McGill

ECSE 421 Lecture 12: Embedded Operating Systems

ESD Chapter 4

Last Time

- Embedded Hardware
  - Sensors, Sampling, ADC
  - Information Processing
  - DAC, Actuators

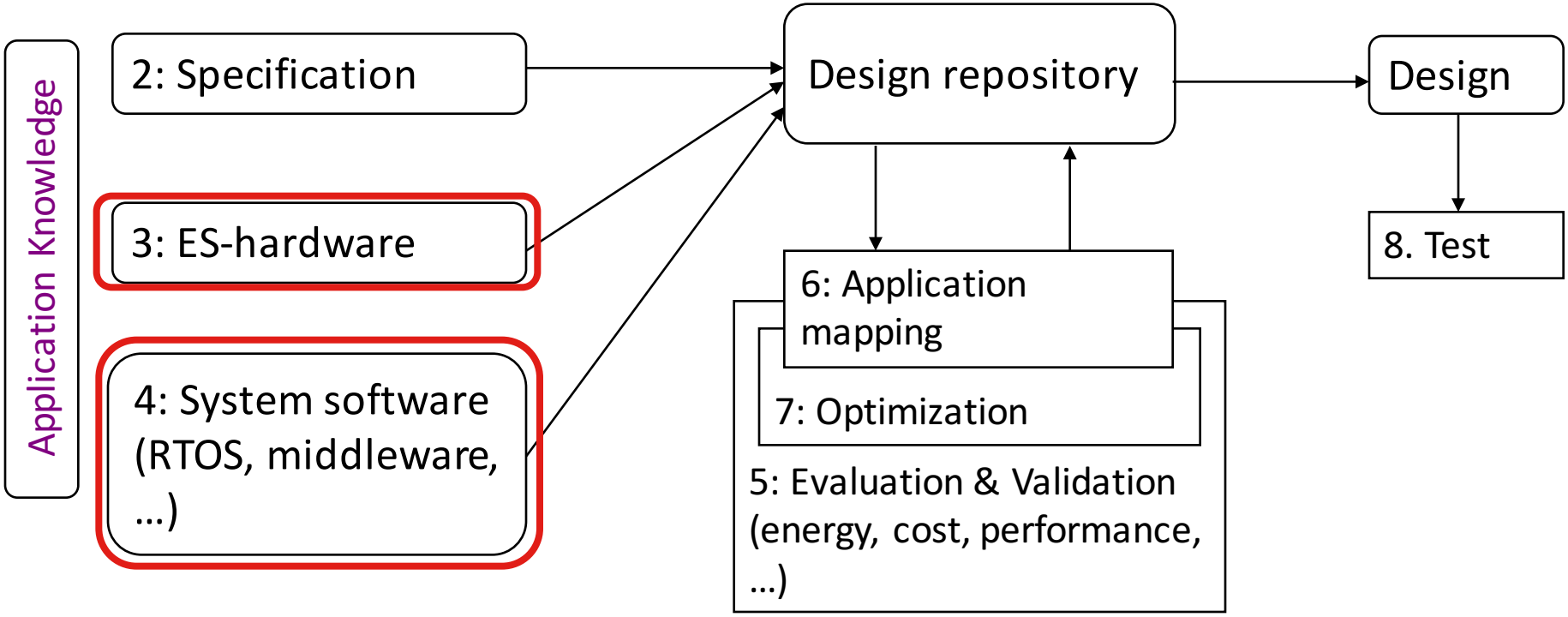
Where Are We?

4	T	2-Feb-2016	L06	Discrete Event Models	2.7	4			G: Zaid Al-bayati
	R	4-Feb-2016	L07	DES / Von Neumann Model of Computation	2.8-2.10	5	LA2	LA1	
5	T	9-Feb-2016	L08	Sensors	3.1-3.2	7.3,12.1-6			
	R	11-Feb-2016	L09	Processing Elements	3.3	12.6-12			
6	T	16-Feb-2016		***No class***				LA2	
	R	18-Feb-2016	L10	More Processing Elements / FPGAs			LA3		
7	T	23-Feb-2016	L11	Memories, Communication, Output	3.4-3.6				
	R	25-Feb-2016		***Midterm exam: in-class, closed book***			P		Chapters 1-3
	T	1-Mar-2016		***No class***					Winter break
	R	3-Mar-2016		***No class***					Winter break
8	T	8-Mar-2016	L12	Embedded Operating Systems	4.1			LA3	
	R	10-Mar-2016	L13	Middleware	4.4-4.5				
9	T	15-Mar-2016	L14	Performance Evaluation	5.1-5.2				
	R	17-Mar-2016	L15	More Evaluation and Validation	5.3-5.8				
10	T	22-Mar-2016	L16	Introduction to Scheduling	6.1-6.2.2				
	R	24-Mar-2016	L17	Scheduling Aperiodic Tasks	6.2.3-6.2.4				
11	T	29-Mar-2016	L18	Scheduling Periodic Tasks	6.2.5-6.2.6				
	R	31-Mar-2016	L19	HW/SW Partitioning	6.3				
12	T	5-Apr-2016	L20	Mapping Applications to Multiprocessors	6.4				
	R	7-Apr-2016	L21	Intro to Compile-time Optimization	7.1-7.2				
13	T	12-Apr-2016	L22	Energy/Memory-aware Compilation	7.3.1-7.3.3				
	R	14-Apr-2016	L23	Further Optimization	7.3.4-7.4				
15	R	28-Apr-2016		***Final Exam: closed book, cumulative***					9:00 AM

Today

- Embedded Operating Systems
  - Requirements
  - Properties
  - Resource access protocols
  - Chapter 4.1

Hypothetical Design Flow



Reuse of Standard Software Components

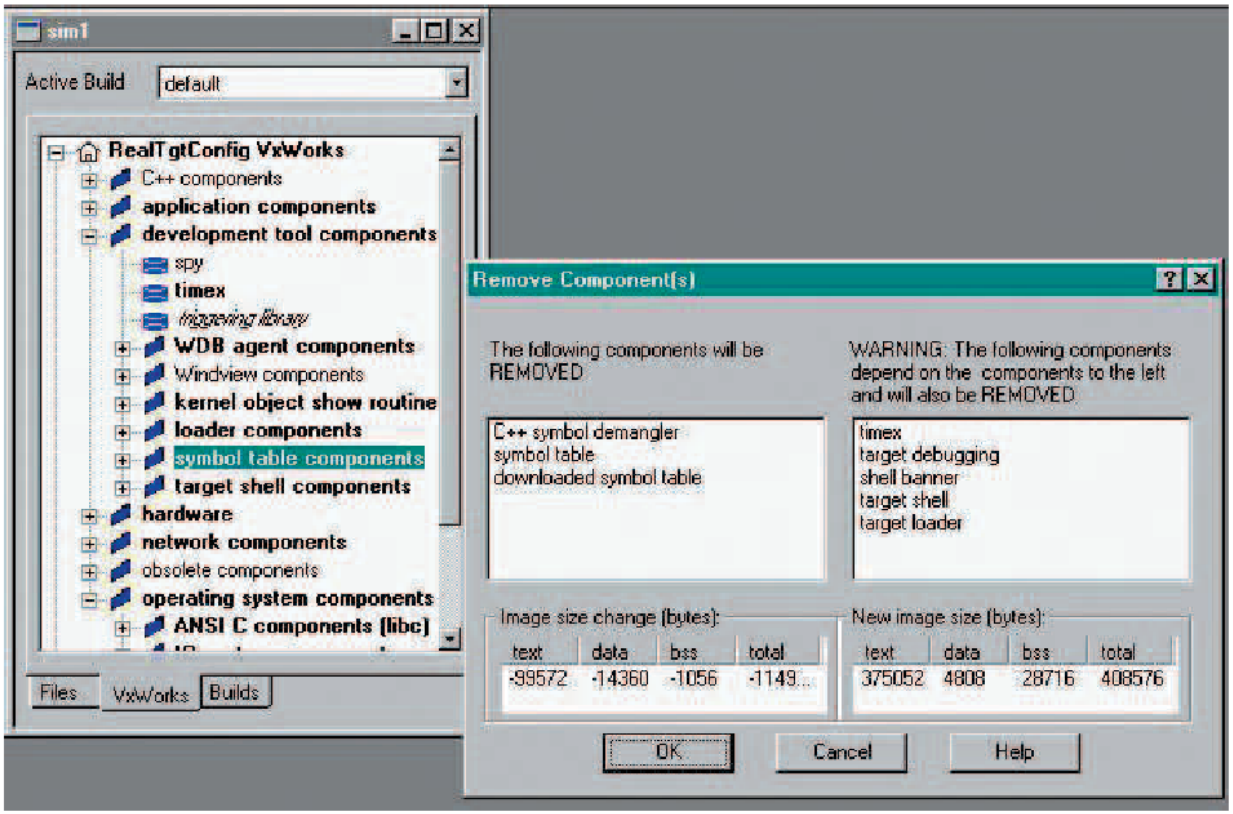
- Embedded systems have time-to-market constraints
- Leverage previous design effort (*intellectual property*) to accelerate current design effort
- Hardware IP
  - Cores, memories, interconnect, *etc*.
- Software IP
  - Operating systems, middleware, *etc*.

Characteristics of Embedded OS (1)

- **Configurability**
- No single OS will fit all needs
- Unnecessary functions should be excluded
  $\Rightarrow$ Configurability needed!
- Simplest form: remove unused functions (at linking)
- Conditional compilation
  - *E.g.*, using pre-processor `#if` and `#ifdef` commands
- Advanced compile-time evaluation
  - Dynamic data might be replaced by static data
- Object-orientation
  - Subclasses derivation, *e.g.*, schedulers
  - Dynamic binding comes with overhead

Example: Configuration of VxWorks

© Wind River

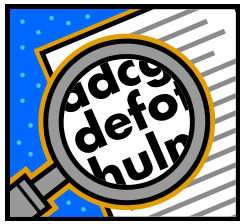


Automatic dependency analysis and size calculations allow users to quickly custom-tailor the VxWORKS operating system.

http://www.windriver.com/products/development_tools/ide/tornado2/tornado2_ds.pdf

Configuration ⇒ Verification

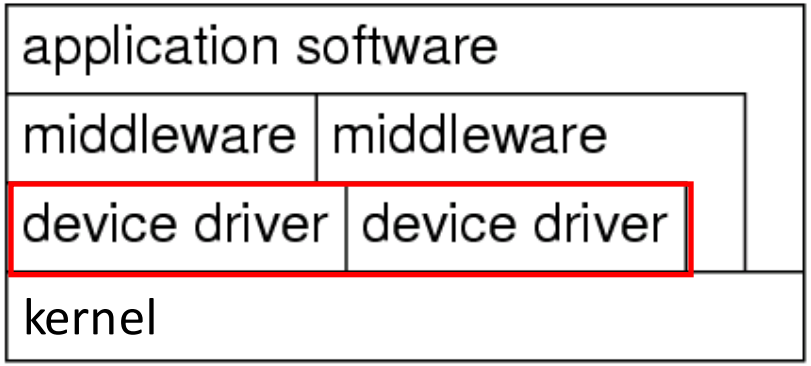
- Each configuration may behave differently
- Each configuration therefore requires verification
  - *E.g.*, does the derived OS achieve the required performance?
- Each derived OS must be tested thoroughly
- Consider eCos (open source RTOS from Red Hat)
  - Includes 100 to 200 configuration points [Takada, 01]



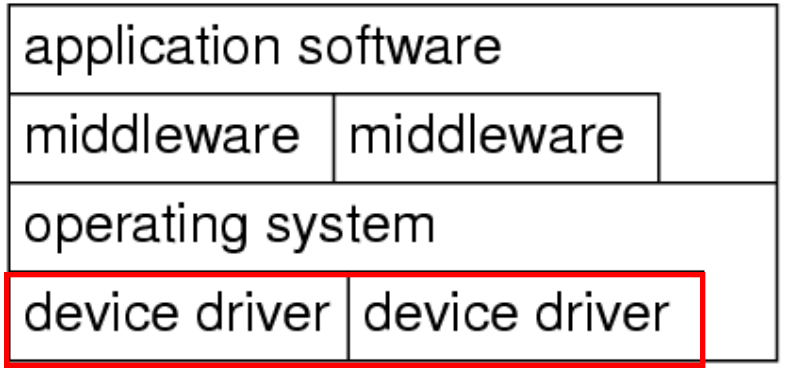
Characteristics of Embedded OS (2)

- **Devices handled directly by tasks**
  - E.g., disc and network
- What devices *must* be supported?
- Besides the system timer, there is no device that must be supported by all variants of an OS
  - Many ES without disc, keyboard, screen or mouse
  - I/O often managed by tasks, not integrated drivers

Embedded OS

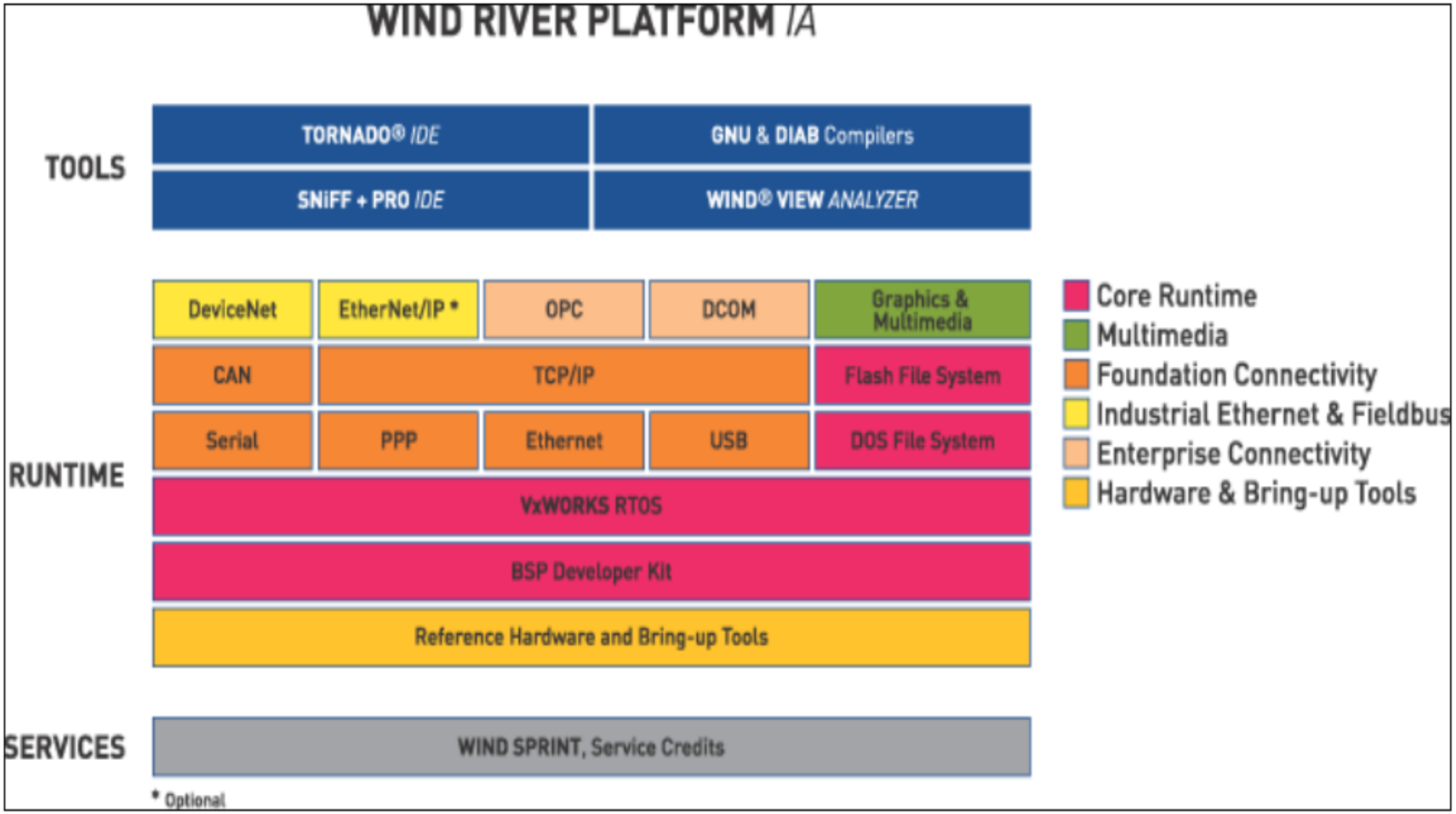


Standard OS



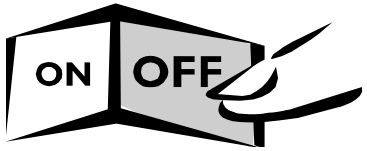
Ex: Wind River Industrial Automation

© Wind River



Characteristics of Embedded OS (3)

- **Protection mechanisms are optional**
- ES are often designed for a single purpose
  - Untested programs rarely loaded (counterexample?)
  - SW considered reliable
- *Privileged* I/O instructions not necessary
  - Tasks can do their own I/O
- Example: Let `switch` be the address of some switch
  - Use `load register,switch`
  - Instead of an OS sys call



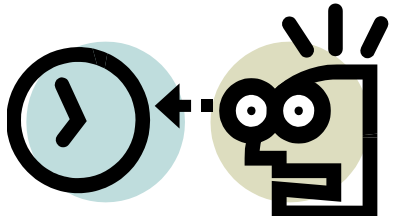
- Protection mechanisms may be needed sometimes for safety and security reasons

Characteristics of Embedded OS (4)

- **Interrupts can be connected to any process**
- For standard OS: serious source of unreliability
- For embedded OS: interrupts can start, stop tasks
  - Embedded programs can be considered to be tested
  - As a result, protection is not necessary
  - Further, efficient control of devices is required
- More efficient than going through OS services
- Results in reduced *composability*
  - If a task is connected to an interrupt, it may be hard to add another task that needs to be started by an interrupt

Characteristics of Embedded OS (4)

- **Real-time capability**
- Many embedded systems are real-time (RT)
  - The OS used in these systems must be real-time operating systems (RTOSs)!



RTOS: Predicability

- **Def**.: *A real-time operating system is an operating system that supports the construction of real-time systems*.
- RTOSs must satisfy three key requirements

1. **The timing behavior of the OS must be predictable**
   - ∀ OS services, there must be a known upper bound on their execution time
   - Only short times during which interrupts are disabled
   - (For hard disks:) files should be contiguous to avoid unpredictable head movements (and therefore, delays)

[Takada, 2001]

RTOS: Managing Timing

2. OS should manage timing and scheduling

- OS may need to be aware of task deadlines (unless scheduling is performed off-line)
- Frequently, the OS should provide precise time services with high resolution

[Takada, 2001]

Time Services

- Time plays a central role in “real-time” systems
- Actual time is described by real numbers
- Two discrete standards are used in real-time equipment
- **International atomic time TAI**
  (French: *Temps Atomic Internationale*)
  - Free of any artificial artifacts
- **Universal Time Coordinated (UTC)**
  - UTC is defined by astronomical standards
- UTC and TAI identical on Jan. 1st, 1958
  - 30 seconds have been added to UTC since then
  - Adjustments not without problems: new year may start twice

Internal Synchronization

- Synchronization with one master clock
  - Typically used during system startup
- Distributed synchronization:
  1. Collect information from neighbors
  2. Compute correction value
  3. Set correction value
- The precision of step 1 depends on how information is collected:
  - Application level: ~500 µs to 5 ms
  - Operation system kernel: 10 µs to 100 µs
  - Communication hardware: < 10 µs

External Synchronization

- External synchronization guarantees consistency with actual physical time
- Trend is to use GPS for external synchronization
- GPS offers TAI and UTC time information
- Resolution is about 100 ns

GPS mouse

© Dell

Problems with External Synchronization

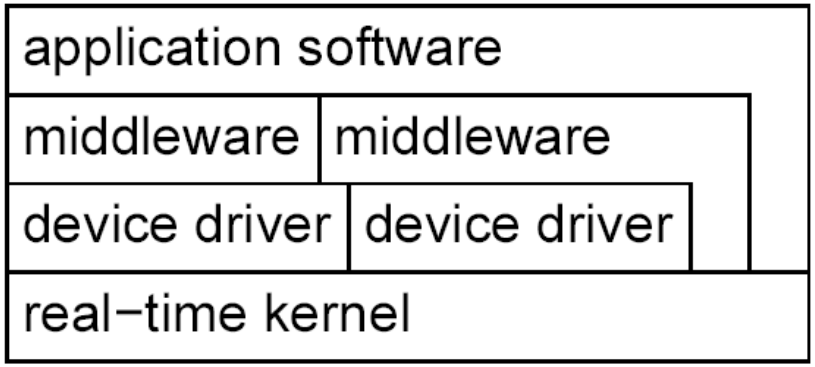
- Fault tolerance
  - Erroneous values are copied to all clocks
  - Mitigation: only accept small changes to local time
- Many time formats too restricted
  e.g.: NTP protocol includes only years up to 2036
- For time services and global clock synchronization see Kopetz, 1997

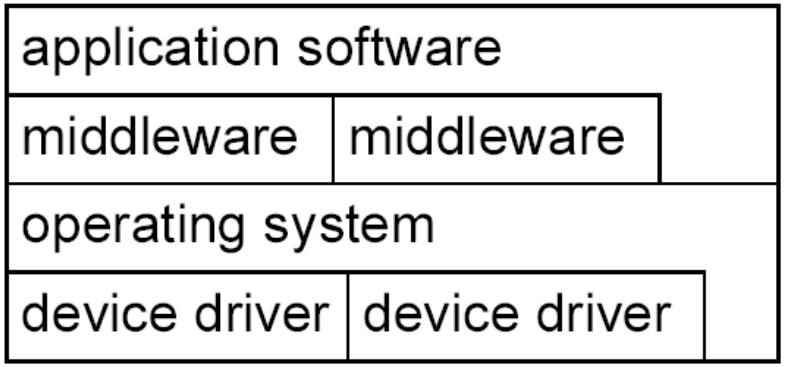
RTOS: Speed

3. **The OS must be fast**
   Practically important.

[Takada, 2001]

RTOS Kernels





- There are important distinctions between …
  - RTOS kernels and RT-enhanced kernels
  - General RTOSs and domain-specific RTOSs
    - E.g., OSEK/VDX and AUTOSAR for automotive applications
  - Standard APIs and Proprietary APIs
    - E.g., standards include POSIX RT-Extension of Unix, ITRON, OSEK/VDX, AUTOSAR, *etc*.

Features of RTOS Kernels

- Processor management,
- Memory management, and
- Timer management;

} *Resource Management*

- Task management (resume, wait etc), and
- Inter-task communication and synchronization.

Classes of RTOS according to R. Gupta

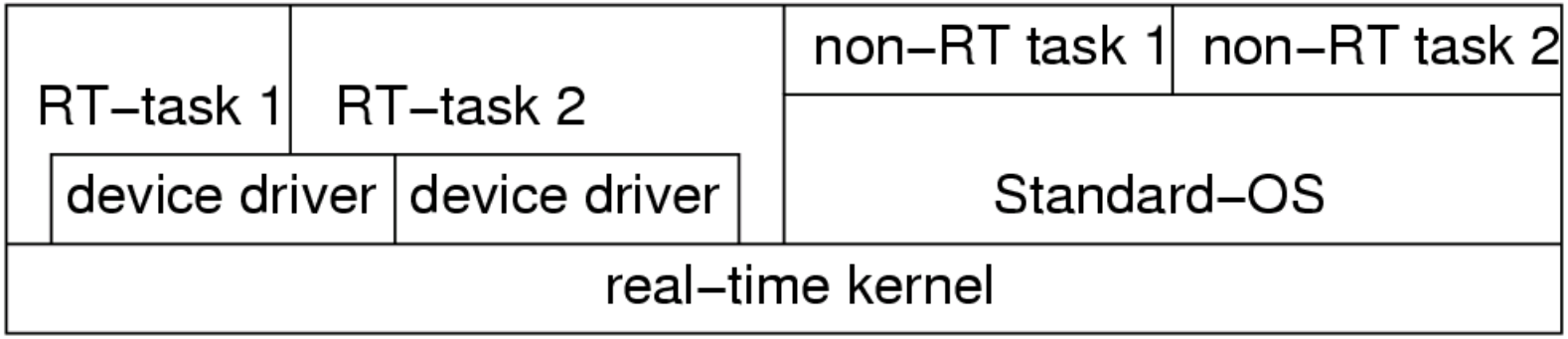
1. Fast proprietary kernels

For complex systems, these kernels are inadequate, because they are designed to be fast, rather than to be predictable in every respect.

[R. Gupta, UCI/UCSD]

- Examples include
  - QNX, PDOS, VCOS, VTRX32, VxWORKS.

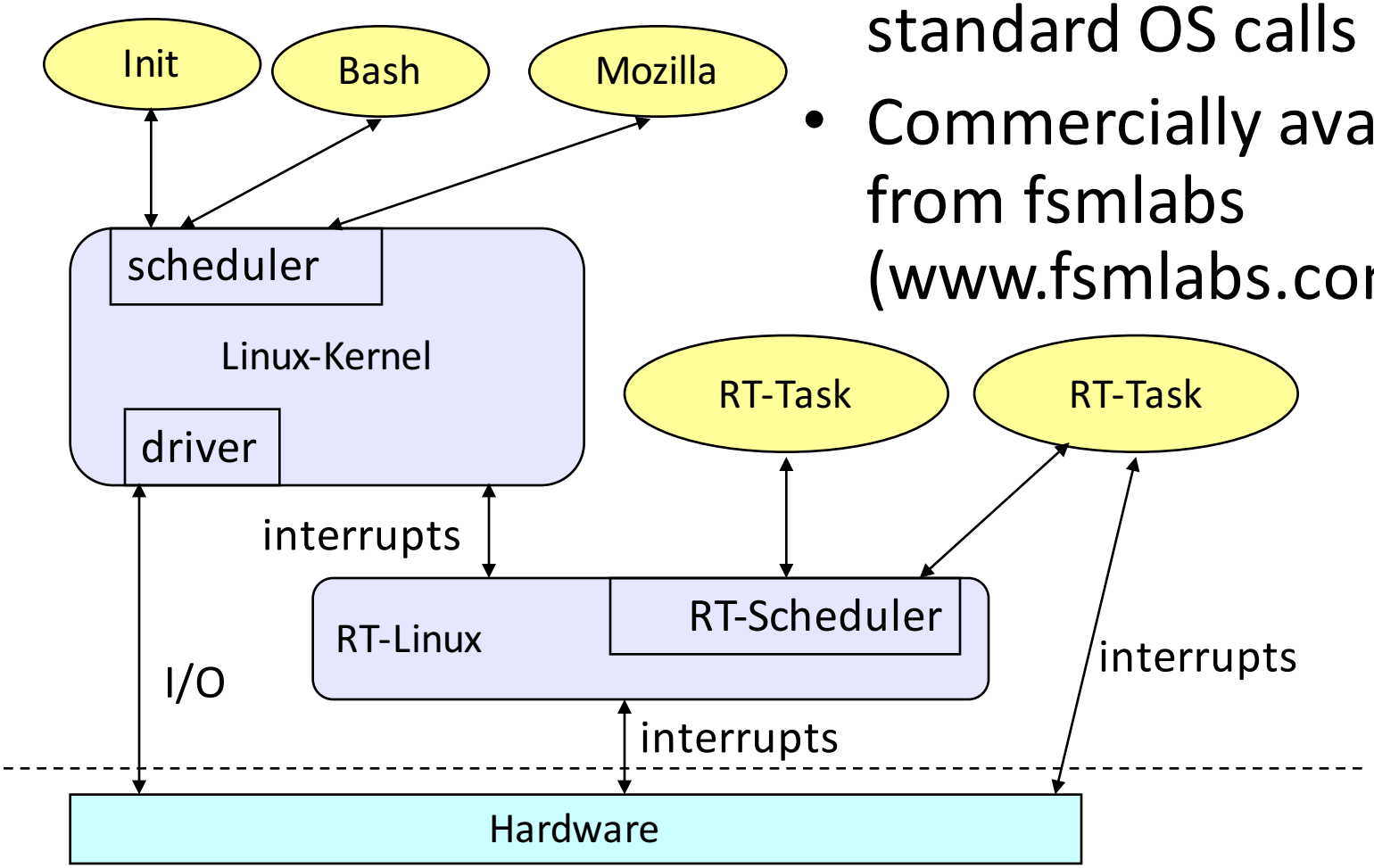
Classes of RTOS according to R. Gupta



2. RT Extensions of Standard OSs

- Attempt to exploit comfortable main stream OS
- RT-kernel running all RT-tasks
- Standard-OS executed as one task
  - Good: Crash of standard-OS does not affect RT-tasks
  - Bad: RT-tasks cannot use Standard-OS services
  - Conclusion: Less comfortable than expected

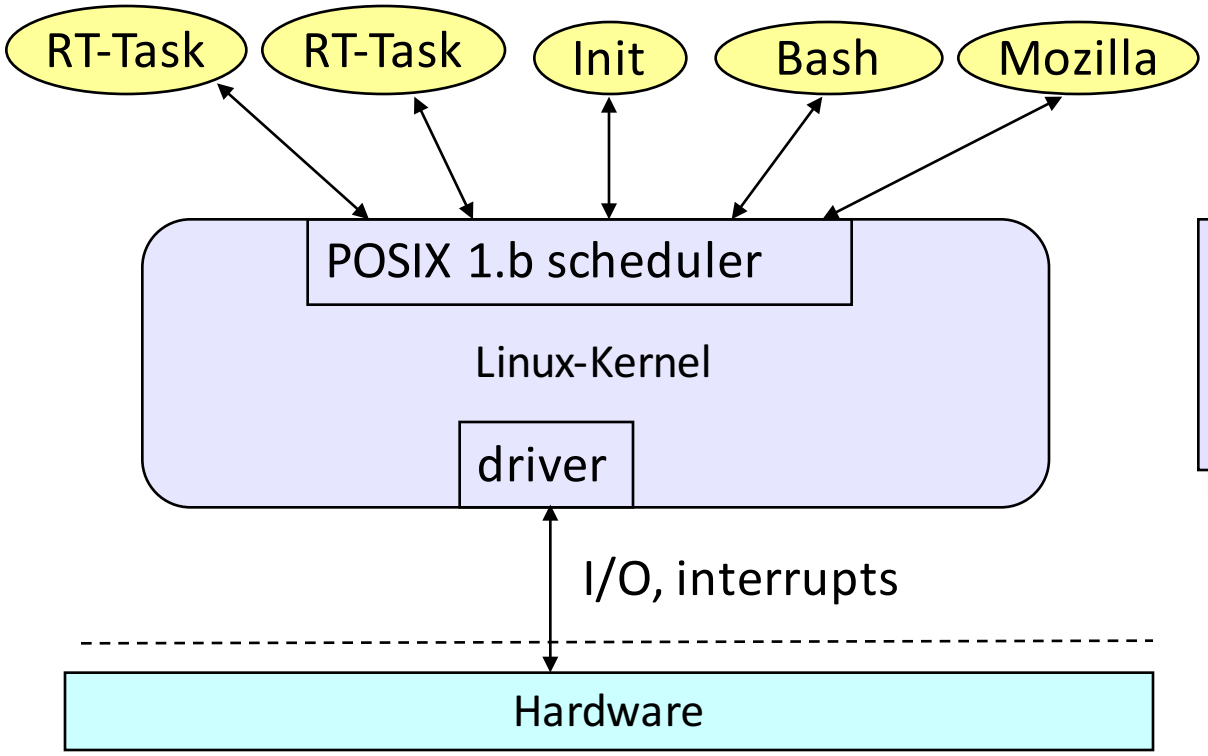
Example: RT-Linux



- RT-tasks cannot use standard OS calls
- Commercially available from fsmlabs (www.fsmlabs.com)

Example: POSIX 1.b RT-extensions to Linux

- Standard scheduler can be replaced by POSIX scheduler implementing priorities for RT tasks



Special RT-calls and standard OS calls available.
Easy programming, no guarantee of meeting deadline

Evaluation of RT Extensions (Gupta)

- According to Gupta, trying to use a version of a standard OS is:

  not the correct approach because too many basic and inappropriate underlying assumptions still exist such as optimizing for the average case (rather than the worst case), ... ignoring most if not all semantic information, and independent CPU scheduling and resource allocation.

- Key support missing for task dependencies
  - Uncommon for standard applications; frequently ignored
  - For ES applications, dependencies between tasks are the rule rather than the exception

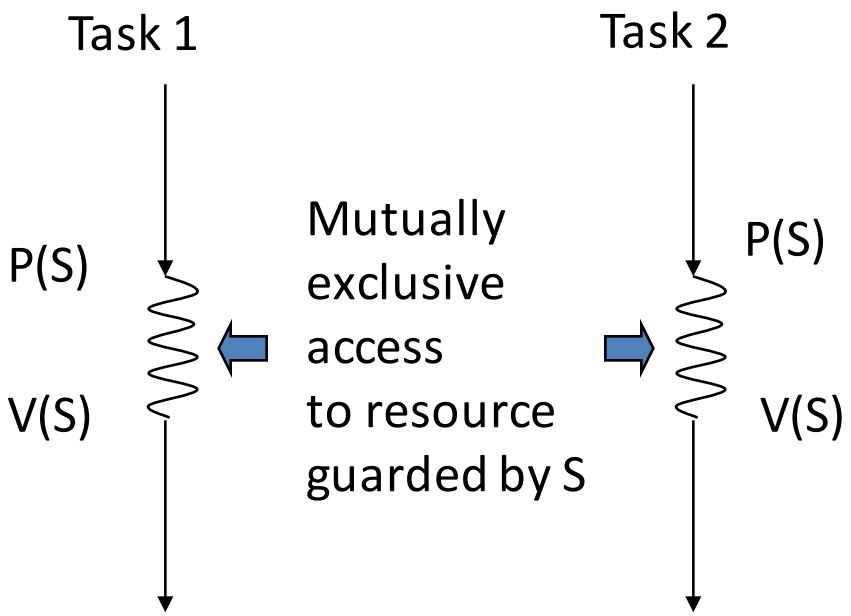
Classes of RTOS according to R. Gupta

3. Research systems trying to avoid limitations

- *E.g.*, MARS, Spring, MARUTI, Arts, Hartos, DARK, and Melody
- Research issues [Takada, 2001]:
  - low overhead memory protection
  - temporal protection of computing resources
  - RTOS for on-chip multiprocessors
  - support for continuous media
  - quality of service (QoS) control

Resource Access Protocols

- **Critical sections:** sections of code in which exclusive access to some resource must be guaranteed
- Can be guaranteed with semaphores S or "mutexes"



P(S) checks semaphore to see if resource is available

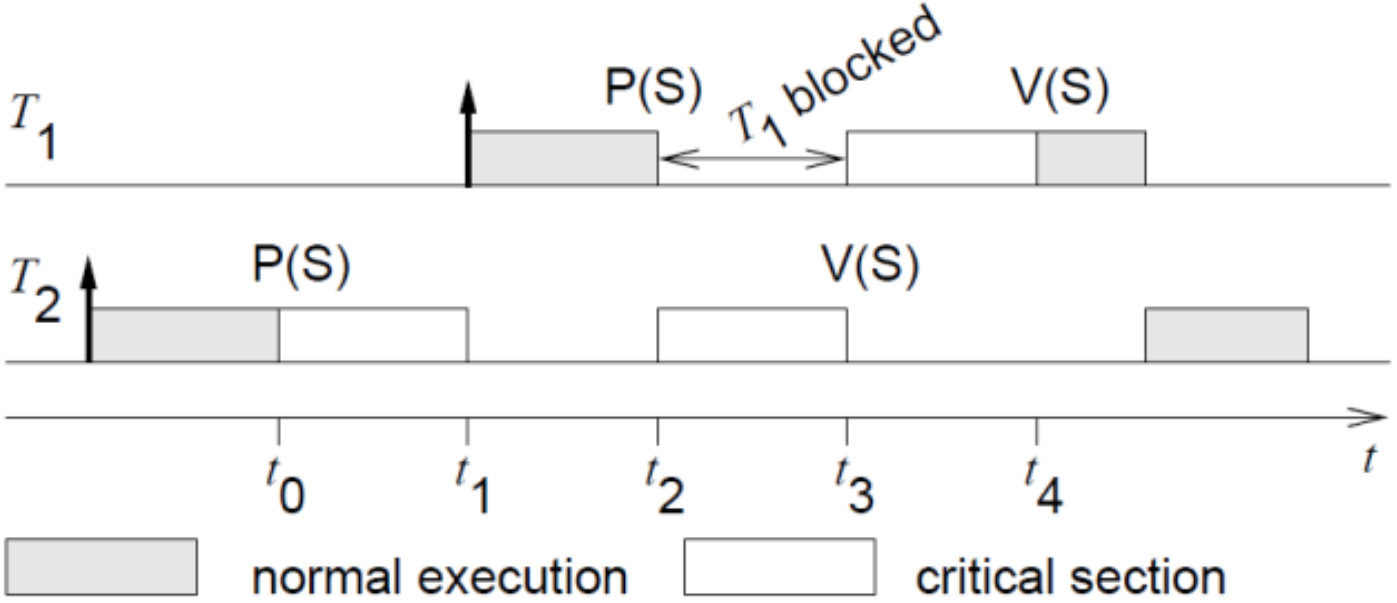
If yes, sets S to "used"
Uninterruptible operations follow!
If no, calling task has to wait

V(S): sets S to "unused" and starts a sleeping task (if any)

Blocking Due to Mutual Exclusion

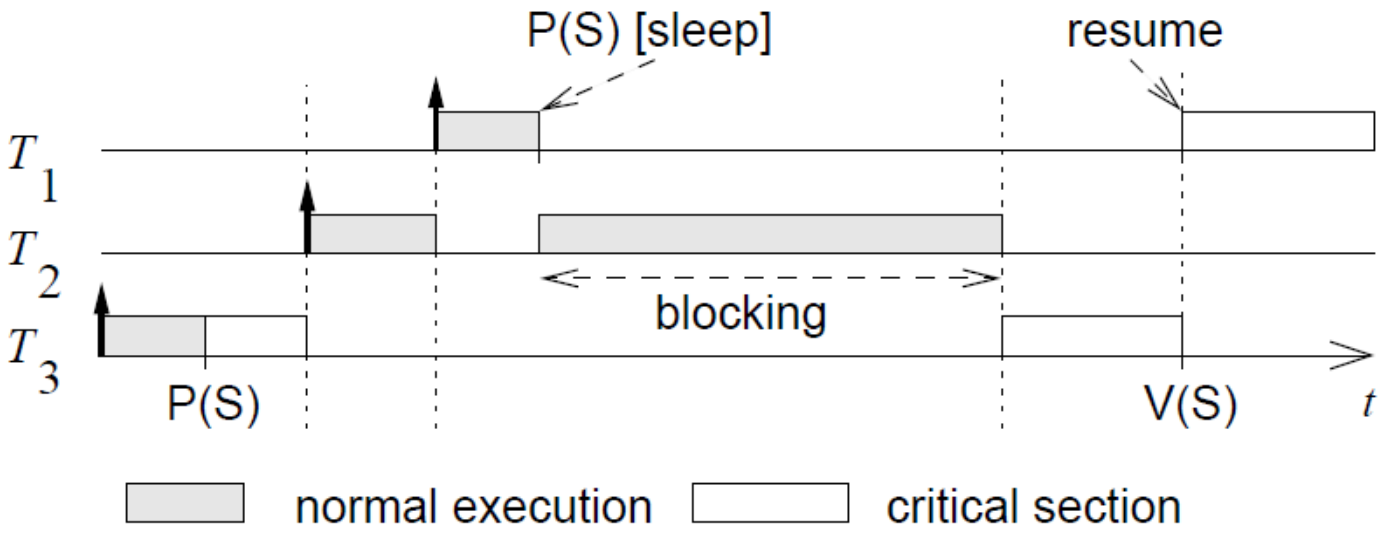
- Assume the priority of T_1 is higher than the priority of T_2
- If T_2 requests exclusive access first (at t_0)
  - T_1 has to wait until T_2 releases the resource (time t_3)
  - Priority is *inverted*: problematic, difficult to avoid

The time T_1 is blocked is bounded by the length of T_2's critical section



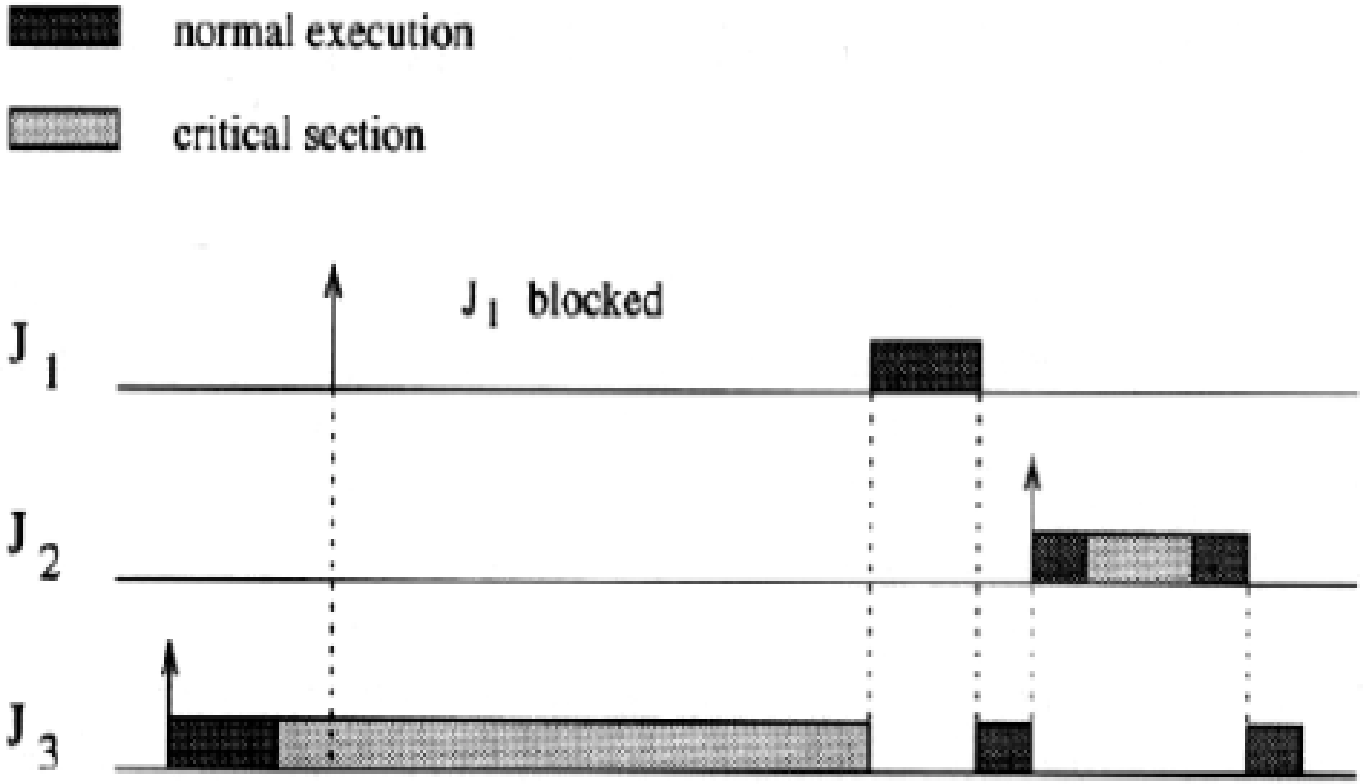
But Wait, There's More!

- Priority of T_1 > priority of T_2 > priority of T_3
- T_2 preempts T_3
- T_2 can prevent T_3 from releasing the resource
  - T_2 blocks T_1 though T_2 is not in a critical section
  - Blocking time is unbounded!
- This is called ***priority inversion***



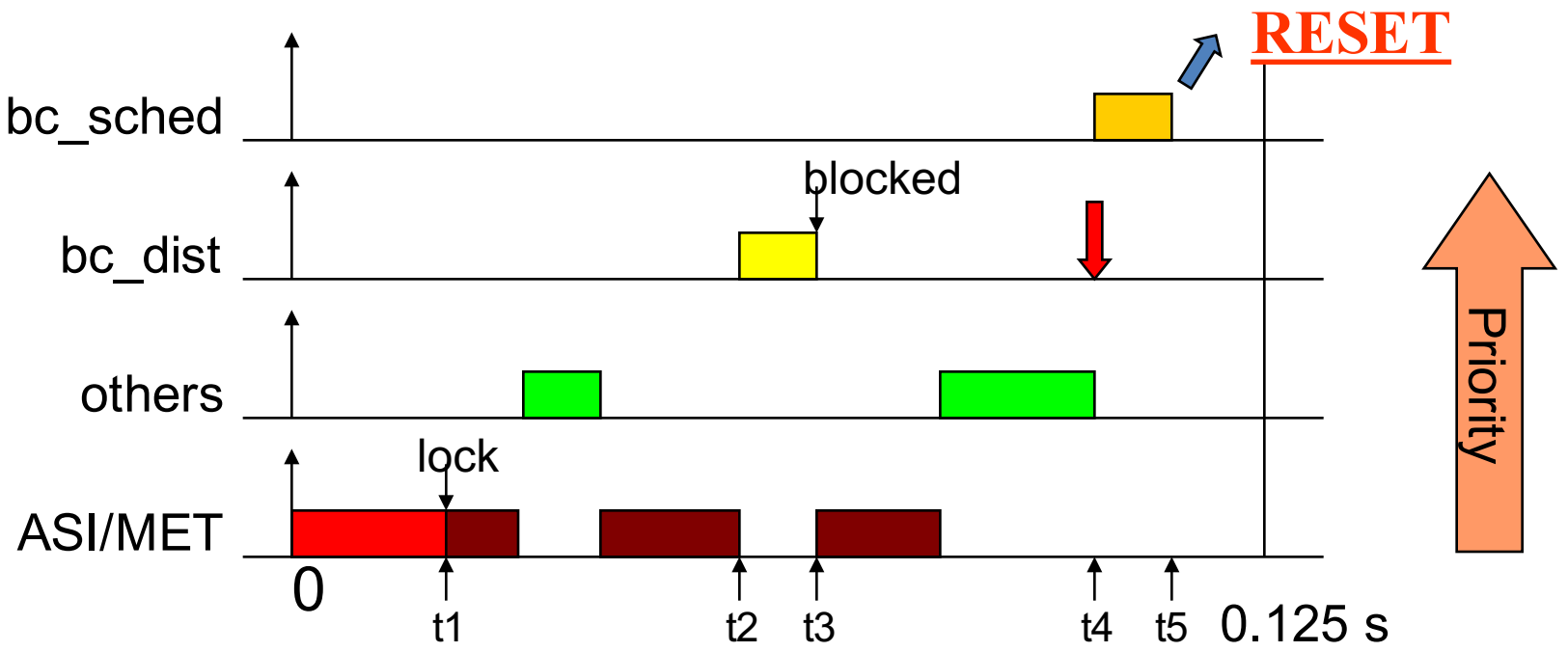
Solutions

- ***Disallow preemption*** during the execution of all critical sections. Simple, but creates unnecessary blocking as unrelated tasks may be blocked.



The Mars Pathfinder

Priority Inversion



- ASI/MET acquires control of the bus (shared resource)
- bc_dist preempts the lower priority task
- bc_dist attempts to lock the resource: blocked!
- bc_sched is activated; bc_dist executes after its deadline
- bc_sched detects the bc_dist timing error and resets the system

Mitigating Inversion: Priority Inheritance

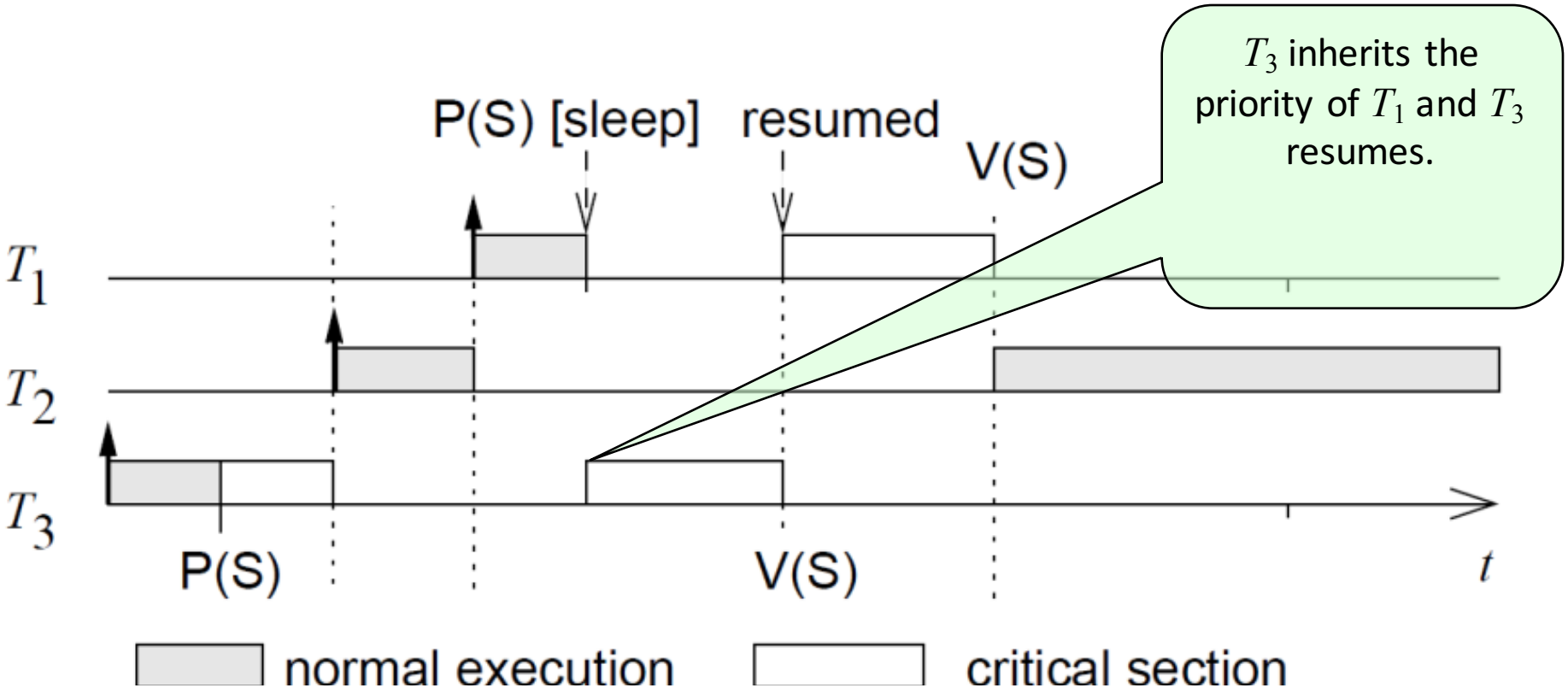
- Tasks are scheduled according to their active priorities
- Tasks with the same priorities are scheduled FCFS
- If task T_1 executes P(S) and T_2 already has exclusive access
  - T_1 will become blocked
- If priority(T_2) < priority(T_1)
  - T_2 inherits the priority of T_1, and T_2 therefore resumes
- Rule: tasks inherit the highest priority of tasks blocked by it

Priority Inheritance, Cont'd

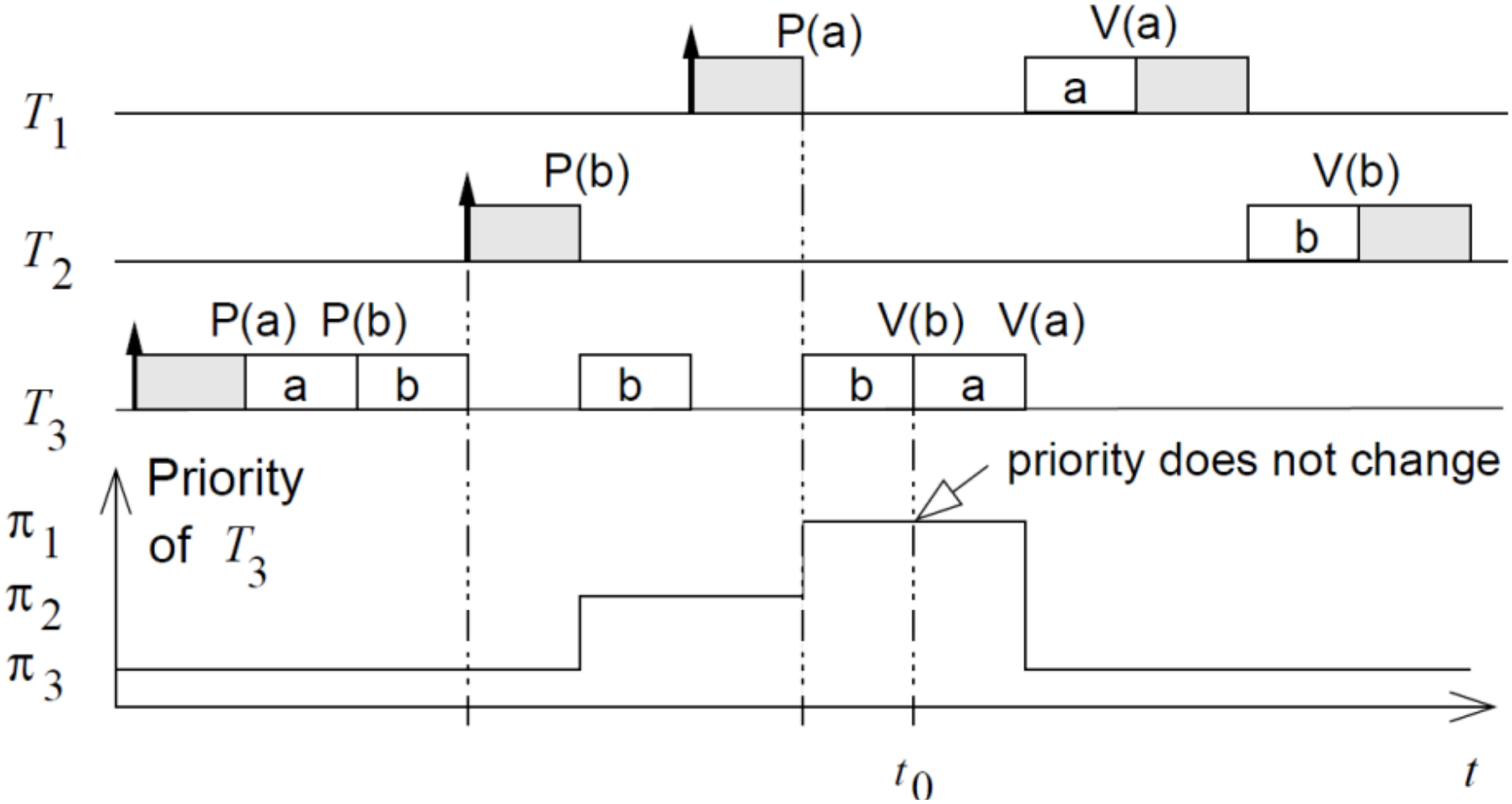
- When T_2 executes V(S), its priority is decreased to the highest priority of the tasks blocked by it
  - If no other task blocked by T_2: priority(T_2) := original
  - Highest priority task so far blocked on S is resumed
- Priority Inheritance is Transitive
  - if T_2 blocks T_1 and T_1 blocks T_0, then T_2 inherits the priority of T_0

Priority Inheritance Example

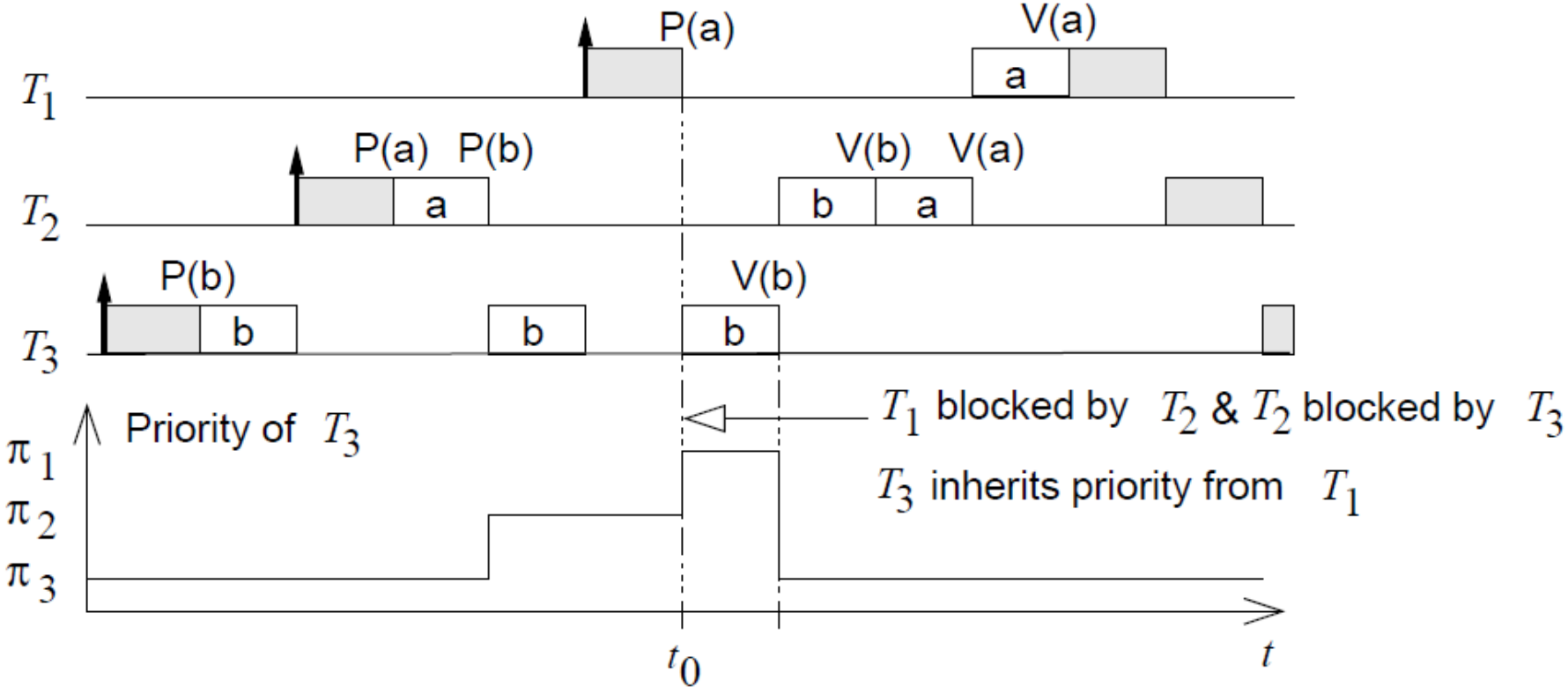
- How would priority inheritance affect our example with three tasks?



Nested Critical Sections

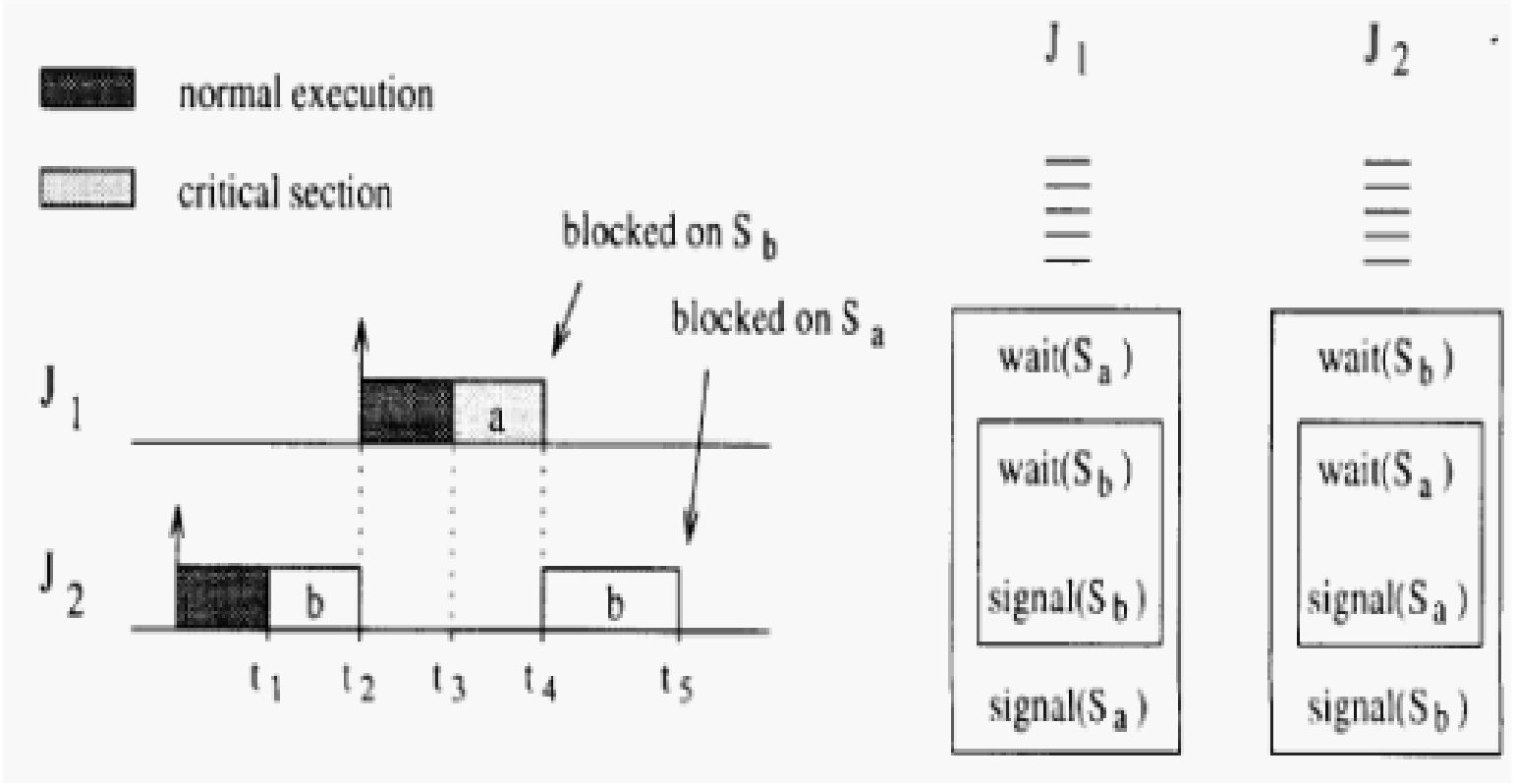


Transitivity of Priority Inheritance



Priority Inheritance Protocol (PIP)

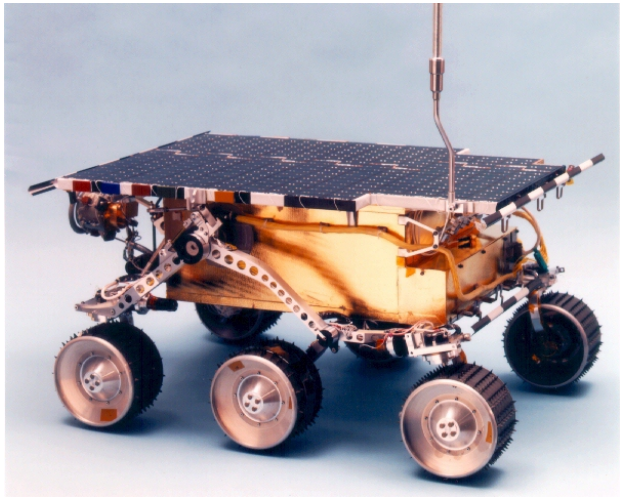
- Problem: ***Deadlock***



[But97, S. 200]

Priority Inversion on Mars

- Priority inheritance also solved the Pathfinder problem
- The VxWorks operating system used in the Pathfinder implements a flag for the calls to mutex primitives
  - This flag allows priority inheritance to be set to "on"
  - When the software was shipped, it was set to "off"
- This was corrected with the VxWorks debugging facilities
  - While the Pathfinder was on Mars
  - Changed the flag to "on"
  - [Jones, 1997]



Is Priority Inheritance the Final Word?

- Possible large number of tasks with high priority
- Possible deadlocks
- Ongoing debate about problems with the protocol

  Victor Yodaiken: Against Priority Inheritance, Sept. 2004, http://www.fsmlabs.com/resources/white_papers/priority-inheritance/
- Finds application in ADA: During *rendez-vous* task priority is set to the maximum
- An alternative protocol for fixed a set of tasks: *priority ceiling*

Summary

- Requirements for embedded operating systems
  - Configurability, task-managed I/O, interrupts
- Properties of real-time operating systems
  - Predictability
  - Scheduling of dependent tasks with deadlines
  - Time services and synchronization
  - Classes of RTOSs
- Resource Access Protocols
  - Priority inversion and priority inheritance

Next Time

- Middleware
  - Chapters 4.4-5