Priority Inversion on Mars

Jim Huang (黃敬群) <jserv@0xlab.org>

Mar 12, 2013 / NCKU, Taiwan

Agenda

- Review Operating System Concepts
- Mars Pathfinder:
  - Problem: Priority Inversion
  - Solutions
- Resource Access Protocols

Review Operating System Concepts

Consider the Function Calls

```
void send_to_printer(int user_id, char* document)
{
    printer_write("Job from user --- %d ---", user_id);
    printer_write("%s", document);
    printer_write("-----------------------");
}
```

Process A

```
send_to_printer(59, "What a beautiful day!");
```

Process B

```
send_to_printer(12, "I hate going to school!");
```

The Results

```
Job from user --- 59 ---
What a beautiful day!
------------------------------
Job from user --- 12 ---
I hate going to school!
------------------------------
```

What I expected

```
Job from user --- 59 ---
Job from user --- 12 ---
What I hate a beautiful day!
----------------------------
going to school!
----------------------------
```

The fact

Race condition: The situation where several processes access – and manipulate shared data concurrently. The result critically depends on timing of these processes, which are “racing”.

Mutual Exclusion

```
void send_to_printer(int user_id, char* document)
{
    printer_write("Job from user --- %d ---", user_id);
    printer_write("%s", document);
    printer_write("-----------------------");
}
```

This piece of code has to be executed **Atomically**, in **Mutual Exclusion**! These three lines constitute a **Critical Section**.

This routine is neither **Thread-Safe** nor **Reentrant**!

mutex

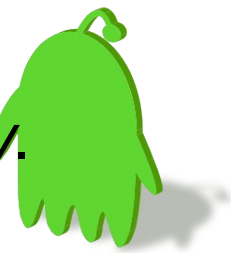
```
void send_to_printer(int user_id, char* document)
{
    lock(MUTEX);

    printer_write("Job from user --- %d ---", user_id);
    printer_write("%s", document);
    printer_write("-----------------------");

    unlock(MUTEX);
}
```

Only one thread/process can be in here!

A **mutex** is a synchronization primitive available for processes and threads. It has two primitives:

lock(): allows a thread to acquire the *mutex*, ensuring that only one flow of execution exists inside the critical section. If a second thread calls lock(), it becomes *blocked*.

unlock(): signals that a thread is leaving the critical section. If there are other threads waiting for the critical section, one of them is allowed to run: it becomes *ready*.



A mutex looks like a Binary Semaphore, but their use is different!

- mutexes needed to be more than just semaphores with a binary value.
- Because of the possibility of unbounded priority inversion, which would break RMA assumptions, ordinary semaphores cannot be used for mutual exclusion.
- Mutexes must prevent unbounded priority inversion.

inventor of semaphore: Edsger Dijkstra

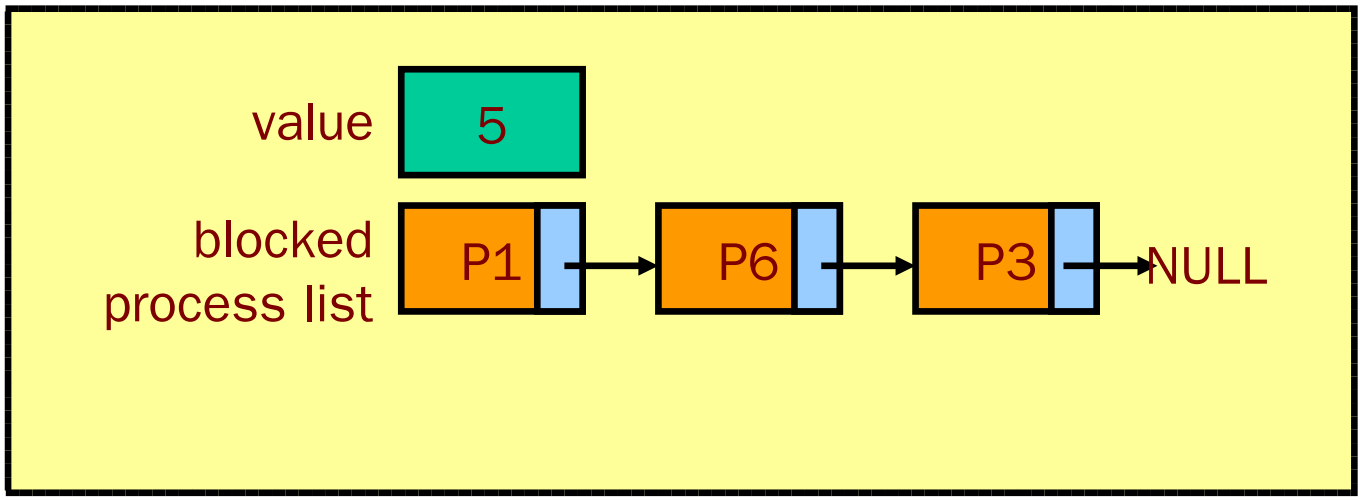
- Inventor of the semaphore, one of the key contributions for modern operating systems; developed the THE operating system
  - Used in all operating systems today.
- Created the Dijkstra algorithm for finding the shortest path in a graph
  - Used in all computer networks today (e.g. in OSPF routing).
- Wrote “A Case against the GO TO Statement”, and was one of the fathers of Algol-60
  - Which introduced the revolution of structured programming.

- Semaphores are used to count things!
- Blocked Processes in a semaphore do not consume resources (CPU)!

semaphore

- **A semaphore is a synchronization object**
  - Controlled access to a counter (a value)
  - Two operations are supported: **wait()** and **post()**
- **wait()**
  - If the semaphore is positive, decrement it and continue
  - If not, block the calling thread (process)
- **post()**
  - Increment the semaphore value
  - If there was any (process) thread blocked due to the semaphore, unblock one of them.

semaphore



"A semaphore"

Semaphores in Reality

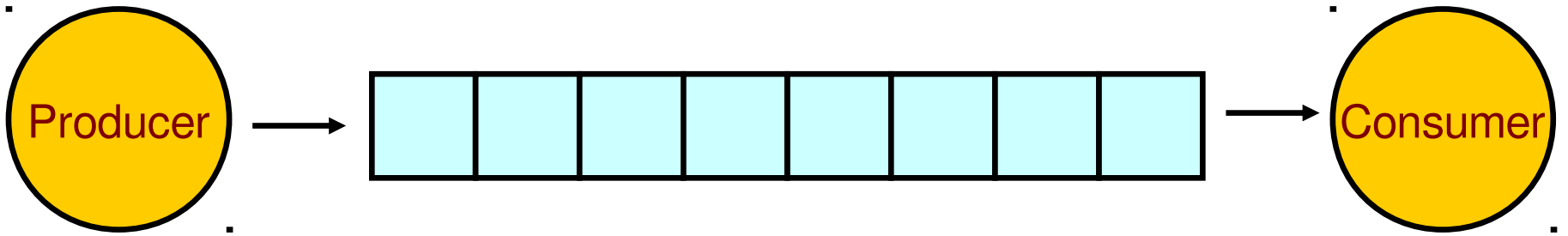
- UNIX System V Semaphores
  - Works with semaphore arrays
  - semget(), semctl(), semop()
- POSIX Semaphores
  - sem_init(), sem_close(), sem_post(), sem_wait()
  - Also work with threads
- Java: Typically uses "*monitors*", now it has:
  - java.util.concurrent.Semaphore
- .NET:Typically uses "*monitors*", but it has:
  - System.Threading.Semaphore

Semaphores are correct and convenient, but...

- Mistakes!
  - easy to forget a wait() or post()...
  - easy to do wait() and post() in different semaphores on opposite order
  - difficult to ensure correctness when several semaphores are involved
- Alternatives: Monitor
  - an abstraction where only one thread or process can be executing at a time.
    - Normally, it has associated data
    - When inside a monitor, a thread executes in mutual exclusion
  - UNIX: conditional variables

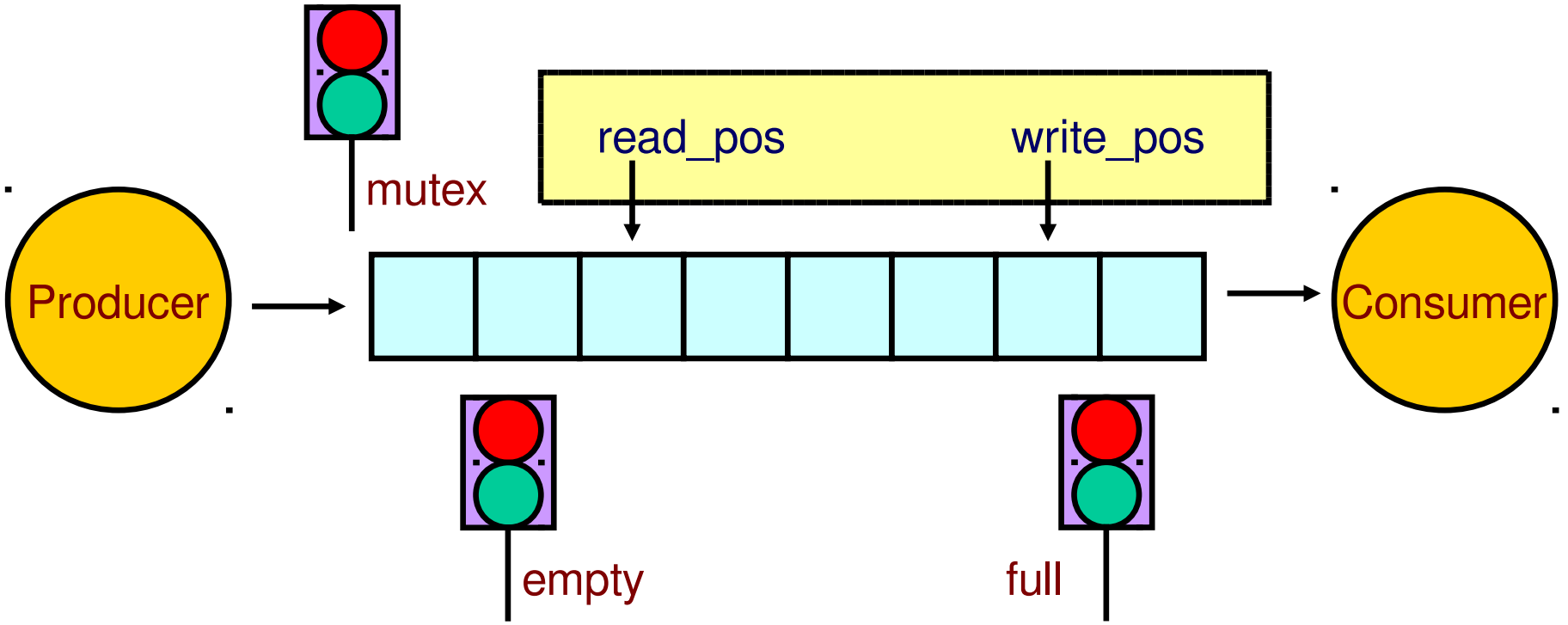
Producer/Consumer Problem

- A producer puts elements on a finite buffer. If the buffer is full, it blocks until there's space.
- The consumer retrieves elements. If the buffer is empty, it blocks until something comes along.



- We will need three semaphores
  - count the empty slots
  - count the full slots
  - provide for mutual exclusion to the shared buffer

Producer/Consumer: Basic Implementation

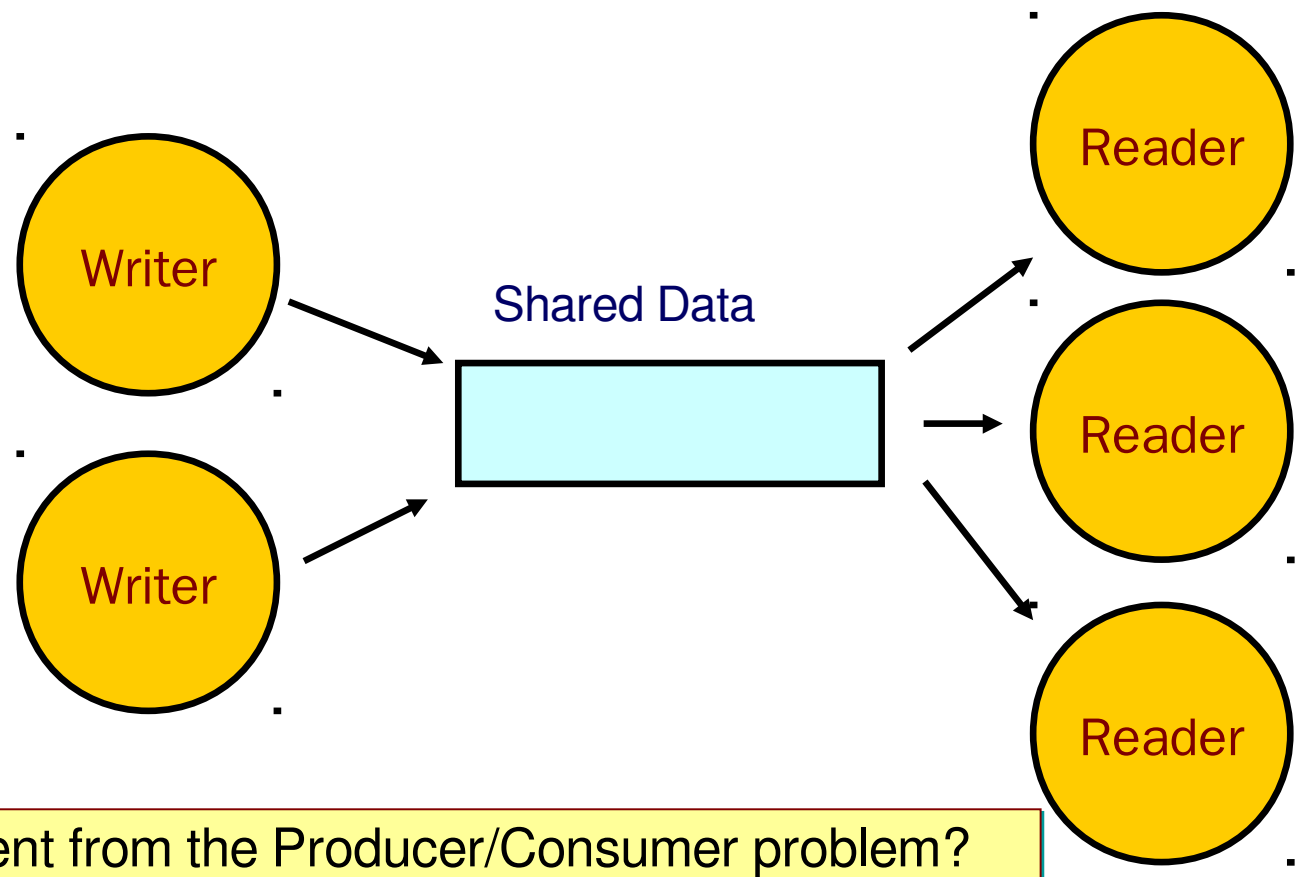



```
put_element(e) {
   sem_wait(empty);
   sem_wait(mutex);
   buf[write_pos] = e;
   write_pos = (write_pos+1) % N;
   sem_post(mutex);
   sem_post(full);
}
```

```
get_element() {
   sem_wait(full);
   sem_wait(mutex);
   e = buf[read_pos];
   read_pos = (read_pos+1) % N;
   sem_post(mutex);
   sem_post(empty);
   return e;
}
```

Readers/Writers Problem

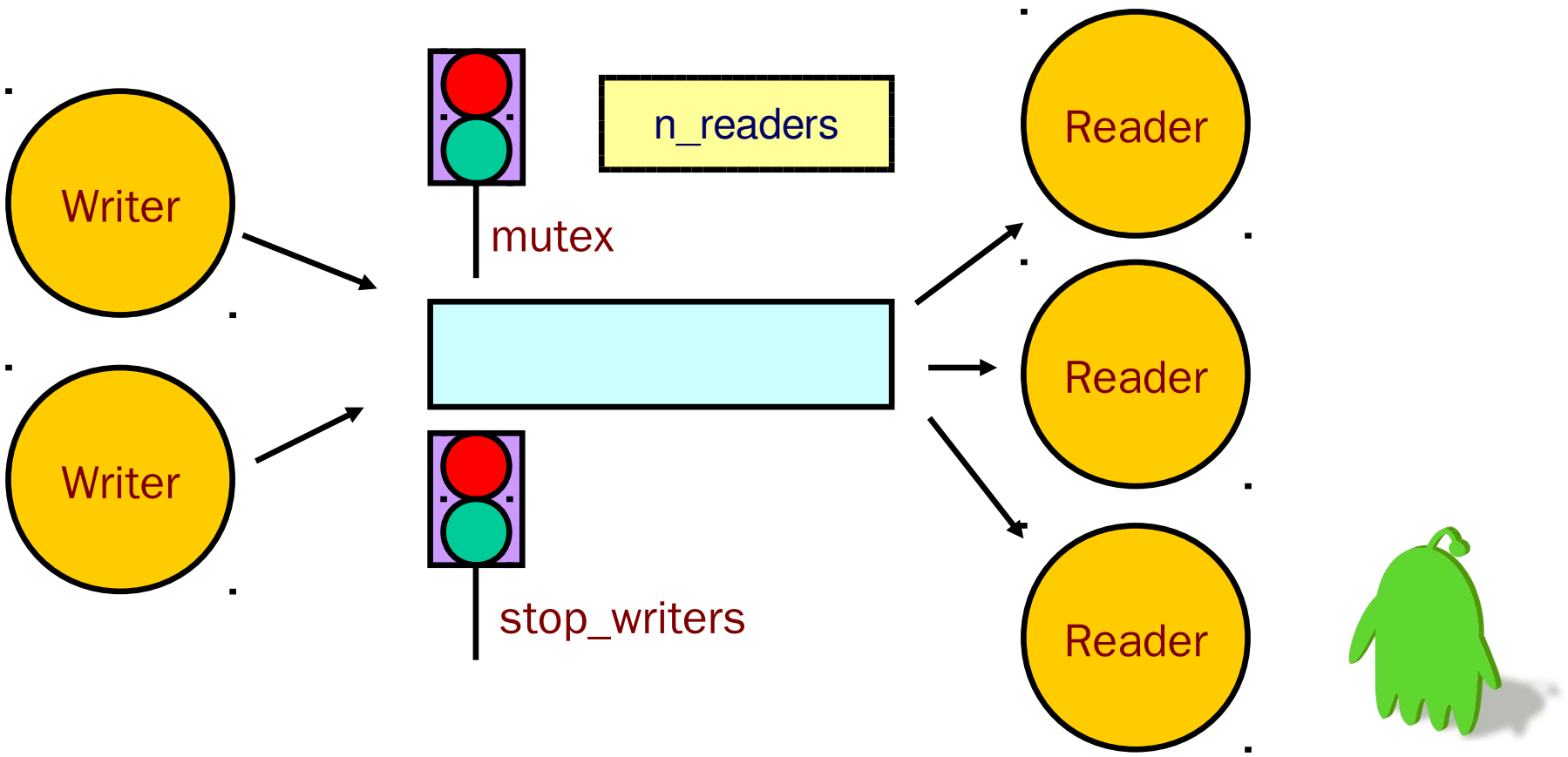
- Writer processes have to update shared data.
- Reader processes have to check the values of the data. The should all be able to read at the same time.



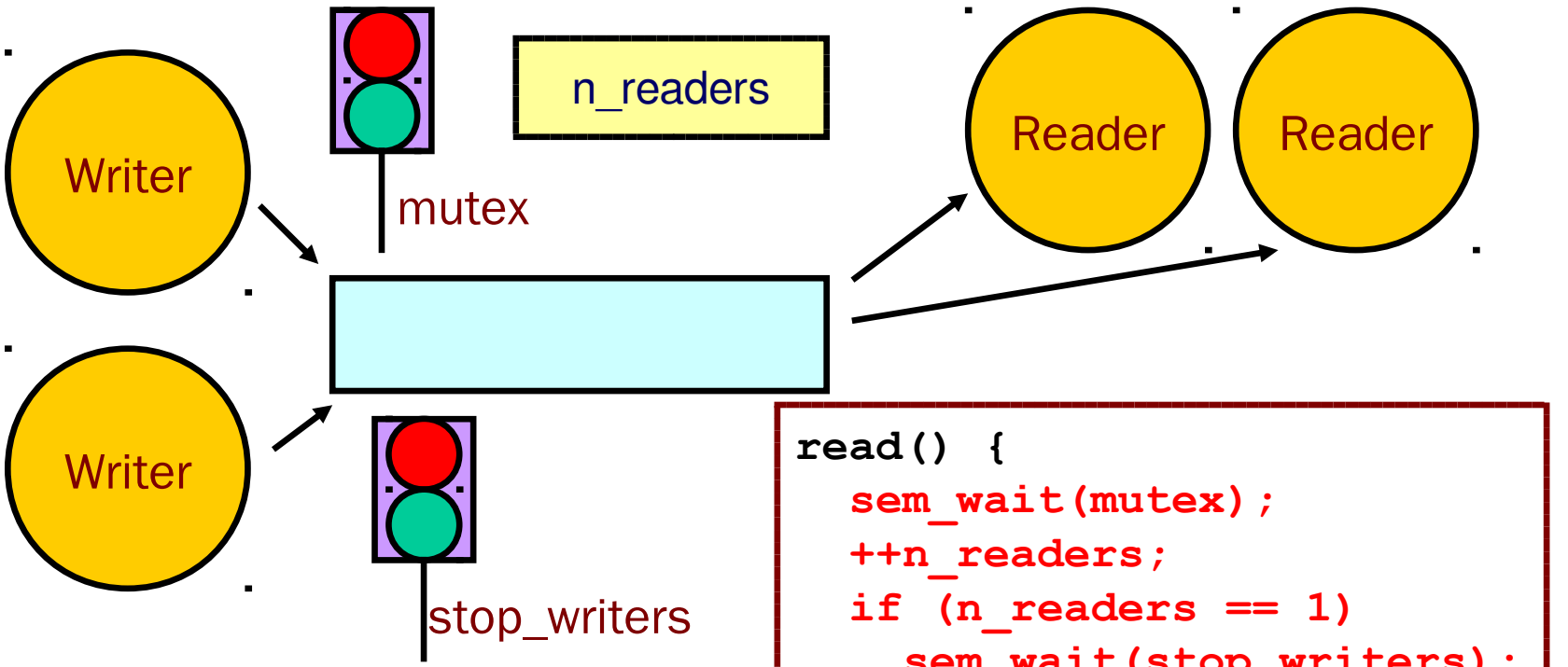
- Why is this different from the Producer/Consumer problem?
- Why not use a simple mutex?

Readers/Writers Problem

- We will need two semaphores:
  - stop the writers and guarantying mutual exclusion when a writer is updating the data
  - protect mutual exclusion of a shared variable that counts readers

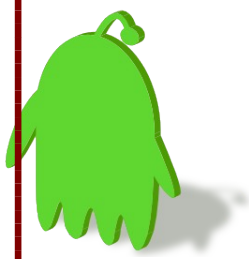


Readers/Writers Algorithm (priority to Readers)




```
write(e) {
   sem_wait(stop_writers);
   buffer = e;
   sem_post(stop_writers);
}
```

```
read() {
   sem_wait(mutex);
   ++n_readers;
   if (n_readers == 1)
      sem_wait(stop_writers);
   sem_post(mutex);

   e = buffer;

   sem_wait(mutex);
   --n_readers;
   if (n_readers == 0)
      sem_post(stop_writers);
   sem_post(mutex);
   return e;
}
```

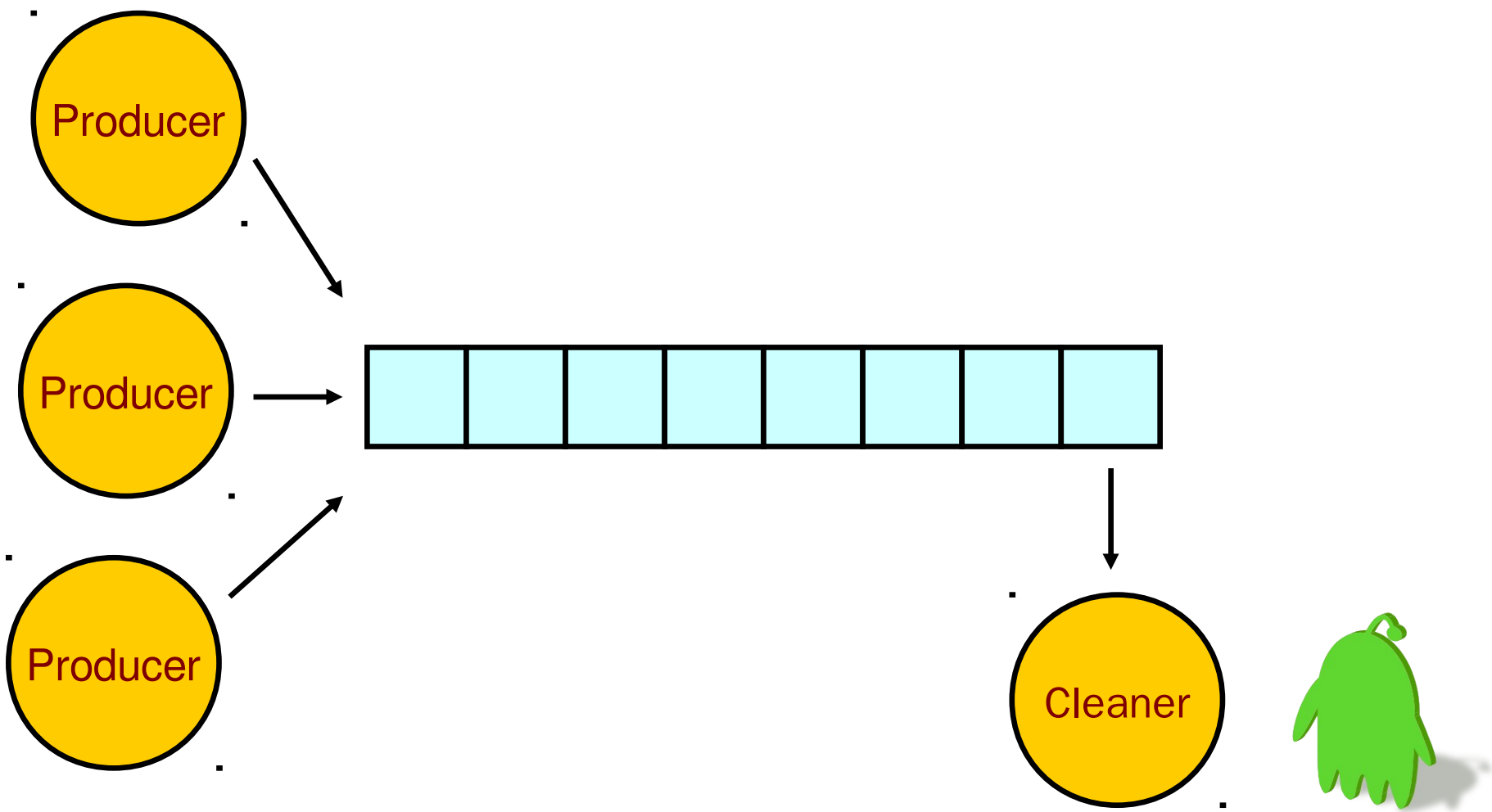


Considerations

- The previous algorithm gives priority to readers
  - Not always what you want to do
- There's a different version that gives priority to writers
- Why should I care?
  - This algorithm is the essential of all database systems! Concurrent reads of data; single update.
  - One bank agency deposits some money in an account; at the same time, all over the country, many agencies can be reading it
  - You are booking a flight. Although someone in England in also booking a flight, you and thousands of people can still see what are the available places in the place.

Buffer Cleaner Problem

- A buffer can hold a maximum of N elements. When it is full, it should be immediately emptied. While the buffer is being emptied, no thread can put things into it.



Synchronization: Basic Rules(1)

- Never Interlock waits!
  - Locks should always be taken in the same order in all processes
  - Locks should be released in the reverse order they have been taken

```
sem_wait(A)
sem_wait(B)

/* Critical
   Section */

sem_post(B)
sem_post(A)
```

Deadlock!

```
sem_wait(B)
sem_wait(A)

/* Critical
   Section */

sem_post(A)
sem_post(B)
```

- One way to assure that you always take locks in the same order is to create a **lock hierarchy**. I.e. associate a number to each lock using a table and always lock in increasing order using that table as reference (index).

Synchronization: Basic Rules(2)

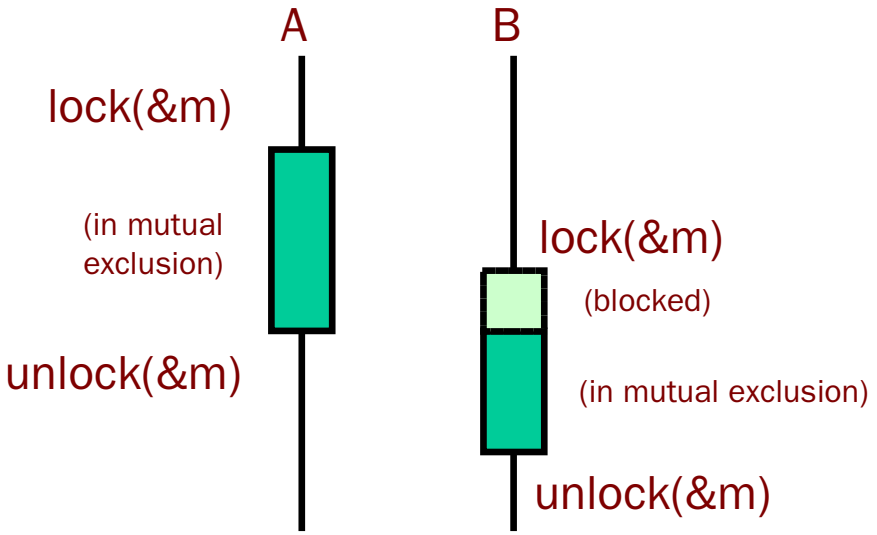
- Sometimes it is not possible to know what order to take when locking (or using semaphores)
  - Example: you are using two resources owned by the operating system. They are controlled by locks. You cannot be sure if another application is not using exactly the same resources and locking in reverse order.

- In that case, use pthread_mutex_trylock() or sem_trywait() and back off if you are unsuccessful.
  - Allow the system to make progress and not deadlock!

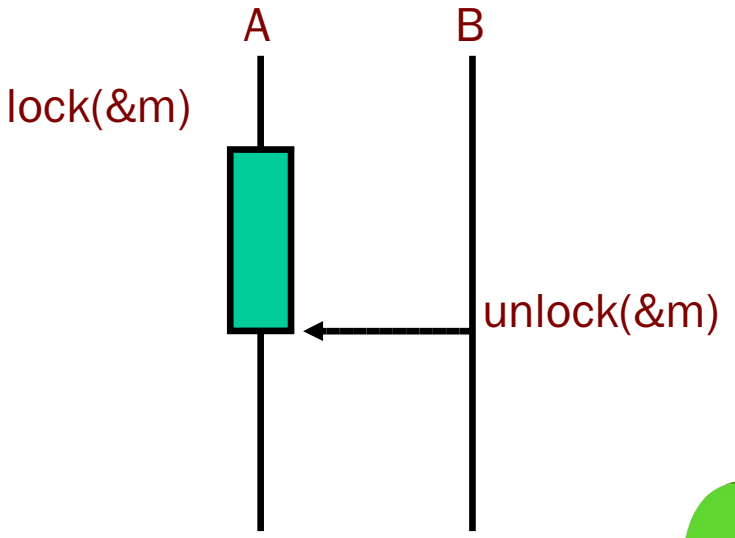
```
// Try to acquire both resources
while (true)
{
  // Acquire the first resource
  pthread_mutex_lock(&lockA);

  // Try to acquire the second one
  if (pthread_mutex_trylock(&lockB) != 0)
  {
    // Failed, back off
    pthread_mutex_unlock(&lockA);
    usleep(BACKOFF_DELAY);
  }
  else
    break;
}

// In mutual exclusion
// ...

// Release the resources
pthread_mutex_unlock(&lockB);
pthread_mutex_unlock(&lockA);
```

Synchronization: Basic Rules(3)

- Mutexes are used for implementing mutual exclusion, not for signaling across threads!!!
  - Only the thread that has locked a mutex can unlock it. Not doing so will probably result in a core dump!
- To signal across threads use semaphores!



CORRECT!



INCORRECT!

Important Concepts

- Deadlock
  - When two or more processes are unable to make progress being blocked waiting for each other
- Livelock
  - When two or more processes are alive and working but are unable to make progress
- Starvation
  - When a process is not being able to access resources that its needs to make progress

Thread Priority

- The scheduling problem applies to sleep queues as well.
- Which thread should get a mutex next? Which thread should wakeup on a signal?
- Should priority matter?
- What if a high-priority thread is waiting for a mutex held by a low-priority thread? This is called priority inversion.

Mars PathFinder Problem: Priority Inversion

Mars Pathfinder

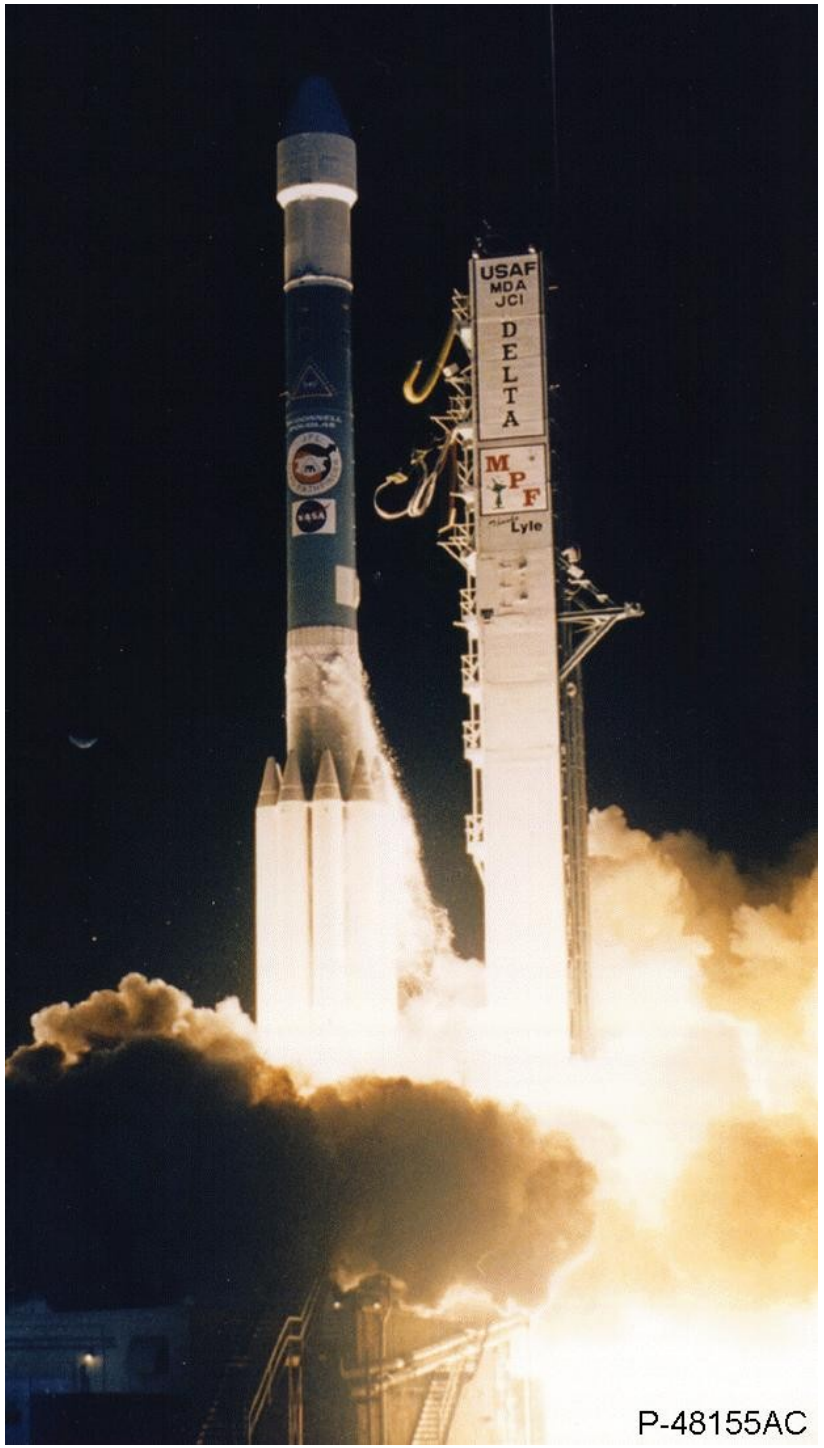
- Mission
  - Demonstrate new landing techniques: parachute and airbags
  - Take pictures
  - Analyze soil samples
  - Demonstrate mobile robot technology: Sojourner
- Major success on all fronts
  - Returned 2.3 billion bits of information
  - 16,500 images from the Lander
  - 550 images from the Rover
  - 15 chemical analyses of rocks & soil
  - Lots of weather data
  - Both Lander and Rover outlived their design life

Operators: NASA and JPL (Jet Propulsion Laboratory)

Low-cost (~$150 million) planetary discovery mission

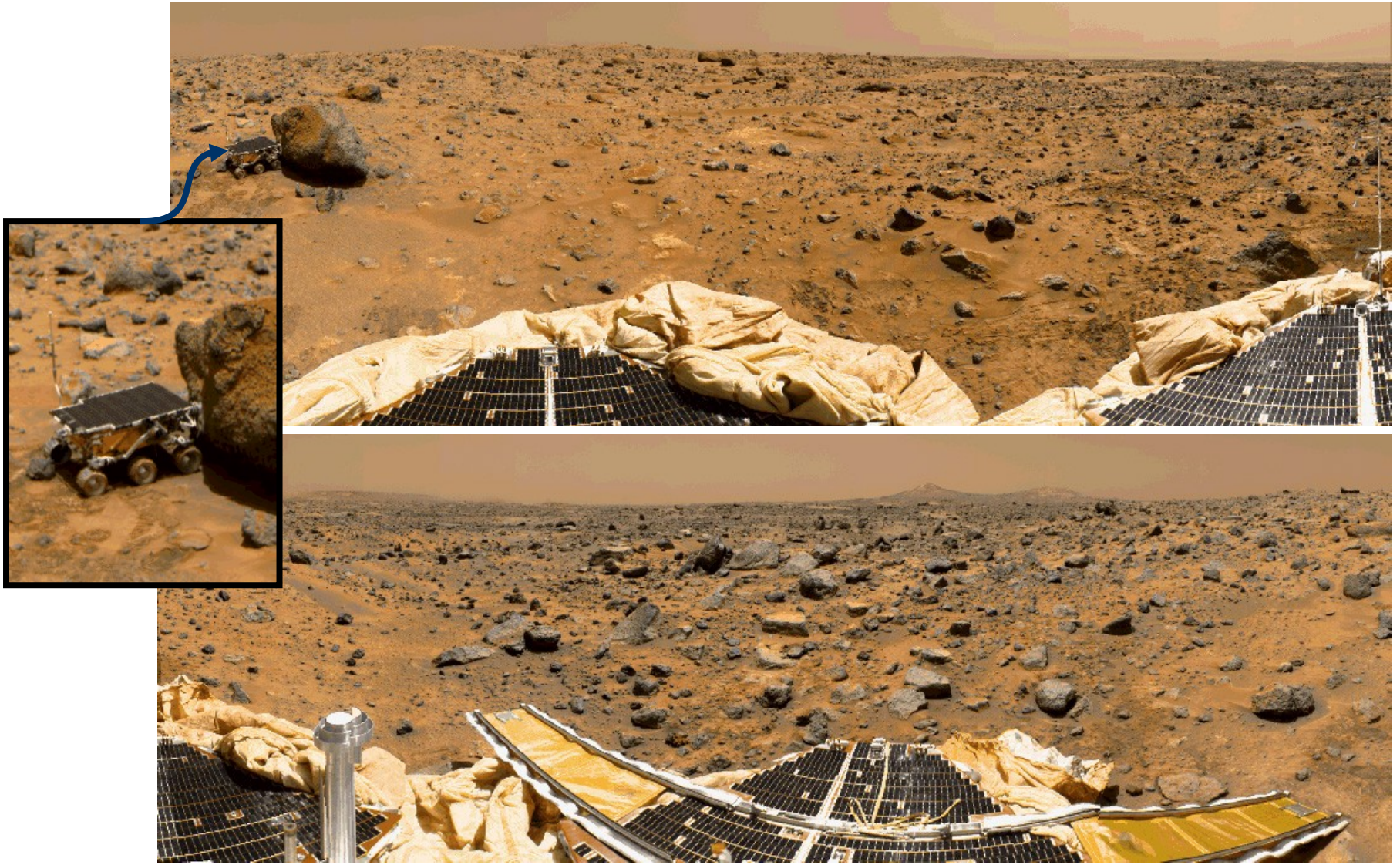
Sojourner Rover

Mars Pathfinder



- Mars Pathfinder was originally designed as a technology demonstration of a way to deliver an instrumented lander and a free-ranging robotic rover to the surface of the red planet.
- Due to limited funds, Pathfinder's development had to be dramatically different from the way in which previous spacecraft had been developed.
- Instead of the traditional 8- to 10-year schedule and $1-billion-plus budget, Pathfinder was developed in 3 years for **less than $150 million =** the cost of some Hollywood movies!

Pictures taken from an early Mars rover



Mars Pathfinder Timeline

- November 16, 1996
  - Russian Mars '96 orbiter/landers launched.
- November 17, 1996
  - Mars '96 fails to achieve insertion into Mars cruise trajectory and re-enters the Earth's atmosphere.
- December 4, 1996
  - Mars Pathfinder launched.
- July 4, 1997
  - Mars Pathfinder lands on Mars and begins a successful mission.
- September 27, 1997
  - last successful data transmission from Mars Pathfinder

New York Times: July 15, 1997

The New York Times
nytimes.com

July 15, 1997

Mars Craft Again Halts Transmissio

The computer aboard the Mars Pathfinder overloaded and reset itself early

The mishap delayed chemical analysis of a tubby rock named Yogi, but no

Mary Beth Murrill, a spokeswoman for NASA's Jet Propulsion Laboratory
personal computer "when we ask it to do too many things at once."

The project manager, Brian Muirhead, said that to prevent a recurrence, co
had reset itself while trying to carry out several activities at once.

The first one occurred on Friday night, delaying the chemical analysis of Yo

In response, controllers reprogrammed the computer over the weekend to

But today, about an hour into a two-hour transmission session, it happened

Mr. Muirhead said that before the problem occurred, Pathfinder had succe
same time the spacecraft was sending down images, it was also collecting a

Controllers could not go back and receive the rest of the color panorama d
visiting Jupiter.

If the data from Sojourner's analysis of the rock's chemical makeup is resto

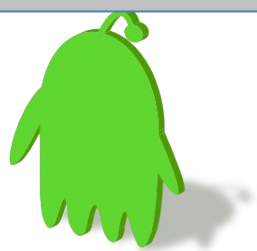
Sojourner's examination of the first rock was delayed for several days, first
instructions being sent to the spacecraft. Then on Friday, the computer rese

NASA planned a news conference on Tuesday to release the latest images

Mary Beth Murrill, a spokeswoman for NASA's Jet Propulsion Laboratory, said transmission of the panoramic shot took “a lot of processing power.” She likened the data overload to what happens with a personal computer “when we ask it to do too many things at once.”

The project manager, Brian Muirhead, said that to prevent a recurrence, controllers would schedule activities one after another, instead of at the same time. It was the second time the Pathfinder's computer had reset itself while trying to carry out several activities at once.

In response, controllers reprogrammed the computer over the weekend to slow down the rate of activities and avoid another reset. But today, about an hour into a two-hour transmission session, it happened again.

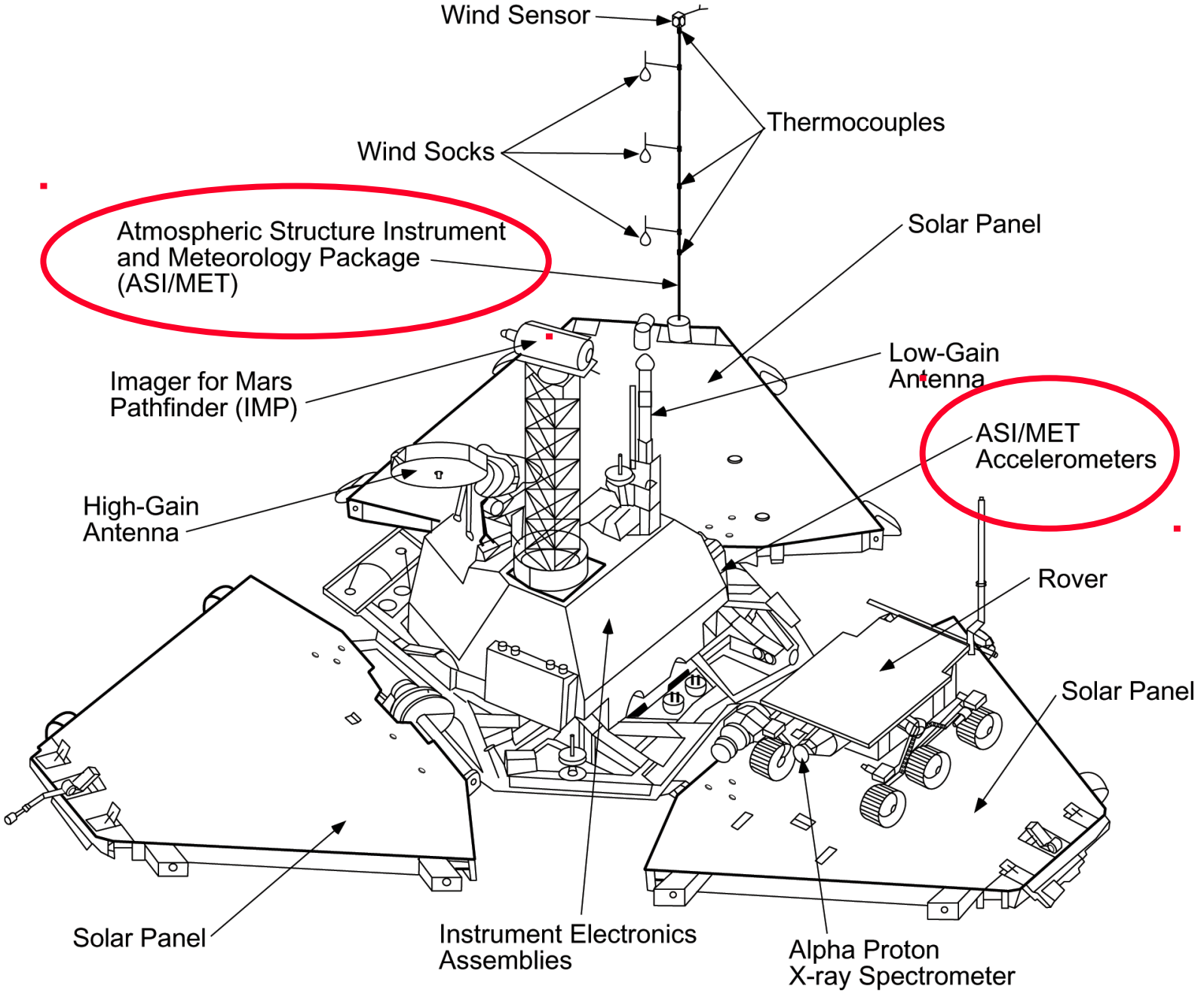


Pathfinder Configurations

- Spacecraft ran IBM RS6000 processor and WindRiver's VxWorks RTOS
  - **20 MIPS 128 MB of DRAM**

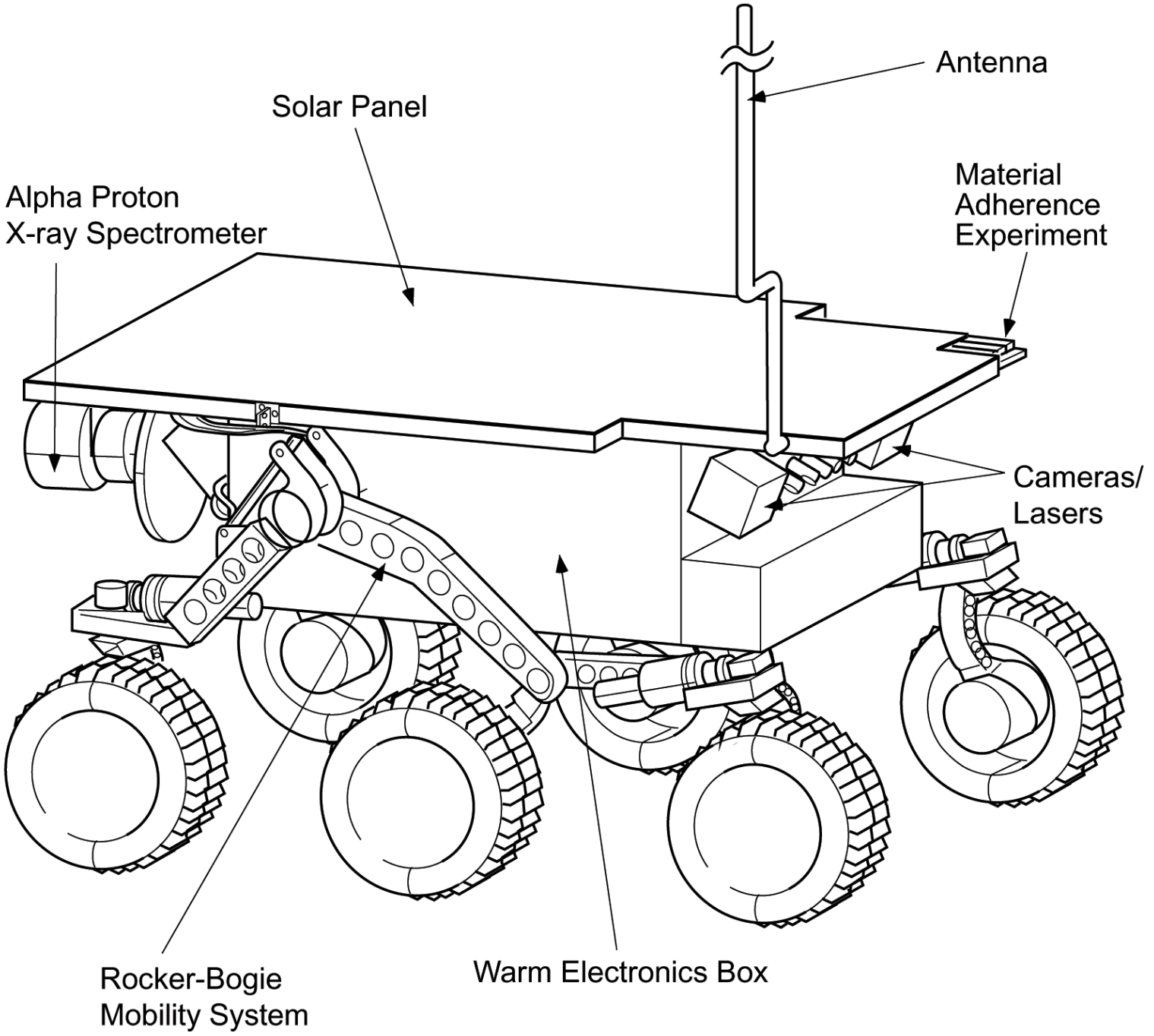
    for storage of flight software and engineering and science data, including images and rover information.
  - **6 MB ROM**

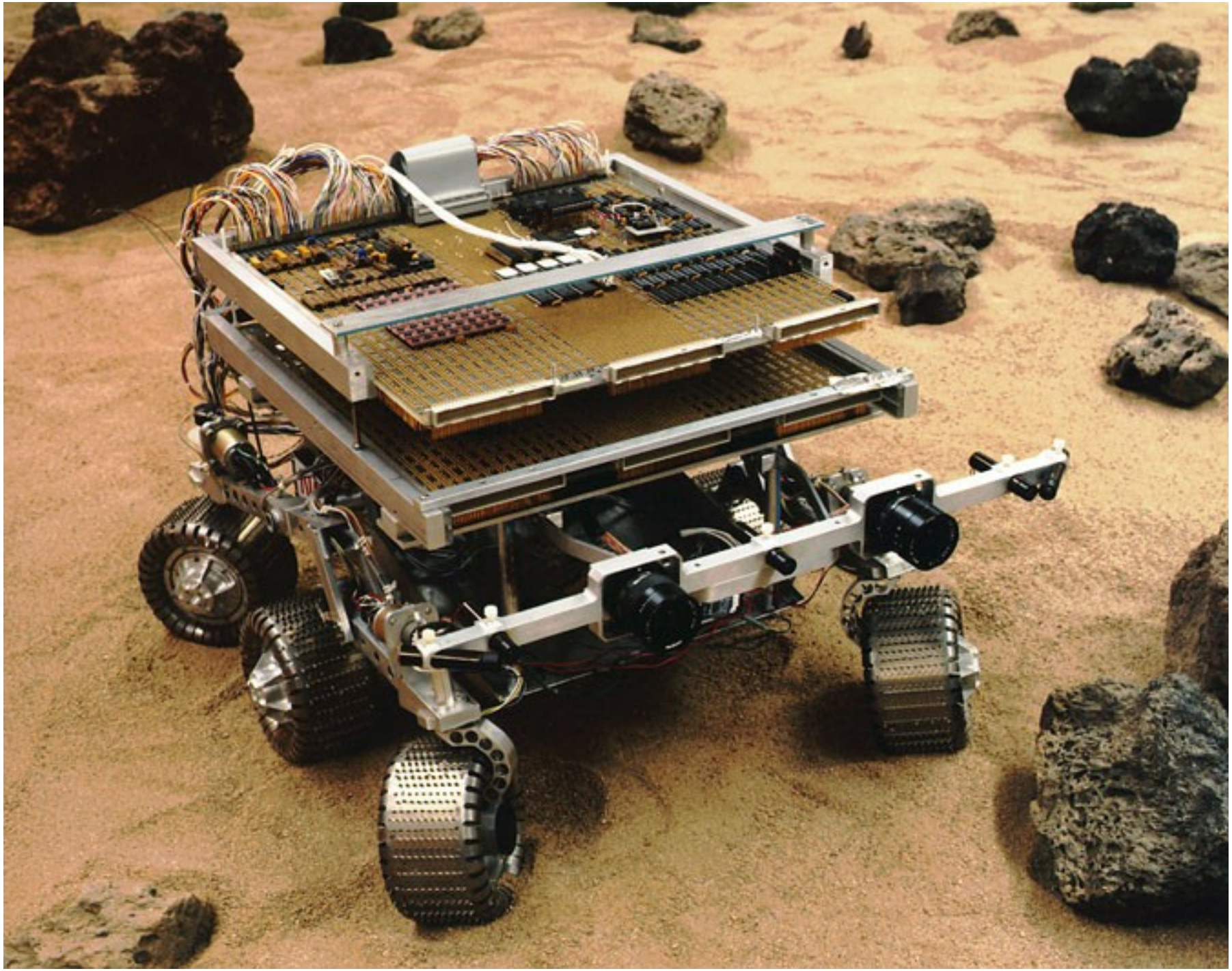
    stored flight software and time-critical data.
- Hard real-time OS with concurrent execution of thread
  - Threads have priorities and are preemptible
- Tasks on Pathfinder were executed as threads with priorities
  - that were assigned reflecting the relative urgency of tasks.
- Pathfinder contained an "information bus”
  - a shared memory area used for passing information between different components of the spacecraft.

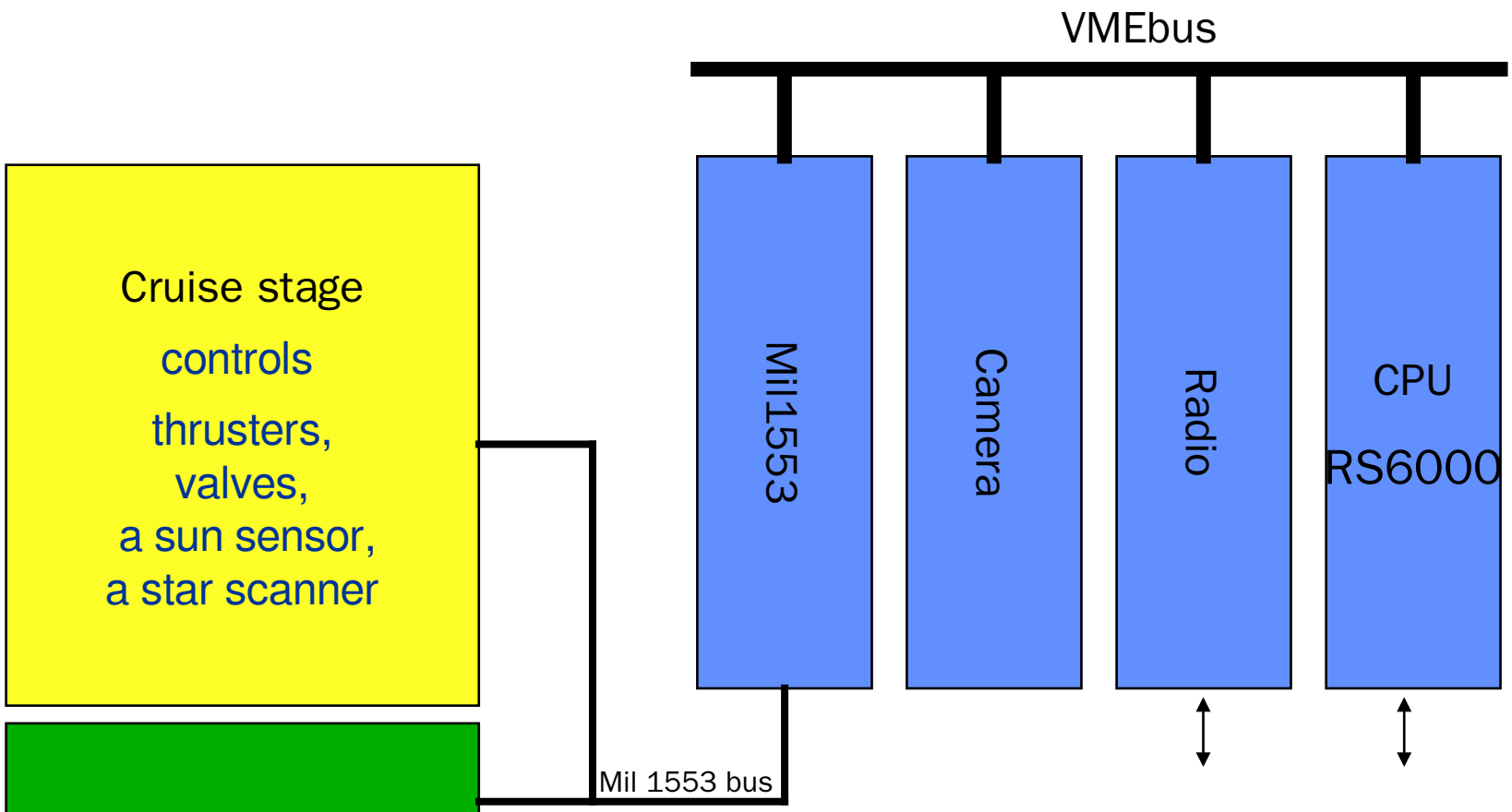

Wind Sensor
Thermocouples
Wind Socks
Atmospheric Structure Instrument and Meteorology Package (ASI/MET)
Solar Panel
Low-Gain Antenna
Imager for Mars Pathfinder (IMP)
ASI/MET Accelerometers
High-Gain Antenna
Rover
Solar Panel
Solar Panel
Instrument Electronics Assemblies
Alpha Proton X-ray Spectrometer

Rover Sojourner

- The rover, capable of autonomous navigation and performance of tasks, communicated with Earth via the lander.
- Sojourner's control system was built around an **Intel 80C85**, with a computing speed of **0.1 MIPS and 500 KB of RAM.**

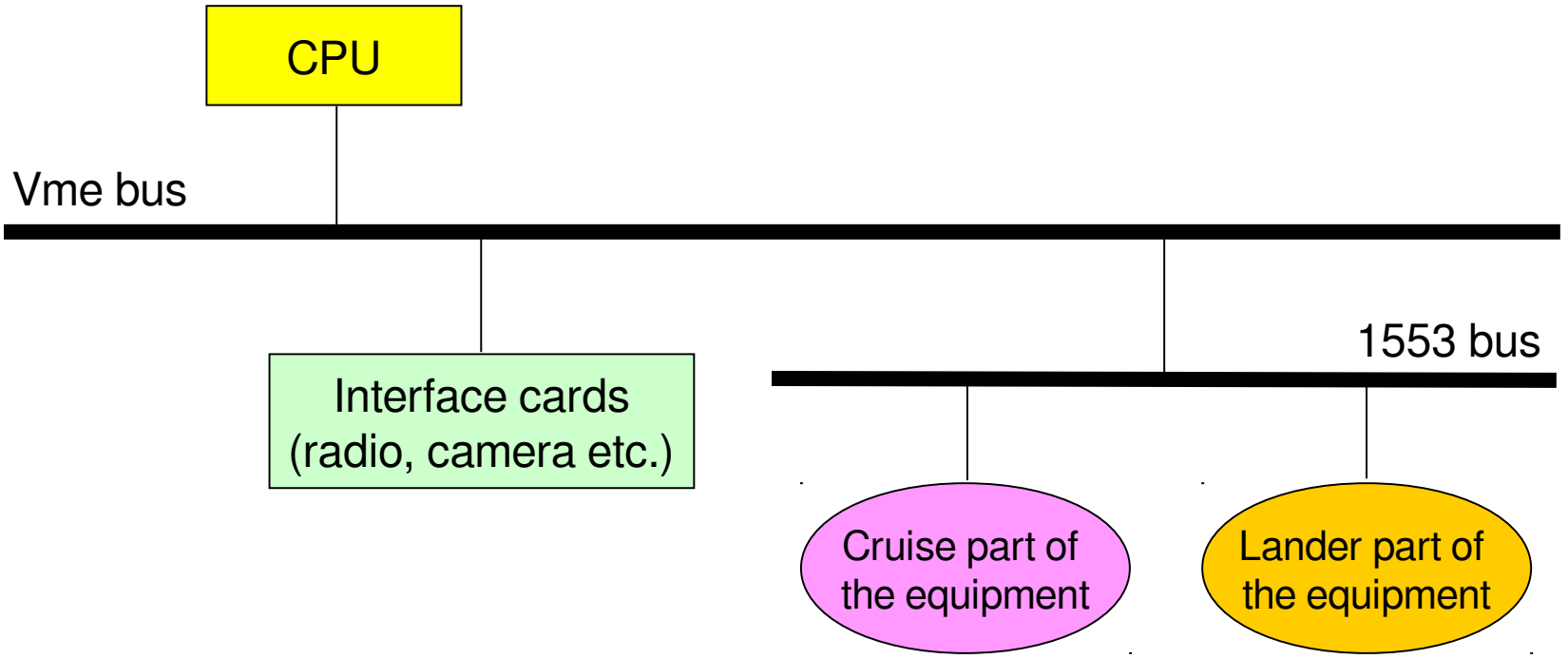

Antenna
Solar Panel
Alpha Proton
X-ray Spectrometer
Material
Adherence
Experiment
Cameras/
Lasers
Rocker-Bogie
Mobility System
Warm Electronics Box





Mil1553: specific paradigm:
the software will schedule activity at an 8 Hz rate.
This **feature** dictated the architecture of the software
which controls both the 1553 bus and the devices attached to it.

Pathfinder Configurations



Pathfinder used VxWorks RTOS

- Threads for the 1553 bus for data collection, scheduled on every 1/8th sec cycle.
- 3 periodic tasks
  - Task 1 – Information Bus Thread: Bus Manager

    high frequency, high priority
  - Task 2 – Communication Thread

    medium frequency / priority, high execution time
  - Task 3 – Weather Thread: Geological Data Gatherer

    low frequency, low priority
- Each checks if the other executed and completed in the previous cycle
  - If the check fails, this is a violation of a hard real-time guarantee and the system is reset

NASA Pathfinder

- fault-tolerance
  - a watchdog timer was used to reset the system in the event that the computer / software locks up
  - essential design feature (no going to Mars to reboot)
  - “watched” for hang ups on the highest priority task
- inter-task communication
  - a shared resource (memory) was used to pass data from the data gatherer (task 3) to the communicator (task 2) via the bus manager (task 1).

Pathfinder Problem

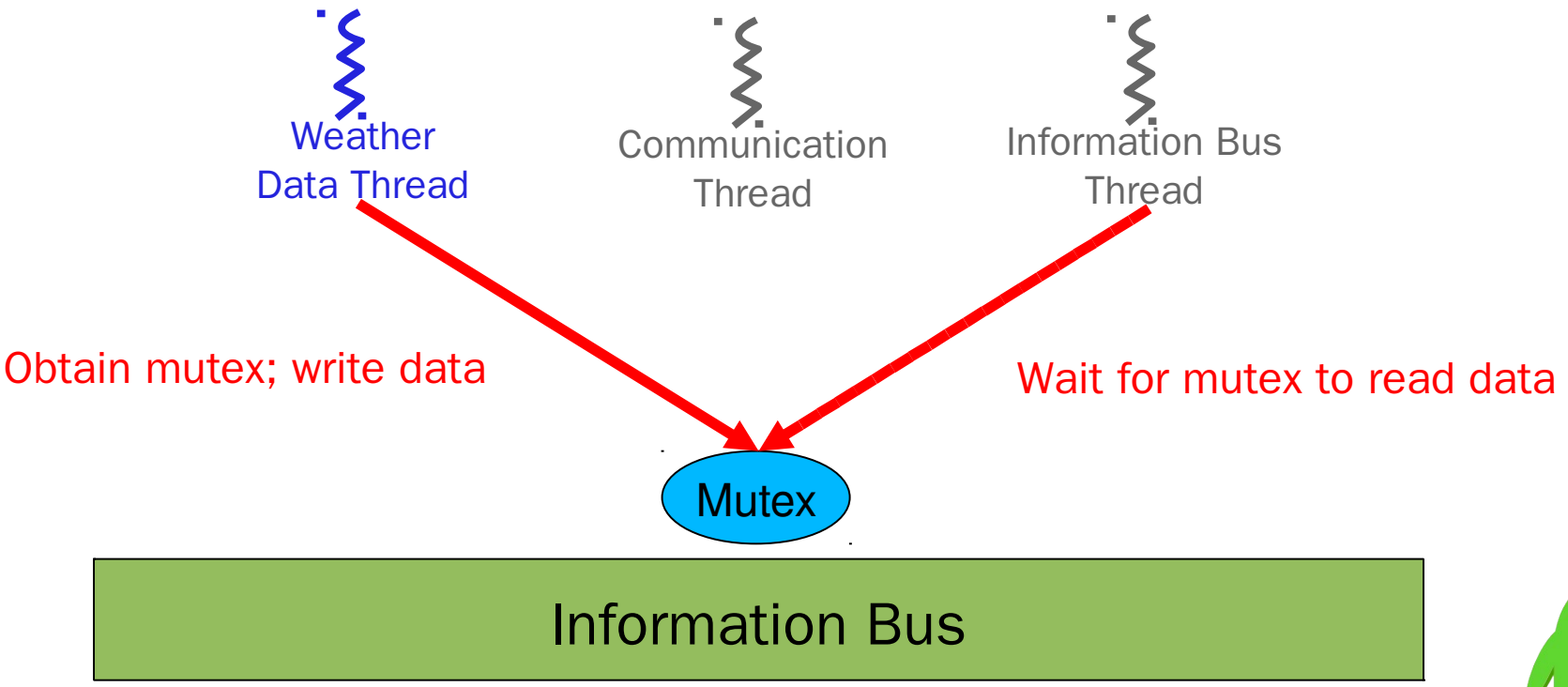
- Within a few days of landing, when Pathfinder started gathering meteorological data, spacecraft began experiencing ***total system resets***
- This resulted in loss of data collected during each cycle
- JPL engineers had exact replica of the spacecraft in their lab
- They turned on the tracing feature of VxWorks
  - All system events such as context switches, uses of synchronization objects, and interrupts traced.
  - Tracing disabled on the actual spacecraft because generates too much data volume.
- After 18 hours of execution, early next morning when all but one engineer had gone home, the symptom was reproduced.

Pathfinder Problem

- Most of the time this combination worked fine
- However, with the following scenario:
  - Data gathering occurs, grab the bus
  - Shared memory buffer full, retrieval to private memory
    - This is blocked because of the bus mutex
  - Period communication task is issued
    - This is preempted because of lower priority
  - Data gathering task takes its time
  - Retrieval task time out due to watchdog timer
    - System reset!

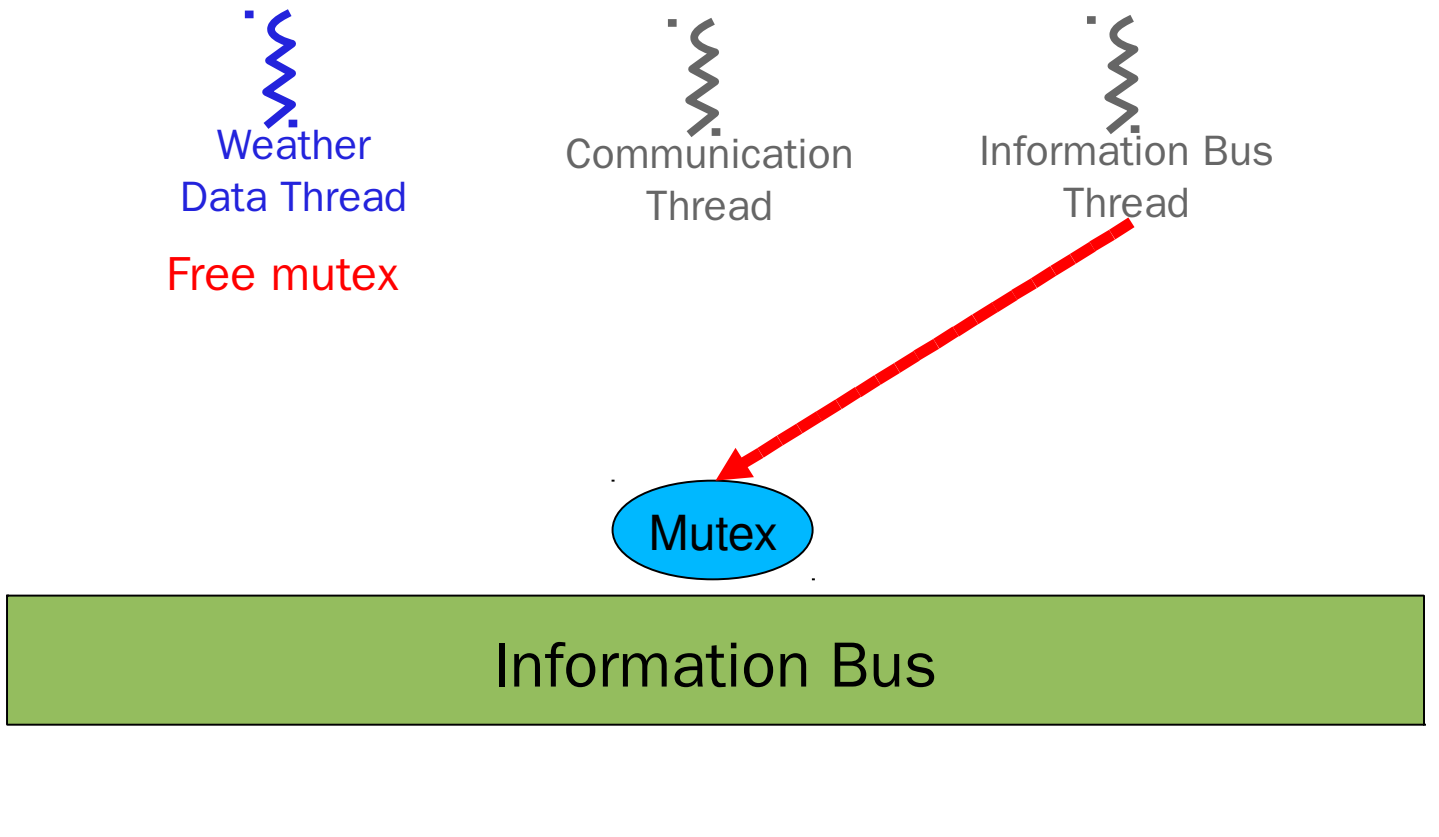
VxWorks RTOS

- Multiple tasks, each with an associated *priority*
  - Higher priority tasks get to run before lower-priority tasks
- Information bus – shared memory area used by various tasks
  - Thread must obtain mutex to write data to the info bus – a *monitor*



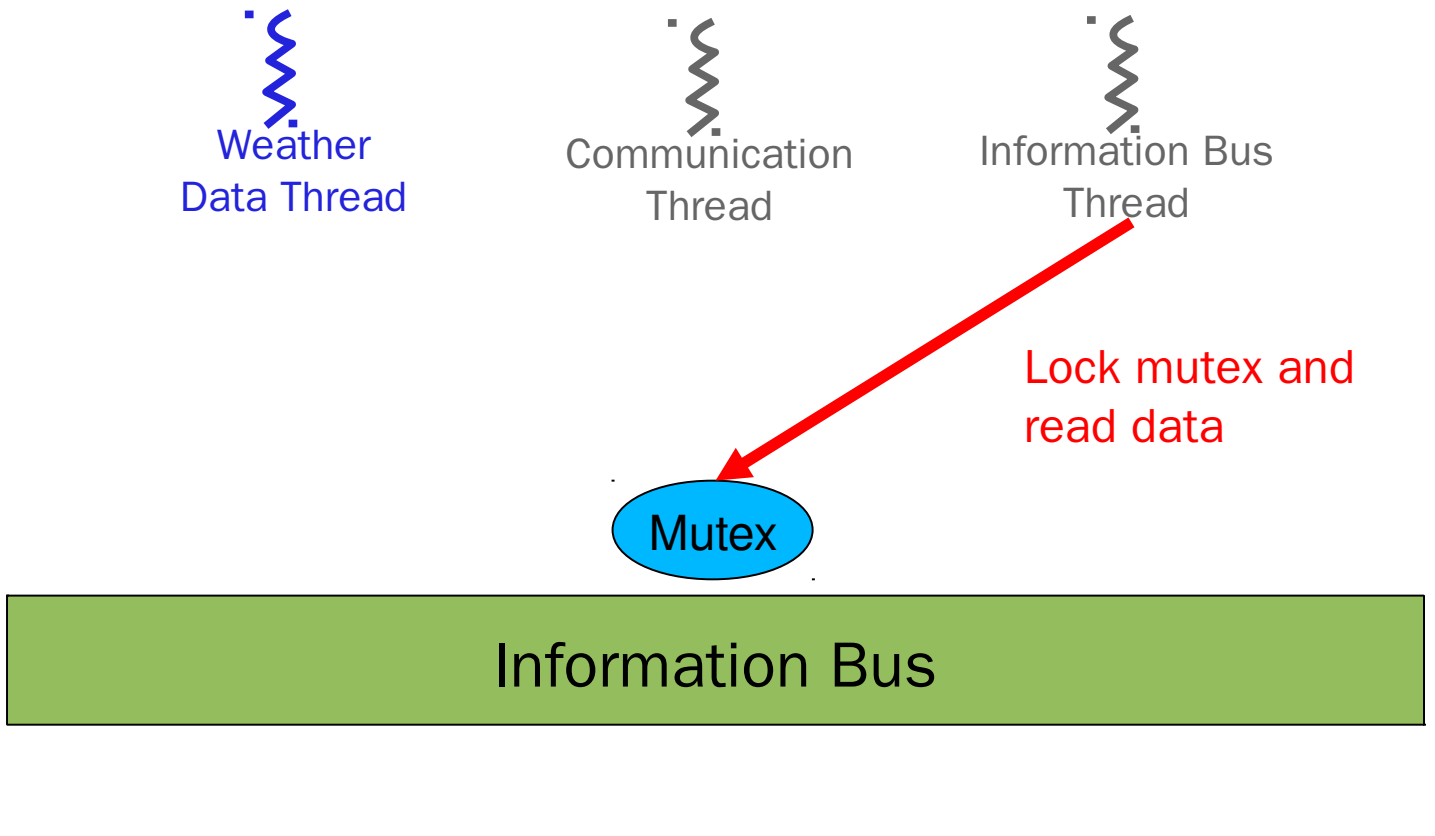
VxWorks RTOS

- Multiple tasks, each with an associated *priority*
  - Higher priority tasks get to run before lower-priority tasks
- Information bus – shared memory area used by various tasks
  - Thread must obtain mutex to write data to the info bus – a *monitor*

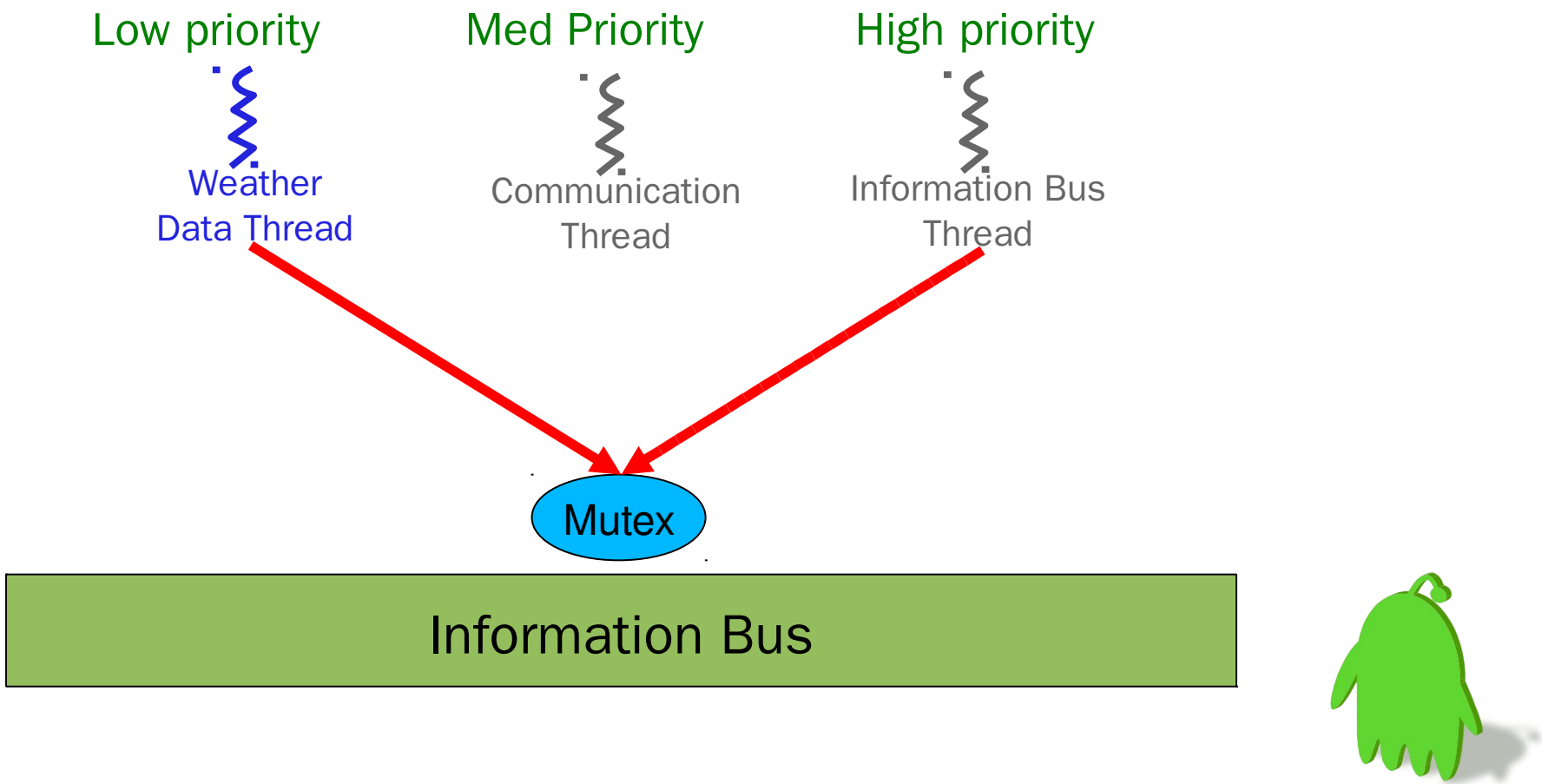


VxWorks RTOS

- Multiple tasks, each with an associated *priority*
  - Higher priority tasks get to run before lower-priority tasks
- Information bus – shared memory area used by various tasks
  - Thread must obtain mutex to write data to the info bus – a *monitor*



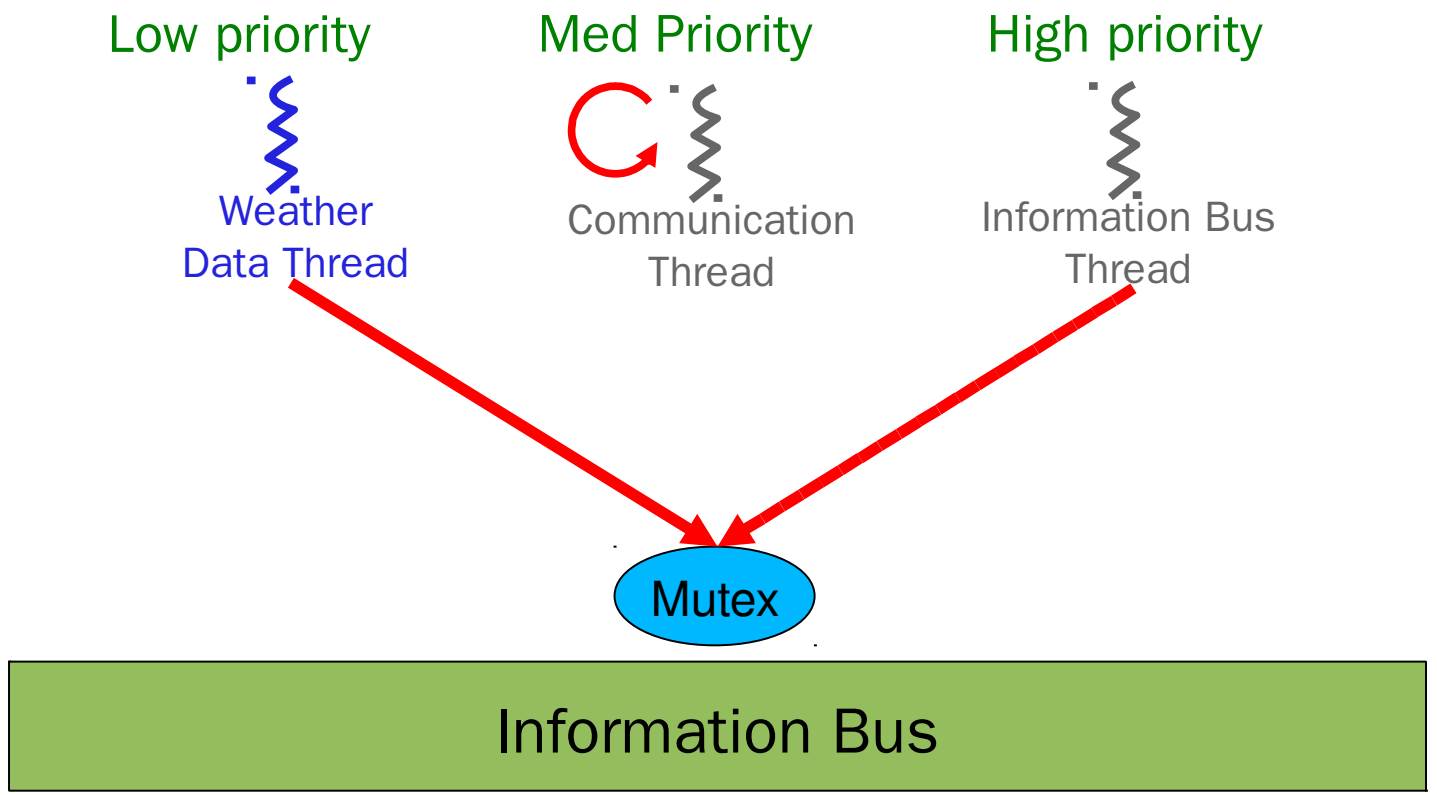
Priority Inversion



Priority Inversion

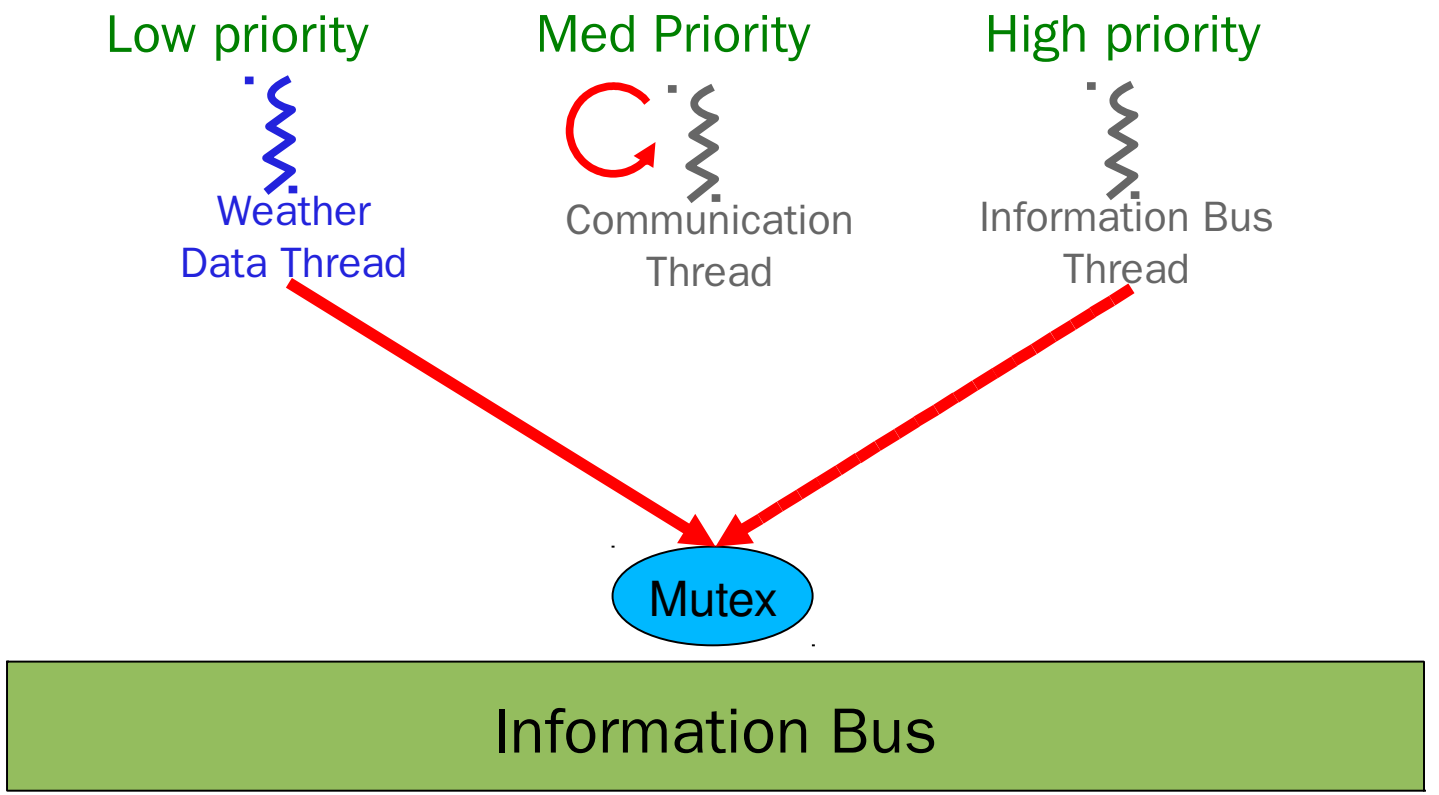
Interrupt!

Schedule comm thread … long running operation



Priority Inversion

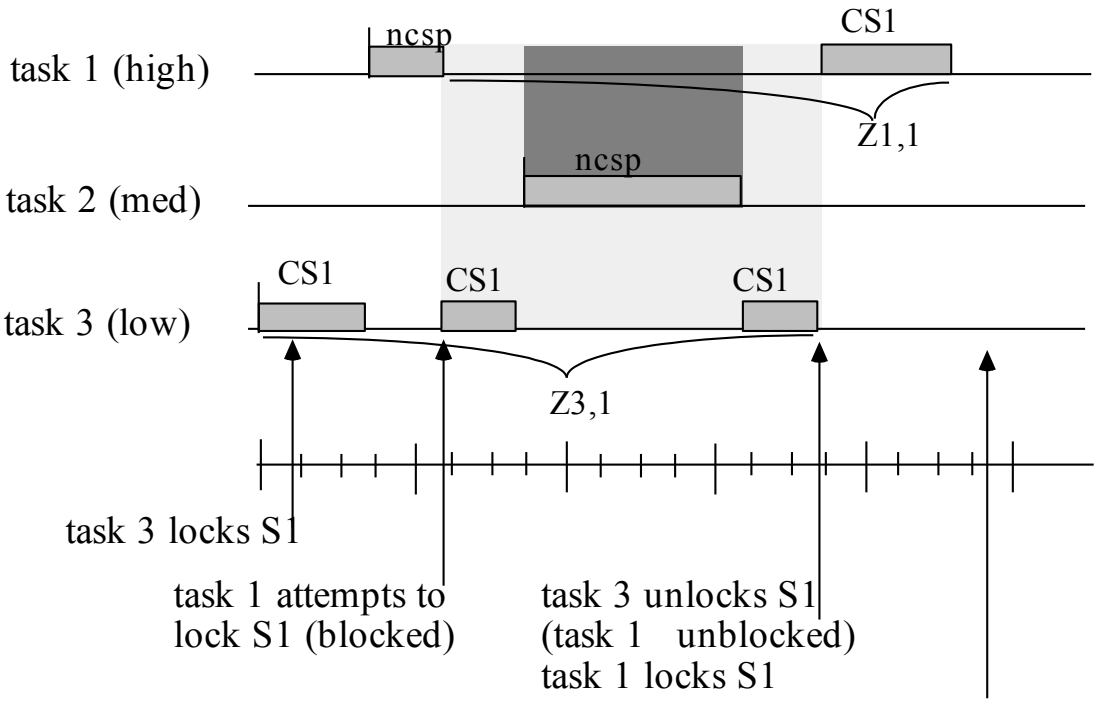
- What happens when threads have different priorities?
- Comm thread runs for a long time
- Comm thread has *higher priority* than weather data thread
- But ... the high priority info bus thread is stuck *waiting!*
- This is called **priority inversion**



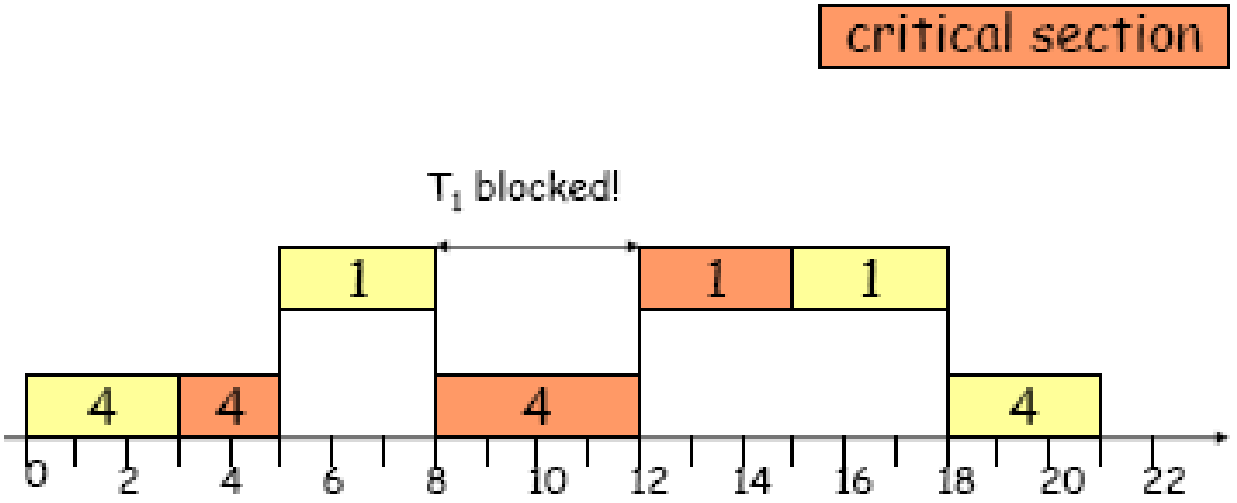
Pathfinder incident: Priority Inversion

Classical priority inversion problem due to shared system bus!

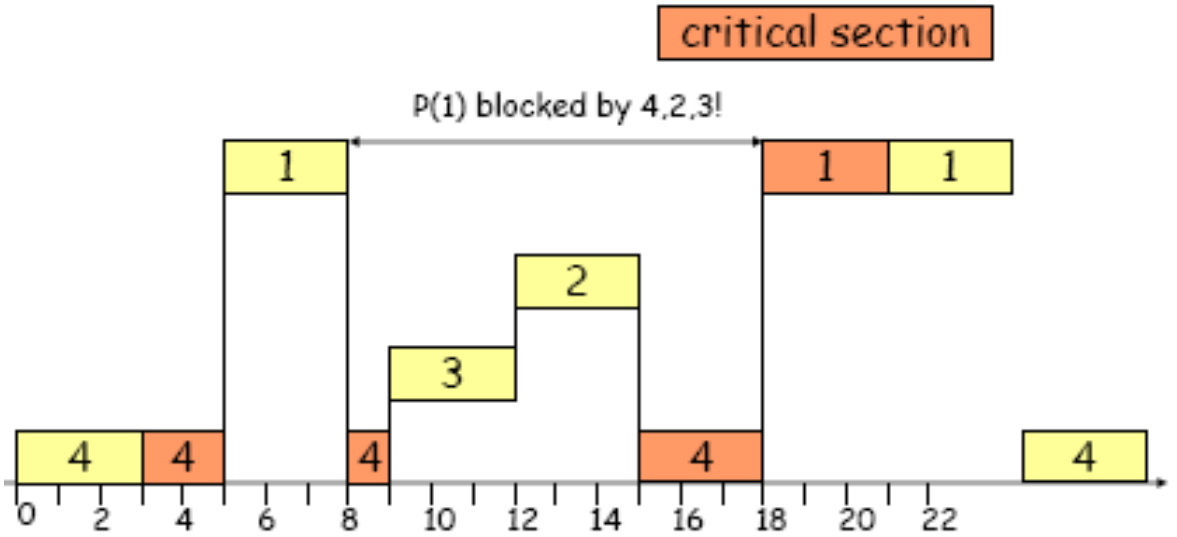
- tasks 1 and 3 share a resource (S1)
- prio(task1) >prio(task2) >prio(task3)
- Task 2 can run for any amount of time... it blocks Task 3 from finishing and unlocking resource needed by task 1.



Priority Inversion

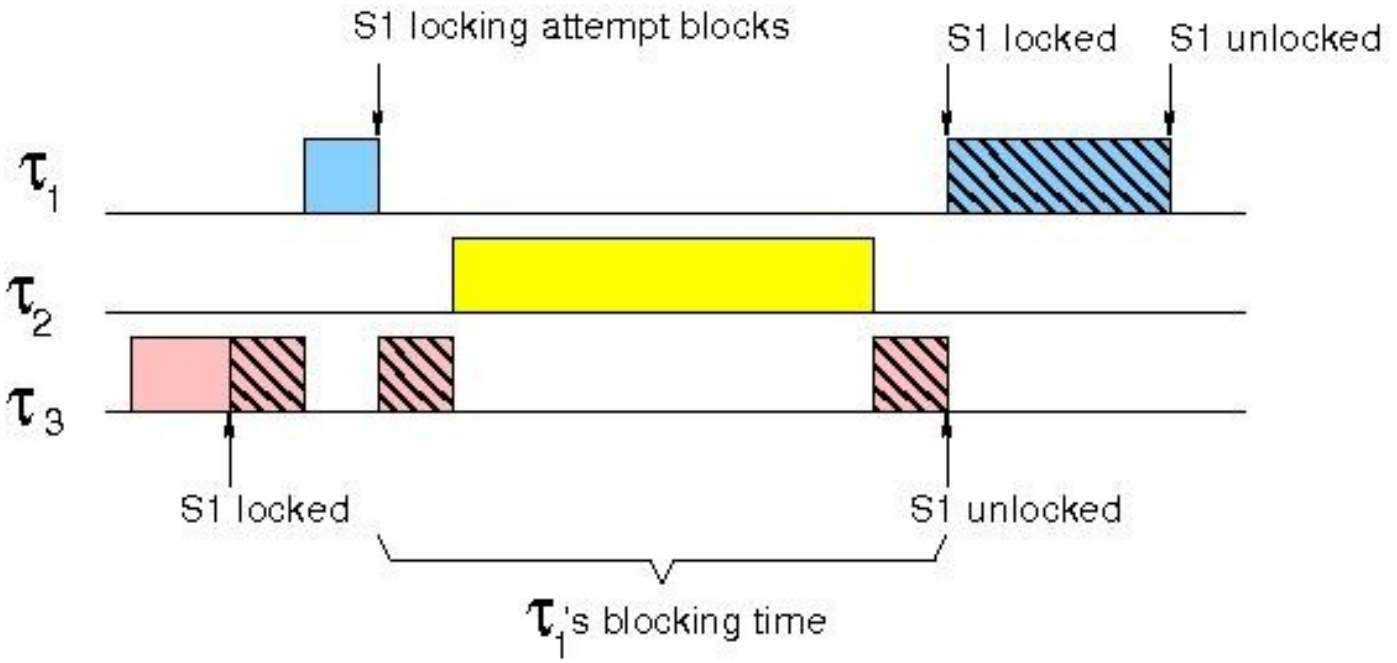


Unbounded Priority Inversion



Variant: Air transportation

- Priority(my_plane) < Priority(obama_plane)
- I arrived at the airport first and called sem_wait(&runway_mutex).
- Obama arrived at the airport after me and called sem_wait(&runway_mutex).
- My plane stopped working.
- Obama now has to wait for the engineers to fix my plane and for my plane to take off.



Look into deeper

- **VxWorks 5.x**
- 2 tasks to control the 1553 bus and the attached instruments.
  - **bc_sched task** (called the bus scheduler)

    a task controlled the setup of transactions on the 1553 bus
  - **bc_dist task** (for distribution) task

    also referred as the “communication task”
    - handles the collection of the transaction results i.e. the data.

Marsrobot general communication pattern

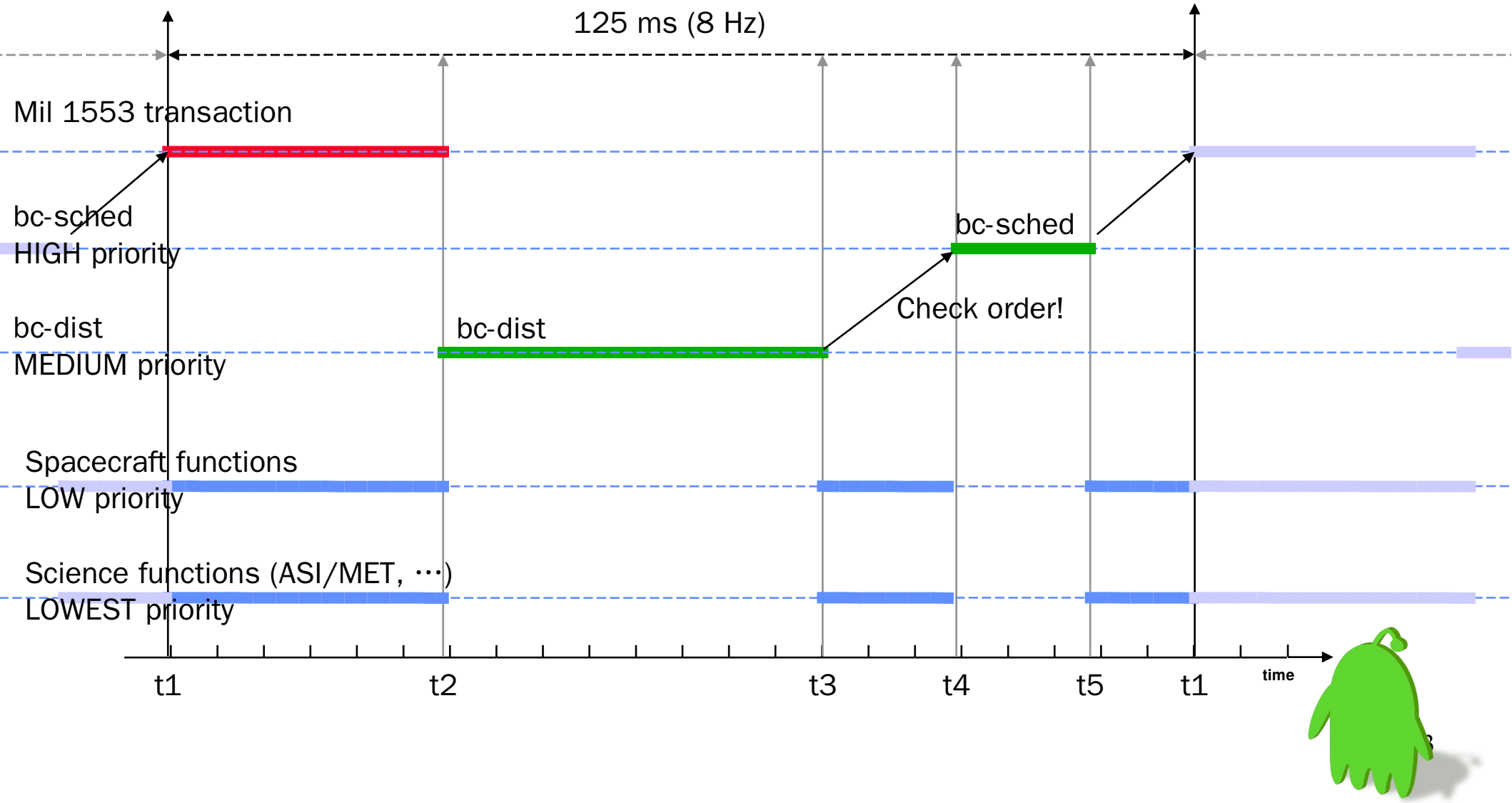
t1 - bus hardware starts via hardware control on the 8 Hz boundary.
The transactions for the this cycle had been set up by the previous execution of the bc_sched task.

t2 - 1553 traffic is complete and the bc_dist task is awakened.

t3 - bc_dist task has completed all of the data distribution

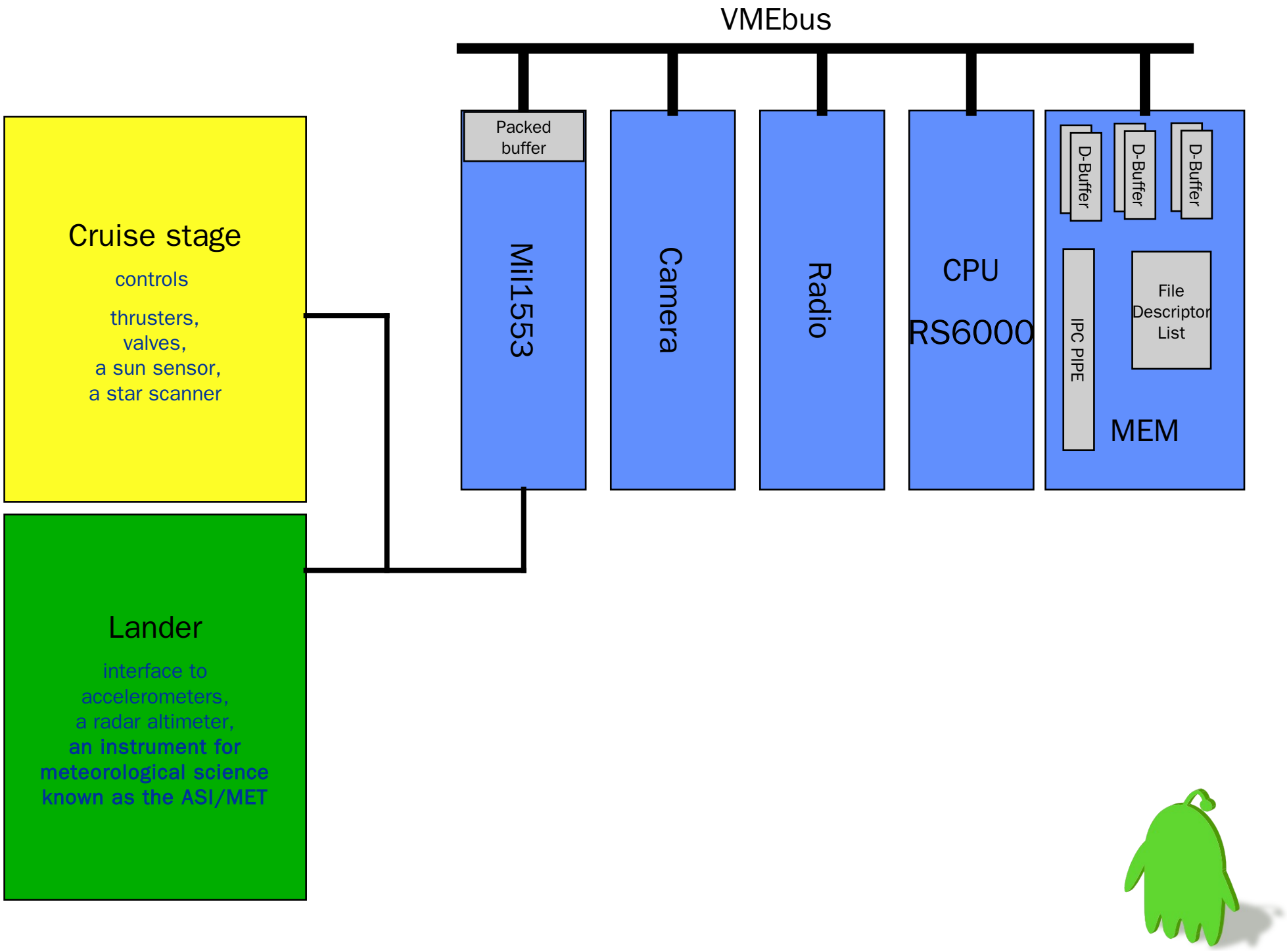
t4 - bc_sched task is awakened to setup transactions for the next cycle

t5 - bc_sched activity is complete

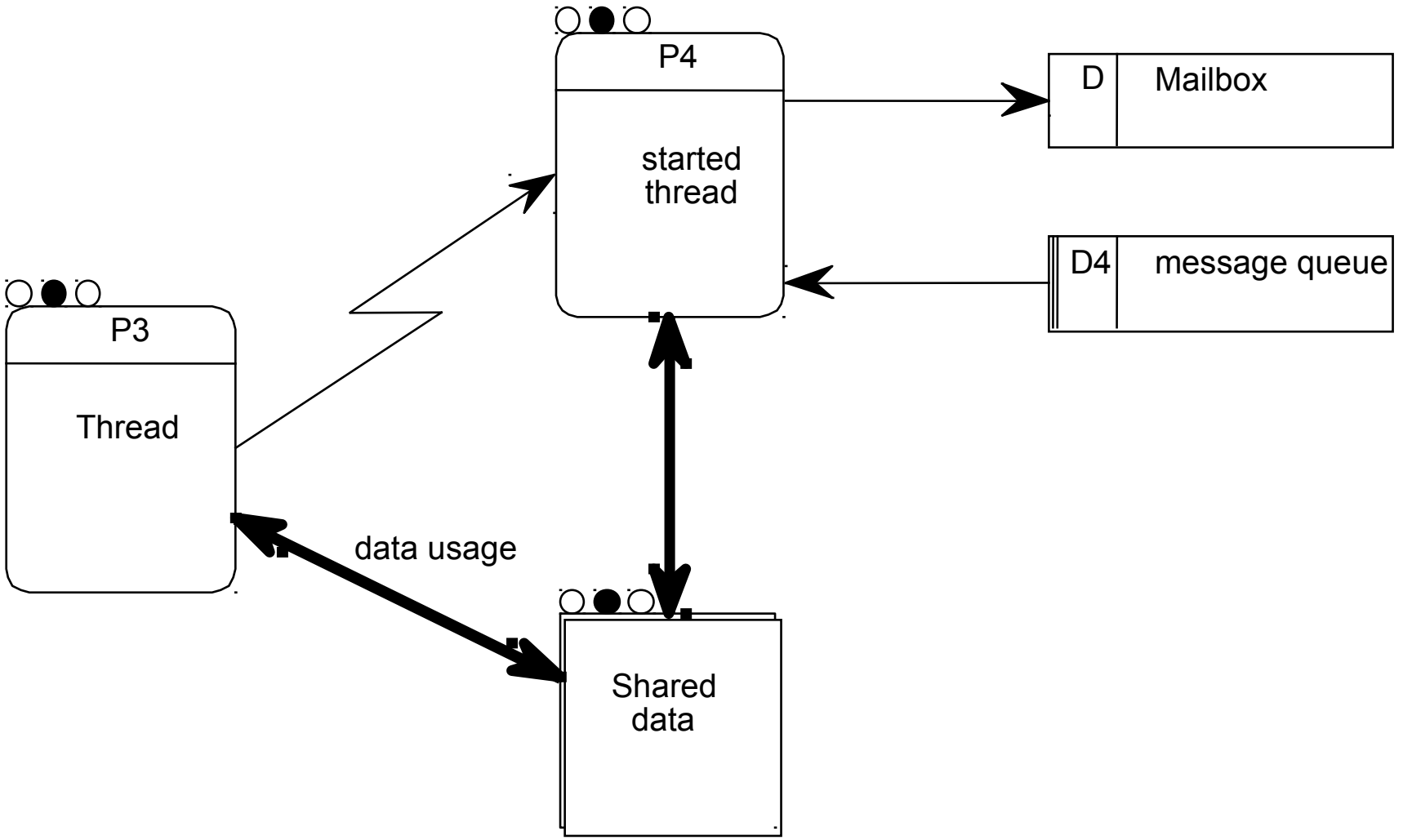


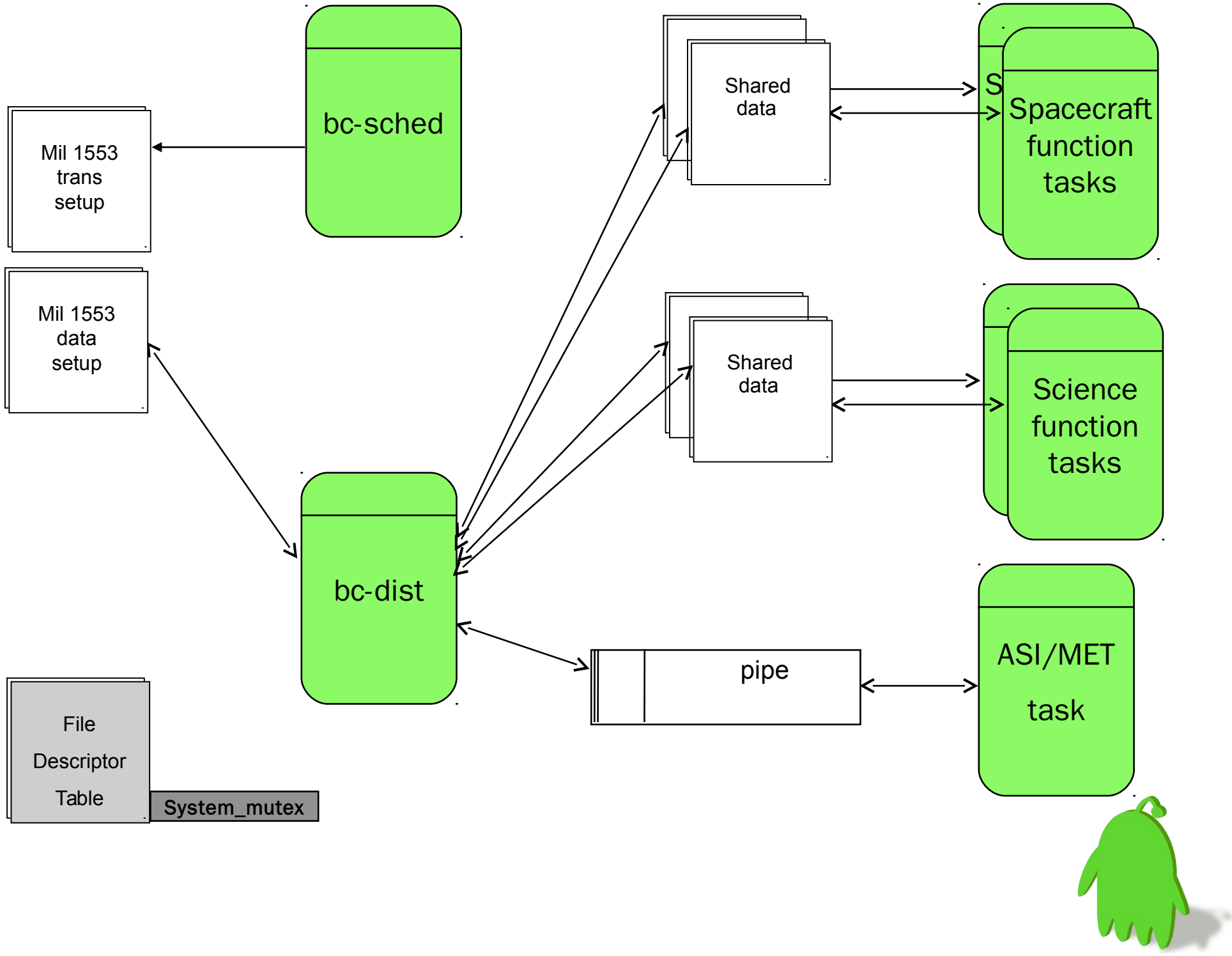
1553 communication

- **Powered 1553 devices deliver data.**
- Tasks in the system that access the information collected over the 1553 do so via a **double buffered shared memory mechanism** into which the bc_dist task places the latest data.
- The exception to this is the ASI/MET task which is delivered its information via an interprocess communication mechanism (IPC). The IPC mechanism uses the VxWorks *pipe()* facility.


VMEbus
Cruise stage
controls
thrusters,
valves,
a sun sensor,
a star scanner
Lander
interface to
accelerometers,
a radar altimeter,
an instrument for
meteorological science
known as the ASI/MET
Packed
buffer
Mil1553
Camera
Radio
CPU
RS6000
D-Buffer
D-Buffer
D-Buffer
IPC PIPE
File
Descriptor
List
MEM

Dedicated systems' tasking model




bc-sched
Mil 1553
trans
setup
Mil 1553
data
setup
bc-dist
Shared
data
Shared
data
Spacecraft
function
tasks
Science
function
tasks
pipe
ASI/MET
task
File
Descriptor
Table
System_mutex

IPC Mechanism

- Tasks wait on one or more IPC "queues" for messages to arrive using the VxWorks *select()* mechanism to wait for message arrival.
- Multiple queues are used when both high and lower priority messages are required.
- Most of the IPC traffic in the system is not for the delivery of real-time data. The exception to this is the use of the IPC mechanism with the ASI/MET task.
- The cause of the reset on Mars was in the use and configuration of the IPC mechanism.

VxWorks select()

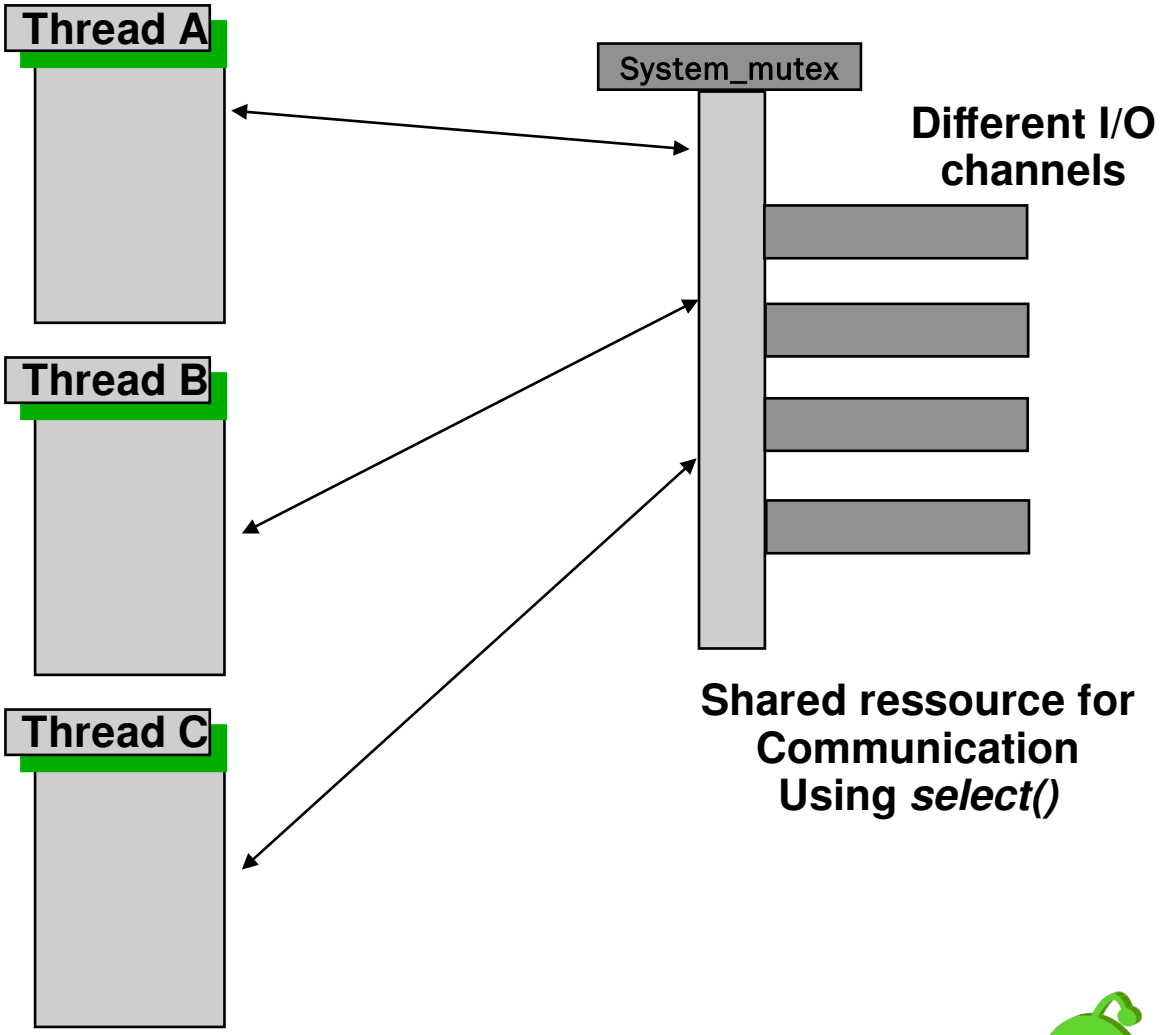
- Pending on multiple file descriptiors:
  this routine permits a task to pend until one of a set of file descriptors becomes available
- Wait for multiple I/O devices (task level and driver level)
- file descriptors
  pReadFds, pWriteFds
- Bits set in *pReadFds* will cause *select()* to pend until data becomes available on
  any of the corresponding file descriptors.
- Bits set in *pWriteFds* will cause *select()* to pend until
  any of the corresponding file descriptors
  becomes available.

Marsrobot

Middle priority long lasting
Comm thread
bc_dist

Low priority thread

Lowest priority sporadic
meteo thread
ASI/MET



The problem, again

- Priority inversion occurs when a thread of low priority blocks the execution of threads of higher priority.
- Two flavours:
  - bounded priority inversion

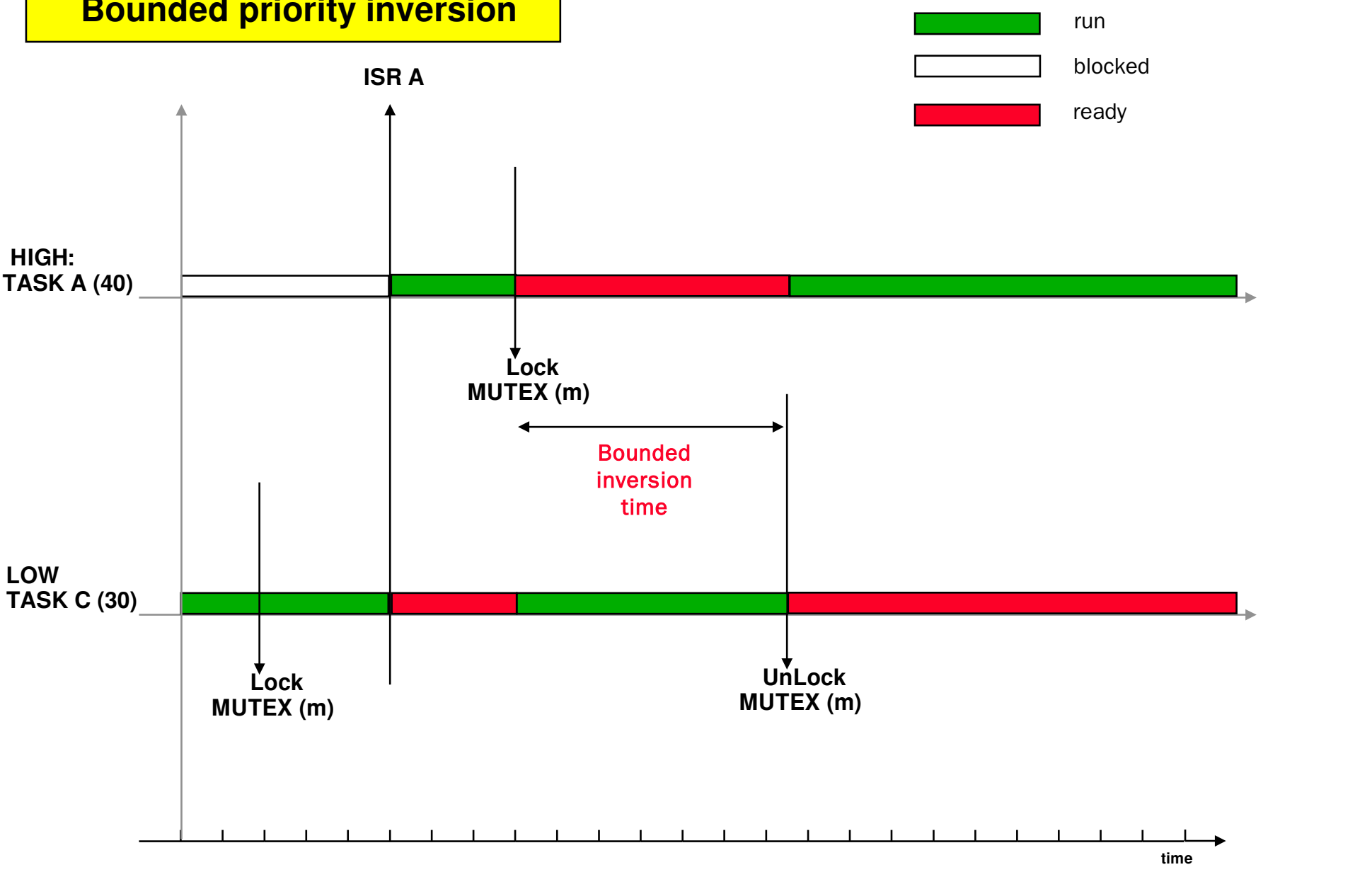
    (common & relatively harmless)
  - unbounded priority inversion

    (insidious & potentially disastrous)

Bounded Priority Inversion

- Suppose a high priority thread becomes blocked waiting for an event to happen. A low priority thread then starts to run and in doing so obtains (i.e locks) a mutex for a shared resource. While the mutex is locked by the low priority thread, the event occurs waking up the high priority thread.
- Inversion takes place when the high priority thread tries to lock the mutex held by the low priority thread. In effect the high priority thread must wait for the low priority thread to finish.
- It is called bounded inversion since the inversion is limited by the duration of the critical section.

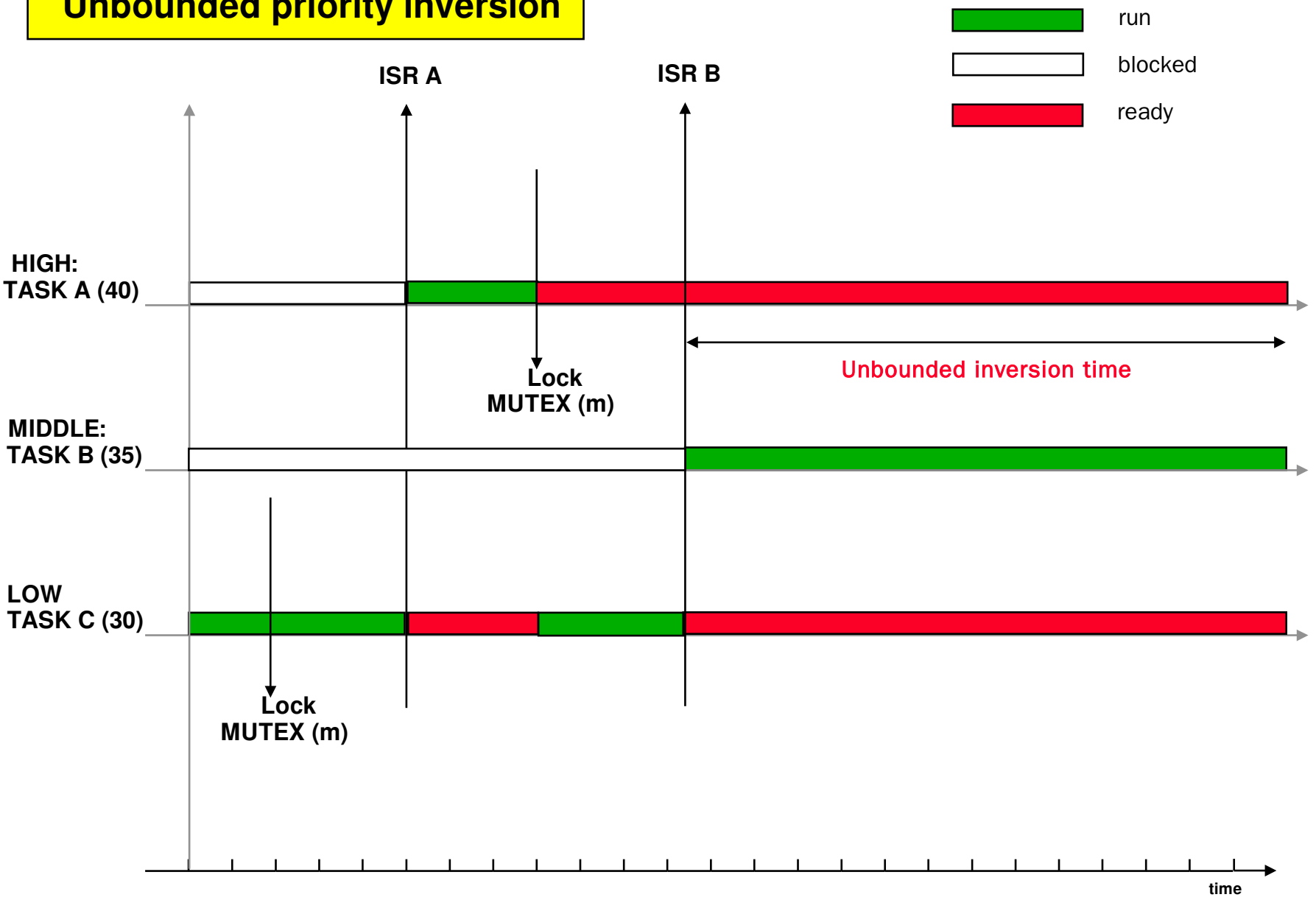
Bounded priority inversion



Unbounded Priority Inversion

- Here the high level thread can be blocked indefinitely by a medium priority thread.
- The medium level thread running prevents the low priority thread from releasing the lock.
- All that is required for this to happen is that while the low level thread has locked the mutex, the medium level thread becomes unblocked, preempting the low level thread.
- The medium level thread then runs indefinitely.

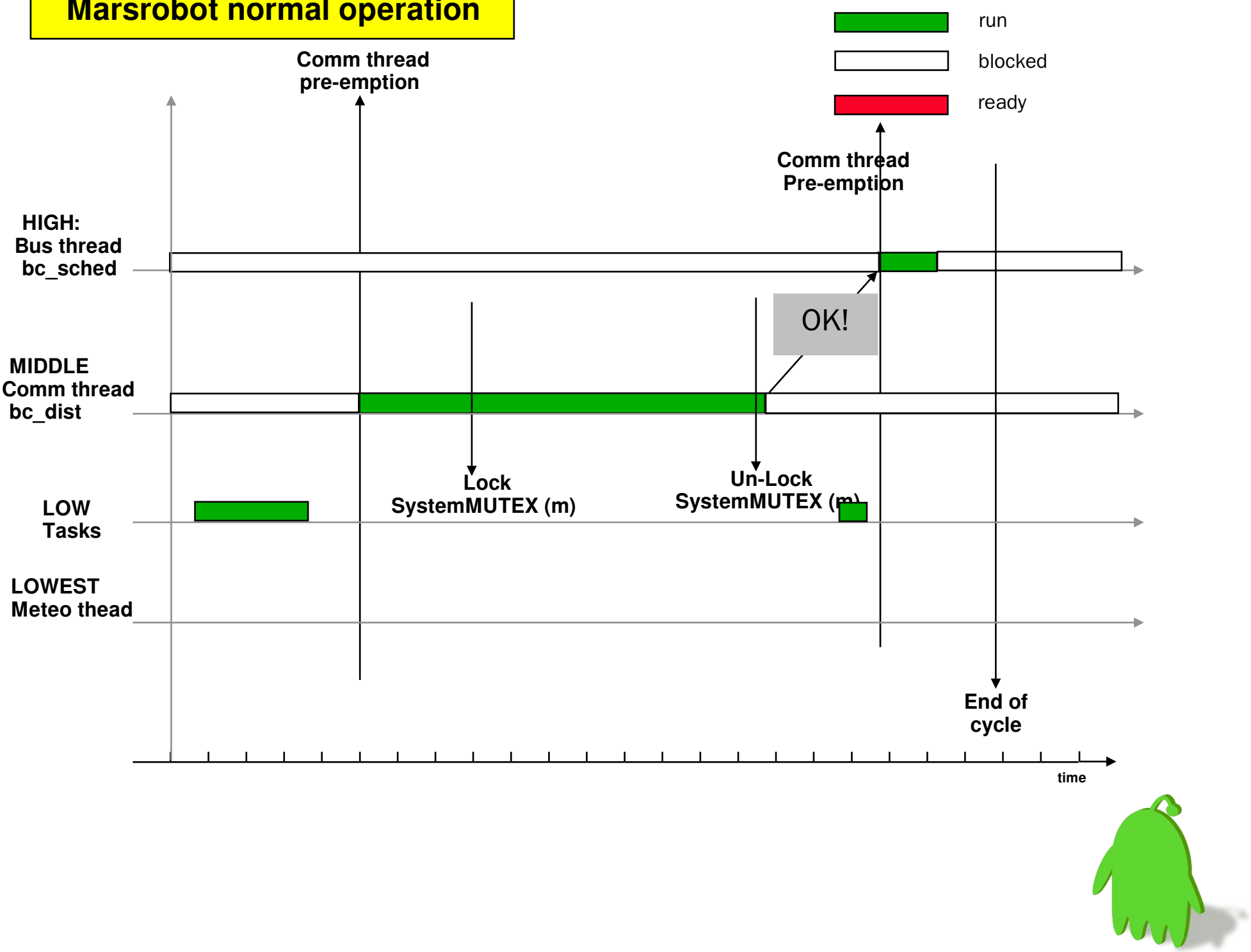
Unbounded priority inversion



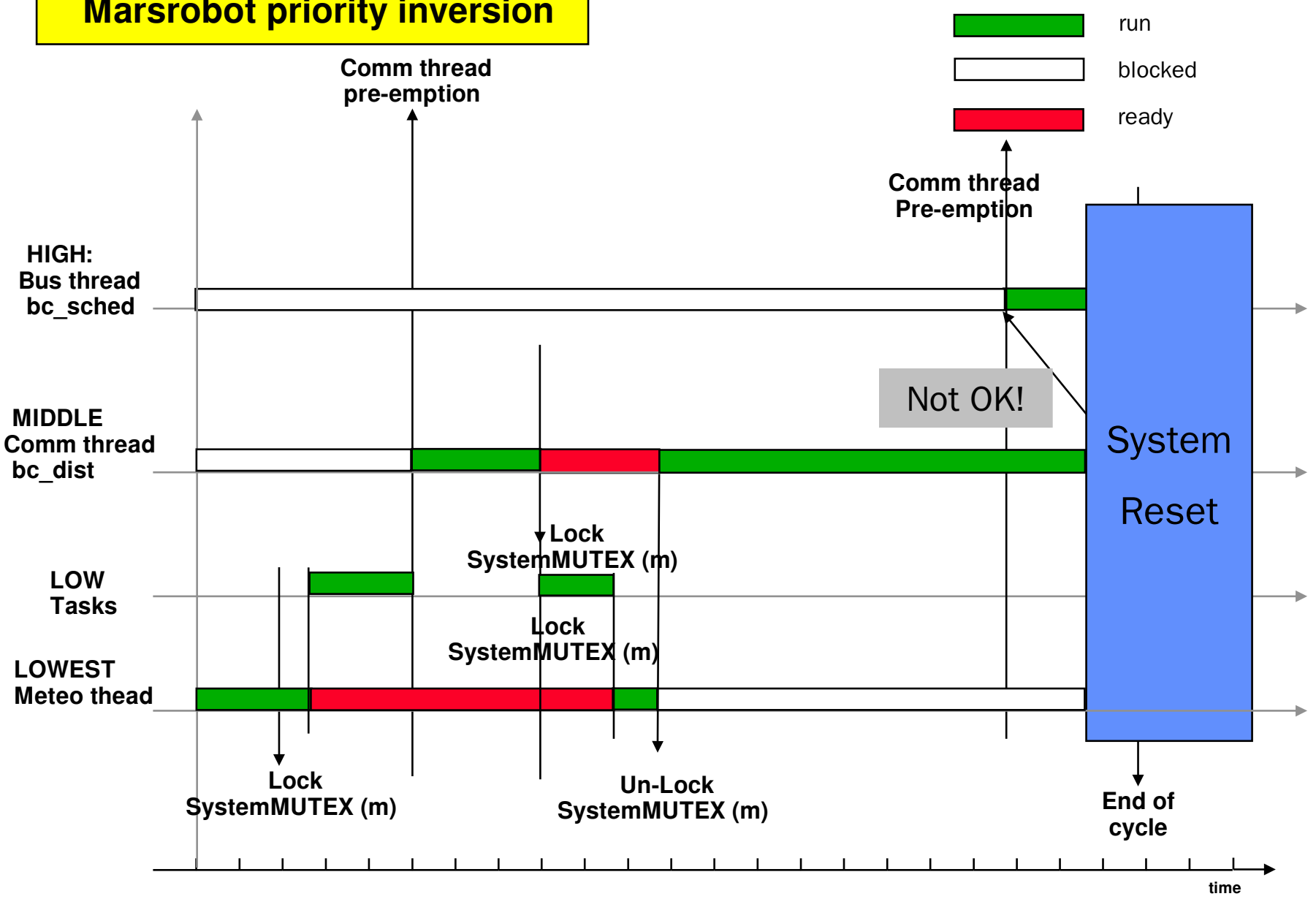
Pathfinder Failure

- The failure was identified by the spacecraft as a failure of the bc_dist task to complete its execution before the bc_sched task started.
- The reaction to this by the spacecraft was to **reset** the computer.
- This **reset** reinitializes all of the hardware and software. It also terminates the execution of the current ground commanded activities. No science or engineering data is lost that has already been collected (the data in RAM is recovered so long as power is not lost).
- The remainder of the activities for that day were not accomplished until the next day.

Marsrobot normal operation



Marsrobot priority inversion



Priority Inversion

- The higher priority bc_dist task was blocked by the much lower priority ASI/MET task that was holding a shared resource.
- The ASI/MET task had acquired this resource and then been preempted by several of the medium priority tasks.
- When the bc_sched task was activated, to setup the transactions for the next 1553 bus cycle, it detected that the bc_dist task had not completed its execution.
- The resource that caused this problem was a mutex (here called **system_mutex**) used within the *select()* mechanism to control access to the list of file descriptors that the *select()* mechanism was to wait on.

Priority Inversion

- The *select()* mechanism creates a **system_mutex** to protect the "wait list" of file descriptors for those devices which support *select().*
- The VxWorks **pipe** mechanism is such a device and the IPC mechanism used is based on using pipes.
- The ASI/MET task had called *select()*, which had called *pipeIoctl(),* which had called *selNodeAdd(),* which was in the process of giving the **system_mutex.**
- The ASI/ MET task was preempted and *semGive()* was not completed.
- Several medium priority tasks ran until the bc_dist task was activated.
- The bc_dist task attempted to send the newest ASI/MET data via the IPC mechanism which called *pipeWrite().*
- *pipeWrite()* blocked, taking the **system_mutex**. More of the medium priority tasks ran, still not allowing the ASI/MET task to run, until the bc_sched task was awakened.
- At that point, the bc_sched task determined that the bc_dist task had not completed its cycle (a hard deadline in the system) and declared the error that initiated the **reset**.

Mars PathFinder Solutions

The New York Times
nytimes.com

July 18, 1997

Problem Stalling Mars Study Is Reported Solved

Engineers reported today that they had solve
Mars Pathfinder computer and will radio up

Until then they will continue to get around the
far has averted resets like three earlier ones t

The Pathfinder rover, meanwhile, has moved
the deputy project manager, Brian Muirhead

The source of the software problem was disc
engineer usually called "the gremlin" because

"We set the gremlin loose in the testbed," Mr
already identified as the probable cause. He

As suspected, the Pathfinder computer, strug
carry out low-priority tasks in the allotted tim
computer.

The low-priority task that kept tripping it up
an electronics board and then into the compu
reprogramming, Mr. Muirhead said.

Changing a single line of computer code "will

As suspected, the Pathfinder computer, struggling with several activities at once, reset itself each time it could not carry out low-priority tasks in the allotted time. A reset is a safety feature similar to hitting a reset button on a home computer.

The low-priority task that kept tripping it up was the transfer of temperature and wind measurements from sensors to an electronics board and then into the computer. The solution is to raise the task's priority through some reprogramming, Mr. Muirhead said.

What is the Fix?

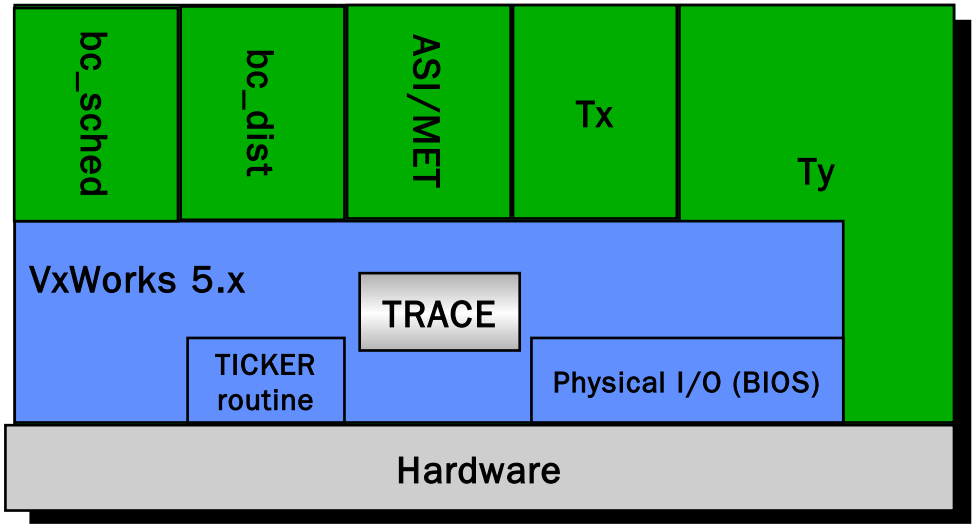
- Problem with priority inversion:
  - A high priority thread is stuck waiting for a low priority thread to finish its work
  - In this case, the (medium priority) thread was holding up the low-priority thread
- General solution: *Priority inheritance*
  - If waiting for a low priority thread, allow that thread to *inherit* the higher priority
  - High priority thread “donates” its priority to the low priority thread
- Why can it fix the problem?
  - Medium priority comm task cannot preempt weather task
  - Weather task inherits high priority while it is being waited on

What was the problem fixed?

- JPL had a replica of the Pathfinder system on the ground
  - Special tracing mode maintrains logs of all interesting system events
    - e.g., context switches, mutex lock/unlock, interrupts
  - After much testing were able to replicate the problem in the lab
- VxWorks mutex objects have an optional priority inheritance flag
  - Engineers were able to upload a patch to set this flag on the info bus mutex
  - After the fix, no more system resets occurred
- Lessons:
  - Automatically reset system to “known good” state if things run amuck
    - Far better than hanging or crashing
  - Ability to trace execution of complex multithreaded code is useful
  - Think through all possible thread interactions carefully!!

Debug the problem

- On replica on earth
- Total Tracing on
  - Context switches
  - Uses of synchronisation objects
  - Interrupts
- Took time to reproduce the error
- Trace analyses ==>priority inversion problem

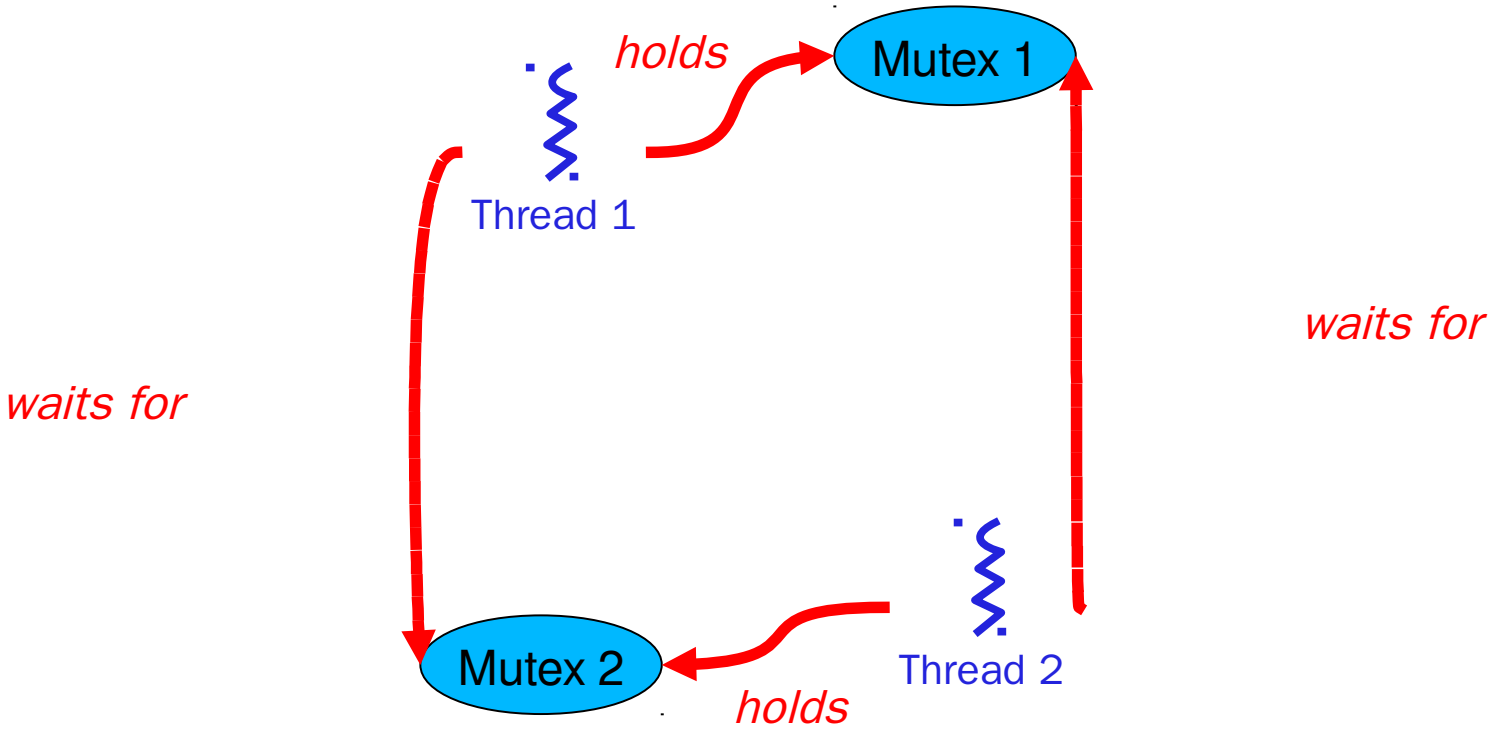


Bug Detection

- The software that flies on Mars Pathfinder has several debug features within it that are used in the lab but are not used on the flight spacecraft (not used because some of them produce more information than we can send back to Earth).
- These features remain in the software by design because JPL strongly believes in the "***test what you fly and fly what you test***" philosophy.

Deadlock

- With priority inversion, eventually the system makes progress
  - e.g., Comm thread eventually finishes and rest of system proceeds
  - Pathfinder watchdog timer reset the system too quickly!
- A far more serious situation is *deadlock*
  - Two (or more) threads waiting for each other
  - None of the deadlocked threads ever make progress



Deadlock Definition

- Two kinds of resources:
  - Preemptible: Can take away from a thread
    - e.g., the CPU
  - Non-preemptible: Can't take away from a thread
    - e.g., mutex, lock, virtual memory region, etc.
- Why isn't it safe to forcibly take a lock away from a thread
- Starvation
  - A thread never makes progress because other threads are using a resource it needs
- Deadlock
  - A circular waiting for resources
    - Thread A waits for Thread B
    - Thread B waits for Thread A
- Starvation ≠ Deadlock

Conditions for Deadlock

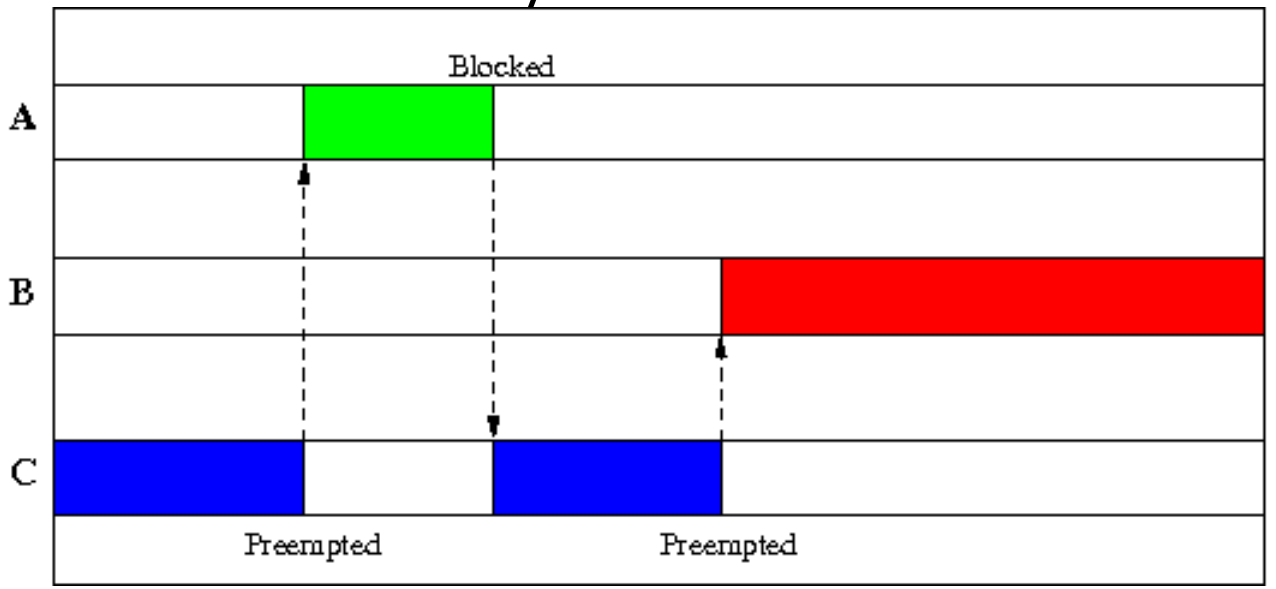
- Limited access to a resource
  - Means some threads will have to wait to access a shared resource
- No preemption
  - Means resource cannot be forcibly taken away from a thread
- Multiple independent requests
  - Means a thread can wait for some resources while holding others
- Circular dependency graph
  - Just as in previous example
- Without *all* of these conditions, can't have deadlock!
  - Suggests several ways to get rid of deadlock

Get rid of deadlock

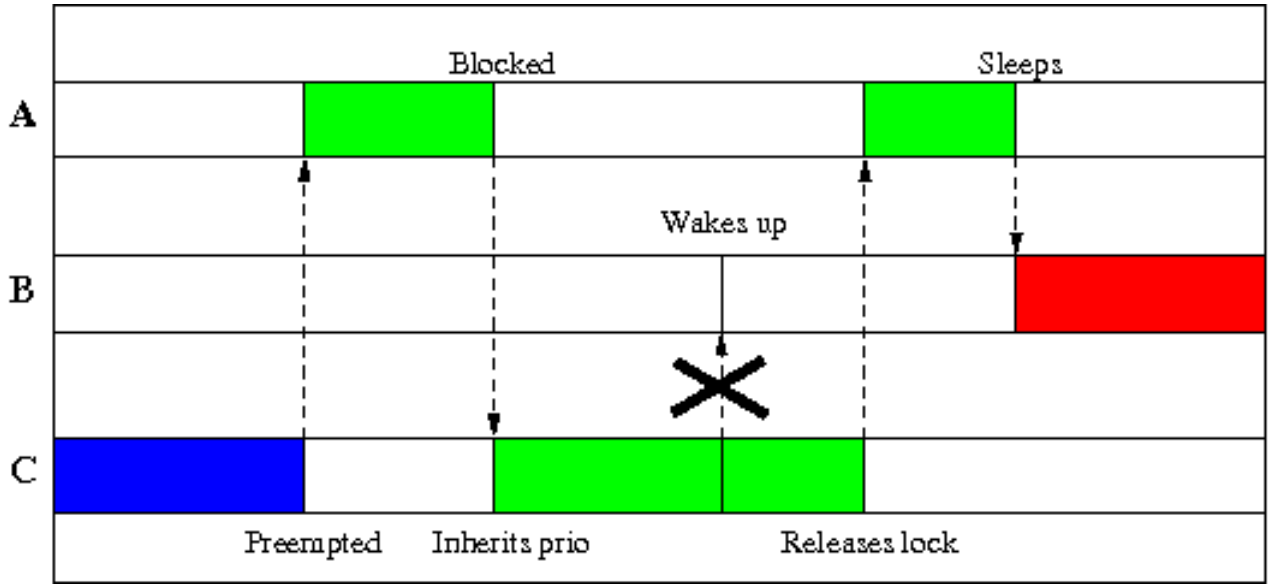
- Unlimited access to a resource?
  - Requires that all resources allow arbitrary number of concurrent accesses
    - Probably not too feasible!
- Always allow preemption?
  - Is it safe to let multiple threads into a critical section?
- No multiple independent requests?
  - This might work!
  - Require that threads grab all resources they need before using any of them!
    - Not allowed to wait while holding some resources!
- No circular chains of requests?
  - This might work too!
  - Require threads to grab resources in some predefined order!

Resource Access Protocols

Priority Inversion

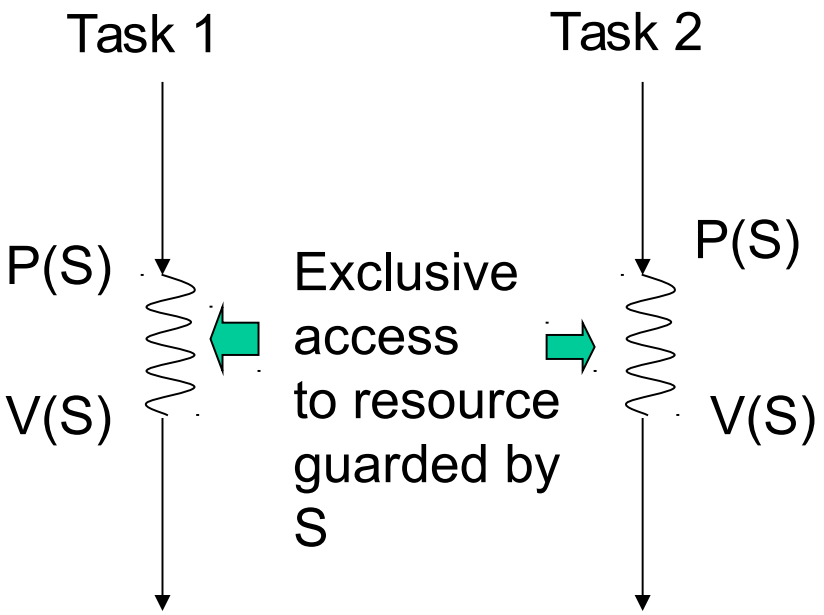


Priority Inheritance



Resource Access Protocols

- Critical sections: sections of code at which exclusive access to some resource must be guaranteed.
- Can be guaranteed with semaphores S.



P(S) checks semaphore to see if resource is available
and if yes, sets S to „used“.
Uninterruptable operations!
If no, calling task has to wait.

V(S): sets S to „unused“ and starts sleeping task (if any).

Reference

- Mars pathfinder failure, Martin TIMMERMAN
- Operating Systems, Paulo Marques, Departamento de Eng. Informática, Universidade de Coimbra
- Synchronization Problems and Deadlock, Matt Welsh
- Mike Jones article "What really happened on Mars?" http://research.microsoft.com/~mbj/Mars_Pathfinder/

http://0xlab.org