Embedded Systems 17

REVIEW: Rate monotonic scheduling (RM)

- 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.

REVIEW: Optimality of Rate Monotonic Scheduling

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

- **Def.:** 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: $R_{i,j} = f_{i,j} - a_{i,j}$.
- The critical instant of a task is the time at which the arrival of the task will produce the largest response time.

REVIEW: Schedulability check

- A set of tasks can be scheduled by RM if

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

- But what can we say about schedulability when processor utilization factor is larger than $n(2^{1/n} - 1)$?

- 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

- Compute an upper bound R_i on the response time:
  - Suppose that $\tau_1, \ldots, \tau_n$ 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

- 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

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");

Example

	τ_1	τ_2	τ_3	τ_4
T_i	4	5	6	11
C_i	1	1	2	1
D_i	3	4	5	10

Example

	τ_1	τ_2	τ_3	τ_4
T_i	4	5	6	11
C_i	1	1	2	1
D_i	3	4	5	10

$R_1^0 = 1$, $R_2^0 = 1$, $R_3^0 = 2$, $R_4^0 = 1$

$R_2^1 = 2$, $R_3^1 = 4$, $R_4^1 = 5$

$R_2^2 = 2$, $R_3^2 = 4$, $R_4^2 = 6$

$R_4^3 = 7$

$R_4^4 = 9$

$R_4^5 = 10$

$R_4^6 = 10$

$\Rightarrow$ RM-schedulable.

Summary

- Problem of scheduling independent and preemptable periodic tasks

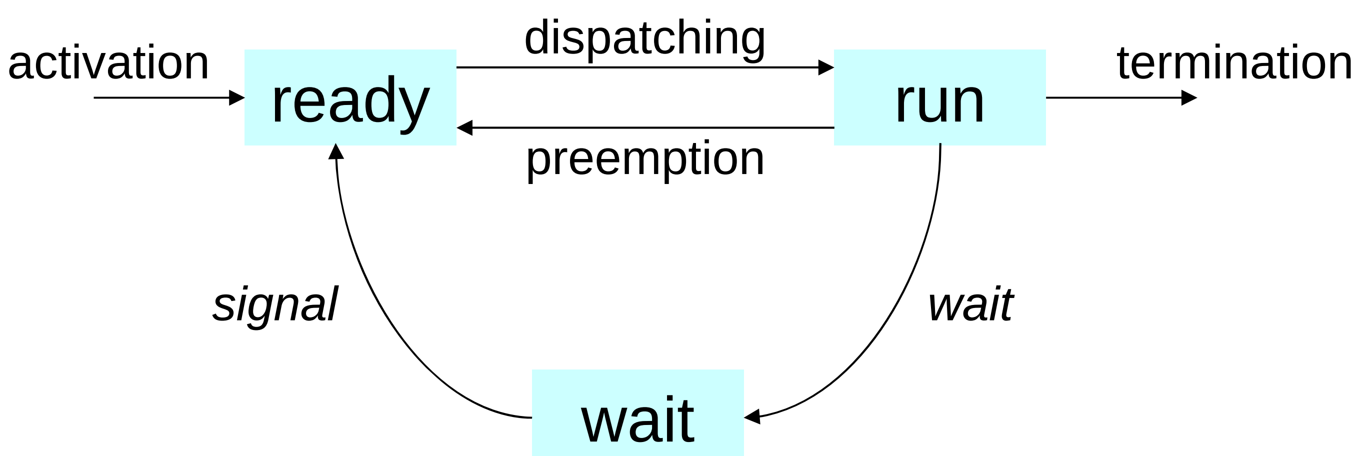
- **Rate monotonic scheduling:**
  - Optimal solution among all fixed-priority schedulers
  - Schedulability of n tasks guaranteed, if processor utilization $U \leq n(2^{1/n} - 1)$.

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

Rate Monotonic Scheduling in Presence of Task Dependencies

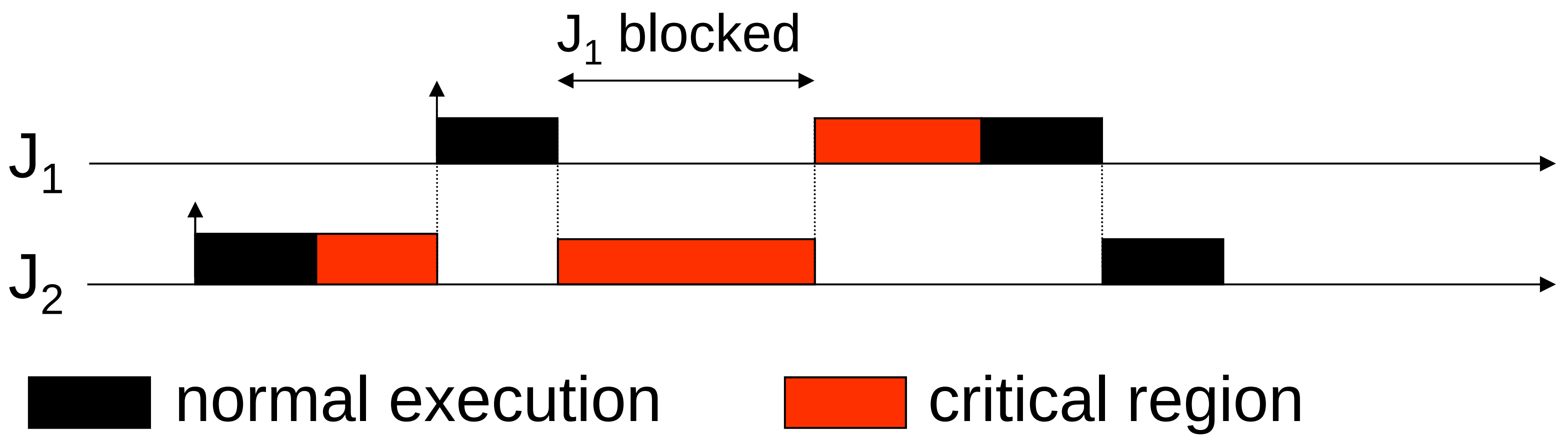
Wait state caused by resource constraints

- 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

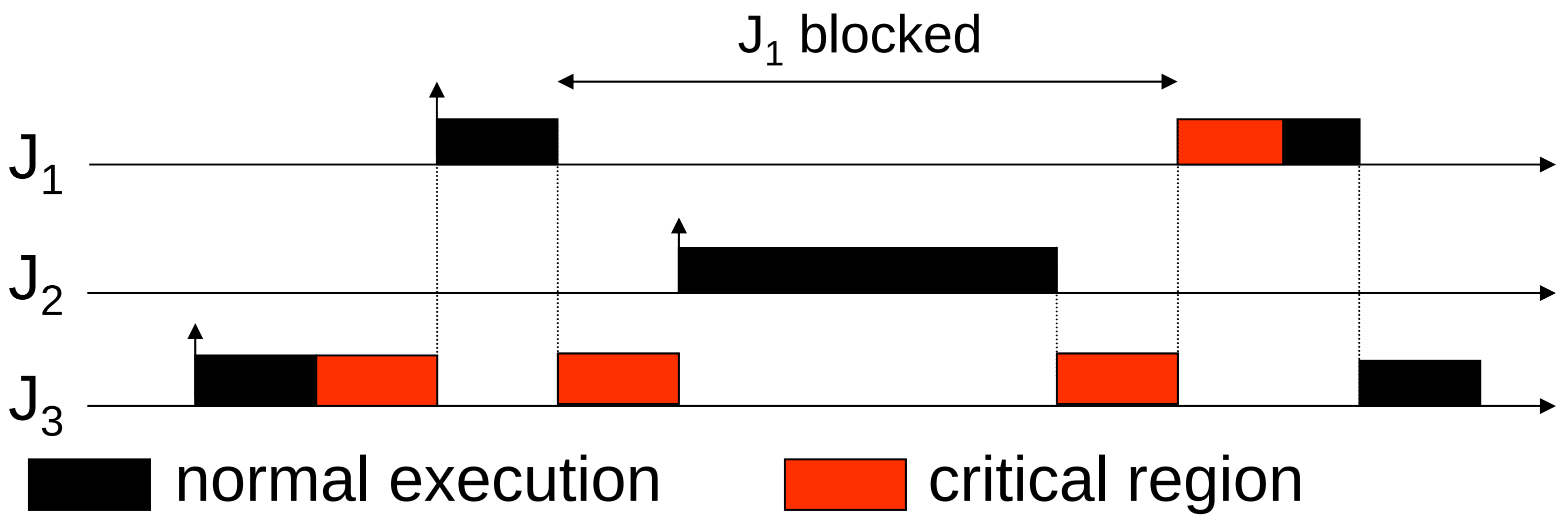
- **Priority inversion** can occur due to resource conflicts (exclusive use of shared resources) in fixed priority schedulers like RM:



priority(J_1) > priority(J_2)

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

The priority inversion problem

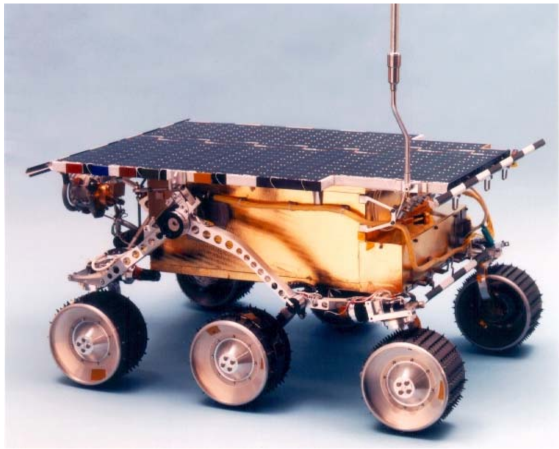


priority(J_1) > priority(J_2) > priority(J_3)

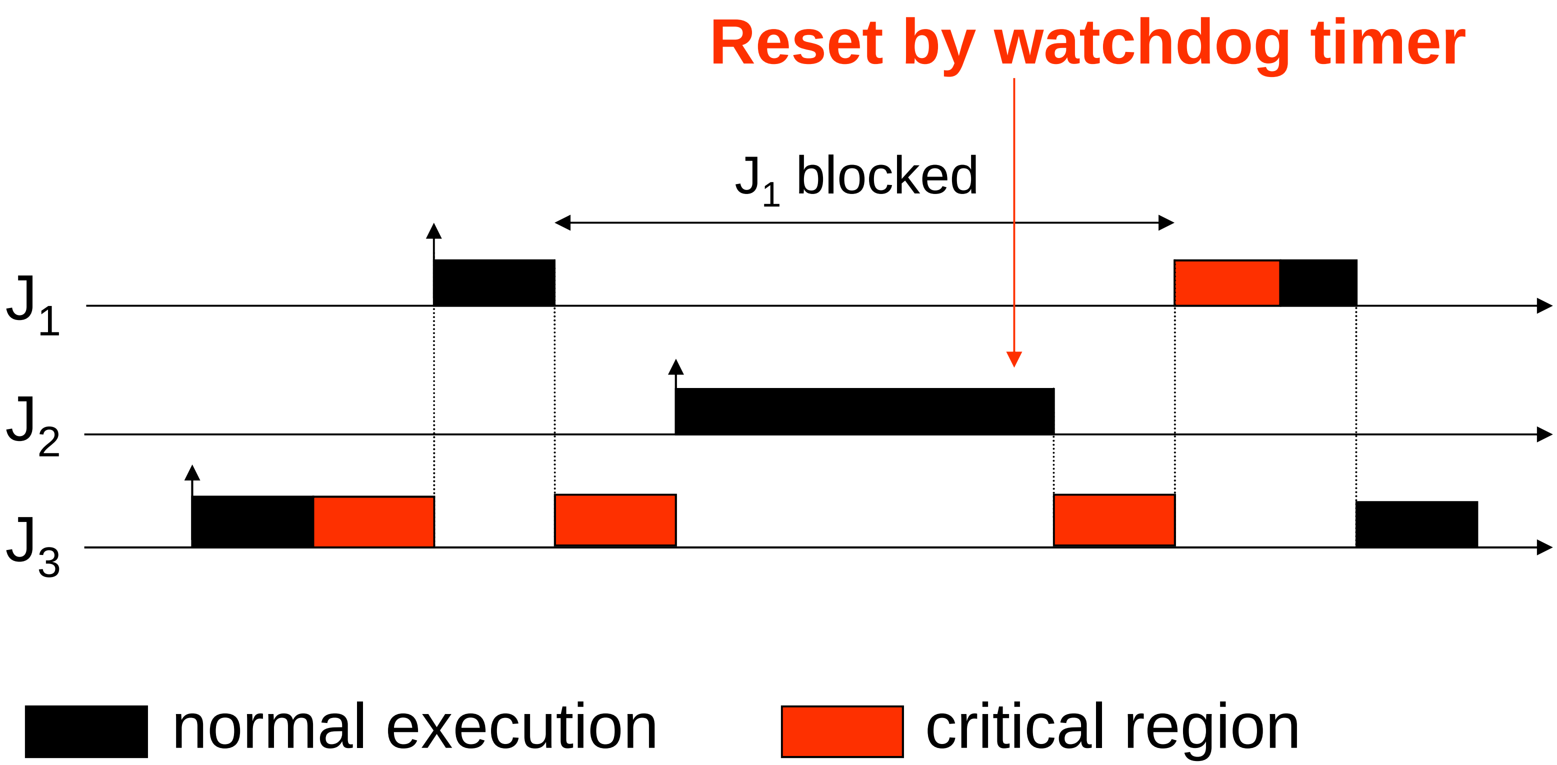
- Blocking time equal to length of critical section + computation time of J_2.
- Unbounded time of priority inversion, if J_3 is interrupted by tasks with priority between J_1 and J_3 during its critical region.

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



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

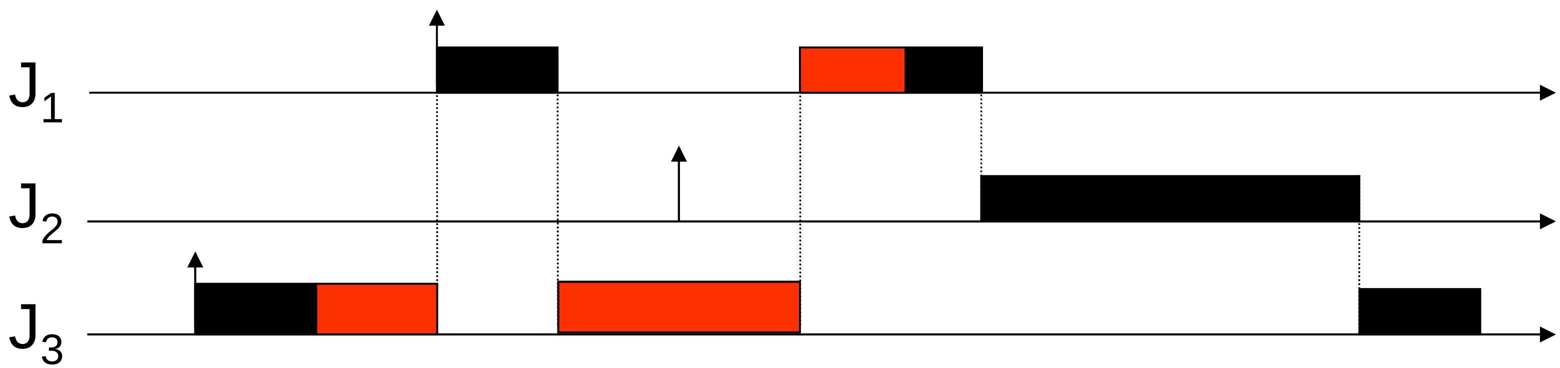
Coping with priority inversion: The priority inheritance protocol

Idea of **priority inheritance protocol**:

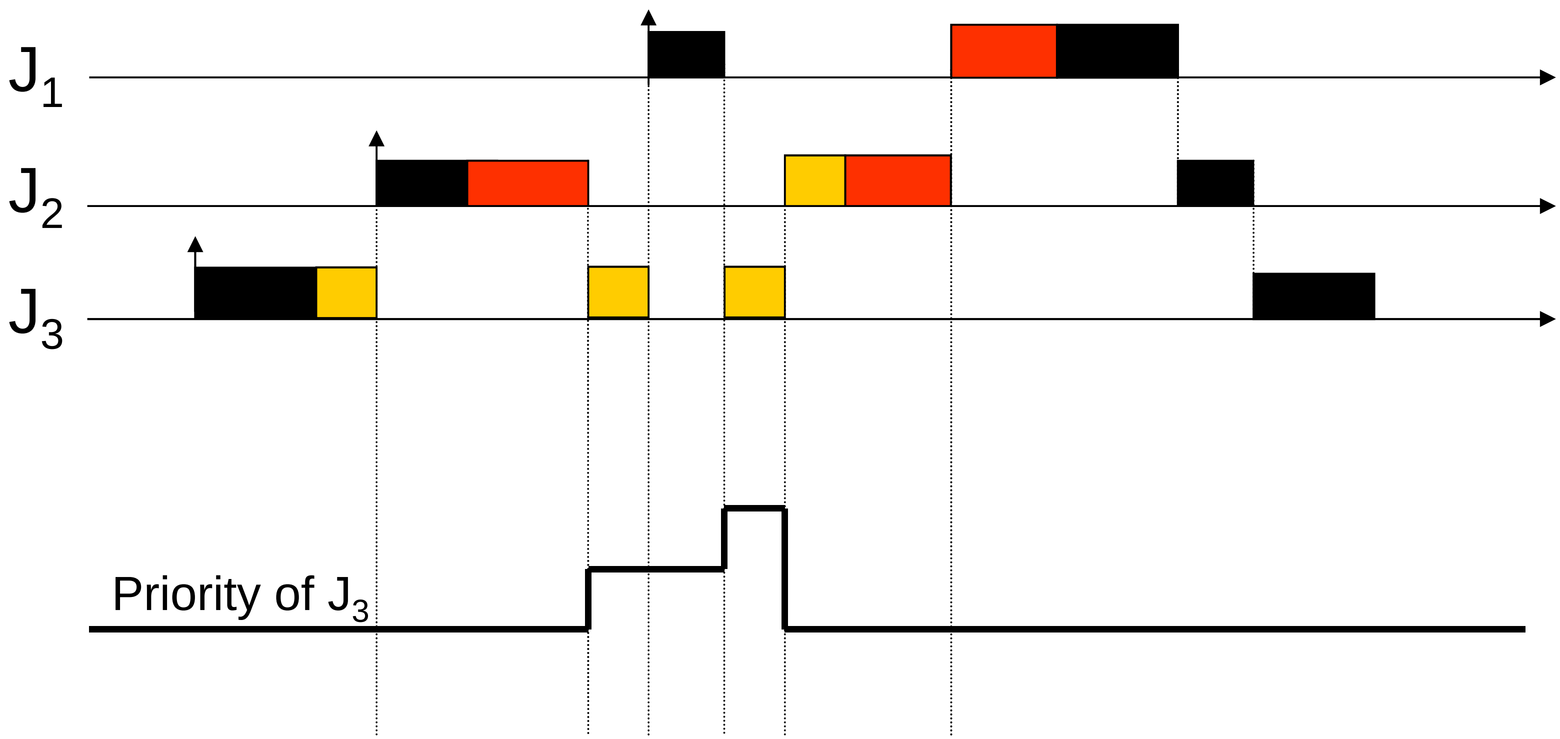
- If a task J_h blocks, since another task J_l with lower priority owns the requested resource, then J_l inherits the priority of J_h.
- When J_l releases the resource, the priority inheritance from J_h is undone.
- **Rule:** Tasks always inherit the highest priority of tasks blocked by it.

Direct vs. push-through blocking

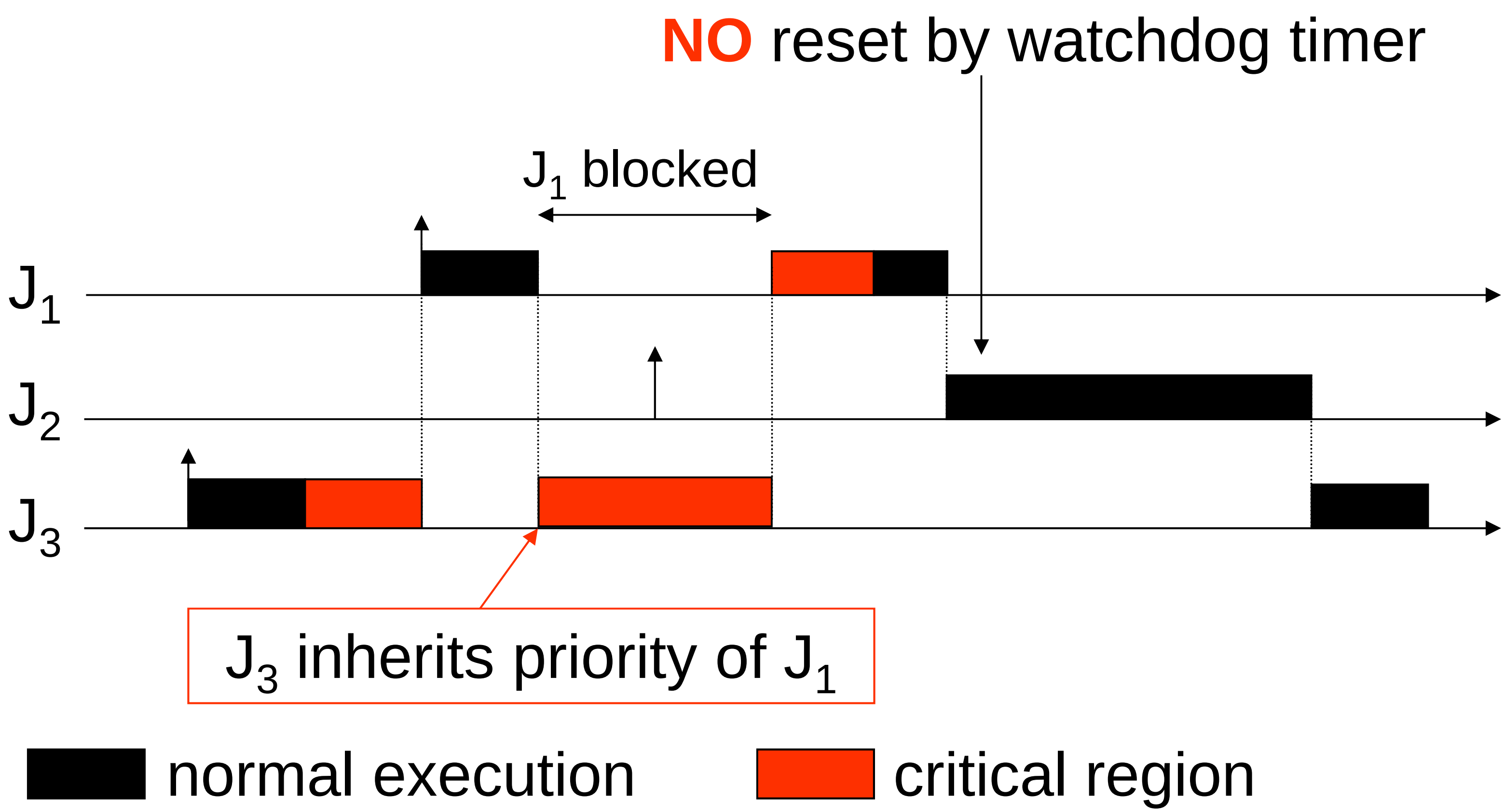
- **Direct blocking:** High-priority job tries to acquire resource already held by lower-priority job
- **Push-through blocking:** Medium-priority job is blocked by lower-priority job that has inherited a higher priority.



Transitive priority inheritance



Priority inheritance for the Pathfinder example

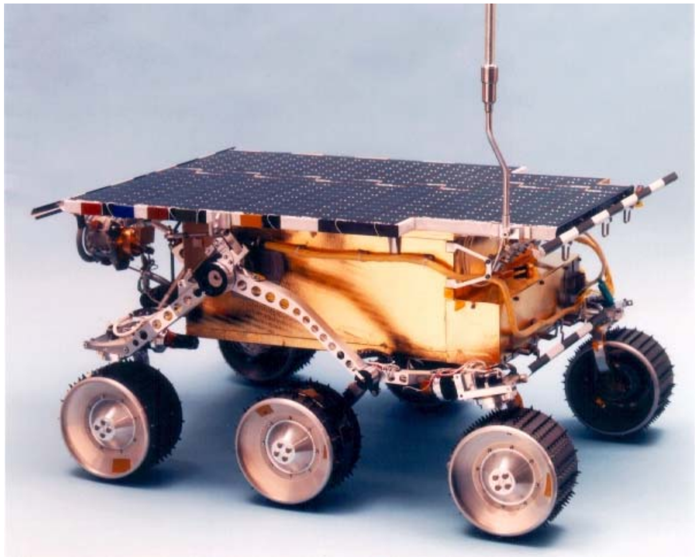


priority(J_1) > priority(J_2) > priority(J_3)

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`

$\quad\forall$ 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`**

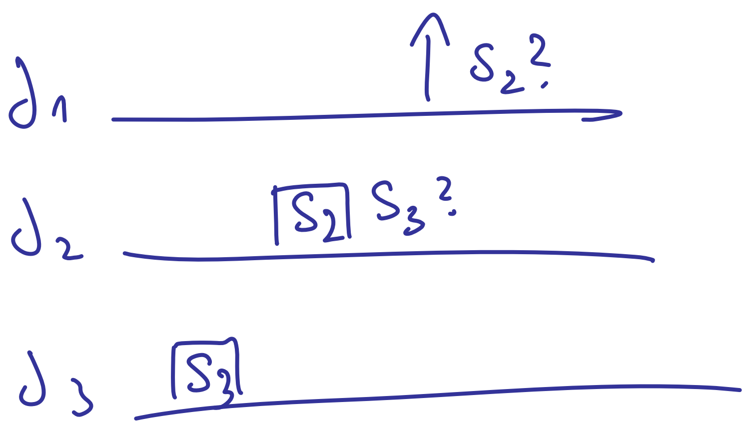
`report`(“RM schedulable”);

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.



Suppose J2 blocks J1, J3 blocks J2;
J2 holds S2
J3 holds S3
⇒ J2 attempted to lock S3 inside the critical region protected by S2.

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') ≤ C(S).

priority (J) < priority (J'):
otherwise J' cannot preempt J

priority (J') ≤ C(S)
any job that uses the S
cannot have a priority > 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.

Example

D_{ik}	S_a	S_b	S_c
τ_1	1	1	*
τ_2	*	8	2
τ_3	7	6	*
τ_4	5	4	3

D_{ik} = * : task τ_i does not use semaphore S_k

$$\left.\begin{array}{l} B_1 \le 8+7+5 = 20 \\ B_1 \le 7+8 = 15 \end{array}\right\} B_1 \le 15$$

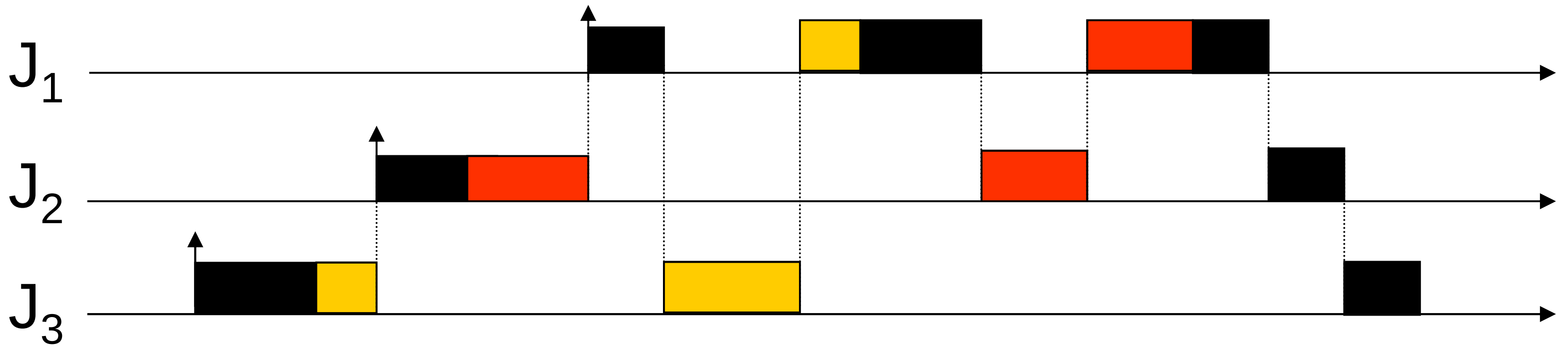
$B_2 \le 7+5 = 12$

$B_2 \le 7+6+3 = 16$

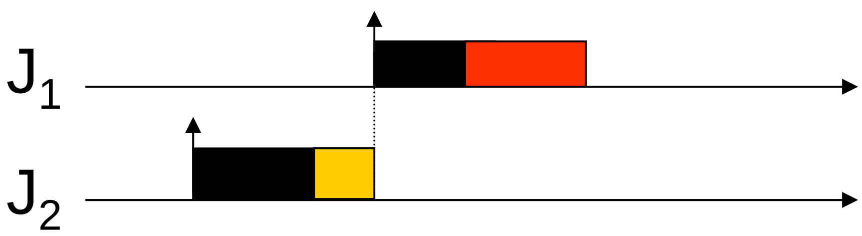
$$\left.\begin{array}{l} B_3 \le 5 \\ B_3 \le 5+4+3 = 12 \end{array}\right\} 5$$

$B_4 = 0$

Problem: Chained Blocking



Problem: Deadlock



J_1:

wait(S_a)

> wait(S_b)
>
> signal(S_b)

signal(S_a)

J_2:

wait(S_b)

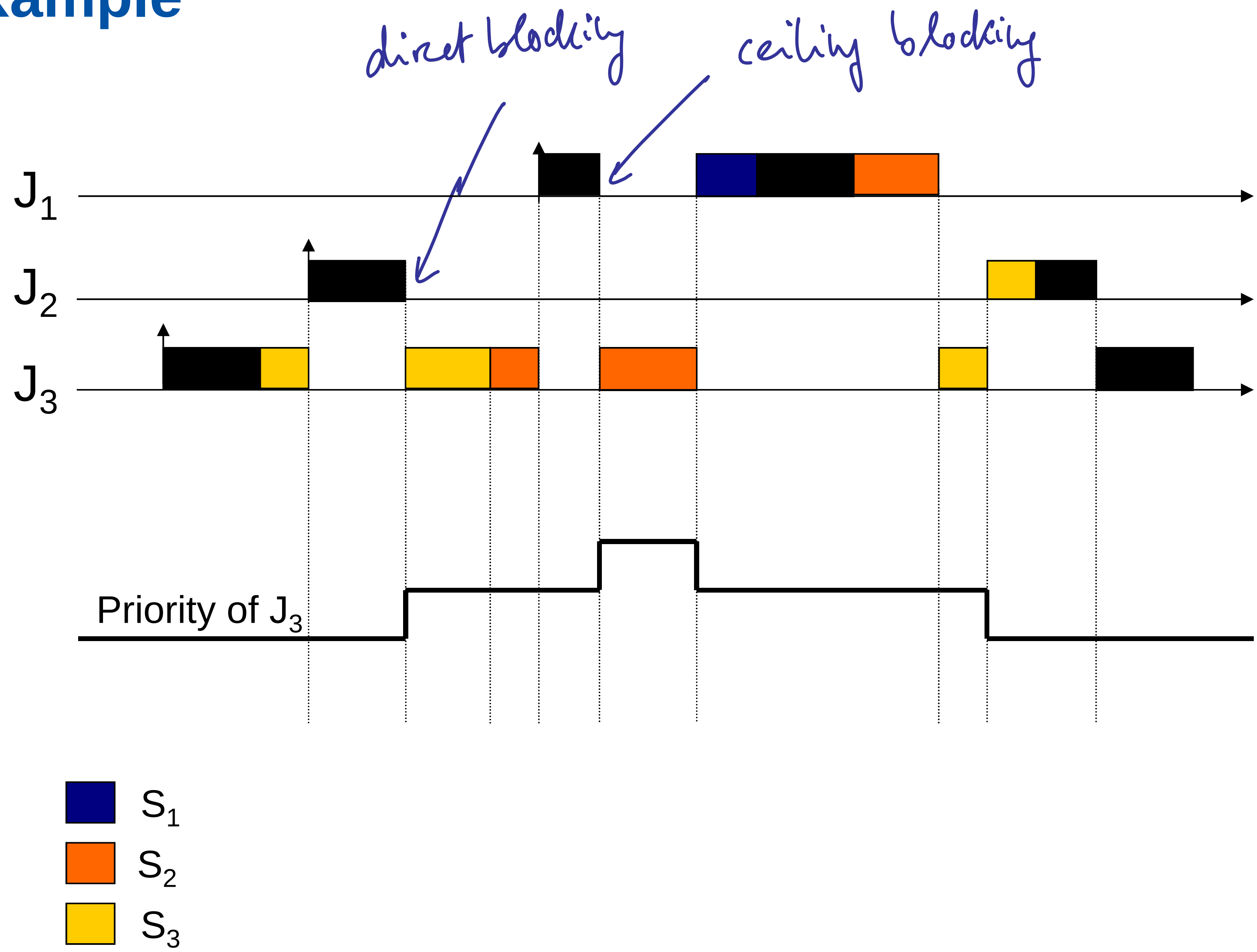
> wait(S_a)
>
> signal(S_a)

signal(S_b)

Priority Ceiling Protocol

- 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



direct blocking
ceiling blocking
J_1
J_2
J_3
Priority of J_3
S_1
S_2
S_3

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.

- Suppose τ_i is blocked by τ_1, τ_2 with lower priority.
- Let τ_1 be the task that enters the critical region first.
- Let $C_1^* = \max_{s \text{ locked by } \tau_1} C(s)$
- When τ_2 enters the critical section $P_2 > C_1^*$, $P_i \leq C_1^*$
- If τ_i is blocked by τ_1, $P_i \leq C_1^* < P_2$ ↯

Hence,

Priority Ceiling Protocol

The Priority Ceiling Protocol prevents deadlocks.

$$T_1 \xrightarrow[\text{held by } T_2]{T_1 \text{ waits for } S_1} T_2 \xrightarrow[\text{held by } T_3]{T_2 \text{ waits for } S_2} T_3 \longrightarrow \dots \longrightarrow T_n$$

Let S_j be the last semaphore acquired to close the circle
$\Rightarrow$ task $J_{j \oplus 1}$ must have priority $> C(S_k)$
$\forall k = 1 \ldots j-1, j+1, \ldots n$

↯ Impossible, because $J_{j \oplus 1}$ can lock $S_{j \oplus 1}$