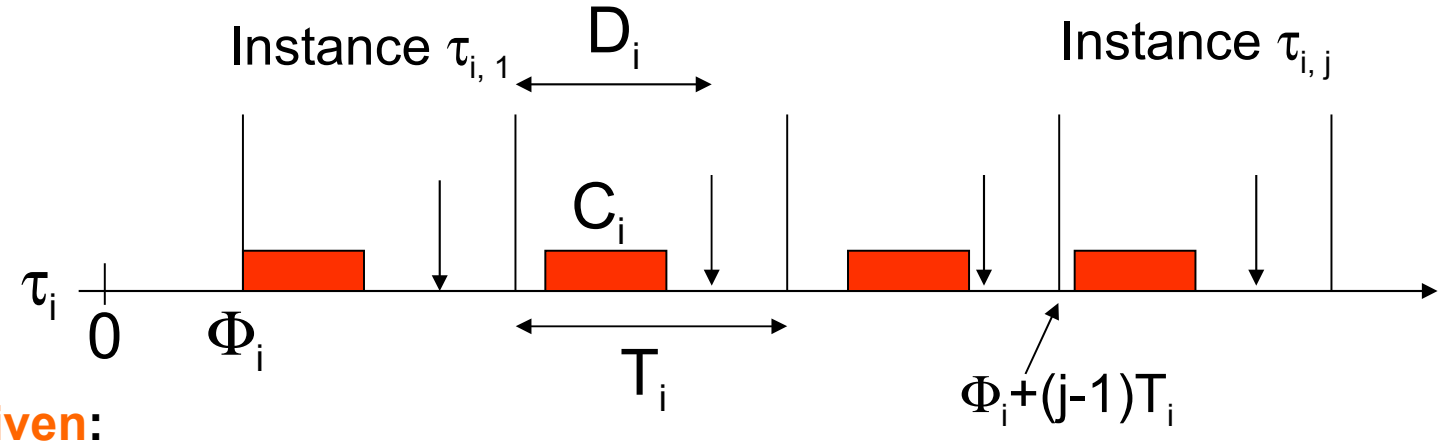
Embedded Systems 16

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.
- Both will be relaxed in the following two lectures.

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 $U_{bnd}(A) = \inf \{U(\Gamma) \mid$ ilable 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 τ_1, ..., τ_n with $D_i = T_i$ is schedulable with EDF iff $U \leq 1$.**

- 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.

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 $U \leq 1$.

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 the same worst case execution time C_i.

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

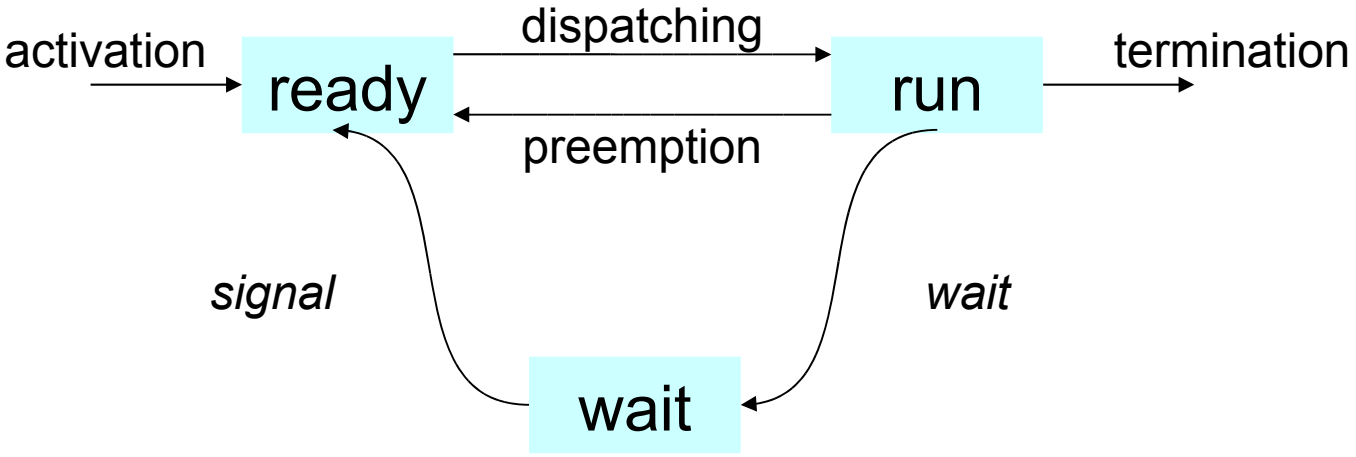
A.4. All tasks in Γ are independent. No precedence relation, no resource constraints.

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

- Basic results based on these assumptions form the core of scheduling theory.
- For practical applications, assumptions A.3. A.4, and A.5. can be relaxed, but results have to be extended.

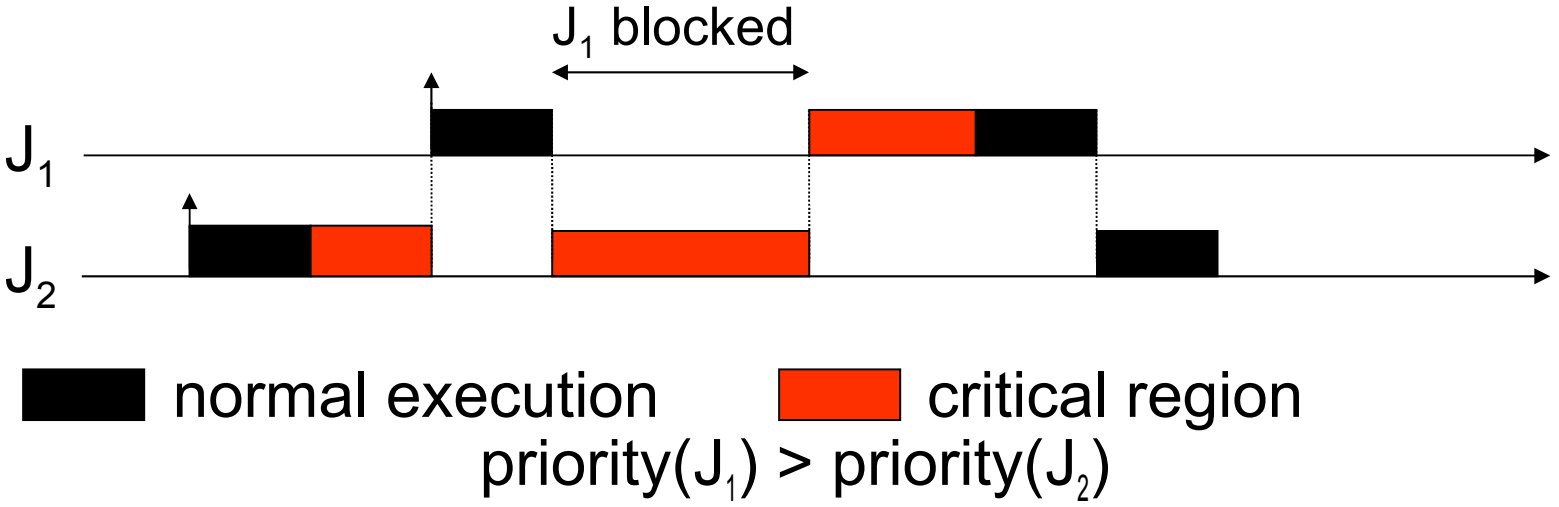
Wait state caused by resource constraints

- 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



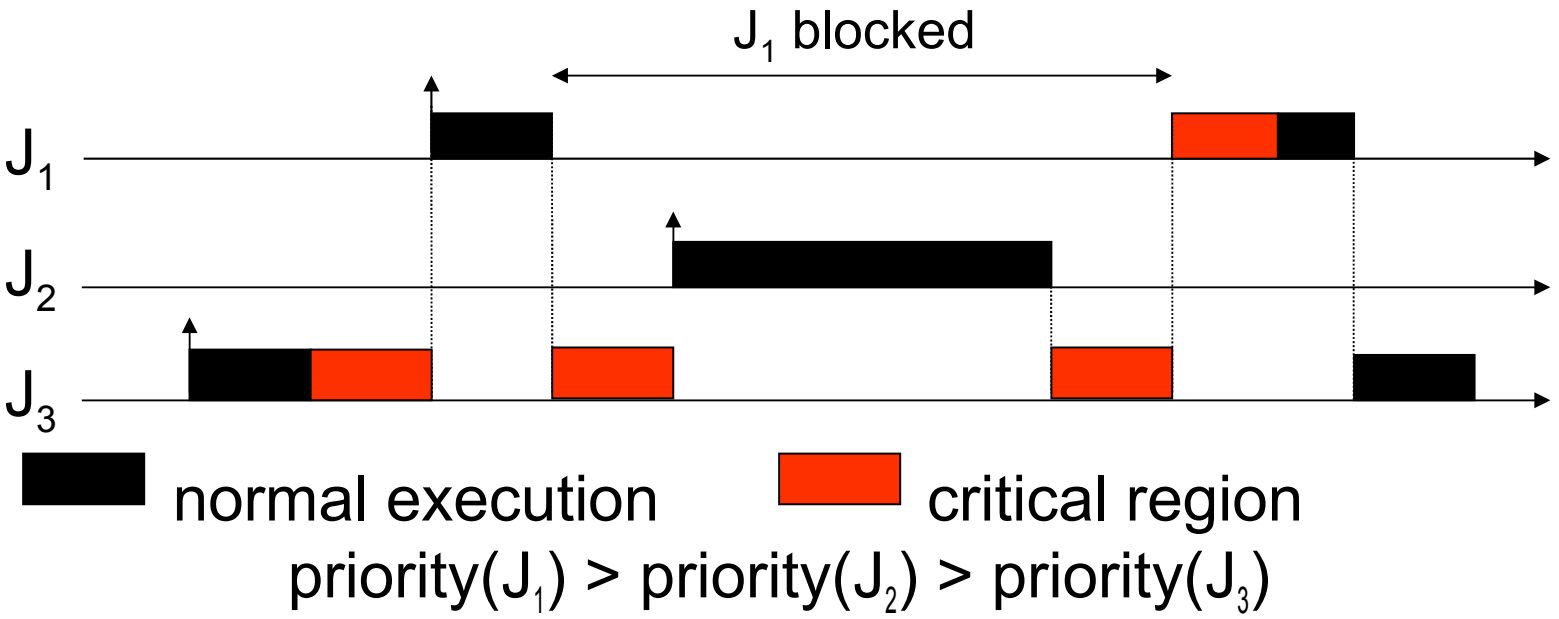
The priority inversion problem

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



- Here: maximum blocking time of J_1 equal to length of critical section of J_2.

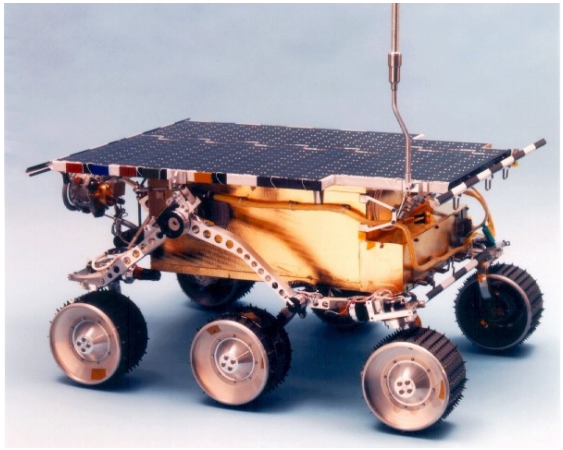
The priority inversion problem



- Blocking time of J_1 equal to length of critical section of J_3 + 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".” …



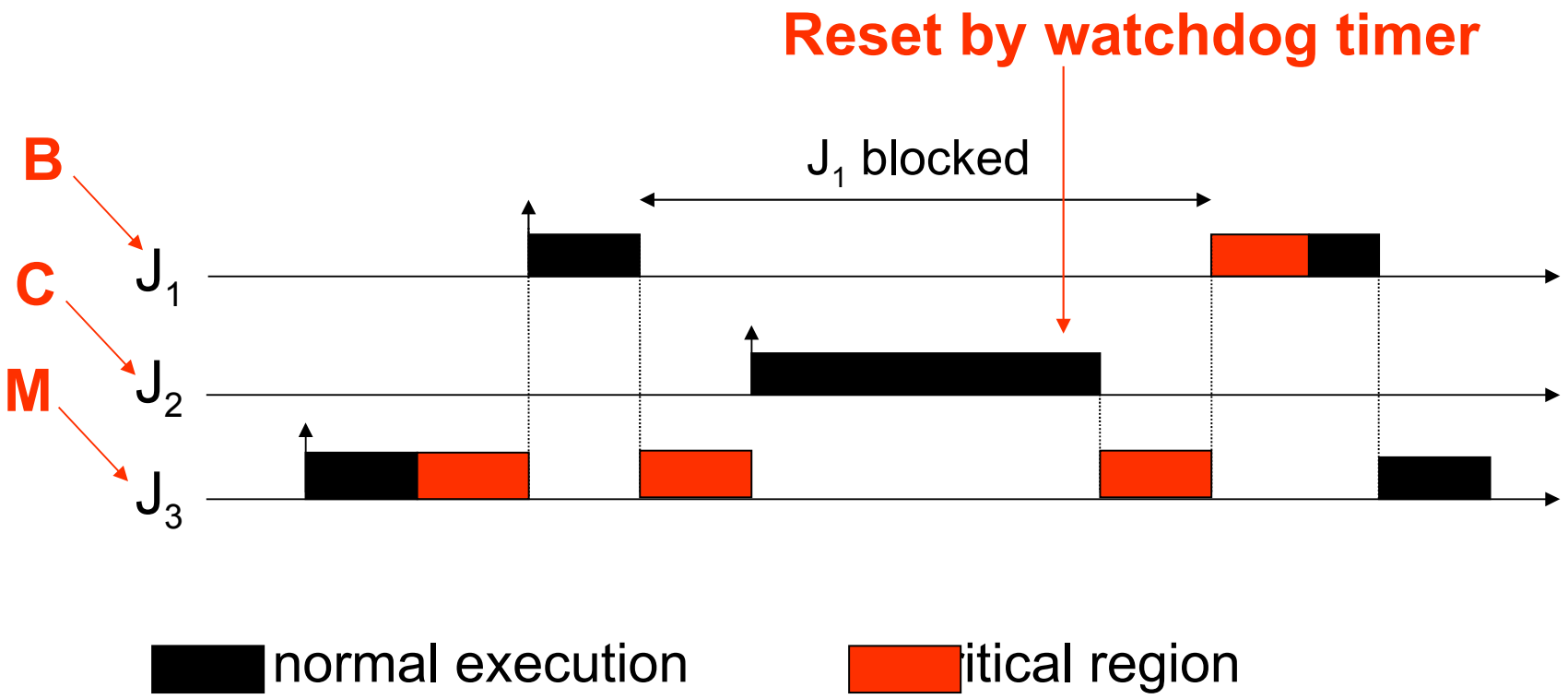
Priority inversion in real life: The MARS Pathfinder problem (2)

- 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

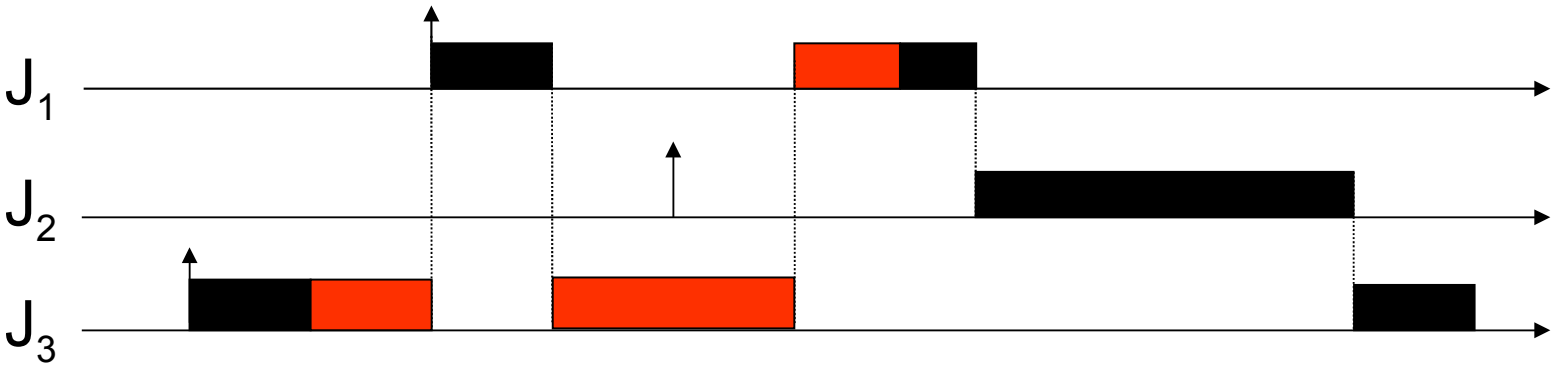
What are the options?

Idea of **priority inheritance protocol**:

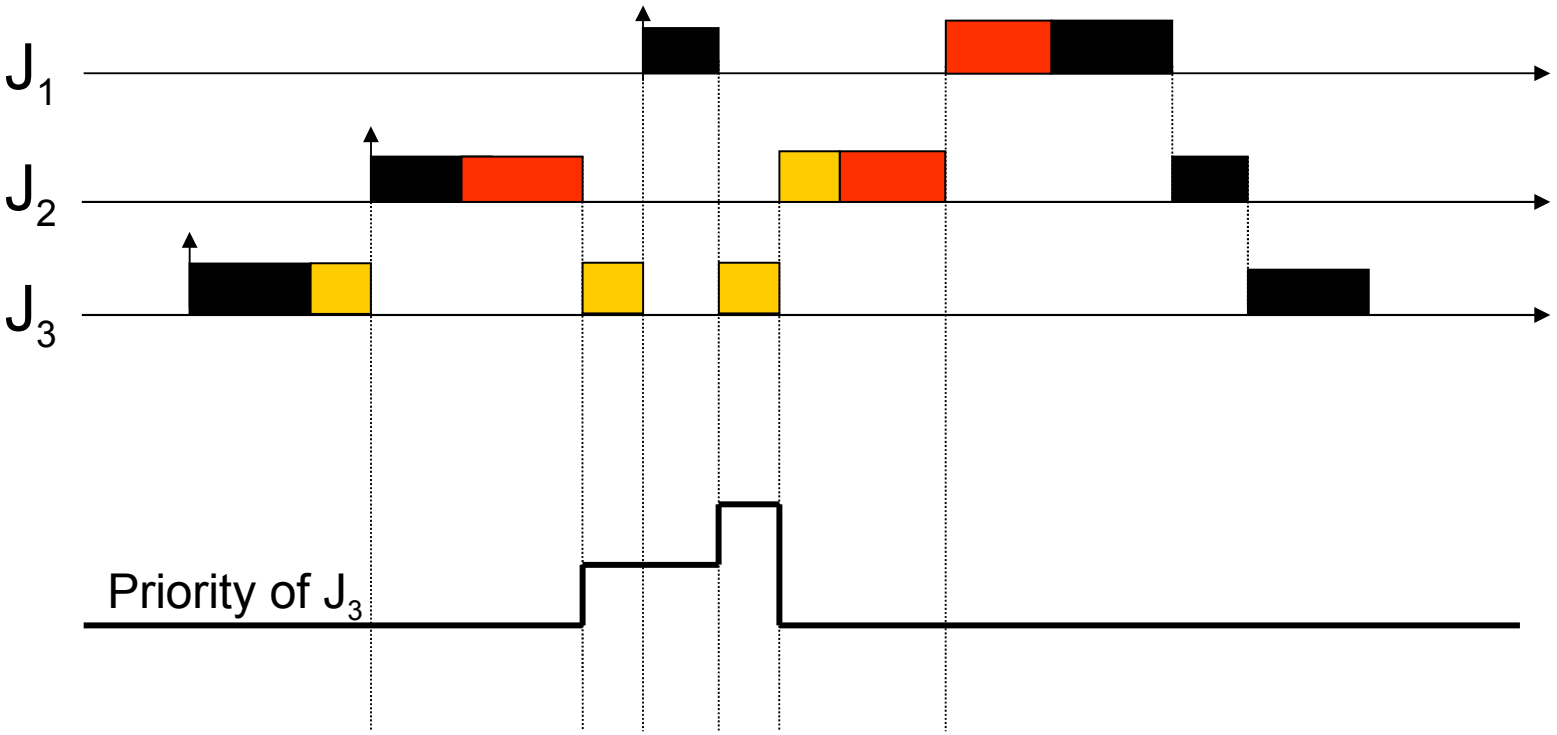
- If a task J_h blocks, since another task J_l with lower priority owns the requested resource, then J_l inherits the priority of J_h.
- When J_l releases the resource, the priority inheritance from J_h is undone.
- Rule: Tasks always inherit the highest priority of tasks blocked by them.

Direct vs. push-through blocking

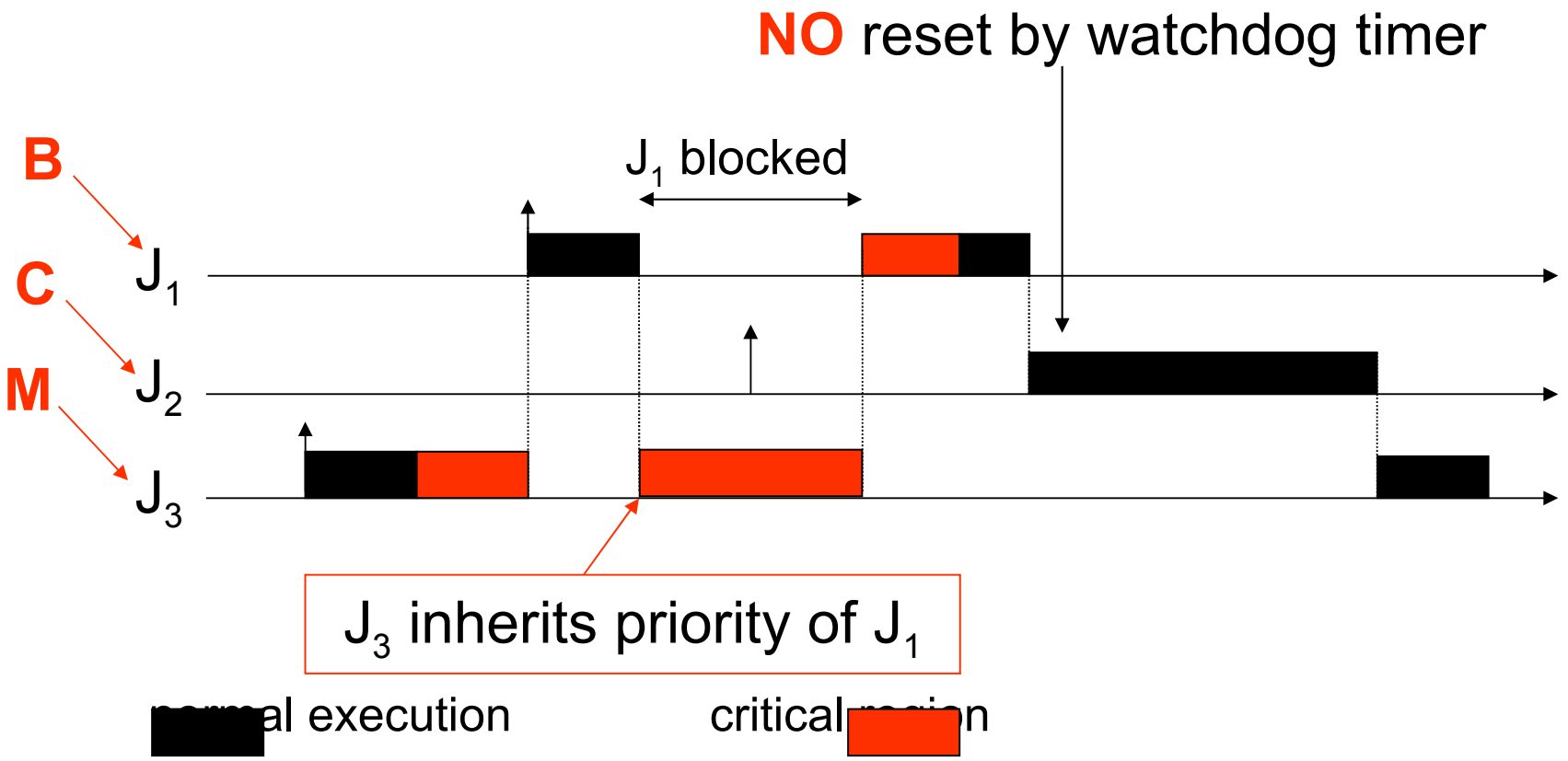
- **Direct blocking:** High-priority job tries to acquire resource already held by lower-priority job
- **Push-through blocking:** Medium-priority job is blocked by lower-priority job that has inherited a higher priority.



Transitive priority inheritance



Priority inheritance for the Pathfinder example

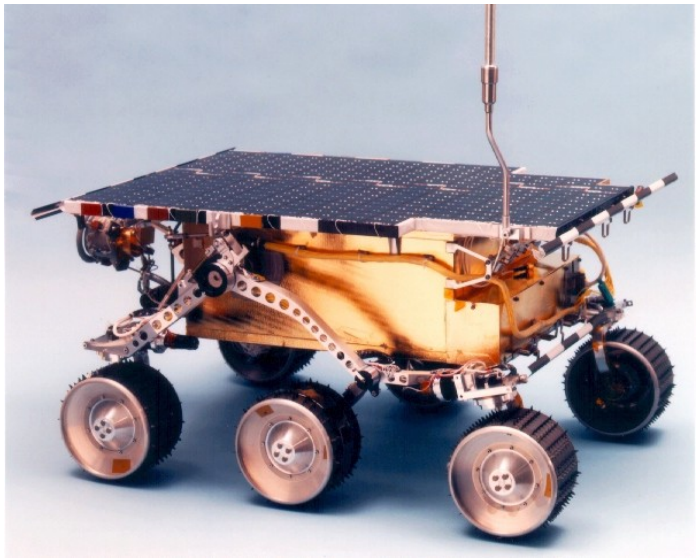


priority(J_1) > priority(J_2) > priority(J_3)

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.

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‘) ≤ C(S).

Blocking Time

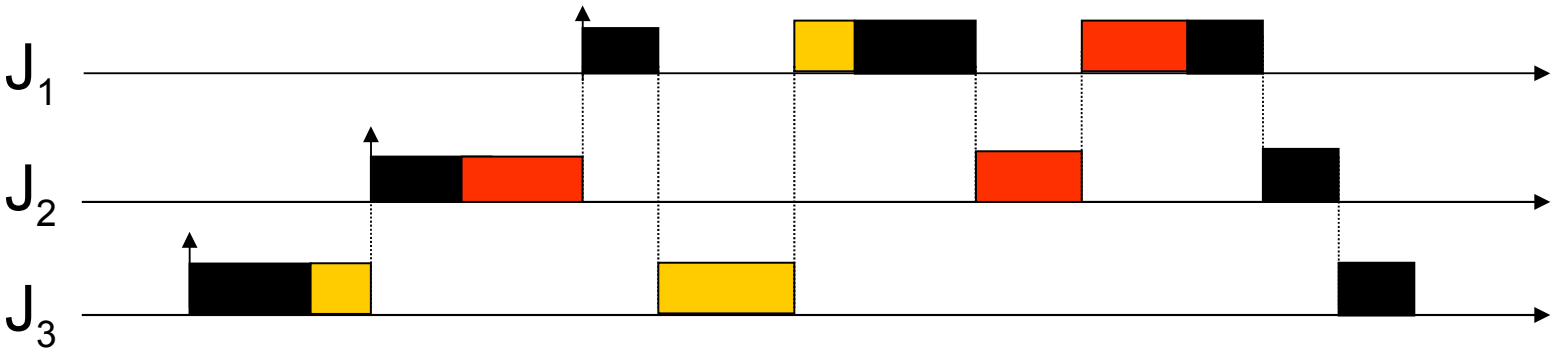
- $D_{j,k}$: duration of longest critical section of task τ_j, guarded by semaphore S_k

- Blocking Time

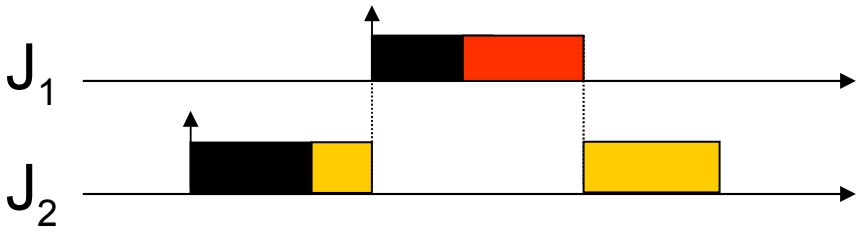
  - $B_i \leq \sum_{j=i+1}^{n} \max_k [D_{j,k} : C(S_k) \geq P_i]$
  - $B_i \leq \sum_{k=1}^{m} \max_{j>i} [D_{j,k} : C(S_k) \geq P_i]$

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

Remaining problem: Chained Blocking



Remaining 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

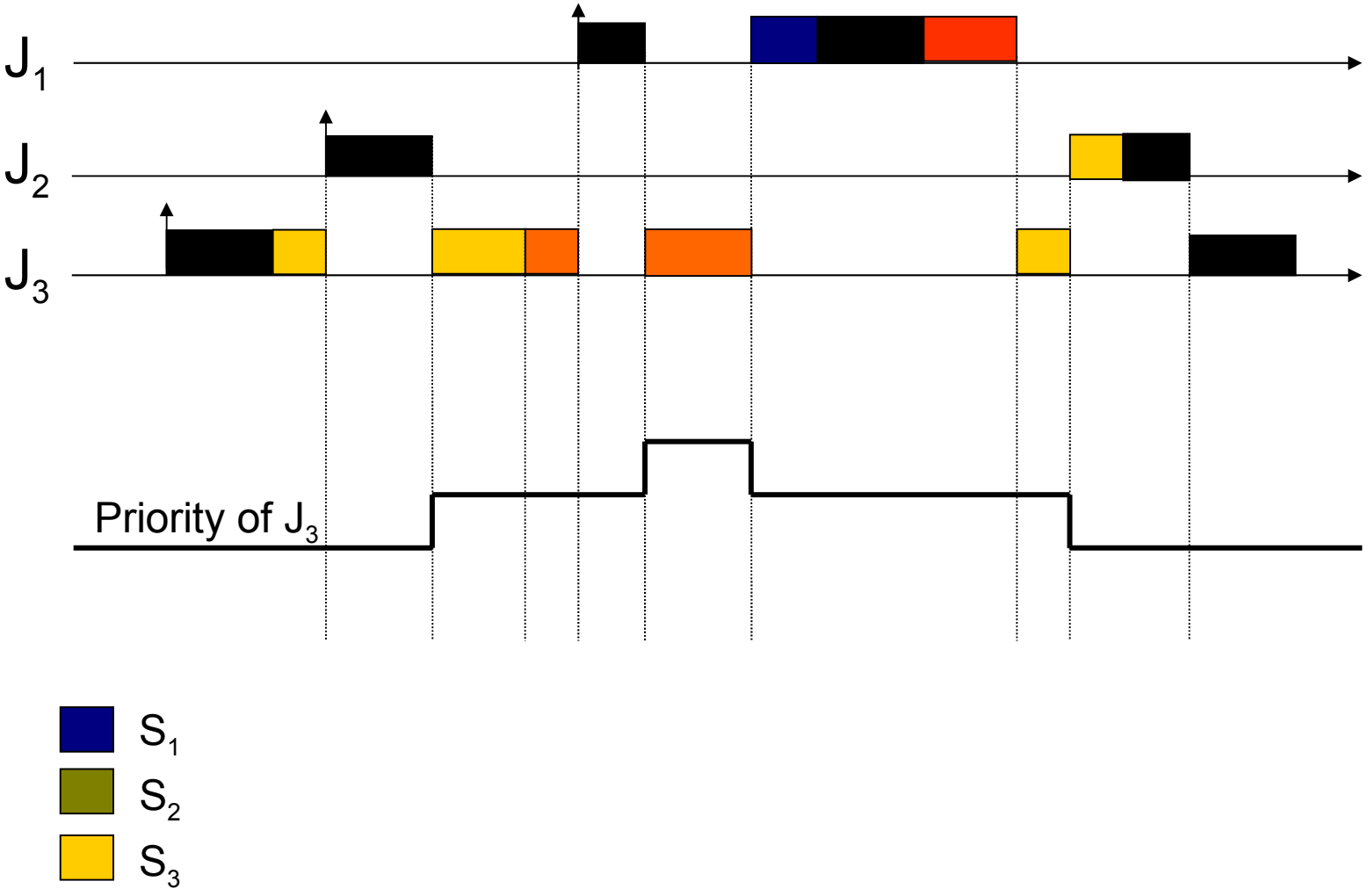
- Each semaphore S is assigned a **priority ceiling**:
  C(S)=priority of the highest-priority job that can lock S

- 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.
  → 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.

The Priority Ceiling Protocol prevents deadlocks.