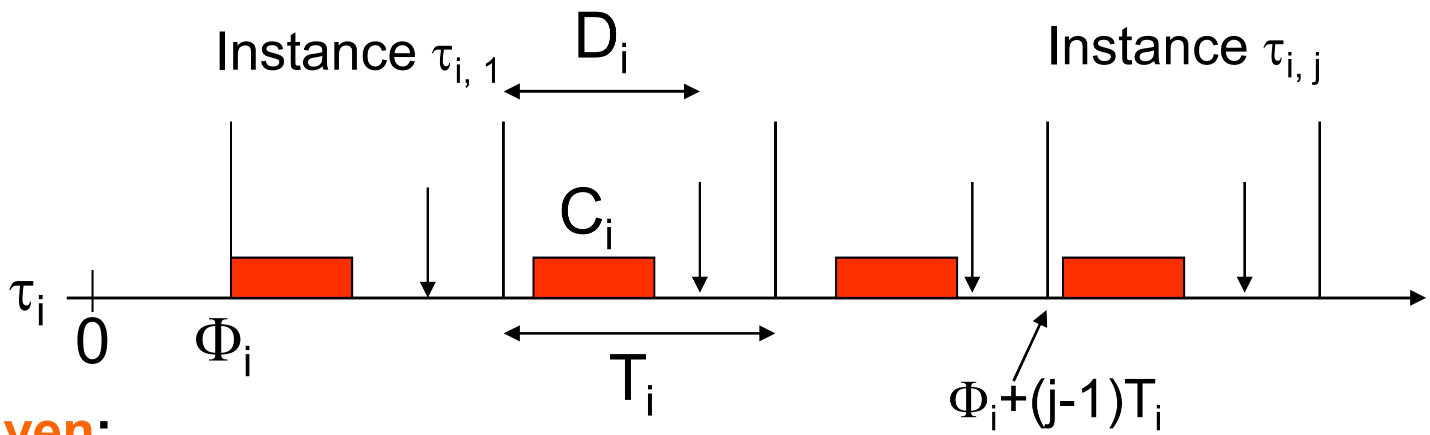
Embedded Systems 16

Periodic scheduling

REVIEW



- **Given**:
  - A set of periodic tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ with
    - phases Φ_i (arrival times of first instances of tasks),
    - periods T_i (time difference between two consecutive activations)
    - relative deadlines D_i (deadline relative to arrival times of instances)
    - computation times C_i

  $\Rightarrow$ *j* th instance $\tau_{i,j}$ of task τ_i with
    - arrival time $a_{i,j} = \Phi_i + (j-1)\,T_i$,
    - deadline $d_{i,j} = \Phi_i + (j-1)\,T_i + D_i$,
    - start time $s_{i,j}$ and
    - finishing time $f_{i,j}$
- **Find a feasible schedule**

Assumptions

REVIEW

A.1. Instances of periodic task τ_i are regularly activated with constant period T_i.

A.2. All instances have same worst case execution time C_i.

A.3. All instances have same relative deadline D_i, here in most cases equal to T_i (i.e., $d_{i,j} = \Phi_i + j \cdot T_i$)

A.4. All tasks in Γ are independent. No precedence relation, no resource constraints.

A.5. Overhead for context switches is neglected, i.e. assumed to be 0 in the theory.

- Basic results based on these assumptions form the core of scheduling theory.
- For practical applications, assumptions A.3. and A.4. can be relaxed, but results have to be extended.

Processor utilization

REVIEW

Definition:

Given a set Γ of n periodic tasks, the **processor utilization U** is given by

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}.$$

- Define $U_{bnd}(A) = \inf \{U(\Gamma) \mid \Gamma$ is not schedulable by algorithm $A\}$.
- If $U_{bnd}(A) > 0$ then a simple, sufficient criterion for schedulability by A can be based on processor utilization:
  - If $U(\Gamma) < U_{bnd}(A)$ then Γ is schedulable by A.
  - However, if $U_{bnd}(A) < U(\Gamma) \leq 1$, then Γ may or may not be schedulable by A.

Earliest Deadline First (EDF)

REVIEW

- **Theorem:** **A set of periodic tasks $\tau_1, ..., \tau_n$ with $D_i = T_i$ is schedulable with EDF iff $U \cdot 1$.**

- EDF is applicable to both periodic and a-periodic tasks.

- If there are only periodic tasks, priority-based schemes like “rate monotonic scheduling (RM)” (see later) are often preferred, since
  - They are simpler due to fixed priorities
    $\Rightarrow$ use in “standard OS” possible
  - sorting wrt. to deadlines at run time is not needed

Rate monotonic scheduling (RM)

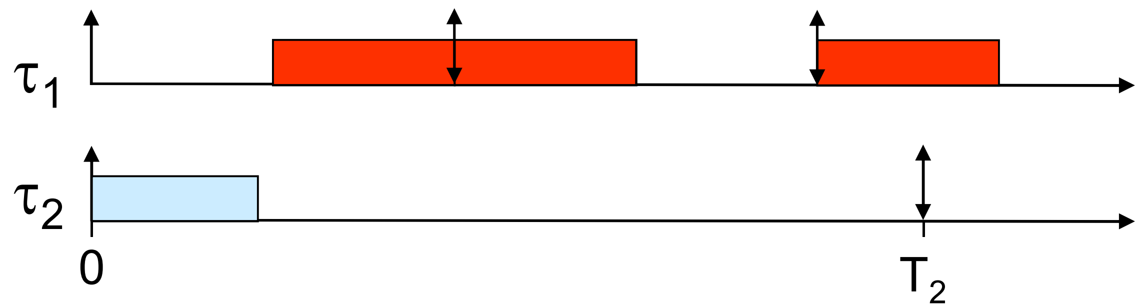
REVIEW

- Rate monotonic scheduling (RM) (Liu, Layland ’73):
  - Assign fixed priorities to tasks τ_i:
    - priority(τ_i) = $1/T_i$
    - I.e., priority reflects release rate
  - Always execute ready task with highest priority
  - Preemptive: currently executing task is preempted by newly arrived task with shorter period.

- **Theorem (Liu, Layland, 1973):**
  RM is optimal among all fixed-priority scheduling algorithms.

Non-RM Schedule

REVIEW

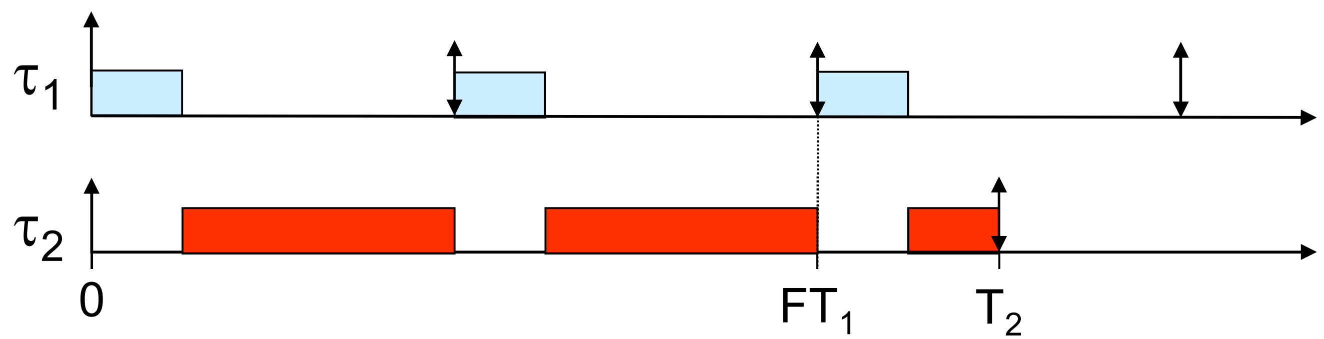


Schedule feasible iff $C_1 + C_2 \leq T_1$

RM-Schedule

REVIEW

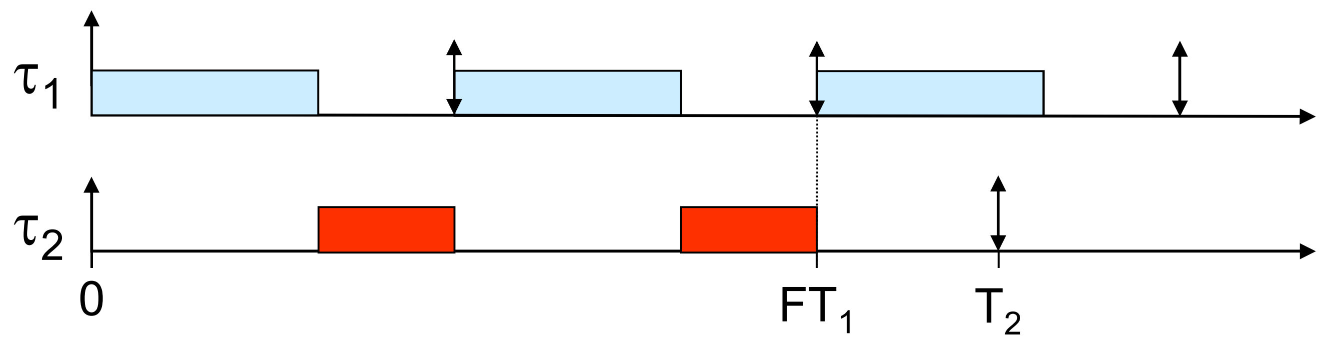
- Let $F = \lfloor T_2 / T_1 \rfloor$ be the number of periods of τ_1 entirely contained in T_2.
- Case 1:
  - The computation time C_1 is short enough, so that all requests of τ_1 within period of τ_2 are completed before second request of τ_2.
  - I.e. $C_1 \leq T_2 - F\,T_1$
  - Schedule feasible if $(F+1)C_1 + C_2 \leq T_2$



RM-Schedule

REVIEW

- Case 2:
  - The second request of τ_2 arrives when τ_1 is running.
  - I.e. $C_1 \geq T_2 - F\,T_1$



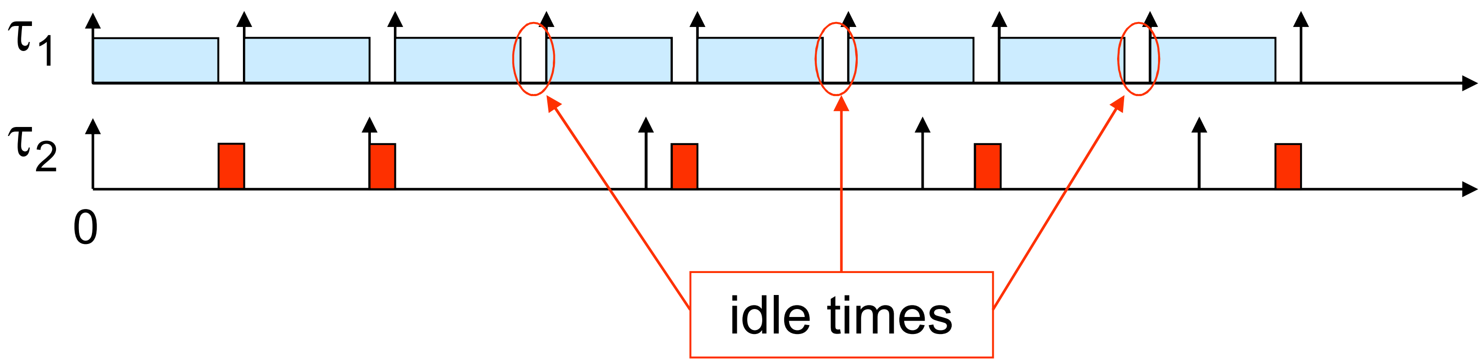
Schedule feasible if $FC_1 + C_2 \leq FT_1$

Computation of U_{bnd}(RM)

- First step: Consider only task sets with 2 tasks
  - Computation of
    $U_{bnd}(RM, 2) = \inf \{U(\Gamma) \mid \Gamma \text{ is not schedulable by RM}, |\Gamma| = 2\}$.
  - Idea:
    - Construct set of tasks with following properties:
      1. Set of tasks is schedulable by RM.
      2. Any increase of computation times makes the set of tasks non-schedulable.
      3. Processor utilization is minimal under properties 1. and 2.
    - “Worst case task set”

Computation of U_{bnd}(RM, 2)

- Worst case situation constructed for 2 processes:



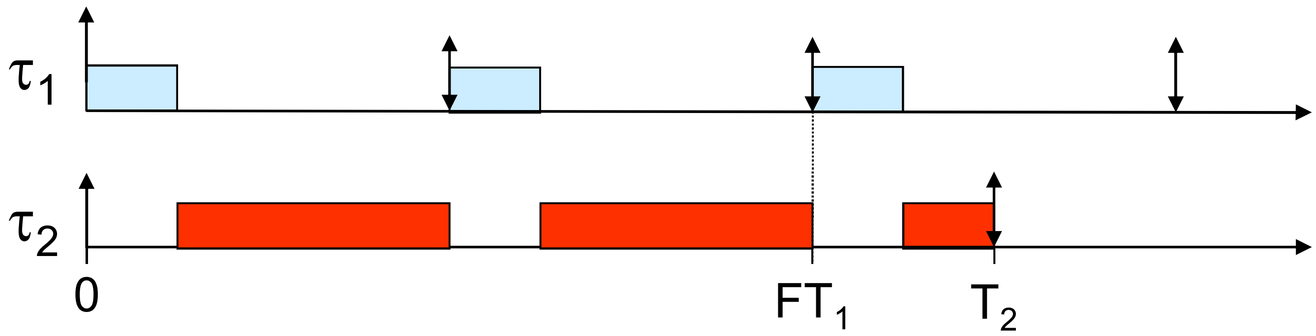
Computation of U_{bnd}(RM, 2)

- Consider a set of 2 periodic tasks τ_1 and τ_2 with $T_1 \leq T_2$ $\Rightarrow$ priority(τ_1) > priority(τ_2).
- We consider the critical instant when τ_1 and τ_2 arrive at the same time.
- We construct a worst case scenario where the processor utilization factor is minimal, but any increase of computation times destroys schedulability

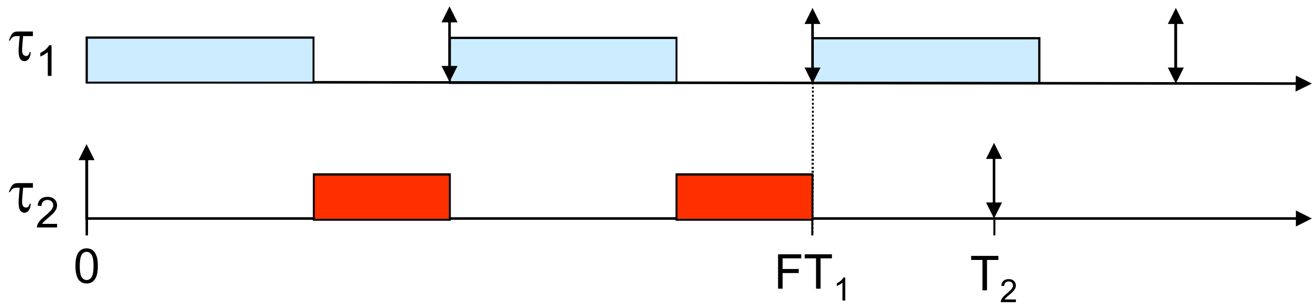
  This is done by manipulating

  - computation times C_1 and C_2 and
  - T_1 and T_2 (more precisely T_2 / T_1)

Case 1: $C_1 \le T_2 - F\ T_1$



Case 2: $C_1 \geq T_2 - F\,T_1$



Manipulating T_2/T_1

Computation of U_{bnd}(RM)

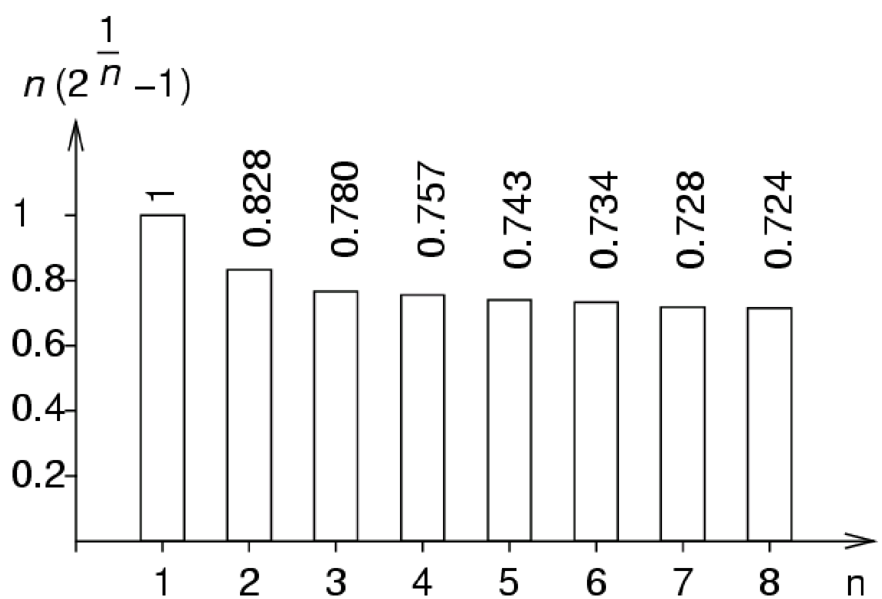
- Result for two processes:
  Any set of two periodic tasks with a processor utilization factor ≤ $\boxed{U_{bnd} = 2(2^{1/2} - 1)}$ can be scheduled by RM.

- Similarly, for the general case of **n** processes the following can be shown:
  Any set of n periodic tasks with a processor utilization factor ≤ $\boxed{U_{bnd} = n(2^{1/n} - 1)}$ can be scheduled by RM.

Computation of U_{bnd}(RM)

- Any set of n periodic tasks with a processor utilization factor ≤ $\boxed{U_{bnd} = n(2^{1/n} - 1)}$ can be scheduled by RM.

- Observation:
  U_{bnd} is decreasing with n.



Schedulability check

- To provide a lower bound for arbitrary n we have to compute $U_{lub} = n(2^{1/n} - 1)$ = ln 2 ≈ 0.69
- Hence, a set of tasks can be scheduled by RM if

$$U < U_{bnd}(\mathrm{RM}) = \ln 2 \approx 0.69$$

- But what can we tell about schedulability when processor utilization factor is larger than $n(2^{1/n} - 1)$?
- Answer:
We can compute a more precise result, if we make use of the knowledge of periods T_i and computation times C_i.

Schedulability check

- Remember:
  The response time $R_{i,\,j}$ of an instance j of task i is the time (measured from the arrival time) at which the instance is finished: $R_{i,\,j} = f_{i,\,j} - a_{i,\,j}$.
- Compute an upper bound R_i on the response time:
  - Suppose that $\tau_1, ..., \tau_n$ are ordered with increasing periods (i.e. decreasing priorities).
  - Consider an arbitrary periodic task τ_i.
  - At a critical instant t, when an instance of τ_i arrives together with all higher priority tasks, we have:
    - $R_i = C_i + \sum_{k=1}^{i-1}$ (# activations of τ_k during $[t, t + R_i]$) $\cdot C_k$
      $= C_i + \sum_{k=1}^{i-1} \lceil R_i/T_k \rceil \cdot C_k$

Schedulability check

$\Rightarrow$ We just need to solve n fixed point equations to compute $R_1, ..., R_n$.

- $R_i = C_i + \sum_{k=1}^{i-1} \lceil R_i/T_k \rceil \cdot C_k$

- R_i is the largest fixed point of the function $f(R_i) = C_i + \sum_{k=1}^{i-1} \lceil R_i/T_k \rceil \cdot C_k$.
- Sufficient condition for schedulability: $R_i \leq D_i \; \forall\, i$
- Problem: Solution of fixed point equation?

Schedulability check

- Solution of fixed point equation?
- Compute the following sequence:
  - $R_i^{(0)} = C_i$.
  - $R_i^{(j+1)} = C_i + \sum_{k=1}^{i-1} \lceil R_i^{(j)} / T_k \rceil \cdot C_k$.
- It is easy to see that this sequence is monotonically increasing, i.e., $f(x) = C_i + \sum_{k=1}^{i-1} \lceil x / T_k \rceil \cdot C_k$ is monotonically increasing.
- $\Rightarrow$ If a least fixed point of f(x) exists, then the sequence converges to this fixed point.

Schedulability check

Lemma: The sequence is unboundedly increasing or converges after a finite number of steps to the least fixed point.

Schedulability check

$\Rightarrow$ Algorithm:

$\forall$ i: $R_i^{(0)} = C_i$

`repeat`

  $\forall$ i: $R_i^{(j+1)} = C_i + \sum_{k=1}^{i-1} \lceil R_i^{(j)} / T_k \rceil \cdot C_k$

`until` ($\exists$ i with $R_i^{(j+1)} > D_i$) **`or`** ($\forall$ i $R_i^{(j+1)} = R_i^{(j)}$);

`if` ($\forall$ i $R_i^{(j+1)} = R_i^{(j)}$) **`then`**

  `report`("RM schedulable");

Summary

- Problem of scheduling independent and preemptable periodic tasks

- Rate monotonic scheduling:
  - Optimal solution among all fixed-priority schedulers
  - Schedulability of n tasks guaranteed, if processor utilization $U \leq n(2^{1/n} - 1)$.

- Earliest deadline first:
  - Optimal solution among all dynamic-priority schedulers
  - Schedulability guaranteed if processor utilization $U \cdot 1$.

Rate Monotonic Scheduling in Presence of Task Dependencies

Assumptions so far

A.1. Instances of periodic task τ_i are regularly activated with constant period T_i.

A.2. All instances have same worst case execution time C_i.

A.3. All instances have same relative deadline D_i, here in most cases equal to T_i (i.e., $d_{i,j} = \Phi_i + j \cdot T_i$)

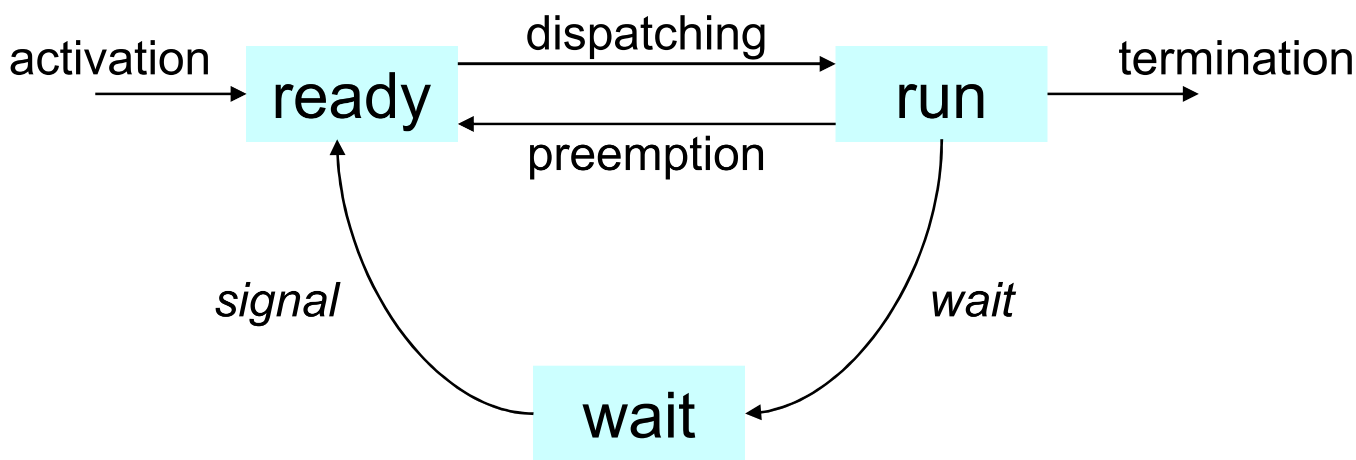
A.4. All tasks in Γ are independent. No precedence relation, no resource constraints.

A.5. Overhead for context switches is neglected, i.e. assumed to be 0 in the theory.

- Basic results based on these assumptions form the core of scheduling theory.
- For practical applications, assumptions A.3. and A.4. can be relaxed, but results have to be extended.

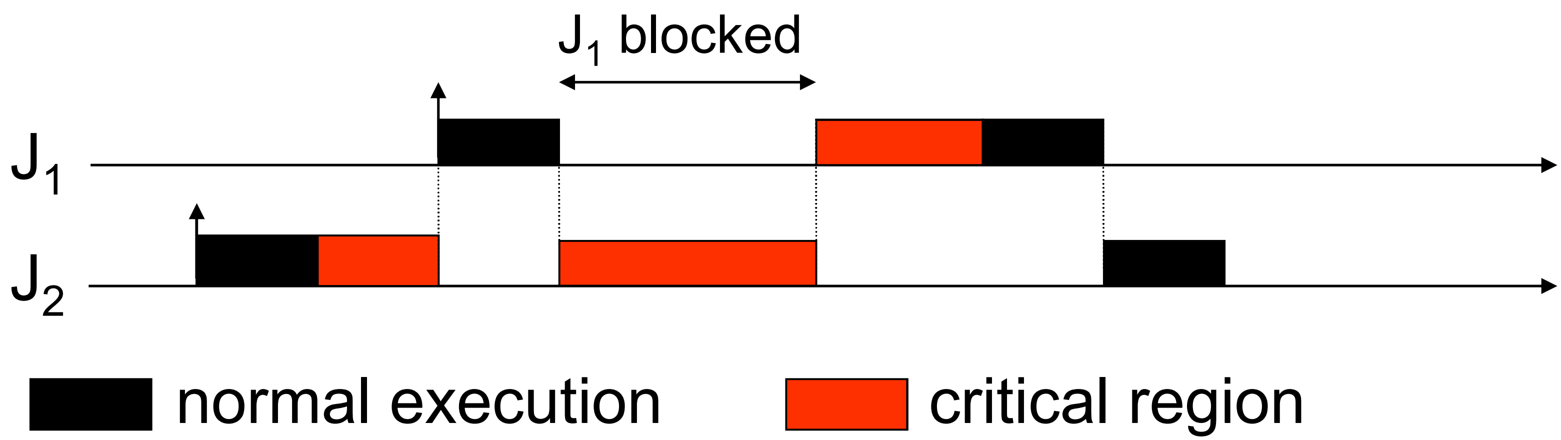
Wait state caused by resource constraints

- Each mutually exclusive resource R_i is protected by a semaphore S_i.
- Each critical section operating on R_i must begin with a *wait*(S_i) primitive and end with a *signal*(S_i) primitive.
- *wait* primitive on locked semaphore
  $\rightarrow$ wait state until another task executes signal primitive



The priority inversion problem

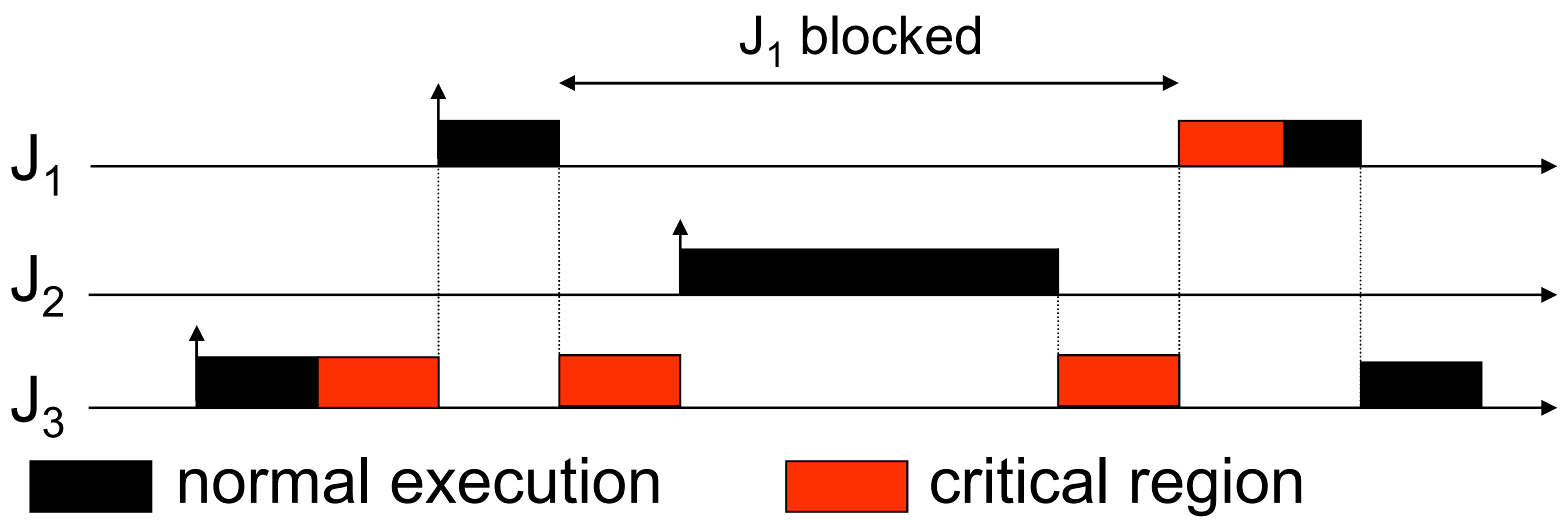
- **Priority inversion** can occur due to resource conflicts (exclusive use of shared resources) in fixed priority schedulers like RM:



priority(J_1) > priority(J_2)

- Here: Blocking time equal to length of critical section.

The priority inversion problem

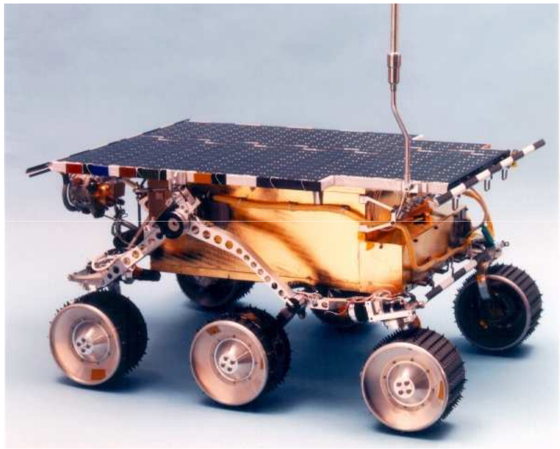


priority(J_1) > priority(J_2) > priority(J_3)

- Blocking time equal to length of critical section + computation time of J_2.
- Unbounded time of priority inversion, if J_3 is interrupted by tasks with priority between J_1 and J_3 during its critical region.

Priority inversion in real life: The MARS Pathfinder problem (1)

“But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once".” …



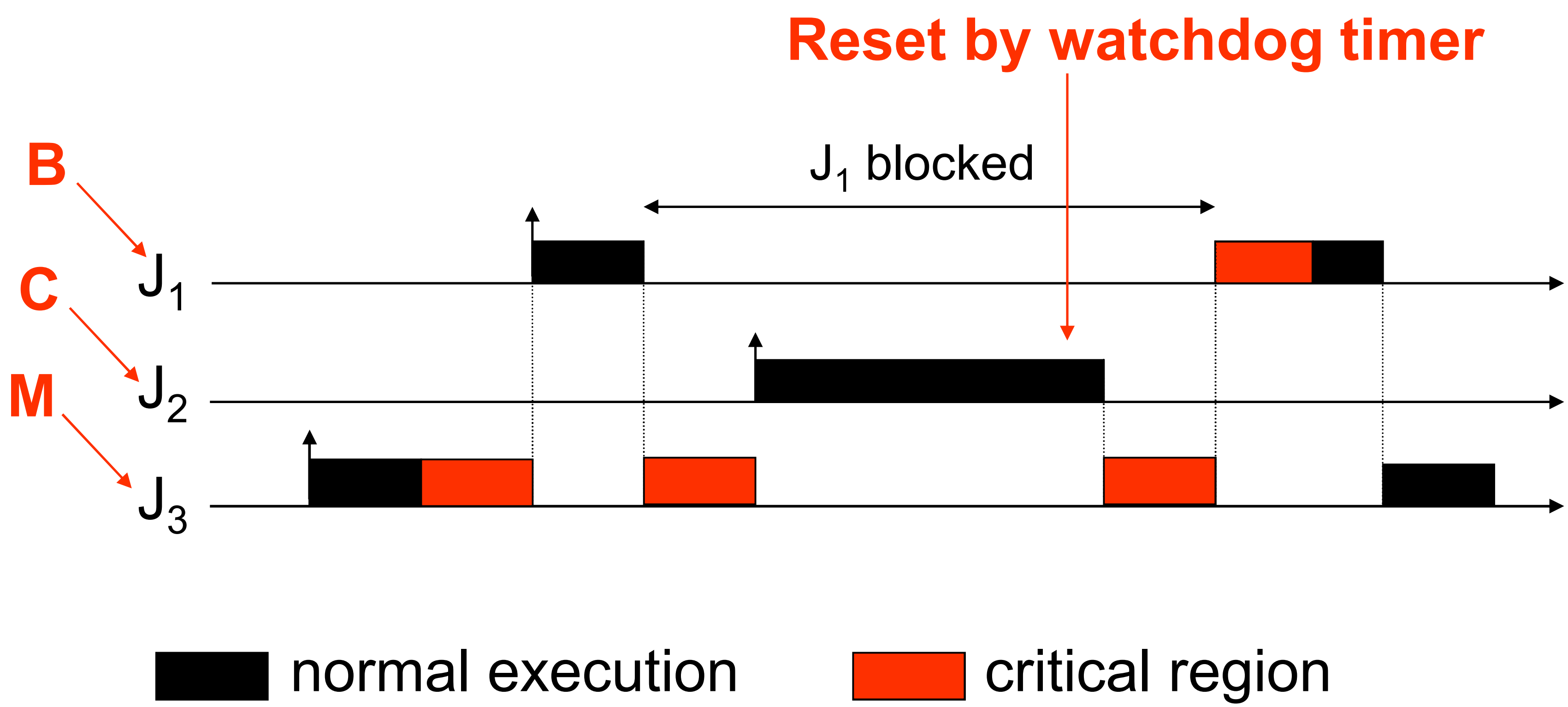
Priority inversion in real life: The MARS Pathfinder problem (2)

- System overview:
  - ***Information Bus (IB):***
    - Buffer for exchanging data between different tasks
    - Shared resource of two tasks M and B
  - Three tasks:
    - ***Meteorological data gathering task (M):***
      – collects meteorological data
      – reserves IB, writes data to IB, releases IB
      – infrequent task, low priority
    - ***Bus management (B):***
      – data transport from IB to destination
      – reserves IB, data transport, releases IB
      – frequent task, high priority

Priority inversion in real life: The MARS Pathfinder problem (3)

- Three tasks:
  - ***...***
  - ***"Communication task" (C):***
    - – medium priority, does not use IB
- Scheduling with fixed priorities.
- ***Watch dog timer (W):***
  - Execution of B as indicator of system hang-up
  - If B is not activated for certain amount of time: Reset the system

Priority inversion in real life: The MARS Pathfinder problem (5)



priority(J_1) > priority(J_2) > priority(J_3)

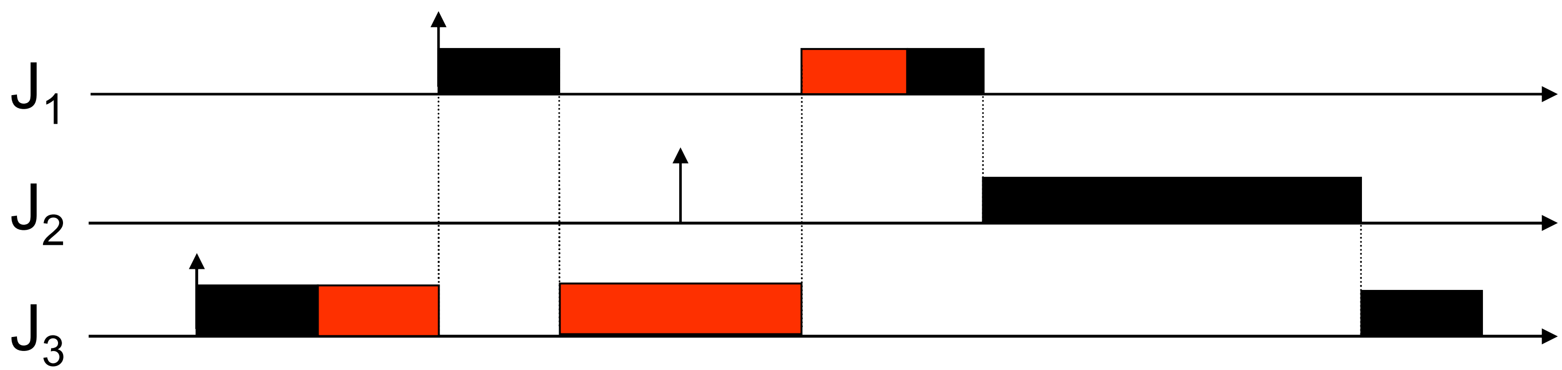
Coping with priority inversion: The priority inheritance protocol

Idea of **priority inheritance protocol**:

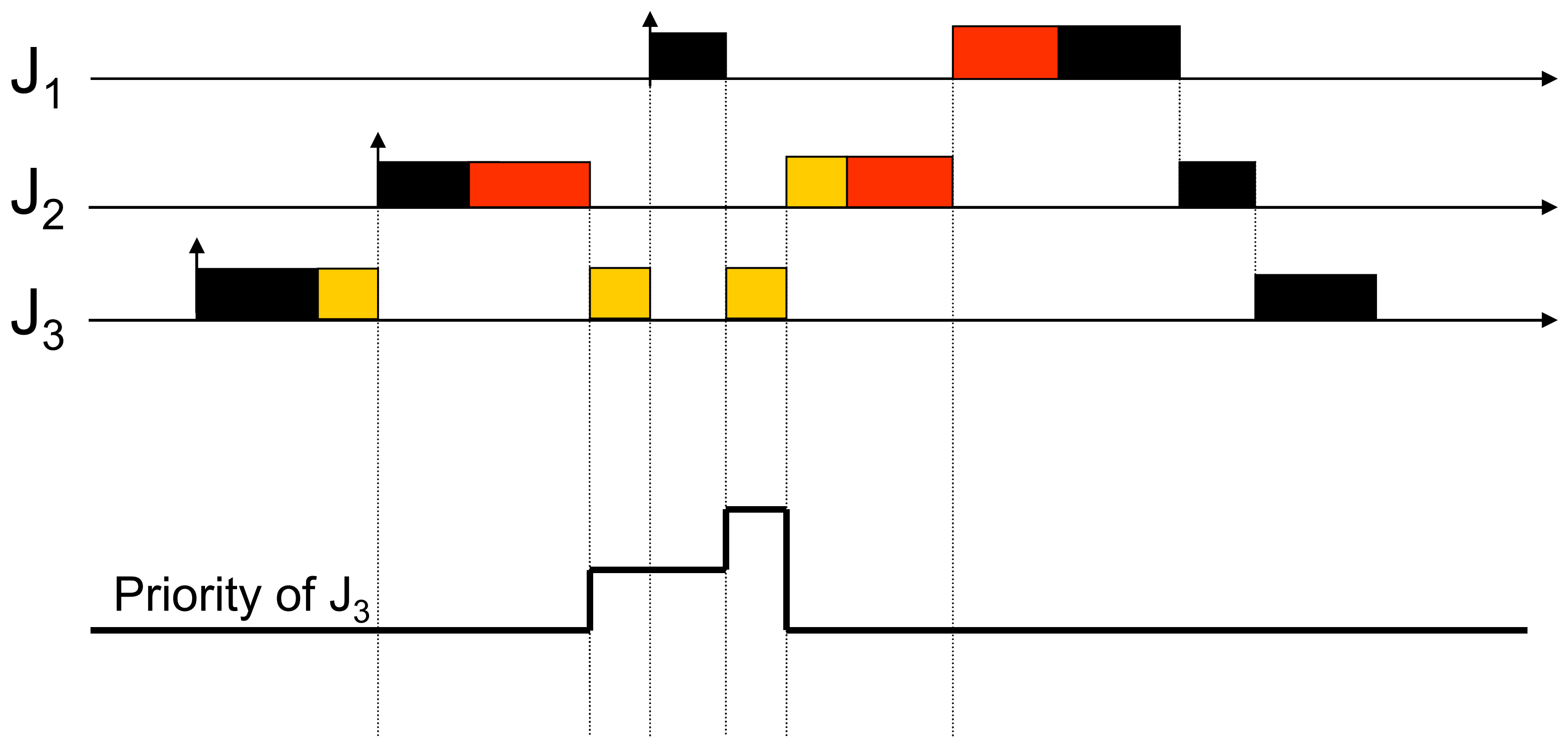
- If a task J_h blocks, since another task J_l with lower priority owns the requested resource, then J_l inherits the priority of J_h.
- When J_l releases the resource, the priority inheritance from J_h is undone.
- Rule: Tasks always inherit the highest priority of tasks blocked by it.

Direct vs. push-through blocking

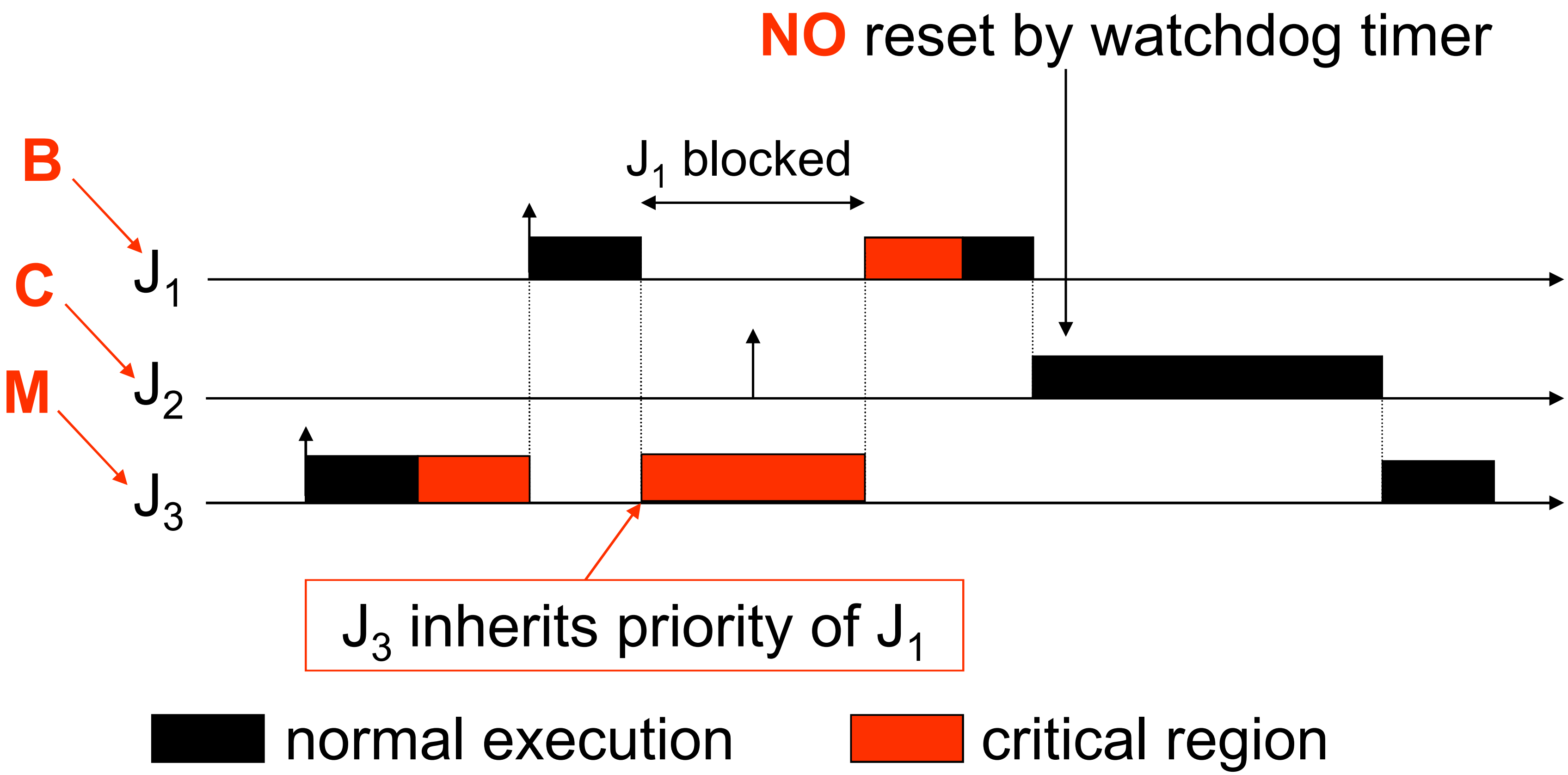
- **Direct blocking:** High-priority job tries to acquire resource already held by lower-priority job
- **Push-through blocking:** Medium-priority job is blocked by lower-priority job that has inherited a higher priority.



Transitive priority inheritance



Priority inheritance for the Pathfinder example

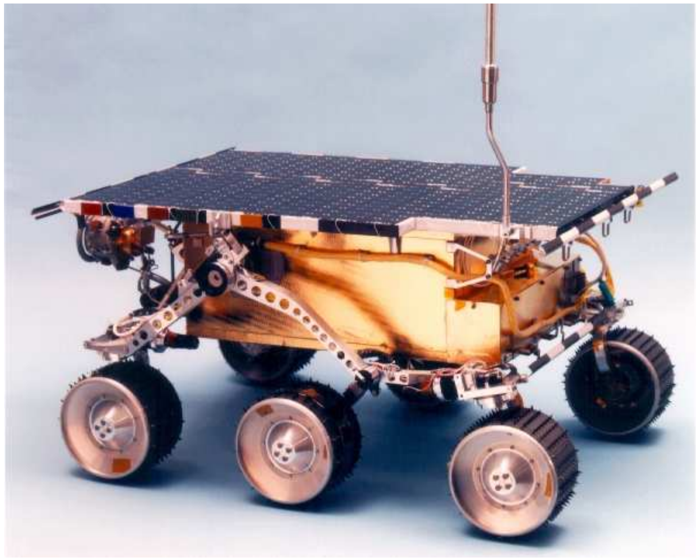


$$\text{priority}(J_1) > \text{priority}(J_2) > \text{priority}(J_3)$$

Priority inversion on Mars

- Priority inheritance also solved the Mars Pathfinder problem:
  - the VxWorks operating system used in the pathfinder implements a flag for the calls to mutual exclusion primitives.
  - This flag allows priority inheritance to be set to “on”.
  - When the software was shipped, it was set to “off”.

The problem on Mars was corrected by using the debugging facilities of VxWorks to change the flag to “on”, while the Pathfinder was already on the Mars [Jones, 1997].



Schedulability check

Let B_i be the maximum blocking time due to lower-priority jobs that a job J_i may experience.

$\forall$ i: $R_i^{(0)} = C_i$

repeat

$\forall$ i: $R_i^{(j+1)} = C_i + B_i + \sum_{k=1}^{i-1} \lceil R_i^{(j)} / T_k \rceil \cdot C_k$

until ($\exists$ i with $R_i^{(j+1)} > D_i$) **or** ($\forall$ i $R_i^{(j+1)} = R_i^{(j)}$);

if ($\forall$ i $R_i^{(j+1)} = R_i^{(j)}$) **then**

report("RM schedulable");

Blocking Time Computation

- Precise algorithm based on exhaustive search: exponential cost

- Here: approximative solution
- Assumption: no nested critical sections

Lemma: Transitive priority inheritance can only occur in the presence of nested critical sections.

Blocking Time

priority ceiling C(S)=priority of the highest-priority job that can lock S

Theorem: In the absence of nested critical sections,
a critical section of job J guarded by semaphore S
can only block job J‘
if priority(J) < priority(J‘) $\leq$ C(S).

Blocking Time

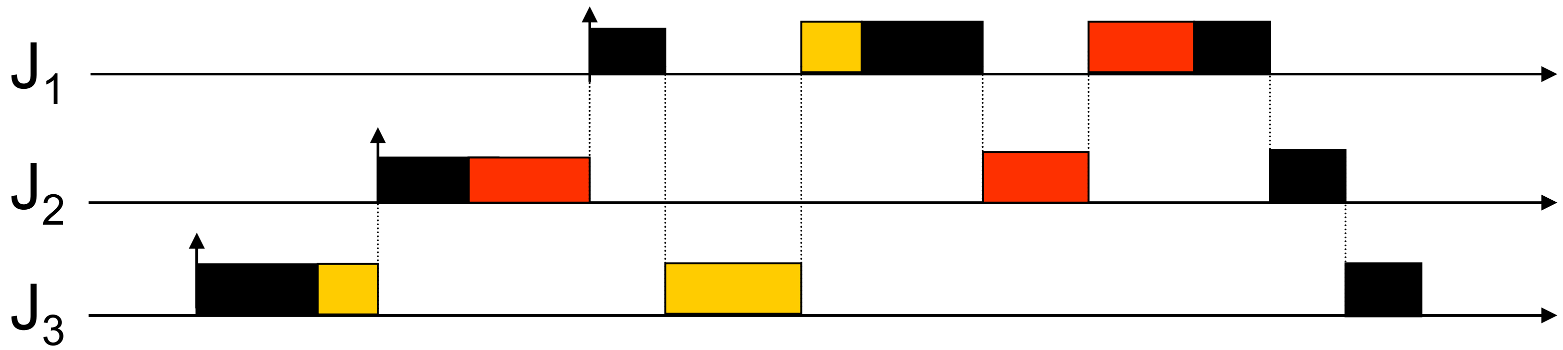
- $D_{j,k}$: duration of longest critical section of task τ_j, guarded by semaphore S_k

- Blocking Time

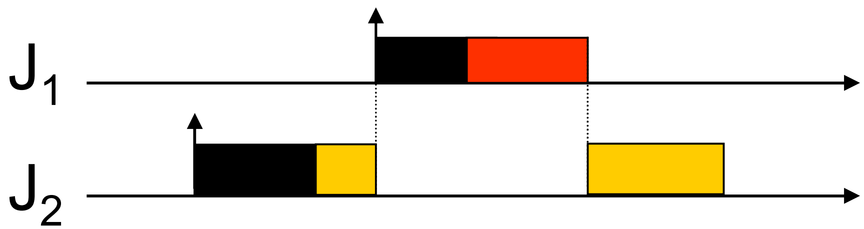
  - $B_i \leq \sum^{n}_{j=i+1} \max_k[D_{j,k} : C(S_k) \geq P_i]$
  - $B_i \leq \sum^{m}_{k=1} \max_{j>i}[D_{j,k} : C(S_k) \geq P_i]$

  where the task set consists of n periodic tasks that use m distinct semaphores.

Problem: Chained Blocking



Problem: Deadlock



J_1:

wait(S_a)

wait(S_b)

signal(S_b)

signal(S_a)

J_2:

wait(S_b)

wait(S_a)

signal(S_a)

signal(S_b)

Priority Ceiling Protocol

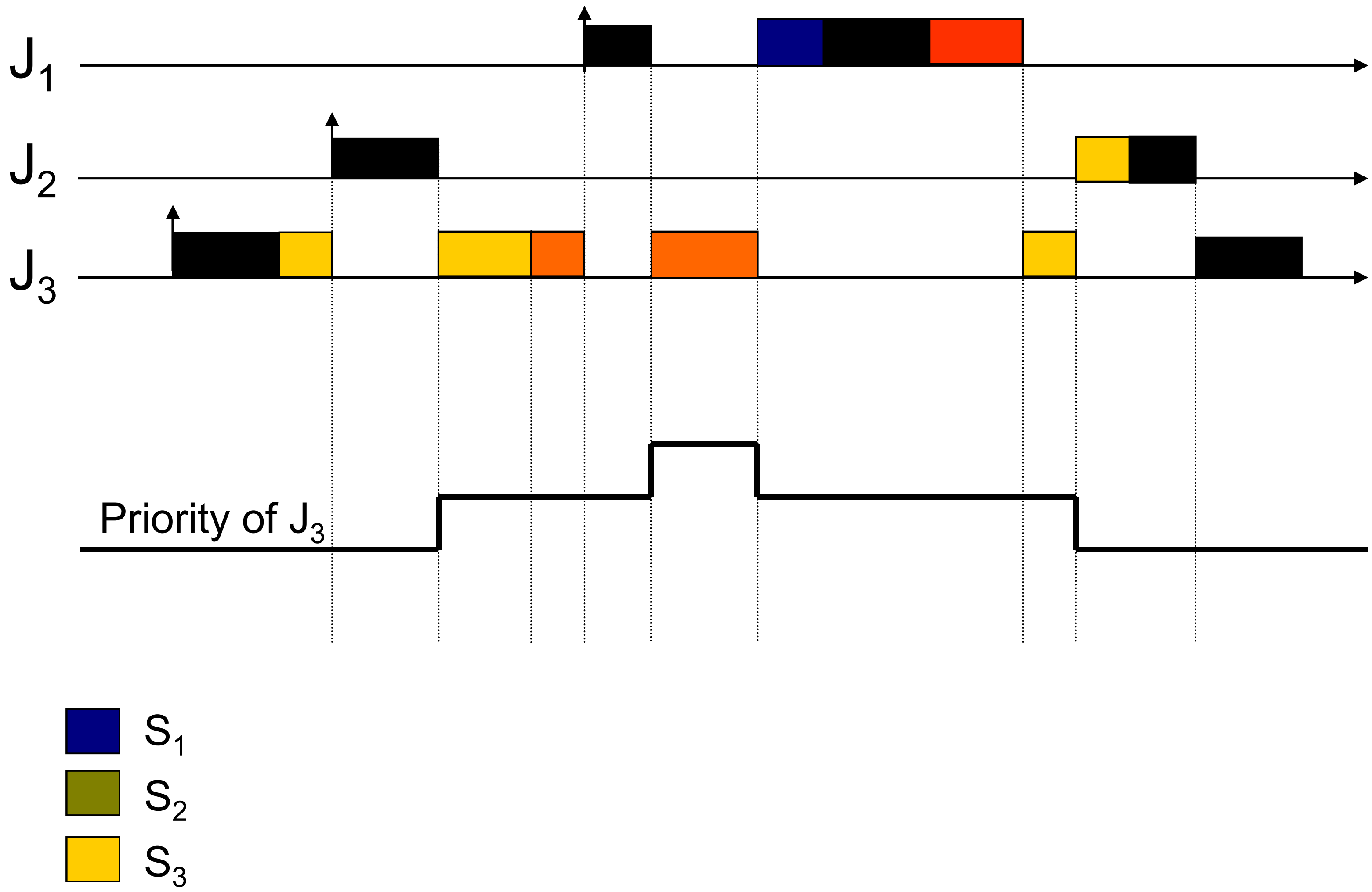
- Each semaphore S is assigned a **priority ceiling**:
  C(S)=priority of the highest-priority job that can lock S

- The processor is assigned to a ready job J with highest priority.

- To enter a critical section, J needs priority > C(S*),
  where S* is the currently locked semaphore with max C.
  $\rightarrow$ otherwise J „blocks on semaphore“ and
  priority of J is inherited by job J‘ holding S*.

- When J‘ exits critical section, its priority is updated to the highest priority of some job that is blocked by J‘ (or to the nominal priority if no such job exists).

Example



Priority Ceiling Protocol

Theorem (Sha/Rajkumar/Lehoczky): Under the Priority Ceiling Protocol, a job can be blocked for at most the duration of one critical section.

Priority Ceiling Protocol

The Priority Ceiling Protocol prevents deadlocks.