University of Glasgow

School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

Real-time Systems: Scheduling Periodic Tasks

Advanced Operating Systems

Lecture 15

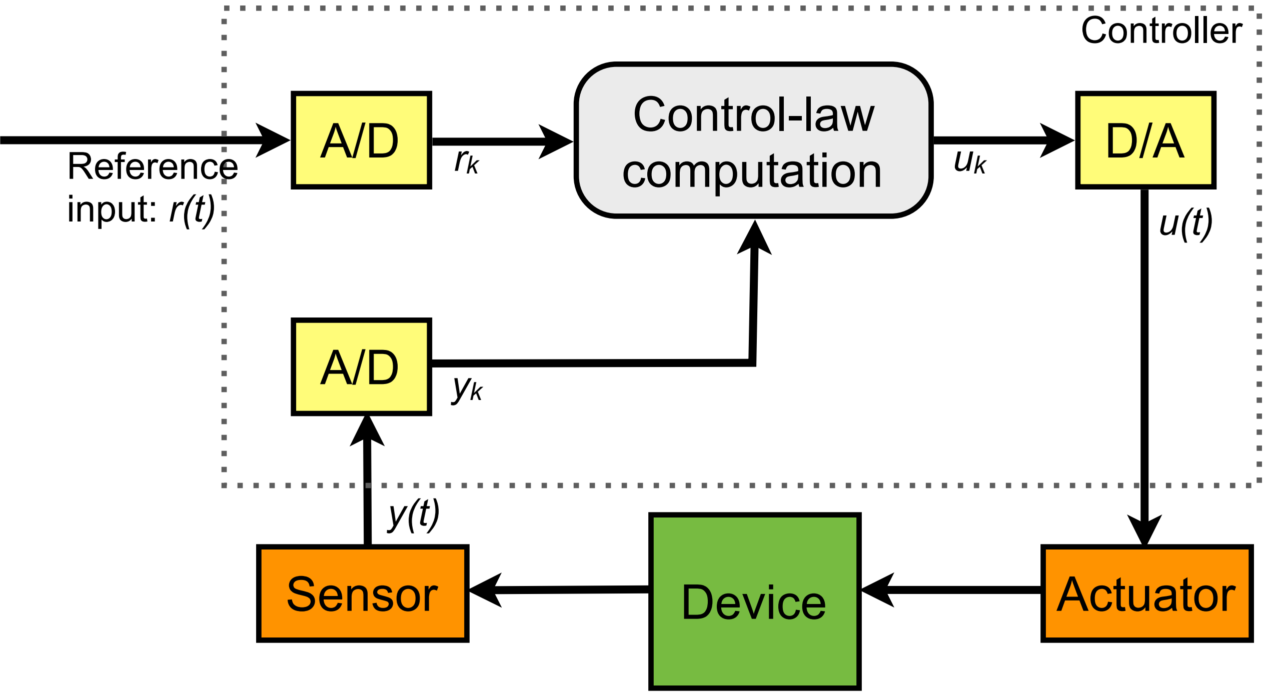
Lecture Outline

- System Model
- Scheduling periodic tasks
- The rate monotonic algorithm
  - Time-demand analysis
  - Maximum utilisation test
- The earliest deadline first algorithm
  - Maximum utilisation test
- Discussion

Introduction to Real-time Systems

- Real-time systems deliver services while meeting timing constraints
  - Not necessarily fast, but must meet some deadline
  - Many real-time systems embedded as part of a larger device or system: washing machine, photocopier, phone, car, aircraft, industrial plant, etc.
- Frequently require validation for correctness
  - Many embedded real-time systems are safety critical – if they don't work in a timely and correct basis, serious consequences result
  - Bugs in embedded real-time systems can be difficult or expensive to repair – e.g., can't easily update software in a car!

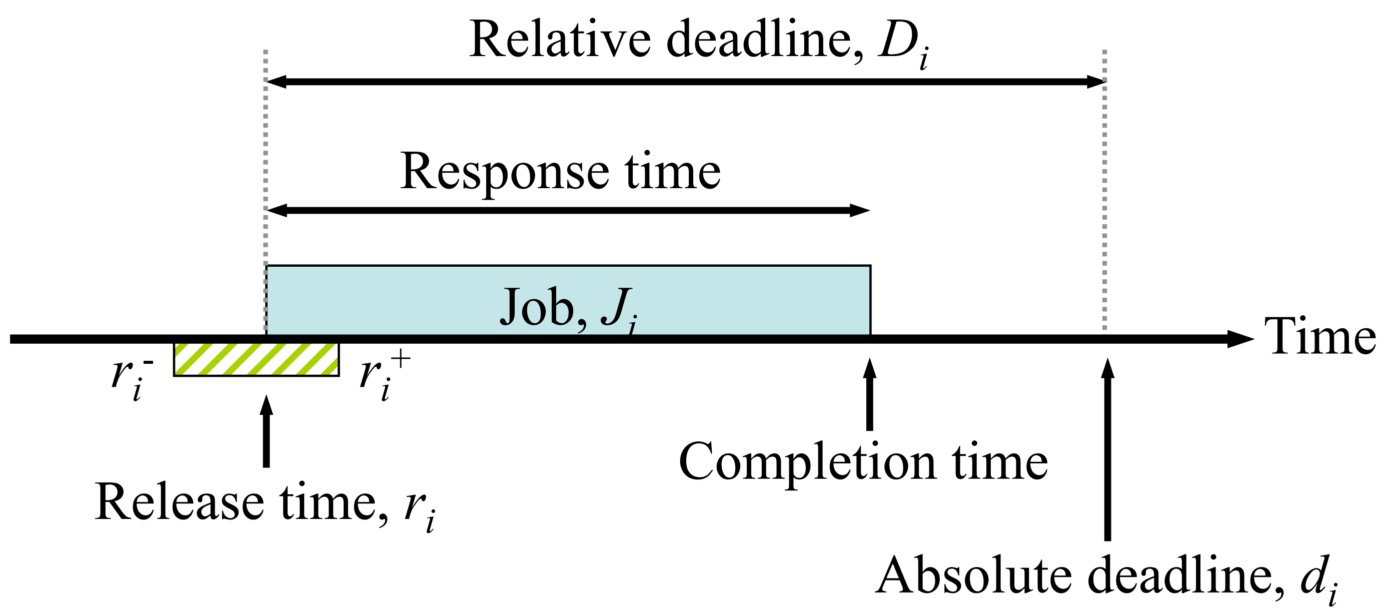
Typical System Model



- Control a device using actuator, based on sampled sensor data
  - Control loop compares measured value and reference
  - Depends on correct control law computation, reference input, accuracy of measurements
  - Time between measurements of $y(t)$, $r(t)$ is the sampling period, T
  - Small T better approximates analogue control but large T needs less processor time; if T is too large, oscillation will result as the system fails to keep up with changes in the input
- Simple control loop conceptually easy to implement
- Complexity comes from multiple control loops running at different rates, of if the system contains aperiodic components

Jobs, Tasks, Processors, and Resources

- Scheduling algorithms describe how jobs execute on a single processor
  - A job is a unit of work scheduled and executed by the system
  - Each job comprises a set of tasks $T = \{J_1, J_2, \ldots, J_n\}$
  - Tasks have execution time e_i and deadline:



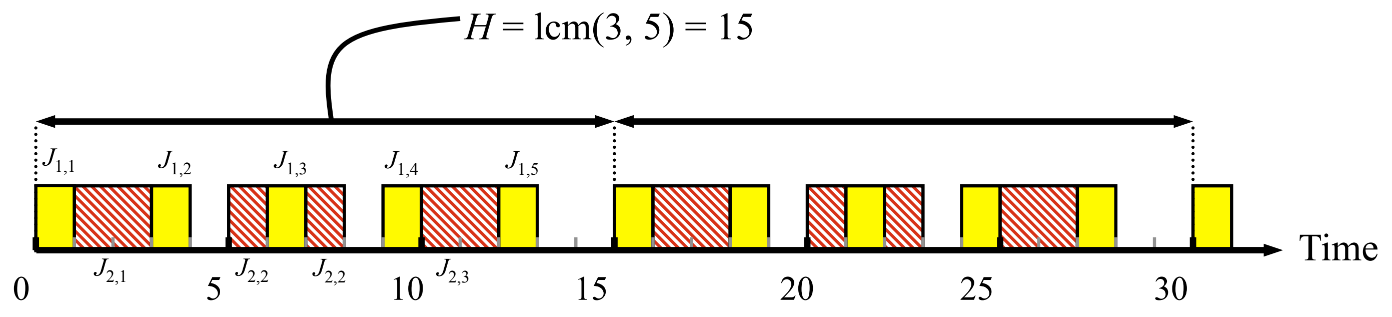
- Jobs can depend on resources
  - e.g., hardware devices
  - Resources have different types or sizes, but no speed, and are not consumed by use
- Jobs compete for resources, and can block if a resource is in use

Periodic Schedules

- If jobs occur on a regular cycle, task is periodic and characterised by parameters $T_i = (\phi_i, p_i, e_i, D_i)$
  - Phase, ϕ_i, of the task is the release time of the first job
  - Period, p_i, of the task is the time between release of consecutive jobs
  - Execution time, e_i, of the task is the maximum execution time of the jobs
    - Utilisation of a task is $u_i = e_i / p_i$
  - Relative deadline, D_i, is the minimum relative deadline of the jobs
- Hyper-period $H = \mathrm{lcm}(p_i)\ \forall\ p_i$

  – $T_1 : p_1 = 3, e_1 = 1$ 🟨
  – $T_2 : p_2 = 5, e_2 = 2$ ▨

Schedule repeats every hyper-period → if correct for one hyper-period, correct for all



Scheduling in Real-time Systems

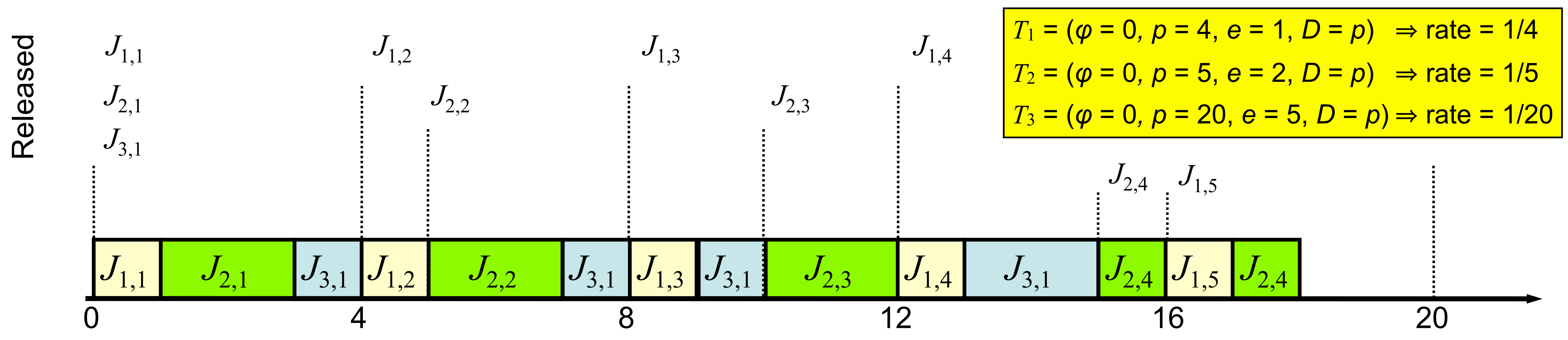
- How do we schedule systems of periodic tasks, so that all tasks provably meet their deadline?
- Two common algorithms:
    - Rate monotonic (RM)
    - Earliest deadline first (EDF)
- Trade-off optimality, stability, and ease of validation

Rate Monotonic Algorithm

- Assign priorities to jobs in each task based on period of the task
  - Shorter period → higher priority; rate (of job releases) is the inverse of the period, so jobs with higher rate have higher priority
  - Rationale: schedule jobs with most deadlines first, fit others around them
  - All jobs in a task have the same priority – fixed priority algorithm

- Three ways of proving correctness of schedule:
  - Exhaustive simulation for the hyper-period
  - Time demand analysis
  - Maximum utilisation

Rate Monotonic: Example



$T_1 = (\varphi = 0, p = 4, e = 1, D = p) \Rightarrow$ rate = 1/4
$T_2 = (\varphi = 0, p = 5, e = 2, D = p) \Rightarrow$ rate = 1/5
$T_3 = (\varphi = 0, p = 20, e = 5, D = p) \Rightarrow$ rate = 1/20

Time	Ready to run	Running
0	$J_{2,1}$ $J_{3,1}$	$J_{1,1}$
1	$J_{3,1}$	$J_{2,1}$
2	$J_{3,1}$	$J_{2,1}$
3		$J_{3,1}$
4	$J_{3,1}$	$J_{1,2}$
5	$J_{3,1}$	$J_{2,2}$
6	$J_{3,1}$	$J_{2,2}$
7		$J_{3,1}$
8	$J_{3,1}$	$J_{1,3}$
9		$J_{3,1}$

Time	Ready to run	Running
10	$J_{3,1}$	$J_{2,3}$
11	$J_{3,1}$	$J_{2,3}$
12	$J_{3,1}$	$J_{1,4}$
13		$J_{3,1}$
14		$J_{3,1}$
15		$J_{2,4}$
16	$J_{2,4}$	$J_{1,5}$
17		$J_{2,4}$
18		
19		

All tasks meet deadlines: proof by exhaustive simulation

Time Demand Analysis

- Exhaustive simulation error prone and tedious – an alternative is *time demand analysis*
  - Fixed priority algorithms predictable; do not suffer scheduling anomalies
    - The worst case execution time of the system occurs with the worst case execution time of the jobs, unlike dynamic priority algorithms which can exhibit anomalous behaviour
  - Basis of general proof that system can be scheduled to meet all deadlines
    - Find *critical instants* when system is most loaded, and has its worst response time
    - Use time demand analysis to check if system can be scheduled at those instants
    - In absence of scheduling anomalies, system will meet all deadlines if it can be scheduled at critical instants

Finding Critical Instants

- Critical instant of a job is the worst-case release time for that job, taking into account all jobs that have higher priority
  - i.e., job is released at the same instant as all jobs with higher priority are released, and must wait for all those jobs to complete before it executes
  - Response time, $w_{i,k}$, of a job released at a critical instant is the maximum possible response time of that job
  - Definition of a critical instant:

if $w_{i,k} \leq D_{i,k}$ for every $J_{i,k}$ in T_i then

 The job released at that instant has maximum response time of all jobs in T_i and $W_i = w_{i,k}$

else if $\exists\, J_{i,k} : w_{i,k} > D_{i,k}$ then

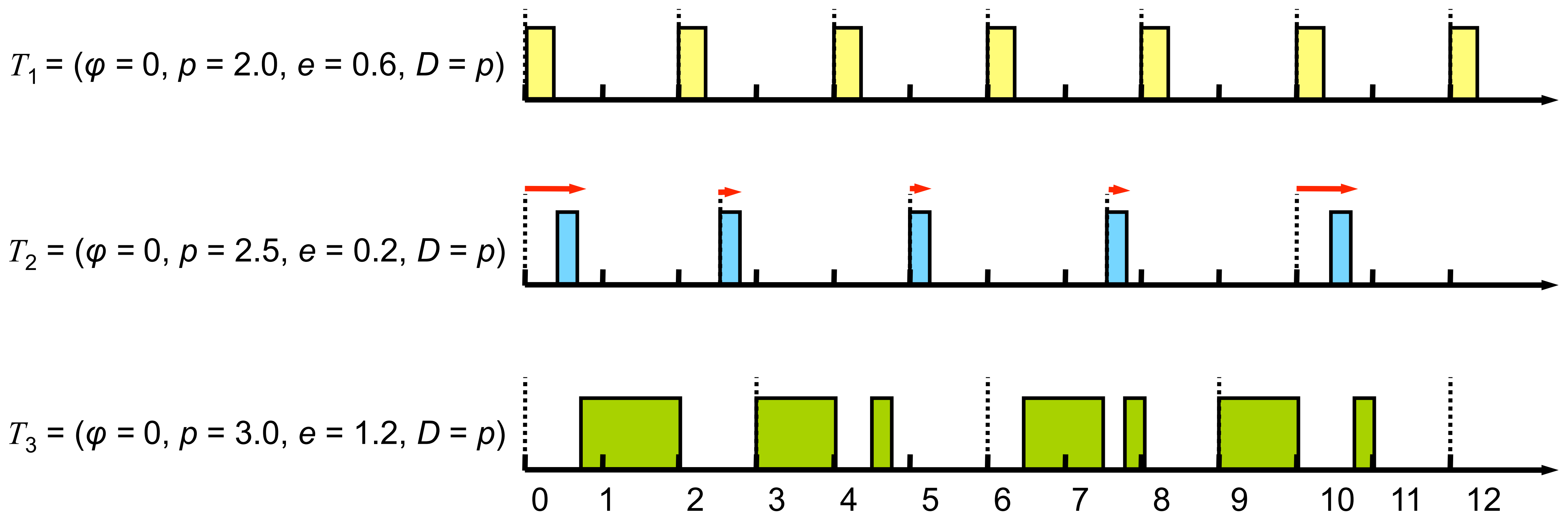
 The job released at that instant has response time $> D$

where $w_{i,k}$ is the response time of the job

All jobs meet deadlines, but this is when job with the slowest response is started

If some jobs don't meet deadlines, this is one of those jobs

Finding Critical Instants: Example



- 3 tasks scheduled using the rate-monotonic algorithm
- Response times of jobs in T_2 are: $r_{2,1} = 0.8$, $r_{2,2} = 0.2$, $r_{2,3} = 0.2$, $r_{2,4} = 0.2$, $r_{2,5} = 0.8$, …
- Therefore critical instants of T_2 are $t = 0$ and $t = 10$

Time-demand Analysis

- Simulate system behaviour at the critical instants
  - For each job $J_{i,c}$ released at a critical instant, if $J_{i,c}$ and all higher priority tasks complete executing before their relative deadlines the system can be scheduled
  - Compute the total demand for processor time by a job released at a critical instant of a task, and by all the higher-priority tasks, as a function of time from the critical instant; check if this demand can be met before the deadline of the job:
    - Consider one task, T_i, at a time, starting highest priority and working down to lowest priority
    - Focus on a job, J_i, in T_i, where the release time, t_0, of that job is a critical instant of T_i
    - At time $t_0 + t$ for $t \geq 0$, the processor time demand $w_i(t)$ for this job and all higher-priority jobs released in $[t_0, t]$ is: $w_i(t) = \underset{\text{Execution time of job } J_i}{e_i} + \underbrace{\sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k}_{\text{Execution time of higher priority jobs started during this interval}}$

$w_i(t)$ is the time-demand function

Using the Time-demand Function

- Compare time-demand function, $w_i(t)$, and available time, t:
  - If $w_i(t) \leq t$ at some $t \leq D_i$, the job, J_i, meets its deadline, $t_0 + D_i$
  - If $w_i(t) > t$ for all $0 < t \leq D_i$ then the task probably cannot complete by its deadline; and the system likely cannot be scheduled using a fixed priority algorithm
    - Note that this is a sufficient condition, but not a necessary condition. Simulation may show that the critical instant never occurs in practice, so the system could be feasible…
- Use this method to check that all tasks are can be scheduled if released at their critical instants; if so conclude the entire system can be scheduled

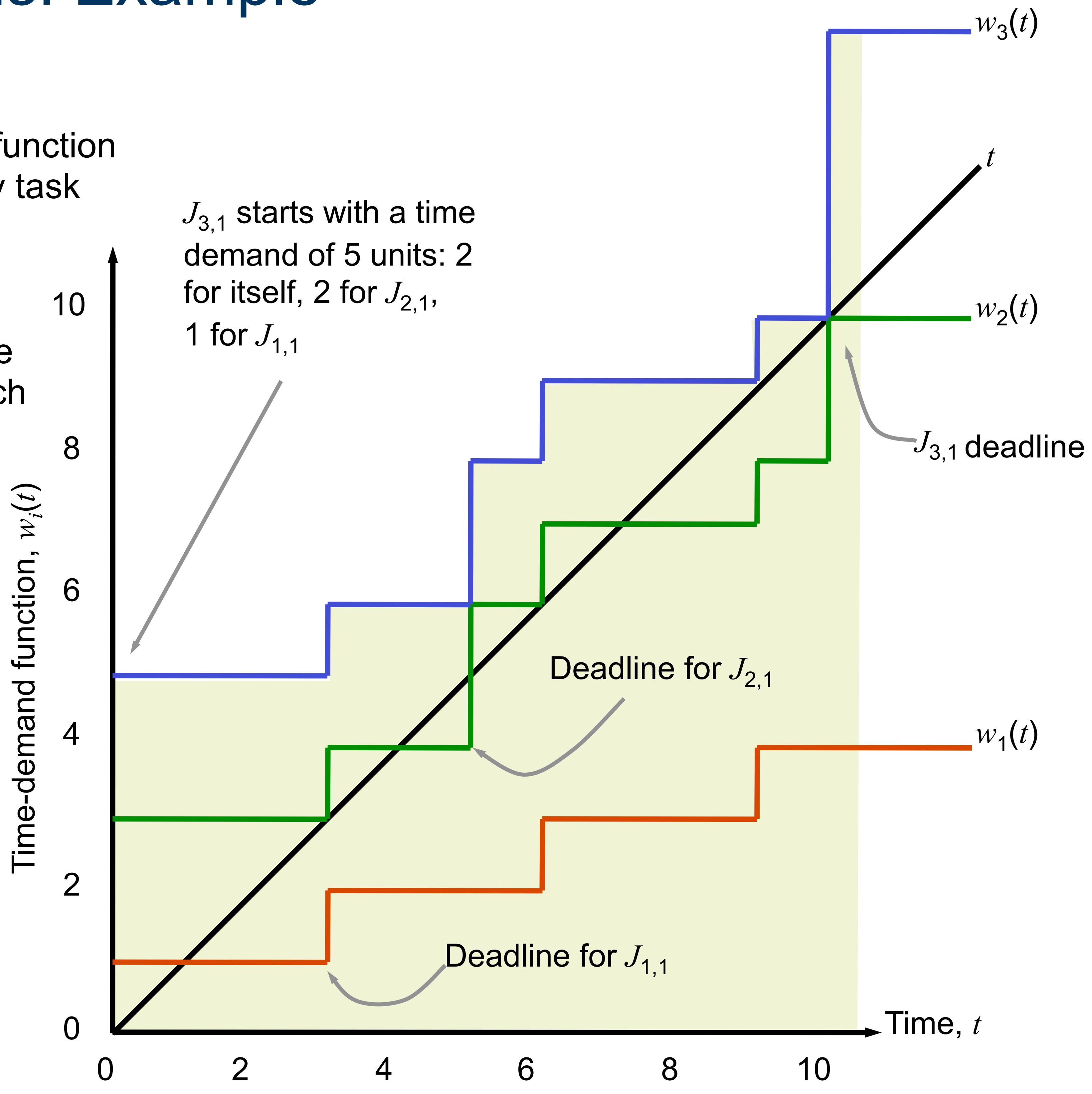
Time-demand Analysis: Example

The time-demand, $w_i(t)$, is a staircase function with steps at multiples of higher priority task periods

Plot time-demand versus available time graphically, to get intuition into approach

Example: a rate monotonic system $T_1 = (3, 1)$, $T_2 = (5, 2)$, $T_3 = (10, 2)$ with $\varphi = 0$, $D = p$ for all tasks

Time-demand functions $w_1(t)$, $w_2(t)$ and $w_3(t)$ are below t at deadlines, so the system can be scheduled – simulate the system to check this!



Time-demand Analysis

- Works for any fixed-priority scheduling algorithm with periodic tasks where $D_i < p_i$ for all tasks

- Only a sufficient test:
  - System can be scheduled if time demand less than time available before critical instants
  - But, might be possible to schedule if time demand exceeds available time – further validation (i.e., exhaustive simulation) needed in this case

Time-demand Analysis

- Time demand analysis is complex – useful for two reasons:
  - General mechanism for proof of correctness – more efficient for machine calculation when hyper-period is large
  - As the theory underlying simpler approaches based on maximum utilisation
    - For simply periodic rate monotonic systems
    - For general rate monotonic systems

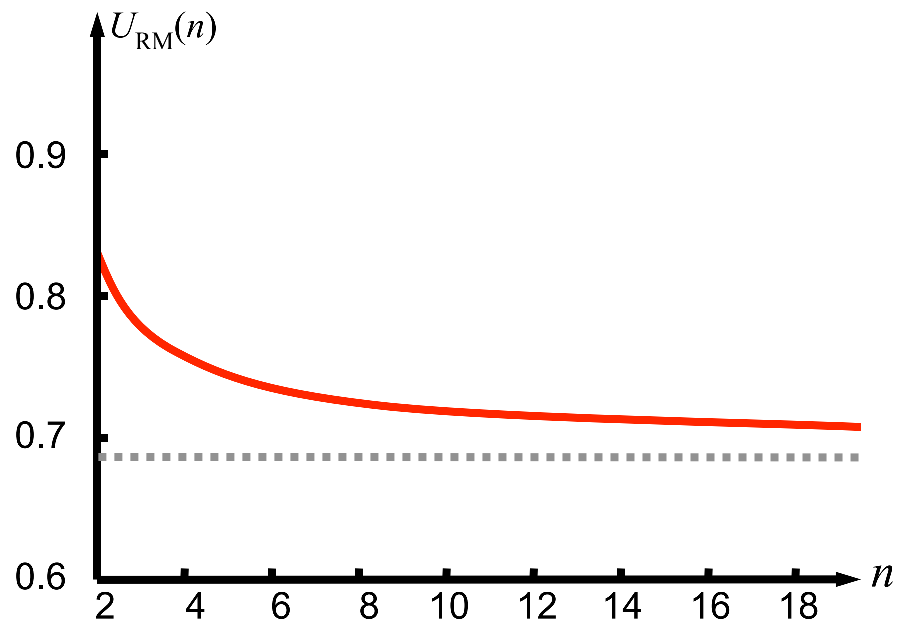
Simply Periodic Rate Monotonic Tasks

- Simply periodic systems: periods of all tasks are integer multiples of each other
- Rate monotonic optimal for simply periodic systems
- 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 $\Rightarrow D_i = p_i$
  - Simply periodic $\Rightarrow t$ integer multiple of periods of all higher priority tasks
  - Total time required to complete jobs with deadline $\leq t$ is $\sum_{k=1}^{i} \frac{e_k}{p_k} t = t \cdot U$ which only fails when $U > 1$
  - A simply periodic rate monotonic system can be scheduled if $U \leq 1$

RM Maximum Utilisation Test: $D_i = p_i$

- Maximum utilisation test can be derived for any RM system:
  - A system of n independent preemptable periodic tasks with $D_i = p_i$ can be scheduled on one processor using rate monotonic if $U \leq n \cdot (2^{1/n} - 1)$
    (Proof derives from time demand analysis)
  - $U_{\mathrm{RM}}(n) = n \cdot (2^{1/n} - 1)$

    For large n, $U_{\mathrm{RM}}(n) \to \ln 2$
    (i.e., $U_{\mathrm{RM}}(n) \to 0.6931\ldots$)



  - $U \leq U_{\mathrm{RM}}(n)$ is a *sufficient, but not necessary*, condition – an RM 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: $D_i \neq p_i$

- For n tasks, where $D_k = \upsilon \cdot p_k$ it can be shown that: $U_{RM}(n, v) = \begin{cases} v & \text{for } 0 \leq v \leq 0.5 \\ n((2v)^{\frac{1}{n}} - 1) + 1 - v & \text{for } 0.5 \leq v \leq 1 \\ v(n-1)[(\frac{v+1}{v})^{\frac{1}{n-1}} - 1] & \text{for } v = 2, 3, \ldots \end{cases}$

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

$\longleftarrow$ $D_i > p_i \Rightarrow$ Maximum utilisation increases

$\uparrow$ $D_i = p_i$

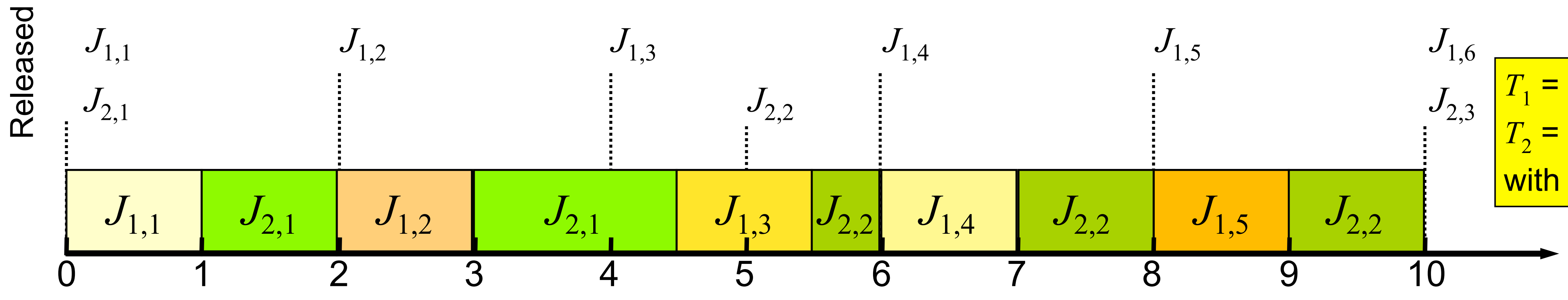
$\longrightarrow$ $D_i < p_i \Rightarrow$ Maximum utilisation decreases

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



T_1 = (p = 2, e = 1)
T_2 = (p = 5, e = 2.5)
with φ = 0, D = p for both

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

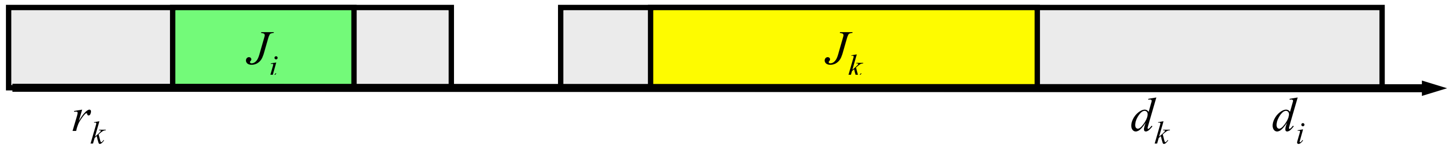
Time	Ready to run	Running
8	$J_{2,2}$	$J_{1,5}$
9		$J_{2,2}$
10	$J_{2,3}$	$J_{1,6}$
…	…	…

Earliest Deadline First is Optimal

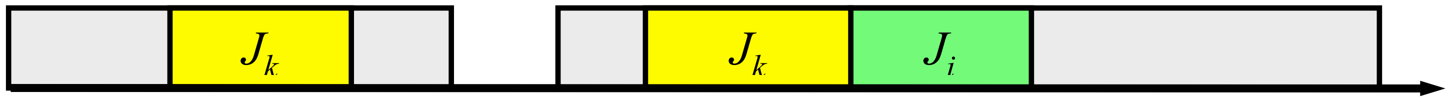
- EDF is optimal, provided the system has a single processor, preemption is allowed, and jobs don’t contend for resources
  - That is, it will find a feasible schedule *if one exists*, not that it will always be able to schedule a set of tasks
- EDF is not optimal with multiple processors, or if preemption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

Earliest Deadline First is Optimal: Proof

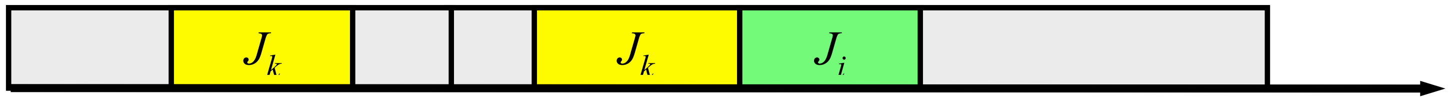
- Any feasible schedule can be transformed into an EDF schedule
  - If J_i is scheduled to run before J_k, but J_i's deadline is later than J_k's either:
    - The release time of Jk is after the J_i completes ⇒ they're already in EDF order
    - The release time of J_k is before the end of the interval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



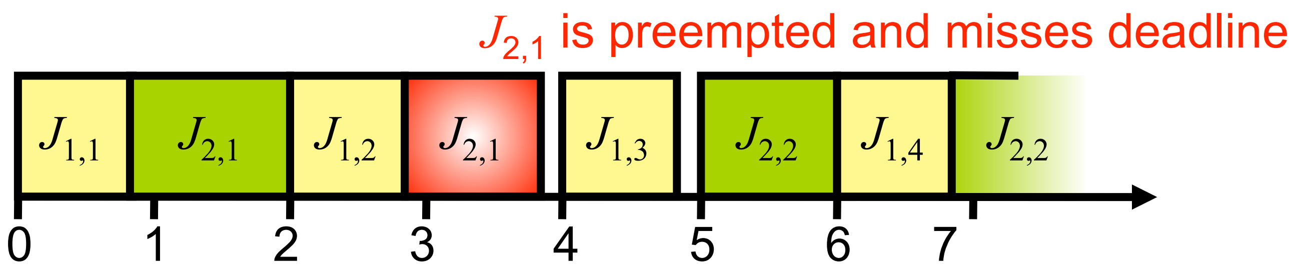
    - Move any jobs following idle periods forward into the idle period



    - 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

- When $D_i \geq p_i$ maximum utilisation = 1
  - Proof follows from optimality of the system
- When $D_i < p_i$ maximum utilisation test fails
  - Example: $T_1 = (2, 0.8)$, $T_2 = (5, 2.3, 3)$ where $\phi = 0, D = p$:



    U = 0.8/2 + 2.3/5 = 0.86 yet cannot be scheduled

  - Use density test: a system T of independent, preemptable periodic tasks can be feasibly scheduled on one processor using EDT if $\Delta \leq 1$ where $\Delta = \delta_1 + \delta_2 + \ldots + \delta_n$ and $\delta_i = e_i / \min(D_i, p_i)$
  - This is a sufficient condition, but not a necessary condition – i.e., 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)

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")