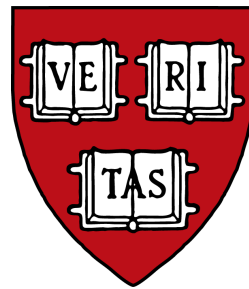


CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu



Lecture 7: Synchronization Problems and Deadlock
February 22, 2007

Today's Lecture

Classic synchronization problems

The THERAC-25 Accidents

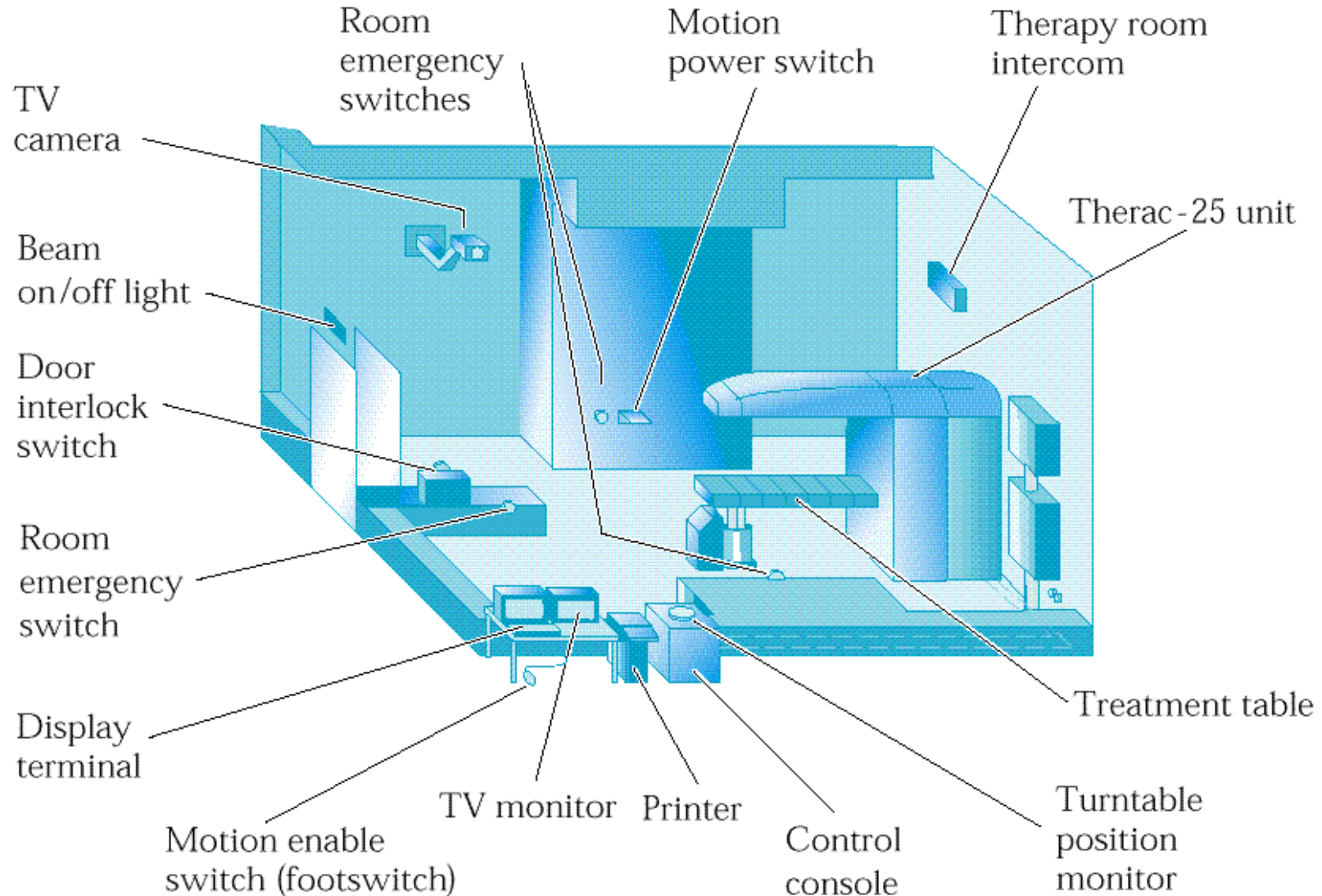
What happened to the Mars Pathfinder?

Deadlock detection and avoidance schemes

Therac-25

Computer-controlled radiation therapy machine

- In operation between 1983 and 1987, 11 installations



Therac-25

Capable of delivering electron and photon (X-Ray) treatments

Completely computer controlled

- No hardware interlocks to prevent misconfigurations or overdoses!

All software written in PDP-11 assembly language

Cryptic error messages delivered to operator console

- “Malfunction 23”
- No documentation of these error codes
- No indication of which errors are potentially life-threatening

Lots of smoke and mirrors by the manufacturer

- Claimed that 10^{-11} chance of delivering wrong dose to patient
- No justification for this claim in the safety analysis documents

Accidents

On several occasions between June '85 and Jan '87

- Massive overdoses to six people
- Some of these were lethal

Typical therapeutic doses in the 200 rad range

Several overdoses delivered energy of 15,000 – 20,000 rads

Various lawsuits, all settled out of court

Initially, manufacturer claimed that overdoses were *impossible*

Race Condition #1

After some trial and error, it was discovered that overdose could be caused by operator editing the dosage on the console *too quickly*

- Operator would enter dosage on console
- Move cursor to bottom of screen, then move cursor back up to edit dosage

“Treat” task

- Periodically checks “entry done” flag
 - *If flag is set, call subroutine to configure the magnets*
 - *Configuring magnets takes about 8 sec*

“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!**

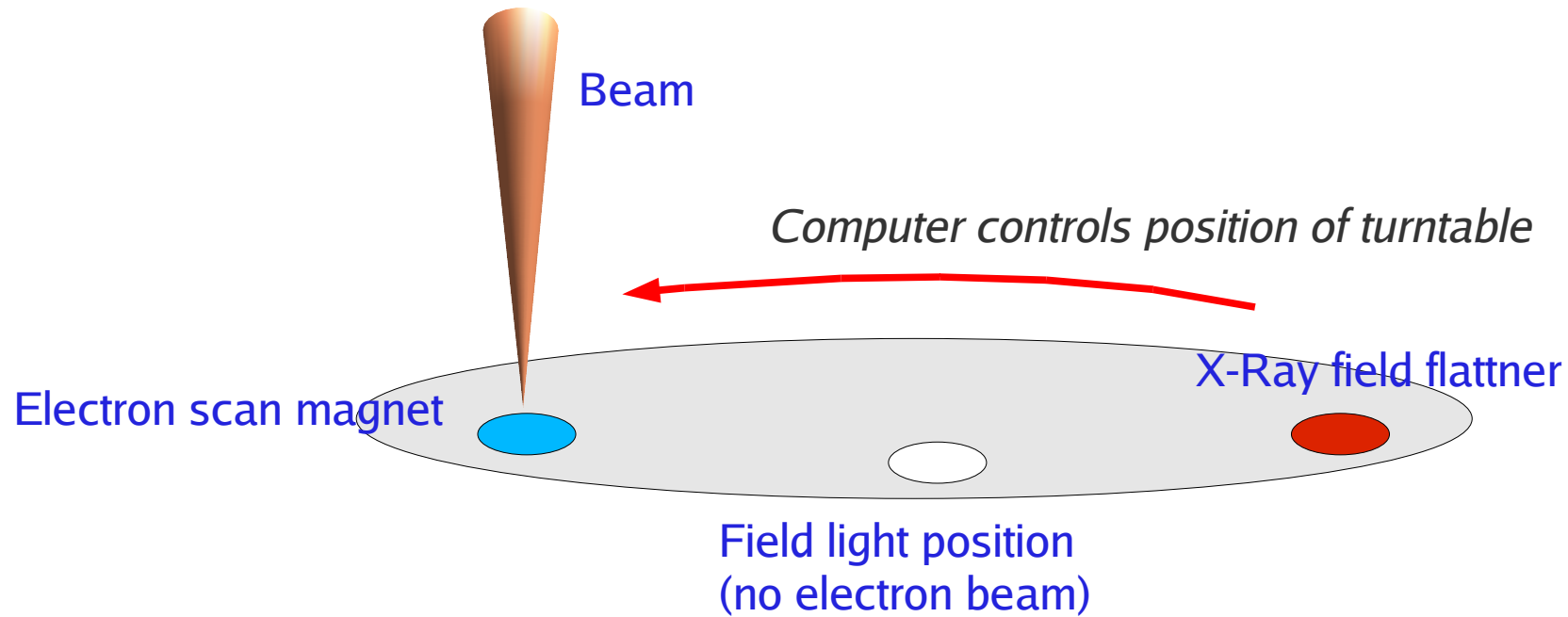
How this led to overdose:

- Operator enters dosage: Triggers magnet setting routine
- Operator edits dosage while the magnets are being configured
- Magnet routine does not notice edits have been made after first call

Race Condition #2

Second bug – totally different causes from the first

THERAC-25 has 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
- Problem: Unfiltered beam exposed to patients on several occasions!

Race Condition #2

- 1) Prescription entered on console
- 2) Operator must press “set” button to configure turntable
- 3) “Set up test” task runs periodically to check position of turntable
 - Increments a variable “Class3” on each iteration
 - If “Class3 == 0”, everything is ready and the dosage can begin
 - Otherwise, a series of interlock checks are performed to ensure turntable in the correct position
 - *These checks will set Class3 to 0 when they are complete*

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, interlocking checks will not be performed
- Operator must press “set” button during the short interval that Class3 overflows

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

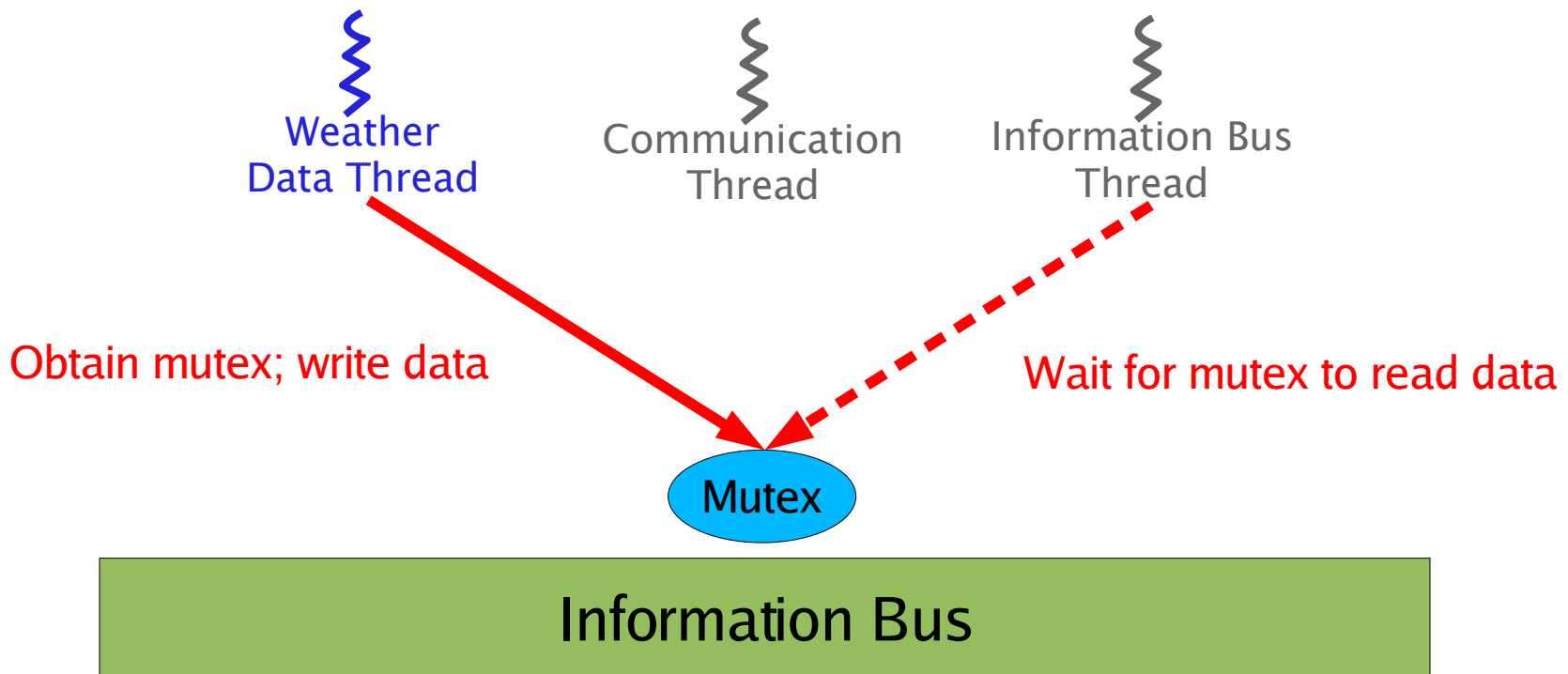
Developed by Wind River Systems – premier real time OS

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

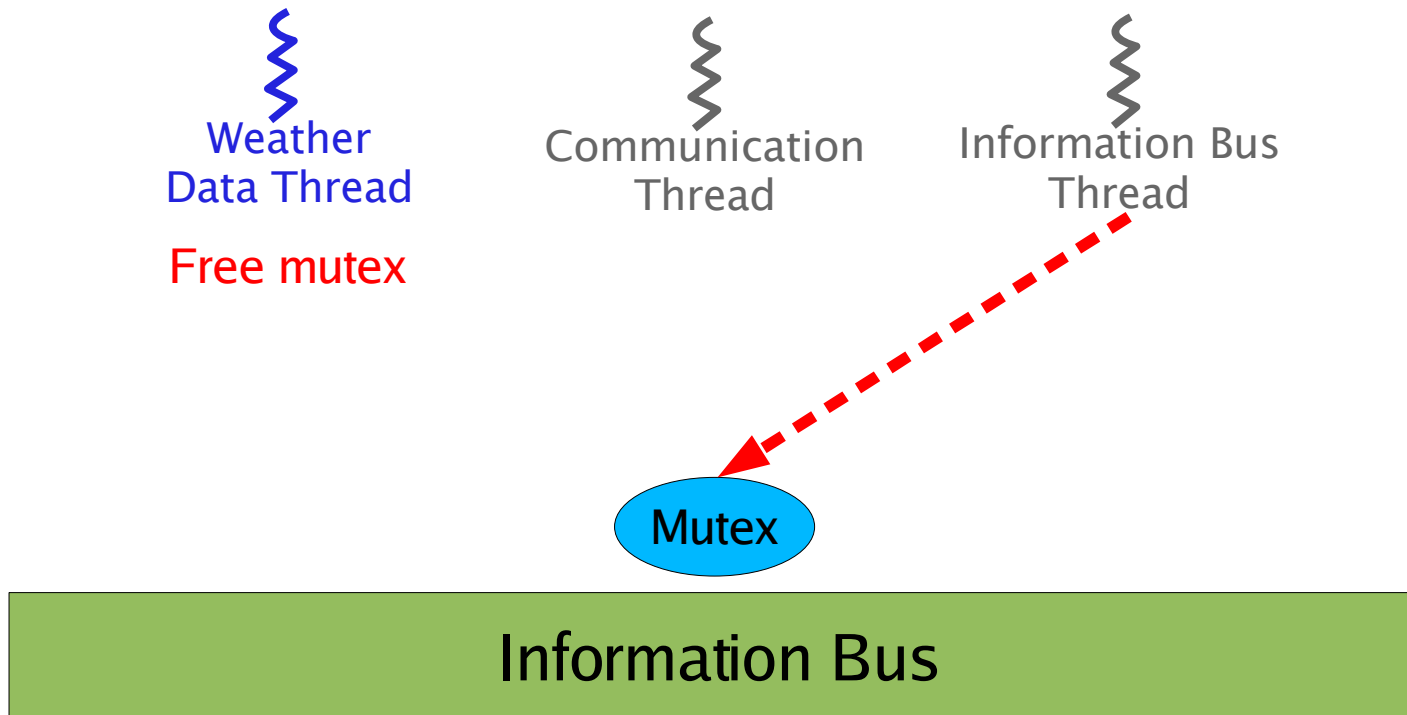
Developed by Wind River Systems – premier real time OS

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

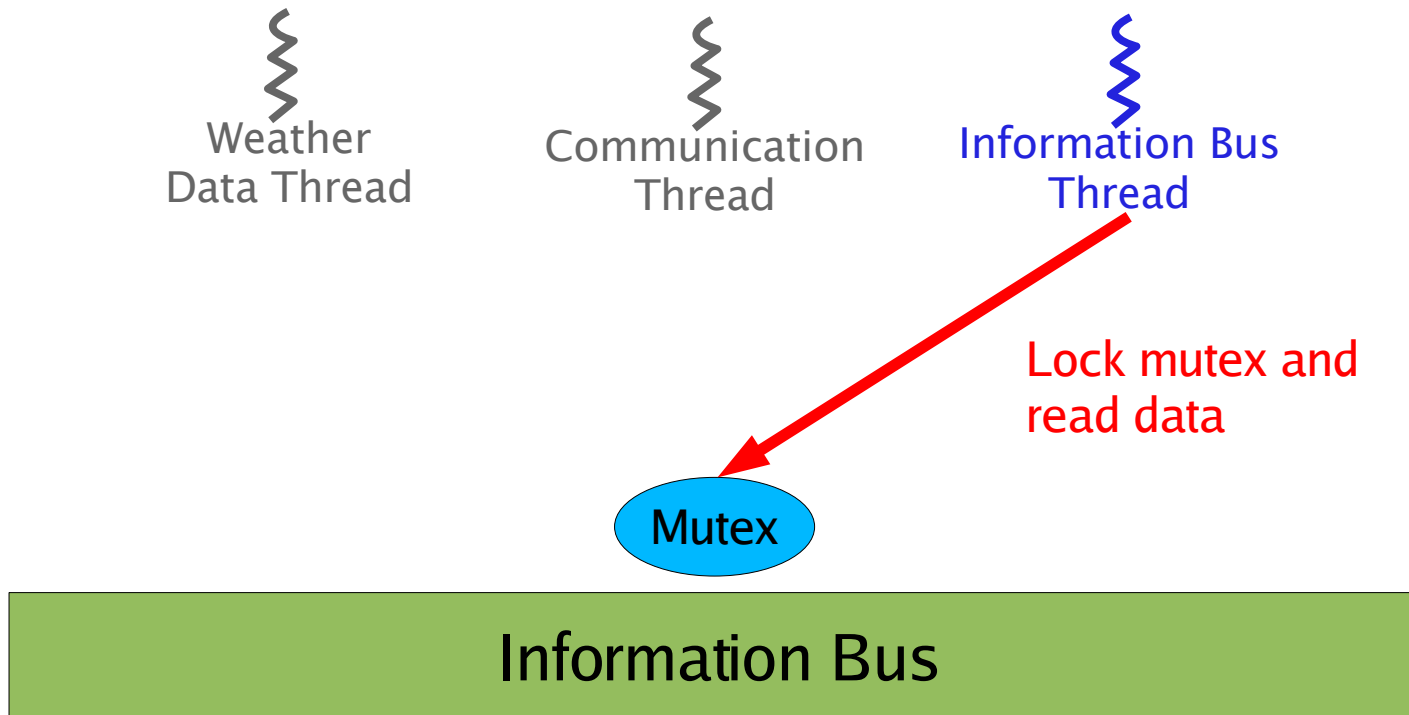
Developed by Wind River Systems – premier real time OS

Multiple tasks, each with an associated *priority*

- Higher priority tasks get to run before lower-priority tasks

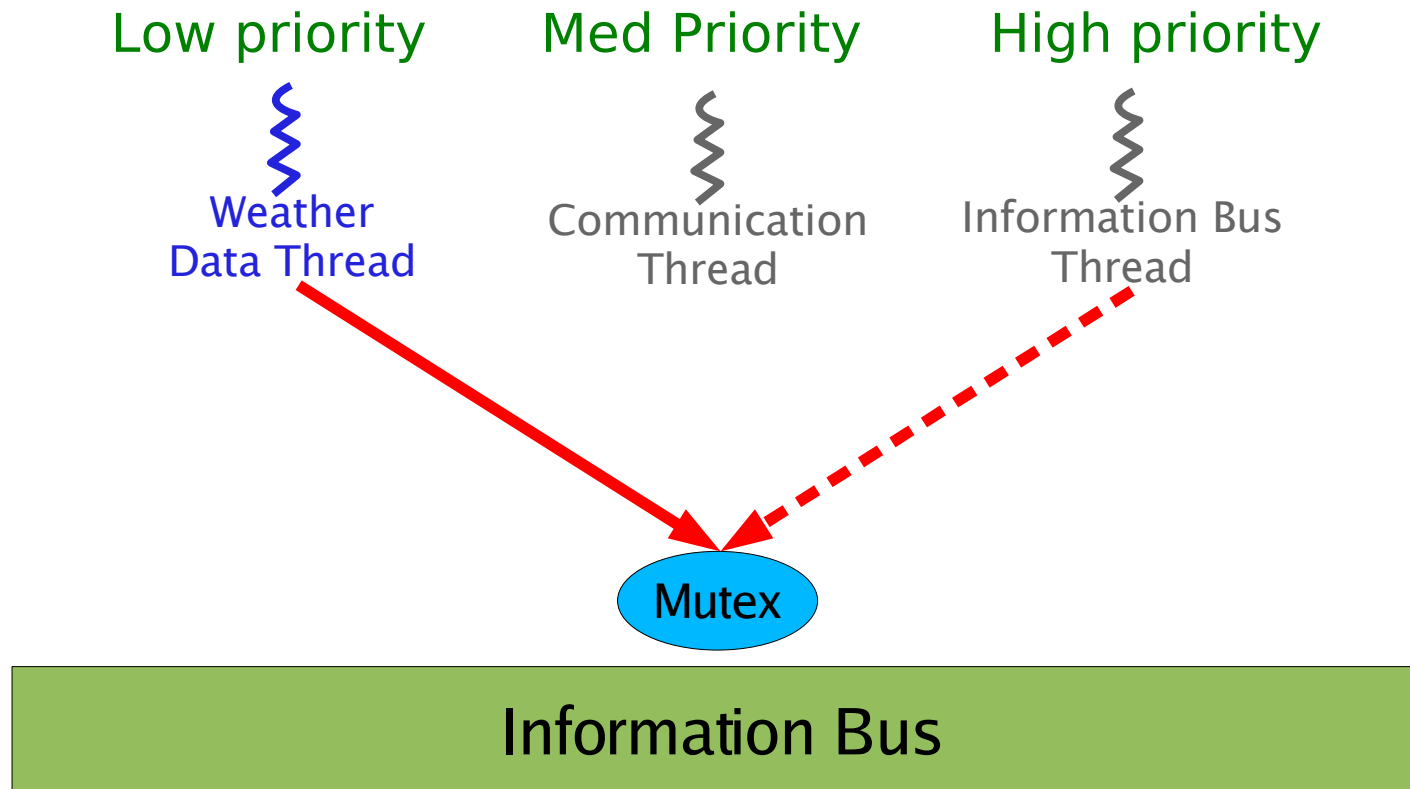
Information bus – shared memory area used by various tasks

- Thread must obtain mutex to write data to the info bus – a *monitor*



Priority Inversion

What happens when threads have different priorities?

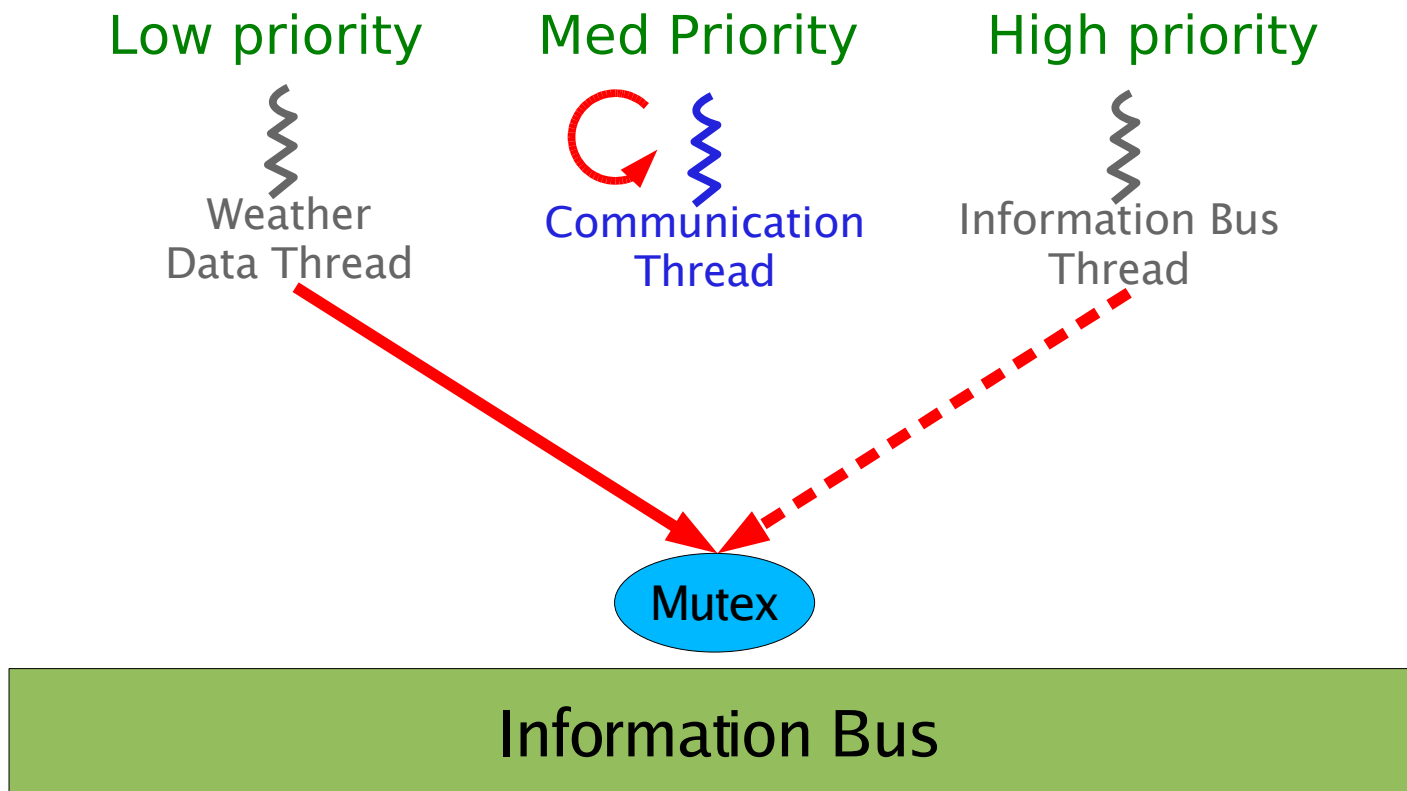


Priority Inversion

What happens when threads have different priorities?

Interrupt!

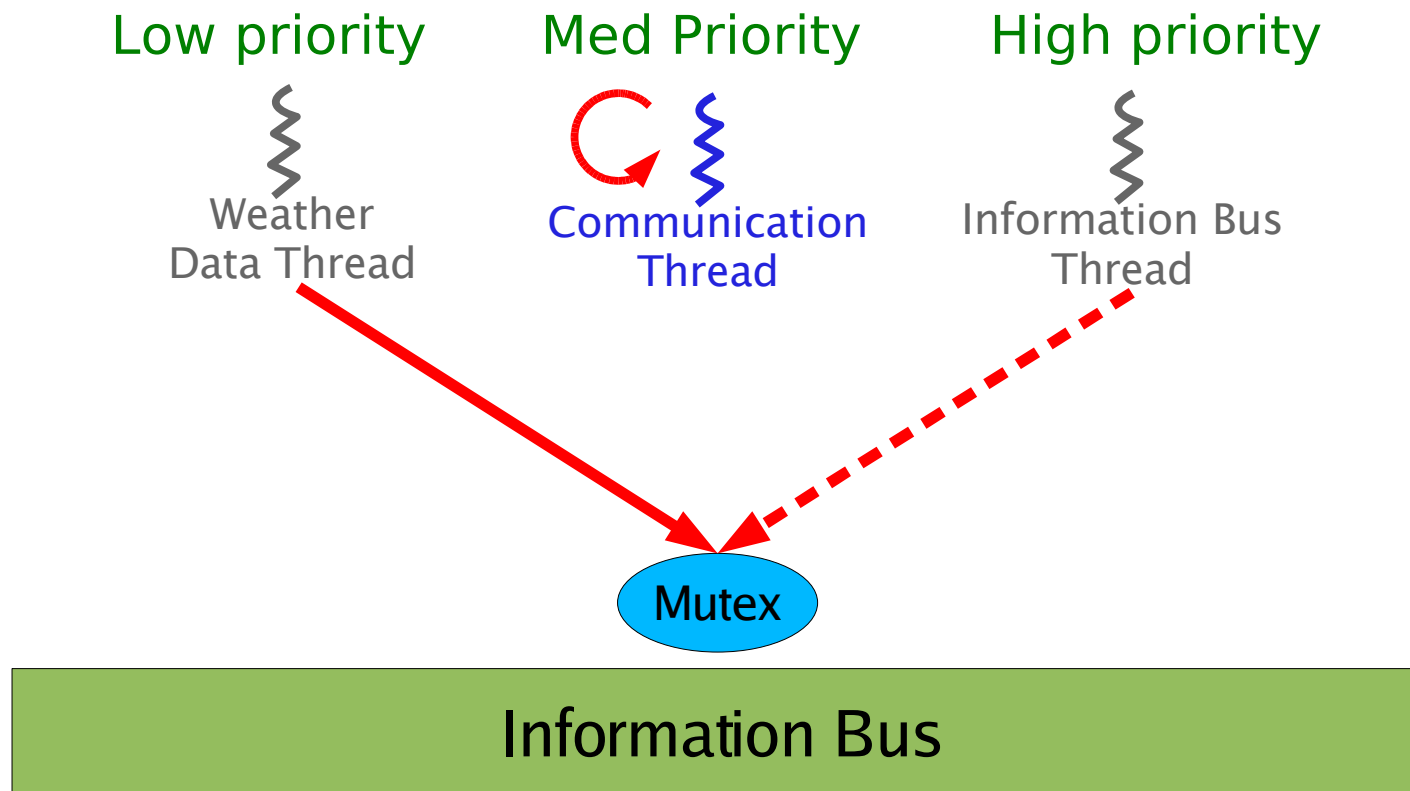
Schedule comm thread ... long running operation



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*



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-prio 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 does this fix the problem?

- Medium priority comm task cannot preempt weather task
- Weather task inherits high priority while it is being waited on

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!!

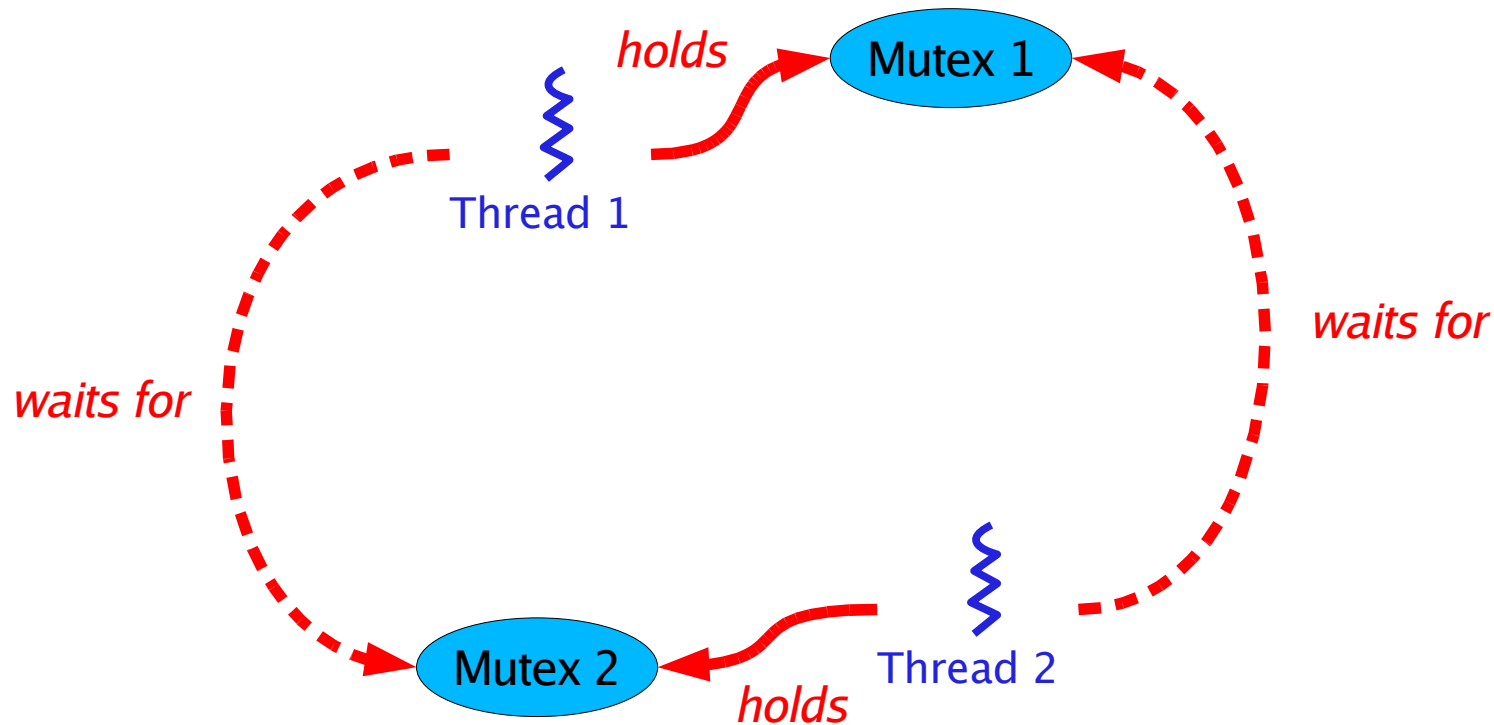
Deadlock

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

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 \neq 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

Getting 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!*

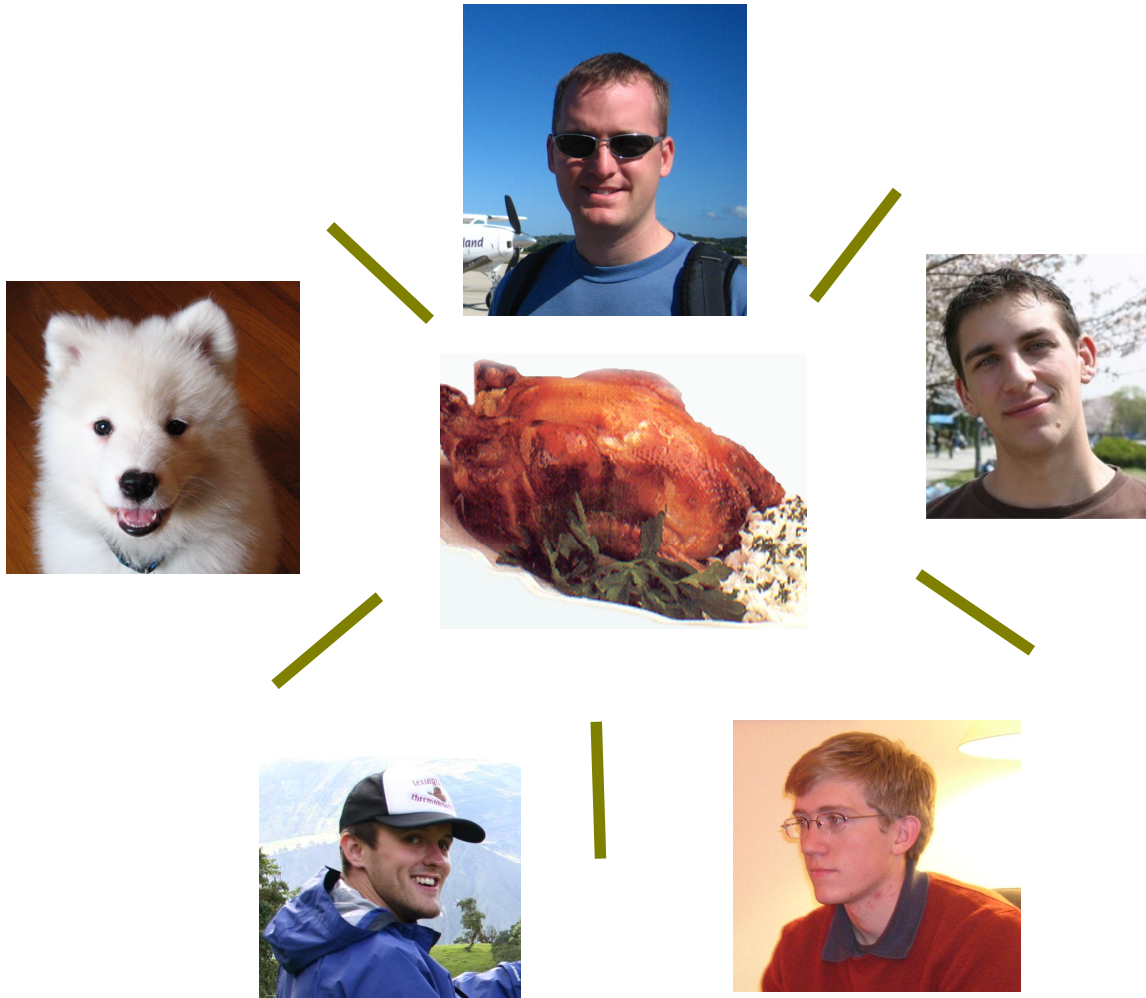
No circular chains of requests?

- This might work too!
- Require threads to grab resources in some predefined order!

Dining Philosophers

Classic deadlock problem

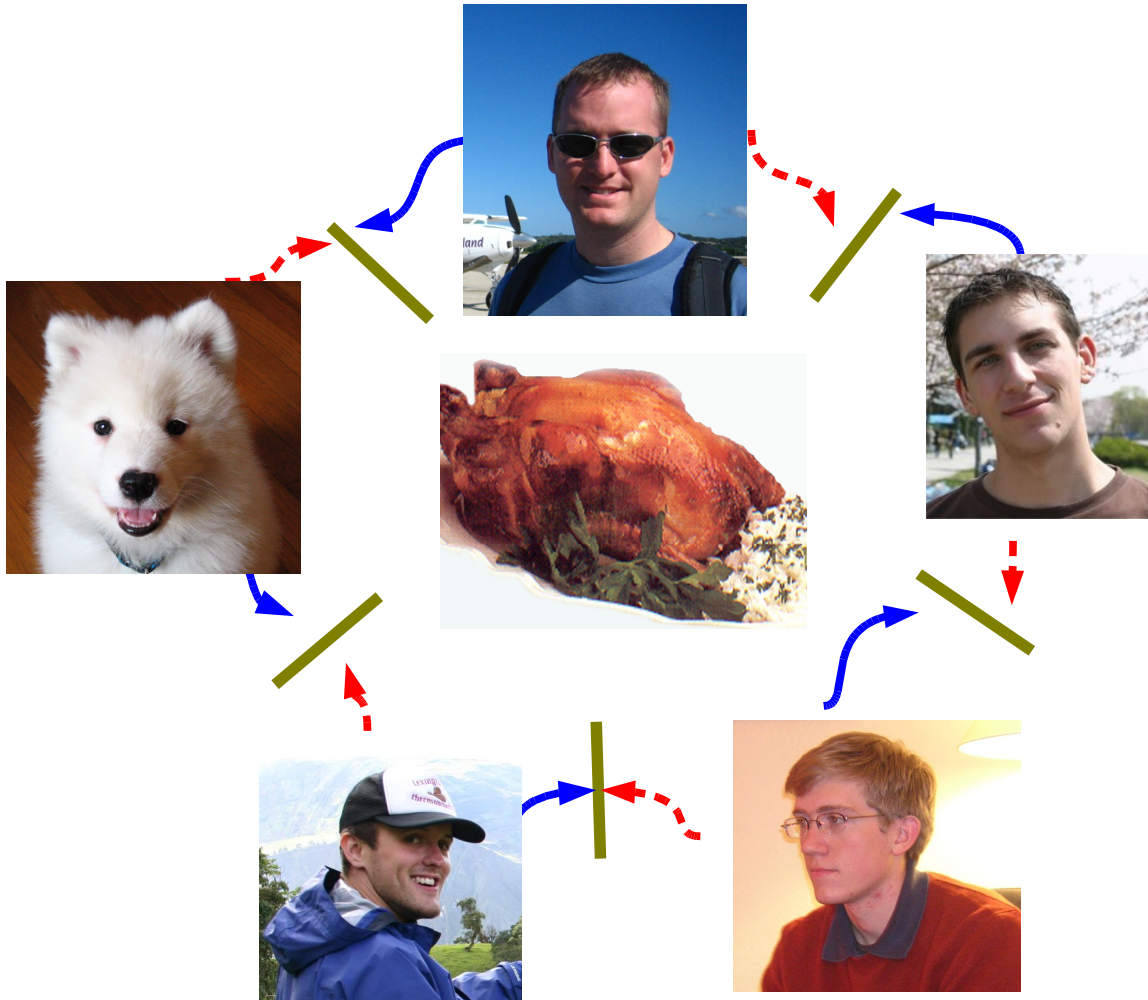
- Multiple philosophers trying to lunch
- One chopstick to left and right of each philosopher
- Each one needs two chopsticks to eat



Dining Philosophers

What happens if everyone grabs the chopstick to their right?

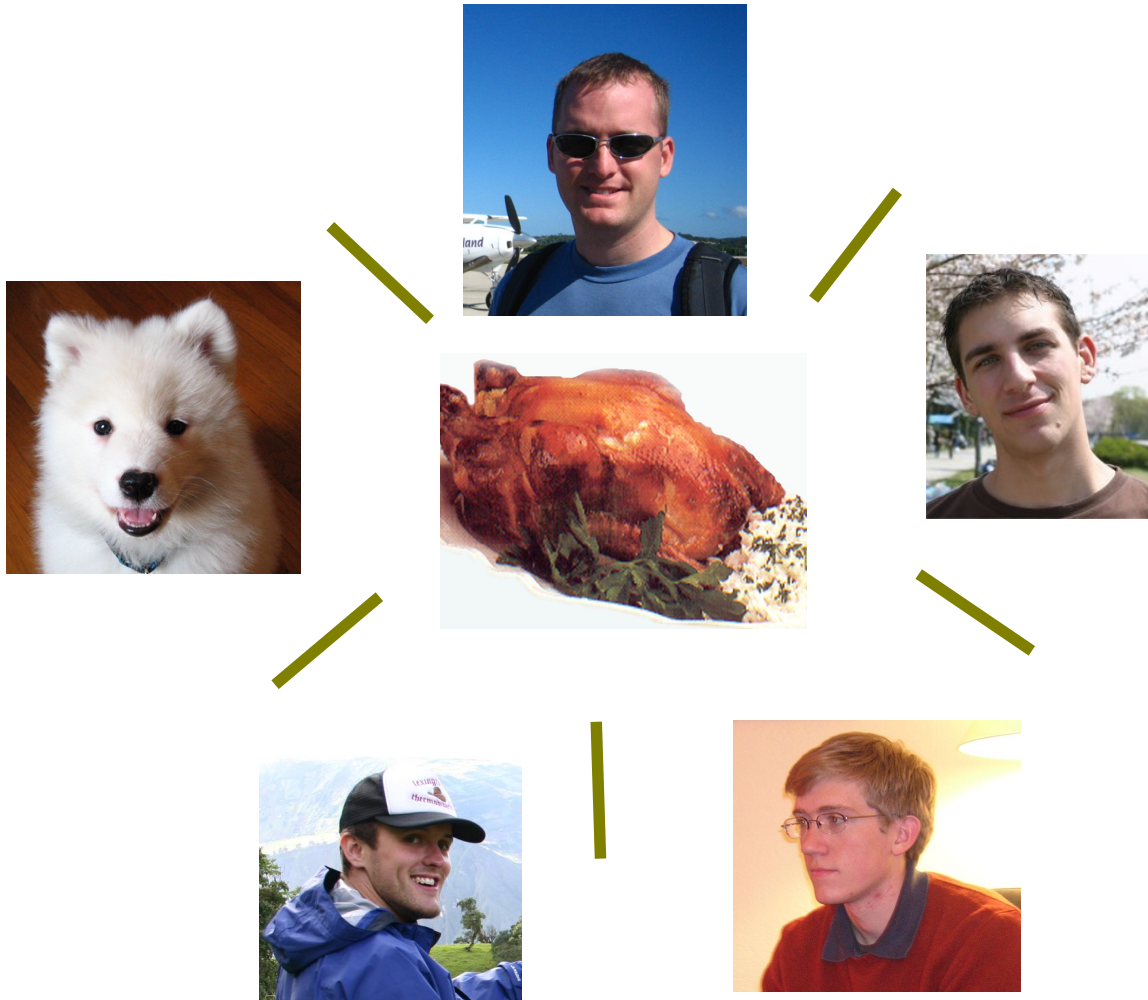
- Everyone gets one chopstick and waits forever for the one on the left
- All of the philosophers starve!!!



Dining Philosophers

How do we solve this problem??

- (Apart from letting them eat with forks.)



How to solve this problem?

Solution 1: Don't wait for chopsticks

- Grab the chopstick on your right
- Try to grab chopstick on your left
- If you can't grab it, put the other one back down
- Breaks “no preemption” condition – no waiting!

Solution 2: Grab both chopsticks at once

- Requires some kind of extra synchronization to make it atomic
- Breaks “multiple independent requests” condition!

Solution 3: Grab chopsticks in a globally defined order

- Number chopsticks 0, 1, 2, 3, 4
- Grab lower-numbered chopstick first
 - *Means one person grabs left hand rather than right hand first!*
- Breaks “circular dependency” condition

Solution 4: Detect the deadlock condition and break out of it

- Scan the waiting graph and look for cycles
- Shoot one of the threads to break the cycle (sorry Geoff !!!)

How to solve this problem?

Solution 1: Don't wait for chopsticks

- Grab the chopstick on your right
- Try to grab chopstick on your left
- If you can't grab it, put the other one back down
- Breaks “no preemption” condition – no waiting!

How to solve this problem?

Solution 1: Don't wait for chopsticks

- Grab the chopstick on your right
- Try to grab chopstick on your left
- If you can't grab it, put the other one back down
- Breaks “no preemption” condition – no waiting!

Solution 2: Grab both chopsticks at once

- Requires some kind of extra synchronization to make it atomic
- Breaks “multiple independent requests” condition!

How to solve this problem?

Solution 1: Don't wait for chopsticks

- Grab the chopstick on your right
- Try to grab chopstick on your left
- If you can't grab it, put the other one back down
- Breaks “no preemption” condition – no waiting!

Solution 2: Grab both chopsticks at once

- Requires some kind of extra synchronization to make it atomic
- Breaks “multiple independent requests” condition!

Solution 3: Grab chopsticks in a globally defined order

- Number chopsticks 0, 1, 2, 3, 4
- Grab lower-numbered chopstick first
 - *Means one person grabs left hand rather than right hand first!*
- Breaks “circular dependency” condition

How to solve this problem?

Solution 1: Don't wait for chopsticks

- Grab the chopstick on your right
- Try to grab chopstick on your left
- If you can't grab it, put the other one back down
- Breaks “no preemption” condition – no waiting!

Solution 2: Grab both chopsticks at once

- Requires some kind of extra synchronization to make it atomic
- Breaks “multiple independent requests” condition!

Solution 3: Grab chopsticks in a globally defined order

- Number chopsticks 0, 1, 2, 3, 4
- Grab lower-numbered chopstick first
 - *Means one person grabs left hand rather than right hand first!*
- Breaks “circular dependency” condition

Solution 4: Detect the deadlock condition and break out of it

- Scan the waiting graph and look for cycles
- Shoot one of the threads to break the cycle (sorry Geoff !!!)

Next Lecture

Scheduling algorithms

- How does the kernel decide which thread to run next?

Read Tanenbaum 2.5-2.7