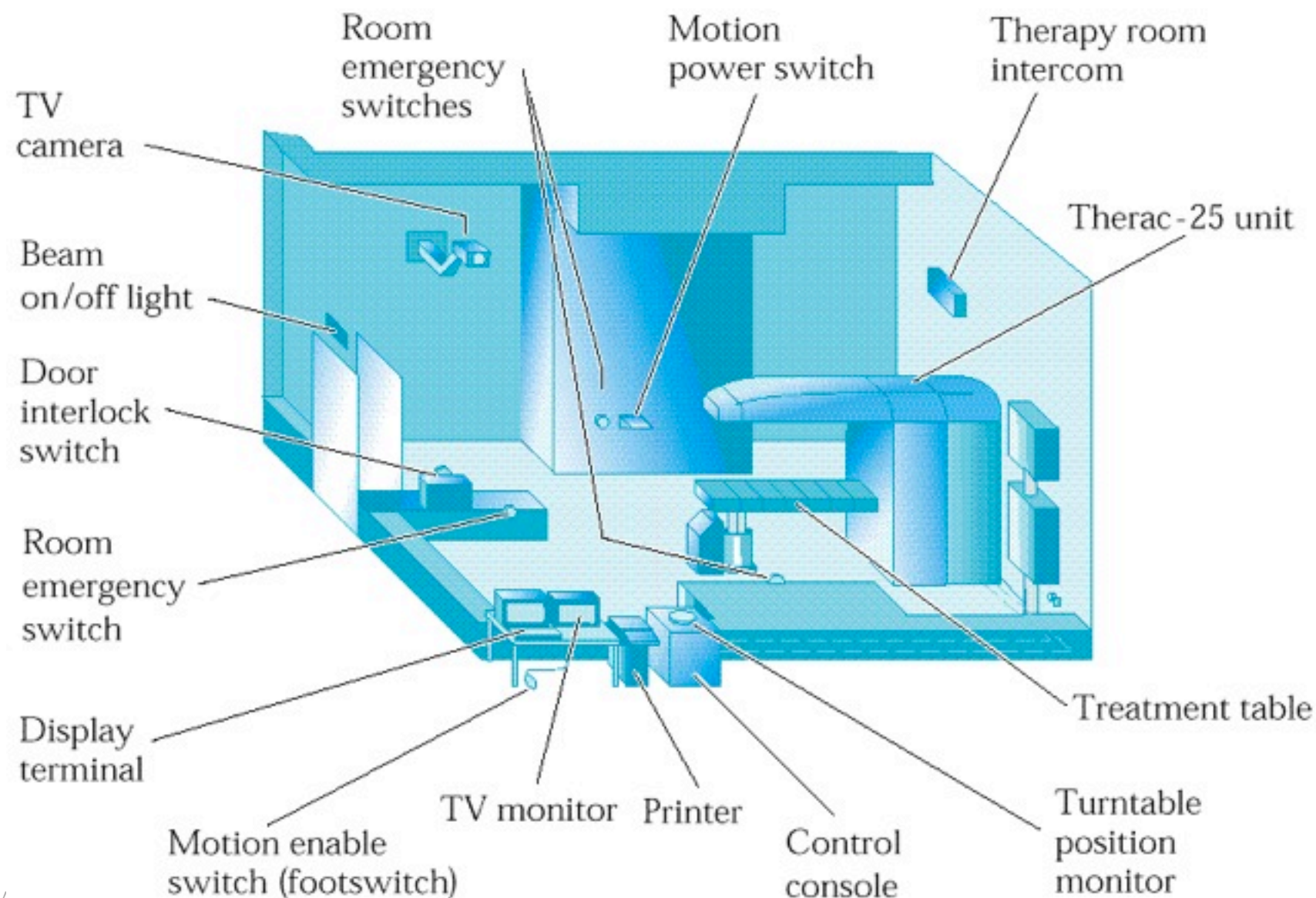


Therac-25

- Computer-controlled radiation therapy machine
 - In operation between 1983 and 1987, 11 installations

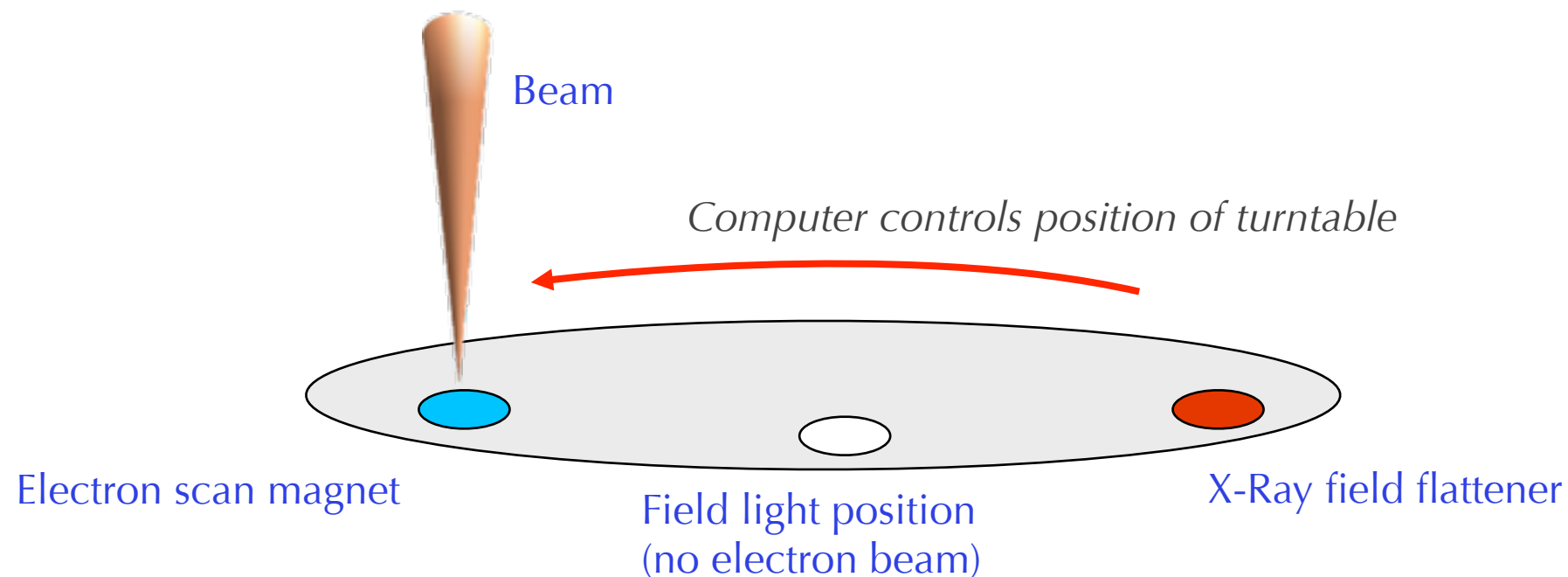


Accidents

- Capable of delivering electron and photon (X-Ray) treatments
- “Evolved” from earlier models, Therac-20 and Therac-6
- On several occasions between June '85 and Jan '87
 - Massive overdoses to six people, some lethal
 - Several overdoses delivered energy of 15,000 – 20,000 rads
 - Typical therapeutic doses in the 200 rad range
- Various lawsuits, all settled out of court
 - No formal investigation
- Initially, manufacturer claimed that overdoses were impossible
- Many issues with the Therac-25
 - Software design methodology
 - Software/hardware engineering
 - User interface
 - Concurrency

Therac-25 operation

- A turntable aperture that moves certain elements into the path of the beam



- Field light mode used to position beam on patient
 - No electron beam expected, instead, a light simulates the beam position
- Electron scan magnet and X-Ray field flattener used to attenuate and spread electron and X-Ray beams

Therac-25 operation

- Unlike previous models, completely computer controlled
 - No hardware interlocks to prevent misconfigurations or overdoses!
 - Software from old models re-used.
- All software written in PDP-11 assembly language
- Operator uses a VT-100 terminal to control machine
- Cryptic error messages delivered to operator console
 - e.g., “Malfunction 23”
 - No documentation of these error codes, no indication of which errors are potentially life-threatening

Therac-25 internals

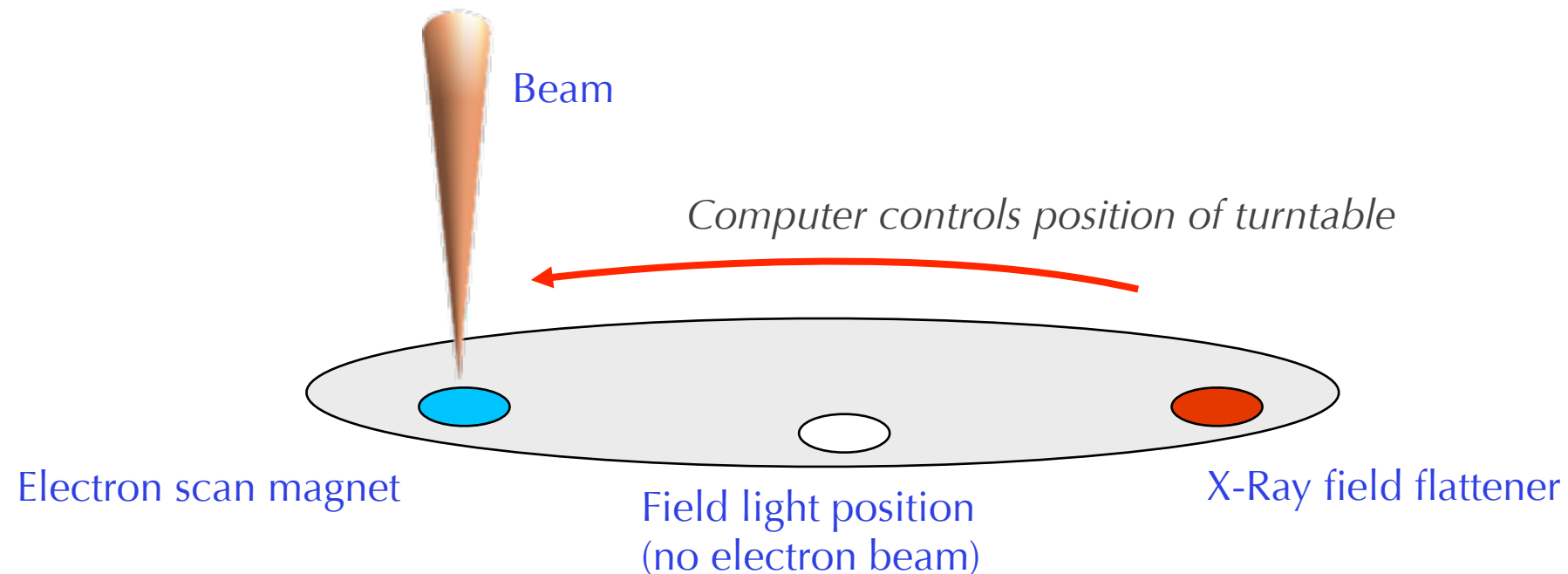
- 4 components: scheduler, critical and non-critical tasks, interrupt services, and stored data
 - Preemptive scheduler schedules critical and non-critical tasks
- Critical tasks include:
 - Treatment task
 - Directs and monitors patient setup and treatment
 - Interacts with keyboard and terminal interrupt services
 - Servo task
 - Controls gun emission, dose rate, turntable, and other machine motions
- Concurrent access to shared memory with no synchronization
 - Test and set are not atomic
 - Race conditions resulting from this play an important part in the accidents

Race Condition #1

- It was discovered that overdose could be caused by operator editing the dosage on the console too quickly
 - Operator enters dosage on screen, moves to bottom, moves back up to edit dosage, and back to bottom
 - Second edit displayed on screen, but ignored by machine
 - Bug not triggered in testing/training, since needs to be done quickly
- What happened?
- Treatment task
 - Periodically checks **entryDone** flag (which is set when cursor moved to bottom of screen)
 - If flag is set, calls subroutine to configure the magnets (takes about 8 seconds)
- Configure magnet task
 - Called periodically to check if magnets are ready
 - Checks if edits have been made to dosage; If so, exits back to calling subroutine to restart the process
 - Critical bug: Only checks if edits made on the first call!
- Also, **entryDone** flag indicates cursor was at bottom of screen, not that it is still there. Race condition between user editing dosage and reading dosage.

Race Condition #2

- Second bug – totally different causes from the first



- Software interlocks intended to stop beam from being turned on unless turntable in correct position
- Problem: Turntable could be in field light position while X-Ray beam on

Race Condition #2

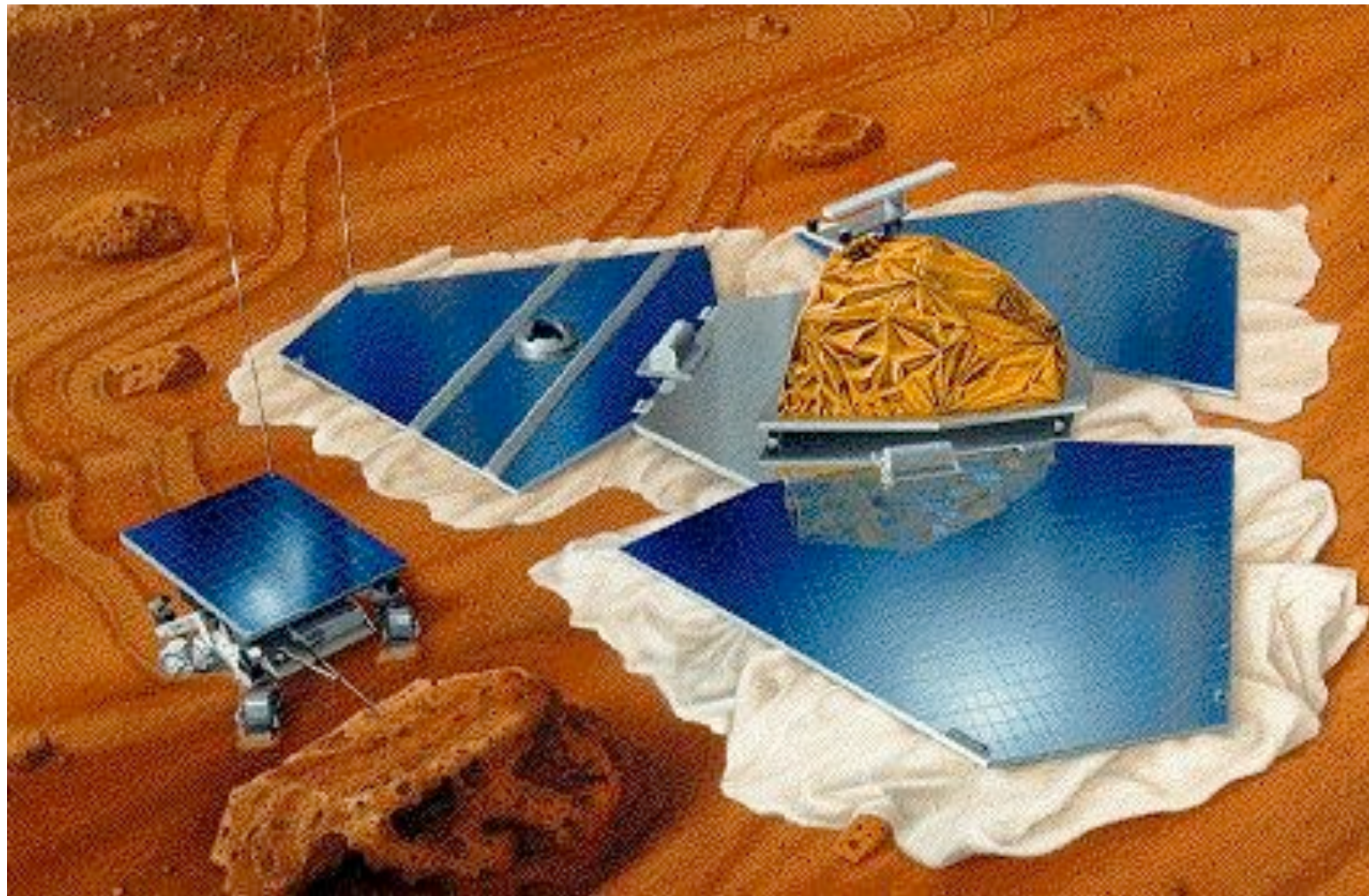
- Dosage entered on console; Operator then presses SET button to set turntable to correct position
- Software interlock:
 - Shared variable `Class3` indicates whether machine configuration consistent with dosage: zero == OK, non-zero == inconsistent
 - Shared variable `Fmal` indicates whether a malfunction exists
 - Set up test task runs after dosage entered, and periodically checks if machine configured consistently with dosage
 - Increments variable "Class3" on each iteration
 - Will be run many times
 - If position correct and no malfunctions (`Fmal == 0`), sets "`Class3 := 0`"
- When SET button is pressed, Housekeeping task runs
 - If `Class3 != 0` check whether turntable in place (set a bit of `Fmal`)
 - Skip check if `Class3 == 0`.
- Can you spot the bug?

Race Condition #2

- The bug: `Class3` variable is 8 bits wide
 - After 256 iterations of “set up test” routine, overflows and becomes zero!
 - So if operator presses `SET` button during short interval that `Class3` overflows, does not check turntable position
- Fix: Set `Class3` to some nonzero value, rather than incrementing it
 - Why was this done? Probably because `inc` instruction was easy enough...

Mars Pathfinder

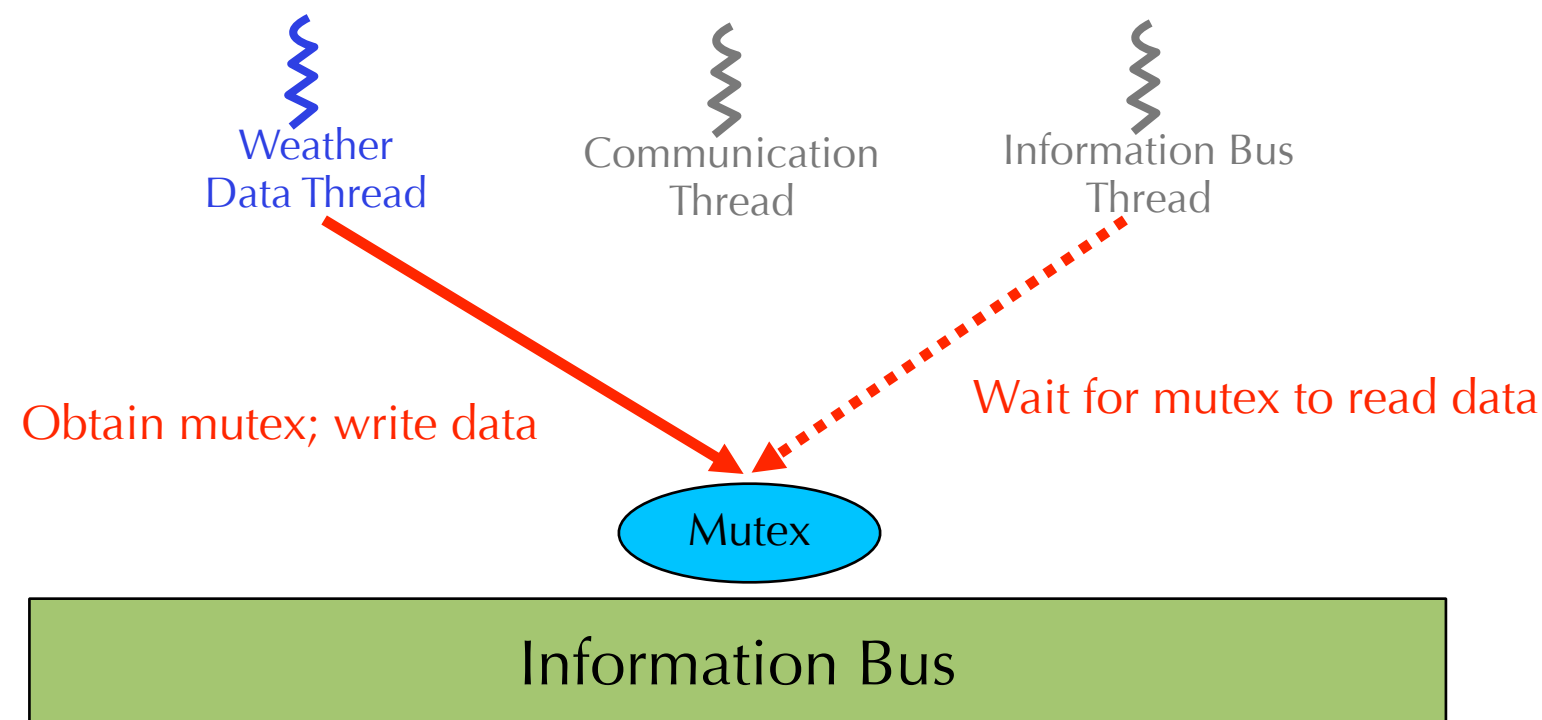
- July 4, 1997 landing on Martian surface, followed by expeditions by Sojourner rover



- Series of software glitches started a few days after landing
- Eventually debugged and patched remotely from Earth!

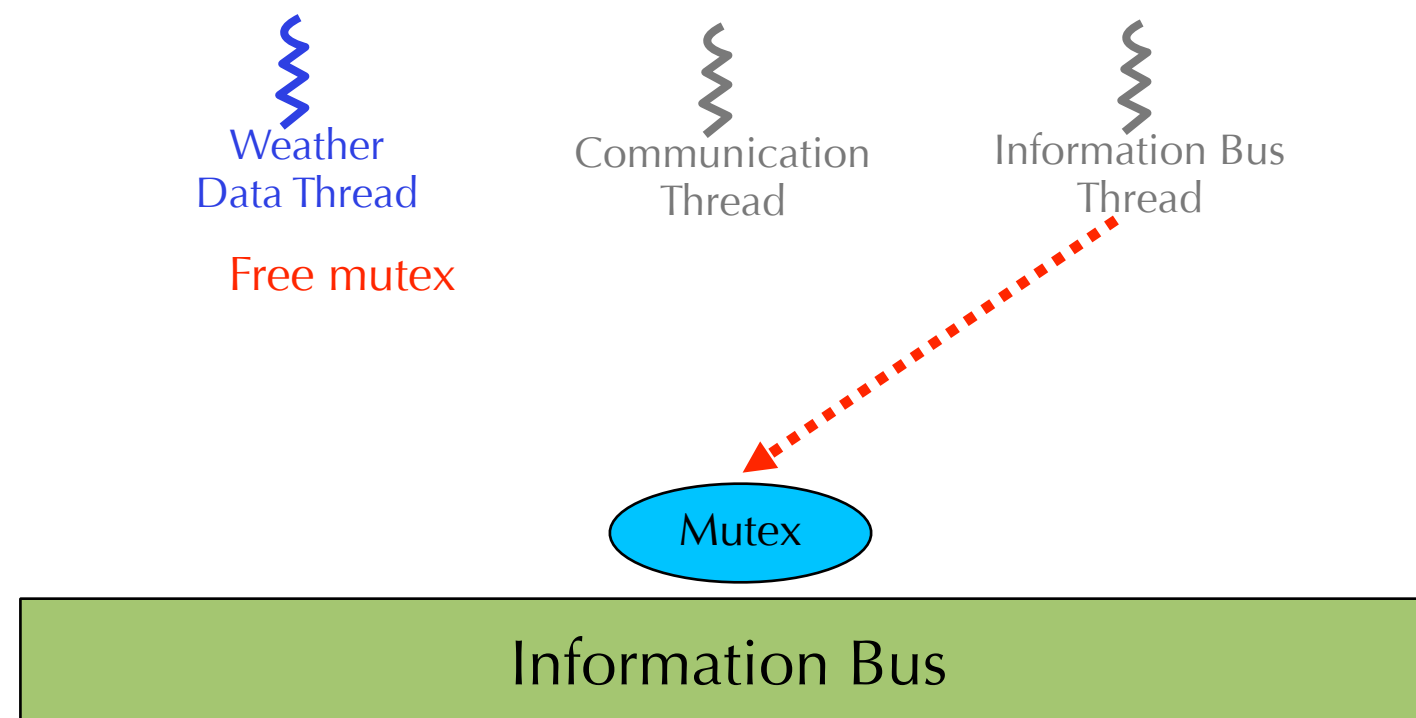
VxWorks Operating System

- Developed by Wind River Systems – premier real time OS
- 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



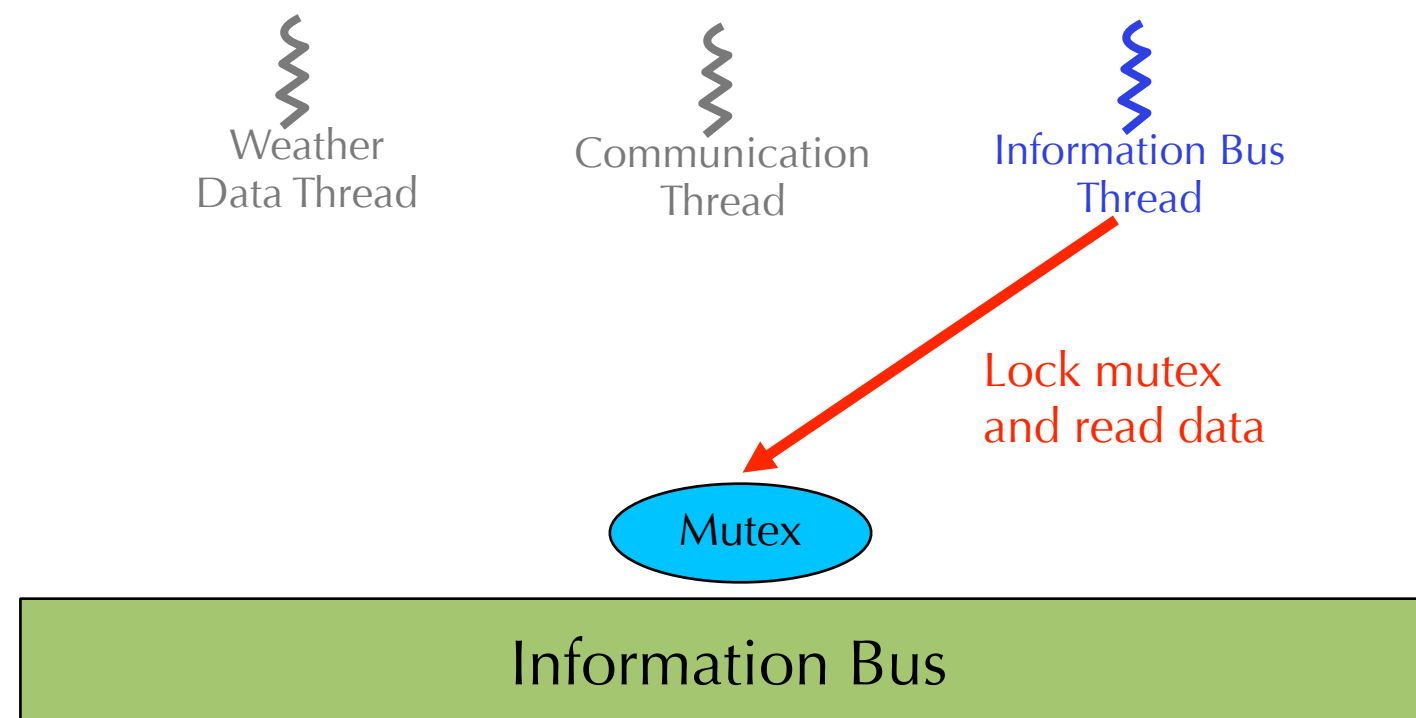
VxWorks Operating System

- Developed by Wind River Systems – premier real time OS
- 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



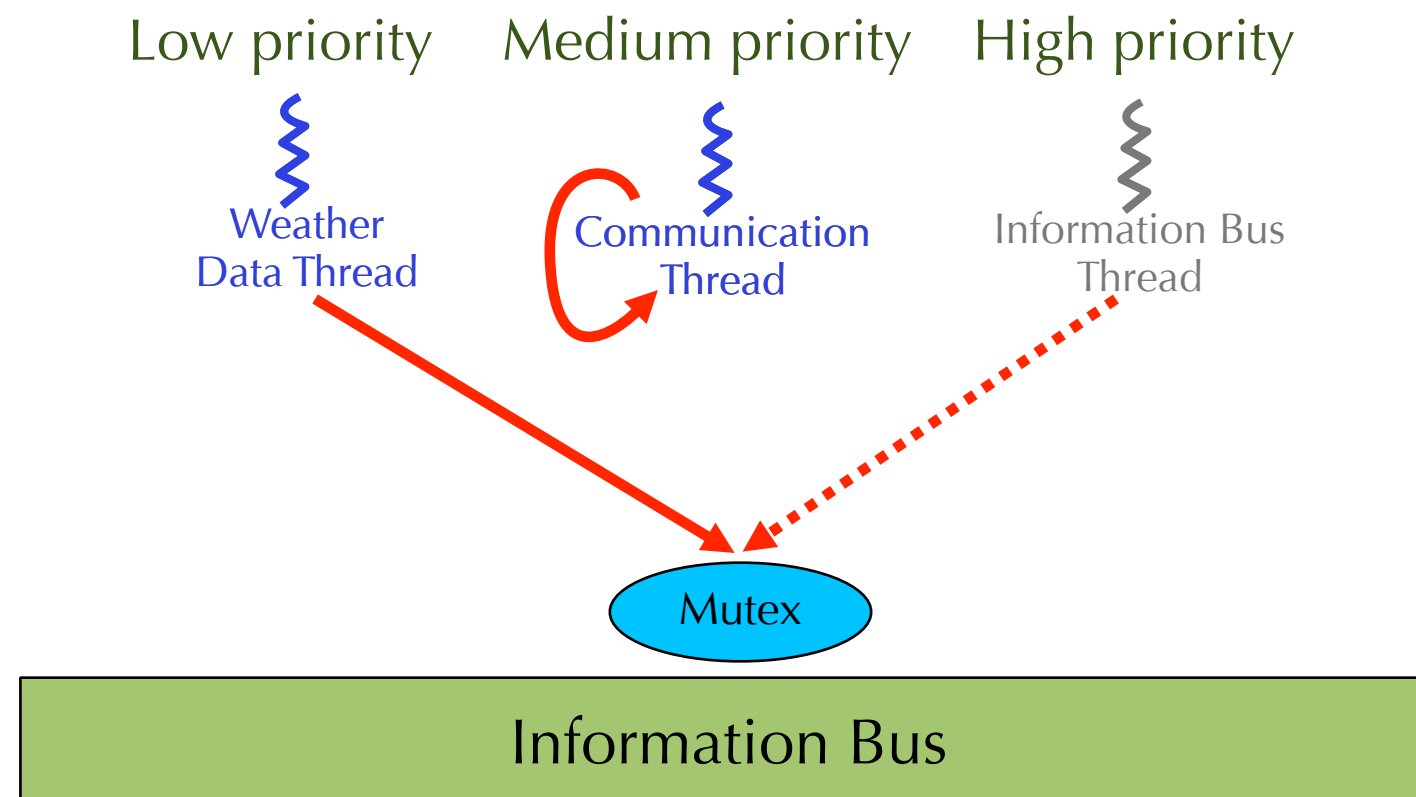
VxWorks Operating System

- Developed by Wind River Systems – premier real time OS
- 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

- What happens when threads have different priorities?
- Suppose the low priority thread has the mutex, and medium priority thread needs the CPU
 - Medium thread has higher priority than Low thread, so gets the CPU. Runs for a long time.
 - But High thread waiting for Low thread to finish! Medium thread running instead of High!
- This is called **priority inversion**



How to fix priority inversion?

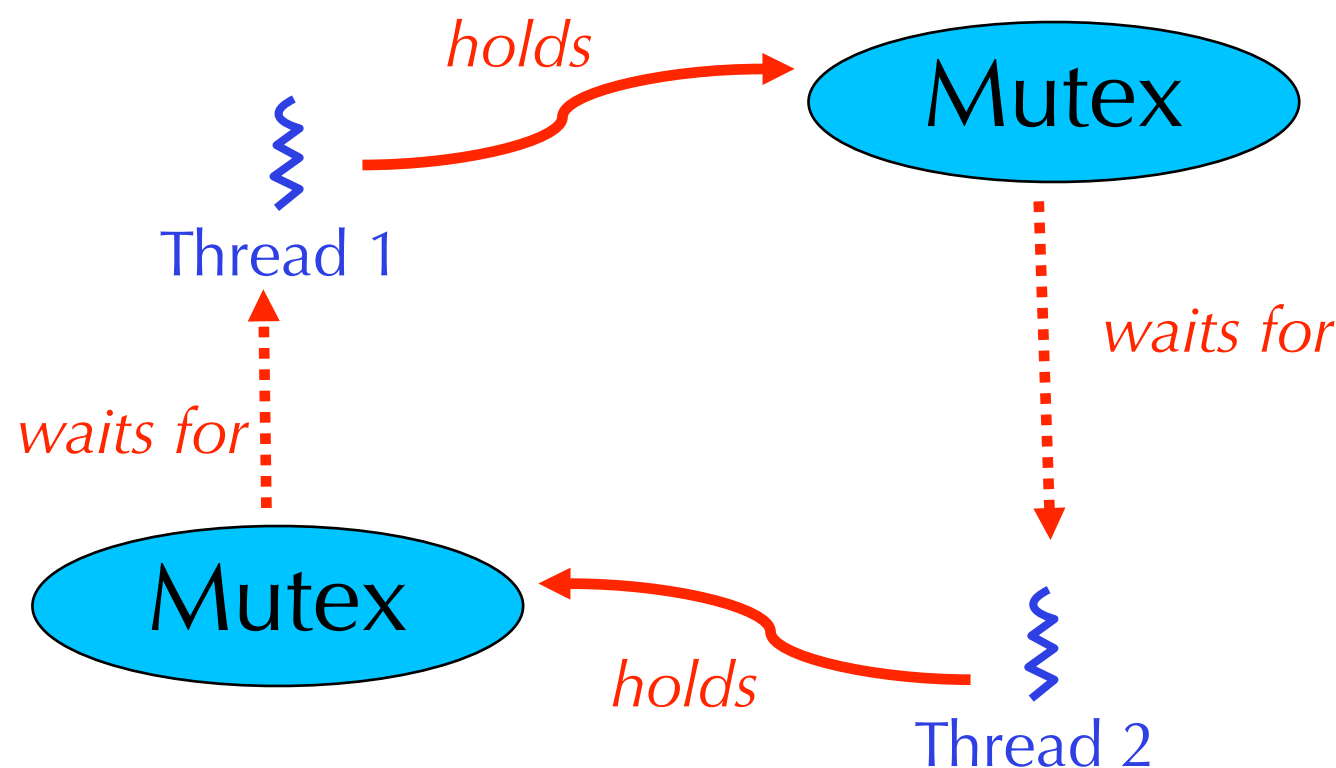
- Priority inversion:
 - A high priority thread is waiting for a low priority thread to finish (this is OK)
 - Medium priority thread comes along and preempts Low thread
 - Now Medium thread running instead of finishing Low thread
- General solution: Priority inheritance
 - If high priority thread is waiting for a low priority thread, temporarily give low thread high priority
 - High priority thread “donates” its priority to the low priority thread
- Why does this fix the problem?
 - Weather task inherits high priority while it is being waited on
 - Now medium priority communications task cannot preempt weather task

How was this problem fixed?

- JPL had a replica of the Pathfinder system on the ground
 - Special tracing mode maintains 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!!

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

- **Deadlock:** A circular waiting for resources
 - E.g., Thread A is waiting for a mutex Thread B has
Thread B is waiting for a mutex Thread C has
Thread C is waiting for a mutex Thread A has
- **Starvation:** a thread never makes progress because other threads are using resources it needs
- Starvation \neq Deadlock
 - Deadlock can be seen as a special case of starvation

Conditions for Deadlock

- Limited access to a resource
 - Means some threads will have to wait to access a shared resource. E.g., mutual exclusion
- No preemption
 - Means resource cannot be forcibly taken away from a thread
 - 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.)
- 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!
 - This suggests several ways to get rid of deadlock

Why is it unsafe to take a lock away from a thread?