University of Glasgow | School of Computing Science

Real-time Scheduling of Periodic Tasks (2)

Advanced Operating Systems
Lecture 3

Lecture Outline

- The rate monotonic algorithm (cont'd)
  - ...
  - Maximum utilisation test
- The deadline monotonic algorithm
- The earliest deadline first algorithm
  - Definition
  - Optimality
  - Maximum utilisation test
- The least slack time algorithm
- Discussion

Rate Monotonic: Other Scheduling Tests

- Exhaustive simulation and time-demand analysis complex and error prone
- Simple scheduling tests derived for some cases:
  - Simply periodic systems
  - Maximum utilisation test

Simply Periodic Systems

- In a *simply periodic* system, the periods of all tasks are integer multiples of each other
  - $p_k = n \cdot p_i$ for all i, k such that $p_i < p_k$ where n is a positive integer
  - True for many real-world systems, since easy to engineer around multiples of a single run loop

Simply Periodic Rate Monotonic Tasks

- Rate monotonic optimal for simply periodic systems
  - A set of *simply periodic*, independent, preemptable tasks with $D_i \geq p_i$ can be scheduled on a single processor using RM provided $U \leq 1$
- Proof follows from time-demand analysis:
  - A simply periodic system, assume tasks in phase
    - Worst case execution time occurs when tasks in phase
- T_i misses deadline at time t where t is an integer multiple of p_i
  - Again, worst case ⇒ $D_i = p_i$
- Simply periodic ⇒ t integer multiple of periods of all higher priority tasks
- Total time required to complete jobs with deadline $\leq t$ is $\sum_{k=1}^{i} \frac{e_k}{p_k} t = t \cdot U_i$
- Only fails when $U_i > 1$

Maximum Utilisation Tests

- Simply periodic systems have a simple *maximum utilisation* test

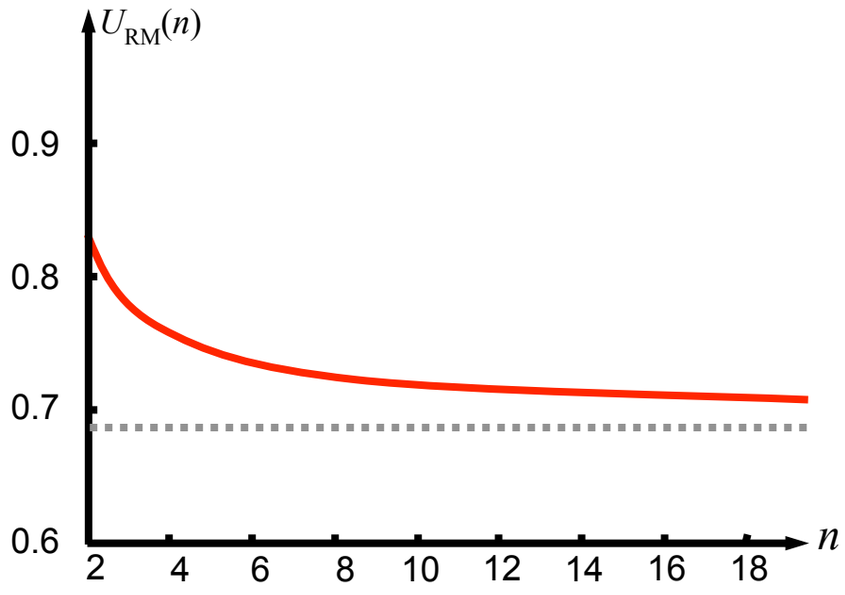
- Possible to generalise the result to general rate monotonic systems

- Derive a maximum utilisation, such that it is guaranteed a feasible schedule exists provided the maximum is not exceeded

RM Maximum Utilisation Test: $D_i = p_i$

- A system of n independent preemptable periodic tasks with $D_i = p_i$ can be feasibly scheduled on one processor using rate monotonic if $U \leq n \cdot (2^{1/n} - 1)$

  - $U_{\mathrm{RM}}(n) = n \cdot (2^{1/n} - 1)$
  - For large $n \rightarrow \ln 2$ (i.e., $n \rightarrow 0.69314718056\ldots$)



See Jane W. S. Liu, “Real-time systems”, Section 6.7 for proof

- $U \leq U_{\mathrm{RM}}(n)$ is a *sufficient, but not necessary*, condition – i.e., a feasible rate monotonic schedule is guaranteed to exist if $U \leq U_{\mathrm{RM}}(n)$, but might still be possible if $U > U_{\mathrm{RM}}(n)$

RM Maximum Utilisation Test: $D_i = v \cdot p_i$

- Maximum utilisation varies if relative deadline and period differ
- For n tasks, where the relative deadline $D_k = v \cdot p_k$ it can be shown that:

$$U_{RM}(n, v) = \begin{cases} v & \text{for } 0 \leq v \leq 0.5 \\ n((2v)^{\frac{1}{n}} - 1) + 1 - v & \text{for } 0.5 \leq v \leq 1 \\ v(n-1)[(\frac{v+1}{v})^{\frac{1}{n}-1} - 1] & \text{for } v = 2, 3, \ldots \end{cases}$$

(you are not expected to remember this formula – but should understand how the utilisation changes in general terms)

RM Maximum Utilisation Test: $D_i = \upsilon \cdot p_i$

n	$\upsilon = 4.0$	$\upsilon = 3.0$	$\upsilon = 2.0$	$\upsilon = 1.0$	$\upsilon = 0.9$	$\upsilon = 0.8$	$\upsilon = 0.7$	$\upsilon = 0.6$	$\upsilon = 0.5$
2	0.944	0.928	0.898	0.828	0.783	0.729	0.666	0.590	0.500
3	0.926	0.906	0.868	0.779	0.749	0.708	0.656	0.588	0.500
4	0.917	0.894	0.853	0.756	0.733	0.698	0.651	0.586	0.500
5	0.912	0.888	0.844	0.743	0.723	0.692	0.648	0.585	0.500
6	0.909	0.884	0.838	0.734	0.717	0.688	0.646	0.585	0.500
7	0.906	0.881	0.834	0.728	0.713	0.686	0.644	0.584	0.500
8	0.905	0.878	0.831	0.724	0.709	0.684	0.643	0.584	0.500
9	0.903	0.876	0.829	0.720	0.707	0.682	0.642	0.584	0.500
∞	0.892	0.863	0.810	0.693	0.687	0.670	0.636	0.582	0.500

$D_i > p_i \Rightarrow$ Maximum utilisation increases

$D_i < p_i \Rightarrow$ Maximum utilisation decreases

$D_i = p_i$

The Deadline Monotonic Algorithm

- Assign priorities to jobs in each task based on the relative deadline of that task
  - Shorter relative deadline → higher the priority
  - If relative deadline equals period, schedule is identical to rate monotonic
  - When the relative deadlines and periods differ: deadline monotonic can sometimes produce a feasible schedule in cases where rate monotonic cannot; rate monotonic always fails when deadline monotonic fails
  - Hence deadline monotonic preferred if deadline ≠ period

- Not widely used – periodic systems typically have relative deadline equal to their period

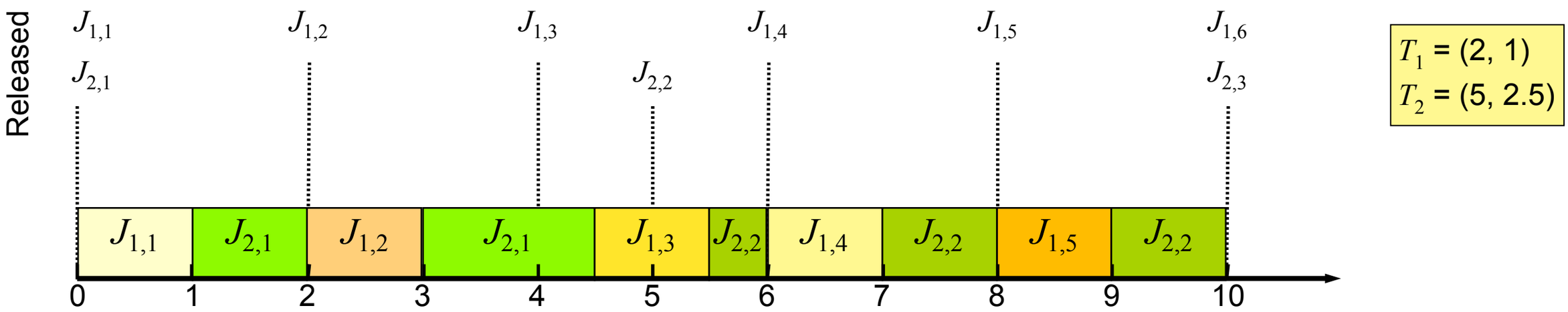
The Earliest Deadline First Algorithm

- Assign priority to jobs based on deadline: earlier deadline = higher priority
- Rationale: do the most urgent thing first
- Dynamic priority algorithm: priority of a job depends on relative deadlines of all active tasks
  - May change over time as other jobs complete or are released
  - May differ from other jobs in the task

Earliest Deadline First: Example

Time	Ready to run	Running
0	$J_{2,1}$	J
1		J
2	$J_{2,1}$	J
3		J
4	$J_{1,3}$	J
4.5		J
5	$J_{2,2}$	J
5.5		J
6	$J_{2,2}$	J
7		J

Time	Ready to run	Running
8	$J_{2,2}$	J
9		J
10	$J_{2,3}$	J
…	…	…



Earliest Deadline First is Optimal

- EDF is optimal, provided the system has a single processor, preemption is allowed, and jobs don't contend for resources
  - That is, it will find a feasible schedule *if one exists*, not that it will always be able to schedule a set of tasks
- EDF is not optimal with multiple processors, or if preemption is not allowed

Earliest Deadline First is Optimal: Proof

- Any feasible schedule can be transformed into an EDF schedule
  - If J_i is scheduled to run before J_k, but J_i's deadline is later than J_k's either:
    - The release time of J_k is after the J_i completes ⇒ they're already in EDF order
    - The release time of J_k is before the end of the interval when J_i executes:



    - Swap J_i and J_k (this is always possible, since J_i's deadline is later than J_k's)



    - Move any jobs following idle periods forward into the idle period



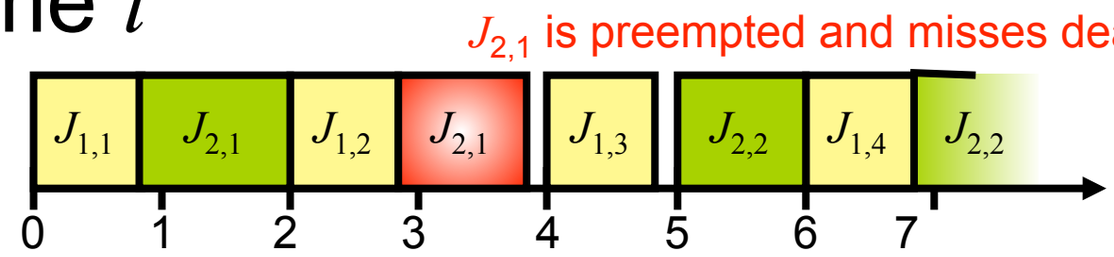
    - The result is an EDF schedule
- So, if EDF fails to produce a feasible schedule, no such schedule exists
  - If a feasible schedule existed it could be transformed into an EDF schedule, contradicting the statement that EDF failed to produce a feasible schedule [proof for LST is similar]

Maximum Utilisation Test: $D_i \geq p_i$

- Theorem:
  - A system of independent preemptable periodic tasks with $D_i \geq p_i$ can be feasibly scheduled on one processor using EDF if and only if $U \leq 1$
  - Note: result is independent of φ_i
- Proof follows from optimality of the system
  - [Proof in the book, Section 6.3.1]

Maximum Utilisation Test: $D_i < p_i$

- Test fails if $D_i < p_i$ for some i
  - E.g. T_1 = (2, 0.8), T_2=(5, 2.3, 3)



- However, there is an alternative test:
  - The density of the task, T_i, is $\delta_i = e_i / \min(D_i, p_i)$
  - The density of the system is $\Delta = \delta_1 + \delta_2 + \ldots + \delta_n$
  - Theorem: A system T of independent, preemptable periodic tasks can be feasibly scheduled on one processor using EDT if $\Delta \leq 1$.
- Note:
  - This is a sufficient condition, but not a necessary condition – i.e. a system is guaranteed to be feasible if $\Delta \leq 1$, but might still be feasible if $\Delta > 1$ (would have to run the exhaustive simulation to prove)

The Least Slack Time Algorithm

- Least Slack Time first (LST)
  - A job J_i has deadline d_i, execution time e_i, and was released at time r_i
  - At time $t < d_i$: remaining execution time $t_{\text{rem}} = e_i - (t - r_i)$
  - Assign priority based on least slack time, $t_{\text{slack}} = d_i - t - t_{\text{rem}}$
  - Two variants:
    - Strict LST – scheduling decision made whenever a queued job's slack time becomes smaller than the executing job's slack time – high overhead, not used;
    - Non-strict LST – scheduling decisions made only when jobs release or complete
- More complex, requires knowledge of execution times and deadlines
- Infrequently used, since has similar behaviour to EDF, but more complex

Discussion

- EDF is optimal, and simpler to prove correct – why use RM?
  - RM more widely supported since easier to retro-fit to standard fixed priority scheduler, and support included in POSIX real-time APIs
  - RM more predictable: worst case execution time of a task occurs with worst case execution time of the component jobs – not always true for EDF, where speeding up one job can *increase* overall execution time (known as a “scheduling anomaly”)

Summary

- The rate monotonic algorithm
  - Simply periodic systems
  - Maximum utilisation test
- The earliest deadline first algorithm
  - Optimality
  - Maximum utilisation tests
- Other algorithms
  - Deadline monotonic
  - Least slack time