

Priority-driven Scheduling of Periodic Tasks (2)

Real-Time and Embedded Systems (M)

Lecture 6

UNIVERSITY
of
GLASGOW



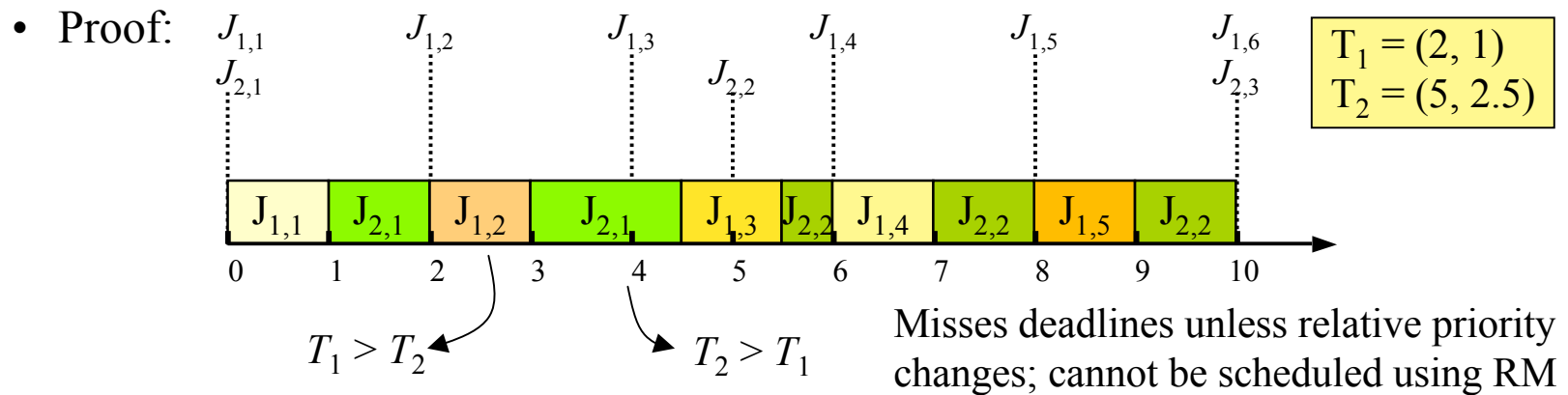
Lecture Outline

- Schedulability tests for fixed-priority systems
 - Conditions for optimality and schedulability
 - General schedulability tests and time demand analysis
- Practical factors
 - Non-preemptable regions
 - Self-suspension
 - Context switches
 - Limited priority levels

[Continues from material in lecture 5, with the same assumptions]

Optimality and Schedulability

- You will recall:
 - EDF and LST dynamic priority scheduling optimal:
 - Always produce a feasible schedule if one exists – on a single processor, as long as preemption is allowed and jobs do not contend for resources
 - Lecture 3 + confirmation last lecture: $U_{EDF} = 1$
 - Fixed priority algorithms non-optimal in general:
 - e.g. RM and DM sometimes fail to schedule tasks that can be scheduled using other algorithms



- Hence introduced schedulability tests in lecture 5

Optimality of RM and DM Algorithms

- Fixed priority algorithms *can be* optimal in restricted systems
- Example:
 - RM and DM are optimal in simply periodic systems
 - A system of periodic tasks is *simply periodic* if the period of each task is an integer multiple of the period of the other tasks:
$$p_k = n \cdot p_i$$
where $p_i < p_k$ and n is a positive integer; for all T_i and T_k
 - True for many real-world systems, e.g. the helicopter flight control system discussed in lecture 1

Optimality of RM and DM Algorithms

- Theorem: A system of *simply periodic*, independent, preemptable tasks with $D_i \geq p_i$ is schedulable on one processor using the RM algorithm if and only if $U \leq 1$
 - Corollary: The same is true for the DM algorithm
- Proof:
 - 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 $\sum_{k=1}^i \frac{e_k}{p_k} t = t \cdot U_i$
 - Only fails when $U_i > 1$

Schedulability of Fixed-Priority Tasks

- Identified several simple schedulability tests for fixed-priority scheduling:
 - A system of n independent preemptable periodic tasks with $D_i = p_i$ can be feasibly scheduled on one processor using RM iff $U \leq n \cdot (2^{1/n} - 1)$
 - A system of *simply periodic* independent preemptable tasks with $D_i \geq p_i$ is schedulable on one processor using the RM algorithm iff $U \leq 1$
 - [similar results for DM]
- But: there are algorithms and regions of operation where we don't have a schedulability test and must resort to exhaustive simulation
 - Is there a more general schedulability test?
 - Yes, extend the approach taken for simply periodic system schedulability

Fixed-Priority Tasks: Schedulability Test

- Fixed priority algorithms are predictable and do not suffer from *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

[See also lecture 3]

- Use this as the basis for a general schedulability test:
 - Find the *critical instant* when the system is most loaded, and has its worst response time
 - Use *time demand analysis* to determine if the system is schedulable at that instant
 - Prove that, if a fixed-priority system is schedulable at the critical instant, it is always schedulable

Finding the Critical Instant

- A critical instant for a job is the worst-case release time for that job, taking into account all jobs that have higher priority
 - i.e. a job 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
 - The response time of a job in T_i released at a critical instant is called the *maximum (possible) response time*, and is denoted by W_i
- The schedulability test involves checking each task in turn, to verify that it can be scheduled when started at a critical instant
 - If schedulable at all critical instants, will work at other times
 - More work than the test for maximum schedulable utilization, but less than an exhaustive simulation

Finding the Critical Instant

- A critical instant of a task T_i is a time instant such that:

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

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

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

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

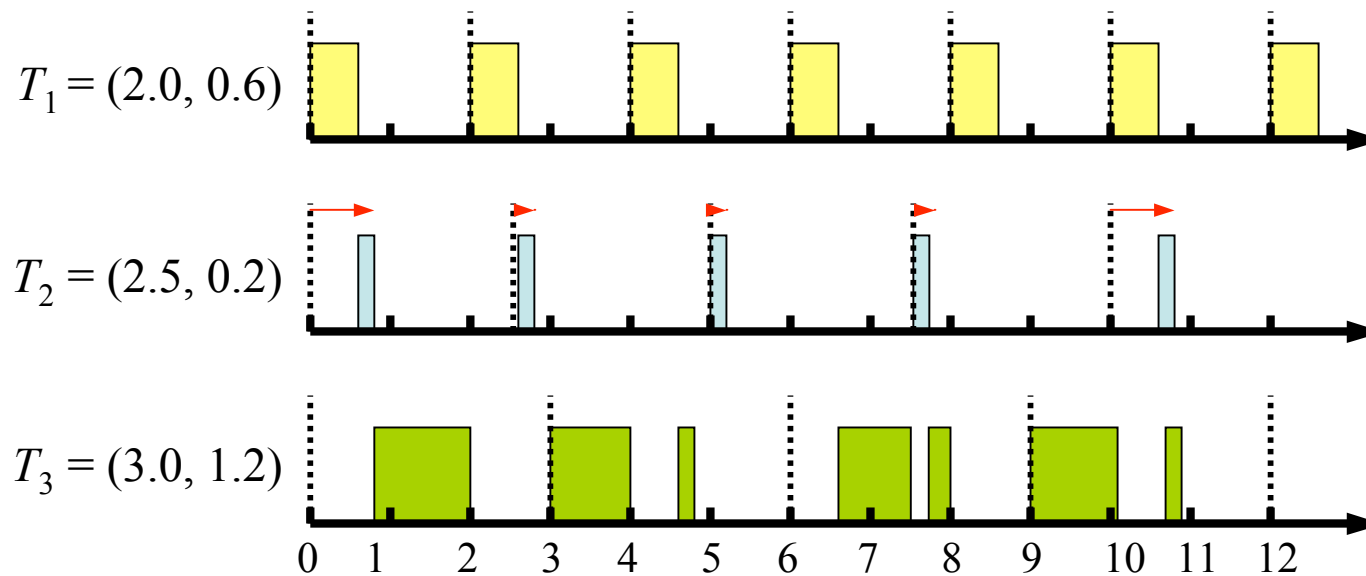
The job released at that instant has response time $> D$

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

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

- Theorem: In a fixed-priority system where every job completes before the next job in the same task is released, a critical instant occurs when one of its jobs $J_{i,c}$ is released at the same time with a job from every higher-priority task.
 - Intuitively obvious, but proved in the book

Finding the Critical Instant: Example



- 3 tasks scheduled using rate-monotonic
- Response times of jobs in T_2 are:
 $r_{2,1} = 0.8, r_{2,3} = 0.3, r_{2,5} = 0.2, r_{2,7} = 0.3, r_{2,9} = 0.8, \dots$
 Therefore critical instants of T_2 are $t = 0$ and $t = 10$

Using the Critical Instant

- Having determined the critical instants, show that for each job $J_{i,c}$ released at a critical instant, that job and all higher priority tasks complete executing before their relative deadlines
- If so, the entire system be schedulable...
- That is: don't simulate the entire system, simply show that it has correct characteristics following a critical instant
 - This process is called *time demand analysis*

Time-Demand Analysis

- 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:

$$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lfloor \frac{t}{p_k} \right\rfloor e_k \quad \text{for } 0 < t \leq p_i$$

Execution time
of job J_i

Execution time of higher priority
jobs started during this interval

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

Time-Demand Analysis

- Compare the time demand, $w_i(t)$, with the available time, t :
 - If $w_i(t) \leq t$ for 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 schedulable if released at their critical instants; if so conclude the entire system can be scheduled

Time-Demand Analysis: Example

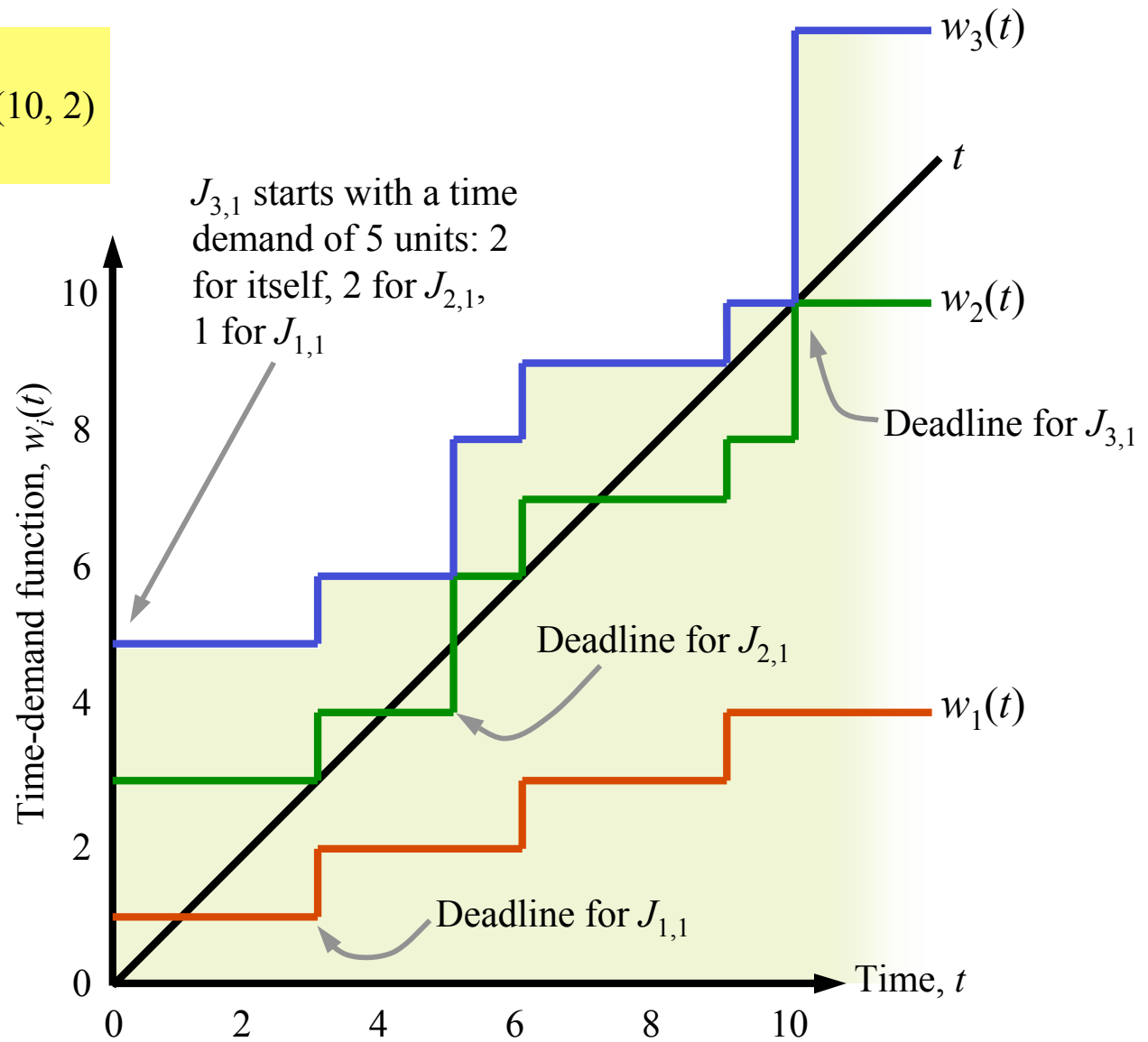
Rate Monotonic:

$$T_1 = (3, 1), T_2 = (5, 2), T_3 = (10, 2)$$

$$U = 0.933$$

The time-demand functions $w_1(t)$, $w_2(t)$ and $w_3(t)$ are not above t at their deadline \Rightarrow system can be scheduled

Exercise: simulate the system to check this!



Time-Demand Analysis

- The time-demand function $w_i(t)$ is a staircase function
 - Steps in the time-demand for a task occur at multiples of the period for higher-priority tasks
 - The value of $w_i(t) - t$ linearly decreases from a step until the next step
- If our interest is the schedulability of a task, it suffices to check if $w_i(t) \leq t$ at the time instants when a higher-priority job is released
- Our schedulability test becomes:
 - Compute $w_i(t)$
 - Check whether $w_i(t) \leq t$ is satisfied at *any* of the instants $t = j \cdot p_k$
where $k = 1, 2, \dots, i$
 $j = 1, 2, \dots, \lfloor \min(p_i, D_i)/p_k \rfloor$

Time-Demand Analysis: Summary

- Time-demand analysis schedulability test is more complex than the schedulable utilization test, but more general
 - Works for *any* fixed-priority scheduling algorithm, provided the tasks have short response time (i.e. $p_i < D_i$)
 - Can be extended to tasks with arbitrary deadlines (see book)
 - Only a sufficient test: guarantees that schedulable results are correct, but requires further testing to validate a result of not schedulable
- Alternative approach: simulate the behaviour of tasks released at the critical instants, up to the largest period of the tasks
 - Still involves simulation, but less complex than an exhaustive simulation of the system behaviour
 - *Worst-case simulation method*

Practical Factors

- We have assumed that:
 - Jobs are preemptable at any time
 - Jobs never suspend themselves
 - Each job has distinct priority
 - The scheduler is event driven and acts immediately
- These assumptions are often not valid... how does this affect the system?

Blocking and Priority Inversion

- A ready job is *blocked* when it is prevented from executing by a lower-priority job; a *priority inversion* is when a lower-priority job executes while a higher-priority job is blocked
- These occur because some jobs cannot be pre-empted:
 - Many reasons why a job may have non-preemptable sections
 - Critical section over a resource
 - Some system calls are non-preemptable
 - Disk scheduling
 - If a job becomes non-preemptable, priority inversions may occur, these may cause a higher priority task to miss its deadline
 - When attempting to determine if a task meets all of its deadlines, must consider not only all the tasks that have higher priorities, but also non-preemptable regions of lower-priority tasks
 - Add the blocking time in when calculating if a task is schedulable

Self-Suspension and Context Switches

- Self-suspension
 - A job may invoke an external operation (e.g. request an I/O operation), during which time it is suspended
 - This means the task is no longer strictly periodic... again need to take into account self-suspension time when calculating a schedule
- Context Switches
 - Assume maximum number of context switches K_i for a job in T_i is known; each takes t_{CS} time units
 - Compensate by setting execution time of each job, $e_{\text{actual}} = e + 2t_{CS}$ (more if jobs self-suspend, since additional context switches)

Tick Scheduling

- All of our previous discussion of priority-driven scheduling was driven by job release and job completion events
- Alternatively, can perform priority-driven scheduling at periodic events (timer interrupts) generated by a hardware clock
 - i.e. tick (or time-based) scheduling
- Additional factors to account for in schedulability analysis
 - The fact that a job is ready to execute will not be noticed and acted upon until the next clock interrupt; this will delay the completion of the job
 - A ready job that is yet to be noticed by the scheduler must be held somewhere other than the ready job queue, the *pending job* queue
 - When the scheduler executes, it moves jobs in the pending queue to the ready queue according to their priorities; once in ready queue, the jobs execute in priority order

Practical Factors

- Clear that non-ideal behaviour can affect the schedulability of a system
- Have touched on how – more details later in the module

Summary

- Have discussed fixed-priority scheduling of periodic tasks:
 - Optimality of RM and DM
 - More general schedulability tests and time-demand analysis
- Outlined practical factors that affect real-world periodic systems
- Tutorial tomorrow will recap the material from lectures 5 and 6
- Problem set 2 now available: due at 2:00pm on 28th January