CIS 480/899 – Embedded and Cyber Physical Systems

Spring 2009

Introduction to Real-Time Scheduling

Insup Lee
Department of Computer and Information Science
University of Pennsylvania
lee@cis.upenn.edu
www.cis.upenn.edu/~lee

Examples of real-time applications

- On-line transaction systems and interaction systems
- Real-time monitoring systems
- Signal processing systems
  - typical computations
  - timing requirements
  - typical architectures
- Control systems
  - computational and timing requirements of direct computer control
  - hierarchical structure
  - intelligent control
- Embedded systems
  - Resource limitations

Hard and Soft Real-Time Tasks

Task: a unit of work - a granule of computation, a unit of transmission

Hard real-time tasks - critical tasks, deterministic tasks:

- Failure of a deterministic task to complete in time ⇒ a fatal error

Soft real-time tasks:

- Essential tasks, statistical tasks: such a task is completed always even when it requires more than available time.
  Examples: display updates, interpretation of operator commands
- Non-essential tasks: such a task may be aborted if it cannot be completed in time.
  Examples: connection establishment, monitoring non-critical changes

Soft vs. hard real-time systems

- hard real-time systems are typical embedded systems.

Basic types of software systems

In order of increasing difficult in scheduling to meet deadlines

- Type 1: purely cyclic, no asynchronous events or variations in computing requirements.
  - Examples: simple control systems such as those for small missiles and target drones
- Type 2: mostly cyclic, some asynchronous events or variations in computing requirements, such as fault recovery, external commands, mode changes.
  - Examples include modern avionics, process control, etc.
- Type 3: asynchronous, or event driven, cyclic processing does not dominate, with known variations in computing requirements.
  - Examples include communication, radar tracking, etc.
- Type 4: same as type 3 but variations in computing requirements unpredictable.

Deadlines in type 3 and type 4 systems are typically soft.

Desired characteristics of RTOS

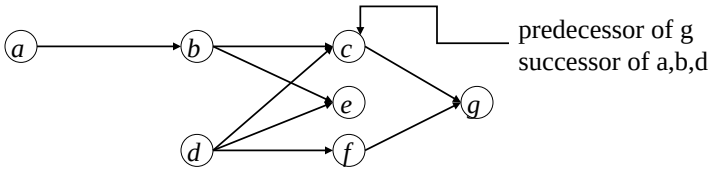
- **Predictability**, not speed, fairness, etc.
  - Under normal load, all deterministic (hard deadline) tasks meet their timing constraints
  - Under overload conditions, failures in meeting timing constrains occur in a predictable manner.

  ⇒ Services and context switching take bounded times
- **Application- directed resource management**
  - scheduling mechanisms allow different policies
  - resolution of resource contention can be under explicit direction of the application.

Tasks and Task Systems

- Task: a unit of work, a granule of computation or communication, with the following parameters:
  - release time
  - deadline
  - processing time
  - resource requirements
- A task system: a set of tasks, which are to be executed on a processor (or processors) and share resources.
  - independent tasks
  - dependent tasks (complex tasks) which precedence (constraint) graph describing their dependencies

Task Systems



How to model conditional branches, or dependencies, resource constrains, etc?

Tasks

Task, *T*: "A unit of work" with the following parameters:

- release time or ready time, r : T can be scheduled (and executed) after, but not before r.
- deadline, d : T must be completed before d.
- processing time, e: amount of processor time required to complete T, if T is executed alone.

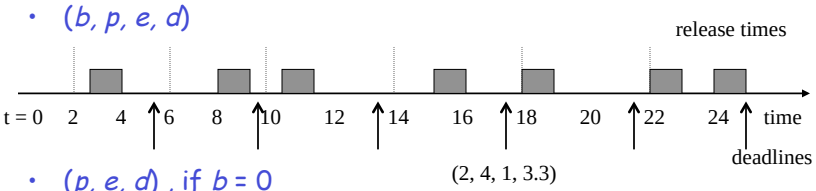
***T* is periodic**: T is a periodic sequence of requests for the same executions

- b = release time of the first request, release times of 2nd, 3rd, 4th,… requests are $b+p$, $b+2p$, $b+3p$ … respectively
- p = period of T
- τ = time to complete each request
  = processing time of T
- d = the time interval between its release time and the instant by which each request must be completed
  = deadline of T.

a request ≡ a subtask, task

Notations for periodic tasks

- (b, p, e, d)



- (p, e, d) , if $b = 0$
  (14, 5, 20), (7, 1, 7), (10, 3, 6)
- (p, e) , if $d = p$
  (15, 2), (10, 1), (3, 1)

A task system $\equiv \{T_1, T_2, \ldots, T_n\}$, a set of tasks

This model is appropriate only for single resource scheduling, typically used in processor scheduling.

Some jargons: schedule T in (a, b); T is scheduled in $(a,$ b$)$; the processor is assigned to T in (a, b)

Paradigms for scheduling
Cyclic tasks (periodic tasks)

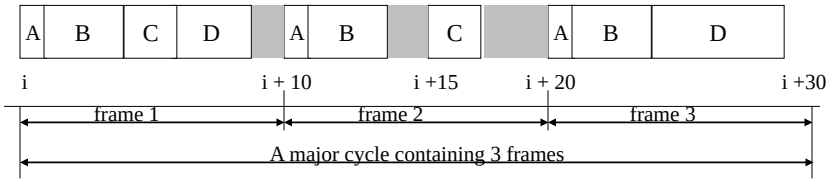
- Cyclic executive
- Weighted round robin
- Priority- driven scheduling of
  - independent periodic tasks
  - independent periodic tasks + aperiodic tasks (sporadic tasks)
  - periodic tasks that share resources

Cyclic Executives

A cyclic executive is a program that deterministically interleaves the execution of periodic tasks on a single processor. The order in which the tasks execute is defined by a cyclic schedule.

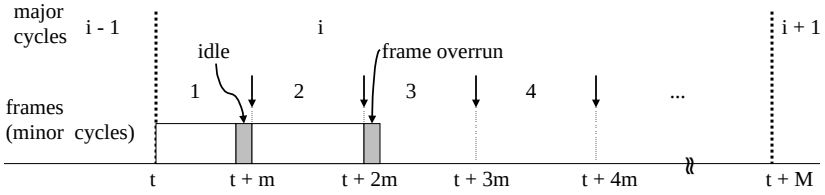
Example:

$A = (10, 1), B = (10, 3), C = (15, 2), D = (30,8) \Rightarrow (30, 3)\ (30, 5)$



Reference: "The Cyclic executive model and ADA" Proc. of RTSS, 1988, by Baker and Shaw

General structure of a major schedule



correct timing enforced at the end of each frame

Rules governing the choice of m for a given $\{(p_i, e_i, d_i)\}$ of n tasks

- $m \le d_i$, $i = 1, 2, \ldots, n$
- $m \ge e_i$, $i = 1, 2, \ldots, n$
- M/n = integer (m divides p_i for at least one i)
- there must be at least one frame in the interval between the release time and deadline of every request.
  $2m - gcd(m, p_i) \le d_i$, for $i = 1, 2, \ldots, n$

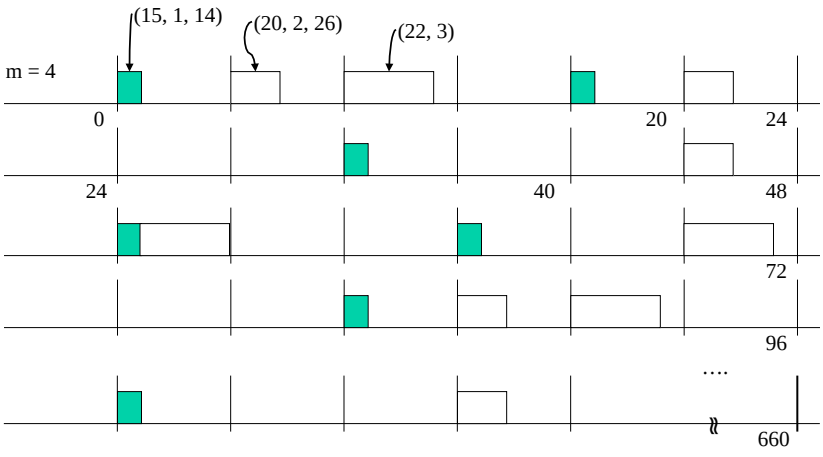
An Example: choices of *m*

T = {(15, 1, 14), (20, 2, 26), (22, 3)}

Possible values of *m*

- $m \le min\{d_i\} = 14 \Rightarrow m = 1, 2, 3, 4, \ldots, 14$
- $m \ge max\{e_i\} = 3 \Rightarrow m = 3, 4, 5, \ldots, 14$
- *m* divides one of $p_i \Rightarrow m = 3, 4, 5, 10, 11$
- $2m - gcd(m, p_i) \le d_i \Rightarrow m = 3, 4, 5$

An Example: choices of *m* (cont)



Example Cont'd

- Suppose that **T** = {(15, 1, 14), (20, 7, 26), (22, 5)}
  Rules 1 $m \le min\{d_i\} = 4$
  2 $m \ge max\{e_i\} = 7$
  cannot be satisfied simultaneously.
  We decompose the longer tasks into shorter ones
  T' = {(15, 1, 14), (20, 3, 26), (20, 4, 26), (22, 2), (22, 3)}
  for scheduling on one processor

Advantages of cyclic executive

- Simplicity and predictability:
  - timing constraints can be easily checked
  - the cyclic schedule can be represented by a table that is interpreted by the executive
  - context switching overhead is small
  - it is easy to construct schedules that satisfy precedence constraints & resource constraints without deadlock and unpredictable delay

Disadvantages

- Given major and frame times, structuring the tasks with parameters *pi*, *ei*, and *di* to meet all deadlines is NP-hard for one processor
- Splitting tasks into subtasks and determining the scheduling blocks of each task is time consuming
- Error in timing estimates may cause frame overrun:
  How to handle frame overrun? It is application dependent:
  - suspense or terminate the overrun task, and execute the schedule of the next frame
  - complete the suspended task as background later
  - complete the frame, defer the start of the next frame
  - log overruns. If too many overruns, do fault recovery

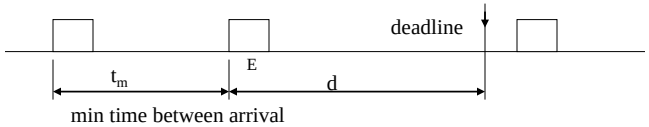
Handling mode change is difficult.

Mode change: deletion and addition of tasks or change the parameters of existing tasks

When to do mode change? Pros and cons of doing it at the end of current frame, current major cycle, execution of the current task, upon interrupt immediately

Handling sporadic tasks

- convert each sporadic task into a periodic one: periodic server (*p*, *e*, *d*)



min time between arrival

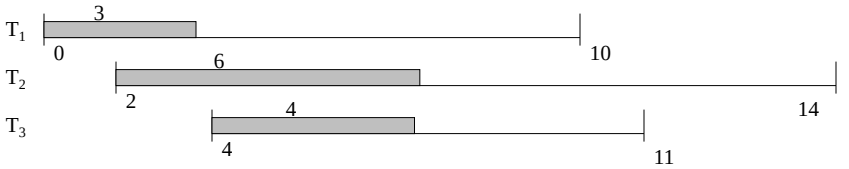
$p = \min (t_m, d - e + 1)$ - too pessimistic - guarantees worst case performance by giving max time to the task

- set aside time in minor cycles for execution of sporadic tasks
  - does not guarantee worst case

Priority-driven algorithms

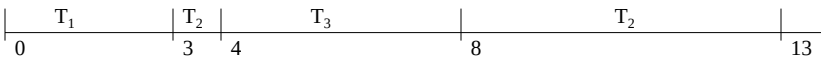
- A class of algorithms that never leave the processor(s) idle intentionally
- Also known as greedy algorithms and list algorithms
- Can be implemented as follows: (preemptive)
  - Assign priorities to tasks
  - Scheduling decisions are made
    - when any task becomes ready,
    - when a processor becomes idle,
    - when the priorities of tasks change
  - At each scheduling decision time, the ready task with the highest priority is executed
- If non-preemptive, scheduling decisions are made only when a processor becomes idle.
- The algorithm is static if priorities are assigned to tasks once for all time, and is dynamic if they change. Static if fixed.

Example



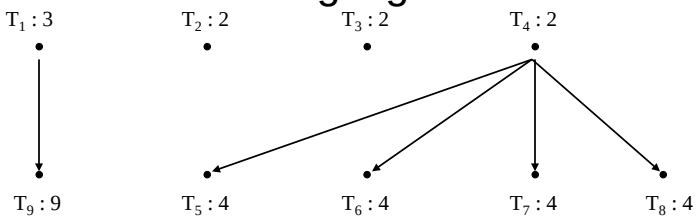
non preemptive EDF, FIFO

preemptive EDF

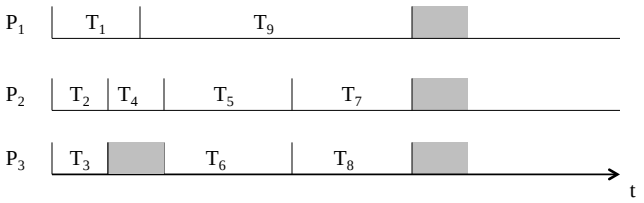


non preemptive, not priority-driven

Unexpected behavior of priority-driven scheduling algorithm

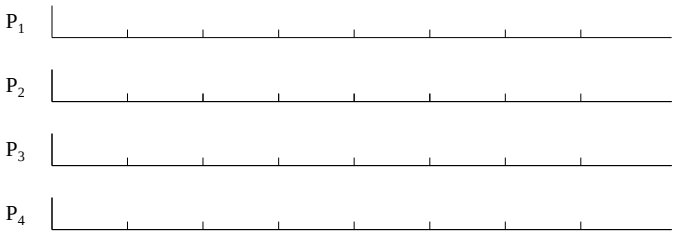


$L = (T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9)$



Unexpected behavior of priority-driven scheduling algorithm (cont.)

- Suppose that we have 4 processors:



- Suppose that execution times are

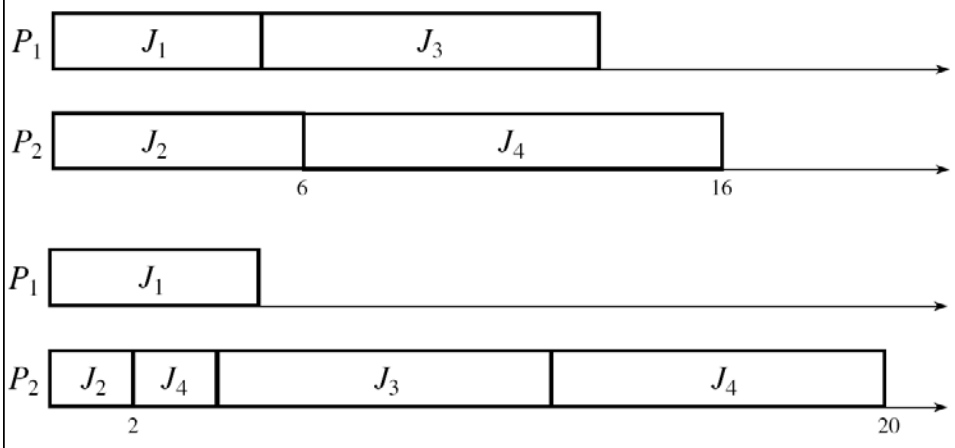
  2, 1, 1, 1, 3, 3, 3, 3, 8
- Suppose that $T_4 < T_5$ and $T_4 < T_6$ are removed

Anomalies of Priority-Driven Systems

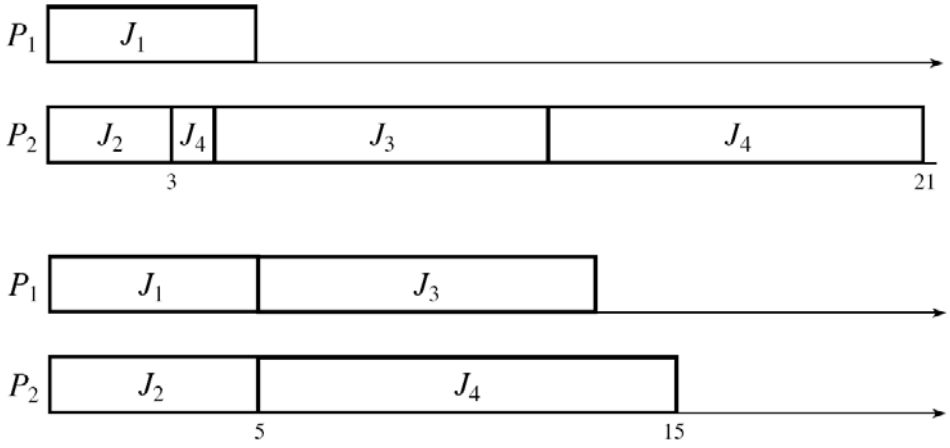
- Given J1,J2,J3,J4
- Priority:
  - J1 > J2 > J3 > J4
- Preemption but no job migration
- Two processors P1 and P2

	r_i	d_i	$[e_i^-, e_i^+]$
J_1	0	10	5
J_2	0	10	[2, 6]
J_3	4	15	8
J_4	0	20	10

Cases: e2 = 6 and e2 = 2



Cases: e2 = 3 and e2 = 5



Validation Problem

- Scheduling anomalies make validation hard
- Can anomalies exist even on single processor systems?

Schedulability Analysis of Periodic Tasks

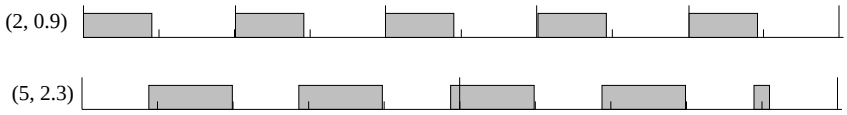
- Main problem:
  - Given a set of periodic tasks, can they meet their deadlines?
  - Depends on scheduling policy
- Solution approaches
  - Utilization bounds (Simplest)
  - Exact analysis (NP-Hard)
  - Heuristics
- Two most important scheduling policies
  - Earliest deadline first (Dynamic)
  - Rate monotonic (Static)

Utilization Bounds

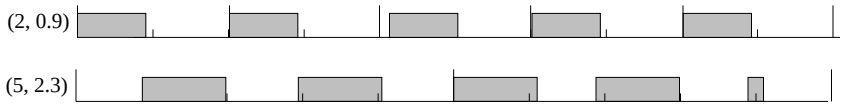
- Intuitively:
  - The lower the processor utilization, U, the easier it is to meet deadlines.
  - The higher the processor utilization, U, the more difficult it is to meet deadlines.
- Question: is there a threshold U_{bound} such that
  - When $U < U_{bound}$ deadlines are met
  - When $U > U_{bound}$ deadlines are missed

An example: schedule (2, 0.9) (5, 2.3) on one processor

- Rate monotone, shortest period first



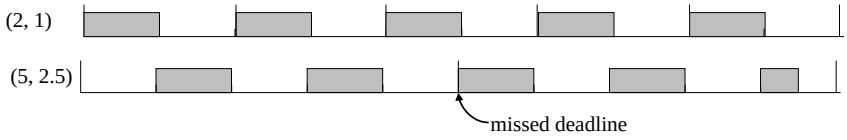
- Earliest deadline first



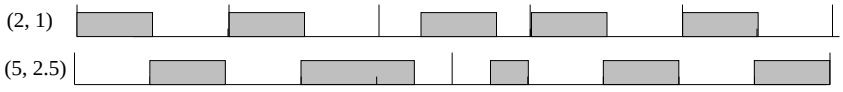
Static priority assignment Vs
Dynamic priority assignment

Schedule (2,1) (5, 2.5)

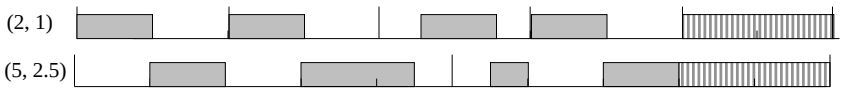
- Rate-monotone



- Earliest deadline first



- Shortest slack time first



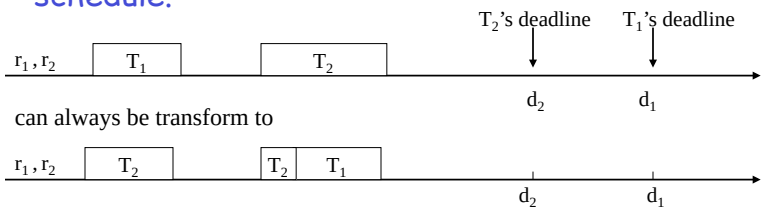
EDF (Earliest Deadline First)

Optimality of EDF

Optimality of the earliest deadline first algorithm for preemptive scheduling on one processor:

feasible schedule = one in which all release time and deadline constraints are met

Given a task system **T**, if the EDF algorithm fails to find a feasible schedule, then **T** has no feasible schedule.

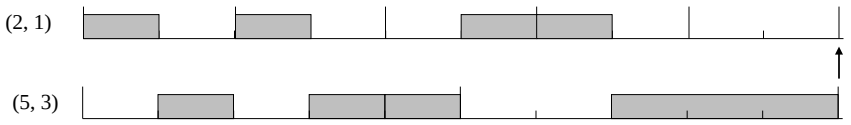


Schedulable Utilization

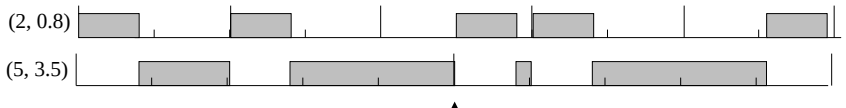
- Utilization of a periodic task (p, τ, d)
  $u = \tau / p \equiv$ the fraction of time the task keeps the processor busy
  U, total utilization of task system
  $\mathbf{T} = \{(p_i, \tau_i, d_i)\}$ contains n tasks
  $$U = \sum_{i=1}^{n} \tau_i / p_i \equiv \text{demand for processor time}$$
- A system of n tasks with $d_i = p_i$ can be feasibly scheduled if and only if $U \leq 1$
  - If $U > 1$, the total demand of processor in the time interval $(0, p_1, p_2 \ldots p_n)$ is $p_2 p_3 \ldots p_n \tau_1 + p_1 p_3 \ldots p_n \tau_2 + \ldots p_1 p_2 \ldots p_{n-1} \tau_n > p_1 p_2 \ldots p_n$
    clearly, no feasible schedule exists.
  - If $U \leq 1$, the EDF algorithm can always find a feasible schedule. To show this statement is true, we suppose that the EDF algorithm fails to find a feasible schedule. And, then show $U > 1$, which is a contradiction to $U \leq 1$

Behavior of earliest deadline algorithm

- Schedule (2, 1) (5, 3) with $U = 1.1$



- Schedule (2, 0.8) (5, 3.5) with $U = 1.1$



Which deadline will be missed
as U increases cannot be predicted

RM (Rate Monotonic)

Fixed (static) Priority Scheduling

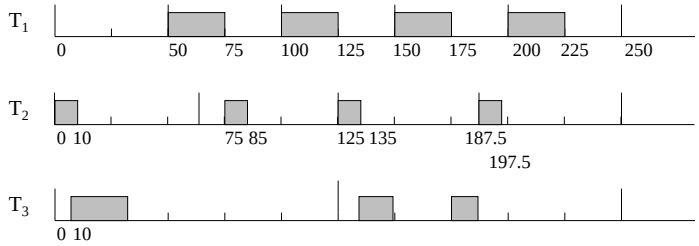
T_1 = (50, 50, 25, 100) u_1 = 0.5

T_2 = (0, 62.5, 10, 50) u_2 = 0.16

T_3 = (0, 125.0, 25, 75) u_3 = 0.2

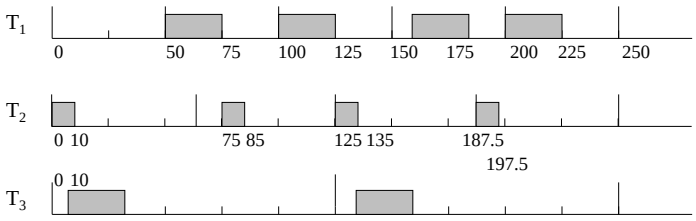
U = 0.86

Rate- monotone schedule (L = T_1, T_2, T_3)

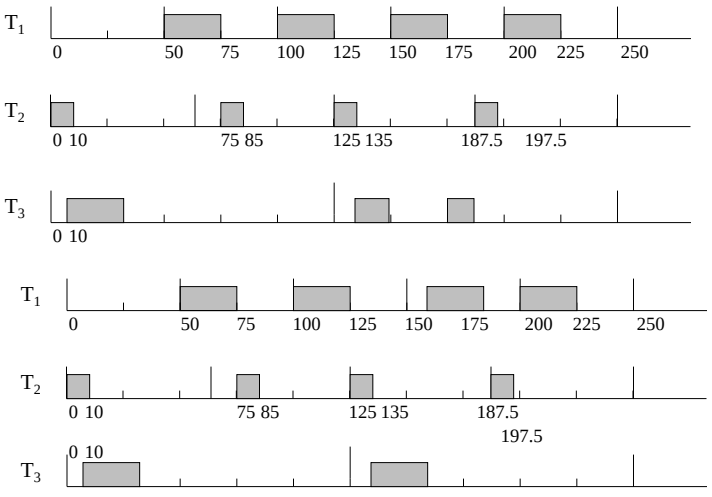


Fixed (static) Priority Scheduling (cont.)

Deadline-monotone schedule ($L = T_2, T_3, T_1$)



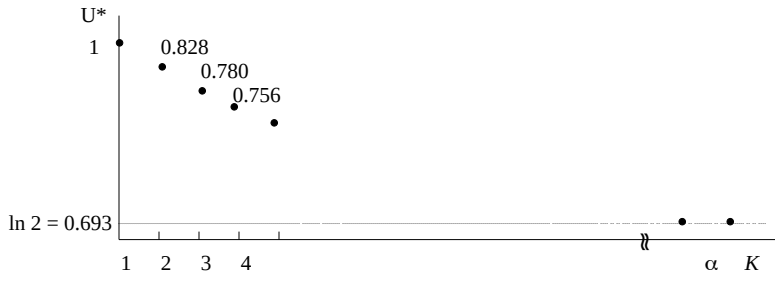
Rate- monotone = deadline-monotone when $d_i = p_i$ for all i



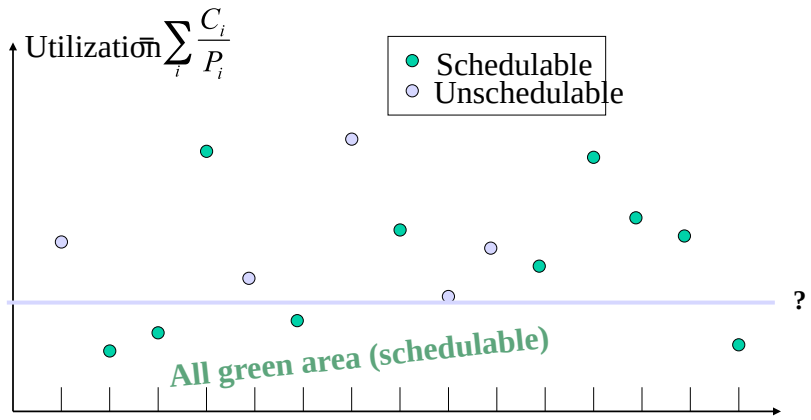
Advantages of Rate-Monotone (and deadline-monotone) algorithm

- Priorities being fixed, the algorithms are easy to implement.
- The resultant schedules have predictable behavior:
  - Lower priority tasks are executed as background of higher priority tasks!!!
  - There are known sufficient conditions which can be used to determine whether a task system is schedulable.
    - o A system of K tasks with $d_i = p_i$ and total utilization U is schedulable by the rate-monotone algorithm if

$$U \le \hat{U} = K(2^{1/K} - 1)$$



A Conceptual View of Schedulability



- Modified Question: is there a threshold U_{bound} such that
  - When $U < U_{bound}$ deadlines are met
  - When $U > U_{bound}$ deadlines may or may not be missed

Advantages (cont.)

- $\hat{U}$ is called the worst case schedulability bound
  - In general, a task system with $d_i = \Delta p_i$ for all i, is schedulable if

$$U \le \hat{U}(\Delta) = \begin{cases} K((2\Delta)^{1/K} - 1) + 1 - \Delta, & 1/2 \le \Delta \le 1 \\ \Delta & 0 \le \Delta \le 1/2 \end{cases}$$

  - Task systems with total utilization larger than the worst-case schedulability bound are often schedulable by rate-monotone algorithm. Average schedulability bound is closer to 1.
  - To check whether all requests in T_1, T_2 … T_n meet there deadlines, where $p_1 < p_2 < \ldots < p_n$, examine a rate-monotone schedule with $b_1 = b_2 = \ldots = b_n = 0$; all deadlines in T_n are met if the first request of T_n meets its deadline
  - if $p_i = kp_j$ for some integer k, for all $i, j = 1, 2, \ldots, n$ the task system containing such n tasks is schedulable by the rate-monotone algorithm if and only if $U \le 1$.

Critical Instants

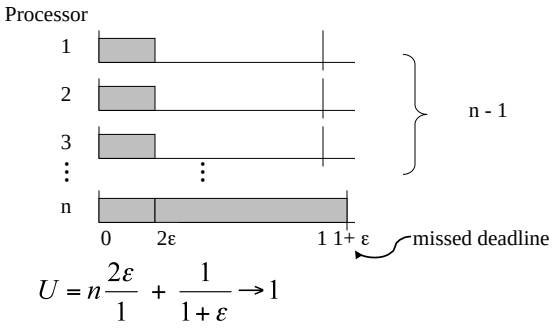
- A critical instant of task T is a time instant such that the job in T released at the instant has the maximum response time of all jobs in T

Harmonic Periods

- A task system is said to be in phase if the time of the first requests are all equal to
  zero, i.e. , $b_1 = b_2 = \ldots = b_k = 0$
  Otherwise, it is said to have arbitrary phase
- The system is simply periodic (or harmonic) if
  $p_i = kp_j$ for some integer k
  or
  $p_i = k_{ij}p_j$ for integers k_{ij}
  for all i and j.

An Example

Given (1, 2ε), (1, 2ε), … , (1, 2ε), (1+ε, 1), schedule them on *n* processors using the rate-monotone algorithm



$$U = n\frac{2\varepsilon}{1} + \frac{1}{1+\varepsilon} \to 1$$

Unacceptable performance

Solution: assign tasks to processors, and schedule tasks on each processor independently of tasks on other processors.

Practical Questions

- When to do mode change: mode change protocol. (by Sha, Rajkumar, Lehoczky and Ramamritham, Journal of RTS, vol 1, 1989)
- What is the context switching overhead and its effect on schedulability?
- What is the effect of variations in processing time?
- What is the effect of limited priority levels?
- How to schedule aperiodic tasks together with periodic ones?
- How to handle synchronization of tasks?

Priority Inversion Problem: An Anomaly of Priority-Driven systems

Modified Workload Model

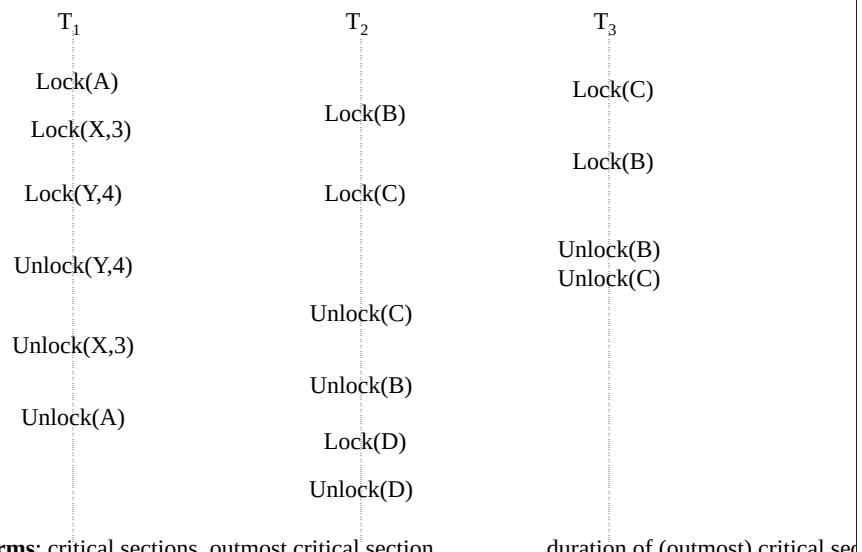
- The system contains
  - processor (s)
  - m types of serially reusable resource, $R_1, R_2, \ldots, R_m$. There are N_i units of resource (resources) of type R_i.
- An execution of a task requires
  - a processor for τ units of time (processing time)
  - some resources for exclusive use
- Every resource can be allocated to one task at a time and, once allocated, is held by the task until it is no longer needed.
- Examples of resources are semaphores, read/write locks, servers.
  - A resource that has one unit but can be shared by many tasks is modeled as a type of resource with many units.

Modified Workload Model (cont.)

- To use a resource of type R_i, a task executes $Lock(R_i)$.
  - To use k resources it executes $Lock(R_i,k)$
  - In examples, we use $L(R_i)$ or $L(R_i,k)$
- After a resource is allocated to a task, the task holds the resource until the task executes an $Unlock(R_i)$ (or in the case of multi-units, $Unlock(R_i,k)$) to release it.
  - In examples, we use $U(R_i)$ or $U(R_i,k)$
- If the OS takes back a resource allocated to a task before the task releases the resource, some work done by the task is undone.

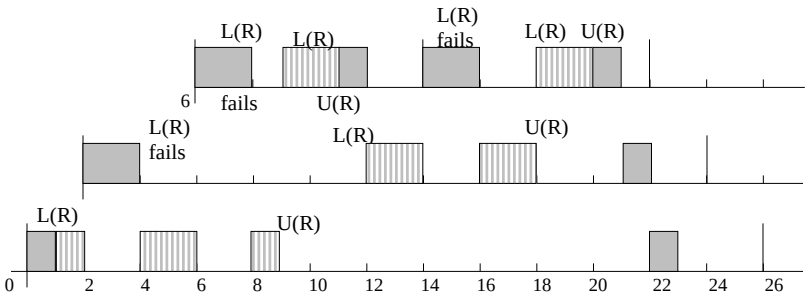
  → **Resources are allocated on nonpreemptive basis.**
- A task holding resources release them in **Last-in-First-out** order.
  - Overlapping critical sections are properly nested.

Example

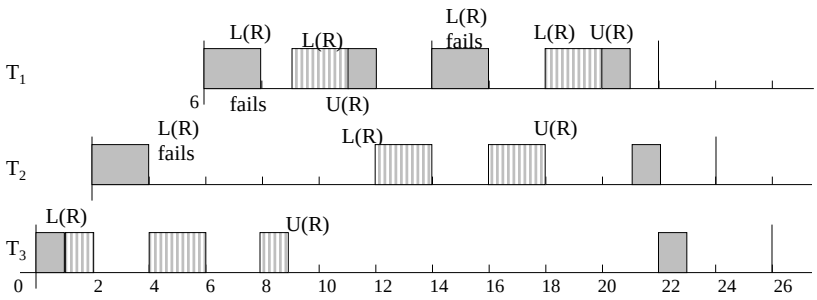


Terms: critical sections, outmost critical section, duration of (outmost) critical sec

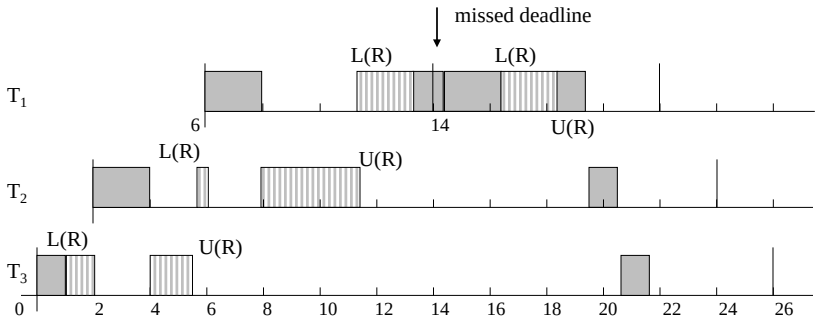
Scheduling anomaly



$T_1 = (8, 5)$, $T_2 = (22, 7)$, $T_3 = (26, 6)$



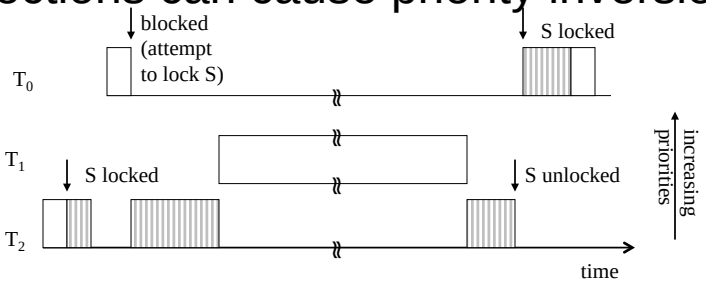
$T_1 = (8, 5)$, $T_2 = (22, 7)$, $T_3 = (26, 6)$



$T_1 = (8, 5)$, $T_2 = (22, 7)$, $T_3 = (26, 4.5)$

How to predict such behavior?

Preemption of tasks in critical sections can cause priority inversion



- Can occur when synchronization mechanisms such as Ada rendezvous, semaphores or monitors are used
- Affect schedulability and predictability if uncontrolled
- Such schedules are allowed by the traditional resource management algorithms, such as the Banker's algorithm, which try to decouple resource management decisions from scheduling decisions.

Direct blockage

- A task T is directly blocked when it executes $Lock(R_i,k)$, but there are less than k units of resource of type R_i available, i.e., not allocated to some task(s).
- The scheduler grants the lock request, i.e., allocates the requested resources, to task T, according to resource allocation rules, as soon as the resources become available.
- T' directly blocks T if T' holds some resources for which T has requested but not allocated

Priority Inheritance: A basic strategy for controlling priority inversion:

If the priority of T', π', is lower than the priority of T, π, (i.e., $\pi' < \pi$), the priority of T' is raised to π whenever T' directly blocks T.

Other form(s) of blocking may be introduced by the resource management policy to control priority inversion or/and prevent deadlock.

References

- Jane W.S. Liu, Real-Time Systems, Prentice Hall, 2000.
- C.L. Liu and J.W. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, Journal of the ACM, Vol. 20 No. 1, pp. 40-61, 1973.
- Lui Sha, et al. "Real Time Scheduling Theory: A Historical Perspective," Journal of Real-time Systems, December 2004.