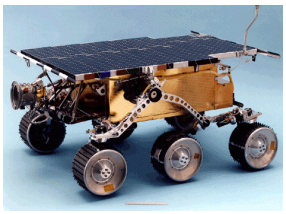
Mars Pathfinder Incident

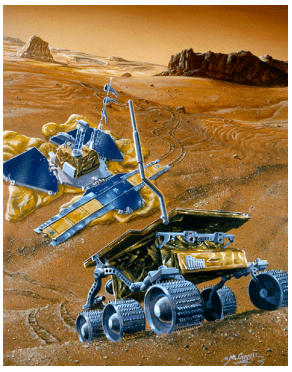
- Landing on July 4, 1997
- "…experiences software glitches…"
- Pathfinder experiences repeated RESETs after starting gathering of meteorological data.

- RESETs generated by watchdog process.
- Timing overruns caused by priority inversion.

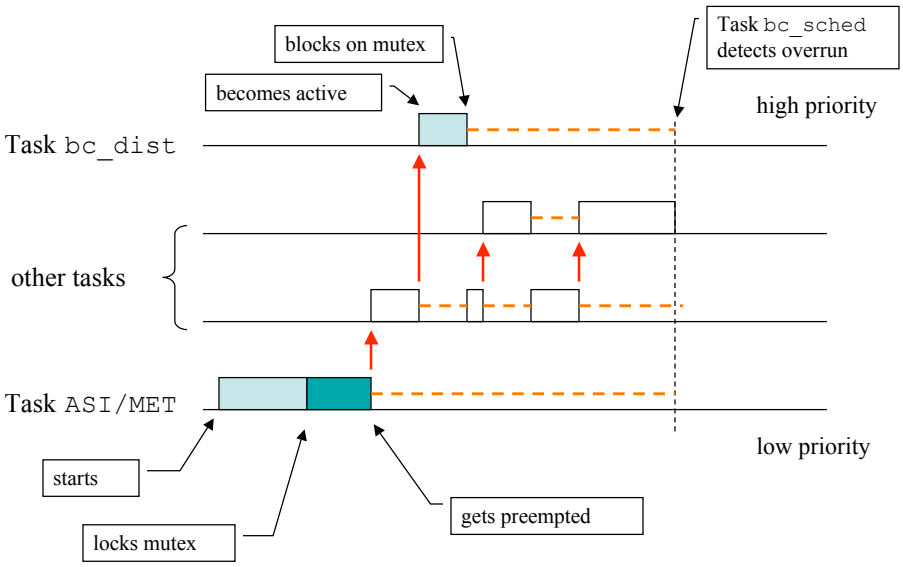
- Resources:

`research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html`





Priority Inversion on Mars Pathfinder



Mars Pathfinder: Resolution

- "Faster, better, cheaper" had NASA and JPL using "shrink-wrap" hardware (IBM RS6000) and software (Wind River VxWorks RTOS).
- Logging designed into VxWorks enabled NASA and Wind River to reproduce the failure on Earth. This reproduction made the priority inversion obvious.
- NASA patched the lander's software to enable priority inheritance.

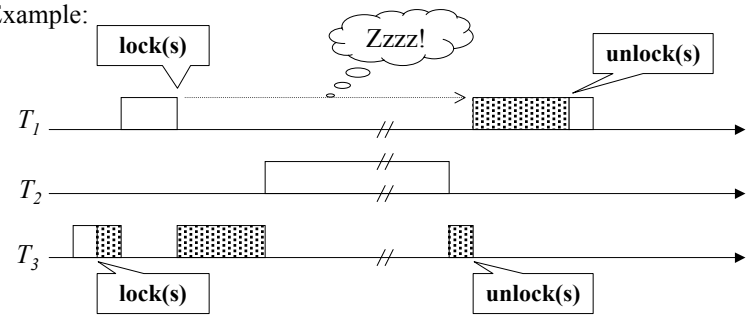
Resource Access

- Processor(s)
  - m types of serially reusable resources $R_1, ..., R_m$
  - An execution of a job J_i requires:
    - a processor for e_i units of time
    - some resources for exclusive use
- Resources
  - serially Reusable: Allocated to one job at a time. Once allocated, held by the job until no longer needed.
  - examples: semaphores, locks, servers, ...
  - operations:

    `lock(Ri)` -----<critical section>------ `unlock(Ri)`
  - resources allocated non-preemptively
  - critical sections properly nested

Preemption During Critical Sections

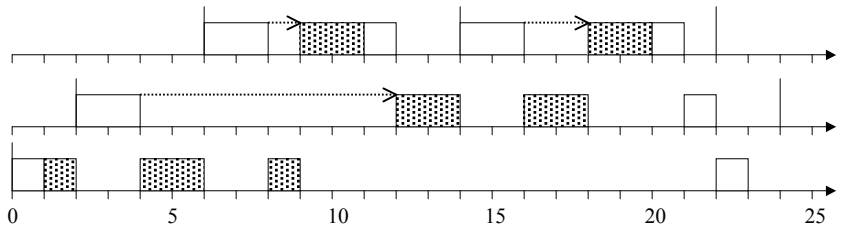
Example:



- Negative effect on schedulability and predictability.

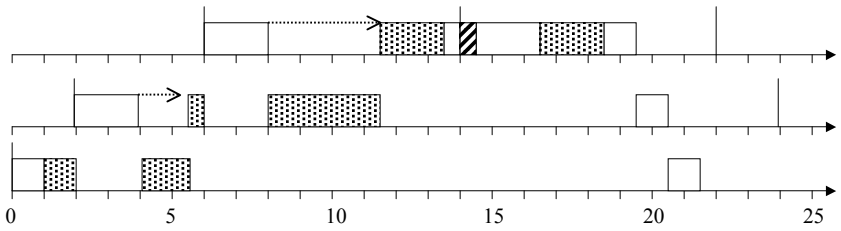
Unpredictability: Scheduling Anomalies

- Example: $T_1 = (c_1=2, e_1 = 5, p_1 = 8)$ $T_2 = (4, 7, 22)$ $T_3 = (4, 6, 26)$

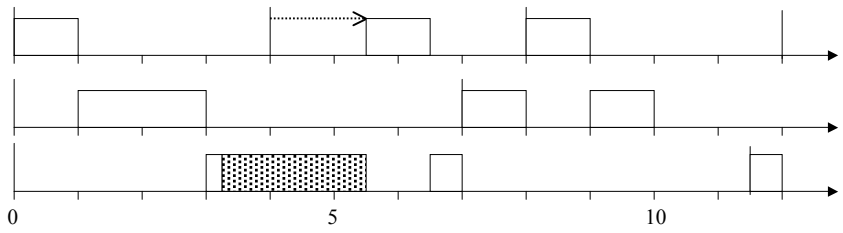


- Shorten critical section of T_3:

$T_1 = (c_1=2, e_1 = 5, p_1 = 8)$ $T_2 = (4, 7, 22)$ $T_3 = (\mathbf{2.5}, 6, 26)$



Disallow Process Preemption in CS



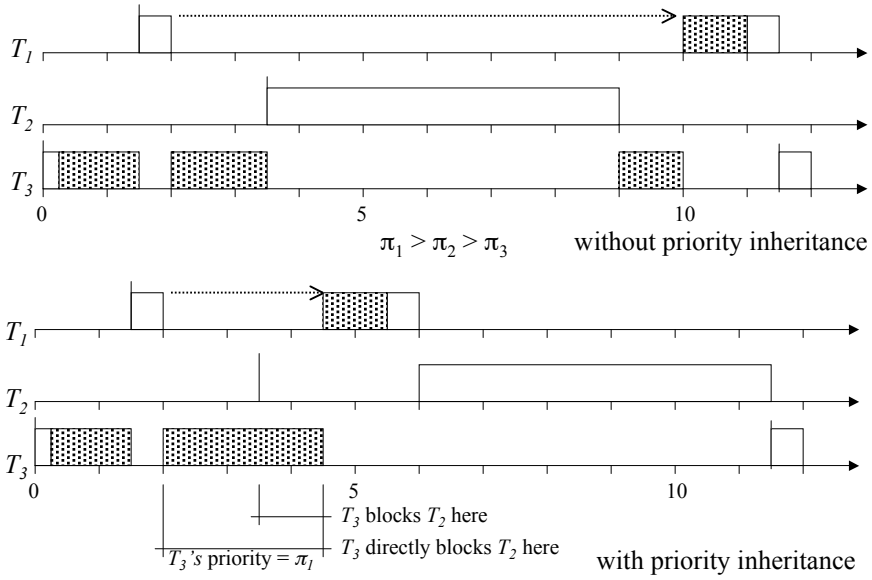
- Analysis identical to analysis with non-preemptable portions
- Define: β = maximum duration of all critical sections
- Task T_i is schedulable if

$$\sum_{k=1}^{i} \frac{e_k}{p_k} + \frac{\beta}{p_i} = U_X(i)$$

X: scheduling algorithm

- Problem: critical sections can be rather long.

Priority Inheritance



Terminology

- A job is directly blocked when it requests a resource R_i, i.e. executes a `lock(R_i)`, but no resource of type R_i is available.
- The scheduler grants the lock request, i.e. allocates the requested resource to the job, according to the resource allocation rules, as soon as the resources become available.
- *J'* directly blocks *J* if *J'* holds some resources that *J* has requested.

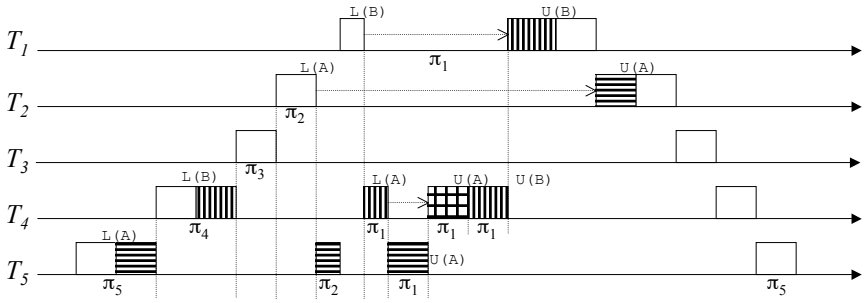
- Priority Inheritance:
  - Basic strategy for controlling priority inversion:
    Let π be the priority of *J*
    and π' be the priority of *J'*
    and $\pi' < \pi$
    then the priority of *J'* is set to π whenever *J'* directly blocks *J*.

- New forms of blocking may be introduced by the resource management policy to control priority inversion and/or prevent deadlocks.

Basic Priority-Inheritance Protocol

- Jobs that are not blocked are scheduled according to a priority-driven algorithm preemptively on a processor.
- Priorities of tasks are fixed, except for the conditions described below:
  - A job *J* requests a resource *R* by executing `lock(R)`
  - If *R* is available, it is allocated to *J*. *J* then continues to execute and releases *R* by executing `unlock(R)`
  - If *R* is allocated to *J'*, *J'* directly blocks *J*. The request for *R* is denied.
  - However: Let π = priority of *J* when executing `lock(R)`
    π' = priority of *J'* at the same time
  - For as long as *J'* holds *R*, its priority is $max(\pi, \pi')$ and returns to π' when it releases *R*.
  - That is: *J'* inherits the priority of *J* when *J'* directly blocks *J* and *J* has a higher priority.
- Priority Inheritance is transitive.

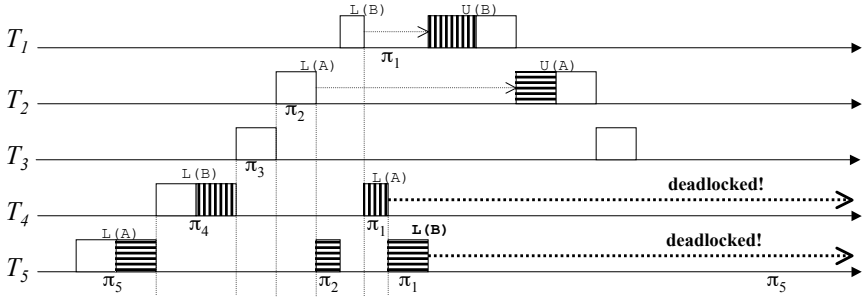
Example: Priority Inheritance Protocol



$\pi_1 > \pi_2 > \pi_3 > \pi_4 > \pi_5$

Task uses A Task uses A and B Task uses B

Example: Priority Inheritance Protocol



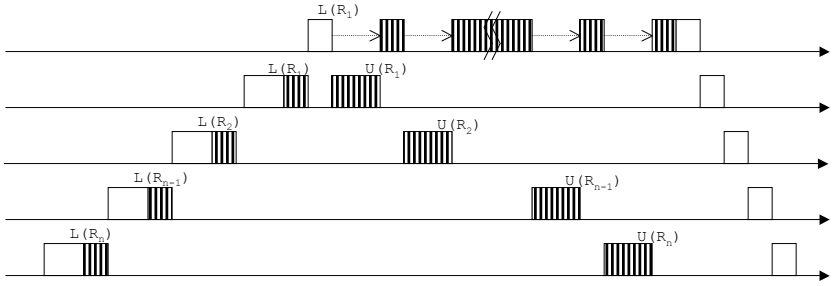
$\pi_1 > \pi_2 > \pi_3 > \pi_4 > \pi_5$

Task uses A Task uses A and B Task uses B

Problem: If T_5 tries to `lock(B)` while it has priority π_1, we have a deadlock!

Properties of PIP

- It does not prevent deadlock.
- Task can be blocked directly by a task with a lower priority at most once, for the duration of the (outmost) critical section.
- Consider a task whose priority is higher than *n* other tasks:



- Each of the lower-priority tasks can directly block the task at most once.
- A task outside the critical section cannot directly block a higher-priority task.