Dynamic Priority Scheduling

Overview

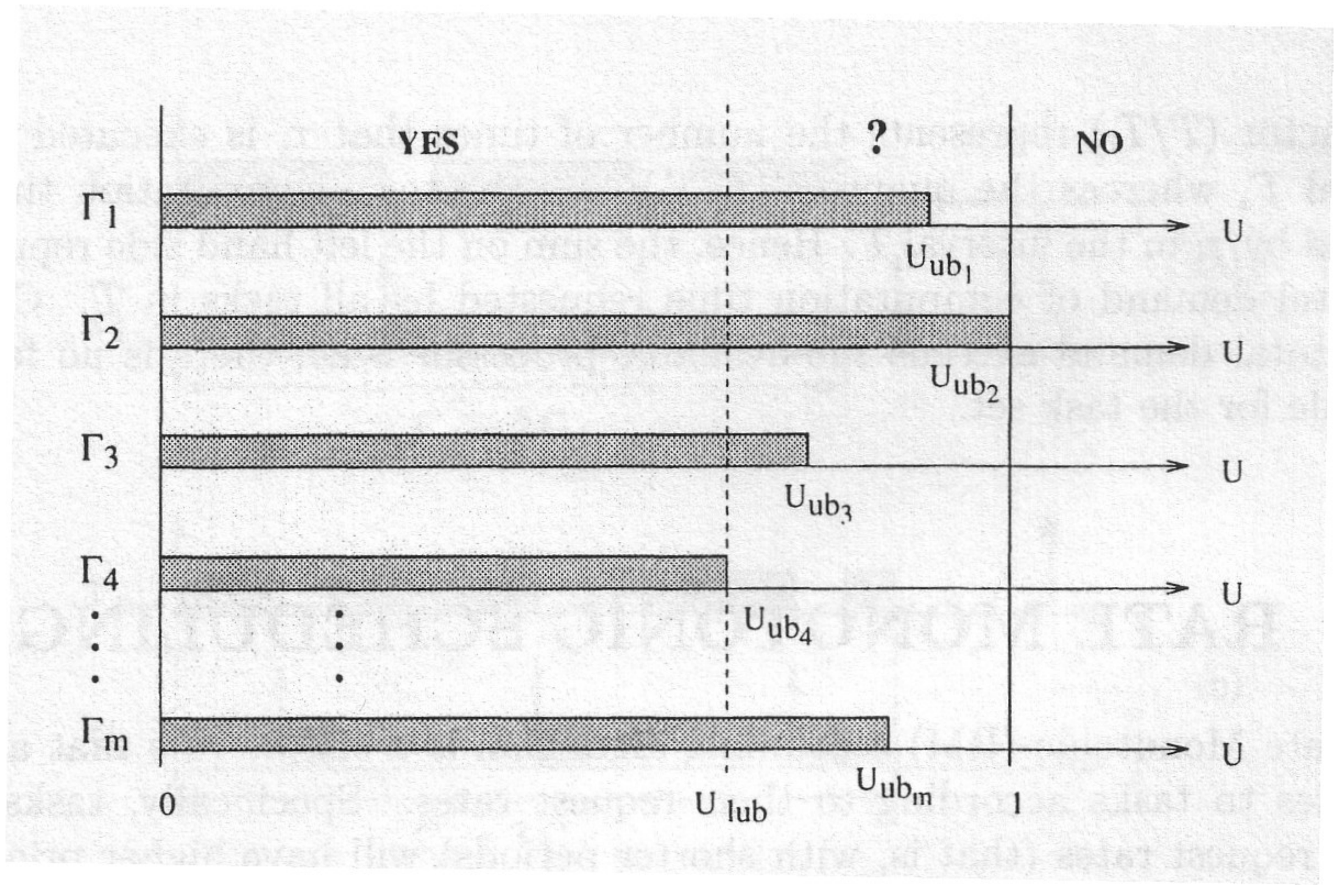
- EDF
  - optimality (Done!)
  - schedulable utilization bound
  - time demand analysis

Dynamic-priority scheduling

- How to assign Priorities? – we already proved that assigning priority based on the absolute deadline is optimal
- How to check the schedulability?

Utilization Bound Check

- For a given algorithm A, we are interested in finding its schedulability bound (e.g., the schedulability bound of EDF is 1)



Processor utilization factor

- More formally, we want to find the *least upper bound* $U_{lub}(A)$ of the processor utilization factor

$$U_{\text{lub}}(A) = \min_{\Gamma} U_{ub}(\Gamma, A)$$

- For example, considering the RM algorithm, $U_{\text{lub}}(\text{RM}) = n\,(2^{1/n} - 1)$. Now, we know how it can be derived.

- Quiz: what is the value of $U_{ub}(\Gamma, EDF)$?

Schedulability of EDF

- Theorem: given a set of *n* (independent) periodic tasks, each deadline will be met if and only if the total utilization of the tasks is no greater than 1. That is:

$$\frac{e_1}{p_1} + \frac{e_2}{p_2} + \ldots + \frac{e_n}{p_n} \leq 1$$

- How in the world can we prove that?

Proving Steps

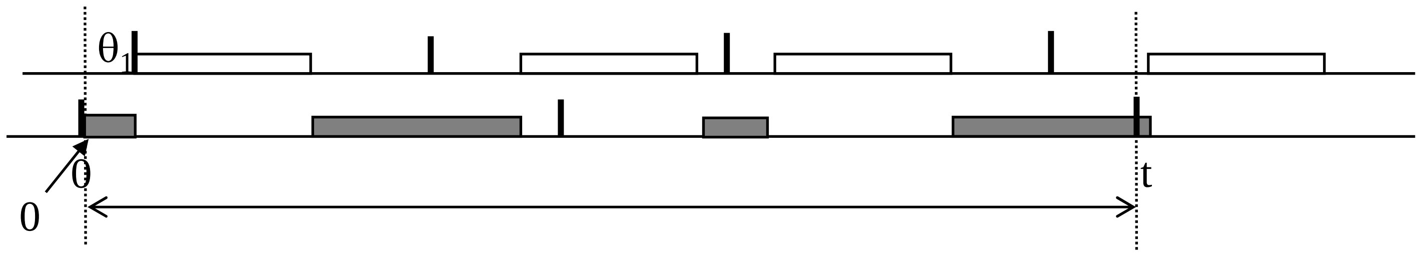
- Necessity: Schedulable $\rightarrow$Total Utilization ≤ 1
  - Total Utilization $> 1 \rightarrow$ Not Schedulable: easy!
  - How prove?
- Sufficiency: Total Utilization $\leq 1 \rightarrow$ Schedulable
  - Prove that *if a job misses deadline, then Total Utilization >1*
  - Two cases
    - Easy Case
    - Difficult Case

Counting Execution Times (Easy Case)

Suppose a job (T_{22}) misses the deadline $\Rightarrow$ "sum of executions prior to *t* " > *t*

Hint: We know that the completion time of a job is the sum of its own execution time plus the interference due to other jobs with earlier deadline. So let's count them.

We need to count all the executions of jobs prior to *t*. The ***execution time of jobs with deadlines prior to t*** is easy to count. So are the execution times of jobs of T_2 by time *t*.



At the deadline of T_{22} at *t*, the deadline of T_{13} is before *t* and the total execution of each task can be expressed neatly

$$T_1 : \left\lfloor \frac{t-\theta_1}{p_1} \right\rfloor e_1; \quad T_2 : \left\lfloor \frac{t}{p_2} \right\rfloor e_2$$

Getting a foothold

If a job (T22) misses the deadline ⇒ "sum of executions prior to *t* " > *t*

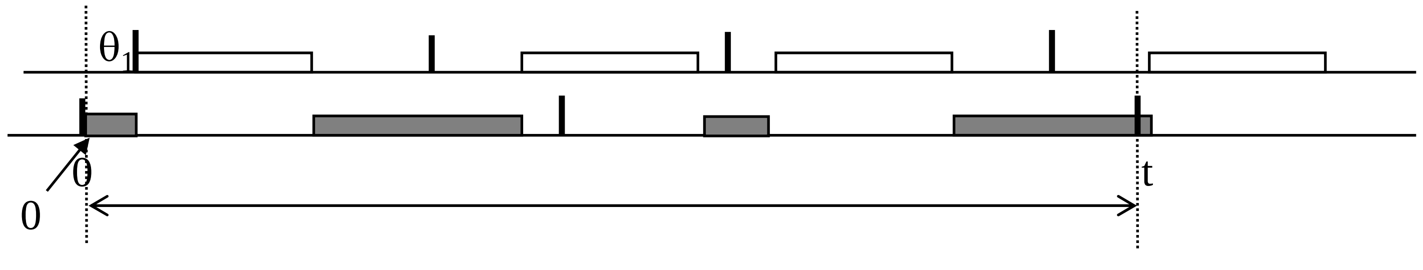
$$\text{"Sum of executions prior to } t\text{"} = \left\lfloor \frac{t-\theta_1}{p_1} \right\rfloor e_1 + \left\lfloor \frac{t}{p_2} \right\rfloor e_2 > t$$

$$\left\lfloor \frac{t}{p_1} \right\rfloor e_1 + \left\lfloor \frac{t}{p_2} \right\rfloor e_2 > t$$

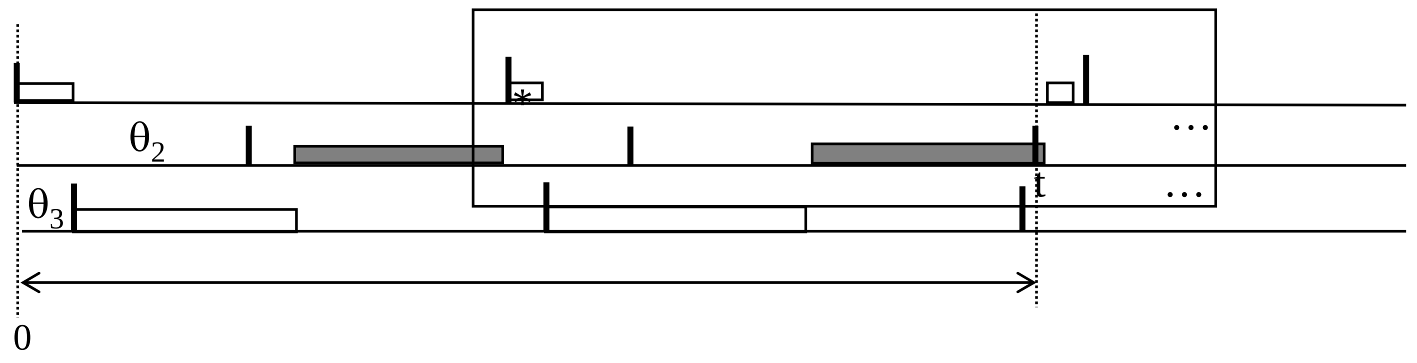
$$\frac{t}{p_1} e_1 + \frac{t}{p_2} e_2 > t$$

$$\frac{e_1}{p_1} + \frac{e_2}{p_2} > 1$$

If a job (T22) misses the deadline ⇒ Total Utilization > 1



Counting Execution Times (Difficult Case)



When T_{22} is sandwiched by the release time and deadline of T_{12}, we do not have neat expression of jobs of T_1 in terms of *t*.

$$T_1 : \left\lfloor \frac{t}{p_1} \right\rfloor e_1 + *; \quad T_2 : \left\lfloor \frac{t-\theta_2}{p_2} \right\rfloor e_2; \quad T_3 : \left\lfloor \frac{t-\theta_3}{p_3} \right\rfloor e_3$$

- Can we still prove the followings?
  - If a job (T_{22}) misses the deadline $\Rightarrow$ "sum of executions prior to *t* " > *t*
  - "sum of executions prior to *t*" > *t* $\Rightarrow$ Total Utilization > 1

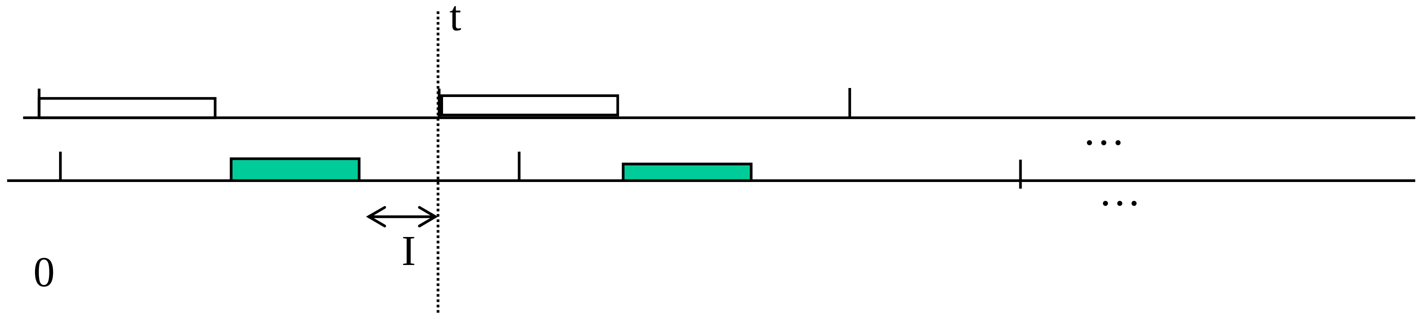
When you have a hammer

- There is who has a hammer. To him, everything looks like a nail. He uses the hammer for everything.

- If you have neat result, call it a lemma (hammer?!). Next, work hard and transform new situations to one that looks like a nail, so that you can *hammer it with your lemma*.

- Question: We have a special pattern and we’ve got a neat result. What should we do when we have a new pattern with *?
- Answer: Transform the new pattern into one that we can hammer it with our lemma. (Remove *)

Key Facts: Idle Interval Lemma



• An idle interval is terminated by the first arrival of a job labeled here as t. The schedulability of jobs after t is not affected by jobs before t, since jobs before t cannot delay the execution of jobs after t.

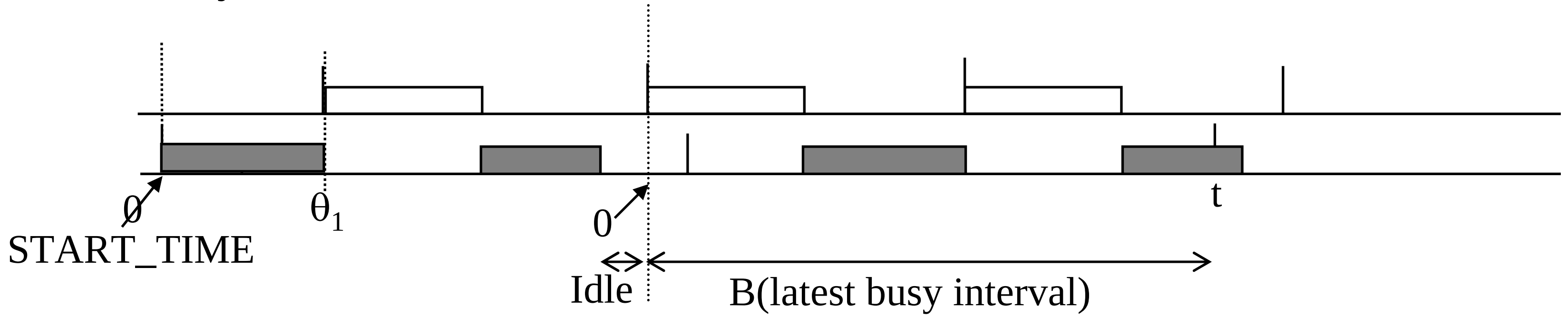
• Thus, we can choose t as the time 0 for studying the schedulability of jobs after t.

Key Facts: Busy Interval

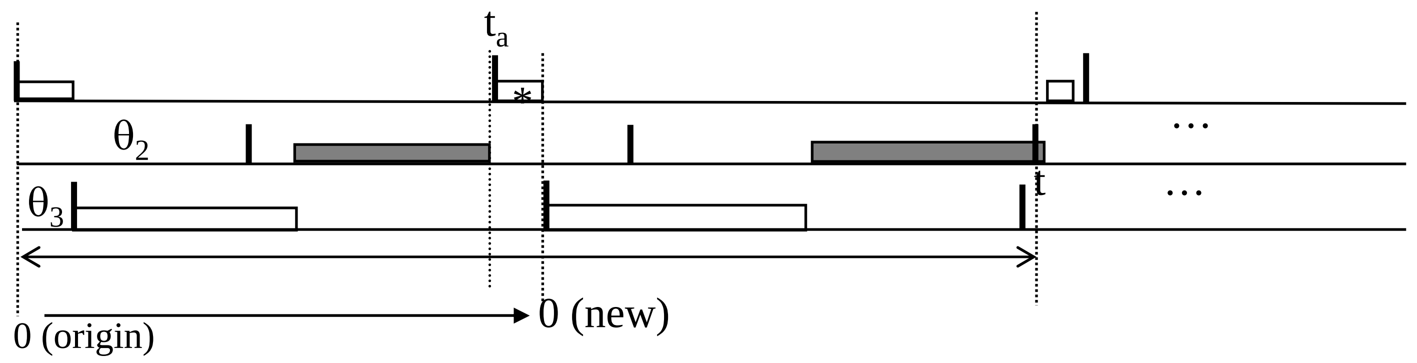
•Suppose that we suspect that some job of T2 may miss its deadline at time t. To keep the time line short and thus things simple, we shall invoke the Idle Time Lemma.

•From release time of this suspected job, we scan the timeline towards START_TIME. When we find the last idle interval, I (if any), between [START_TIME, t], we pick the end of I as the time origin, 0 for schedulability analysis.

•This interval is known as the (latest) busy interval. <u>All the scheduling history prior to this busy interval is irrelevant</u>. The notion of (latest) busy interval is very useful in studies concerning with time line, including scheduling theory and queuing theory.



Transform Difficult One to Easy One

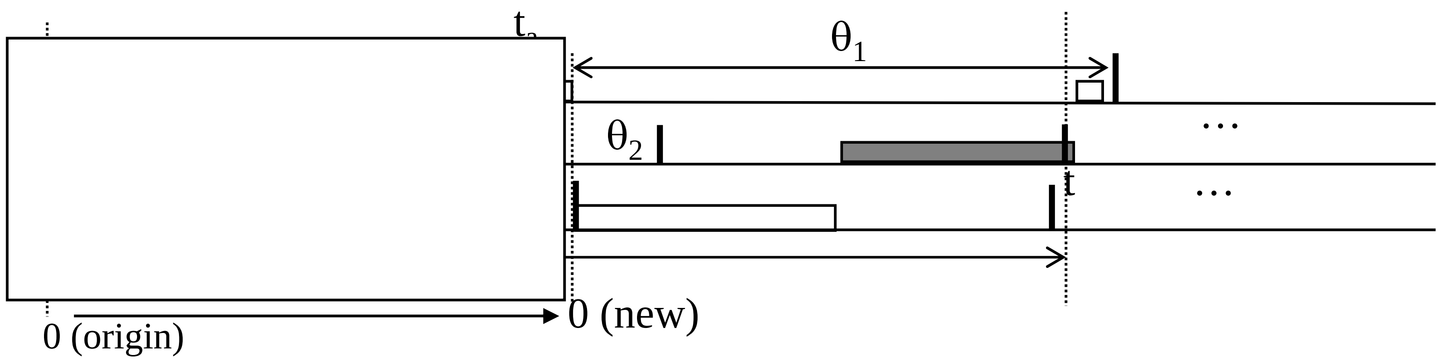


•The fact that * can be executed implies that all jobs with deadlines shorter than *t* have been completed before t_a. (Why?) ---- Idle Interval

•After t_a, when the first job with deadline shorter than t shows up, we consider it as “new 0”. The scheduling history prior to “new 0” can now be ignored. ---- Latest Busy Interval

•So we get back to the old pattern.

Transform Difficult One to Easy One



So, the scheduling history prior to “new 0” can now be ignored. So we get back to the old pattern.

$$T_1 : \left\lfloor \frac{t-\theta_1}{p_1} \right\rfloor e_1 < 0 \text{ Ignored!}; \quad T_2 : \left\lfloor \frac{t-\theta_2}{p_2} \right\rfloor e_2; \quad T_3 : \left\lfloor \frac{t}{p_3} \right\rfloor e_3$$

How about N tasks?

- Theorem: Given N independent periodic tasks with deadline at the end of the period, every job can meet its deadline provided that the total utilization is no more than 1.

- Proof:……

Time-Demand Analysis
(Alternative way for schedulability check)

- Check if Time-Demand is smaller than Time-Supply at all the times.

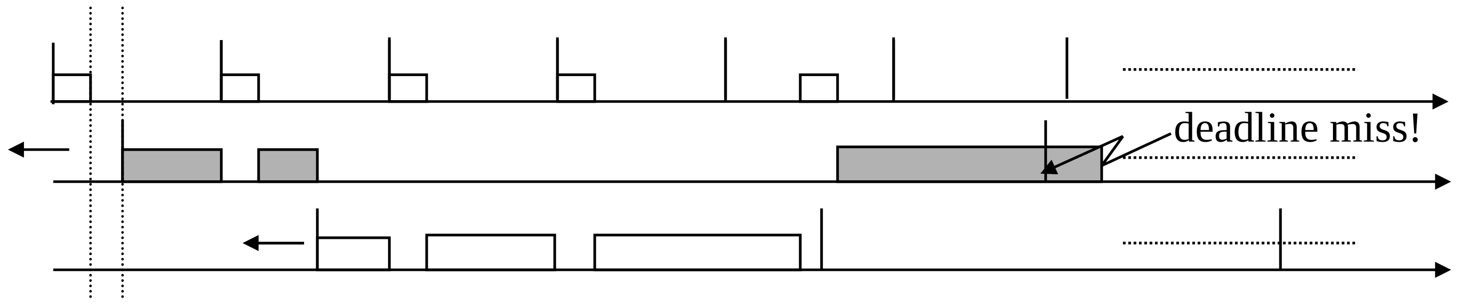
- Can we check this
  - For all possible arrival patterns?
  - Infinitely?

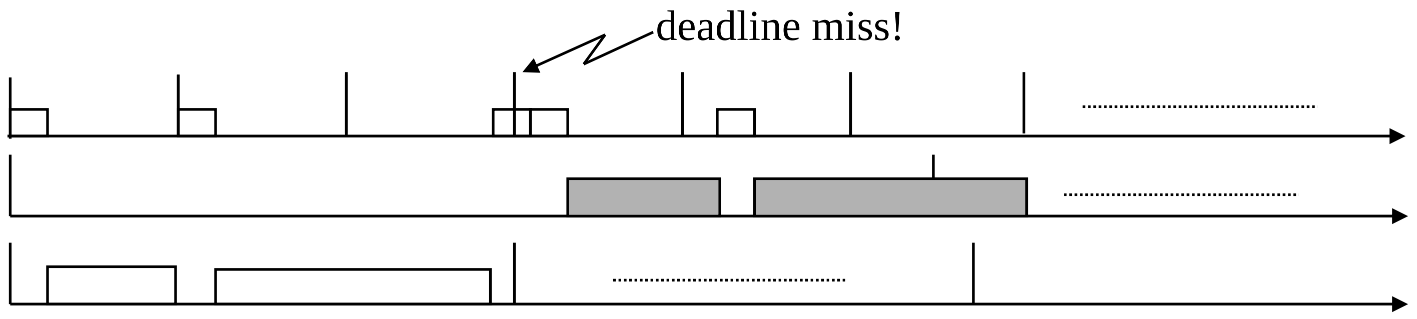
Worst-case pattern with EDF

- When EDF is used to schedule a set of tasks on a processor, if there is an overflow for a certain arrival pattern, then there is an overflow without idle time prior to it in the pattern in which all tasks are released synchronously.

- Proof:….

Worst-case pattern with EDF





- Quiz: what is the difference between the worst-case pattern of EDF and the critical instance theorem for static priority scheduling algorithms?

Finite Check

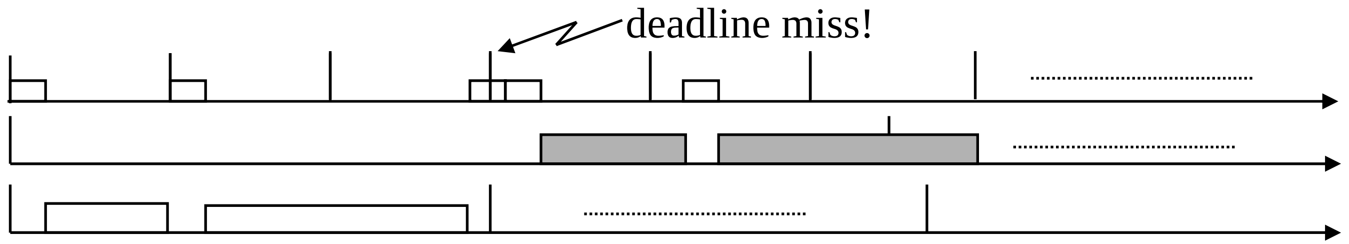
- The schedulability of a task set scheduled according to EDF can be checked by analyzing the worst-case pattern (synchronous release times) and verifying if any deadline is missed within [0, min(idle time, Hyperperiod)]
- Why?

Time-demand approach

- A set of periodic tasks is schedulable by EDF if and only if for all L, 0 ≤ L ≤ min[idle time, Hyperperiod],

Processor demand within [0, L]

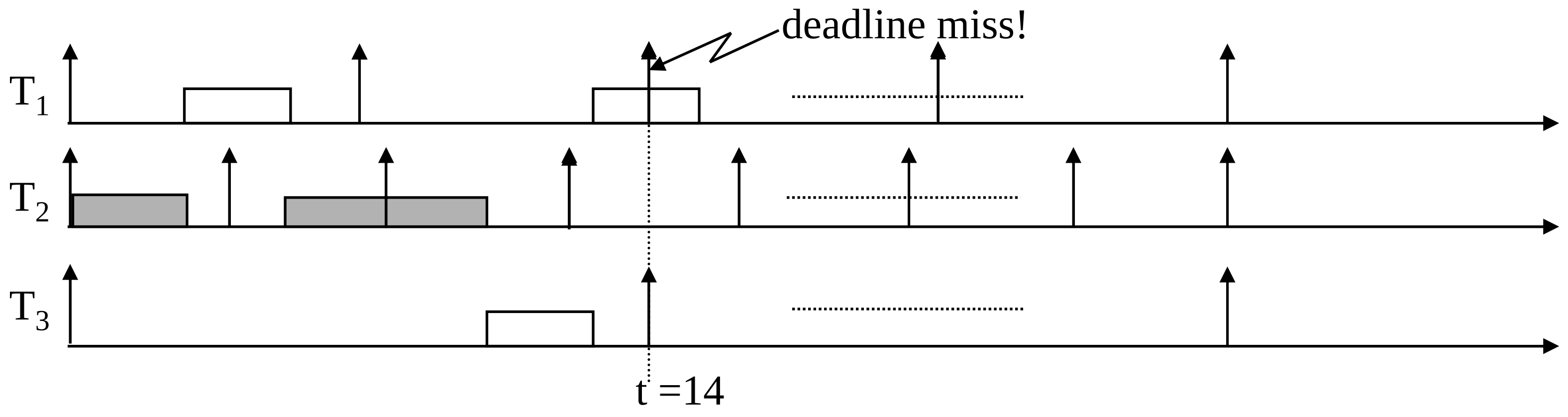
$$L \geq D[0, L] = \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor \cdot e_i$$



- Quiz:do we really need to check the processor demand “for every L”?

Class exercise 1

- Suppose that we have 3 tasks with the following periods and execution times:
  - $\{T_1\ (p_1 = 7,\ e_1 = 2),\ T_2\ (p_2 = 4,\ e_2 = 3),\ T_3\ (p_3 = 14,\ e_3 = 2)\}$
  - The task set utilization is $U=1.18 > 1$
  - Let's apply the processor demand approach!
  - $L = 4 \Rightarrow D[0, 4] = 3 < L$
  - $L = 7 \Rightarrow D[0, 7] = \lfloor 7/7 \rfloor * 2 + \lfloor 7/4 \rfloor * 3 + \lfloor 7/14 \rfloor * 2 = 5 < L$
  - $L = 8 \Rightarrow D[0, 8] = \lfloor 8/7 \rfloor * 2 + \lfloor 8/4 \rfloor * 3 + \lfloor 8/14 \rfloor * 2 = 8 = L$
  - $L = 12 \Rightarrow D[0, 12] = \lfloor 12/7 \rfloor * 2 + \lfloor 12/4 \rfloor * 3 + \lfloor 12/14 \rfloor * 2 = 11 < L$
  - $L = 14 \Rightarrow D[0, 14] = \lfloor 14/7 \rfloor * 2 + \lfloor 14/4 \rfloor * 3 + \lfloor 14/14 \rfloor * 2 = 15 > L$
  - We got $D[0, 14] > 14 \Rightarrow$ we found the deadline miss!



Time-demand approach (proof)

- **Proof**:

  The theorem is proved by showing that the processor demand equation is equivalent to the classical schedulability condition:

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1 \quad \Leftrightarrow \quad L \geq \sum_{i} \left\lfloor \frac{L}{p_i} \right\rfloor \cdot e_i$$

  Hence, first of all, we prove that:

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1 \quad \Rightarrow \quad L \geq \sum_{i} \left\lfloor \frac{L}{p_i} \right\rfloor \cdot e_i$$

  If U ≤ 1, then for all L (L ≥ 0),

$$L \geq UL = \sum_{i=1}^{n} \frac{L}{p_i} e_i \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor \cdot e_i$$

Processor demand approach

Then, we prove that:

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1 \qquad \Leftarrow \qquad L \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor \cdot e_i$$

we prove it by contradiction; hence, suppose that U > 1, there exists a L ≥ 0 such that L < D[0, L]

if U > 1, then for L = *lcm*(p_1,p_2,…,p_n),

$$L < UL = \sum_{i=1}^{n} \frac{L}{p_i} e_i = \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor \cdot e_i$$

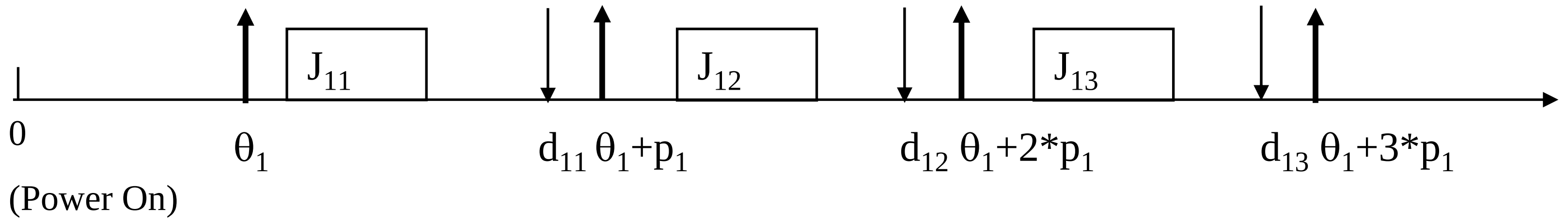
Why time-demand analysis is useful ?

- The processor demand approach is an alternative way for checking the schedulability of a task set under EDF.

- Well, we already have the $U \leq 1$ as schedulability condition; way do we need another equivalent test?

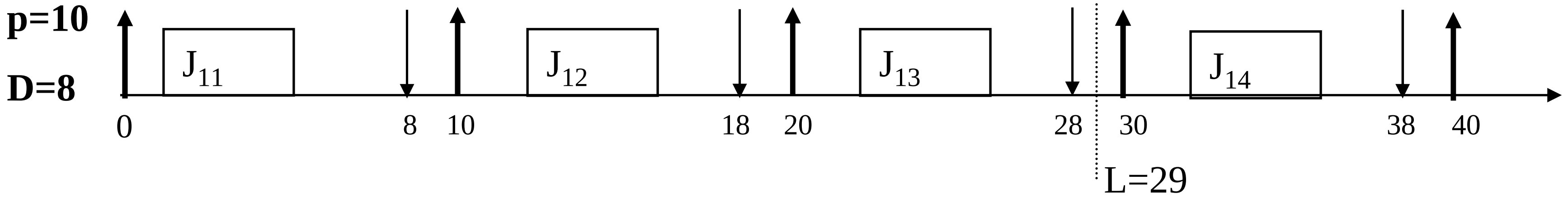
- Answer: the processor demand approach is more powerful and it allows to determine the schedulability when the classical condition cannot be applied.

- An example: task sets with deadlines less then periods!



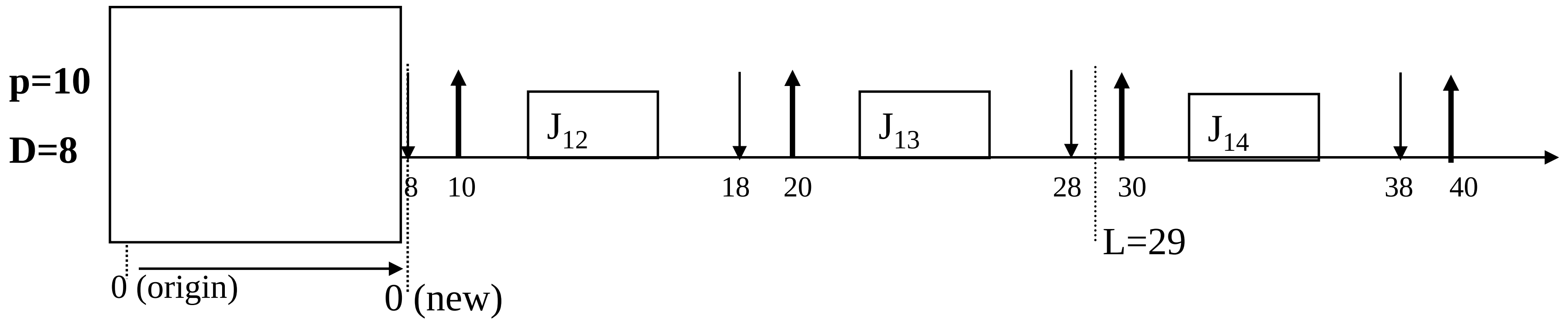
EDF with deadlines less than periods

- How can we compute the processor demand D[0, L] when $D_i < p_i$?



- Quiz: is it correct to use the already known formula? $D[0,L] = \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor \cdot e_i$

EDF with deadlines less than periods

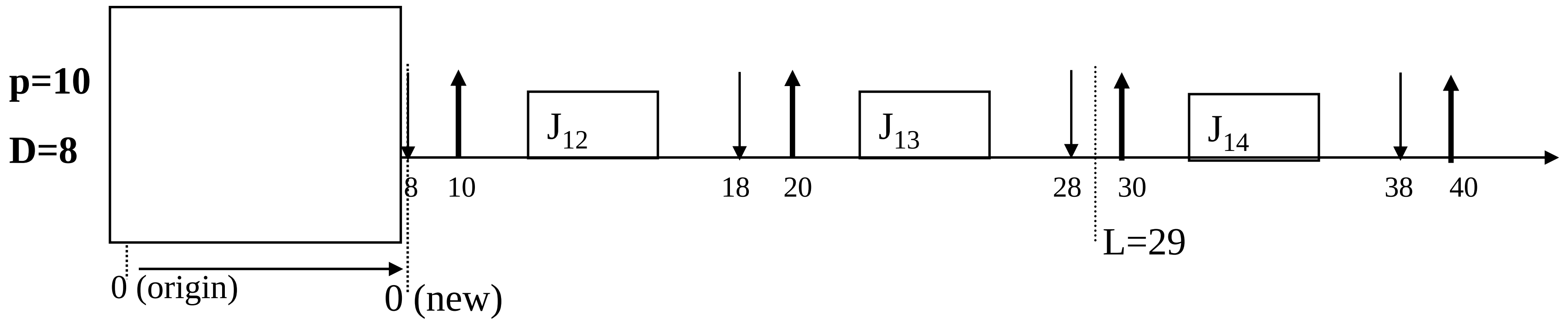


- First step: we shift the interval origin by a D amount. So we get:

$$D[0,L] = \sum_{i=1}^{n} \left\lfloor \frac{L - D_i}{p_i} \right\rfloor \cdot e_i$$

- However, the first task instance remains outside of the processor demand; we fix it by adding 1

EDF with deadlines less than periods



- A set of periodic tasks with deadlines less than periods is schedulable by EDF if and only if for all L, $0 \le L \le \min[\text{idle time, Hyperperiod}]$,

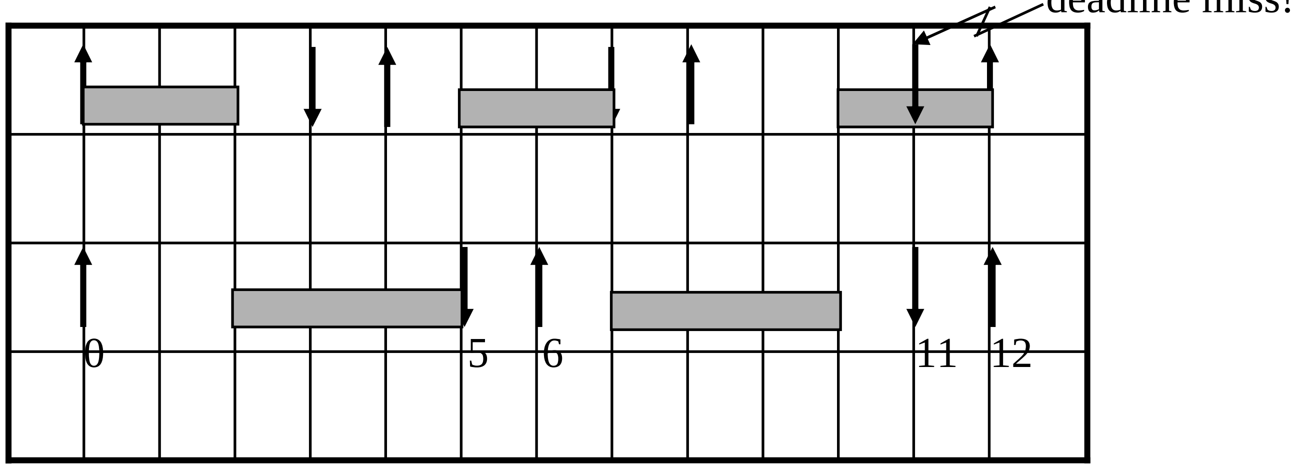
$$L \ge \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{p_i} \right\rfloor + 1 \right) \cdot e_i$$

Class exercise 2

- Suppose that we have 2 tasks with the following periods, relative deadlines and execution times:
  - $\{T_1\ (p_1 = 4, D_1=3, e_1= 2), T_2\ (p_2 = 6, D_2=5\ e_2 = 3)\}$

  - The task set utilization is U=1
  - Is the task set schedulable?
  - Let's apply the processor demand approach!

  - L = 5 ➔ $D[0, 5] = (\lfloor(5-3)/4\rfloor +1)*2 + (\lfloor(5-5)/6\rfloor +1)*3 = 5 = L$
  - L = 7 ➔ $D[0, 7] = (\lfloor(7-3)/4\rfloor +1)*2 + (\lfloor(7-5)/6\rfloor +1)*3 = 7 = L$
  - L = 11 ➔$D[0, 11] = (\lfloor(11-3)/4\rfloor +1)*2 + (\lfloor(11-5)/6\rfloor +1)*3 = 12 > L$
  - We got D[0, 11] > 11 ➔ we found a deadline miss!



Summary

- All of the above schedulability check works only under limited conditions
  - Preemptable at any time
  - Context switch overhead is negligible
  - Scheduling decision is made immediately upon jobs release and completion
- Practical Issues
  - What if the deadline is earlier than the period?
  - What if there is a non-preemptable code section (e.g., system call)?
  - What if the context switch overhead is not negligible?
  - Tick scheduling?

EDF with deadlines less than periods

- Quiz: can you find an easy but sufficient schedulability condition for a task set with deadlines less than periods?

$$D_i < p_i$$

$$\sum_{j=1}^{n} \frac{e_j + p_j - D_j}{p_j} \leq 1; \text{ increase execution time}$$

$$\sum_{j=1}^{n} \frac{e_j}{D_j} \leq 1; \text{ decrease period}$$

Non-preemptable code section

- a non-preemptable code section (NPS) of a low priority task ***blocks*** high priority task
  - How to take this into account in utilization bound check?

Theorem: Only job J_k with longer relative deadline D_k can block a job J_i with shorter relative deadline D_i

$$b_i = \max_{j=i+1}^{n} NPS_j \ /* D_i < D_k \text{ if } i < k */$$

$$\sum_{j=1}^{n} \frac{e_j}{p_j} + \frac{b_i}{p_i} \le 1; \text{ task by task check (only for task } i\text{)}$$

$$\sum_{j=1}^{n} \frac{e_j}{p_j} + \max_{j=1}^{n} \frac{b_j}{p_j} \le 1; \text{ single check (for all tasks)}$$

Earlier deadline and Non-preemptable code section

$$\sum_{j=1}^{n} \frac{e_j}{\min(D_j, p_j)} + \frac{b_i}{\min(D_i, p_i)} \le 1; \text{ task by task check (only for task } i\text{)}$$

$$\sum_{j=1}^{n} \frac{e_j}{\min(D_j, p_j)} + \max_{j=1}^{n} \frac{b_j}{\min(D_i, p_j)} \le 1; \text{ single check (for all tasks)}$$

Static Priority Scheduling vs. EDF

- Which one is more popular in the real-time market?
- You can’t beat 100% schedulability, can you? Strangely enough, static priority scheduling algorithm with schedulability significantly less than 100% has taken over the world of real-time computing. Why?

- What prevents EDF becoming popular?
  - 1) EDF has small marketing budget and it loses the marketing war. Sometimes inferior technology wins, isn’t it?
  - 2) There is a dark side of EDF.

The Stability Problem

There is a dark side (serious problem) with EDF.

- When a system becomes overloaded and not all the tasks can meet their deadlines, it is important to keep meeting deadlines of critical tasks. This is an easy problem for fixed priority scheduling. Unfortunately, there is no low complexity solution for this problem under EDF since each job’s priority is changing and it is expensive to keep track the execution states during runtime.

- Fortunately, in certain applications, the penalty of missing a deadline can be lessened by application level measures, for example, dropping a B frame in video is ok. Even in feedback control, there are new results that “soften” the deadline. So EDF will become more popular.