Independent Task Scheduling

Real-Time and Embedded Operating Systems

Prof. Li-Pin Chang
ESSLab@NYCU

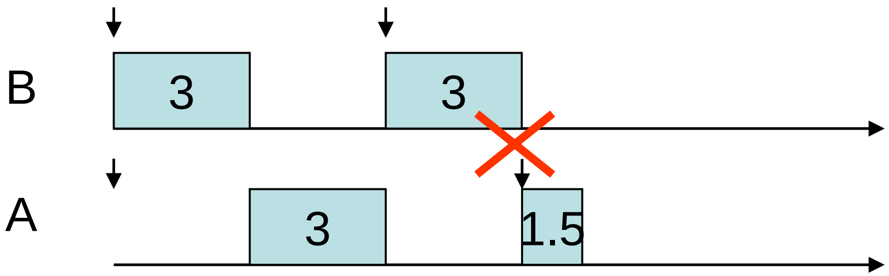
Motivation

- Scheduling on shared resources: CPU, I/O, etc
- Take yourself as an example
  - You have a bunch of things to do, with time pressure
    - Project deadlines, meeting times, class times, and deadlines of bills
  - Some of them regularly recur but some don't
    - Going for lunch at 12:30 everyday
    - Seeing a movie at 8:00pm

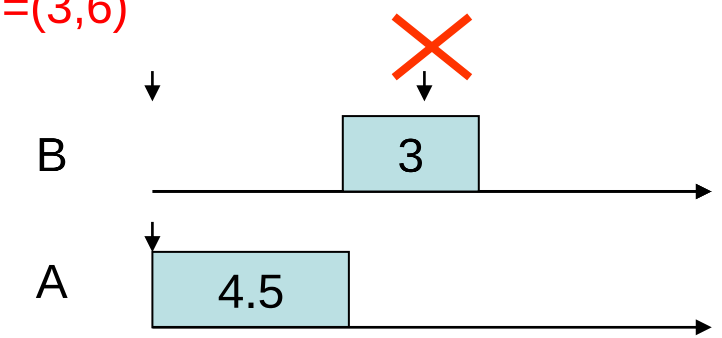
Motivation

- You schedule yourself to meet deadlines
  - Course A: a homework is announced every 9 days, and each costs you 4.5 days
  - Course B: a homework is announced every 6 days, and each costs you 3 days
- You miss deadlines of one course, if you favor either one of the two courses

*A=(4.5,9), B=(3,6)



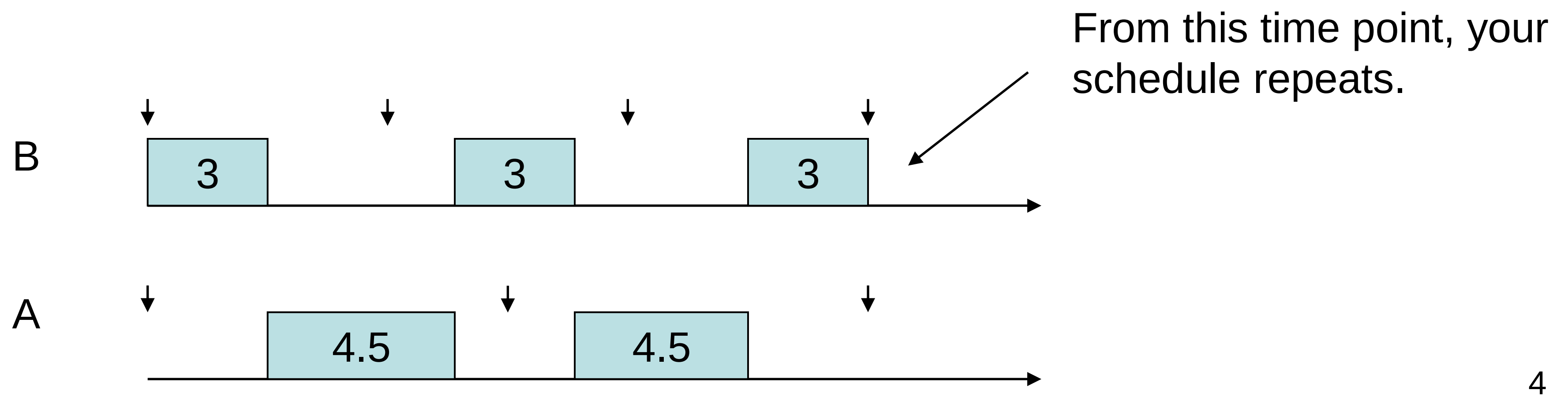
Favor course B



Favor course A

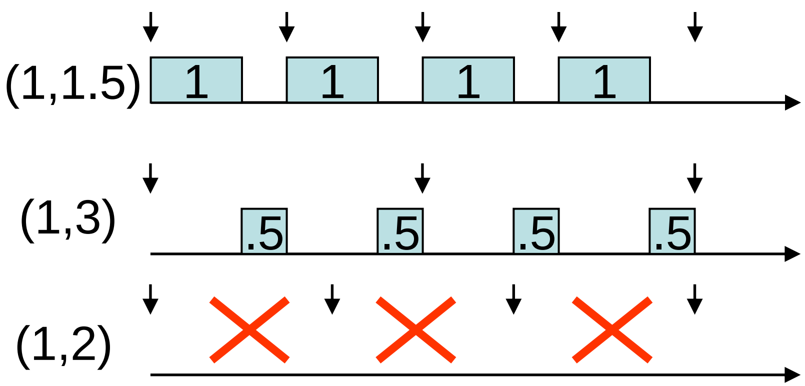
Motivation

- Scheduling to meet deadlines (cont'd.)
  - Course A: (4.5, 9)
  - Course B: (3, 6)
- All deadlines are met if you do the homework whose deadline is the nearest

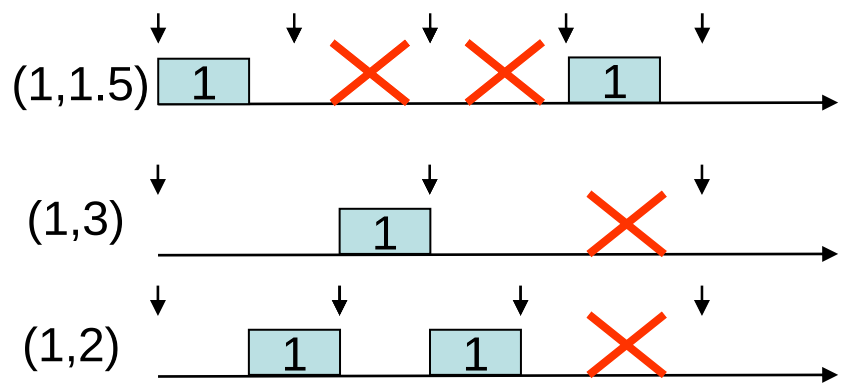


Motivation

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



Favoring (1,1.5)→(1,3)→(1,2)
You fail one course



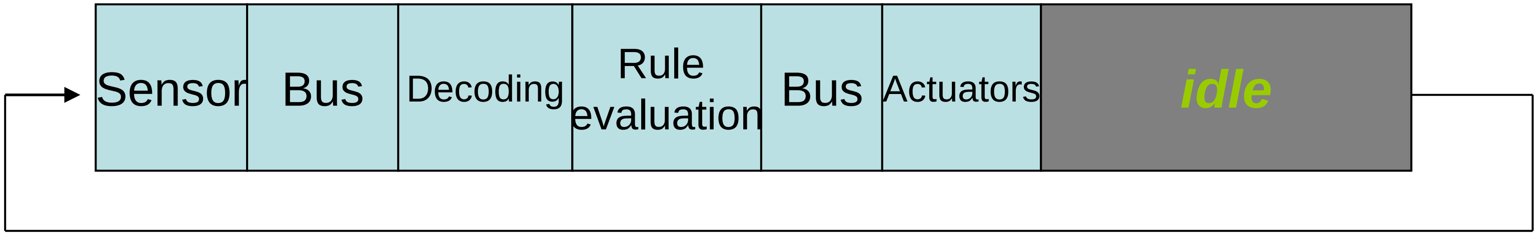
Do whatever has the closest deadline.
You may fail all the courses…

* Tie breaking is arbitrary

Cyclic-Executive

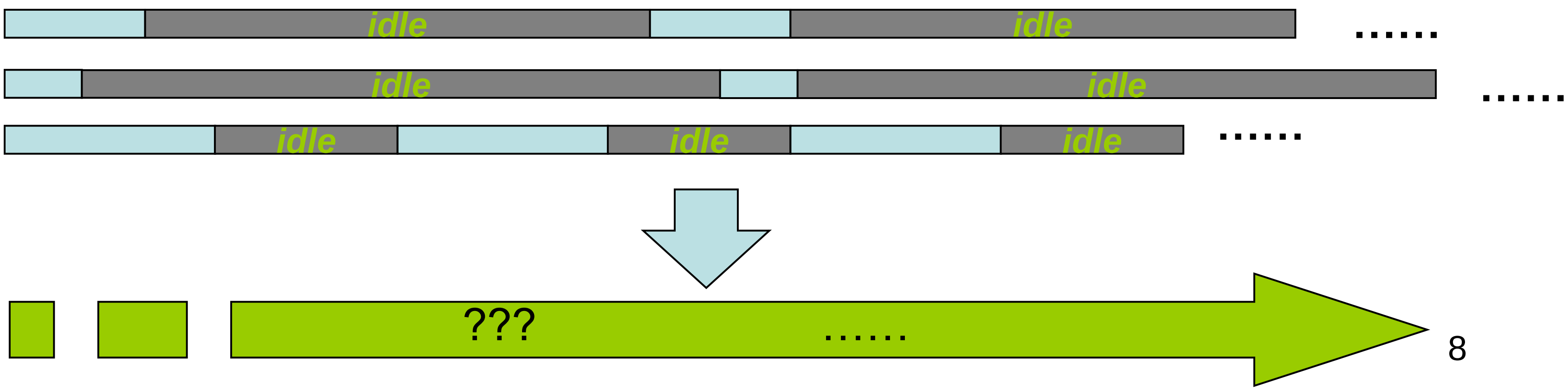
Cyclic Executive

- The system repeatedly executes 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, modify, and upgrade
    - A program should be divided into many pieces (like an FSM)
    - The table needs a major revision for every little change



Cyclic Executive

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

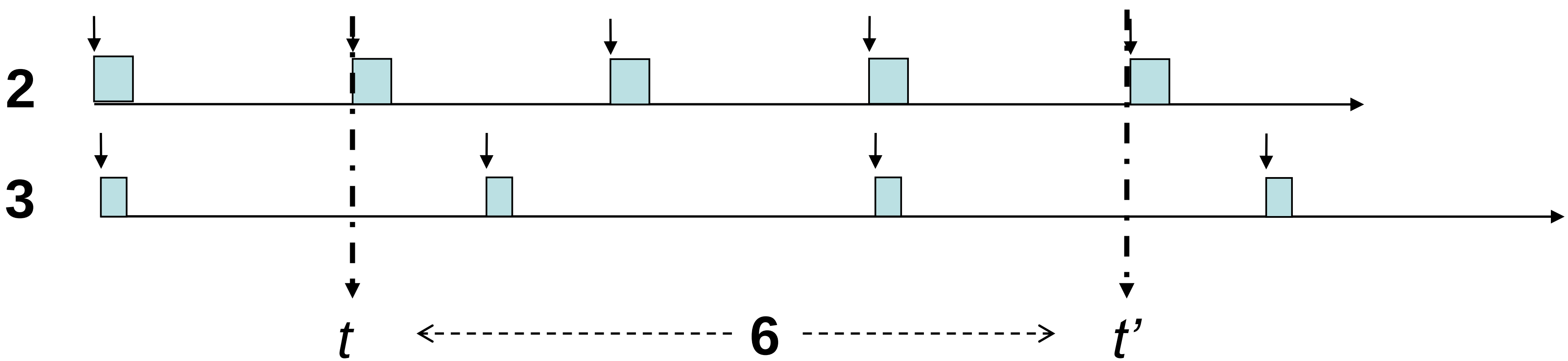


Cyclic Executive

- ***Definition***: The hyper-period (*h*) of a collection of loops is a time window whose length is the least-common-multiplier of all the loops
- ***Theorem***: The job sequence in [t,t+x] is identical to that in [t+h,t+h+x] (generic: x=h)

Cyclic Executive

- (Informal) Proof:

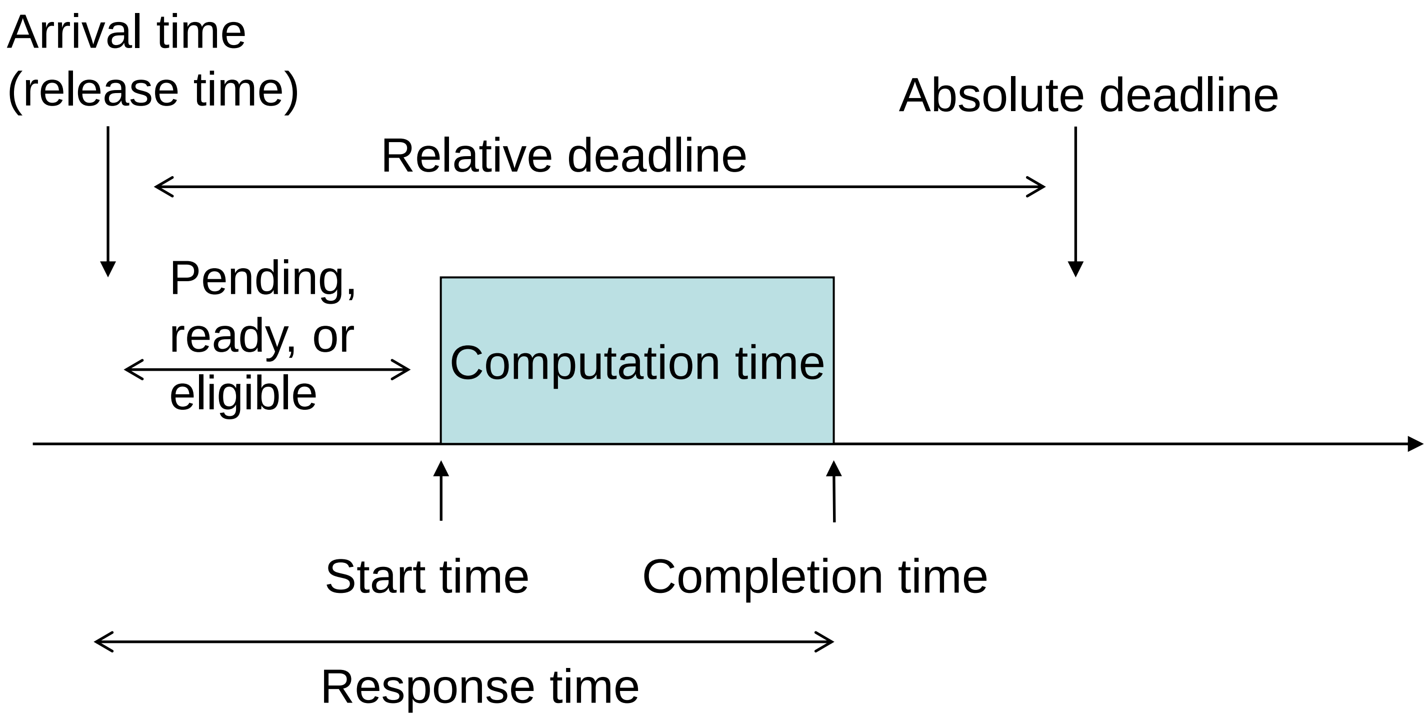


- The arrivals of loops (routines) since *t’* and those since *t* are exactly the same

Rate-Monotonic Scheduling (Fixed-Priority Scheduling)

System Model

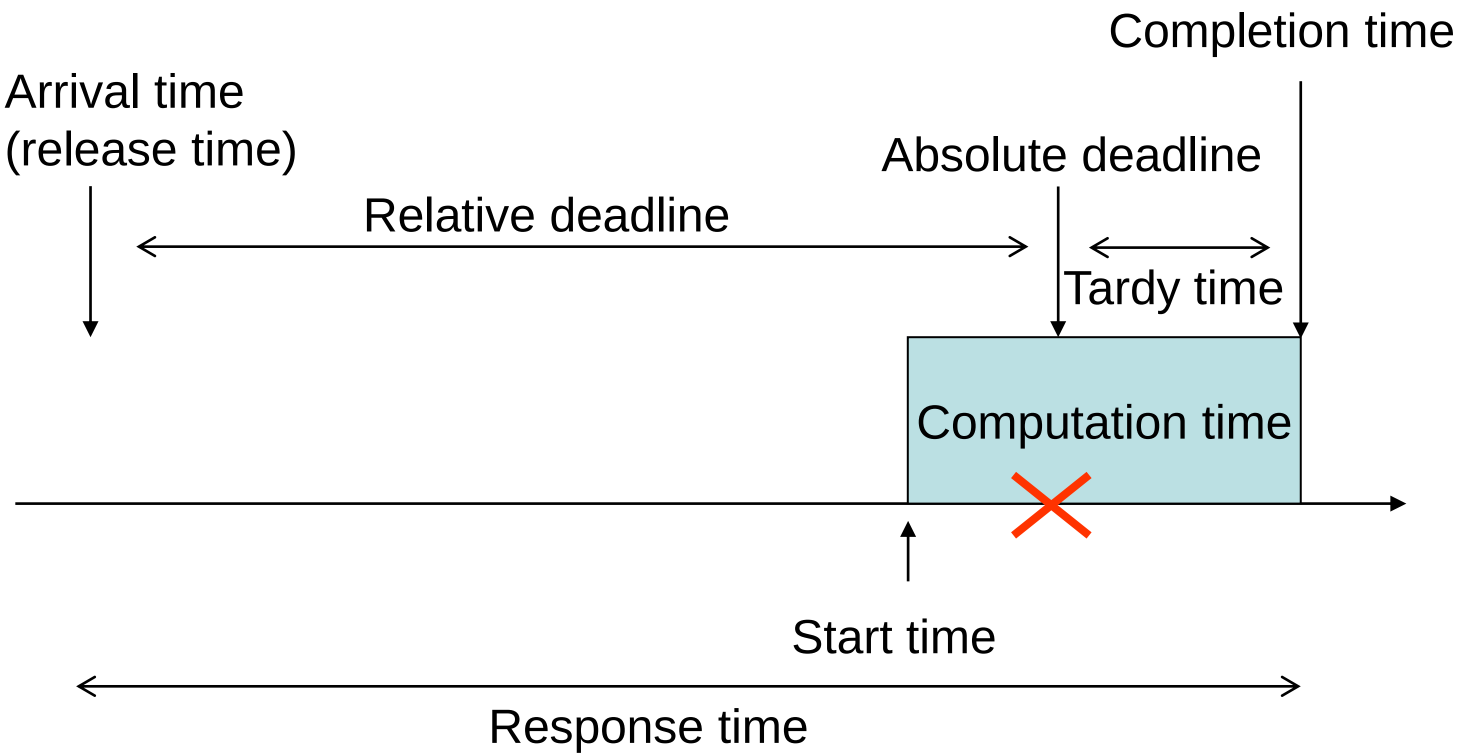
- A job with a deadline (no preemption)



Response time <= deadline → deadline is satisfied

System Model

- A job that misses its deadline



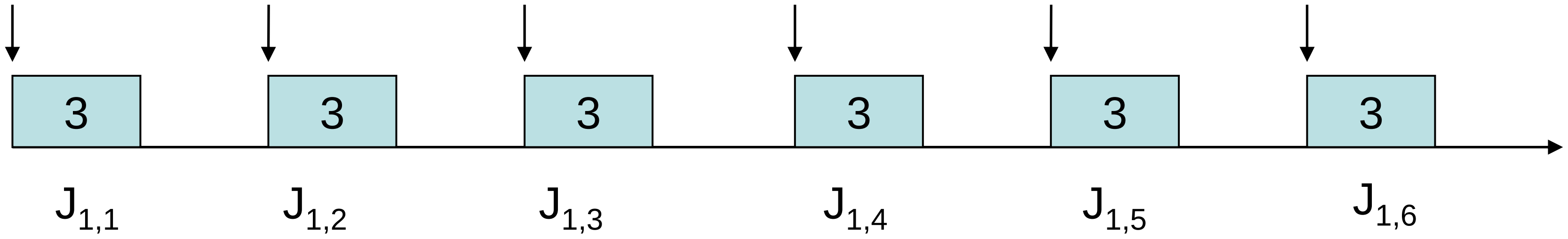
Response time > deadline → deadline missed or violated

System Model

- A task set consists of multiple tasks
  - $\{T_1, T_2, \ldots, T_n\}$
  - Tasks are independent and share nothing but the CPU
- A task T_i is a template of recurring jobs
  - Every job executes the same piece of code
    - $J_{i,j}$ refers to the j-th job of task T_i
  - The longest (worst-case) computation time c_i of these is known in advance

System Model

- A purely periodic task
  - Jobs of a task T recur every p units of time
  - 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

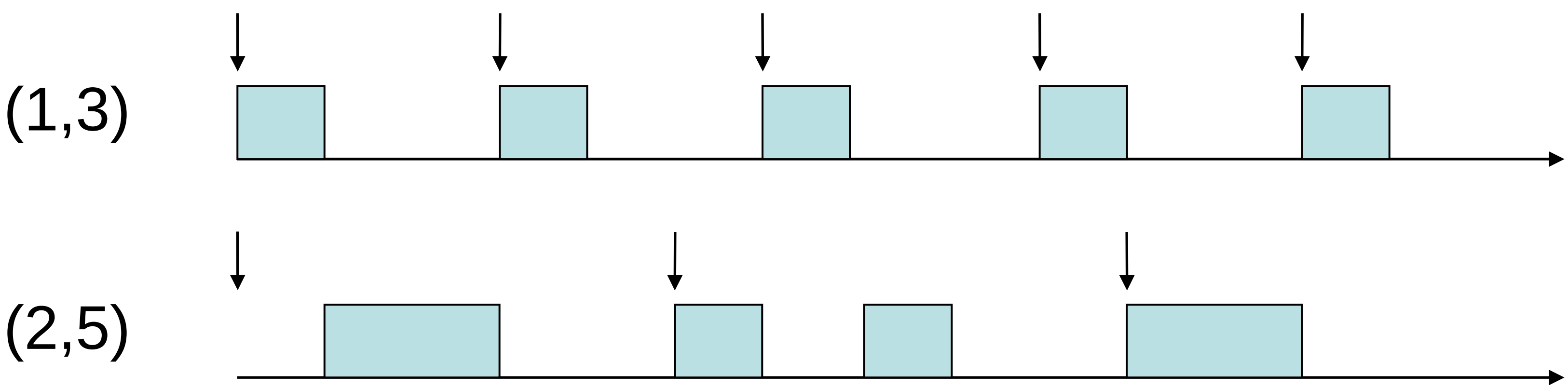
- Priority
  - Reflect the importance of jobs
  - Jobs of the same task may have the same or different priorities
- Preemption
  - When a high-priority task becomes ready, it preempts the (lower-priority) running task

System Model

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

Rate-Monotonic Scheduling

- A fixed-priority scheduling algorithm
  - All jobs of a task have the same priority
- Tasks priorities are proportional to task rates
  - Shorter periods → higher priorities

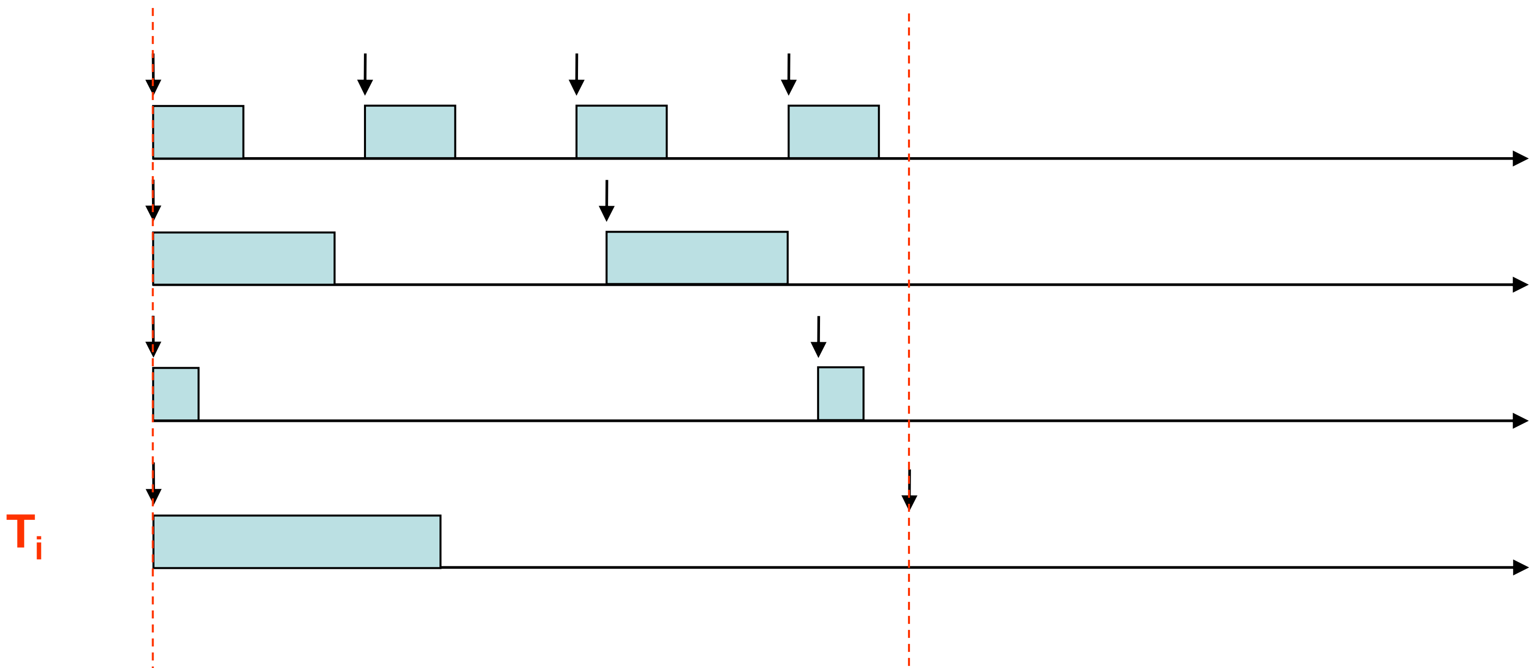


Rate-Monotonic Scheduling

- Does task T_i always meet its deadlines?
- Define: Critical instance of task T_i
  - A job $J_{i,c}$ of task T_i released at T_i's critical instance will have the longest response time
  - In this case, to meet the deadline of $J_{i,c}$ would be "the hardest"
  - If $J_{i,c}$ satisfyies its deadline for the critical instance, then any other jobs of T_i will succeed

Rate-Monotonic-Scheduling

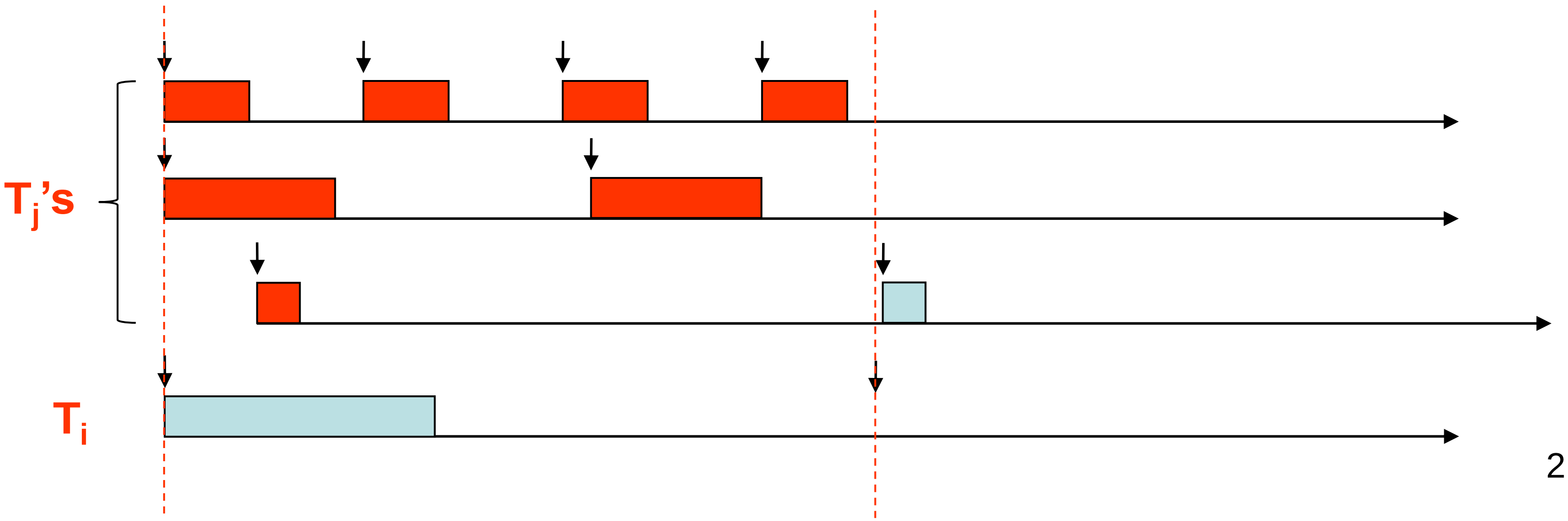
- ***Theorem***: A critical instance of a task T_i happens when the task and all the higher-priority tasks release a job at the same time (i.e., all tasks are in-phase)



Rate-Monotonic-Scheduling

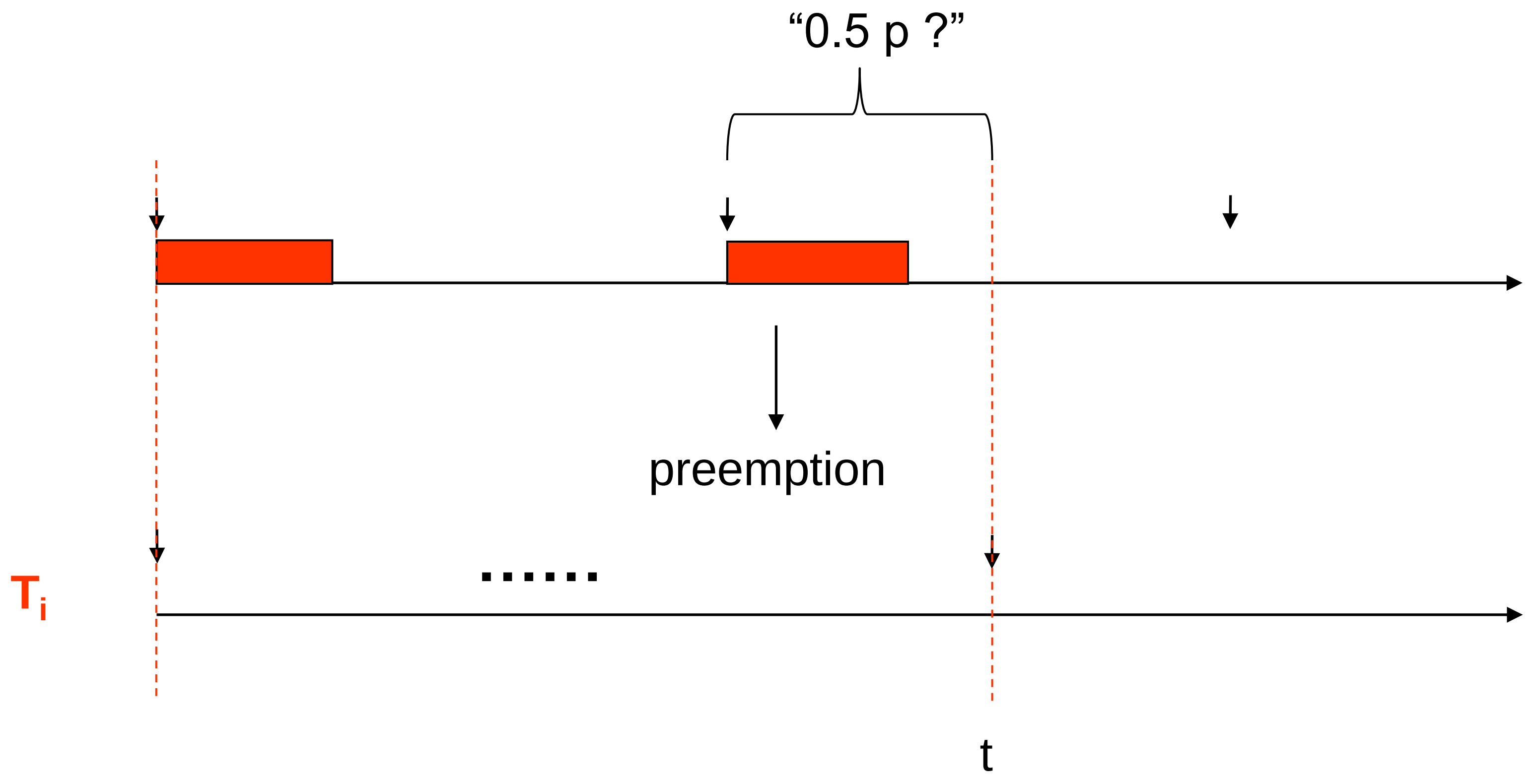
- The "interference" from high-priority tasks in the first period of T_i is never larger than

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



Rate-Monotonic-Scheduling

- Critical instance: Why ceiling function?

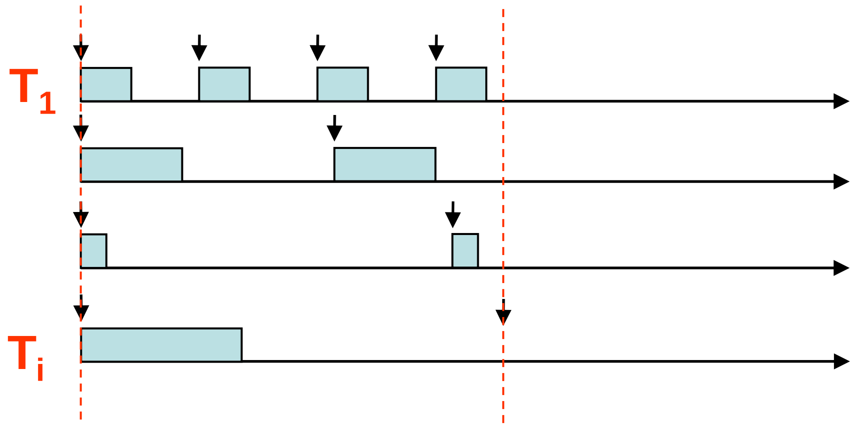


Rate-Monotonic Scheduling

- (Recursive) Response time analysis
  - The response time of a job of T_i at the critical instance can be computed by a recursive function

$$r_0 = \sum_{1 \ldots i} c_i$$

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



  - Observation: r may or may not converge before p_i

Rate-Monotonic Scheduling

- ***Theorem***: A task set={T_1,T_2,…,T_n} is schedulable by RM if and only if the worst-case response times of all tasks are shorter than their periods

- Observations
  - If all task survive their critical instance, then they survive any task phasing
  - The analysis is an exact schedulability test for RMS, aka "Rate-Monotonic Analysis (RMA)"

Rate-Monotonic Scheduling

- Example: T1=(2,5), T2=(2,7), T3=(3,8)
  - T1:
    - $R_0=2 \leq 5$ ok
  - T2:
    - $R_0=2+2=4 \leq 7$
    - $R_1=2*\lceil 4/5 \rceil+2*\lceil 4/7 \rceil=4 \leq 7$ ok
  - T3:
    - $R_0=2+2+3=7 \leq 8$
    - $R_1= 2*\lceil 7/5 \rceil+2*\lceil 7/7 \rceil+ 3*\lceil 7/8 \rceil =9> 8$ failed
  - Note: every task must pass this test!

Let's try
{(1,3),(1,6),(6,12)}   {(3,6),(3.1,9)}

Rate-Monotonic Scheduling

- (informal) Proof:
  - If the response time converges at r_n, then the lowest-priority job completes at r_n
  - If r_n is before p_n , then the lowest-priority job meets its deadline for its critical instance
  - Since the job survives the critical instance, it will survives any task phasing
  - All tasks must be tested for their critical instance

Rate-Monotonic Scheduling

- Test every task 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
    - Does this mean {T1,T2,T3} schedulable?
  - No, T2 fails the test when considering {T1, T2}
    - This task set is not schedulable!!!

Rate-Monotonic Scheduling

- Time complexity
  - $O(n^2 * p_n)$, pseudo-polynomial time
  - Very fast when task periods are harmonically related
  - Would be *extremely slow* when periods of tasks are prime to each other

(2,4),(4,7),(1,100)→ T3: 14 interactions, fails

(2,5),(4,7),(1,100)→ T3: 11 interactions, succeeds

(2,4),(9,20),(1,100)→ T3: 4 interactions, succeeds

T1	T2	T3	R0	7
2	4	1	R1	9
4	7	100	R2	15
			R3	21
			R4	25
			R5	31
			R6	37
			R7	45
			R8	53
			R9	61
			R10	69
			R11	77
			R12	85
			R13	97
			R14	107
			R15	120

T1	T2	T3	R0	7
2	4	1	R1	9
5	7	100	R2	13
			R3	15
			R4	19
			R5	21
			R6	23
			R7	27
			R8	29
			R9	33
			R10	35
			R11	35
			R12	35
			R13	35
			R14	35
			R15	35

T1	T2	T3	R0	12
2	9	1	R1	16
4	20	100	R2	18
			R3	20
			R4	20
			R5	20
			R6	20
			R7	20
			R8	20
			R9	20
			R10	20
			R11	20
			R12	20
			R13	20
			R14	20
			R15	20

Rate-Monotonic Scheduling

- Observation
  - RTA is an exact test for fixed-priority scheduling, but it is not often used, especially not in dynamic systems, because of its high time complexity
  - RTA is more suitable for static systems
- Are there any schedulability tests efficient enough for on-line use?
  - Should not be 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 is only one task
    - $c/p \leq 1$ (CPU utilization LEQ 100%)
  - The algorithm is efficient enough (i.e., O(1))
  - But useless

Rate-Monotonic Scheduling

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

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

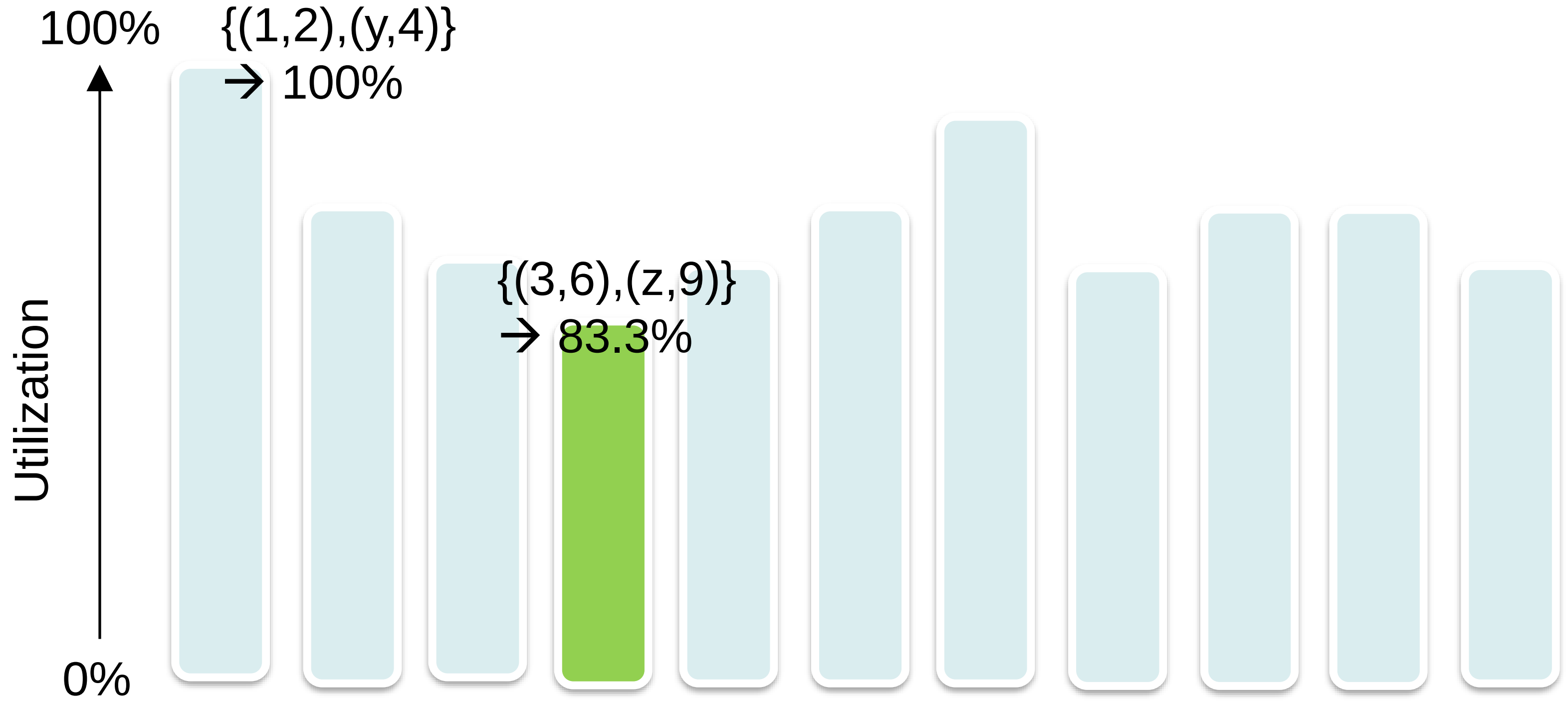
  - CPU utilization of a task set $\{T_1, T_2, \ldots, T_n\}$ is

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

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

Rate-Monotonic Scheduling

- To find "the lowest" among "the achievable processor utilizations of different task sets"
  - highly related to task periods



Rate-Monotonic Scheduling

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

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

"The least achievable CPU utilization of *n* tasks"

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

Rate-Monotonic Scheduling

- Proof: Let us consider 2 tasks only

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

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

The largest possible C2 is

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

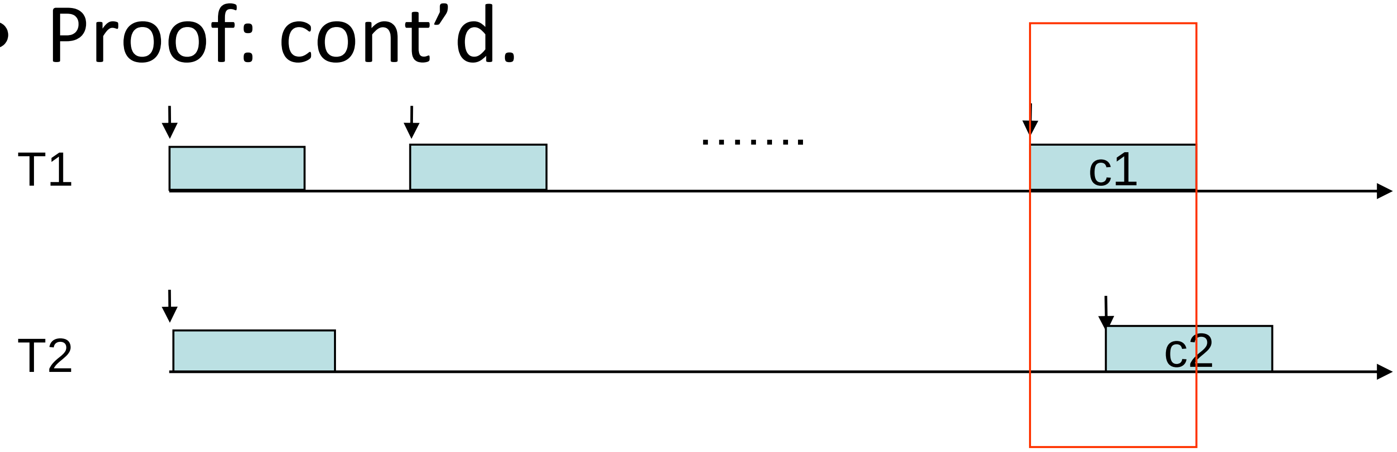
Achievable processor utilization is

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

- U monotonically decreases with C_1
  - C1's right-coefficient is negative because $1/P_1 < (1/P_2)(\lceil P_2/P_1 \rceil)$

Rate-Monotonic Scheduling

- Proof: cont'd.



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

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

The largest possible C_2 is

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

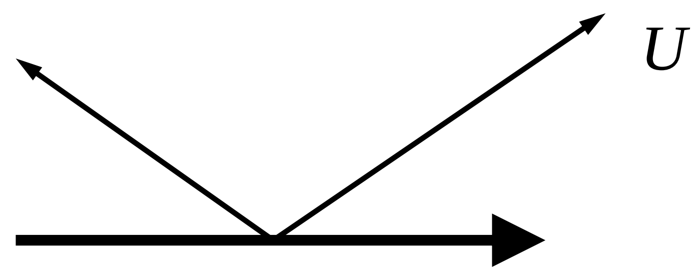
Achievable processor utilization

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

- 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

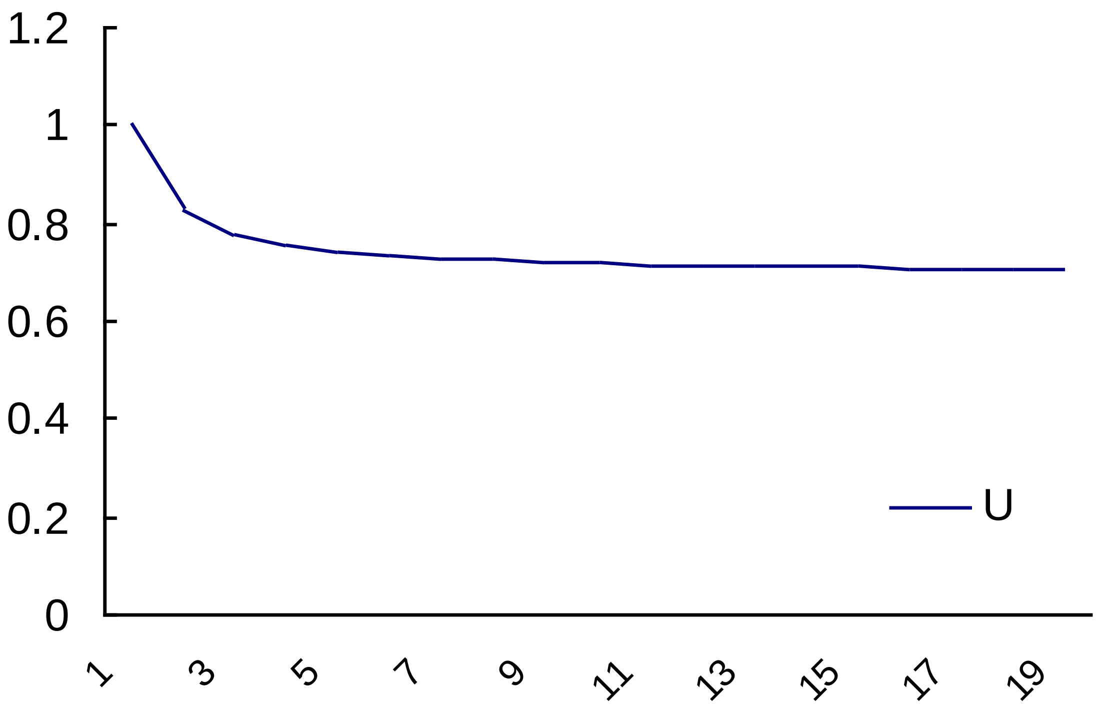
- The generalized result for *n* tasks is

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

- If a task set of n tasks whose total utilization is not larger than U(n), then this task set is guaranteed to be schedulable by RM
  - The time complexity of the test is O(n), which is efficient enough for on-line implementation

Rate-Monotonic Scheduling

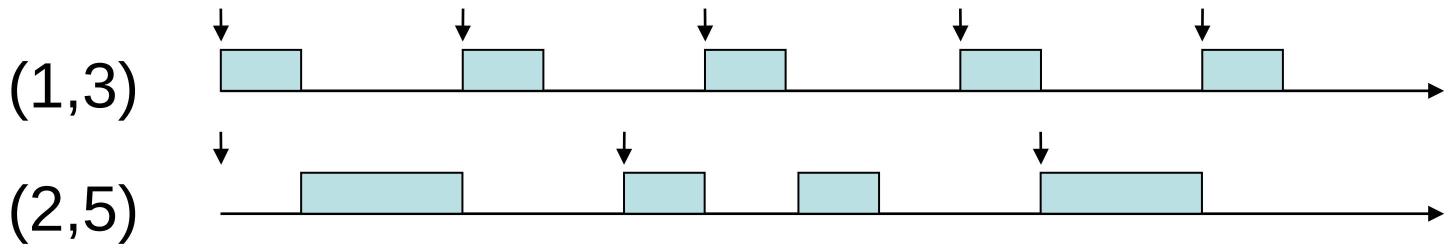
- When x → infinitely large, U(x)→0.68



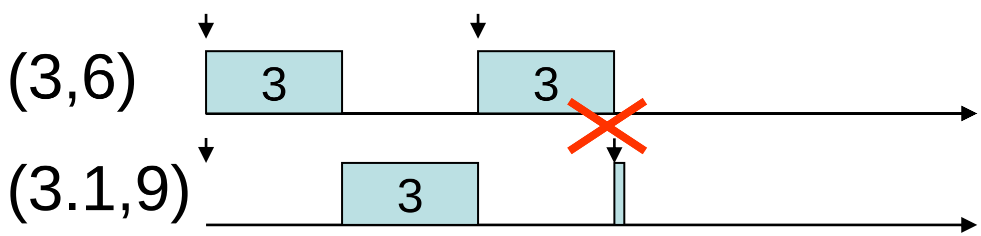
1	1
2	0.828427
3	0.779763
4	0.756828
5	0.743492
6	0.734772
7	0.728627
8	0.724062
9	0.720538
10	0.717735
11	0.715452
12	0.713557
13	0.711959
14	0.710593
15	0.709412

Rate-Monotonic Scheduling

- Example 1: (1,3), (2,5)
  - Utilization =0.73 $\leqq$ U(2)=0.828

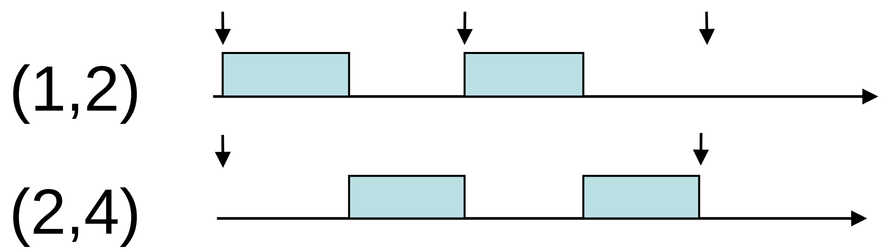


- Example 2: (3,6), (3.1,9)
  - Utilization =0.84 > U(2)=0.828



Rate-Monotonic Scheduling

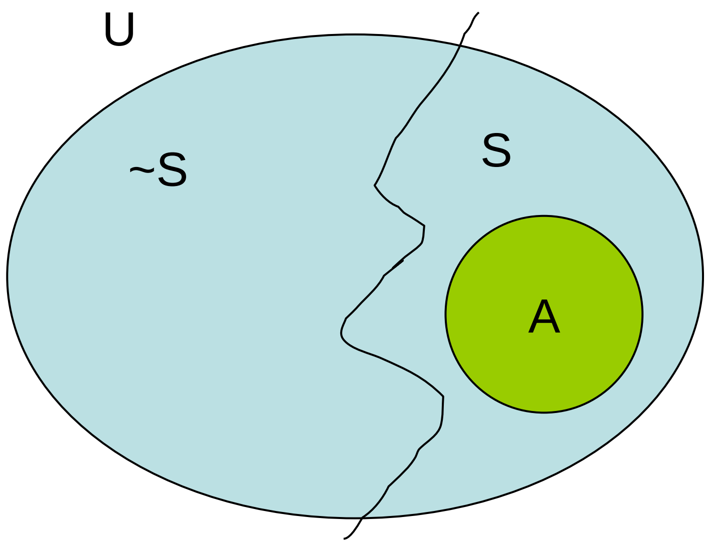
- Example 3: (1,2), (2,4)
  - Utilization =100% > U(2)=0.828



- Example 2 and 3 shows that, we know nothing about whether a taskset is schedulable if its total utilization is greater than the U(n) bound!

Rate-Monotonic Scheduling

- U bound test: Sufficient but not necessary
  - Utilization test provides a fast way to check if a task set is schedulable
  - A task set that fails the utilization test: maybe schedulable or maybe not schedulable



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

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

A: Those pass the 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
    - Hyper-period method
    - Response time analysis (RMA)
  - Sufficient tests (LL U Bound Test) for fast test
    - Sufficient condition

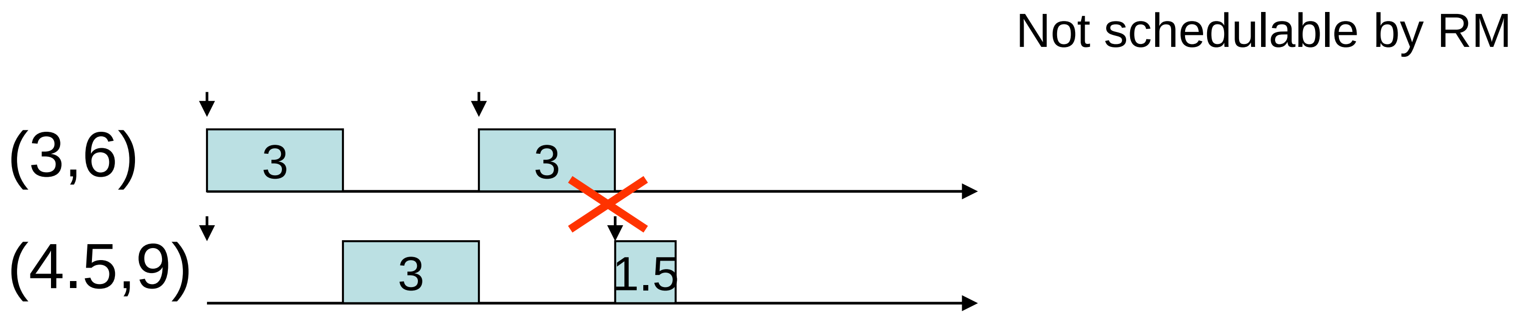
Earliest-Deadline First (Dynamic-Priority Scheduling)

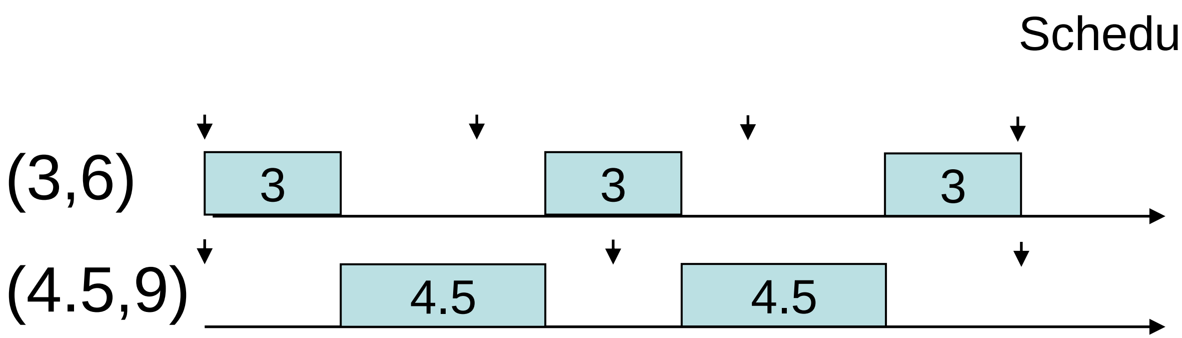
Earliest-Deadline-First Scheduling

- Definition
  - Feasible
    - A set of tasks is feasible if there exists some way to schedule the tasks without any deadline violations
  - Schedulable
    - Given a scheduling algorithm A
    - A set of tasks is schedulable by A, if algorithm A successfully schedule the tasks without any deadline violations
- Example
  - {(3,6),(4.5,9)} is not schedulable by RM, but is schedulable by EDF. Therefore, the task set is feasible.
  - See the next slice

Earliest-Deadline-First Scheduling

- Example





Earliest-Deadline-First Scheduling

- If for an algorithm, schedulable ←→ feasible
  - then it is a *universal* scheduling algorithm

- What are the universal scheduling algorithms for periodic and preemptive uniprocessor systems?
  - EDF, LLF/LSF, and more

Earliest-Deadline-First Scheduling

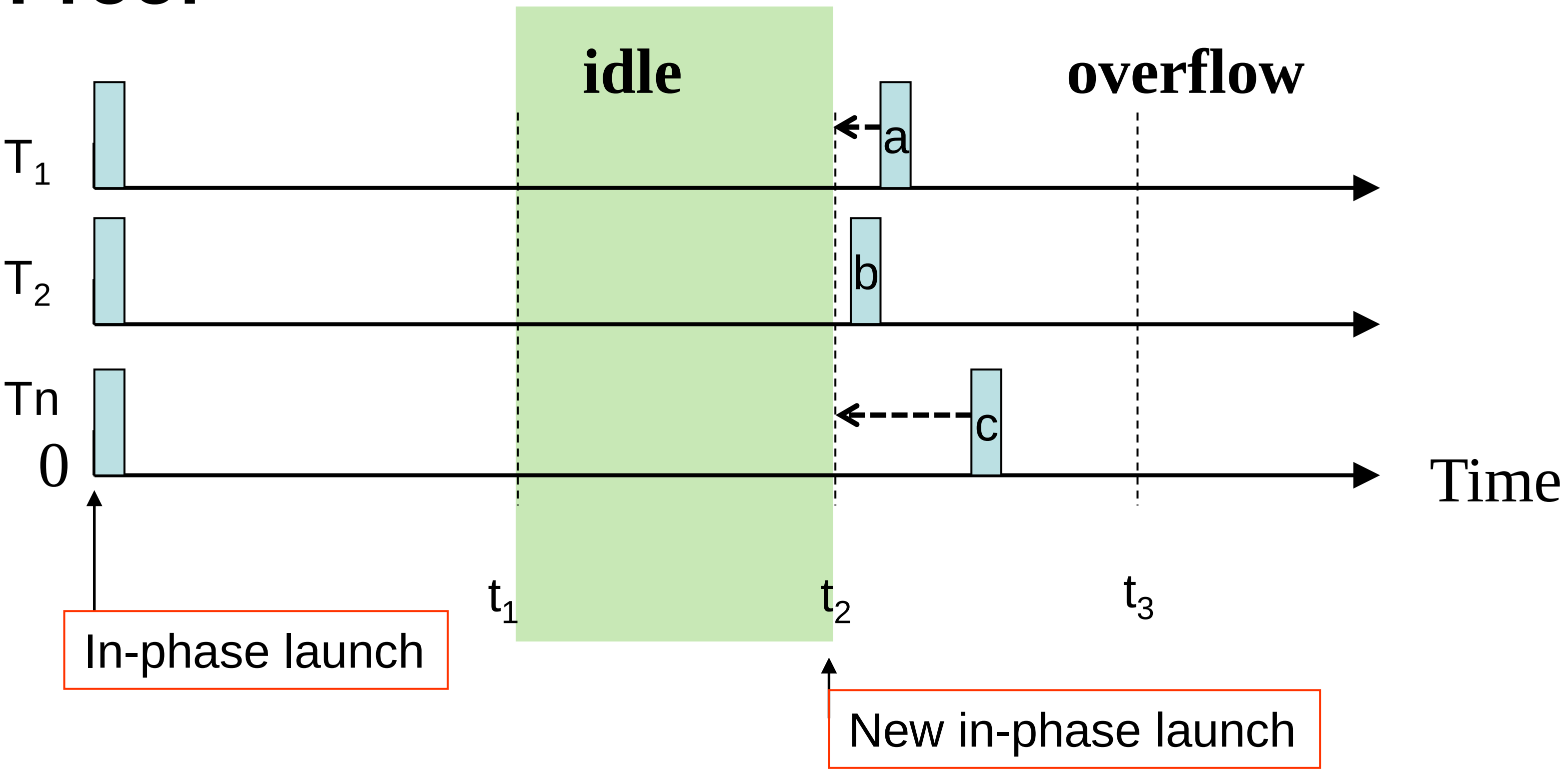
- EDF always executes a job whose deadline is the earliest
  - The earlier the deadline of a job is, the more urgent the job is
  - Task “priorities” are not static
  - No such “Task A has a higher priority than task B”, better say “Job A is more urgent than job B”

Earliest-Deadline-First Scheduling

- Observation: The critical instance of a task for EDF is the same as that for RM

- ***Lemma***: With EDF, there is no idle time before an overflow
  - This is a very strong statement that implies the optimality of EDF in terms of schedulability

Proof



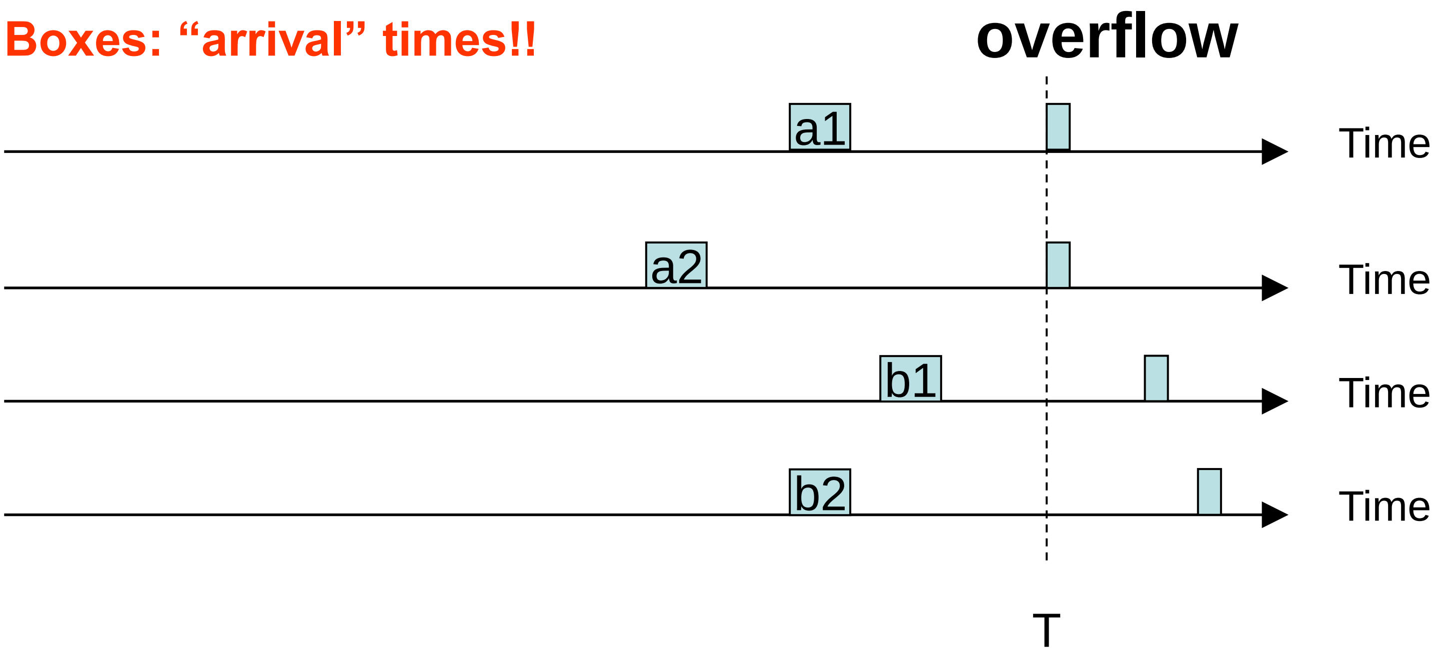
- Consider in-phase launching of all tasks. Suppose that there is an overflow at time t_3, and the processor idles between t_1 and t_2
- If we move “a” forward to be aligned to t_2, the overflow would occur earlier than it was (i.e., at or before t_3)
  - That is because EDF’s discipline: moving forward 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 , forming another in-phase launch
  - →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 not higher than 1

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

Boxes: "arrival" times!!



←: 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

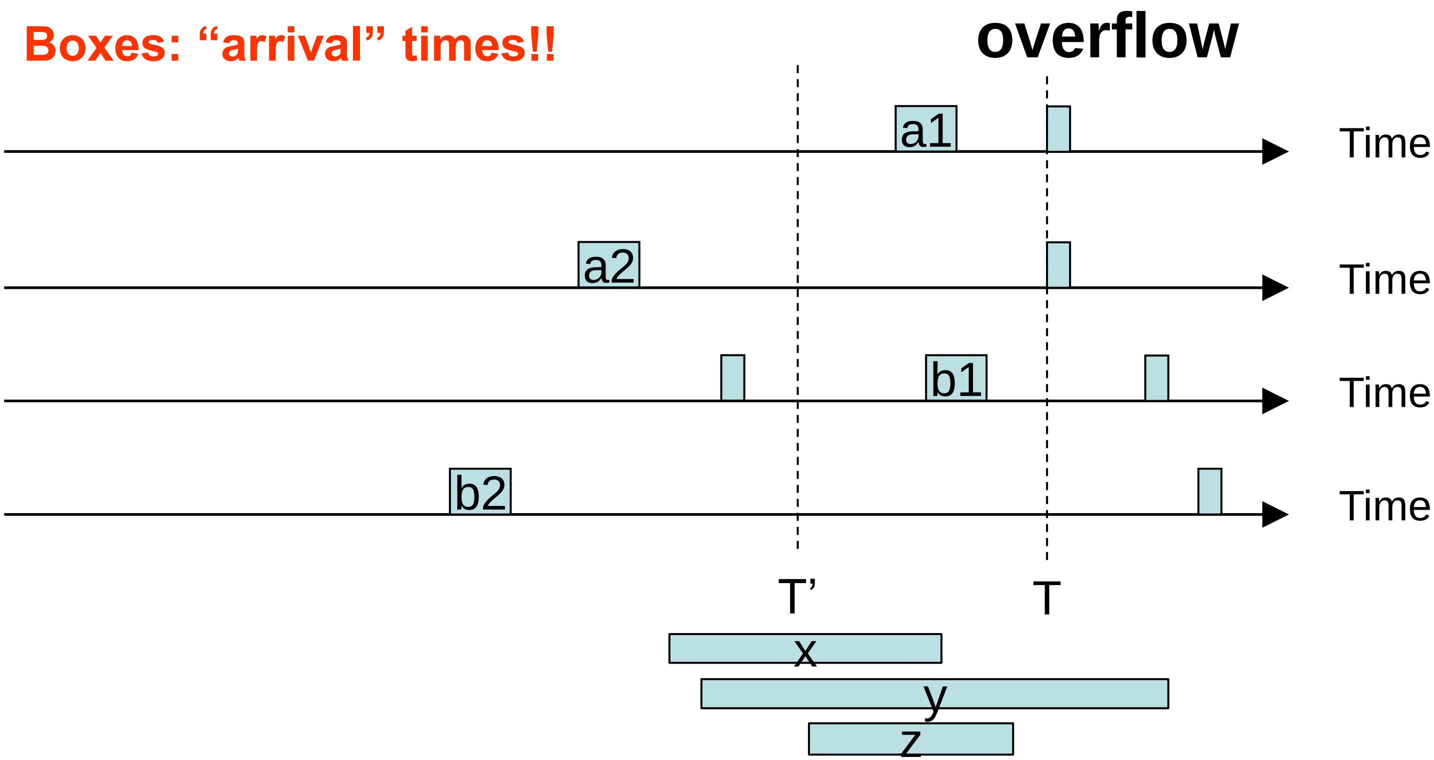
$$C_1(\lfloor T/P_1 \rfloor) + C_2(\lfloor T/P_2 \rfloor) + \ldots + C_n(\lfloor T/P_n \rfloor)$$

  - 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!!



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
- Before T’
  - Job x: already completed, Job y: not affecting a’s, Job z: will interfere a’s
- Back to [T’,T], the total computation demand is no less than

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

  - Since there is no idle before the deadline violation, so

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

  - $\rightarrow\leftarrow$

Earliest-Deadline-First Scheduling

- Summary
  - A universal scheduling algorithm for real-time periodic tasks
  - Dynamic priority scheduling (job priorities are static, however)

Independent Task Scheduling

- Summary
  - Tasks share nothing but the CPU
    - Periodic and preemptive
  - Priority-driven scheduling vs. deadline-driven scheduling
    - Robustness vs. utilization
  - Admission control policies
    - On-line tests vs. exact tests

Comparison

	RM	EDF
Optimality	Optimal for fixed-priority scheduling	Universal
Schedulability test	Exact test is slow (PP), conservative tests O(n)	O(n) for exact test
Overload survivability	High and predictable	Unmanageable
Responsiveness	High priority tasks always have shorter response time	We don’t know
Ease of implementation	Pretty simple	Complicated
Run-time overheads (like preemption)	Should be low??	Should be high??

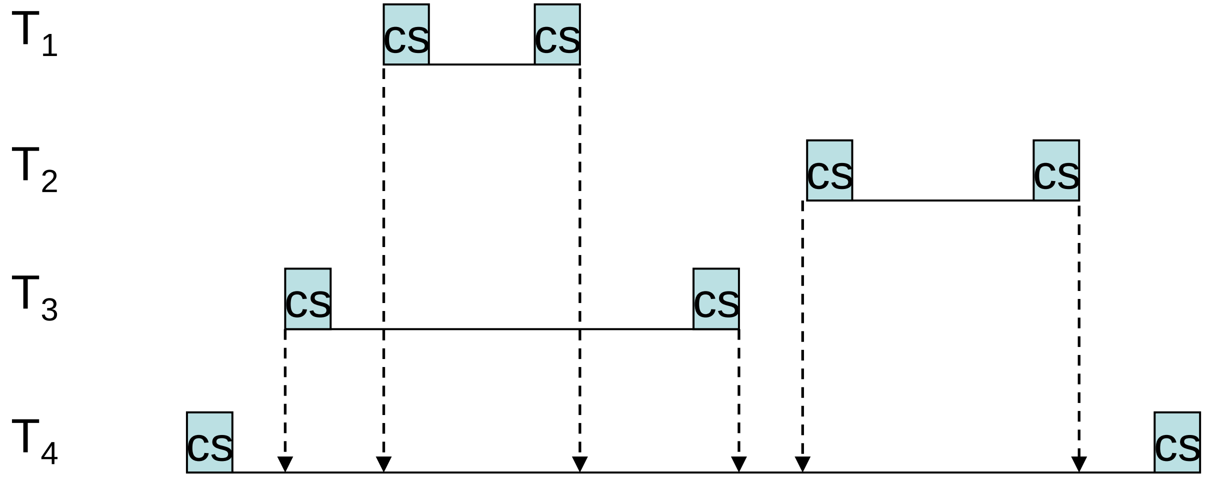
Advanced Topics

Context-Switch Overhead

- Context switching takes time
- For RM
  - A job can preempt/be preempted by others
  - How to take the cxtsw overhead into account? Do we need the RTA-like method for analysis?
- The cxtsw cost should be associated with the preempting job, not the preempted job
  - Let the time cost of a cxtsw operation be x
  - Add 2x to the computation time c. Done.

Context-Switch Overhead

- Context-switch overheads under RM
  - (c,p) → (c+2x, p)
  - ***Proof***. A task only preempts once, unless it suspends in the middle of execution

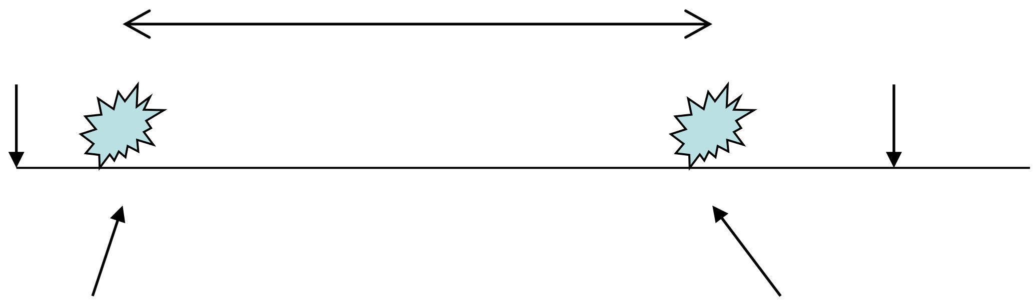


Context-Switch Overhead

- Preemption overhead in EDF
  - ***Paradox***: because EDF is dynamic-priority scheduling, any two arbitrary tasks can preempt each other and preemption may be more frequent in EDF than in RM
  - ***Fact***: Context switch overheads of a job in EDF are the same as that in RM (i.e., 2x)

Remark: Preemption in EDF

- Another fact: In EDF, a task can only be preempted by shorter-period ones, just like in RM



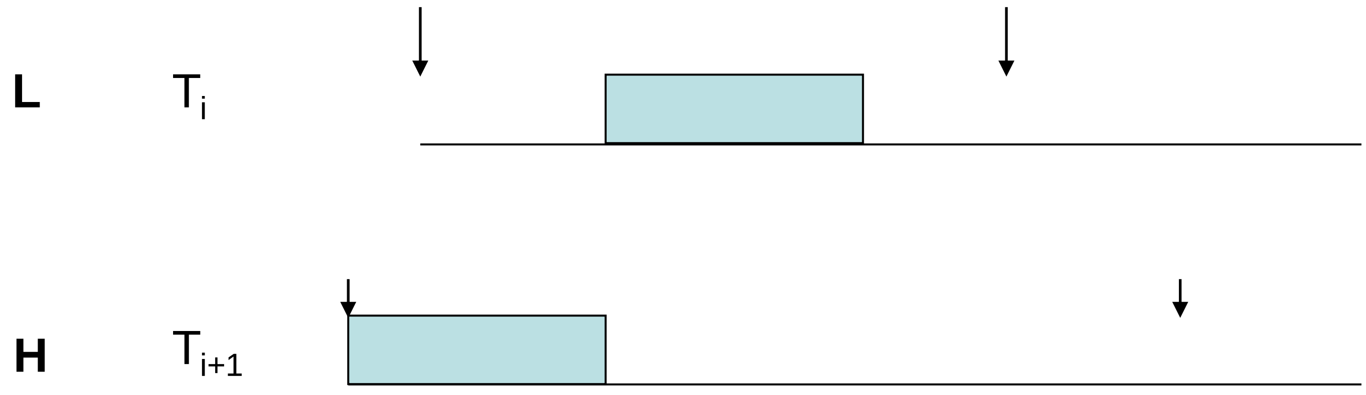
If the job is “preempted”, then the preempting job arrives later than the preempted job

If the job is previously “preempted”, then the preempting job has an earlier deadline than the preempted job

- Then what makes EDF different from RMS?
  - A job may be “delayed” by a longer-period one (see the example of (3,6),(4.5,9) for EDF)

Optimality of RM

- ***Theorem***: If a task set is schedulable by fixed-priority scheduling with an arbitrary priority assignment, then the task set is schedulable by RM
- ***Proof***. To swap priorities until it becomes RM



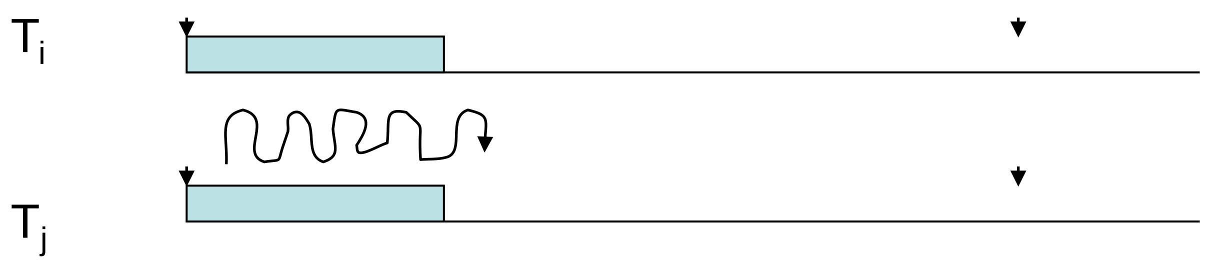
Optimality of RM

- Arbitrary task priority assignment
  - With a non-RM priority assignment, the U(n) test is no longer applicable!
  - Utilization test is for RM only

- RTA is still applicable
  - Let’s exercise RTA for L=(1,4) and H=(3,8)
  - Of course, hyper periods can be used too

Optimality of EDF

- EDF is universal to periodic, preemptive tasks
- Least-Slack-Time (LST) or Least-Laxity First (LLF) is also universal to periodic, preemptive tasks
  - At any time instant, run the job having the least slack time
  - Let's try {(8,16),(8,18)}
  - Problem of LST: highly frequent context switches



Optimality of EDF

- The good(s) of EDF
  - [Liu and Layland] EDF is universal to periodic, preemptive tasks with arbitrary arrival times
  - [Jackson’s Rule] EDF is optimal to non-periodic, non-preemptive jobs whose ready times are all 0
  - [Horn’s Rule] EDF is optimal to preemptive and non-periodic jobs with arbitrary arrival times

Optimality of EDF

- The not-good(s) of EDF
  - [Jeffay] EDF is not optimal to periodic, non-preemptive tasks with arbitrary arrival times (NP-complete)
  - [Mok] EDF is not optimal for multiprocessor partitioned scheduling (NP-complete)

Optimality of EDF

- [Jeffay] An interesting observation on non-preemptible EDF
  - Consider non-preemptive, periodic tasks (3,5) and (4,10) both become ready at time 0
  - Consider the same two tasks with release times 1 and 0

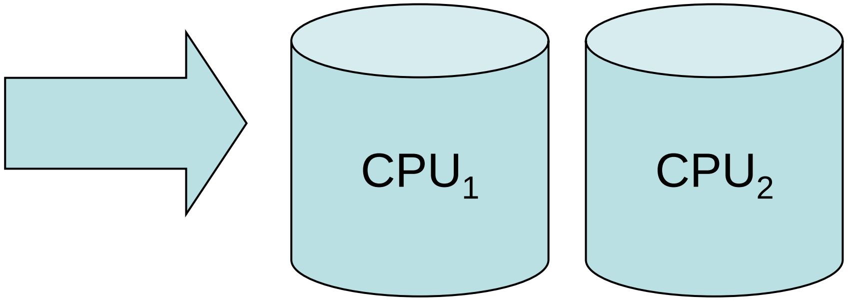
- The “Critical instance” succeeds but an “easier instance” fails?!

Optimality of EDF

- Limitation of EDF
  - [Mok] EDF is not optimal for multiprocessor partitioned scheduling (NP-complete)

{(5,10), (5,10), (8,12)}

EDF with load balancing



Harmonically-related tasks

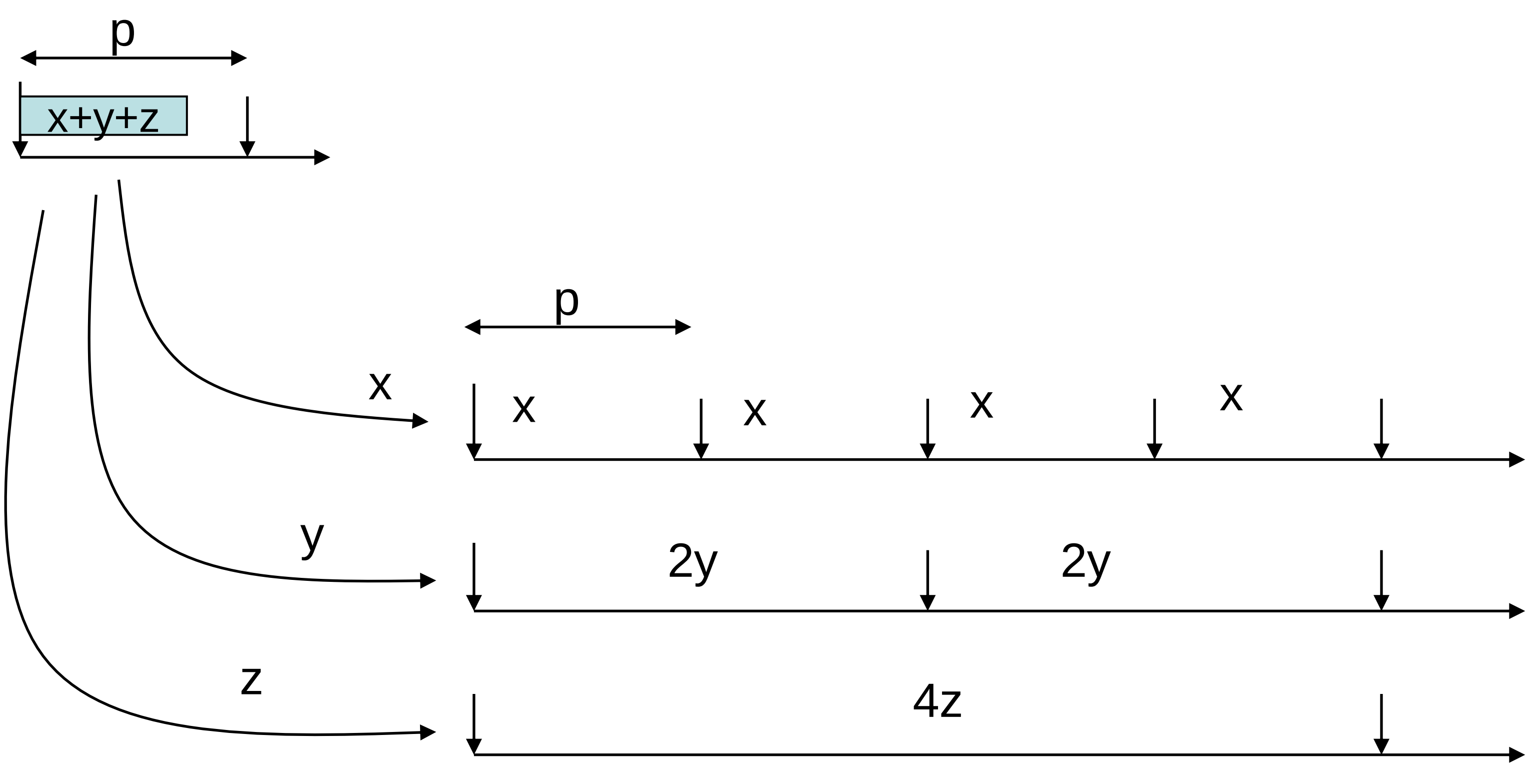
- Harmonic chain H is a set of tasks in which short periods divides long periods

Given a set of tasks $S = \{T_1, T_2, \ldots, T_n\}$ and harmonic chain $H = \{T'_1, T'_2, \ldots, T'_m\}$. If $\{(\sum_{T'_i \in H} c'_i \,(p'_1 / p'_i),\, p_1)\} \cup S$ is schedulable then $H \cup S$ is schedulable.

  - E.g., {(1,4),(1,8),(1,7),(1,16)}
    - {(1+0.5+0.25,4),(1,7)}
  - Useful in RM: a harmonic chain is represented by a task and thus small *n* in *U(n)* can be used

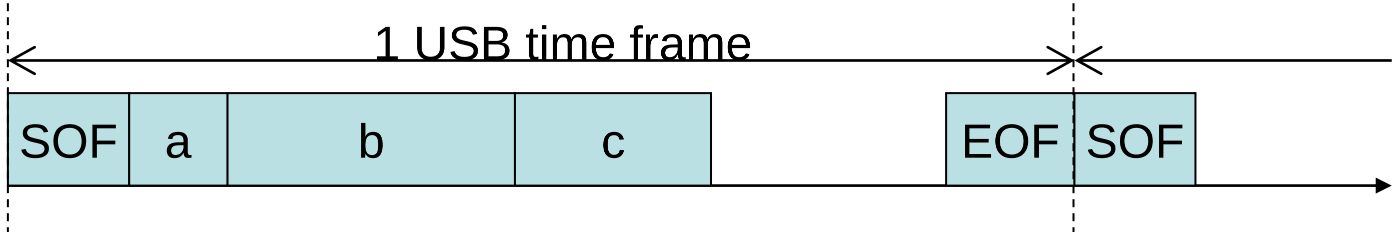
Harmonically-related tasks

– ***Proof***.



Cycle-Based Scheduling

- Cycle-based scheduling (A.K.A. Frame-based scheduling)
  - Many I/O buses divides time into frames
    - Requests are periodically services for every frame
  - A representative example: USB 1.1
    - USB use 1ms time frame to service isochronous requests
    - A transfer rate *r* KB/s is translated as to transfer ┌(r*1024)/1000┐ bytes every 1 ms frame
    - Very simple admission control: request sizes should not exceed the capacity of one time frame



SOF+a+b+c+EOF <= 1500 bytes (1ms for 1500 bytes)

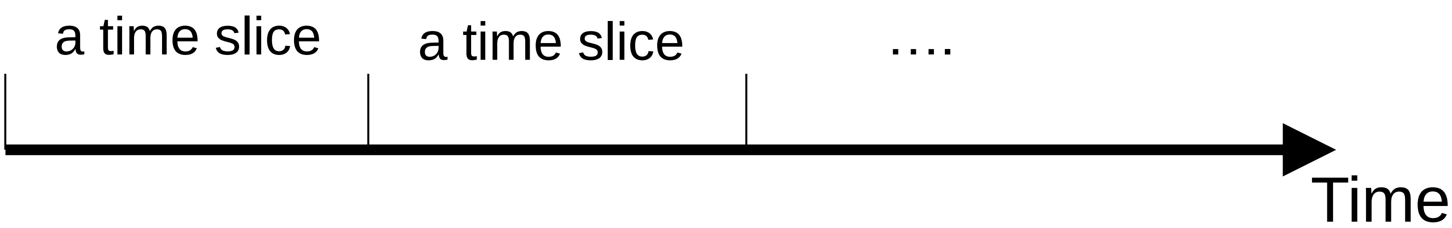
Cycle-Based Scheduling

- Cycle-based scheduling
  - Different from purely periodic tasks, tasks in cycle-based scheduling have the same period, i.e., the frame size.
  - If we care about “bandwidth” only , then cycle-based scheduling is very useful!

Cycle-Based Scheduling

Theorem: Given a set of m tasks, it is schedulable by some priority-driven scheduler if $U \leq 1$.

Proof.



For every time slice, τ_i receives a share of is c_i/p_i .

Within p_i , τ_i receives c_i!!

Could you disprove this paradox?

End of Chapter 1