

# Embedded Systems Development

The background of the slide is a light blue gradient. On the right side, there is a large, semi-transparent image of a clock face with hands. On the left side, there is a semi-transparent image of an airplane in flight, viewed from a low angle.

## Lecture 14 Real-Time Scheduling

Dr. Daniel Kästner  
AbsInt Angewandte Informatik GmbH  
kaestner@absint.com

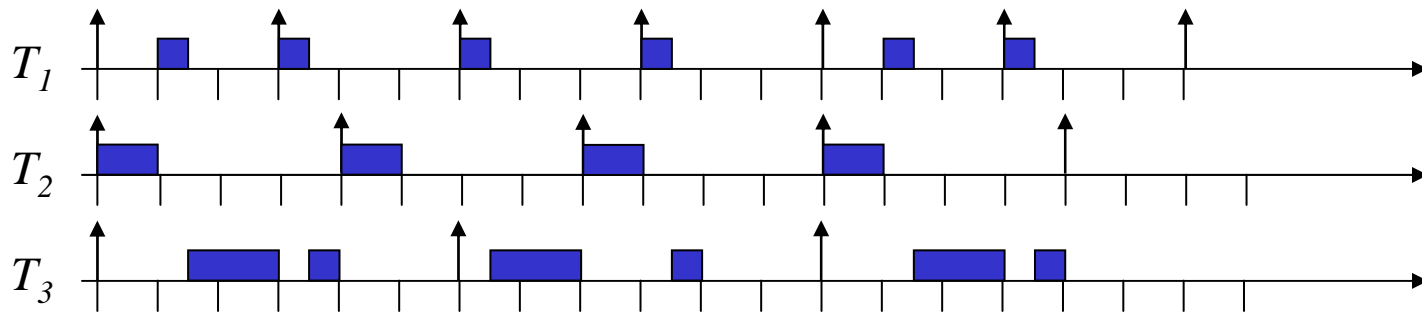
# Lecture Evaluation

- Please do lecture evaluation – Deadline tomorrow!
  - <http://www.clix.uni-saarland.de>



# Deadline Monotonic Scheduling

- Let each process have a unique priority  $P_i$  based on its relative deadline  $d_i$ .
- We assume that **the shorter the deadline, the higher the priority**, ie  $d_i < d_j \Leftrightarrow P_i > P_j$ .
- Same as rate monotonic, if each task's relative deadline equals its period.
- Example schedule:  $T_1$  with  $\pi_1 = d_1 = 3$  and  $c_1 = 0.5$ ,  $T_2$  with  $\pi_2 = 4$ ,  $d_2 = 2$  and  $c_2 = 1$  and  $T_3$  with  $\pi_3 = d_3 = 6$  and  $c_3 = 2$ .



# Schedulability Analysis

- The rate monotonic **schedulability test** can be applied also to deadline monotonic scheduling, by reducing periods to relative deadlines:

$$\sum_{i=1}^N \frac{c_i}{d_i} \leq N(2^{\frac{1}{N}} - 1)$$

- However, this test significantly **overestimates** the workload on the processor.
- Observations:
  - The **worst-case processor demand** occurs when all tasks are released at their critical instants.
  - For each task  $T_i$  the sum of its processing time and the **interference** (preemption) imposed by higher priority tasks must be less than or equal to its deadline  $d_i$ .

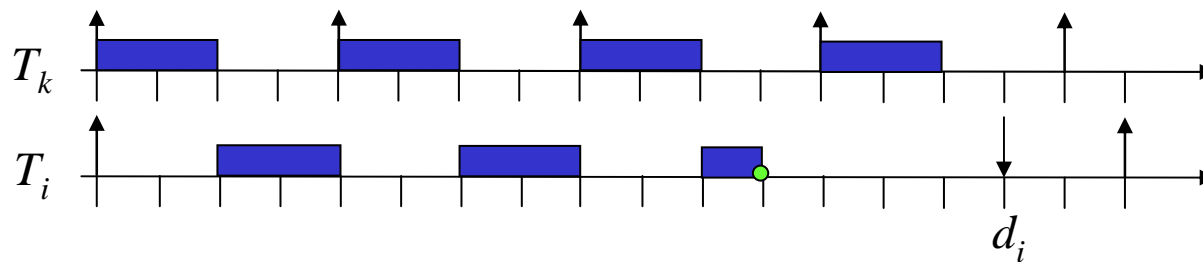


# Schedulability Analysis

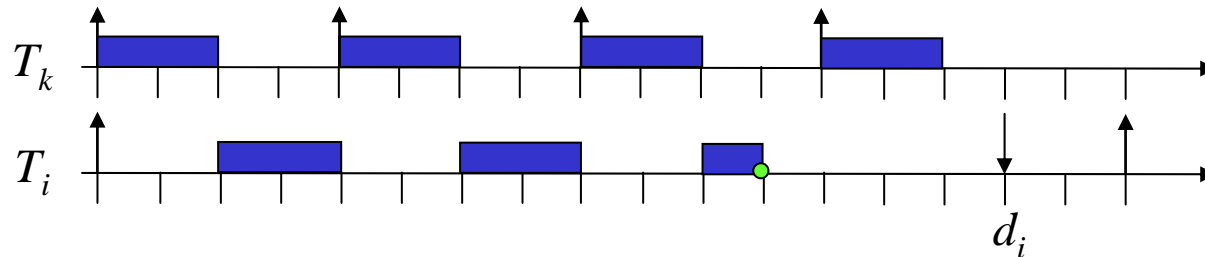
- Assume that tasks are ordered by increasing relative deadlines:  
 $i < j \Leftrightarrow d_i < d_j \Leftrightarrow P_i > P_j$ .
- Then a task set  $\Gamma = \{ T_i \}_{i=1..N}$  is schedulable if the following condition is satisfied:  

$$\forall 1 \leq i \leq N : c_i + I_i \leq d_i$$
- where  $I_i$  is a measure of the interference of  $T_i$ , which can be computed as the sum of the processing times of all higher-priority tasks released before  $d_i$ :

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{d_i}{\pi_j} \right\rceil c_j$$



# Schedulability Analysis



- Note that this test is **sufficient but not necessary**.
- $I_i$  is calculated by assuming that each higher-priority task exactly interferes  $\left\lceil \frac{d_i}{\pi_j} \right\rceil$  times during the execution time of  $T_i$ . However, since  $T_i$  may terminate earlier, the **actual interference** may be **smaller**.
- A sufficient and necessary schedulability test for DM must take the **exact** interleaving of higher-priority tasks into account for each process.



# Response Time Analysis

- The longest **response time**  $R_i$  of a periodic task  $T_i$  is computed as the sum of its computation time and the interference due to preemption by higher-priority tasks at the critical instant.

$$R_i = c_i + I_i$$

where

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{\pi_j} \right\rceil c_j$$

such that

$$R_i = c_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{\pi_j} \right\rceil c_j \quad (*)$$

The worst-case response time is the smallest value of  $R_i$  that satisfies Eq.(\*).



# Response Time Analysis

- Solution: **Fixed point iteration**.
- Let  $R_i^{(k)}$  be the k-th value of  $R_i$  and let  $I_i^{(k)}$  be the interference on task  $T_i$  in the interval  $[0, R_i^{(k)}]$ :

$$I_i^{(k)} = \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(k)}}{\pi_j} \right\rceil c_j$$

- Let  $R_i^{(0)}$  be the first point in time that  $T_i$  could possibly complete:

$$R_i^{(0)} = \sum_{j=1}^i c_j$$

For  $k > 0$  repeatedly compute  $R_i^{(k+1)}$  until  $R_i^{(k+1)} = R_i^{(k)}$ .

$$R_i^{(k+1)} = I_i^{(k)} + c_i = \left( \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(k)}}{\pi_j} \right\rceil c_j \right) + c_i$$



# Response Time Analysis

- The task set is schedulable if  $R_i \leq d_i$  holds for the fixed point  $R_i$ .
- RTA is **necessary and sufficient**.
- Let  $N$  be the number of tasks and  $m$  the number of iterations of the fixed point algorithm. Then the complexity of the RTA algorithm is  $O(Nm)$ .



# Response Time Analysis – Example

- Consider the task set on the right side.
- Assume T1-T3 have been shown to be schedulable. Is also the task set with T4 schedulable?

Process	Period $\pi$	WCET $c$	Deadline $d$
T1	4	1	3
T2	5	1	4
T3	6	2	5
T4	11	1	10

$$R_4^{(0)} = \sum_{j=1}^4 c_j = 1+1+2+1=5$$

$$R_4^{(1)} = \left( \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(0)}}{\pi_j} \right\rceil c_j \right) + c_i = \left\lceil \frac{R_4^{(0)}}{\pi_1} \right\rceil c_1 + \left\lceil \frac{R_4^{(0)}}{\pi_2} \right\rceil c_2 + \left\lceil \frac{R_4^{(0)}}{\pi_3} \right\rceil c_3 + c_4 = \left\lceil \frac{5}{4} \right\rceil 1 + \left\lceil \frac{5}{5} \right\rceil 1 + \left\lceil \frac{5}{6} \right\rceil 2 + 1 = 2 + 1 + 2 + 1 = 6$$

$$R_4^{(2)} = \left( \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(1)}}{\pi_j} \right\rceil c_j \right) + c_i = \left\lceil \frac{6}{4} \right\rceil 1 + \left\lceil \frac{6}{5} \right\rceil 1 + \left\lceil \frac{6}{6} \right\rceil 2 + 1 = 2 + 2 + 2 + 1 = 7$$

$$R_4^{(3)} = \left( \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(2)}}{\pi_j} \right\rceil c_j \right) + c_i = \left\lceil \frac{7}{4} \right\rceil 1 + \left\lceil \frac{7}{5} \right\rceil 1 + \left\lceil \frac{7}{6} \right\rceil 2 + 1 = 2 + 2 + 4 + 1 = 9$$

$$R_4^{(4)} = \left( \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(3)}}{\pi_j} \right\rceil c_j \right) + c_i = \left\lceil \frac{9}{4} \right\rceil 1 + \left\lceil \frac{9}{5} \right\rceil 1 + \left\lceil \frac{9}{6} \right\rceil 2 + 1 = 3 + 2 + 4 + 1 = 10$$

$$R_4^{(5)} = \left( \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(4)}}{\pi_j} \right\rceil c_j \right) + c_i = \left\lceil \frac{10}{4} \right\rceil 1 + \left\lceil \frac{10}{5} \right\rceil 1 + \left\lceil \frac{10}{6} \right\rceil 2 + 1 = 3 + 2 + 4 + 1 = 10$$



# Earliest Deadline First

- Earliest Deadline First (EDF) is a **dynamic** scheduling scheme that selects tasks according to their absolute deadline. **Tasks with earlier deadlines will be executed at higher priorities.**
- So far:  $d_i$  **relative deadline**, ie the time between  $T_i$  becoming available and the time until which  $T_i$  has to finish execution.
- Let  $T_{i,j}$  denote the  $j$ -th instance of task  $T_i$ .
- Let  $r_{i,j}$  be the **release time** of the  $j$ -th instance of task  $T_i$ .
- Let  $\Phi_i$  denote the **phase** of task  $T_i$ , ie the release time of its first instance ( $\Phi_i = r_{i,1}$ ).
- $d_{i,j}$  denotes the **absolute deadline** of the  $j$ -th instance of task  $T_i$  which is given by  $d_{i,j} = \Phi_i + (j - 1) \pi_i + d_i$
- EDF assumes tasks are preemptive; tasks can be periodic or aperiodic.



# Earliest Deadline First

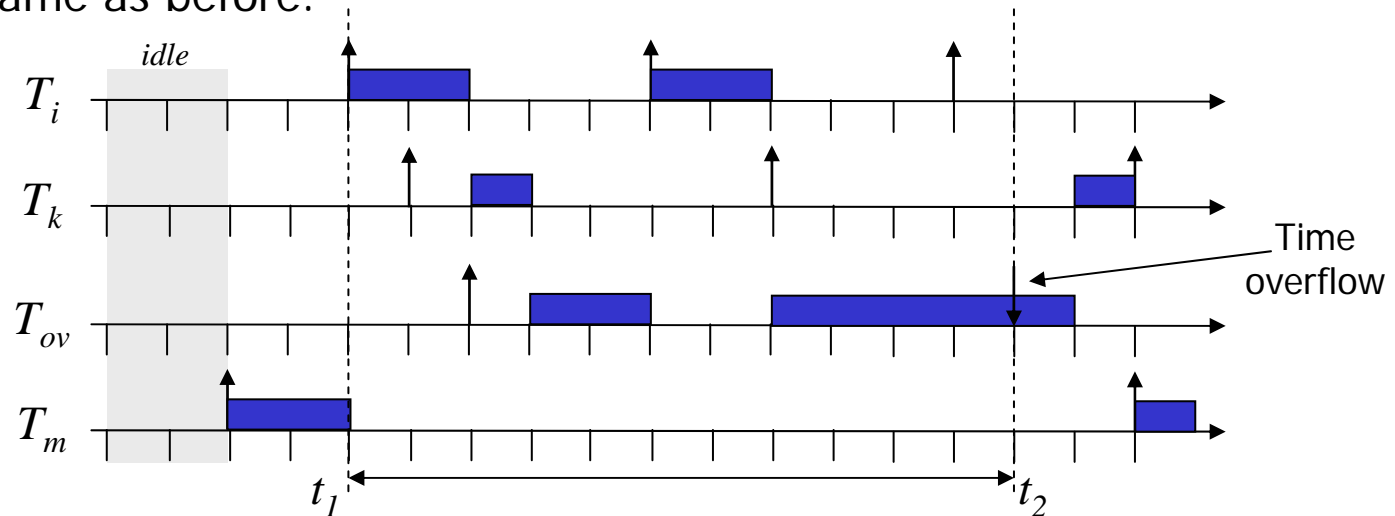
- Theorem: A set of periodic tasks is schedulable with EDF if and only if

$$U = \sum_{i=1}^N \frac{c_i}{\pi_i} \leq 1$$

- Proof:

- "⇒" Same as before.

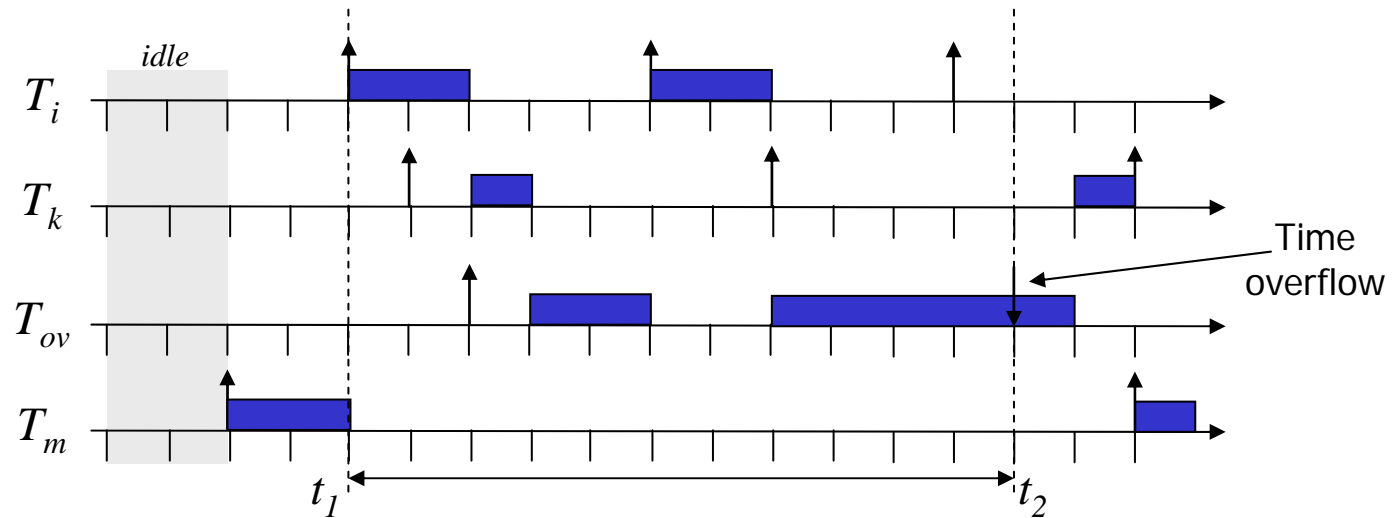
- "⇐"



- Assume that  $U \leq 1$  and the task set is **not** schedulable.
    - Let  $t_2$  be the instant where the deadline violation occurs.
    - Let  $[t_1, t_2]$  be the longest interval of continuous utilization before the overflow, such that only instances with deadline  $\leq t_2$  are executed in  $[t_1, t_2]$ .



# Earliest Deadline First



- Let  $C_p(t_1, t_2)$  be the total computation time demanded by periodic tasks in  $[t_1, t_2]$ . Then

$$C_p(t_1, t_2) = \sum_{r_k \geq t_1, d_k \leq t_2} c_k = \sum_{i=1}^N \left\lfloor \frac{t_2 - t_1}{\pi_i} \right\rfloor c_i \leq \sum_{i=1}^N \frac{t_2 - t_1}{\pi_i} c_i = (t_2 - t_1)U$$

- Since a deadline is missed at  $t_2$ ,  $C_p(t_1, t_2)$  must be greater than the available processor time  $t_2 - t_1$ . Thus

$$t_2 - t_1 < C_p(t_1, t_2) \leq (t_2 - t_1)U \Leftrightarrow U > 1 \quad \text{⚡}$$



# EDF vs. RM

- **Fixed-priority** scheduling is **easier to implement** since priorities are static.
- **Dynamic** schemes require a more complex run-time system which will have **higher overhead**.
- It is easier to incorporate **processes without deadlines** into RM; giving a process an arbitrary deadline is more artificial.



# EDF vs. RM

- During **overload situations**
  - RM is more predictable. Low priority processes miss their deadlines first.
  - EDF is unpredictable; a domino effect can occur in which a large number of processes miss deadlines.
    - To counter this detrimental domino effect, many on-line schemes have two mechanisms:
      - an admissions control module that limits the number of processes that are allowed to compete for the processors, and
      - an EDF dispatching routine for those processes that are admitted
    - An ideal admissions algorithm prevents the processors getting overloaded so that the EDF routine works effectively



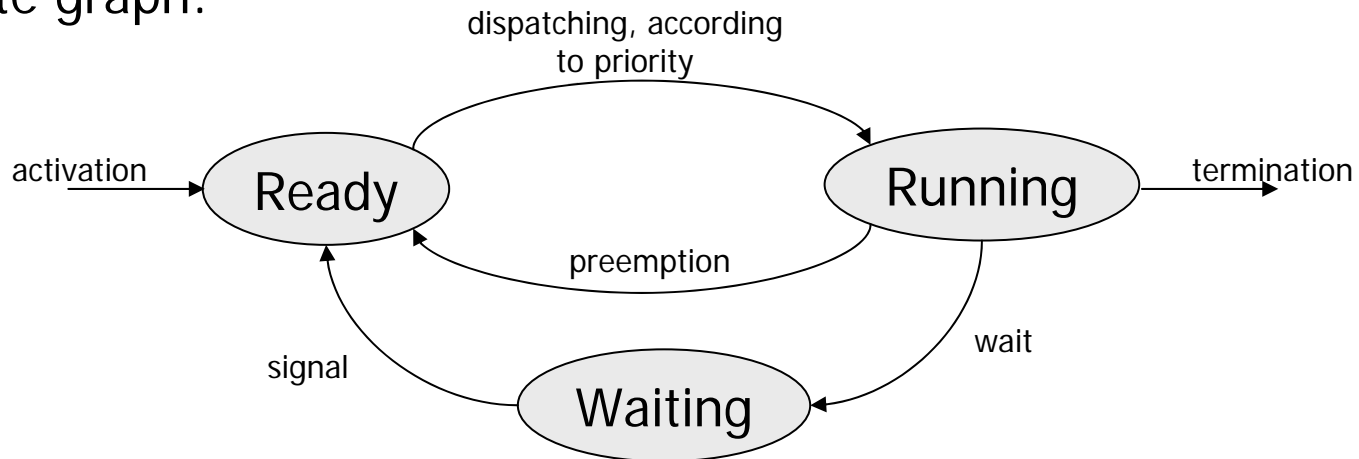
# Resource Access Protocols

- Resources: data structures, files, devices, ...
  - private resource: dedicated to particular task
  - shared resource: available to more than one task
- To ensure consistency of shared resources, tasks must be granted **exclusive access** → **mutually exclusive resources**. Program sections during which exclusive access to a resource is required are called **critical sections**. A task waiting for a mutually exclusive resource is called **blocked** on that resource.
- Any task which needs to enter a critical section must wait until no other task is holding the resource. Otherwise the task enters the critical section and hold the resource. When the task leaves the critical section, the resource becomes free again.



# Resource Access Protocols

- Classical approach: Each mutually exclusive resource  $R_i$  is protected by a semaphore  $S_i$ . Each critical section on  $R_i$  must begin with  $wait(S_i)$  and end with  $signal(S_i)$  – the only operations supported on semaphores.
- Task state graph:

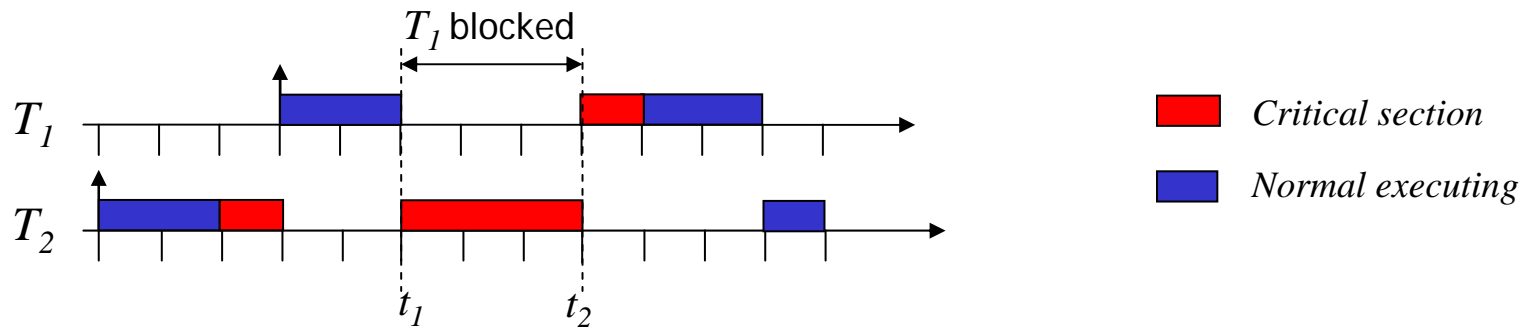


- Problem: **Priority Inversion**.



# Priority Inversion

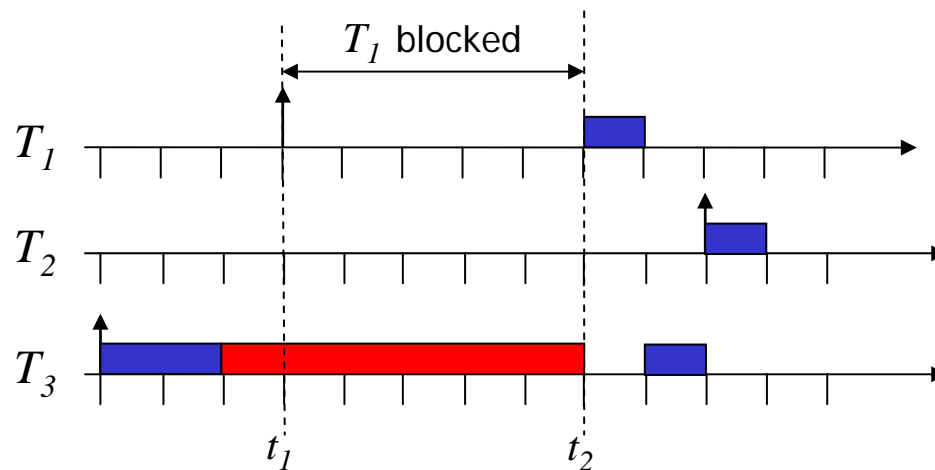
- Let two tasks  $T_1$  and  $T_2$  with priorities  $P_1 > P_2$  be given that share a mutually exclusive resource  $R_k$ .



- $T_2$  is activated first, enters the critical section and locks the semaphore. When  $T_1$  is released, it preempts  $T_2$  since its priority is higher. However, when attempting to enter its critical section at  $t_1$ ,  $T_1$  is blocked on the semaphore, so  $T_2$  resumes – although its priority is lower. In  $[t_1, t_2]$  a priority inversion occurs.

# Priority Inversion

- Naive solution: Disallow preemption during execution of critical sections.
  - May cause unnecessary blocking for a long period of time.
  - Example: Assume  $P_1 > P_2 > P_3$ .  $T_1$  is blocked for a long time although it does not use any resource.



- Better solutions required.



# Priority Inheritance Protocol (PIP)

- Idea: modify the **priority** of the **blocking** tasks.
- Let  $J_i$  denotes a job, ie a generic instance of task  $T_i$ .
- When a job  $J_i$  blocks one or more higher-priority tasks, it temporarily **inherits** the **highest priority** of the blocked tasks. This prevents medium-priority tasks from preempting  $J_i$ .
- Let  $\Gamma = \{T_i\}$  be a set of periodic tasks cooperating through  $M$  shared resources  $R_1, \dots, R_M$ .
- Each resource  $R_i$  is guarded by a distinct semaphore  $S_i$ .
- Assume  $d_i = \pi_i$  for all tasks  $T_i$ .
- The protocol can modify the priority of tasks. Thus:
  - **nominal priority**  $P_i$
  - **active priority**  $p_i$  ( $p_i \geq P_i$ ), which is **dynamic** and initially set to  $P_i$ .



# Priority Inheritance Protocol (PIP)

- Only one job at a time can be within the critical section corresponding to a particular semaphore  $S_i$ .
- Let  $z_{i,j}$  denote the  $j$ th critical section of job  $J_i$ . The  $S_{i,j}$  is the semaphore guarding  $z_{i,j}$  and  $R_{i,j}$  is the resource associated with  $z_{i,j}$ .
- Let  $u_{i,j}$  denote the duration of  $z_{i,j}$ , i.e. the time needed by  $J_i$  to execute  $z_{i,j}$  without interruption.
- We assume priority ordering for jobs  $J_1, J_2, \dots, J_n$  wrt **nominal** priorities such that  $P_1 \geq P_2 \geq \dots \geq P_n$ .
- Critical sections are **perfectly nested**, i.e. either  $z_{i,j} \subset z_{i,k}$  or  $z_{i,k} \subset z_{i,j}$ , or  $z_{i,j} \cap z_{i,k} = \emptyset$ .

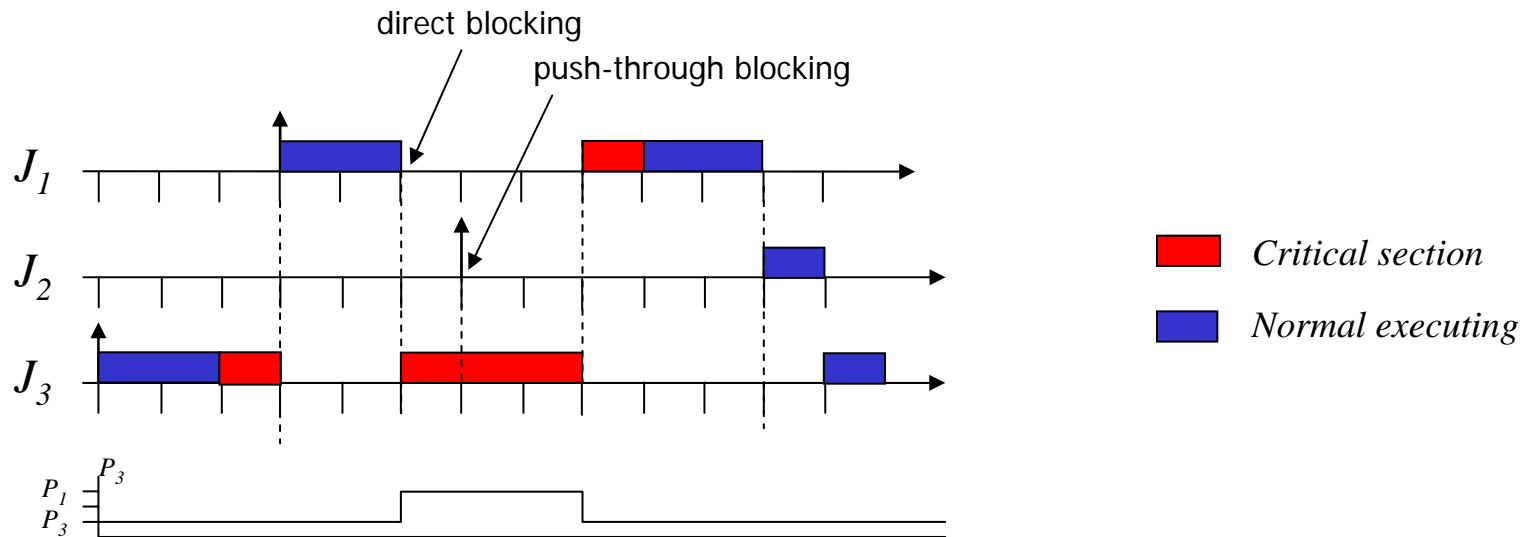


# Definition of Priority Inheritance Protocol

- Jobs are scheduled based on **active priorities**.
- Jobs with the **same** priority are executed **first come first served**.
- When a job  $J_i$  tries to enter a critical section  $z_{i,j}$  and resource  $R_{i,j}$  is already held by a lower-priority job,  $J_i$  will be blocked. Otherwise  $J_i$  enters  $z_{i,j}$ .
- When a job  $J_i$  is **blocked** on a semaphore, it **transmits its active priority** to the job  $J_k$  that holds that semaphore. Then  $J_k$  resumes and executes the rest of its critical section with the **inherited** priority  $p_k = p_i$ .
- When  $J_k$  **exits** a critical section, it unlocks the semaphore and the highest-priority job blocked on that semaphore is awakened. The active priority of  $J_k$  is updated as follows: if no other jobs are blocked by  $J_k$ ,  $p_k$  is set to its nominal priority  $P_k$ , otherwise it is set to the highest priority of the jobs blocked by  $J_k$ .
- Priority inheritance is **transitive**; if  $J_3$  blocks  $J_2$  and  $J_2$  blocks  $J_1$  then  $J_3$  inherits the priority of  $J_1$  via  $J_2$ .



# Priority Inheritance Protocol – Example



- **Direct blocking**: a high-priority job tries to acquire a resource held by a lower-priority job. Necessary to ensure consistency of shared resources.
- **Push-through blocking**: a medium-priority job is blocked by a lower-priority job that has inherited a higher priority from a job it directly blocks. Necessary to avoid unbounded priority inversion.

# Priority Inheritance Protocol – Properties

- **Lemma:** If there are  $n$  lower-priority jobs that can block a job  $J_i$ , then  $J_i$  can be blocked for at most the duration of  $n$  critical sections (one for each of the  $n$  lower-priority jobs), regardless of the number of semaphores used by  $J_i$ .
- **Proof:**
  - A job  $J_i$  can be blocked by a lower-priority job  $J_k$  only if  $J_k$  has been preempted within a critical section  $z_{k,j}$  and is still suspended in the moment when  $J_i$  is initiated.
  - Once  $J_k$  exits  $z_{k,j}$ , it can be preempted by  $J_i$ ; thus  $J_i$  cannot be blocked by  $J_k$  again.
  - The same situation may happen for each of the  $n$  lower-priority jobs; therefore  $J_i$  can be blocked at most  $n$  times.



# Priority Inheritance Protocol – Properties

- **Lemma:** If there are  $m$  distinct semaphores that can block a job  $J_i$ , then  $J_i$  can be blocked for at most the duration of  $m$  critical sections, one for each of the  $m$  semaphores.
- **Proof:**
  - Since semaphores are binary, only one of the lower-priority jobs  $J_k$  can be within a blocking critical section corresponding to a particular semaphore  $S_i$ .
  - Once  $S_i$  is unlocked,  $J_k$  can be preempted and can no longer block  $J_i$ . If all  $m$  semaphores that can block  $J_i$  are locked by  $m$  lower-priority jobs, then  $J_i$  can be blocked at most  $m$  times.



# Priority Inheritance Protocol – Properties

- Theorem (Sha-Rajkumar-Lehoczky):  
Under the Priority Inheritance Protocol, a job  $J$  can be blocked for at most the duration of  $\min(n,m)$  critical sections, where  $n$  is the number of low-priority jobs that could block  $J$  and  $m$  is the number of distinct semaphores that can be used to block  $J$ .
- Proof: immediately follows from the two previous lemmas.



# PIP – Schedulability Analysis

- Liu/Layland: 
$$\sum_{i=1}^N \frac{c_i}{\pi_i} \leq N(2^{\frac{1}{N}} - 1) \quad (*)$$
- Let  $B_i$  be the maximum **blocking time**, due to lower-priority jobs, that a job  $J_i$  may experience.

- **Theorem:** A set of  $n$  periodic tasks using the Priority Inheritance Protocol can be scheduled by the Rate-Monotonic algorithm if

$$\forall 1 \leq i \leq n: \sum_{k=1}^i \frac{c_k}{\pi_k} + \frac{B_i}{\pi_i} \leq i(2^{\frac{1}{i}} - 1)$$

- Proof:

If the criterion holds then a job  $J_i$  has enough time even if it lasted for  $c_i + B_i$ , taking into account the preemption  $c_k/\pi_k$  from higher priority jobs.



# PIP – Response Time Analysis

- To take resources into account, the blocking factor  $B_i$  must be added to the computation time of each task. This gives the following response time equation

$$R_i = c_i + B_i + I_i = c_i + B_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{\pi_j} \right\rceil c_j$$

- The corresponding recurrence equation is:

$$R_i^{(k+1)} = c_i + B_i + \left( \sum_{j=1}^{i-1} \left\lceil \frac{R_i^{(k)}}{\pi_j} \right\rceil c_j \right)$$



# PIP – Computing the Blocking Time

- Let the **ceiling**  $C(S_k)$  of a semaphore  $S_k$  be defined as

$$C(S_k) = \max \{ P_i \mid \text{job } J_i \text{ uses } S_k \}$$

- Let  $D_{i,k}$  denote the duration of the longest critical section of task  $T_i$  among those guarded by semaphore  $S_k$ .
- Let a set of  $N$  periodic tasks that use  $M$  binary semaphores be given. Then the **maximum blocking time**  $B_i$  for each task  $T_i$  can be determined as follows:

$$B_i^l = \sum_{j=i+1}^N \max_k \{ D_{j,k} / C(S_k) \geq P_i \}$$

$$B_i^s = \sum_{k=1}^M \max_{j>i} \{ D_{j,k} / C(S_k) \geq P_i \}$$

$$B_i = \min(B_i^l, B_i^s)$$



# PIP – Example

- Let a set of four tasks with three semaphores be given. The table shows the values  $D_{i,k}$  for a job  $J_i$  and a semaphore  $S_k$ . The semaphore ceilings are given in parentheses.

$D_{i,k}$	$S_1(P_1)$	$S_2(P_1)$	$S_3(P_2)$
$J_1$	1	2	0
$J_2$	0	9	3
$J_3$	8	7	0
$J_4$	6	5	4

- Then the blocking factors for job  $J_1$  are computed as follows:

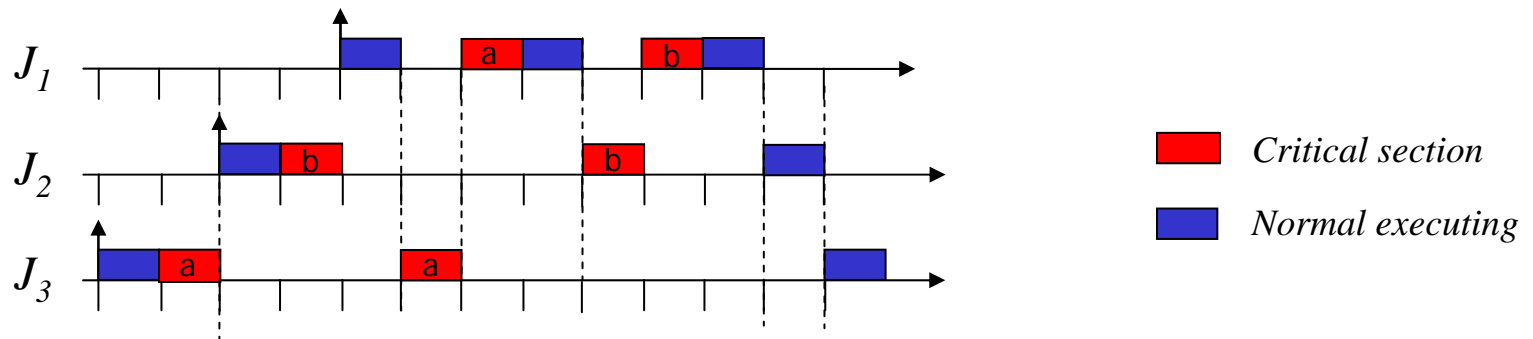
$$B_1^l = \sum_{j=1+1}^4 \max_k \{D_{j,k} / C(S_k) \geq P_1\} = 9 + 8 + 6 = 23$$

$$B_1^s = \sum_{k=1}^M \max_{j>1} \{D_{j,k} / C(S_k) \geq P_1\} = 8 + 9 = 17$$

$$\Rightarrow B_1 = 17$$

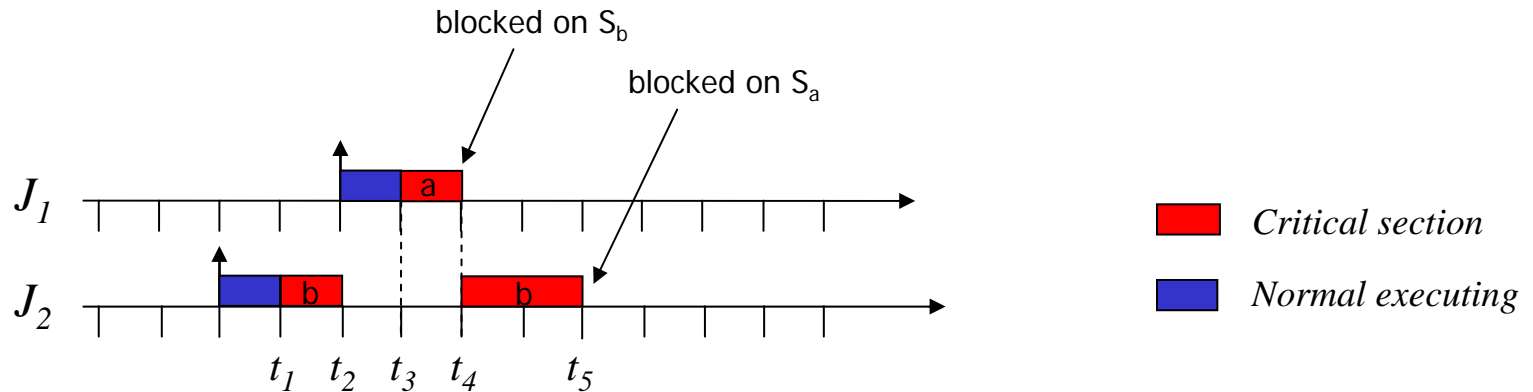


# PIP – Chained Blocking



- In the worst case, if  $J_1$  accesses  $m$  distinct semaphores that have been locked by  $m$  lower-priority jobs, then  $J_1$  will be blocked for the duration of  $m$  critical sections.

# PIP – Deadlocks



- $t_1$ :  $J_2$  locks  $S_b$ .
- $t_2$ :  $J_2$  is preempted by the higher-priority job  $J_1$ .
- $t_3$ :  $J_1$  locks  $S_a$ .
- $t_4$ :  $J_1$  is blocked on  $S_b$ .  $J_2$  resumes and continues execution at the priority of  $J_1$ .
- $t_5$ :  $J_2$  attempts to lock  $S_a$ . => Deadlock!
- Note: deadlock is caused by erroneous use of semaphores.

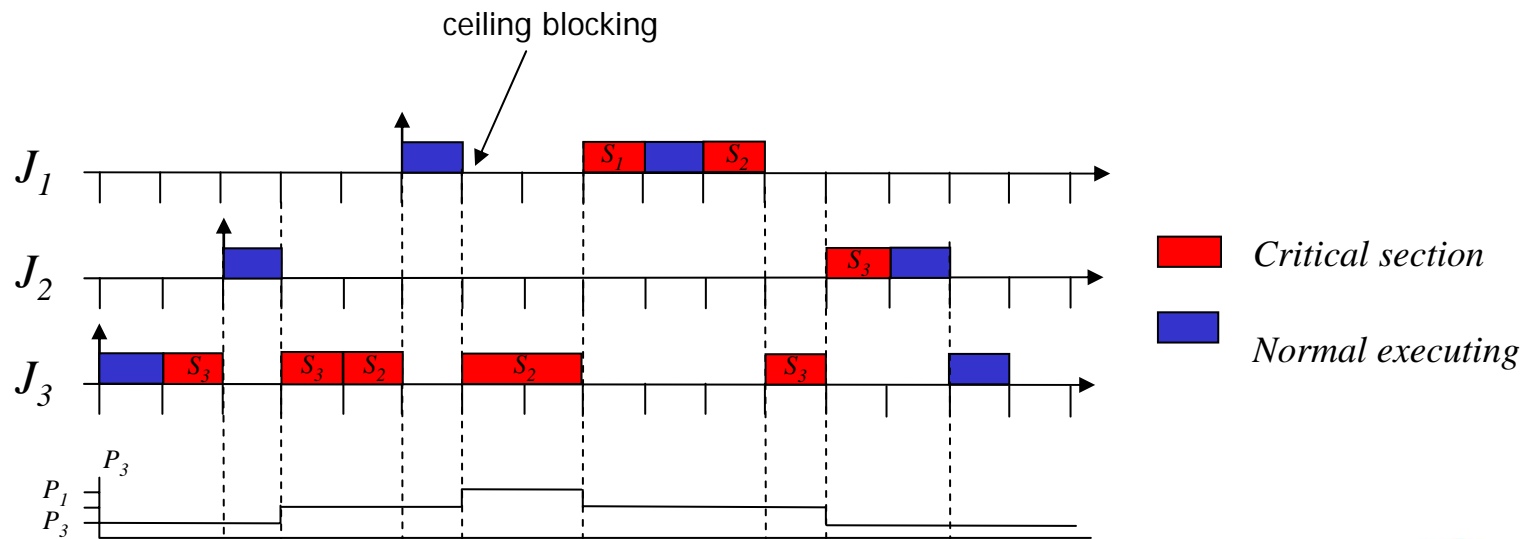
# The Priority Ceiling Protocol

- Each semaphore  $S_k$  is assigned a **priority ceiling**  $C(S_k)$ , equal to priority of the highest-priority task that can lock it. Note that  $C(S_k)$  is a static value that can be computed **offline**.
- When a task  $T_i$  wants to lock a semaphore  $S_k$ , let  $H_i$  be the set of semaphores held by tasks different from  $T_i$  and  $P^* = \max\{C(S') \mid S' \in H_i\}$ .
- Task  $T_i$  gets the lock  $S_k$  only if  $P_i > P^*$ .**
- Note that  $P^*$  is independent from the semaphore  $S_k$ .
- When a job  $J_i$  is blocked on a semaphore it transmits its priority to the job  $J_k$  that holds the semaphore. Hence,  $J_k$  resumes and executes the rest of its critical section with the priority of  $J_i$ .  $J_k$  is said to **inherit** the priority of  $J_i$ .
- When  $J_k$  exits a critical section, it unlocks the semaphore and the highest-priority job, if any, blocked on that semaphore is awakened. The active priority of  $J_k$  is set to the normal priority  $J_k$  if no other jobs are blocked by  $J_k$ , otherwise it is set to the highest priority of the jobs blocked by  $J_k$ .



# The Priority Ceiling Protocol – Example

- Let three jobs  $J_1$ ,  $J_2$ , and  $J_3$  having decreasing priorities be given.
- $J_1$  sequentially accesses two critical sections guarded by semaphores  $S_1$  and  $S_2$ .
- $J_2$  only accesses a critical section guarded by  $S_3$ .
- $J_3$  uses semaphore  $S_3$  and then makes a nested access to  $S_2$ .
- This gives the following priority ceilings:  $C(S_1)=P_1$ ,  $C(S_2)=P_1$ ,  $C(S_3)=P_2$ .

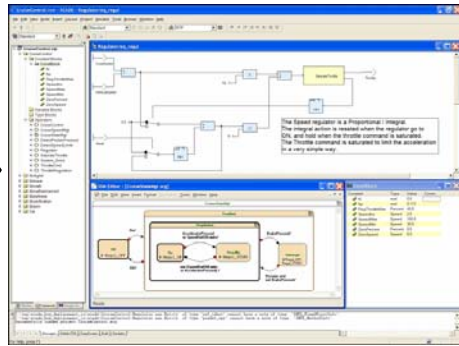


# The Priority Ceiling Protocol - Properties

- A high-priority process can be blocked at most once during its execution by any lower-priority process.
- Deadlocks are prevented.
- Transitive blocking is prevented.



# Model-based Software Development



Generator

Lustre programs  
Esterel programs

Compiler

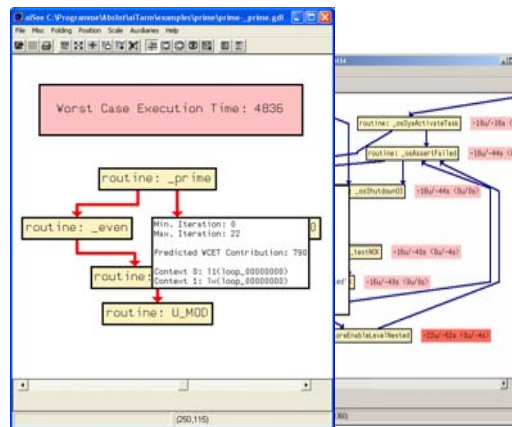
C Code

Compiler

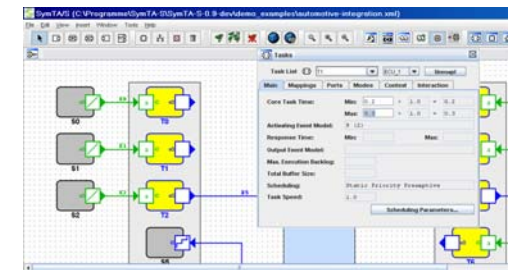
Binary Code

Esterel SCADE

- **SCADE language** ✓
- **SyncCharts** ✓



aiT WCET Analyzer  
- Timing Validation



SymTA/S

- **System-level Scheduling & Schedulability Analysis**

