

# CT420 REAL-TIME SYSTEMS

## PROCESS SYNCHRONIZATION AND RTS

Dr. Michael Schukat



# Background

2

- We already know a RTS may miss deadlines under the following circumstances:
  - ▣ Poorly structured CE or process hierarchy
  - ▣ Erroneous task / process WCET assumptions
  - ▣ Non-preemptable kernel sections
  - ▣ Too many asynchronous events, i.e. signals or interrupts, to be processed
  - ▣ Incorrect use of timers to control process schedule (i.e., nanosleep() example)
- However, there is another problem area in pre-emptive multitasking environments (and RTOS): **Poorly designed resource sharing of processes**
  - ▣ Problematic for both pure OS and OS RT extensions
- In this lecture we are going to investigate this problem and at solutions to make sure that resource sharing does not interfere with the timely execution particularly of high-priority tasks

# Lecture Overview

3

- Introduction to real-time synchronisation, problems and workarounds, i.e.
  - critical regions
  - priority inversion
  - priority inheritance
  - the priority ceiling protocol
  - The Mars Pathfinder case study

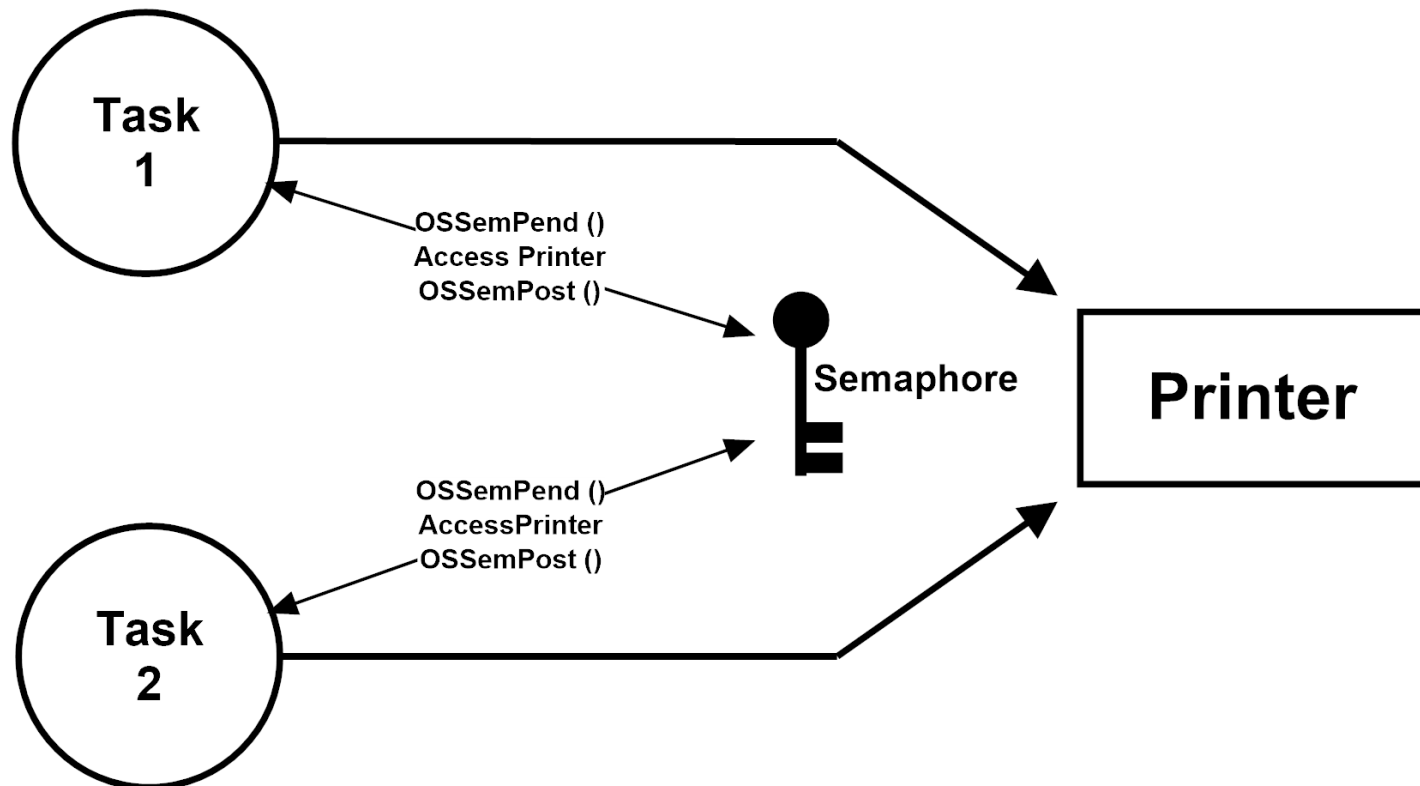
# Critical Region

- Attention is needed when more than one process requires access to single resource
  - Necessary to prevent *race conditions*  
*“The condition of an electronics, software, or other system where the system's behaviour is dependent on the sequence or timing of other uncontrollable events”*
  - Ensure that only one process can access resource at any time
    - E.g. memory write / read
- A *Critical Region* is a section of code that interacts with the shared resource
  - Once task enters critical region, it must be allowed to complete
  - This is achieved via mutexes & **semaphores**

# Semaphore Example

5

- Two tasks have a print job, but only one task can access the printer at a time



# Critical Region Example

- Task 1 and 2 share common a resource
  - ▣ The access to the resource is provided via a critical region
  - ▣ A single mutex (binary semaphore) S shared between both tasks is used to enforce single access
  - ▣ Semaphores and their API are provided by OS
  - ▣ `wait(S)` and `signal(S)` are used to lock and unlock access to region respectively

## Task 1

.  
. .  
  
wait(S)  
critical region  
signal(S)

## Task 2

wait(S)  
critical region  
signal(S)

TIME

.  
.

# Typical Semaphore Implementation

```
typedef struct SEMAPHORE {
    int value; /* the integer value for semaphore */
    other stuff;
} sem_t;

init(sem_t *S, int i)
{
    S->value = i;
}

void wait(sem_t *S)
{
    S->value--;
    if (S->value < 0)
        block on semaphore
}

void signal(sem_t *S)
{
    S->value++;
    if (S->value <= 0)
        unblock one process or thread that is blocked on semaphore
}
```

value determines how many processes can access the protected resource at a time

# Example: Binary Semaphore


8

- In this example a binary semaphore  $S$  is used by processes  $P_1, \dots, P_4$  to control access to some resource within a CS

State no.	Actions		Results		
	Calling process	Operation	Running in CS	Blocked on S	Value of S
0					1
1	P1	Wait(S)	P1		0
2	P1	Signal(S)			1
3	P2	Wait(S)	P2		0
4	P3	Wait(S)	P2	P3	-1
5	P4	Wait(S)	P2	P3,P4	-2



# POSIX Semaphore API

<b>Header file name</b>	<code>#include &lt;semaphore.h&gt;</code>
<b>Semaphore data type</b>	<code>sem_t</code>  See next slide
<b>Initialization</b>	<code>int sem_init(sem_t *sem, int pshared, unsigned value);</code>
<b>Semaphore Operations</b>	<code>int sem_destroy(sem_t *sem);</code> <code>int sem_wait(sem_t *sem);</code> <code>int sem_post(sem_t *sem);</code> <code>int sem_trywait(sem_t *sem);</code>
<b>Compilation</b>	<code>cc filename.c -o filename -lrt</code>

# POSIX Semaphore API

All of the POSIX.1b semaphore functions return **-1** to indicate an error.

`sem_init` function initializes the semaphore to have the value `value`. The `value` parameter cannot be negative. If the value of `pshared` is not `0`, the semaphore can be used between processes (i.e. the process that initializes it and by children of that process). Otherwise it can be used only by threads within the process that initializes it.

`sem_wait` is a standard semaphore wait operation. If the semaphore value is `0`, the `sem_wait` blocks until it can successfully decrement the semaphore value.

`sem_trywait` is similar to `sem_wait` except that instead of blocking when attempting to decrement a zero-valued semaphore, it returns `-1`.

`sem_post` is a standard semaphore signal operation. The POSIX.1b standard requires that `sem_post` be reentrant with respect to signals, that is, it is asynchronous-signal safe and may be invoked from a signal-handler.

# Semaphore Pitfalls in RTS: The Priority Inversion Problem

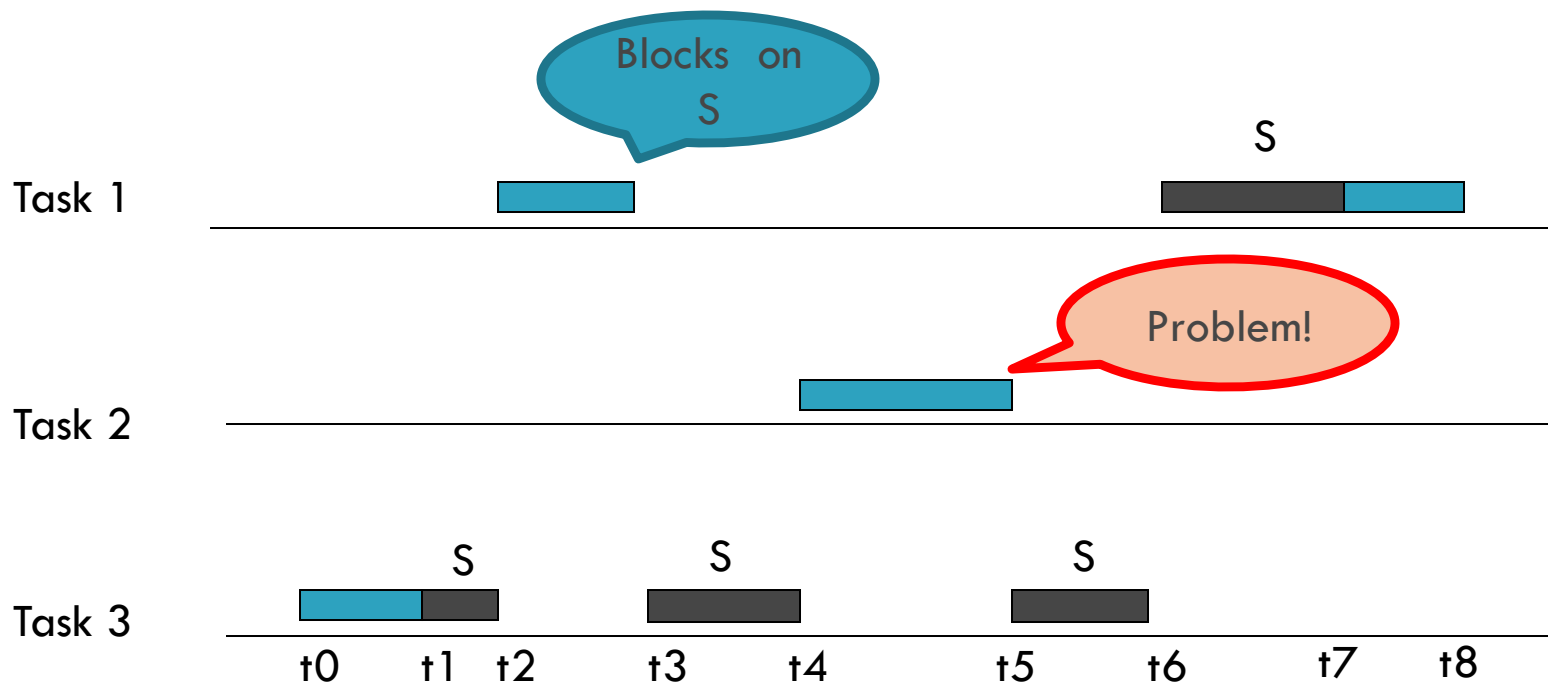
11

- Priority inversion is a scenario in process scheduling in which a high-priority task is indirectly pre-empted by a lower priority task effectively inverting the relative priorities of the two tasks
- This behaviour is undesirable, particularly in RTS, where a high-priority task needs to meet a deadline
- Let's have a look at an example

# Example: Priority Inversion Problem

- Consider Task 1, Task 2, Task 3 with decreasing task priorities
- Task 1 and 3 share resource that is protected via mutex (binary semaphore) S
- Timeline (next slide)
  - t0: Task 3 commences
  - t1: Task 3 enters critical region S
  - t2: Task 1 pre-empts Task 3
  - t3: Task 1 attempts to enter shared critical region S
    - Locked out by Task 3
    - Blocks → Task 3 resumes
  - t4: Task 2 released and pre-empts Task 3
  - t5: Task 2 completes
    - Task 3 resumes S
  - t6: Task 3 completes critical region.. Releases S
    - Task 1 enters S
  - t7: Task 1 releases S

# Example: Priority Inversion Problem



- Task 1, Task 2, Task 3 have decreasing task priorities
- Task 1 and 3 share resource that is protected via the mutex (binary semaphore) S

# Example: Priority Inversion Problem

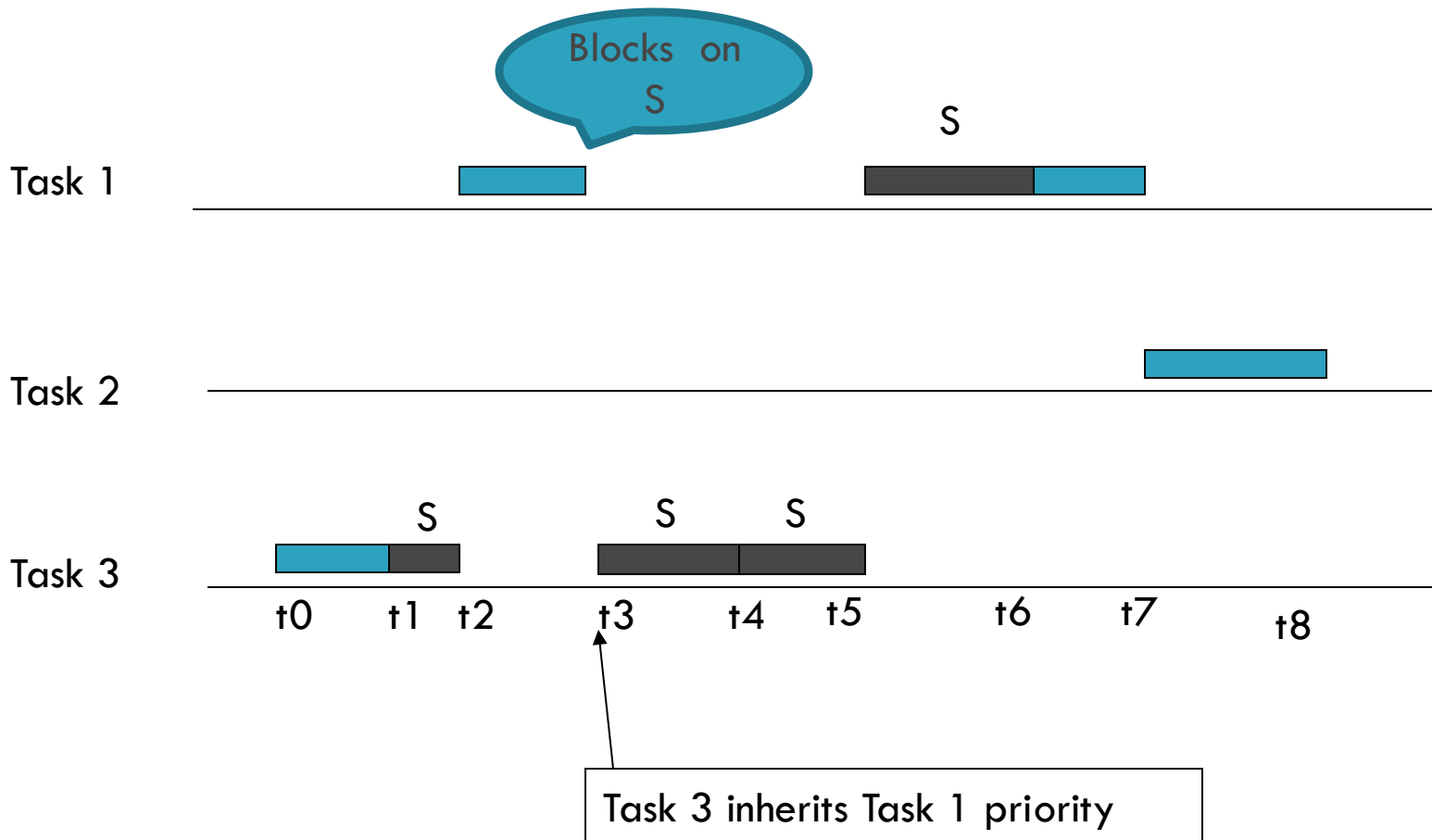
- Task 2 delays Task 1 indirectly by period  $t_5 - t_4$ 
  - ▣ Pre-empts Task 3 which was blocking Task 1
- Lower priority Task 3 indirectly delays higher priority Task 1 for indeterminate period
  - ▣ Priority Inversion
  - ▣ Intermediate Task 2 can repeatedly pre-empt Task 3
- Ideally, Task 2 should be prohibited from pre-empting Task 3 when Task 3 was already blocking Task 1
- Priority Inheritance facilitates this solution

# Solution: Priority Inheritance

- If a higher priority task is blocked by a lower priority task (due to a critical region), the lower priority task inherits the priority of the higher priority task for the duration of critical region, after which lower priority task restored to initial priority
- More generally, a task causing blocking executes with priority
  - ▣ `max(Own Priority, Highest Priority of any tasks it is blocking)`

prevents an intermediate task priority from delaying a higher priority task

# Example: Priority Inheritance



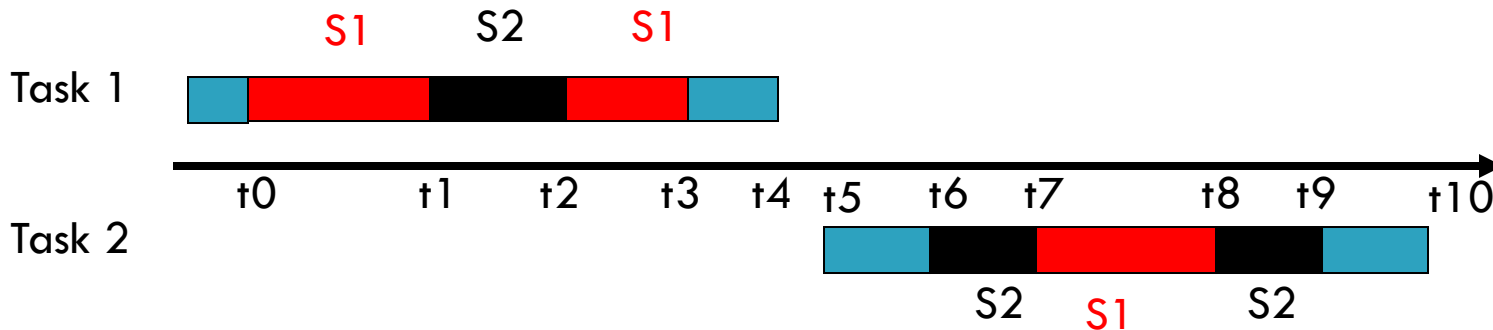


# Example: Priority Inheritance

- Timeline
  - t0: Task 3 commences
  - t1: Task 3 enters critical region S
  - t2: Task 1 pre-empts Task 3
  - t3: Task 1 attempts to enter shared critical region S
    - Locked out by Task 3
    - Blocks → Task 3 resumes but *inherits Task 1 priority*
  - t4: Task 2 is released but does not pre-empt Task 3
    - Task 3 continues
  - t5: Task 3 completes critical region and releases S
    - Task 1 enters S
  - t6: Task 1 completes critical region and releases S
    - Task 1 continues to completion
  - t7: Task 2 runs
- Task 1 is only delayed (blocked) by time interval required by Task 3 to complete S
- It will then get access to S
- Note:
  - A task T can be blocked **multiple** times by different lower priority tasks which have locked various resources shared with Task T
  - The total blocked time can be significant and needs to be considered by system designers

# Another Problem: Task Deadlocks

- Consider Task 1 and Task 2 with Task 1 having the highest priority
- They share 2 resources R1 and R2, protected by S1 and S2
- Task 1: Accesses S1, then S2 nested within
- Task 2: Accesses S2, then S1 nested within



**Above scenario: Time separated access → No problem**

$t_0$ - $t_1$ : Task 1 is running S1 locked

$t_1$ - $t_2$ : Task 1 nested lock of S2, S2 unlocked at  $t_2$

$t_3$ - $t_4$ : Task 1 running : no locks, complete at  $t_4$

$t_5$ : Task 2 runs, locks S2 at  $t_6$

$t_7$ : Nested lock of S1, unlocked at  $t_8$ , S2 unlocked at  $t_9$

# Example for a Deadlock Situation

- Scenario 2

t0: Task 2 is running and locks S2

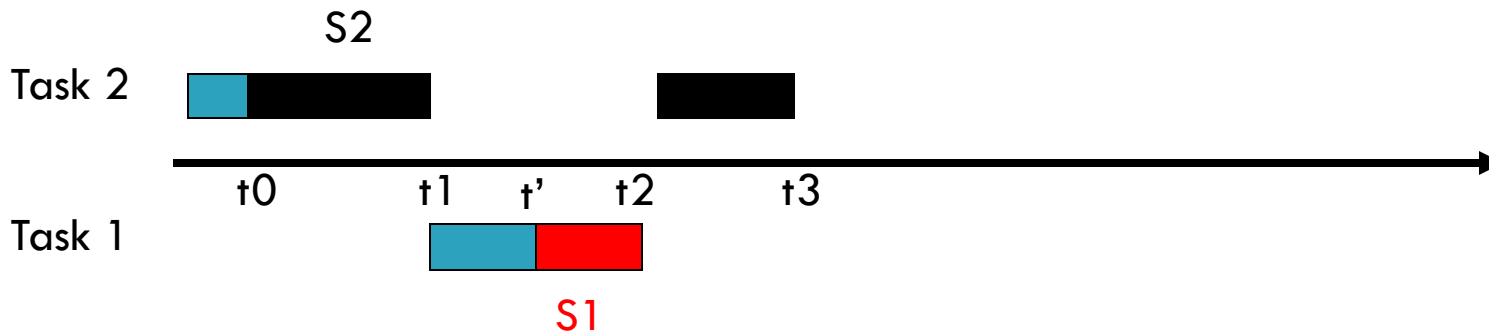
t1: Task 1 pre-empts Task 2 and locks S1 at t'

t2: Task 1 attempts to lock S2 ... Fails → Task 2 runs

t3: Task 2 attempts to lock S1 ... Fails

→ Deadlock

M



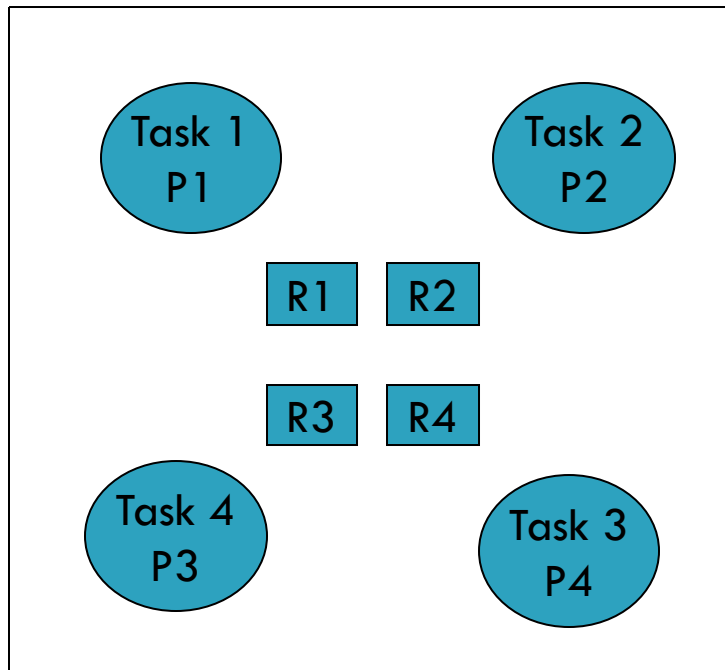
**The Priority Ceiling Protocol avoids this situation**

# The Priority Ceiling Protocol

- Extends the Priority Inheritance Protocol
  - ▣ Prevents deadlock situations
  - ▣ Reduces total potential blocking time
- Define
  - ▣ The **Priority Ceiling of a resource** (critical region)  $R$  protected by semaphore  $S$  denoted  $\pi(R)$  is defined as the highest priority of all the tasks that may utilise  $R$ 
    - With priority inheritance, priority of task will change if it is locking a resource requested by a higher priority task BUT by definition the Priority Ceiling of that resource is unchanged by this
  - ▣ The **Current Priority Ceiling** of the overall system denoted  $\Pi(t)$  is defined as highest priority ceiling of all the resources that are in use at time  $t$  (presuming some resources are in use)

# Example: Priority Ceiling Protocol

## RTS



## □ Tasks T1 to T4

- Decreasing priorities P1 to P4
- $\pi(R)$  constant and shown in the table
- $\Pi(t)$  depends on resources in use at time  $t$

Resource	Tasks	$\pi(R)$
R1	T1, T2, T4	P1
R2	T3, T4	P3
R3	T2, T3	P2
R4	T1, T4	P1

# Priority Ceiling Protocol

## □ Rules

### □ At time $t$ , if Task $T$ requests resource $R$

- If  $R$  is already locked, the request fails and task  $T$  is blocked

- If  $R$  is free

- If the priority of task  $T$  is  $> \Pi(t)$ ,  $R$  is allocated to  $T$

- If the priority of task  $T$  is not  $> \Pi(t)$ ,  $R$  is allocated **iff** task  $T$  is already holding the resource  $R_2$  whose priority ceiling  $\pi(R_2)$  is  $\Pi(t)$

- Note: If another task is holding the resource  $R_2$ , with priority ceiling  $\pi(R_2)$ ,  $T$  is blocked even if it never actually requires access to  $R_2$

- Otherwise, task  $T$  is blocked

There's no lower priority task that shares (potentially) a resource with  $T \rightarrow$  No deadlock possible

$T$  holds the resource, i.e. it can't block itself

If we can't guarantee the above, we need to be prudent and block  $T$

i.e. we only consider that a resource is potentially shared, but we do not further consider if and when this will ever happen during the execution of the program

# Priority Ceiling Protocol

- General Rule: By definition, at time  $t$ , if the priority of a task  $T$  is higher than the priority ceiling of the system at that time  $\Pi(t)$ ,
  - task  $T$  does not ever require the resources in use at time  $t$
  - tasks with priorities equal to or higher than  $T$  will not ever use them either (otherwise  $\Pi(t)$  would be equal or higher than  $T$ )
    - ➔  $\Pi(t)$  by default tells us the subset of tasks to which we can grant free resources at time  $t$ , i.e. all the tasks that have priorities higher than  $\Pi(t)$

# Recall Example Deadlock

- Scenario 2

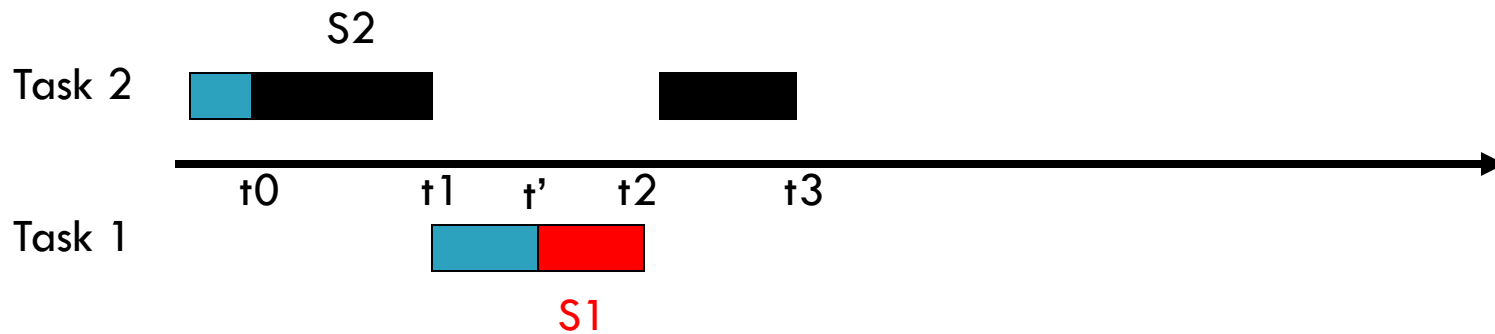
t0: Task 2 is running and locks S2

t1: Task 1 pre-empts Task 2 and locks S1 at t'

t2: Task 1 attempts to lock S2 ... Fails → Task 2 runs

t3: Task 2 attempts to lock S1 ... Fails

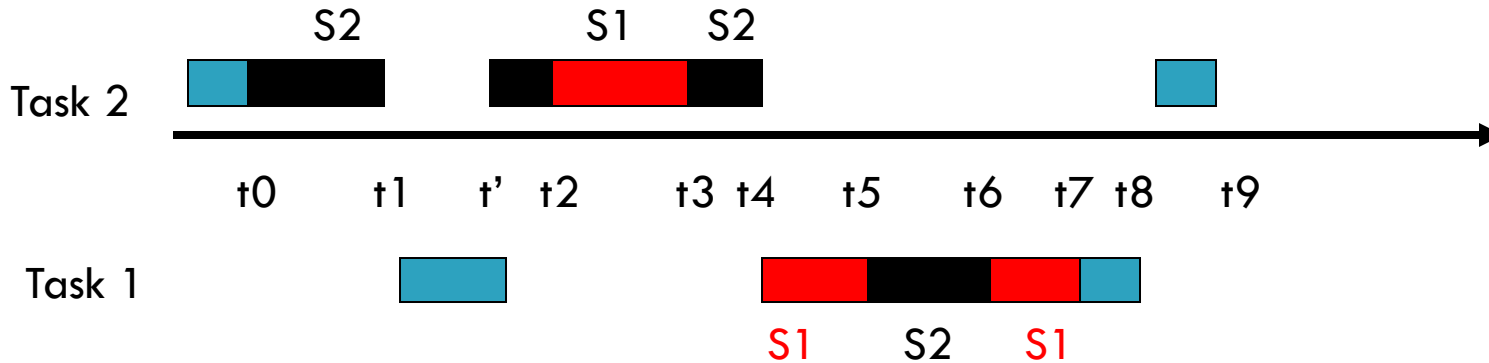
→ Deadlock



Priority Ceiling Protocol avoids this situation



# Example: Deadlock Avoidance via the Priority Ceiling Protocol



# Example: Deadlock Avoidance via the Priority Ceiling Protocol

- In the previous example,
  - ▣ Have 2 resources R1, R2 protected by S1 and S2 and shared by 2 Tasks, Task 1 and Task 2
    - R1 → S1 : Priority Ceiling is  $\pi(R1) = \text{Prior}(\text{Task 1})$
    - R2 → S2 : Priority Ceiling is  $\pi(R2) = \text{Prior}(\text{Task 1})$
  - ▣ At time  $t'$ , Task 1 attempts to lock S1
    - Priority Ceiling of System  $\Pi(t)$  is that of resource R2 which is locked by Task 2 →  $\Pi(t) = \pi(R2) = \text{Prior}(\text{Task 1})$
    - S1 is unlocked (R1 free) BUT
      - Priority(Task 1) is not  $\geq \Pi(t)$  as  $\Pi(t) = \text{Prior}(\text{Task 1})$
      - Task 1 is NOT holding resource whose priority ceiling =  $\Pi(t)$ 
        - i.e. Task 2 holding R2
      - Task 1 blocked from locking S1
      - Task 2 continues and inherits Task 1 priority

# Example: Deadlock Avoidance via the Priority Ceiling Protocol

- Scenario 2: Full timeline

t0: Task 2 is running and locks S2

t1: Task 1 pre-empts Task 2 and attempts to lock S1 at t'

t': Task 1 blocked, Task 2 resumes and inherits Task 1 priority

t2: Task 2 attempts to lock S1 .. Successful

Note:  $\text{Prior}(\text{Task 2}) = \text{Prior}(\text{Task 1}) = P$  through inheritance

$P \text{ not } \geq \Pi(t)$  as  $\Pi(t) = \pi(R2) = \text{Prior}(\text{Task 1})$  BUT

Task 2 actually holds R2 → ok

t3: Task 2 releases S1, continues with S2

t4: Task 2 releases S2, priority restored to  $\text{Prior}(\text{Task 2})$

t4: Task 1 pre-empts Task 2, resumes, locks S1

t5: Task 1 attempts to lock S2 .. Successful

Same reason as above t2 but logic applied to Task 1

t6: Task 1 releases S2, continues with S1

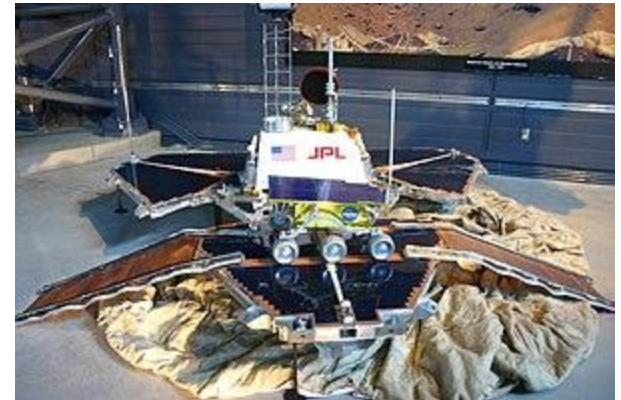
t7: Task 1 releases S1, no resources held. Task 1 continues

t8: Task 1 complete, Task 2 resumes

t9: Task 2 complete

# Case Study: Mars Pathfinder

- ❑ Launched 1996 and landed July 1997
- ❑ Consisted of a Lander (Pathfinder) and a Rover (Sojourner)
- ❑ It was the first mission to Mars since the Viking programme in 1976 (2 probes were sent)
- ❑ Inflation corrected,
  - ❑ the Viking programme did cost \$7 billion
  - ❑ Pathfinder did only cost \$485 million
- ❑ It was a “faster-better-cheaper” project and a demonstrator for using new landing techniques (parachute and airbags) and standard components where possible (e.g. computer boards or OS)
- ❑ See
  - ❑ <https://www.youtube.com/watch?v=5-cBjI2zgB0>
- ❑ Compare this to the 2021 landing of Perseverance:
  - ❑ <https://www.youtube.com/watch?v=rzmd7RouGrM>



# Mars Pathfinder Hardware

29

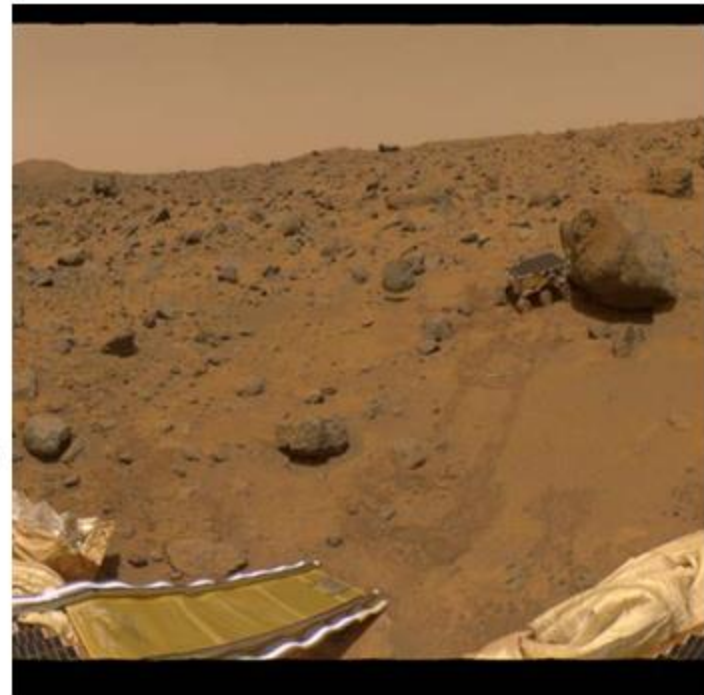
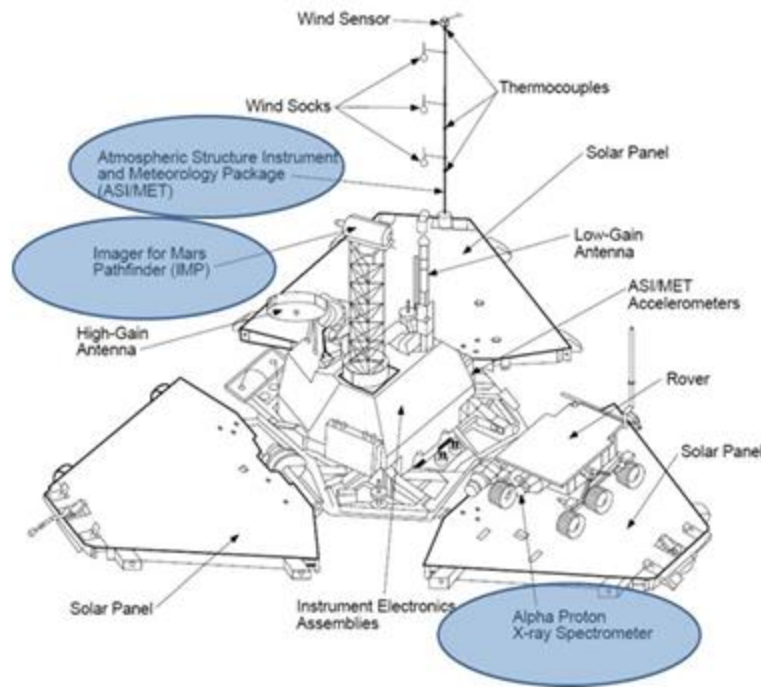
- The computer on board the rover was based on a 2 MHz Intel 80C85 CPU with 512 KB of RAM and 176 KB of flash memory solid-state storage, running a bare-bone **cyclic executive**
- The computer of the Pathfinder lander was a radiation hardened IBM RISC 6000 (Rad6000 SC) CPU with 128 MB of RAM and 6 MB of EEPROM; it used the RTOS **VxWorks**

# VxWorks

- Proprietary RTOS by Wind River Systems
  - See <http://www.windriver.com/products/vxworks/>
  - Fully POSIX.4 Compliant included pre-emptive FIFO priority scheduling
  - Continuously improved since the 1990s
- Widely used, even in safety critical systems
  - Boeing 787 (aviation industry)
  - Router/Switches
  - Mars Pathfinder
- However, a few days after being deployed on Mars, Pathfinder suffered repeated system resets

# Mars Pathfinder Hardware Architecture

- The Rad6000 SC CPU controlled the entire spacecraft (excluding the rover)



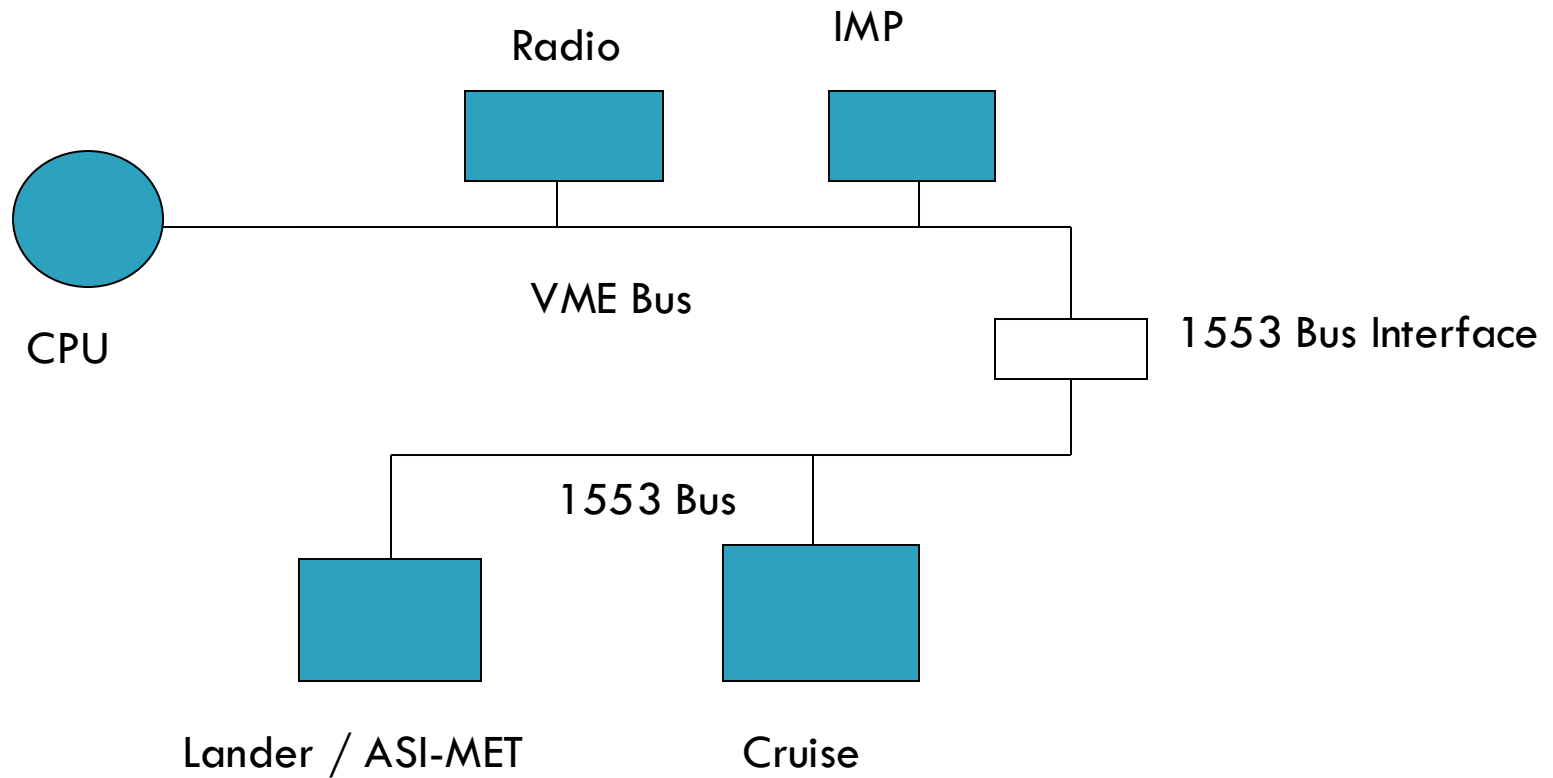
# Pathfinder Instruments

32

- Imager for Mars Pathfinder (IMP):
  - ▣ Hosted on the lander
  - ▣ Used for imaging the surface of Mars and helped to navigate the rover
- Atmospheric Structure Instrument and Meteorology Package (ASI/MET):
  - ▣ Hosted on the lander
  - ▣ Used to acquire atmospheric information (e.g. pressure, temperature, wind)
- Alpha Proton X-ray Spectrometer (APXS):
  - ▣ Hosted on the rover
  - ▣ Designed to determine the elements that make up the rocks and soil on Mars



# Mars Pathfinder Hardware Architecture

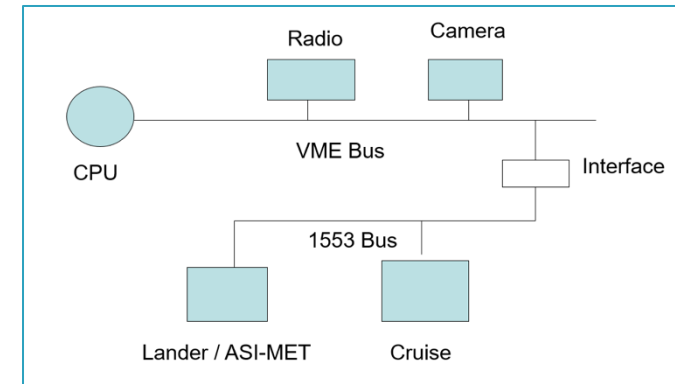


# Mars Pathfinder Bus Architecture

- The CPU was connected to a VME hardware bus, which linked it to the radio, the camera, and the interface to a 1553 bus
  - ▣ The VME bus is a parallel bus originally designed for the Motorola 68000 series
- The 1553 bus is a military grade serial bus, and it connected to:
  - ▣ The "cruise stage" part of the spacecraft and
  - ▣ The "lander" part of the spacecraft

# Mars Pathfinder: Lander/ASI-MET and Cruise Subsystem

- The hardware on the **Cruise** part of the spacecraft controlled thrusters, valves, a sun sensor, and a star scanner
  - ▣ Was only operational during the flight to Mars
- The hardware on the **Lander / ASI-MET** part provided
  - ▣ an accelerometers and a radar altimeter (used during the landing phase only)
  - ▣ the aforementioned ASI-MET instrument (used when the lander was on the Mars surface)



# The VME Bus

36

- Data (video images, meteorological readings, etc.) from the various instruments on the lander (Pathfinder) and the rover (Sojourner) had to pass through this bus to be transmitted to Earth
- Likewise, commands to control the instruments on Pathfinder (such as the camera or the ASI-MET) had to pass through this bus
- Obviously, this couldn't happen all at once, therefore data threads and command strings had to take turns using the bus
- It was the job of VxWorks to schedule traffic through the bus according to the pre-assigned priorities of data and commands

# Pathfinder Software Architecture

37

- VxWorks provided pre-emptive fixed-priority scheduling
- Tasks were executed via a cyclic scheduler with a cycle of 125 ms
  - ▣ Basically, the scheduler was organised as a CE, a bit like the example in previous lectures, but with
    - just one task per priority
    - tasks exited after completion and were executed again in the next cycle

# Pathfinder Software Architecture

- The software to control the 1553 bus and the attached instruments was implemented in two main tasks
  - ▣ Bus Scheduler: `bc_sched`
    - Decided what instrument would transmit data next and transmitted the schedule to the instrument
  - ▣ Data Collection: `bc_dist`
    - Handled the collection of the instrument data selected by `bc_sched`
- Additional tasks perform other spacecraft functions
  - ▣ Communication task (for radio comms): `communication`
  - ▣ Meteorological data processing task (processing data from ASI-MET instrument): `ASI-MET`
- Process priorities were as follows:
  - ▣ `prio("bc_sched") > prio("bc_dist") > prio("communication") > prio("ASI-MET")`

# Pathfinder Software Architecture

39

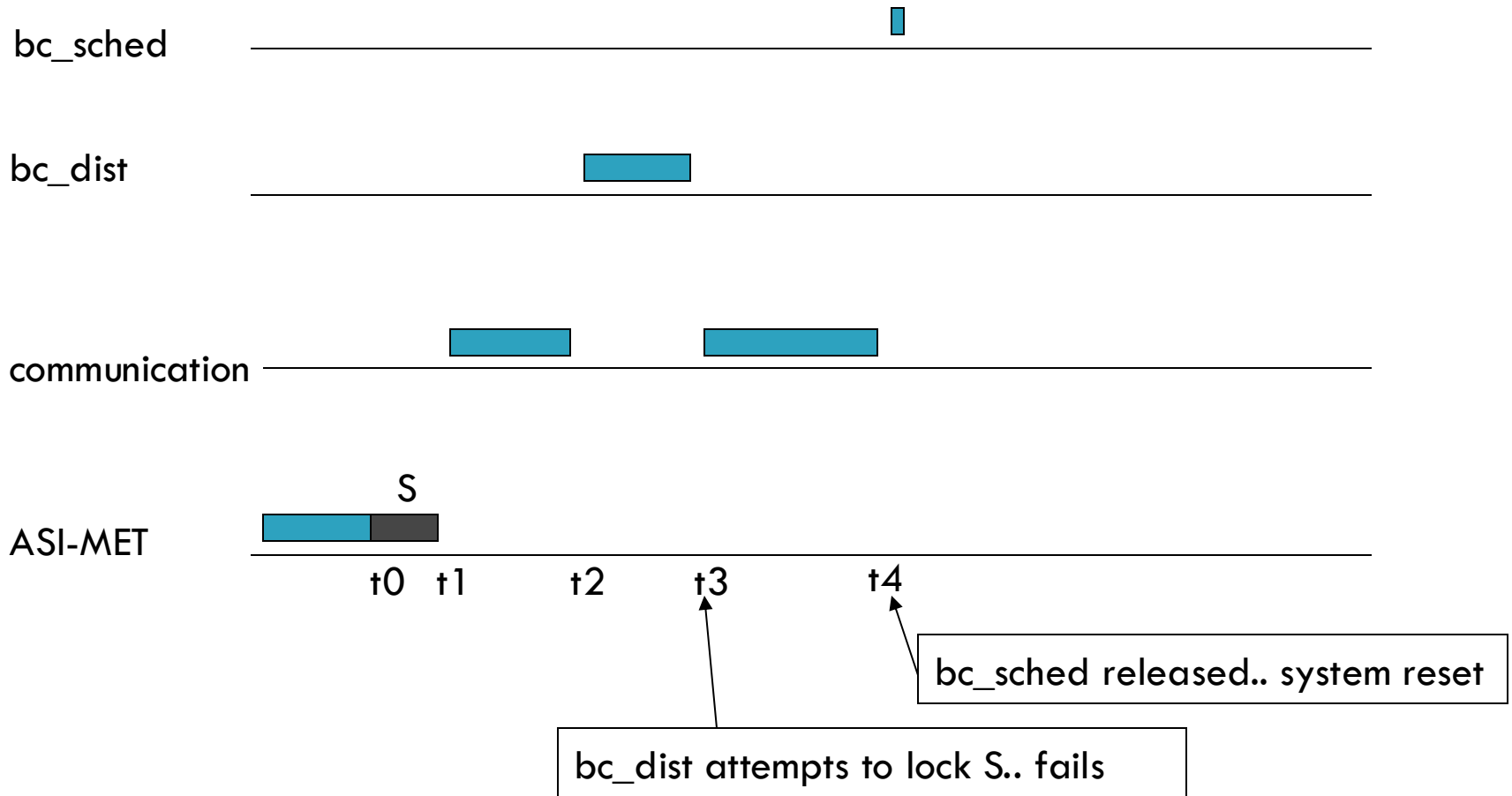
- Using a watchdog timer, the `bc_sched` task checked at the beginning of its execution whether the `bc_dist` task had completed its execution in the previous cycle
  - ▣ Similar to `TimerFlag` variable in the CE example
- If `bc_dist` had not completed, `bc_sched` initiated a system reset
- A system reset caused a cold restart of the pathfinder to bring it back to a safe state
  - ▣ This terminated all current ground commanded activities including rover control, data upload etc. for hours
  - ▣ This was critical, since the probe had only a limited expected lifetime (30 days for the lander and 7 days for the rover) because of dust slowly covering the solar panels

# The Problem

- The `ASI-MET` task and the `bc_dist` task shared a resource managed by a binary semaphore
- The fault sequence looks as follows:
  - `t0`: The `ASI-MET` task acquires semaphore
  - `t1`: communication task pre-empts `ASI-MET`
  - `t2`: `bc_dist` is released and pre-empts the communication task
  - `t3`: `bc_dist` attempts to lock semaphore which is already locked
    - communication task resumes
  - `t4`: `bc_sched` is released and determines that `bc_dist` has not completed
    - Forces system reset
- → **Classical Priority Inversion Problem**



# The Problem



# Root Cause Analysis and Fix

- The problem only manifested when ASI-MET collected data and the communication task was heavily loaded
- There were two oversights by NASA engineers
  - ▣ Firstly, testing before launch was limited to the **"best case" communication task activity**, therefore the problem did never occur
  - ▣ Secondly, engineers were not aware that VxWorks sets the priority inheritance flag **off** for semaphores by default, i.e. it needs to be set via a compiler switch
- However, the problem was identified (as a priority inversion issues) by engineers within a day
- The problem was subsequently rectified by recompiling the code with the Priority Inheritance option set, and uploaded it to the pathfinder probe

# Timeline with Priority Inheritance Option enabled

