Independent Task Scheduling

Chin-Fu Kuo
Real-Time and Embedded System Lab.
National University of Kaohsiung
Motivation

• Those shared can be scheduled
• 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, and each spends you 4.5 days
  – Course B: one homework is announced every 6 days, and each spends you 3 days

• You miss deadlines of one course if your policy favors either one course

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

• Schedule to meet deadlines (cont’d.)
  – Course A: (4.5, 9)
  – Course B: (3, 6)

• All deadlines are met if you do the project of the closest deadline

From this time point, your schedule repeats.
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

• 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
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 \([t+h,t+h+x]\)
Cyclic Executive

• **Proof:**

• From on $t'$, every loop arrives as does from $t$
Priority-Driven Scheduling: Fixed-Priority Algorithms
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

Arrival time (release time) → Relative deadline → Absolute deadline

Completion time

Computation time

Start time → Response time

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, ..., 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 $(c,p)$

\[ J_{1,1} \quad J_{1,2} \quad J_{1,3} \quad J_{1,4} \quad J_{1,5} \quad J_{1,6} \]

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

(1,3)  
(2,5)
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: “interference” from high-priority tasks is the maximum within the first period of $T_i$

$$\leq \sum_{j < i} c_j \left[ \frac{p_i}{p_j} \right]$$
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

\[
r_0 = \sum_{i} c_i
\]

\[
r_n = \sum_{i} c_i \left[ \frac{r_{n-1}}{p_i} \right]
\]

- Observation: the sequence of \( r_x, x \geq 0 \) 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: $T1=(2,5)$, $T2=(2,7)$, $T3=(3,8)$
  - $T1$:
    - $R_0=2 \leq 5$ ok
  - $T2$:
    - $R_0=2+2=4 \leq 7$
    - $R_1=2 * \left\lfloor \frac{4}{5} \right\rfloor + 2 * \left\lfloor \frac{4}{7} \right\rfloor = 4 \leq 7$ ok
  - $T3$:
    - $R_0=2+2+3=7 \leq 8$
    - $R_1=2 * \left\lfloor \frac{7}{5} \right\rfloor + 2 * \left\lfloor \frac{7}{7} \right\rfloor + 3 * \left\lfloor \frac{7}{8} \right\rfloor = 9 > 8$ failed

  - Note: each task succeeds $\Rightarrow$ the task set succeeds

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

  – \{T_1=(3,6), T_2=(3.1,9), T_3=(1,18)\}

  – Response analysis of T_3:
    • R_0=7.1, R_1=10.1, R_2=13.2, R_3=16.2, R_4=16.2<18

  – \{T_1, T_2, T_3\} is schedulable!? 

  – However, \{T_1, T_2\} 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

(2,4),(4,7),(1,100) → T3: 15 interactions, fails
(2,5),(4,7),(1,100) → T3: 11 interactions, succeeds
(2,4),(9,20),(1,100) → T3: 3 interactions, succeeds
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
    • $c_1/p_1 \leq 1$
  – 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 $\{T_1, T_2, ..., T_n\}$ 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

\[
\sum_{i=1}^{n} \frac{C_i}{p_i} \leq U(n) = n \left( 2^{1/n} - 1 \right)
\]

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

• Proof: Let us consider two tasks only

\[ C_1 \leq P_2 - P_1 \left( \left\lfloor \frac{P_2}{P_1} \right\rfloor \right) \]

The largest possible \( C_2 \) is

\[ P_2 - C_1 \left( \left\lfloor \frac{P_2}{P_1} \right\rfloor \right) \]

Total utilization factor is

\[ U = 1 + C_1 \left( \frac{1}{P_1} - \frac{1}{P_2} \left( \left\lfloor \frac{P_2}{P_1} \right\rfloor \right) \right) \]

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

\[
U = \left( \frac{P_1}{P_2} \right) \lfloor P_2 / P_1 \rfloor + C_1 \left( \frac{1}{P_1} - \frac{1}{P_2} \right) \left( \lfloor P_2 / P_1 \rfloor \right)
\]

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
    \[ C_1 = P_2 - P_1 \left\lfloor \frac{P_2}{P_1} \right\rfloor \]
  – By some differentiation, the minimal achievable utilization is
    \[ U(2) = 2 \left( 2^{1/2} - 1 \right) \]
Rate-Monotonic Scheduling

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

![Graph showing utilizations and feedback messages: Too pessimistic, Better, but not good enough, You hit the point!, Keep going!, 0%, 100% Utilization]
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 $x \rightarrow$ infinitely large, $U(x) \rightarrow 0.68$
Rate-Monotonic Scheduling

• Example 1: (1,3), (2,5)
  – Utilization = $0.73 \leq U(2) = 0.828$

• 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

(1,2)

(2,4)

• 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 imply that it is not schedulable

![Diagram showing U (universe of task sets), ~S (task sets unschedulable by RMS), S (task sets schedulable by RMS), and A (those can be found by utilization test)]

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

• If an algorithm schedulable $\iff$ 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

- 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

• Example

(3,6)  [3]  [3]  [X]
(4.5,9)   [3]  [1.5]
Not scheduable by RMS

(3,6)  [3]  [3]  [3]
(4.5,9)   [4.5]  [4.5]  [3]
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
• 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: $\rightarrow$ is easy, $\leftarrow$ requires some reasoning similar to the proof of the last theorem
\(\rightarrow\): suppose that \(U \leq 1\) 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 \(U > 1\)
  - \(\rightarrow \leftarrow\)
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
  • If some job b’s is executed before T, they must be executed before T’
    • That means all job a’s must be completed before T’
      • because job a’s have deadlines earlier than that of job b’s.
      • If b’s can run then a’s have been completed!

→←
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

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

Is it true?
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
Advanced Topics
Advanced Topics

- RMS vs. EDF, revisited
  - Preemption cost
  - Optimality
- Earliest-Deadline-First, revisited
  - What’s its limitations?
- Rate-Monotonic Scheduling, revisited
  - Harmonically-related tasks
  - More on critical instant
  - Arbitrary task phasing
  - Arbitrary priority assignments
- Periods vs. Deadlines
- Cycle-based scheduling
RMS vs. EDF, revisited

• Preemption overheads
  – RMS
    • A job could preempt another job and/or be preempted by other jobs
    • The cost of context switch should be associated to the job preempting other jobs
      – Let the time cost of a task preempting another task be $x$, its computation time should be added to $2x$
RMS vs. EDF, revisited

• Preemption overheads
  – RMS/EDF
    • The execution time of a task should be added to 2x
    • **Proof.** any preempted task would resume its execution only if the preemption tasks complete

Like an LIFO
RMS vs. EDF, revisited

- Preemption overheads
  - *Paradox*: because EDF is a dynamic scheduling algorithm, a job may preempt other jobs for multiple times
  - *Fact*: A job can only be preempted by jobs having relative deadlines (i.e., task periods) shorter than its
  - Context switch overheads are accounted in the same way as that for RMS (i.e., 2x+c)
RMS vs. EDF, revisited

• Preemption overheads
  – EDF

  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 differ from RMS?
  • A job may be “delayed” by a job having a longer period (see the example of (3,6),(4.5,9) for EDF)
RMS vs. EDF, revisited

- Optimality of RMS and EDF
  - **Theorem**: If a task set is schedulable by fixed-priority scheduling with an arbitrary priority assignment, then the task set is also schedulable by RMS.
  - **Proof**: To swap priorities until it becomes RM.
RMS vs. EDF, revisited

• Optimality of RMS and EDF
  – EDF is a universal scheduling for periodic tasks
  – There is another universal scheduling algorithm, Least-Slack-Time (LST) or Least-Laxity (LL)
    – At any time instant, run the job having the least slack time
    – Problem of LST: uncontrolled context switches
Earliest-Deadline-First, revisited

• Limitation of EDF
  – Good news:
    • [Liu and Layland] EDF is universal for periodic and preemptive tasks with arbitrary arrival times
    • [Jackson’s Rule] EDF is optimal for non-periodic and non-preemptive jobs which are all ready at time 0
    • [Horn’s Rule] EDF is optimal for preemptive and non-periodic jobs with arbitrary arrival times
Earliest-Deadline-First, revisited

• Limitation of EDF
  – Bad news:
    • [Gary and Johnson] EDF is not optimal for non-periodic and non-preemptive jobs with arbitrary arrival times (NP-complete)
    • [Jeffay] EDF is not optimal for periodic and non-preemptive tasks with arbitrary arrival times (NP-complete)
    • [Mok] EDF is not optimal for multiprocessor scheduling (NP-complete, without task migration)
Earliest-Deadline-First, revisited

- Limitation of EDF
  - [Gary and Johnson] EDF is not optimal for non-preemptive jobs with arbitrary arrival times (NP-complete)
Earliest-Deadline-First, revisited

- Limitation of EDF
  - [Mok] EDF is not optimal for multiprocessor scheduling (NP-complete, without task migration)
Earliest-Deadline-First, revisited

• [Jeffay] A very interesting observation on EDF
  – Consider non-preemptive, periodic tasks (3,5) and (4,10) released simultaneously at time 0
  – Consider the same two tasks, with release times 1 and 0

• Observations
  – EDF is not universal for this case
  – Critical instant?!
Rate-Monotonic Scheduling, revisited

• Harmonically-related tasks
  – Harmonic chain $H$ is a set of tasks in which any task period can be divided by the shorter task periods
    • E.g., $2 \rightarrow 6 \rightarrow 12 \rightarrow 36$
  – Given a set of tasks $S = \{T_1, T_2, ..., T_n\}$ and harmonic chain $H = \{T'_1, T'_2, ..., T'_m\}$. If $\{(\sum_{T'_i \in H} c'_i (p'_1/p'_i), p_1)\} \cup S$ is schedulable then $H \cup S$ is schedulable.
  – A higher utilization bound can be used for admission control because the “number of tasks” is decreased
Rate-Monotonic Scheduling, revisited

• Harmonically-related tasks
  – Proof.
Rate-Monotonic Scheduling, revisited

• More on critical instant (preemptive and periodic task)
  – If tasks are not in phase, critical instant might not happen
  – Let’s see (4.5, 9) and (3, 6)
    • Offset the former by 1.5
  – If a set of tasks succeed RMA for critical instant → they succeed with a given task phasing
    • \( \leftarrow \) is not necessarily true!!
Rate-Monotonic Scheduling, revisited

• Often we use RMA to analyze the schedulability for the case of in phase
  – If tasks are not in phase, RMA still works

• To analyze tasks with initial offsets is much of theory interests

• Tasks are not in phase is not so difficult compared the case of in phase
Rate-Monotonic Scheduling, revisited

- Arbitrary priority assignment
  - Under an arbitrary priority assignment, the utilization test is no longer applicable!
    - However, RMA can still be adopted
    - Let’s exercise RMA for (1,4) and (3,8)
Periods vs. Deadlines

\[ \sum \frac{c_i}{p_i} \leq 1 \]

• Pre-period deadlines with preemptible EDF
  – Sufficient, necessity is unsure
• Post-period deadlines with preemptible EDF
  – Sufficient but not necessary
Periods vs. Deadlines

- Per-period deadlines with preemptible EDF

\[ \forall L > 0, \sum_{i=1}^{n} \left[ \frac{L + T_i - D_i}{T_i} \right] C_i \leq L \]

- An exact test (sufficient and necessary)
- Pseudo polynomial time
Periods vs. Deadlines

• Pre-period deadlines with preemptible “fixed-priority scheduling”
  – Deadline-monotonic scheduling (DM) is optimal
  – A sufficient condition of schedulability
    \[ \sum \frac{c_i}{d_i} \leq 1 \]

• Post-period deadlines with preemptible RM
  – Unsure

• RMA is still an exact test for the two cases!!
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
    - USB use 1ms time frame to service isochronous requests
    - A transfer rate $r$ KB/s is translated as to transfer $\lceil r/1000 \rceil$ bytes every 1 ms frame
    - Very simple admission control: request sizes should not exceed the capacity of one time frame

\[
\text{SOF} + a + b + c + \text{EOF} \leq 1500 \text{ bytes}
\]
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.
  – Periodic task (3,6) may not be emulated by (0.5,1) because the request arrives every 6 units of time!
    • Could be a hardware event
  – However, if we care about “bandwidth” only, then cycle-based scheduling is very useful!
Theorem: Given a set of m tasks, it is schedulable by some priority-driven scheduler if $U \leq 1$.

Proof.

For every time slice, $\tau_i$ receives a share of $c_i/p_i$.

Within $p_i$, $\tau_i$ receives $c_i!!$

Could you disprove this paradox?
Co-NP Hardness of Feasibility

• [Leung and Whitehead] If tasks do not arrive in phase, the task set $T$ is not schedulable if and only if there is deadline violations in the interval

$$[0, \max(o_i)+2 \cdot h)$$

• $o_i$ is the initial offset of task $\tau_i$ and $h$ is the least-common multiplier of all the task periods