

Chapter 11 Real-time Scheduling

- Classification of Scheduling Algorithms
- Schedulability Test
- Dynamic Scheduling
 - Scheduling of Independent Tasks
 - Priority Inheritance Protocol for Scheduling of Dependent Tasks
 - Priority Ceiling Protocol for Scheduling of Dependent Tasks
 - Distributed Systems
- Static Scheduling

Mar-08

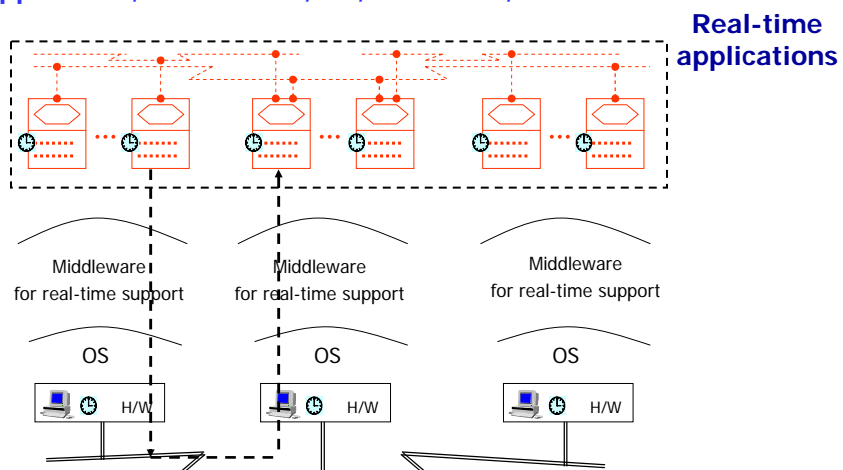
1

UCI
DREAM Lab



Factors impacting response times

- Application, Middleware, OS, Hardware, Comm Network



Mar-08

2

UCI
DREAM Lab



Real-Time Task Scheduling

- The scheduling problem
 - A hard real-time system must execute a set of concurrent real-time tasks in such a way that all time-critical tasks meet their specified deadlines.
 - Every task needs computational & data resources to proceed.
 - Therefore, the scheduling problem is concerned with the allocation of the resources to satisfy all timing requirements.

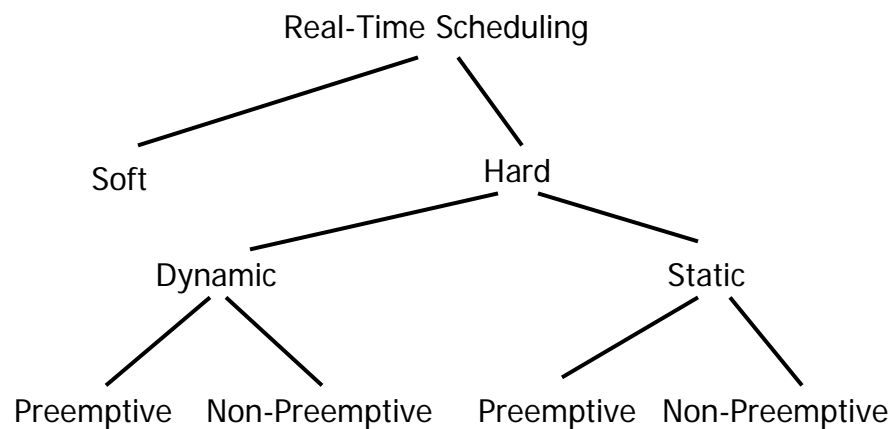
Mar-08

3

UCI
DREAM Lab



Classification of Scheduling Algorithm



Mar-08

4

UCI
DREAM Lab



Dynamic vs. Static

- **Dynamic Scheduling**
 - A scheduler is called **dynamic** (or **on-line**) if it makes its scheduling decisions at run time, selecting one out of the current set of ready tasks.
 - It is flexible and adapt to an evolving task scenario
 - The run-time overhead involved in finding a schedule can be substantial
- **Static Scheduling**
 - A scheduler is called **static** (or **pre-run-time**) if it makes its scheduling decisions at compile time.
 - It needs complete prior knowledge about the task-set characteristics.
 - The run-time overhead is small.
- **Hybrid**

Mar-08

5

UCI
DREAM Lab



Preemptive vs. Non-preemptive

- **Preemptive scheduling**
 - The current executing task may be preempted, i.e., interrupted, if a more urgent task requests service.
- **Non-preemptive scheduling**
 - The current executing task will not be interrupted until it decides on its own to release the allocated resources – normally after completion.

Mar-08

6

UCI
DREAM Lab



Schedulability Test

- A test that determines whether a set of ready tasks can be scheduled in such a way that each task meets its deadline
 - **Exact** schedulability test
 - The test can determine exactly whether a set of ready tasks are schedulable or not.
 - Known as a NP-complete problem
 - **Necessary** schedulability test
 - The negative test result means that a set of tasks are definitely not schedulable.
 - **Sufficient** schedulability test
 - The positive test result means that a set of tasks are definitely schedulable.
- **Optimal** scheduler
 - If a set of tasks can be scheduled to meet all deadlines under a certain scheduler, then the set of tasks can also be scheduled to meet all deadlines under an optimal scheduler.

Mar-08

7

UCI
DREAM Lab



Periodic vs. Sporadic tasks

- Task request time
 - The point in time when a request for a task execution is made.
- **Periodic** task
 - A task whose all future request times are known *a priori* by adding multiples of the known period to the initial request time if the initial request time is determined.
- **Sporadic** task
 - A task whose request times are not known *a priori*.
 - To be schedulable, there must be a minimum interval between any two request times of sporadic tasks.
- **Aperiodic** task
 - A task that has no constraint on the request times of task activation

Mar-08

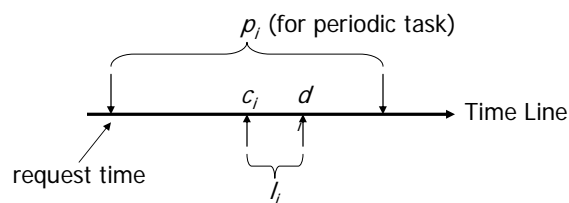
8

UCI
DREAM Lab



A Necessary Schedulability Test for a set of periodic tasks

- Assuming there is a task set, $\{T_i\}$, of periodic tasks with periods p_i , deadline interval d_i , and execution time c_i .
 - **Deadline interval** : the difference between the deadline of a task and the task request time.
 - **Laxity** (l_i) : the difference $d_i - c_i$



Mar-08

9

UCI
DREAM Lab



A Necessary Schedulability Test for a set of periodic tasks

- **Utilization factor** ($\mu_i = c_i / p_i$)
 - The percentage of time the task requires service from a processor
- A necessary schedulability test of a set of periodic tasks:
 - The sum of the utilization factors must be less or equal to n , where n is the number of available processors.

$$\mu = \sum c_i / p_i \leq n$$

Mar-08

10

UCI
DREAM Lab



Static Scheduling

- In static or pre-runtime scheduling, a feasible schedule of a set of tasks is calculated offline.
- The schedule must guarantee all deadlines, considering the resources, precedence, and synchronization requirements of all tasks.
- A **static schedule** is a periodic time-triggered schedule.
- One of the weakness of static scheduling is the assumption of strictly periodic tasks.
 - Although the majority of tasks in hard real-time applications is periodic, there are also sporadic requests for service that have hard deadline requirements.

Mar-08

11

UCI
DREAM Lab



Rate Monotonic Scheduling Framework (A very narrow application range in practice)

- A dynamic preemptive algorithm based on static task priorities
 - The task with the shortest iteration period gets the highest priority.
- Assumptions
 - i. The requests for all tasks of the task set $\{T_i\}$, for which hard deadlines exist, are periodic.
 - ii. All tasks are **independent of** each other. There exists no precedence constraints or mutual exclusion constraints between any pair of tasks
 - iii. The deadline interval of every task T_i is **equal** to its period p_i .
 - **A severe restriction ! Holds very rarely in practice !!**
 - iv. The required maximum computation time of each task c_i is known *a priori* and is constant.
 - v. The time required for context switching can be ignored.
 - vi. The sum of the utilization factor μ of the n tasks satisfies :

$$\mu = \sum c_i / p_i \leq n(2^{1/n} - 1)$$

Approaching ln 2 as
n goes to infinity

UCI
DREAM Lab



Mar-08

12

Rate Monotonic Scheduling Framework (A very narrow application range in practice) (cont)

- The rate monotonic algorithm assigns static priorities based on the task periods.
 - The task with the shortest period gets the highest static priority.
- At run time, the dispatcher selects the task request with the highest static priority.
- If all assumptions are satisfied, the rate monotonic algorithm guarantees that all tasks will meet their deadlines.
- The algorithm is an optimal static priority assignment algorithm for single processor systems which run the **special class of task sets** satisfying assumptions i - vi.
- If the task periods are multiples of the period of the highest priority task (**a severely restrictive case**), the assumption (vi) can be relaxed by :

$$\mu = \sum c_i / p_i \leq 1$$

- For the task sets not satisfying assumption iii, very little is known about the performance of this algorithm.

UCI
DREAM Lab



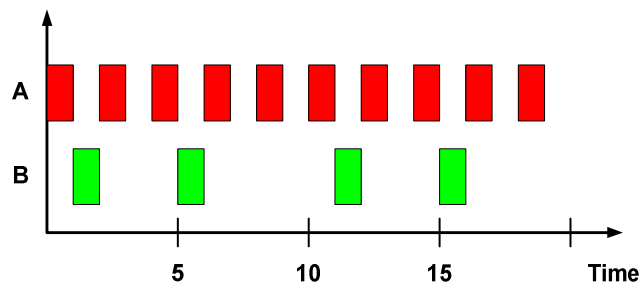
Mar-08

13

Rate Monotonic Scheduling Framework

Task	Deadline interval	Execution Time
A	2	1
B	5	1

$$\mu = \sum c_i / p_i = \frac{1}{2} + \frac{1}{5} = 0.7 \leq 2 * (2^{1/2} - 1) \approx 0.83$$



UCI
DREAM Lab



Mar-08

14

Earliest-Deadline-First (EDF) Scheduling Framework

- This algorithm is an optimal dynamic preemptive algorithm for single processor systems which run a **special highly restrictive class of task sets** (i.e., the deadline of interval of every task T_i is **equal** to its period p_i).
- Assumptions
 - ~ v. **the same as those for rate monotonic algorithm.**
 - iv. The processor utilization factor can go up to 1, even when the task periods are not multiples of the smallest period.
- The task with the earliest deadline is assigned the highest dynamic priority.
- The dispatcher selects the task request with the highest priority.
- For the task sets not satisfying assumption iii, very little is known about the performance of this algorithm.

Mar-08 15

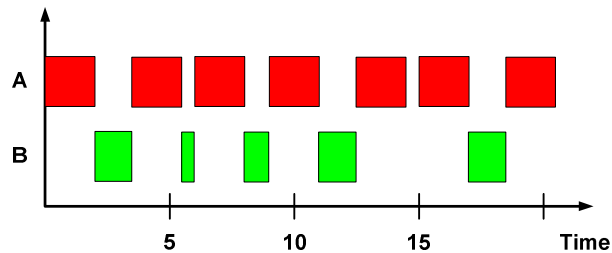
UCI
DREAM Lab



Early-Deadline-First (EDF) Scheduling Framework

Task	Deadline interval	Execution Time
A	3	2
B	5	1.5

$$\mu = \sum c_i / p_i = \frac{2}{3} + \frac{1.5}{5} \approx 0.97 \leq 1$$

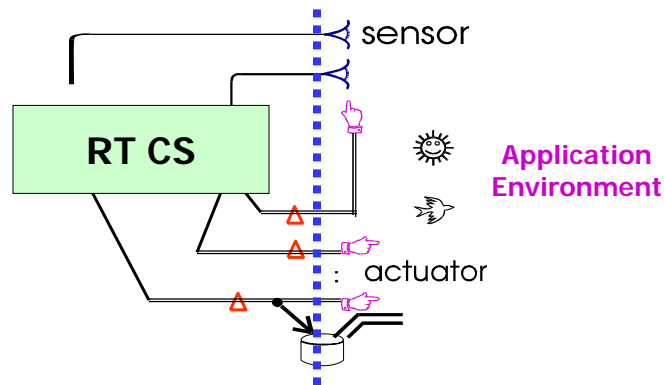


Mar-08 16

UCI
DREAM Lab



How do we determine **deadlines** and **intermediate deadlines** ?



Why has **EDF** been practiced rarely ?

Mar-08

17

UCI
DREAM Lab



Least-Laxity-First (LLF) Scheduling Framework

- This algorithm is another optimal dynamic preemptive algorithm for single processor systems which run a **special highly restrictive class of task sets** (i.e., the deadline of interval of every task T_i is equal to its period p_i).
- Assumptions
 - i. ~ vi. the same as those of EDF algorithm
- **Laxity** (l_i) at time t : the difference between the time remaining before the deadline d_i at t and remaining execution time c_i of a task T_i at t , i.e., $(d_i - c_i)$.
- The task with the least laxity is assigned the highest dynamic priority and the dispatcher selects the task request with the highest priority.
- In multiprocessor systems, neither the EDF or the LLF algorithm is optimal, although the LLF algorithm can handle task scenarios which the EDF algorithm cannot handle.

Mar-08

18

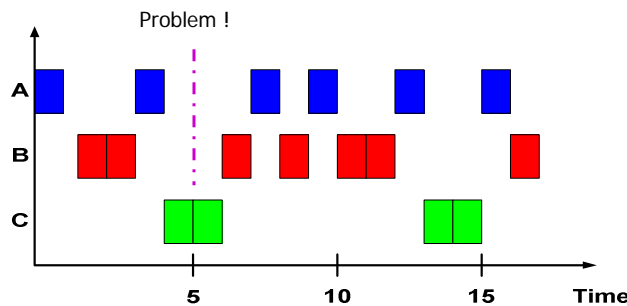
UCI
DREAM Lab



Least-Laxity-First (LLF) Scheduling Framework

Task	Deadline interval	Execution Time
A	3	1
B	5	2
C	9	2

$$\mu = \sum c_i / p_i = \frac{1}{3} + \frac{2}{5} + \frac{2}{9} \approx 0.96 \leq 1$$



Mar-08

19

UCI
DREAM Lab



A Flaw in Pure Least-Laxity-First (LLF) Preemptive Scheduling

- Pure LLF can create an **oscillation**.

Task	Deadline interval	Execution Time	Initial Laxity
A	5	2	3
B	6	2	4
C	10	4.5	5.5

- After Task A executes for one time-unit, what are the laxities for all tasks ?
- A practical variation: **Semi-preemptive LLF**
 - Make a selection of a running task periodically (i.e., after each time-slice) by using LLF.

Mar-08

20

UCI
DREAM Lab



More Realistic General Situations: Cooperating Tasks & Distributed Systems

- It is difficult to guarantee tight deadlines by dynamic scheduling techniques in a single processor multi-tasking system if mutual exclusion and precedence constraints among the tasks must be considered.
- The situation is more complex in a distributed system, where non-preemptive access to the communication medium must be controlled.

Mar-08

21

UCI
DREAM Lab



Priority Inversion

- Consider a set of 3 tasks T1, T2 and T3 (T1 has the highest priority and T3 has the lowest priority), which are scheduled with the rate-monotonic algorithm.
- T1 and T3 require exclusive access to a common resource protected by the semaphore S.
- It can happen that the low priority task T3 has exclusive access to the common resource within a critical section and releases the semaphore S.
- If during this time interval T2 requests service, this service will be granted and T2, the medium priority task, effectively delays T3 and consequently T1, the highest-priority task.
- This phenomenon is called [priority inversion](#).

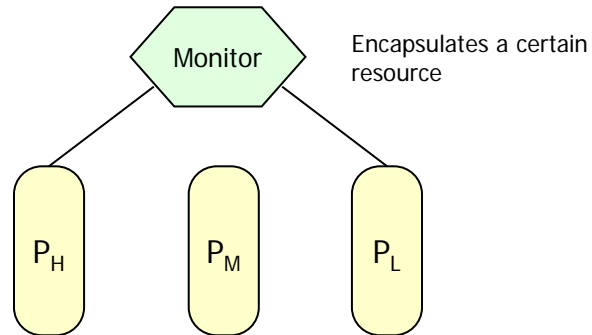
Mar-08

22

UCI
DREAM Lab



Nature of the Priority Inversion



- P_H : High-priority process
- P_M : Medium-priority process
- P_L : Low-priority process

Mar-08

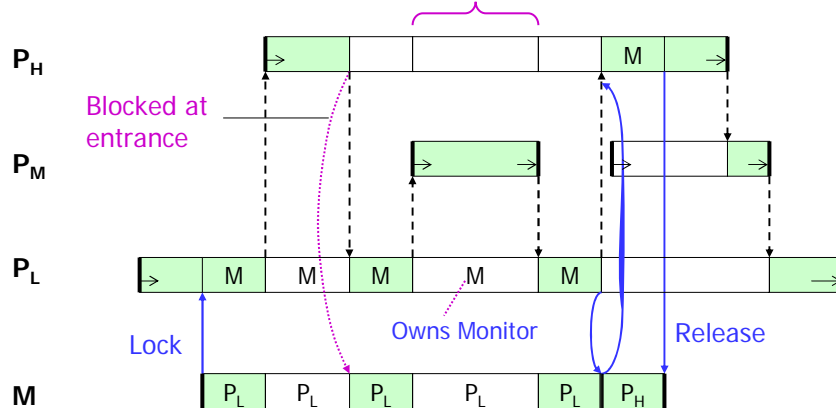
23

UCI
DREAM Lab



Nature of the Priority Inversion (cont)

Priorities of P_H and P_M are *inversed*.



-----> Processor switch

|> |> Begin & end of a Ready state

█ Process or Monitor in execution

↑ ↓ Monitor locking & release

Mar-08

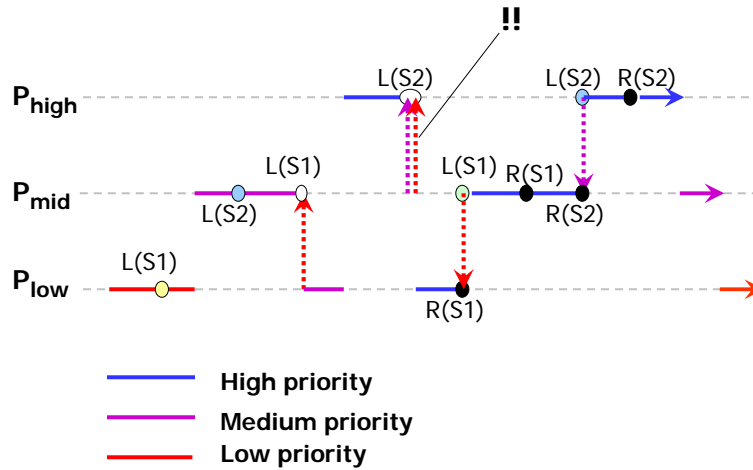
24

UCI
DREAM Lab



Priority Inheritance Protocol

- Priority inheritance is **transitive**.



Mar-08

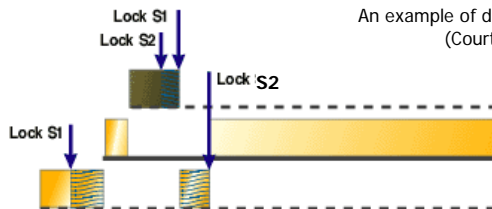
27

UCI
DREAM Lab



Priority Inheritance Protocol

- Tasks L & H share two data buffers, guarded by semaphores S1 & S2.
- Task L locks semaphore S1, but before it manages to lock S2, a switch to a higher priority task occurs.
- Task H locks S2, then tries to lock S1. Through the rules of priority inheritance, task L inherits a high priority and resumes execution.
- L tries to lock S2, but unfortunately S2 is already locked. Neither H nor L can run, so task M resumes execution. H and L never resume execution; they are deadlocked.



An example of deadlock involving priority inheritance
(Courtesy from Embedded.com)

Is this deadlock caused by priority inheritance or the design of the application?

Mar-08

28

UCI
DREAM Lab



Summary of the PI Protocol

- Priority inheritance is transitive.
- An efficient schedulability test is hard to obtain.
- As will be shown later, it does not eliminate all possible priority inversions.

Mar-08

29

UCI
DREAM Lab



Priority Ceiling Protocol

- **Priority ceiling** of a semaphore := the priority of the highest priority task that may lock this semaphore.
- A task T is allowed to enter a critical section only if its assigned priority is **higher than the priority ceilings of all semaphores currently locked by tasks other than T**.
- Task T runs at its assigned priority **unless** it is in a critical section **and blocks higher priority tasks**.
 - In this case it inherits the priority of the tasks it blocks.
- When it exits the critical section it resumes the priority it had at the point of entry into the critical section.

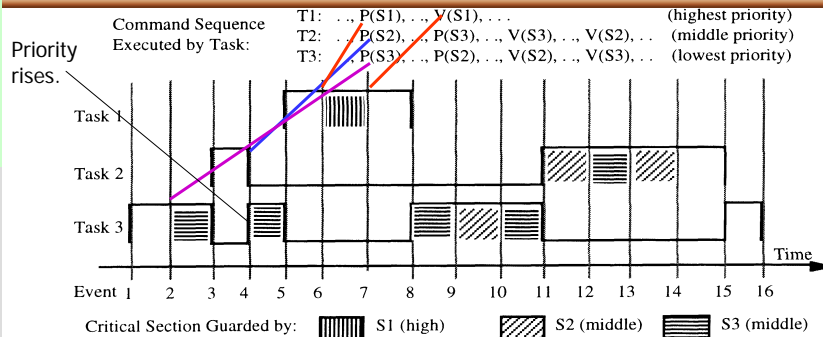
Mar-08

30

UCI
DREAM Lab



Priority Ceiling Protocol – Ex. 1



Event	Action
1	T3 begins execution.
2	T3 locks S3. The priority of T3 does not change.
3	T2 is started and preempts T3.
4	T2 becomes blocked when trying to access S2 since the priority of T2 is not higher than the priority ceiling of the locked S3. T3 resumes the execution of its critical section at the inherited priority of T2.
5	T1 is initiated and preempts T3.
6	T1 locks the semaphore S1. The priority of T1 is higher than the priority ceiling of all locked semaphores.
7	T1 unlocks semaphore S1.

Mar-08 31

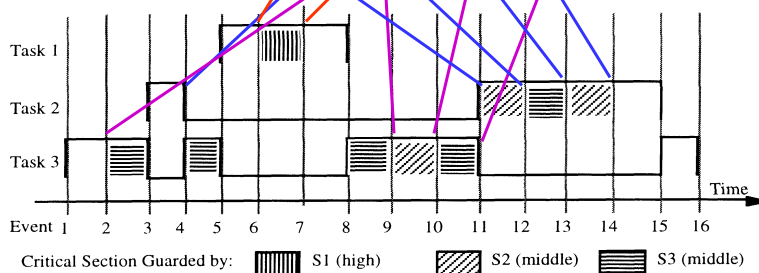
UCI
DREAM Lab



Priority Ceiling Protocol – Ex. 1

Command Sequence Executed by Task:

T1:	... P(S1), ..., V(S1), ...	(highest priority)
T2:	... P(S2), ..., P(S3), ..., V(S3), ..., V(S2), ...	(middle priority)
T3:	... P(S3), ..., P(S2), ..., V(S2), ..., V(S3), ...	(lowest priority)



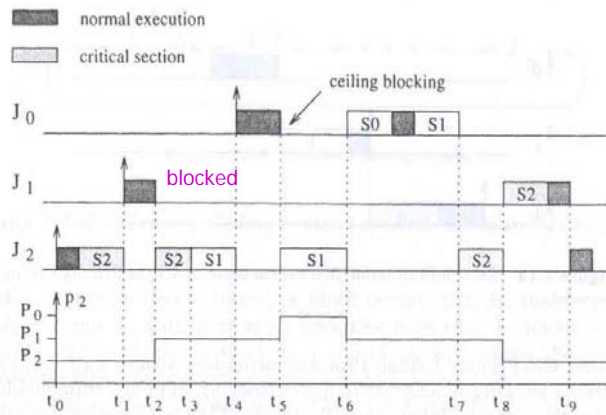
8	T1 finishes its execution. T3 continues with the inherited priority of T2.
9	T3 locks semaphore S2.
10	T3 unlocks S2.
11	T3 unlocks S3 and returns to its lowest priority. At this point T2 can lock S2.
12	T2 locks S3.
13	T2 unlocks S3.
14	T2 unlocks S2.
15	T2 completes. T3 resumes its operation.
16	T3 completes.

Mar-08 32

UCI
DREAM Lab



Priority Ceiling Protocol – Ex. 2



Ceiling(S0) = P0 C(S1) = P0 C(S2) = P1

t2 : J1 can not lock S2. Currently J2 is holding S2 and C(S2) = P1 and the current priority of J1 is also P1.

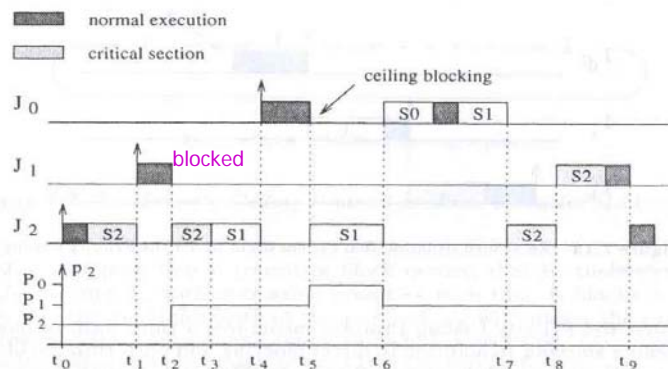
UCI
DREAM Lab



Mar-08

33

Priority Ceiling Protocol – Ex. 2



t5 : J0 can not lock S0. Currently J2 is holding S2 and S1 and C(S1) = P0 and the current priority of J0 is also P0. The (inherited) priority of J2 is now P0.

t6 : J2 unlocks S1. Now J2's (inherited) priority is only P1 and $P0 > C(S2) = P1$. => J0 preempts J2 and runs to completion.

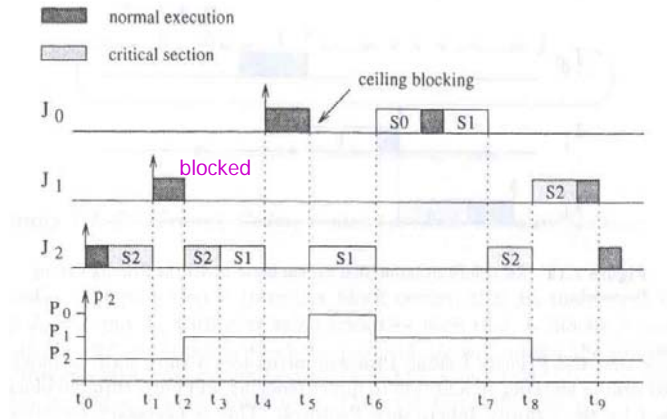
UCI
DREAM Lab



Mar-08

34

Priority Ceiling Protocol – Ex. 2



- t7** : J2 resumes execution with priority P1.
- t8** : J2 unlocks S2 and goes back to its nominal priority P2.
So, J1 preempts J0 and runs to completion.

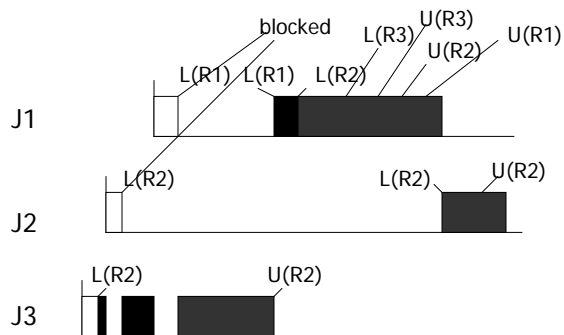
UCI
DREAM Lab



Mar-08 35

Priority Ceiling Protocol – Ex. 3

- Task H can get blocked only for the duration of one critical section of a lower-priority task.



L(R1): Lock R1
 U(R1): Unlock R1

UCI
DREAM Lab



Mar-08 36

Schedulability Test for Priority Ceiling Protocol

- Assume a set of periodic tasks, $\{T_i\}$ with period p_i and computation times c_i .

We denote the worst-case blocking time of a task t_i by lower priority tasks by B_i .

The set of n periodic tasks $\{T_i\}$ can be scheduled, if the following set of inequalities holds:

$$\forall i, 1 \leq i \leq n: \left(\frac{c_1}{p_1} + \frac{c_2}{p_2} + \dots + \frac{c_i}{p_i} + \frac{B_i}{p_i} \right) \leq i(2^{1/i} - 1)$$

Blocking time due to all lower priority tasks

UCI
DREAM Lab



Mar-08

37

Summary of the PC Protocol

- Can only be used for **fixed-priority systems**
- Requires extensive design-time analysis of the system** (i.e., how tasks with different priorities access the resources) in order to calculate the ceiling priority for each semaphore
- A task can be **blocked for at most the duration of one critical section of a lower-priority task ?**
- Less efficient than the priority inheritance protocol ?
- Minor point -- Reduce the chance of deadlocks by suppressing a lot of concurrency

UCI
DREAM Lab

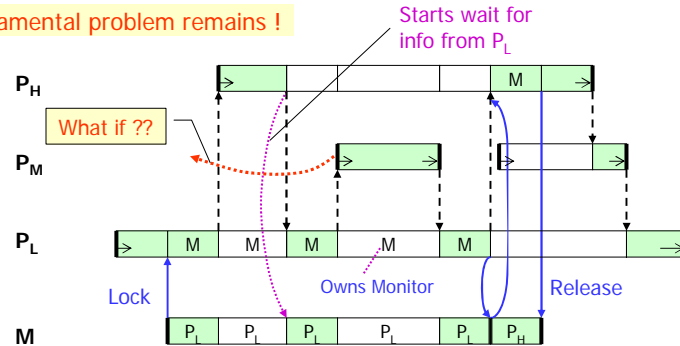


Mar-08

38

Nature of the Priority Inversion (cont)

- Fundamental problem remains !



- If P_M became ready after P_L had acquired the exclusive ownership of M but before P_H became ready, P_M would have preempted P_L in using CPU and other execution resources until P_H became ready, even under the priority inheritance protocol.
- Should we not complain about this delay caused by P_M in execution of P_L , which will in turn cause a delay in the progress of P_H later

Mar-08

39

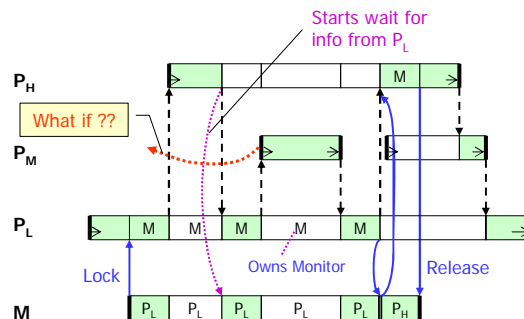
UCI
DREAM Lab



Nature of the Priority Inversion (cont)

- Fundamental problem I :

Low-quality program structure



- Urgent computation-segments given inappropriately low priority specifications cannot avoid poor treatment by the scheduler in the OS !

Mar-08

40

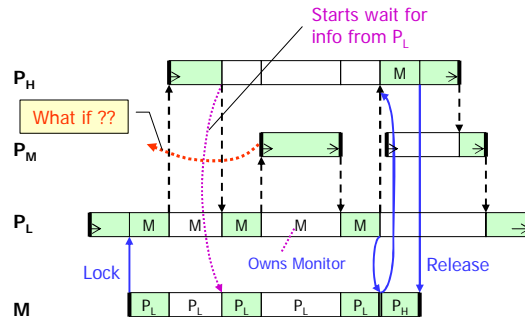
UCI
DREAM Lab



Nature of the Priority Inversion (cont)

- **Fundamental problem II :**

For the **schedulability analysis**, also called the **execution safety analysis**, which is to analyze the **deadline violation possibility**,



- A search for a **complicated global analysis procedure**, not just **simple calculation of processor utilization bounds**, is dictated but finding such a procedure is not easy.
 - This problem was introduced largely when we made it possible for P_H to be blocked at the entrance of the monitor and possibly inside the monitor.

UCI
DREAM Lab

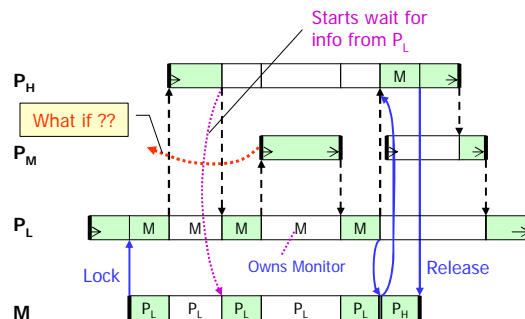


Mar-08

41

Nature of the Priority Inversion (cont)

- **Fundamental problems I & II (cont)**



- Why should P_H wait inside the monitor for information to be supplied by P_L ?
- Shouldn't a supplier of such information to P_H also be a high-priority process ?
- In a sense, **wrong priorities** were specified for certain computation-segments.

UCI
DREAM Lab



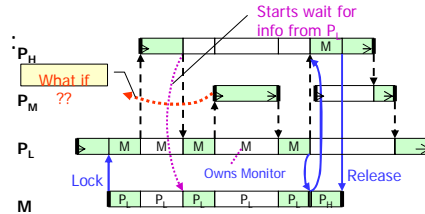
Mar-08

42

Nature of the Priority Inversion (cont)

Fundamental problem III

Modern RT distributed computing applications are much more complicated than what can be adequately represented by fixed-priority processes.



- Typically involve multi-step fusion of data from distributed sensors.
 - Fusion of new sensor data with the contents of a database, e.g., historical records on sensor data, is also of frequent necessity.
- Often necessary to facilitate sharing of data between periodic, or more generally, time-triggered (TT) data acquisition operations and event-triggered reactive operations.
- Co-location of RT processes for handling of incoming fresh data and those processes for internal health check and adaptive reconfiguration within the system is a frequent practice.

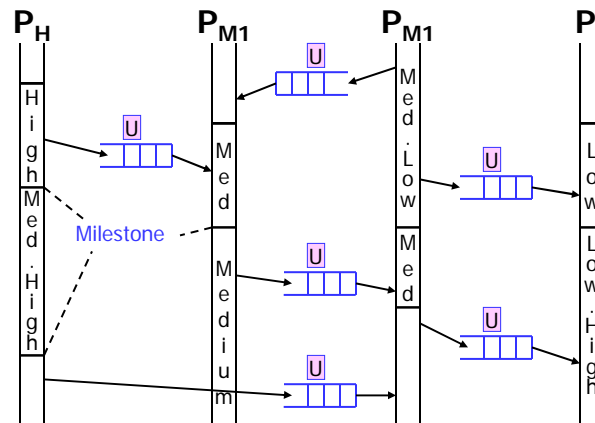
UCI
DREAM Lab



Mar-08 43

Fundamental Limitations of Fixed-Priority Processes

- A process often acts as both a producer and a consumer.
- Such a process is generally subject to multiple deadlines, each imposed on a different milestone within the process.



- Translating such multiple deadlines into an effective single fixed-priority number attached to the entire process is impractical in most cases.

UCI
DREAM Lab



Mar-08 44

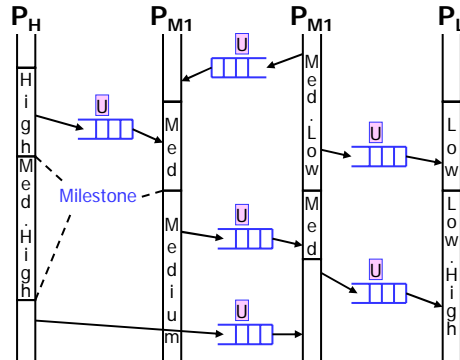
Fundamental Limitations of Fixed-Priority Processes (cont)

- Under the fixed-priority process structuring scheme, a process given a high priority number will be executed before a process given a low priority even if the former has the laxity of 100 milliseconds and the latter has the laxity of zero or one millisecond.

=> Except in very simple cases, **fixed-priority process structuring is a fundamentally a rough design approach.**

- A more natural alternative : Let each process **retain reasonably accurate descriptions of the nature of the deadlines** imposed on itself and let the **execution engine do its best** to meet those deadlines.

E.G., At least, **laxity-driven** or **penalty-driven scheduling is much better.**



Mar-08 45

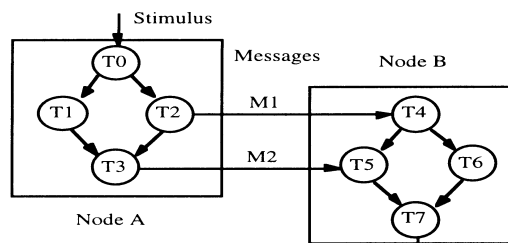
DREAM Lab



Static Scheduling

- In static or pre-runtime scheduling, a feasible schedule of a set of tasks is calculated offline.
- The schedule must guarantee all deadlines, considering the resources, precedence, and synchronization requirements of all tasks.
- Construction of such a schedule can be considered as a constructive sufficient schedulability test.

- The precedence relations between the tasks executing in different nodes can be depicted in the form of a **precedence graph**.



Mar-08 46

DREAM Lab



Static Scheduling

- A static schedule is a periodic time-triggered schedule.
- There is only one interrupt in the system: a periodic clock interrupt denoting the start of a new basic granule.
- Every transaction is periodic, its period being a multiple of the basic granule.
- The least common multiple of all transaction periods is the schedule period.
- At compile time, the scheduling decision for every point of the schedule period must be determined and stored in a dispatcher table for the OS.
- At run time, the preplanned decision is executed by the dispatcher after every clock interrupt.

Mar-08

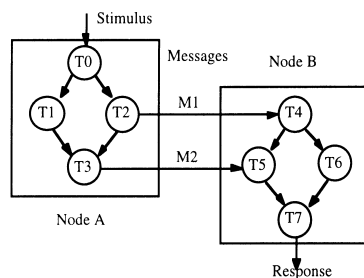
47

UCI
DREAM Lab

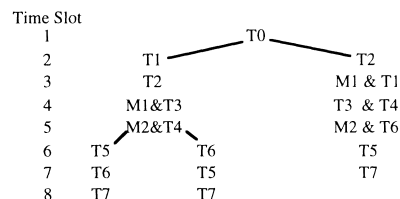


The Search Tree

- The solution to the scheduling problem can be seen as finding a path, a feasible schedule, in a **search tree** by applying a search strategy.
- The goal of the search to find a complete schedule is that observes all precedence and mutual exclusion constraints, and which completes before the deadline.



Precedence graph



The corresponding search tree

Mar-08

48

UCI
DREAM Lab



Increasing the Flexibility in Static Schedules

- One of the weaknesses of static scheduling is the assumption of strictly periodic tasks.
- Although the majority of tasks in hard real-time applications is periodic, there are also sporadic requests for service that have hard deadline requirements.

Mar-08

49

UCI
DREAM Lab



Two Methods to Increase the Flexibility in Static Schedules

- Transformation of a sporadic request to a periodic request:
 - A sporadic task is treated as a periodic task continuously demanding a substantial fraction of the processing resources in order to guarantee its deadline, although it might request service very infrequently.

Mar-08

50

UCI
DREAM Lab



Two Methods to Increase the Flexibility in Static Schedules

- **Mode changes:**
 - During the operation of most real-time applications a number of different operating modes can be distinguished.
 - If the system leaves one operating mode and enters another, a corresponding change of schedules must take place.
 - For each mode, a static schedule that will meet all deadlines is calculated off line.
 - Whenever a mode change is requested at run time, the applicable mode change schedule will be activated immediately.

