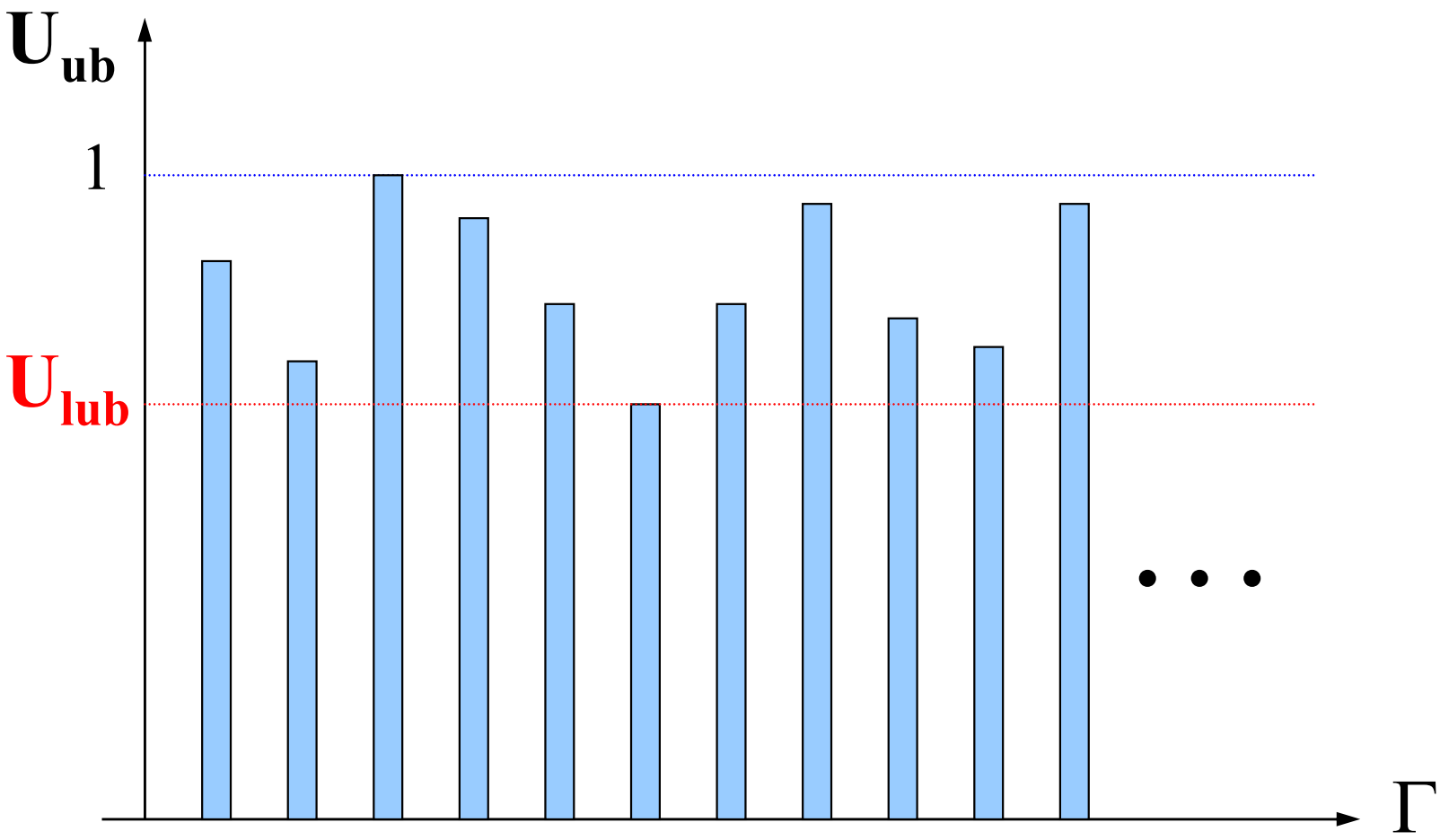
…and Yet Another One

$$U_p = \frac{2}{4} + \frac{4}{8} = 1$$



- The upper bound U_{ub} depends on the specific task set.
  - In these examples: $U_{ub} = 0.833, 0.9, 1, \ldots$.
  - Is there anything more we can tell about U_{ub}?

The Least Upper Bound



A Sufficient Condition

- Given a task set Γ:
  - If $U_p \leq U_{lub}$, Γ is **certainly schedulable** with RM
  - If $U_{lub} < U_p \leq 1$, we **cannot say anything** about Γ's feasibility

Least Upper Bound for RM

- Liu & Layland, 1973
- Given a set of *n* periodic tasks:

$$U_{\text{lub}}^{RM} = n\left(2^{1/n} - 1\right)$$

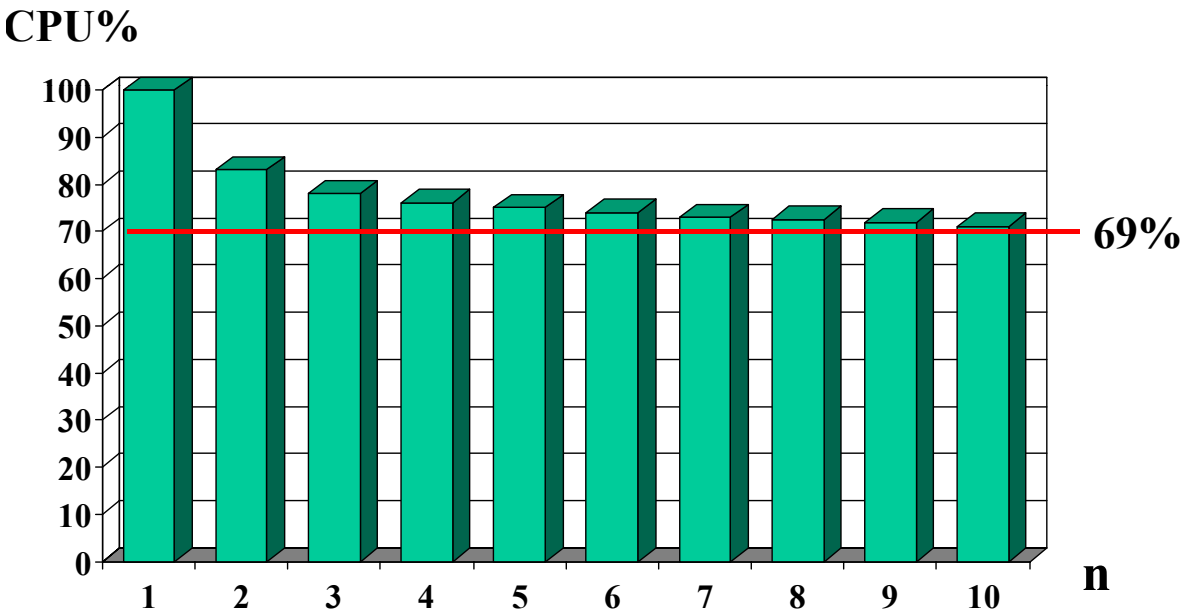
for $n \to \infty$ $U_{lub} \to ln\ 2$

- Used for **RM guarantee test**:
  - Compute processor utilization
  - Verify that it does not exceed the least upper bound

RM Schedulability

- U_{lub}^{RM} for n tasks

n	$n(2^{1/n}-1)$
1	1
2	0.828
3	0.780
4	0.757
5	0.743
10	0.718
20	0.705
50	0.698
100	0.696
1000	0.693



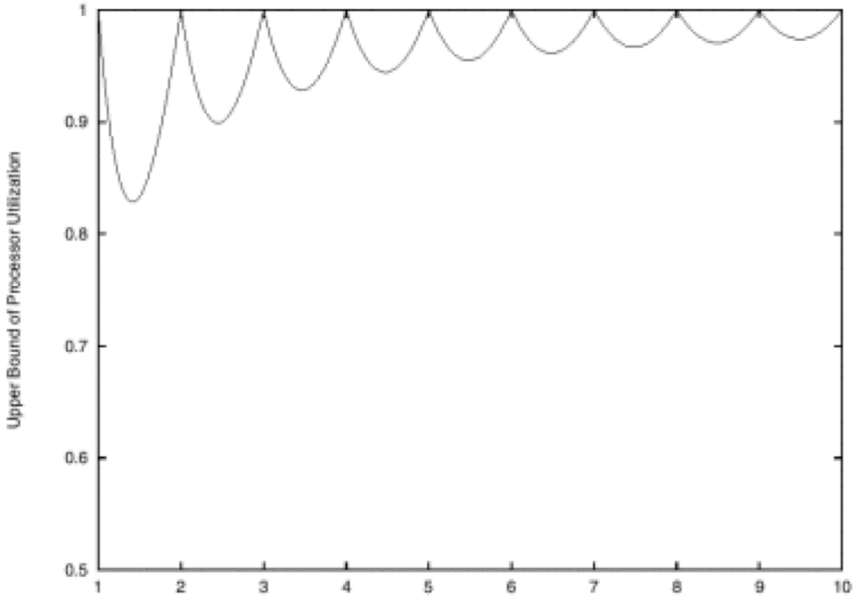
RM Schedulability

- U_{ub}^{RM} for 2 tasks is a function of $F = \lfloor T_2/T_1 \rfloor$

n	$n(2^{1/n}-1)$
1	1
2	0.828
3	0.780
4	0.757
5	0.743
10	0.718
20	0.705
50	0.698
100	0.696
1000	0.693

F	U^*
1	0.828
2	0.899
3	0.928
4	0.944
5	0.954
10	0.976
20	0.988
50	0.995
100	1.000
1000	1.000

$$U^* = 2\left(\sqrt{F(F+1)} - F\right)$$



Real-Time Systems Laboratory

A Tighter Least Upper Bound for RM

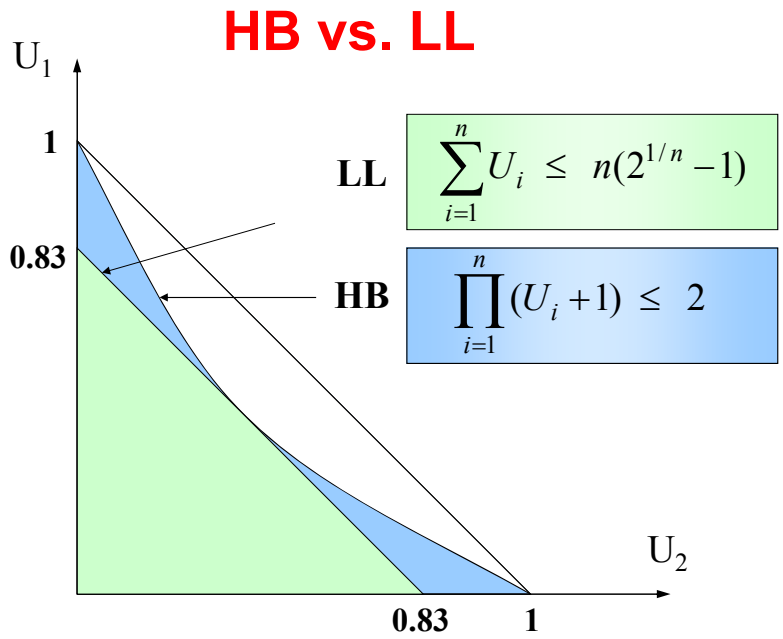
- Bini & Buttazzo[2], 2000
- **Hyperbolic Bound**
- A set of n periodic tasks is schedulable with RM if:

$$\prod_{i=1}^{n}(U_i + 1) \leq 2$$

- It is a **"tight" bound**: given any set of utilizations that violate the HB, it is always possible to produce an unfeasible task set with those utilization.

Comparison

- The hyperbolic bound can be compared with the Liu-Layland bound in the "task utilization space" (U-space)



- The gain achieved by HB over LL increases with *n* (it tends to √2)

Exercise 4.1

	C_i	T_i
τ_1	2	6
τ_2	2	8
τ_3	2	12

Verify the schedulability and construct the RM schedule.

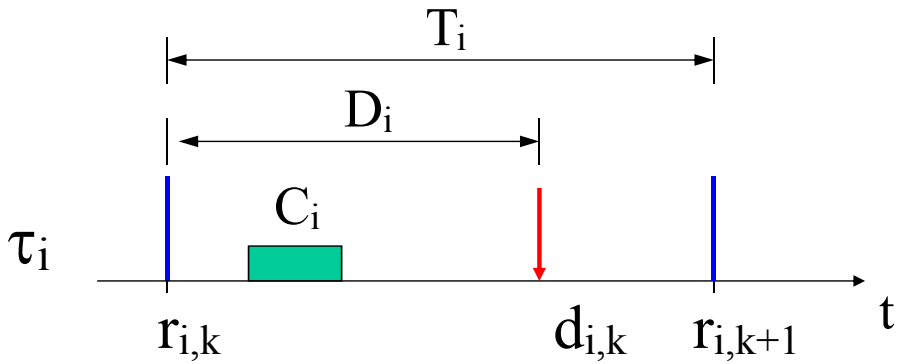
Exercise 4.2

	C_i	T_i
τ_1	3	5
τ_2	1	8
τ_3	1	10

Verify the schedulability and construct the RM schedule.

Extension to Tasks with D < T

- More modeling possibilities. For instance:
  - Tasks with jitter constraints;
  - Activities with shorter response time with respect to their period.

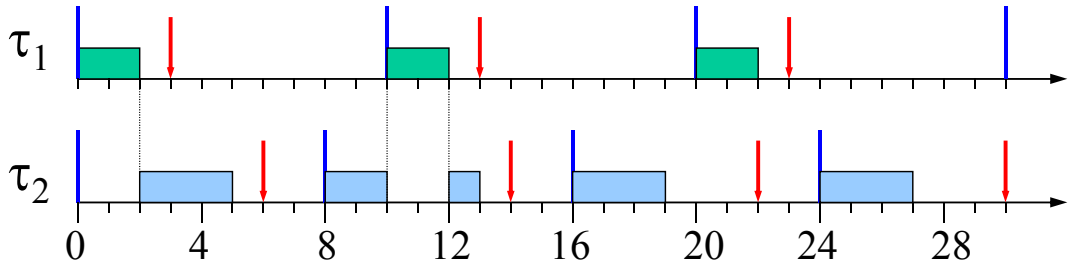


- **Deadline Monotonic (DM):**
  - $P_i \propto 1/D_i$ (static)
- **Earliest Deadline First (EDF):**
  - $P_i \propto 1/d_{i,k}$ (dynamic)

Deadline Monotonic

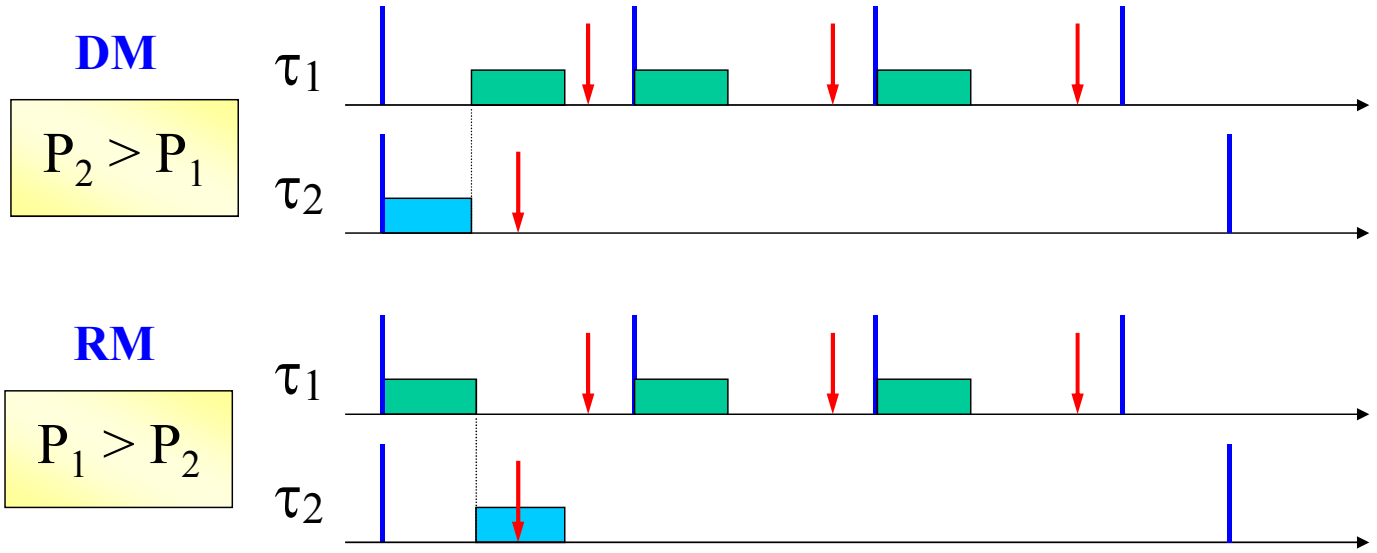
"Assign each task a fixed priority inversely proportional to its relative deadline"

(Leung & Whitehead 1982)

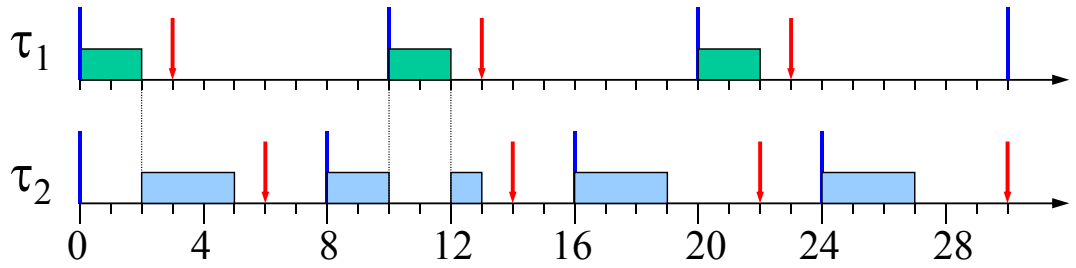


Deadline Monotonic is Optimal

- If $D_i < T_i$, if a task set is schedulable by some fixed priority assignment, then it is also schedulable by DM.



Good News, Bad News



- Good news → DM gives **optimal** priority assignment.
- Bad news → problem with the **utilization bound**:
  - $U_p = \sum_{i=1}^{n} \frac{C_i}{D_i} = \frac{2}{3} + \frac{3}{6} = 1.16 > 1$
  - but the task set *is schedulable*
    - CPU workload overestimated
    - RM guarantee test **too pessimistic** for DM

Seen so far…

- Problem definiton (periodic task scheduling)
- Concepts (processor utilization, critical instant, upper bound)
- Scheduling Algorithms
  - Theoretical (Proportional Share)
  - Paper & pencil (Timeline Scheduling)
  - Fixed Priority (optimal)
    - Rate Monotonic if D=T
    - Deadline Monotonic if D<T
- Shedulability Analysis
  - Least Upper Bound
    - Liu-Layland
    - Hyperbolic Bound
- Next?

Response Time Analysis

- A sufficient and necessary schedulability test for DM (Audsley et al., 1990)
- For each task τ_i, compute the **interference** (preemption) due to higher priority tasks:

$$I_i = \sum_{D_k < D_i} C_k$$

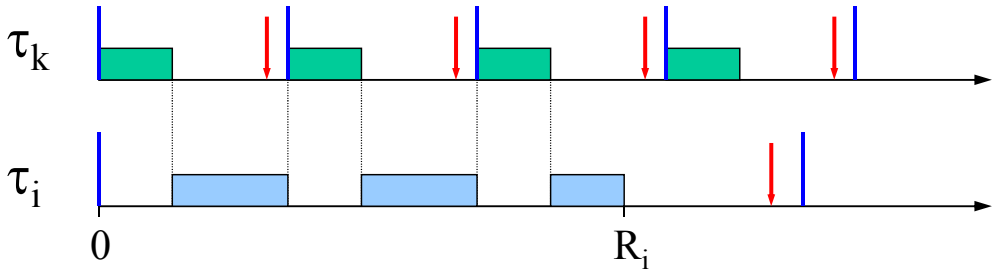
- Compute its **response time** as:

$$R_i = C_i + I_i$$

- Verify whether $R_i \leq D_i$

Computing the Interference

- Assume tasks are ordered by increasing relative deadlines
  - i < j if and only if $D_i \le D_j$



Interference of τ_k on τ_i in the interval $[0, R_i]$:

$$I_{ik} = \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Interference of high priority tasks on τ_i:

$$I_i = \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Computing the Response Time

$$R_i = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Iterative solution:

$$\begin{cases} R_i^0 = \sum_{k=1}^{i} C_k \\ R_i^s = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i^{(s-1)}}{T_k} \right\rceil C_k \end{cases}$$

iterate while $R_i^s > R_i^{(s-1)}$

Exercise 4.7

	C_i	D_i	T_i
τ_1	2	5	6
τ_2	2	4	8
τ_3	4	8	12

Verify the schedulability and construct the DM schedule.

Exercise 4.3

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	10

Verify the schedulability and construct the RM schedule.

Exercise 4.4

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	8

Verify the schedulability and construct the RM schedule.

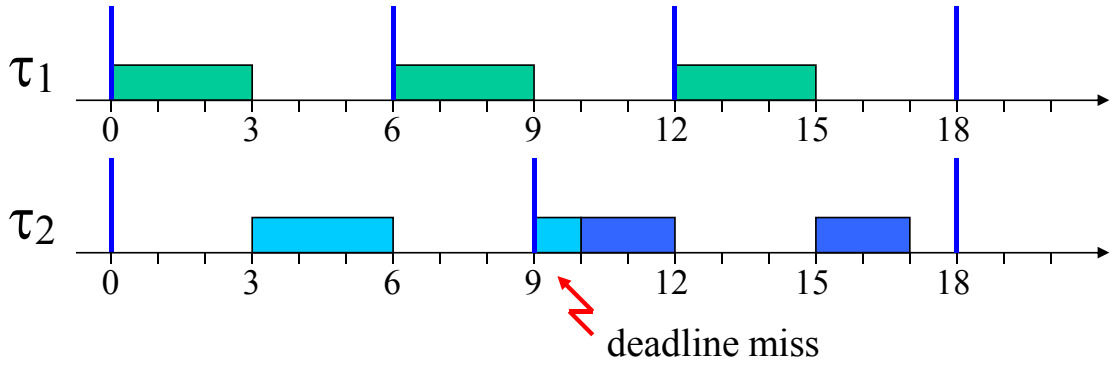
Dynamic Priority Scheduling

EDF

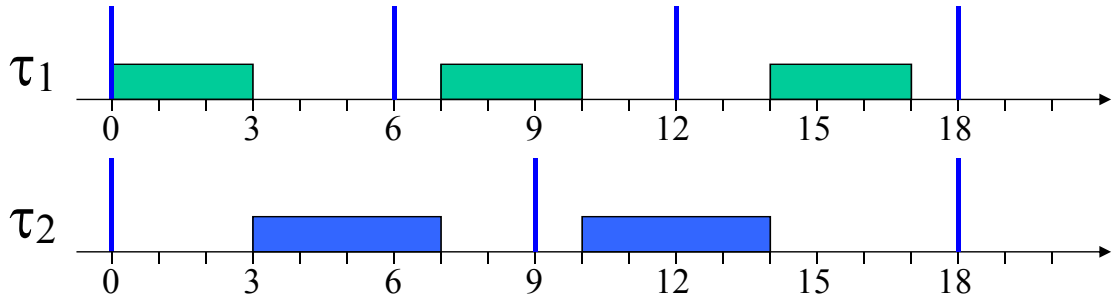
- $D_i = T_i$
- No constraints

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$

- **RM**: unfeasible.



- **EDF**: feasible!
- With EDF, **any** task set can utilize the processor **up up to 100%**



EDF Optimality & Schedulability

- **Optimality.** EDF is optimal among all algorithms (Dertouzos 1974)
  - If there exists a feasible schedule for Γ, then EDF will find a feasible schedule
  - If Γ is not schedulable by EDF, then it cannot be scheduled by any algorithm

  (result independent of periodicity)

- **Schedulability**. For a set of *n* periodic tasks,

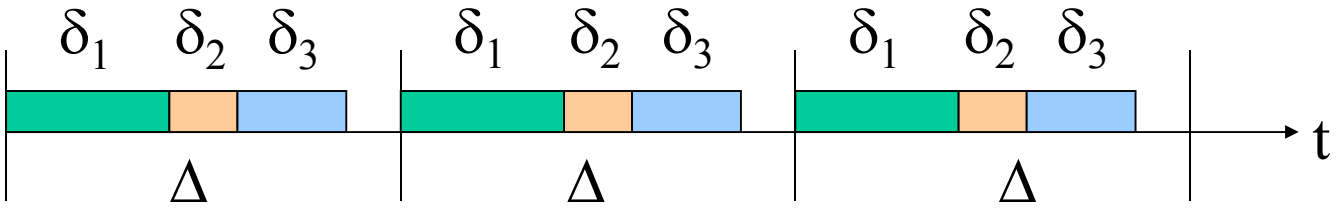
$$U_{\text{lub}}^{EDF} = 1$$

(Liu & Layland 1973)

  - In other words, a task set Γ is EDF-schedulable **if and only if** $U_p \leq 1$

EDF Schedulability

- Necessity: schedulable → $U_p \leq 1$ is trivial
- To prove sufficiency: $U_p \leq 1$ → schedulable
  1. We find any algorithm for which the above condition holds
  2. Then, for the EDF optimality, we can say that the above condition also holds for EDF.



- Consider the algorithm which schedules in every interval of length Δ a fraction of task: $\delta_i = U_t \Delta$
  - Proportional Share Algorithm
  - Feasibility is ensured if $\sum_{i=1}^{n} \delta_i \leq \Delta$, that is, if $U_p \leq 1$.

Exercise 4.5

	C_i	T_i
τ_1	1	4
τ_2	2	6
τ_3	3	8

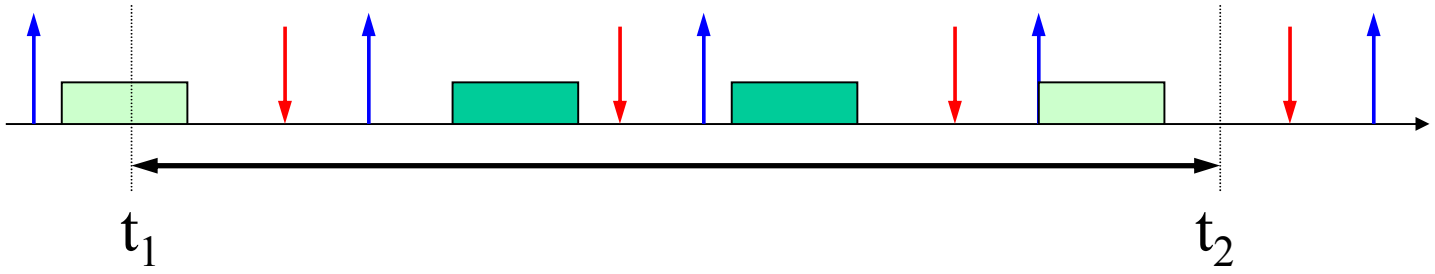
Verify the schedulability and construct the EDF schedule.

Schedulability with Dynamic Priority

- What if D ≤ T ?
- **Processor Demand Criterion**

"in any interval, the computation demanded by the task must be no greater than the available time"

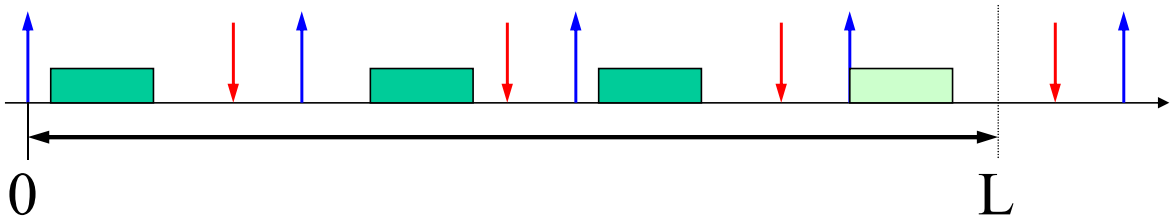
(Baruah, Rosier & Howell 1990)



- Demand of a task τ_i in $[t_1,t_2]$: amount of processing time $g_i(t_1,t_2)$ of all instances of τ_i that are activated in $[t_1,t_2]$ and must be completed in $[t_1,t_2]$.
- For the whole task set: $g(t_1,t_2)$

$$g(t_1,t_2) = \sum_{\substack{r_i \geq t_1 \\ d_i \leq t_2}} C_i$$

Processor Demand Test



- **Processor Demand** in [0,L] aka **Demand Bound Function**, **dbf(L)**

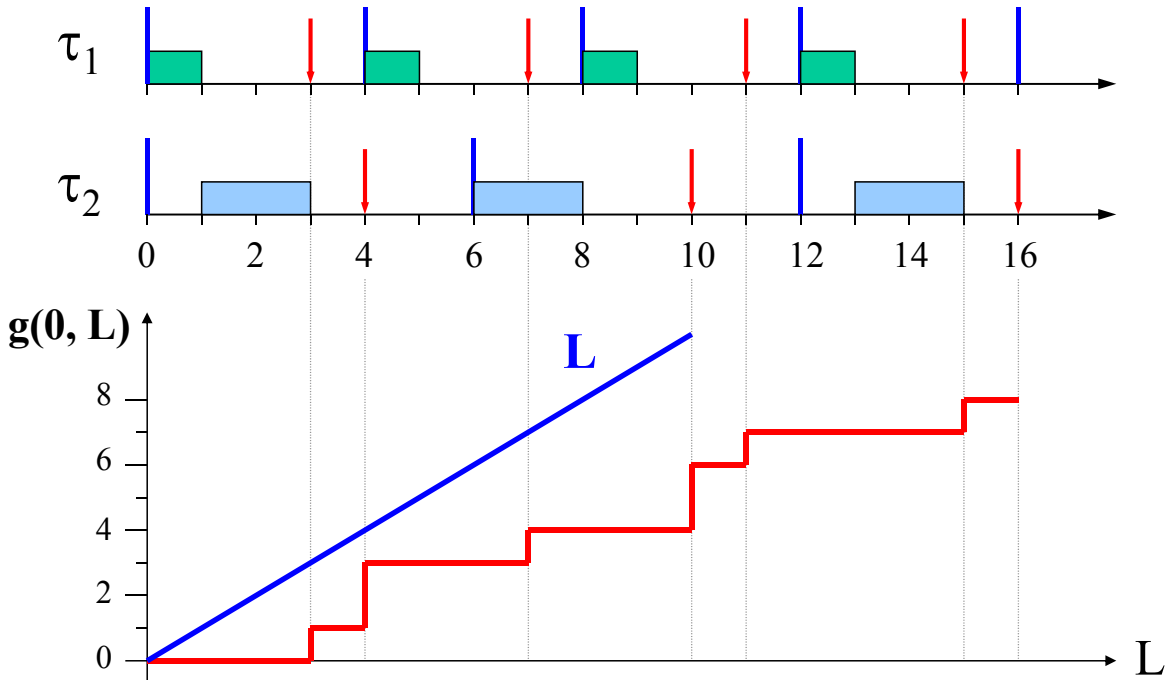
$$g(0,L) = \sum_{i=1}^{n} \left\lfloor \frac{L - D_i + T_i}{T_i} \right\rfloor C_i$$

- Demand Test

$$\forall L > 0, \quad g(0,L) \leq L$$

- *How can we bound the number of intervals in which the test has to be performed?*

Example



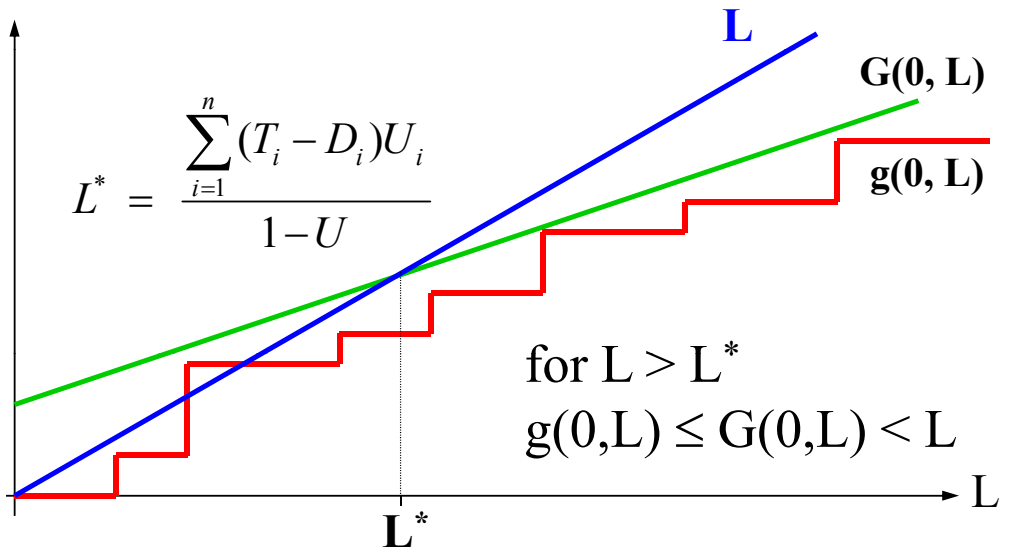
Bounding Complexity

- Some considerations:
  - Since g(0,L) is a step function, it suffices to check feasibility only at deadline points (d_k)
  - If tasks are synchronous and $U_p < 1$, it suffices to check feasibility only up to the hyperperiod $H = lcm(T_1, \ldots, T_n)$
  - $g(0,L) \le G(0,L)$ and, if $U < 1$, there exists an L^* for which $G(0,L^*) = L^*$.

$$G(0,L) = \sum_{i=1}^{n} \left(\frac{L + T_i - D_i}{T_i} \right) C_i$$

$$= \sum_{i=1}^{n} L \frac{C_i}{T_i} + \sum_{i=1}^{n} (T_i - D_i) \frac{C_i}{T_i}$$

$$= LU + \sum_{i=1}^{n} (T_i - D_i) U_i$$



Processor Demand Test

- A set of synchronous periodic tasks with relative deadlines less than or equal to periods **can be scheduled by EDF if and only if**
  - $U < 1$, and
  - $$\forall L \in D, \quad g(0, L) \leq L$$

$$D = \{d_k \mid d_k \leq \min(H, L^*)\}$$

$$\begin{cases} H = lcm(T_1, \ldots, T_n) \\ L^* = \dfrac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U} \end{cases}$$

$$g(0, L) = \sum_{i=1}^{n} \left\lfloor \frac{L - D_i + T_i}{T_i} \right\rfloor C_i$$

Exercise 4.6

	C_i	D_i	T_i
τ_1	2	5	6
τ_2	2	4	8
τ_3	4	8	12

Verify the schedulability and construct the EDF schedule.

Quizzes

- True or False?
  - If a task set is DM-schedulable, it is EDF-schedulable
  - If a task set is EDF-schedulable, its processor utilization U_p is below the HB
  - If a task set's processor utilization U_p is below the Liu-Layand bound, then U_p is also below the HB
  - A task set consisting of two tasks, τ_1 and τ_2, with $D_i=T_i$ and $T_1=2T_2$, is RM-feasible if and only if the total processor utilization is at most 1
  - Response Time Analysis can be used to study schedulability, even if relative deadline and pediod coincide (for all τ_i, $D_i=T_i$)
  - The Processor Demand Test can be used to study schedulability, even if relative deadline and pediod coincide
  - If for all τ_i, $D_i=T_i$, the Processor Demand Test and Response Time Analysis of a given task set give the same schedulability result

Summary

Periodic Task Scheduling

- **Three scheduling approaches**
  - Off-line construction (timeline)
  - Fixed priority (RM, DM)
    - $P_i \propto 1/T_i$
    - $P_i \propto 1/D_i$
  - Dynamic priority (EDF)
    - $P_i \propto 1/d_{i,k}$, $d_{i,k} = r_{i,k} + D_i$

- **Three analysis techniques:**

Processor Utilization Bound	$U \leq U_{lub}$	O(n)
Response Time Analysis	for all i, $R_i \leq D_i$	*
Process Demand Criterion	for all L, $g(0,L) \leq L$	*

* pseudo-polynomial complexity

RM vs. EDF

- **RM**
  - Simpler to implement in commercial operating systems
    - fixed priorities
  - More predictable during overloads
    - highest priority tasks are known
- **EDF**
  - More efficient
  - Reduces context switches
  - Better responsiveness in handling aperiodic tasks
  - Period rescaling during permanent overloads