Independent Task Scheduling

Embedded OS Implementation

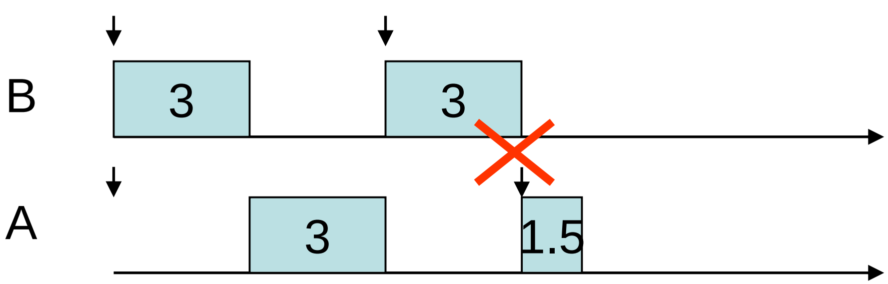
Prof. Ya-Shu Chen
National Taiwan University
of Science and Technology

Motivation

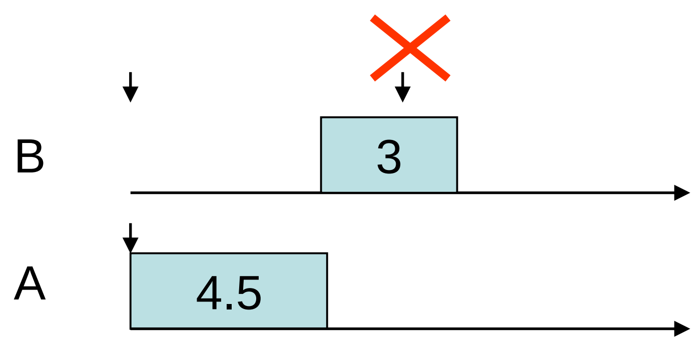
- Take yourself as an example
  – Naturally you have a number of things to do with time pressure
    - Project deadlines, meeting time, class time, and deadlines for bills
  – Some of them regularly recur but some don't
    - To eat meal on 12:30 everyday
    - Go to the movies on 8:00pm

Motivation

- You schedule yourself to meet deadlines
  - Course A: one homework is announced every 9 days, each costs you 4.5 days to do
  - Course B: one homework is announced every 6 days, each costs you 3 days to do
- You miss deadlines of one course if your policy favors either one course



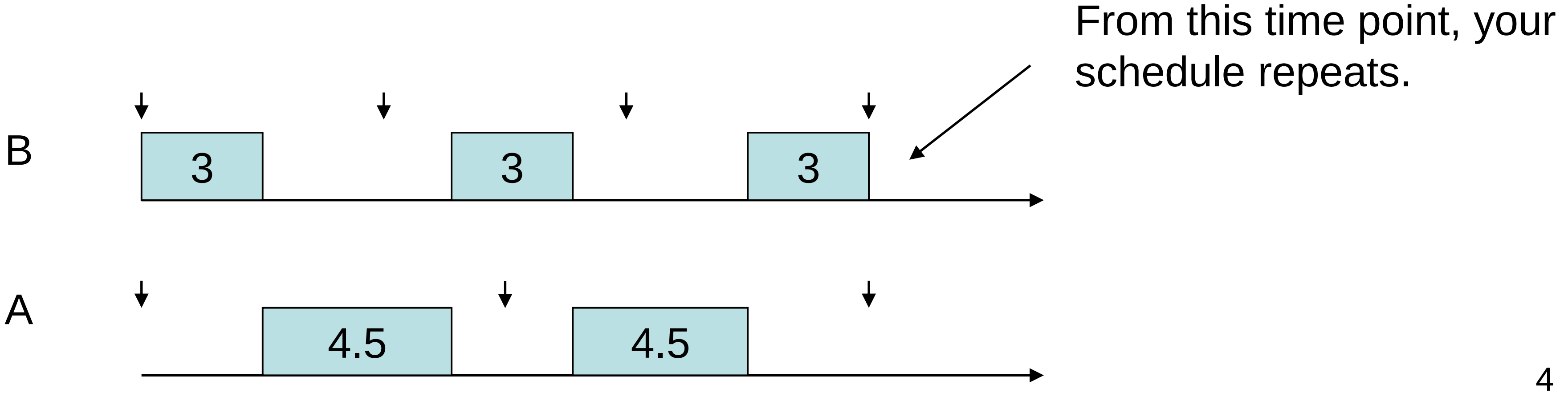
Favor course B



Favor course A

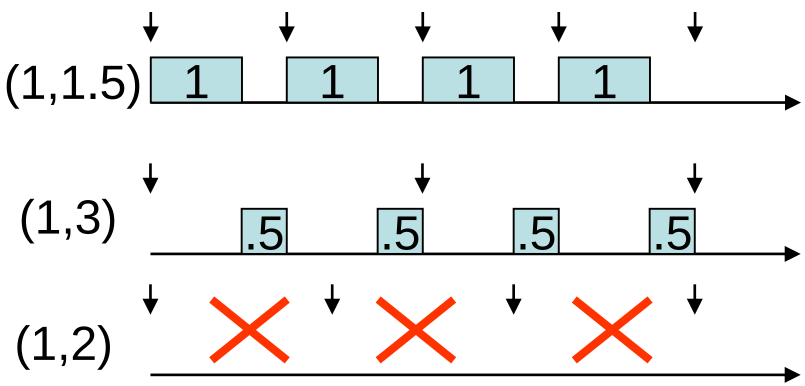
Motivation

- Schedule to meet deadlines (cont'd.)
  – Course A: (4.5, 9)
  – Course B: (3, 6)
- All deadlines are met if you whatever has the closest deadline

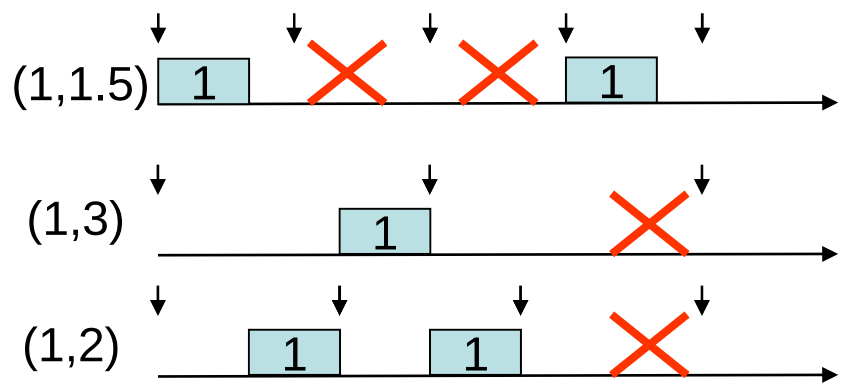


Motivation

- You schedule yourself to survive overloads
  - (1,2), (1,3), (1,1.5)



Favoring (1,1.5)→(1,3)→(1,2)
2 lecturers are happy, 1 will flunk you though…



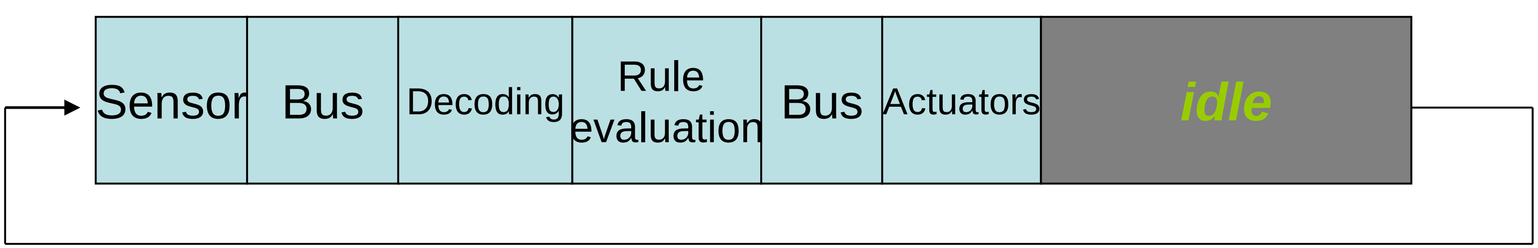
Do whatever has the closest deadline.
You are in deep shit!

Cyclic-Executive

Cyclic Executive

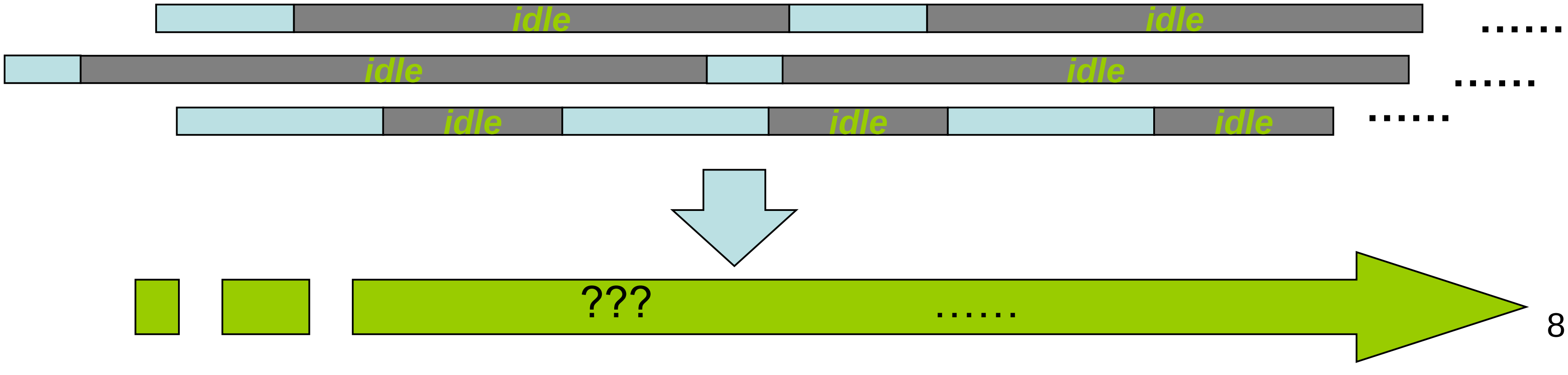
pros:方便degug
cons:難以修改

- The system repeatedly exercises a static schedule
  - A table-driven approach
- Many existing systems still take this approach
  - Easy to debug and easy to visualize
    - Highly deterministic
  - Hard to program, to modify, and to upgrade
    - A program should be divided into many pieces (like an FSM)



Cyclic Executive

- The table emulates an infinite loop of routines
  - However, a single independent loop is not enough to many complicated systems
  - Multiple concurrent loops should be considered
- How large should the table be when there are multiple loops?



Cyclic Executive

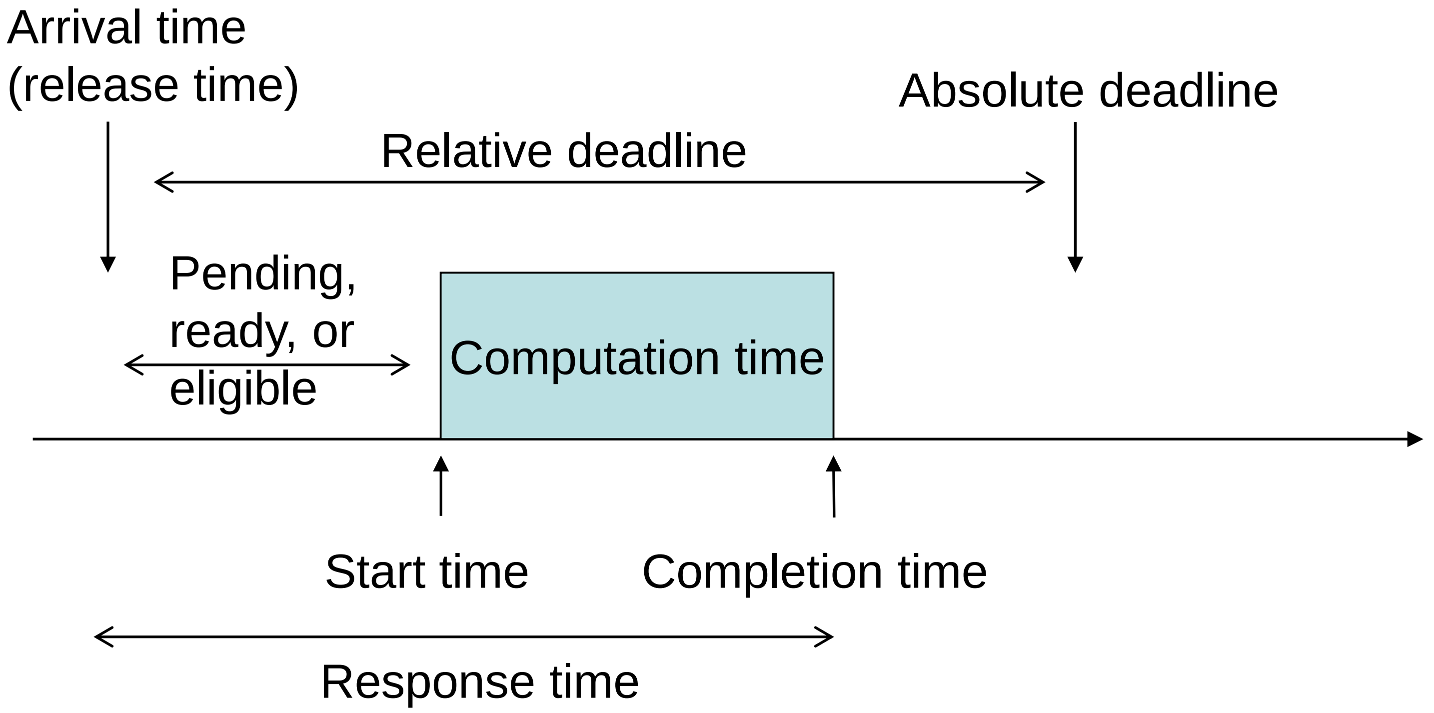
- ***Definition***: Let the hyper-period of a collection of loops be a time interval which's length is the least-common-multiplier of the loops' lengths
  - Let the length of the hyper-period be abbreviated as "h"

- ***Theorem***: The number of routines to be executed in any time interval [t,t+x] is identical to that in [t+h,t+h+x]

Priority-Driven Scheduling

System Model

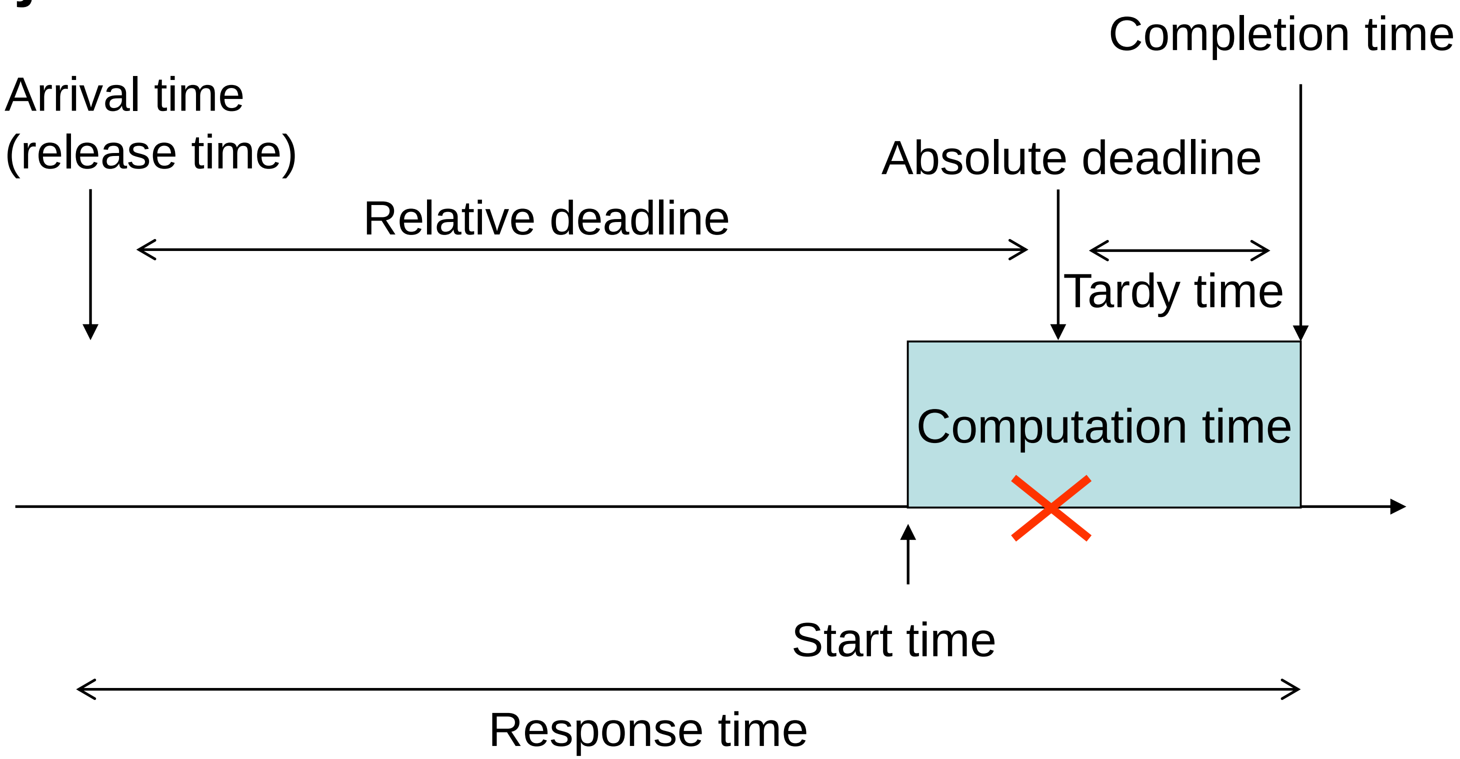
- A job with real-time constraints



The job completes before its deadline, that means the deadline is satisfied.

System Model

- A job with real-time constraints



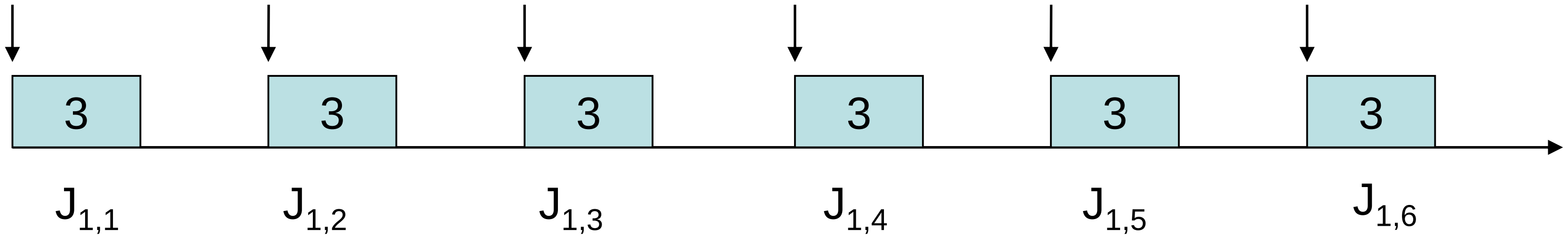
The job completes after its deadline, that means the deadline is violated or an overflow occurs.

System Model

- A task set is of a number of tasks
  - $\{T_1, T_2, \ldots, T_n\}$
  - Tasks share nothing but CPU, and tasks are independent to one another
- A task T_i is a template of jobs, where jobs refer to recurrences of task.
  - Every job executes the same piece of code
    - Of course, different input and run-time conditions cause jobs behaves differently
    - $J_{i,j}$ refers to the j-th job of task T_i
  - The computation time c_i of jobs is bounded and known a priori

System Model

- A purely periodic task
  - Jobs of a task T recur every fixed time interval p
  - A job must be completed before the next job arrives
    - Relative deadlines for jobs are, implicitly, the period
  - T is defined as (c,p)



Periodic task T_1=(3,6)

System Model

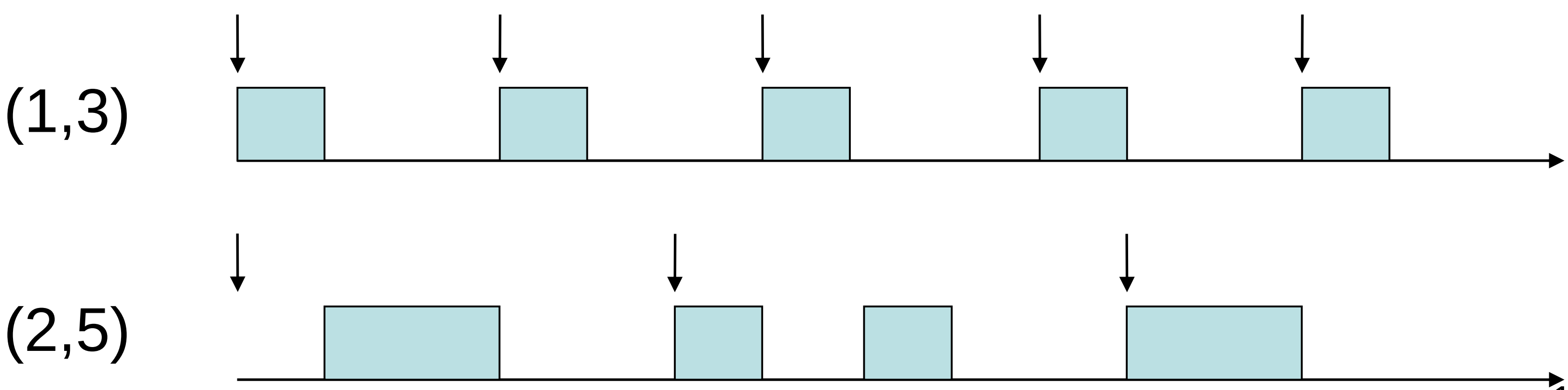
- Priority
  - Reflect the urgency of jobs
  - Any job inherits its task's priority
- Preemptivity
  - As a high-priority task arrives, it preempts the execution of any low-priority tasks

System Model

- Checklist
  - Periodic tasks
  - Real-time constraints
  - Priority
  - Preemptivity

Rate-Monotonic Scheduling

- Task-level fixed-priority scheduling
  - All jobs inherit its task's priority
  - Usually abbreviated as fixed-priority scheduling
- Tasks' priorities are inversely proportional to their period lengths

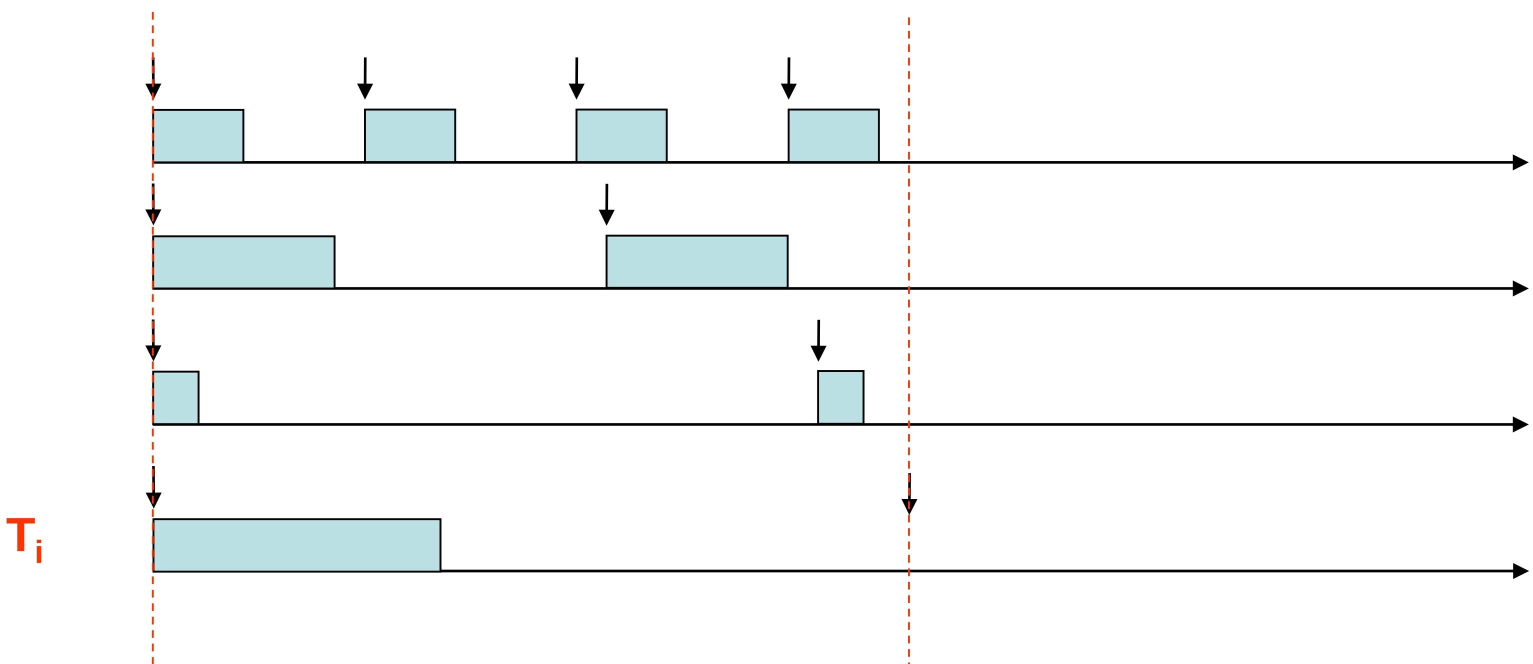


Rate-Monotonic Scheduling

- Critical instant (critical instance) of task T_i
  - A job $J_{i,c}$ of task T_i released at T_i's critical instant would have the longest response time
  - $J_{i,c}$ would be the one that is “hardest” to meet its deadline
  - If $J_{i,c}$ succeeds in satisfying its deadline, then any job of T_i always succeeds for any cases
    - Since in any other cases deadlines are easier to meet

Rate-Monotonic Scheduling

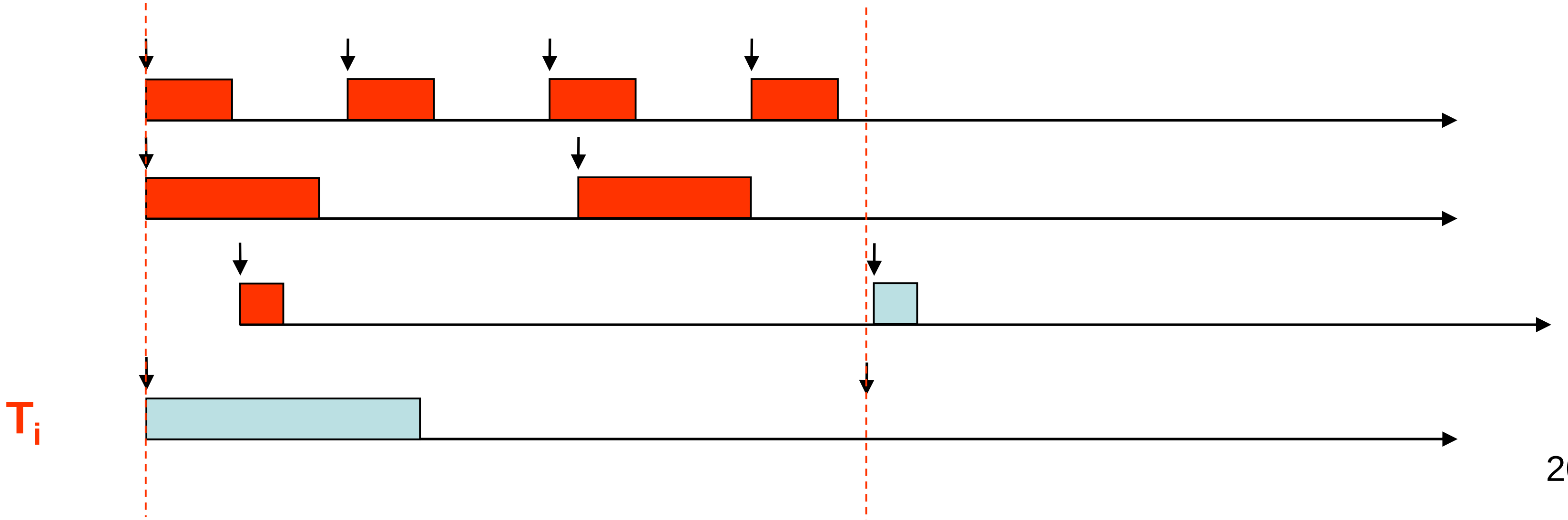
- ***Theorem***: A critical instant of any task T_i occurs when one of its job $J_{i,c}$ is released at the same time with a job of every higher-priority task (i.e., in-phase).



Rate-Monotonic Scheduling

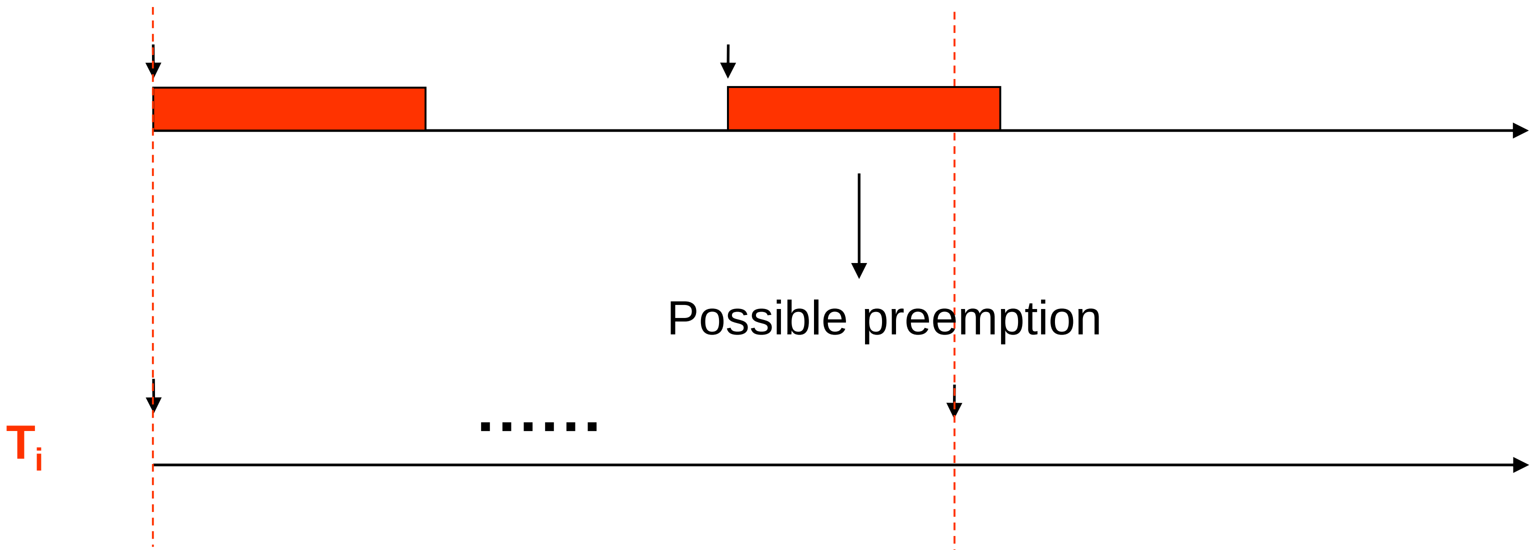
- Proof: "interferences" from high-priority tasks is the maximum within the first period of T_i

$$\sum_{j<i} c_j \left\lceil \frac{p_i}{p_j} \right\rceil$$



Rate-Monotonic Scheduling

- Critical instant: Why ceiling function?

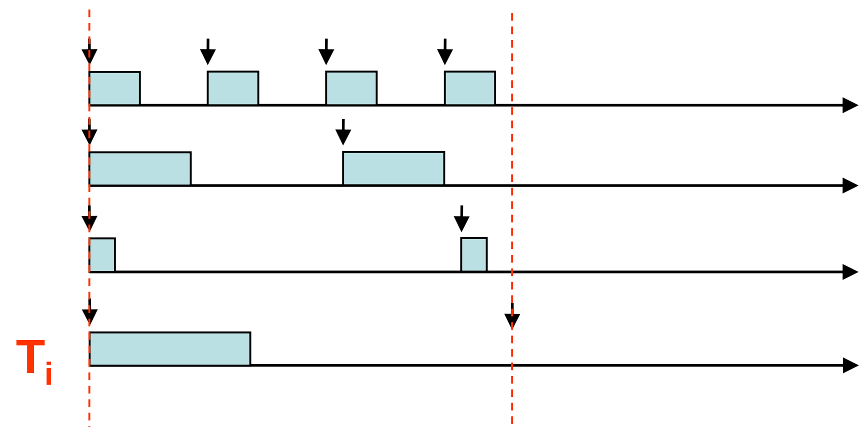


Rate-Monotonic Scheduling

- Response time analysis
  - The response time of the job of Ti at critical instant can be calculated by the following recursive function

$$r_0 = \sum_{\forall i} c_i$$

$$r_n = \sum_{\forall i} c_i \left\lceil \frac{r_{n-1}}{p_i} \right\rceil$$



  - Observation: the sequence of r_x, x>=0 may or may not converge

Rate-Monotonic Scheduling

- ***Theorem***: Given a task set=$\{T_1,T_2,\ldots,T_n\}$, if at critical instant the response time of the first job of task T_i, for each i, converges no later than p_i, then jobs never miss their deadlines

- Observations
  – If the task set survives critical instant, then it will survive any task phasing
  – The analysis is an exact schedulability test for RMS
  – Usually referred to as "Rate-Monotonic Analysis", RMA for short

Rate-Monotonic Scheduling

- Example: T1=(2,5), T2=(2,7), T3=(3,8)
  - T1:
    - R_0=2 ≤5 ok
  - T2:
    - R_0=2+2=4 ≤7
    - R_1=2* $\lceil 4/5 \rceil$+2* $\lceil 4/7 \rceil$=4 ≤7 ok
  - T3:
    - R_0=2+2+3=7 ≤8
    - R_1= 2* $\lceil 7/5 \rceil$+2* $\lceil 7/7 \rceil$+ 3* $\lceil 7/8 \rceil$ =9> 8 failed
  - Note: each task succeeds→the task set

Rate-Monotonic Scheduling

- Proof:
  - If the response time converges at r_n, then the first lowest-priority job completes at r_n
  - If r_n is before p_n , then the first lowest-priority job meets its deadline if critical instant occurs
  - Since the job survives critical instant, it always succeed satisfying its deadline under any task phasing

Rate-Monotonic Scheduling

- Test every Ti for schedulability!!

  - {T1=(3,6),T2=(3.1,9),T3=(1,18)}
  - Response analysis of T3:
    - R0=7.1, R1=10.1, R2=13.2, R3=16.2, R4=16.2<18
  - {T1,T2,T3} is schedulable!?

  - However, {T1, T2} is not schedulable!!!

Rate-Monotonic Scheduling

- Computational complexity
  - $O(n^2 * p_n)$, pseudo-polynomial time
  - Would be *extremely slow* when periods of tasks are small and prime to one another
  - Would be very fast when periods are harmonically related

Rate-Monotonic Scheduling

- Phenomena
  - Even though RMA is an exact test for fixed-priority scheduling, it is not often used, especially for those dynamic systems
  - RMA is more suitable for static systems

  - Are there any schedulability tests being efficient enough for on-line implementation?
    - No slower than polynomial time

Rate-Monotonic Scheduling

- A trivial schedulability test
  - The system accepts a task set T if the following conditions are both true
    - There are no other tasks in the system
    - $c1/p1 \leq 1$
  - The algorithm is efficient enough (i.e., O(1))
  - Too conservative!! Very Poor CPU utilization!!

Rate-Monotonic Scheduling

- Definition
  – Utilization factor of task T=(c,p) is defined as

$$\frac{c}{p}$$

  – CPU utilization of a task set $\{T_1,T_2,\ldots,T_n\}$ is

$$U=\sum_{i=1}^{n}\frac{c_i}{p_i}$$

  – Observation: if the total utilization exceeds 1 then the task set is not schedulable

Rate-Monotonic Scheduling

- ***Theorem***: [LL73] Given a task set $\{T_1, T_2, \ldots, T_n\}$. It is schedulable by RMS if

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \leq U(n) = n(2^{1/n} - 1)$$

- Observation:
  - If the test succeeds then the task is schedulable
  - A sufficient condition for schedulability

Rate-Monotonic Scheduling

- Proof: Let us consider two tasks only

$C_1 \le P_2 - P_1(\lfloor P_2/P_1 \rfloor)$

The largest possible C2 is

$P_2 - C_1(\lceil P_2/P_1 \rceil)$

Total utilization factor is

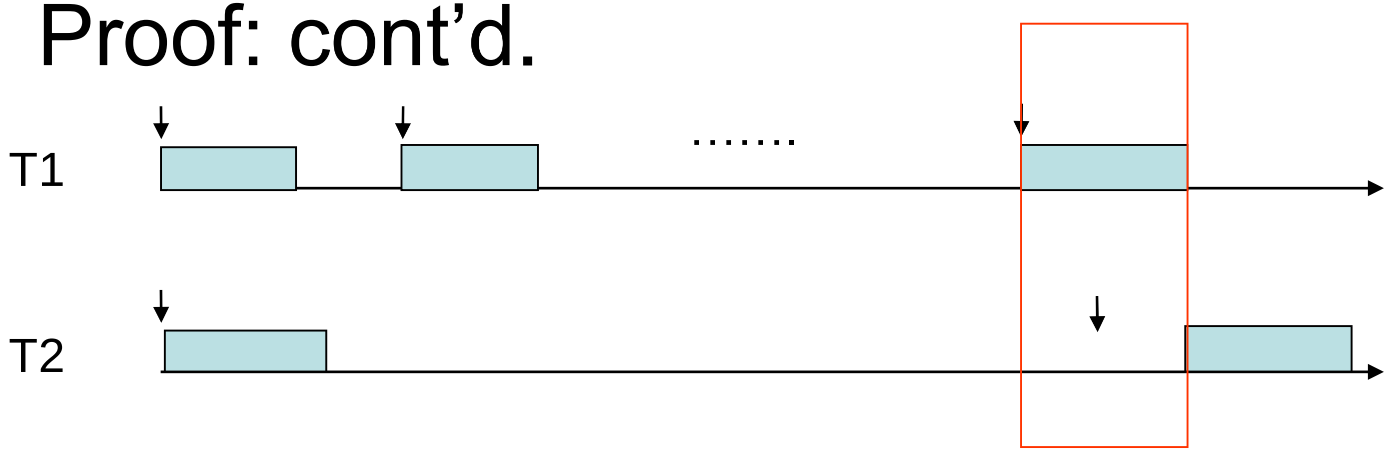
$U = 1 + C_1(1/P_1 - (1/P_2)(\lceil P_2/P_1 \rceil))$

T2's 2nd job does not overlap the immediately preceding job of T1

- U monotonically decreases with C_1
  - Because U never greater than 1, so the rightmost term in the last equation is always negative

Rate-Monotonic Scheduling

- Proof: cont'd.



$$C_1 \geq P_2 - P_1(\lfloor P_2/P_1 \rfloor)$$

The largest possible C_2 is

$$-C_1(\lfloor P_2/P_1 \rfloor) + P_1(\lfloor P_2/P_1 \rfloor)$$

Total utilization factor is

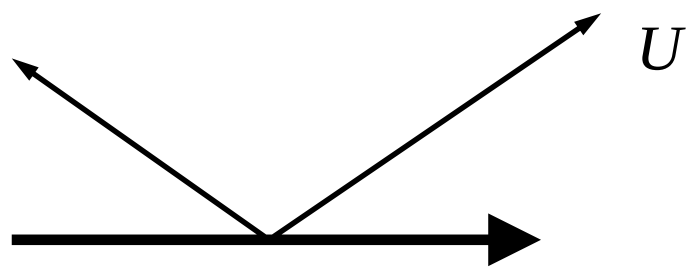
$$U = (P_1/P_2)\lfloor P_2/P_1 \rfloor + C_1((1/P_1) - (1/P_2)(\lfloor P_2/P_1 \rfloor))$$

T2's 2nd job overlaps the immediately preceding job of T1

- U monotonically increases with C_1

Rate-Monotonic Scheduling

- Proof: Cont'd.
  - It can be found that the minimal U occurs at $C_1 = P_2 - P_1(\lfloor P_2 / P_1 \rfloor)$



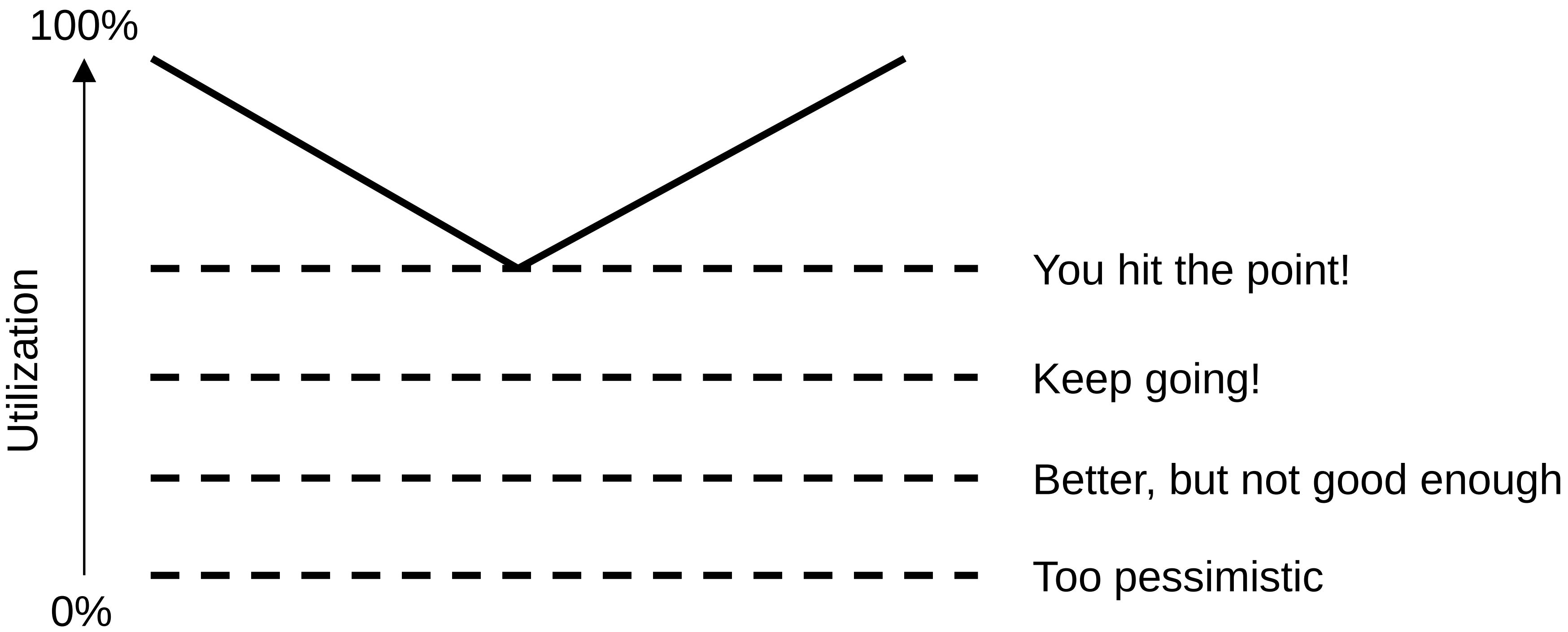
C1 = P2 – P1 * floor(P2/P1)

  - By some differentiation, the minimal achievable utilization is

$$U(2) = 2(2^{1/2} - 1)$$

Rate-Monotonic Scheduling

- To find "the smallest" among "the largest achievable processor utilizations that can be achieved by different task sets"



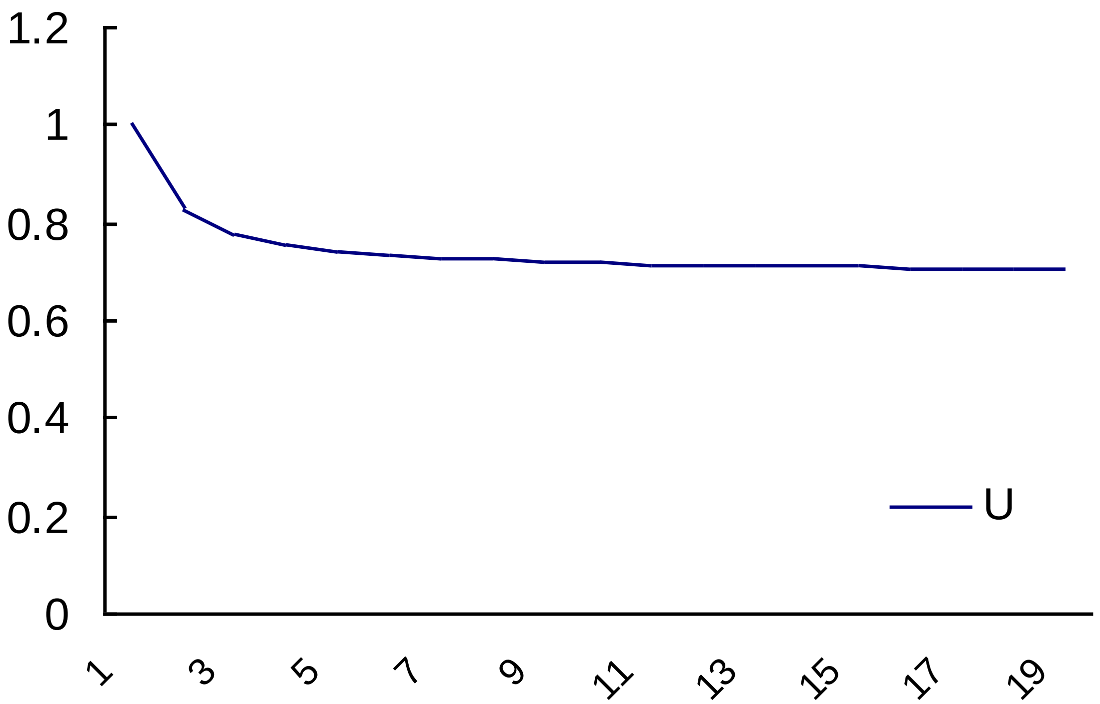
Rate-Monotonic Scheduling

- Simon says: To generalize the proof to n tasks is easy :)

- If a task set of n tasks has a total utilization being no greater than U(n), then it is guaranteed to be schedulable by RMS
  - Because the most hard-to-schedule task set having the same total utilization is schedulable
  - The test's time complexity is O(n), which is very efficient for on-line implementation

Rate-Monotonic Scheduling

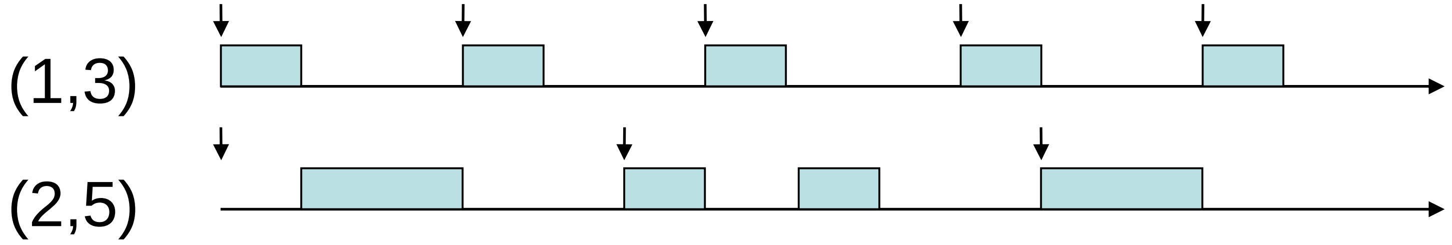
- When x → infinitely large, U(x)→0.68



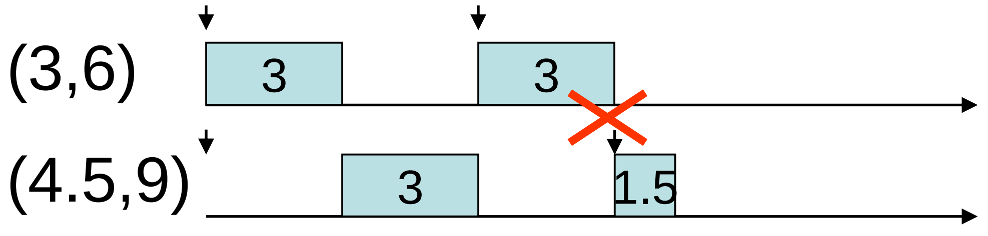
1	1
2	0.828427
3	0.779763
4	0.756828
5	0.743492
6	0.734772
7	0.728627
8	0.724062
9	0.720538
10	0.717735
11	0.715452
12	0.713557
13	0.711959
14	0.710593
15	0.709412

Rate-Monotonic Scheduling

- Example 1: (1,3), (2,5)
  - Utilization =0.73 ≦ U(2)=0.828

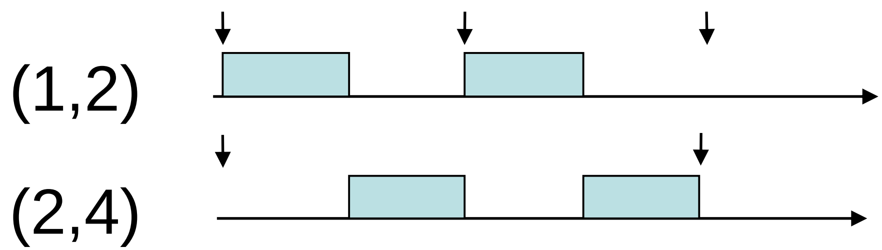


- Example 2: (4.5,9), (3,6)
  - Utilization =100%>U(2)=0.828



Rate-Monotonic Scheduling

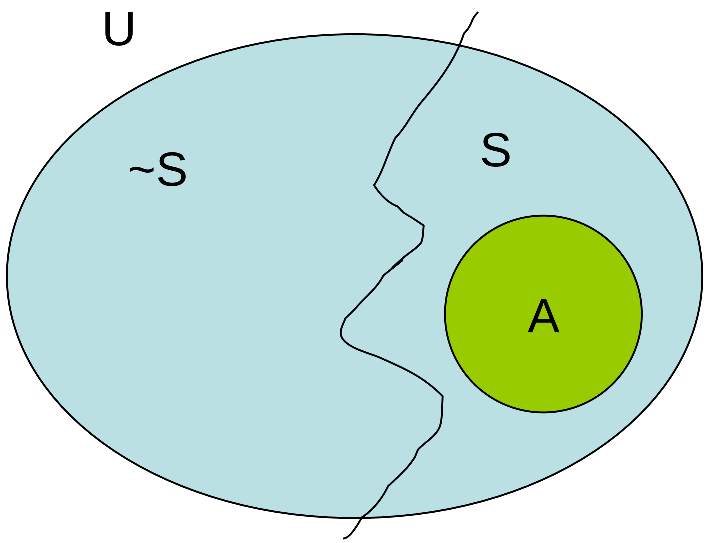
- Example 3: (1,2), (2,4)
  - Utilization =100%>U(2)=0.828



- Example 2 and 3 shows that, we know nothing about those task sets of total utilization > the utilization bound!
- But we do know those ≦ the utilization bound is schedulable!

Rate-Monotonic Scheduling

- Sufficiency but no necessity
  - Utilization test provides a fast way to check if a task set is schedulable
  - Any task set fails utilization test, does not implies that it is not schedulable



U: universe of task sets
~S: task sets unschedulable by RMS
  - Example 2

S: task sets schedulable by RMS
  - Example 1 and Example 3

A: Those can be found by utilization test
  - Example 1

Example 3 is in S-A

Rate-Monotonic Scheduling

- Summary
  - Explicit prioritization over tasks
  - To decide task sets' schedulability is costly
  - Sufficient tests were developed for fast admission control

Priority-Driven Scheduling: Dynamic-Priority Scheduling

Earliest-Deadline-First Scheduling

- Definition
  - Feasible
    - A set of tasks is said to be feasible if there is some way to schedule the tasks without any deadline violations
  - Schedulable
    - Given a scheduling algorithm A
    - A set of tasks is said to be schedulable if algorithm A successfully schedule the tasks without any deadline violations
- Observations
  - A feasible task set may not be schedulable by RMS
  - If a task set is schedulable by some algorithm A, then it is feasible

Earliest-Deadline-First Scheduling

- EDF scheduling algorithm always pick a pending job which has the earliest deadline for execution
  - A job having an earlier deadline is assigned to a higher “priority”
  - Priority in EDF is not a task-wide notion
    - Jobs of a task may have different “priorities”
    - But due to the relative deadline of a job never changes, EDF can be classified as a job-level fixed-priority scheduling
    - You’d better to avoid using the term “priority” for EDF since there is no explicit definition

Earliest-Deadline-First Scheduling

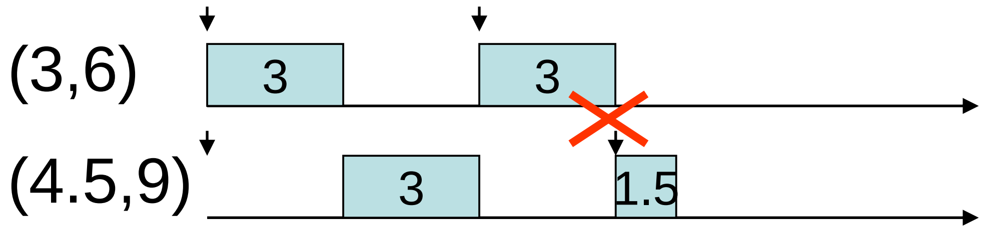
- If an algorithm schedulable ←→ feasible
  - It can be referred to as a universal scheduling algorithm

- What is the universal scheduling algorithm for periodic and preemptive uniprocessor systems?
  - EDF!

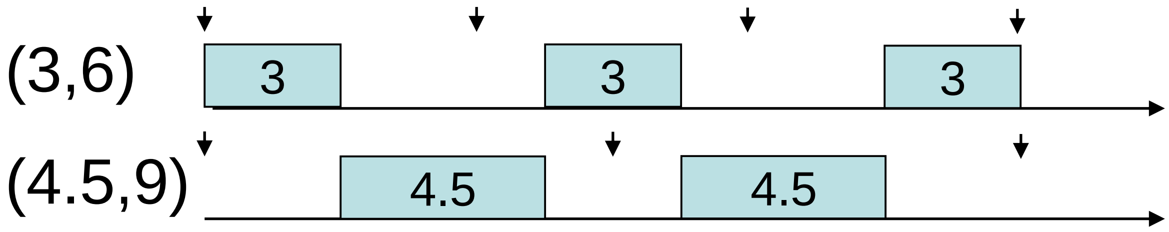
Earliest-Deadline-First Scheduling

- Example

Not scheduable by RMS



Schedulable by EDF

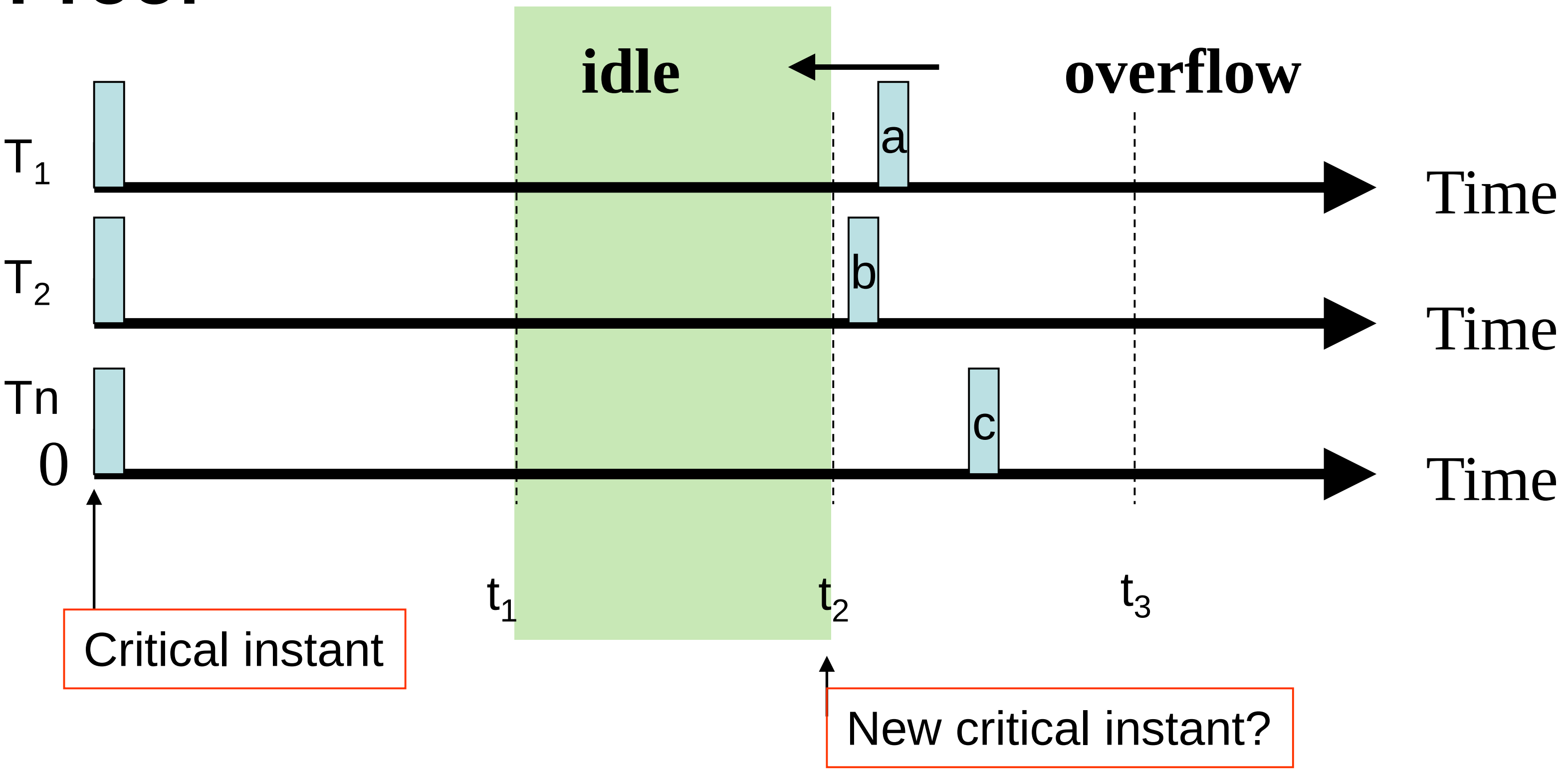


Earliest-Deadline-First Scheduling

- ***Theorem***: With EDF, there is no idle time before an overflow

- Observation: A very strong statement that implies optimality of EDF in terms of schedulability

Proof

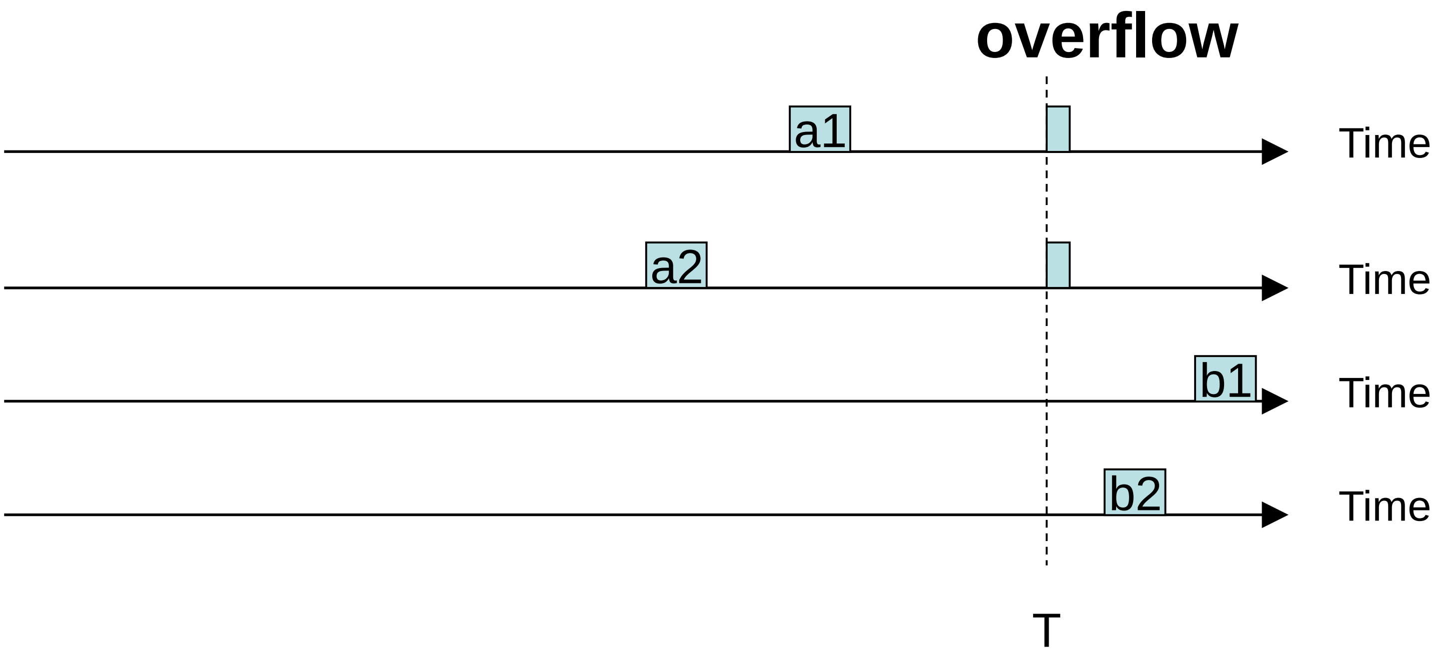


- Suppose that there is an overflow at time t_3, and the processor idles between t_1 and t_2
- If we move "a" forward to be aligned with t_2, the overflow would occur earlier than it was (i.e., at or before t_3)
  - That is because EDF's discipline: moving forward means promoting the urgency of T_1's jobs
- By repeating the above action, jobs a,b, and c can be aligned at t_2
  - →that contradicts the assumption! From t_2 on, there is no idle until the overflow

Earliest-Deadline-First Scheduling

- ***Theorem***: A set of tasks is schedulable by EDF if and only if its total CPU utilization is no more than 1

- Observation: → is easy, ← requires some reasoning similar to the proof of the last theorem



→: suppose that U<=1 but the system is not schedulable by EDF

- Suppose that there is an overflow at time T
  - Jobs a's have deadlines at time T
  - Job b's have deadlines after time T

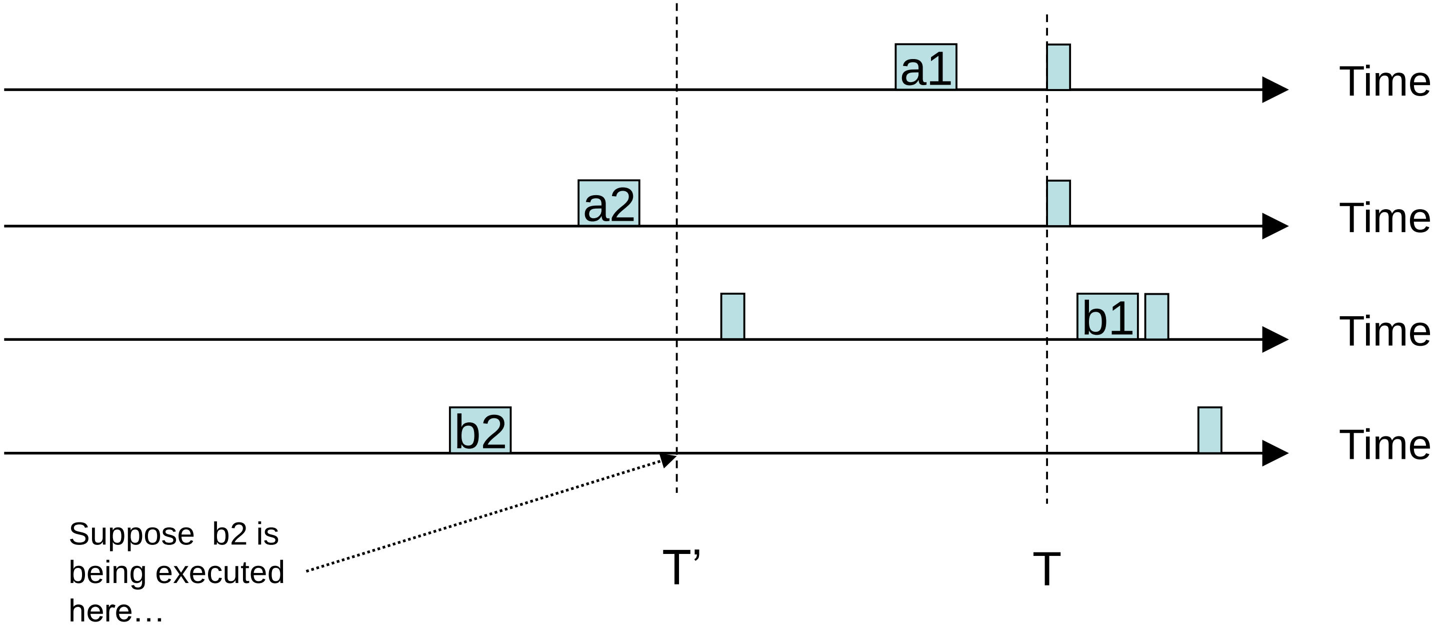
- Case A: non of job b's is executed before T
  - The total computational demand between [0,T] is

    $$C_1(\lfloor T/P_1 \rfloor) + C_2(\lfloor T/P_2 \rfloor) + \ldots + C_n(\lfloor T/P_n \rfloor)$$
  - Since there is no idle before an overflow

    $$C_1(\lfloor T/P_1 \rfloor) + C_2(\lfloor T/P_2 \rfloor) + \ldots + C_n(\lfloor T/P_n \rfloor) > T$$
  - That implies U>1
  - →←

Boxes: “arrival” times!!

overflow



Case B: some of job b’s are executed before T

- Because an overflow occurs at T, the violated jobs must be a’s
  - Right before T, there must be some job a’s being executed
  - Let in [T’,T] there is no job b’s being executed
- Just before T’, some of b’s is being executed! (the definition of T’)
  - It means that all jobs have deadlines <= T and arrive before T’ have completed before T’
- Back to [T’,T], the total computation demand is no less than

  $$C_1(\lfloor T - T'/P_1 \rfloor) + C_2(\lfloor T - T'/P_2 \rfloor) + \ldots + C_n(\lfloor T - T'/P_n \rfloor)$$

  - Because there is deadline violations, so

  $$C_1(\lfloor T - T'/P_1 \rfloor) + C_2(\lfloor T - T'/P_2 \rfloor) + \ldots + C_n(\lfloor T - T'/P_n \rfloor) \geq T - T'$$

  - $\rightarrow\leftarrow$

Earliest-Deadline-First Scheduling

- Summary
  - A universal scheduling algorithm for real-time periodic tasks
  - Urgency of tasks is dynamic
    - But static for jobs
  - Job-level fixed-priority scheduling

Comparison

	RMS	EDF
Optimality	Optimal for fixed-priority scheduling	Universal
Schedulability test	Exact test is slow (PP), conservative tests are adopted	O(n) for exact test
Algorithm time complexity	O(1) job insertion is possible	Both job insertion and dispatch take O(log n) time
Overload survivability	High and predictable	Low and unmanageable**
Responsiveness	High priority tasks always have shorter response time	Non-intuitive to reach conclusions
Ease of implementation	Pretty simple	Relatively complicated
Run-time overheads (like preemption)	Low	High

Is it true?