


# Embedded Software

**Kai Huang**



- CSE 2014 On-Going





- *Global Embedded Software Market To Grow At A CAGR Of 10.7 Percent Over The Period 2012-2016*

- Embedded Software Market - 2013 Report

- According to the report, main driver being the increased demand for embedded software is in the Automotive segment. Embedded software is used in the development of in-vehicle infotainment systems, in-car navigation, and telematics. It provides users with additional entertainment and communication features that are now major requirements for vehicle buyers. One major challenge confronting the market is the safety and security issues associated with the use of embedded software. There are higher chances of the embedded software to malfunction as opposed to application software.

[http://www.researchandmarkets.com/research/2x3m7b/global\\_embedded](http://www.researchandmarkets.com/research/2x3m7b/global_embedded)



# Outline

- Embedded Operating System
- Resource Access Protocols



# Why an OS at all?

- Why not just bare-metal
  - Same reasons why we need one for a traditional computer.
  - Not all services are needed for any device.
- Large variety of requirements and environments:
  - Critical applications with high functionality (medical applications, space shuttle, ...).
  - Critical applications with small functionality (ABS, pace maker, ...)
- Not very critical applications with varying functionality (PDA, phone, smart card, microwave, oven, ...)

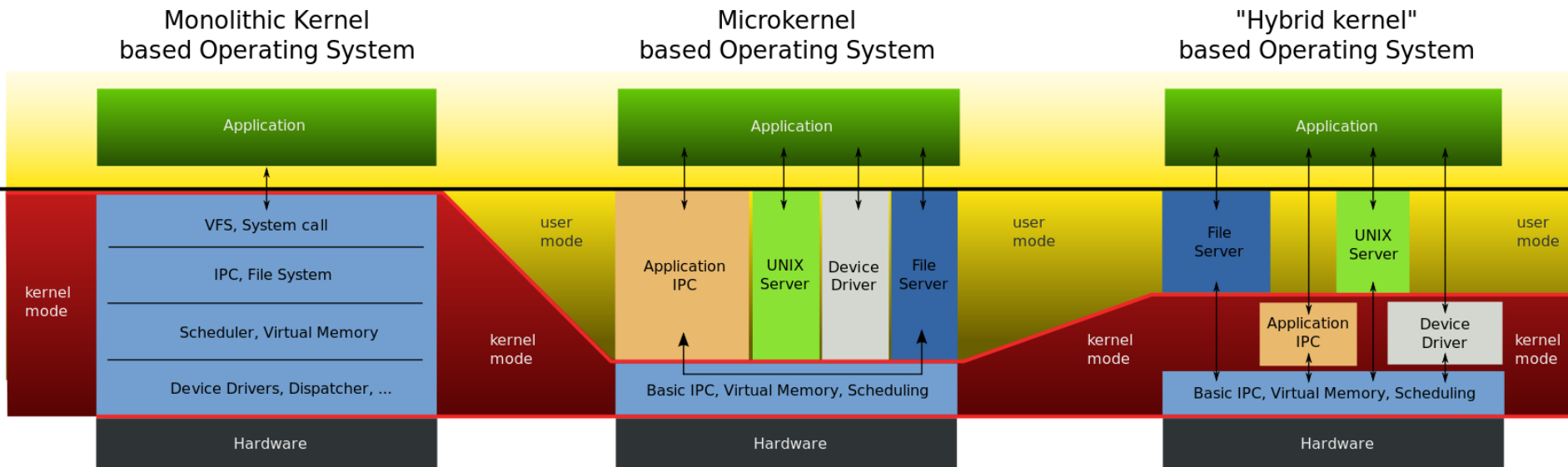


# Why is a Desktop OS not Suited?

- Monolithic kernel is too feature reach.
- Monolithic kernel is not modular, fault-tolerant, configurable, modifiable, ... .
- Takes too much space.
- Not power optimized.
- Not designed for mission-critical applications.



# Operating System Architecture



From wikipedia



# Embedded OS - Configurability

- No single RTOS will fit all needs, no overhead for unused functions/data tolerated, configurability needed.
  - Simplest form: remove unused functions (by linker for example).
  - Conditional compilation (using `#if` and `#ifdef` commands).
  - Static data might be replaced by dynamic data (but then real-time behavior is in danger).





# Example: Configuration of VxWorks

The screenshot displays the Wind River Workbench interface, specifically the 'Filesystem Layout Configuration' window. The window shows a table of files and directories with columns for File, Mode, Size, Type, Owner/Group, Package, and Location. The 'games' directory is highlighted in blue.

File	Mode	Size	Type	Owner/Group	Package	Location
tmp	drwxrwxrwt	4096	directory	root/root	filesystem-2.4.0-1	
usr	drwxr-xr-x	4096	directory	root/root	filesystem-2.4.0-1	
var	drwxr-xr-x	4096	directory	root/root	filesystem-2.4.0-1	
apache	drwxrwxrwx	4096	directory	root/root		
proxy	drwxr-xr-x	4096	directory	root/root	apache-ssl-1.3.37	
apache.config	-rw-r--r--	4	file	dreyna/users	added	/tmp/apache.config
cache	drwxr-xr-x	4096	directory	root/root	filesystem-2.4.0-1	
db	drwxr-xr-x	4096	directory	root/root	filesystem-2.4.0-1	
empty	drwxr-xr-x	4096	directory	root/root	filesystem-2.4.0-1	
games	drwxr-xr-x	4096	directory	root/root	filesystem-2.4.0-1	
kerberos	drwxr-xr-x	4096	directory	root/root	krb5-server-1.6-3	
krb5kdc	drwxr-xr-x	4096	directory	root/root	krb5-server-1.6-3	
lib	drwxr-xr-x	4096	directory	root/root	arm-versatile_92	
alternatives	drwxr-xr-x	4096	directory	root/root	chkconfig-1.3.34	
dhclient	drwxr-xr-x	4096	directory	root/root	dhclient-3.0.5-34	
dhcpd	drwxr-xr-x	4096	directory	root/root	dhcp-3.0.5-34_W	
dhcpd.leases	-rw-r--r--	0	empty	root/root		dhcp-3.0.5-34_W
games	drwxr-xr-x	4096	directory	root/root	filesystem-2.4.0-1	
misc	drwxr-xr-x	4096	directory	root/root	filesystem-2.4.0-1	
nfs	drwxr-xr-x	4096	directory	root/root	nfs-utils-1.0.12-4	
rpc_pipefs	drwxr-xr-x	4096	directory	root/root	nfs-utils-1.0.12-4	
statd	drwx-----	4096	directory	rpcuser/rpcuser	nfs-utils-1.0.12-4	

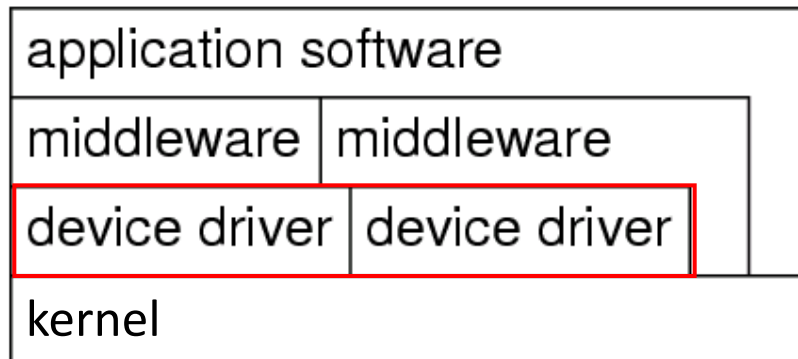
http://www.windriver.com/products/product-notes/PN\_Workbench\_0611.pdf



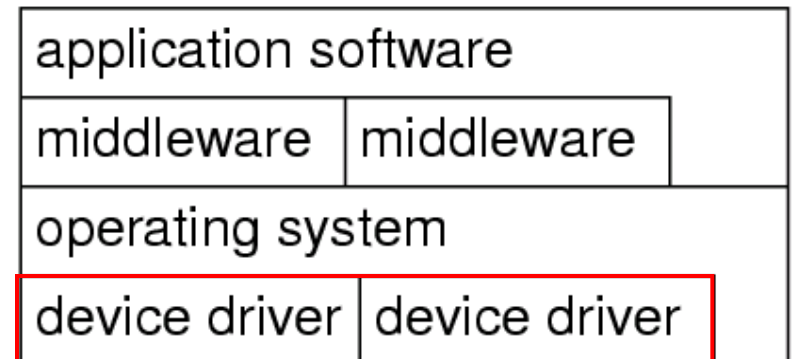
# Embedded OS - Device Driver

- Device drivers handled by tasks instead of integrated drivers:
  - Improve predictability; everything goes through scheduler
  - Effectively no device that needs to be supported by all versions of the OS, except maybe the system timer.

Embedded OS



Standard OS



# Embedded OS - Interrupt

- Interrupts can be employed by any process
  - For standard OS: would be serious source of unreliability.
  - Embedded programs can be considered to be tested ... .
  - It is possible to let interrupts directly start or stop tasks (by storing the tasks start address in the interrupt table). More efficient and predictable than going through OS services.
  - However, composability suffers: if a specific task is connected to some interrupt, it may be difficult to add another task which also needs to be started by the same event.

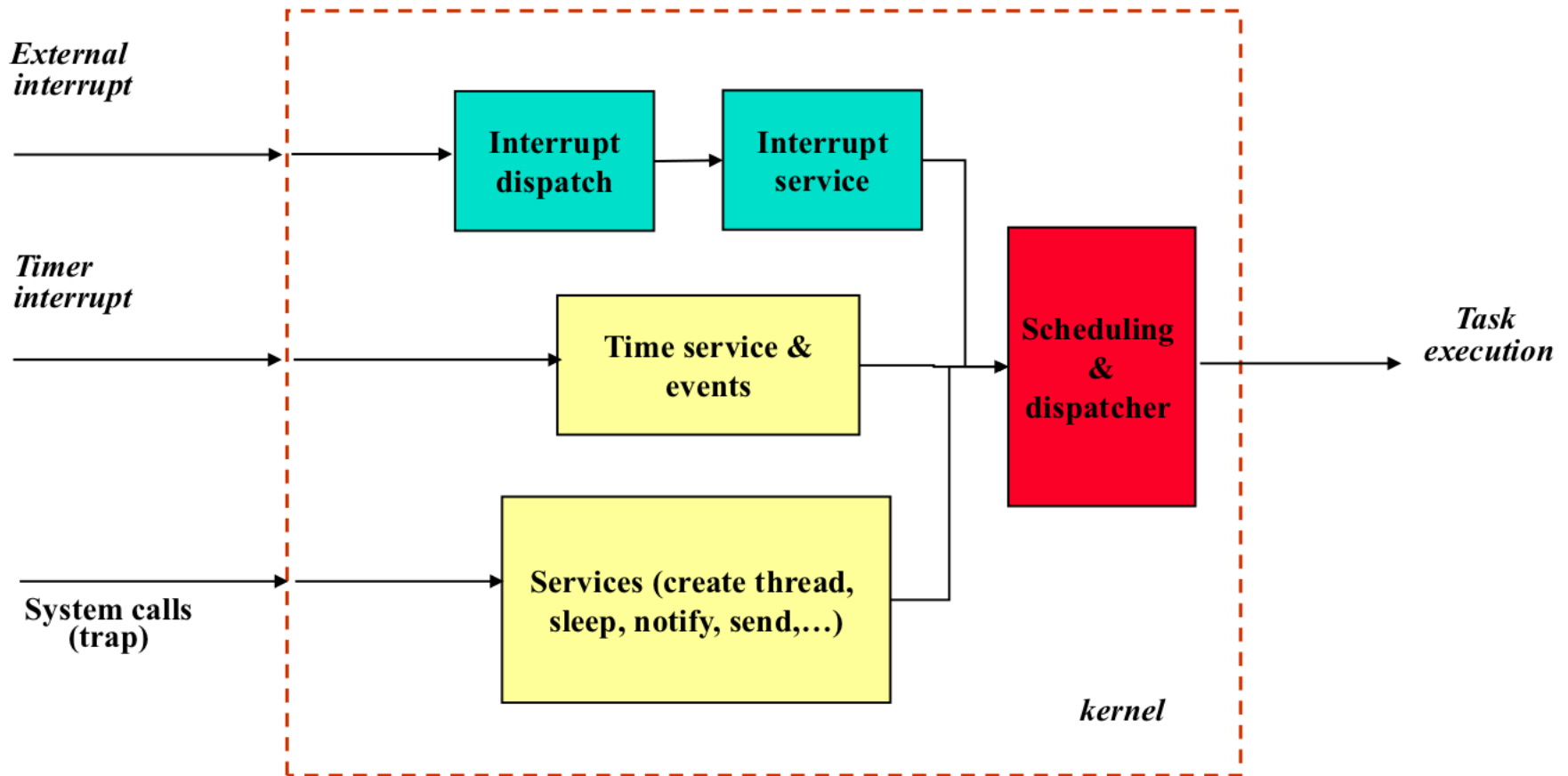


# Real-Time OS

- Definition: A real-time operating system is an operating system that supports the construction of real-time systems.
- Three key requirements:
  1. The timing behavior of the OS must be **predictable**.
    - Every services of the OS: Upper bound on the execution time!
    - RTOSs must be deterministic:
      - unlike standard Java,
      - upper bound on times during which interrupts are disabled,
      - almost all activities are controlled by scheduler.
  2. OS must manage the **timing and scheduling**
    - OS possibly has to be aware of task deadlines; (unless scheduling is done off-line).
    - OS must provide precise time services with high resolution.
  3. The OS must be **fast**
    - Practically important.



# Process Management Services

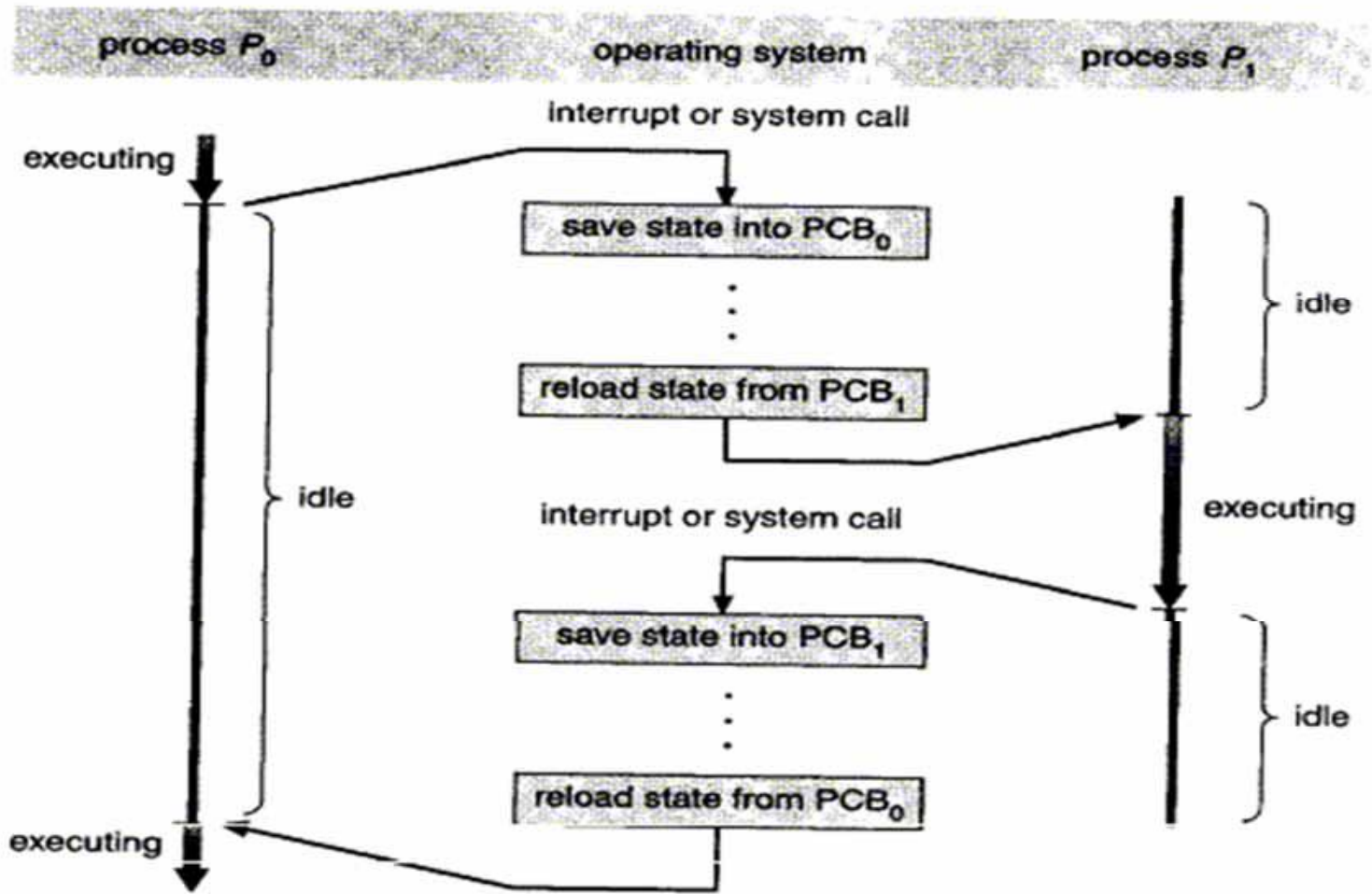


# Main Functionality of RTOS-Kernels

- Process management:
  - Execution of **quasi-parallel tasks** on a processor using processes or threads (lightweight process) by
    - maintaining process states, process queuing,
    - preemptive tasks (fast context switching) and quick interrupt handling
  - CPU **scheduling** (guaranteeing deadlines, minimizing process waiting times, fairness in granting resources such as computing power)
  - Process **synchronization** (critical sections, semaphores, monitors, mutual exclusion)
  - Inter-process **communication** (buffering)
  - Support of a **real-time clock** as an internal time reference



# Context Switching

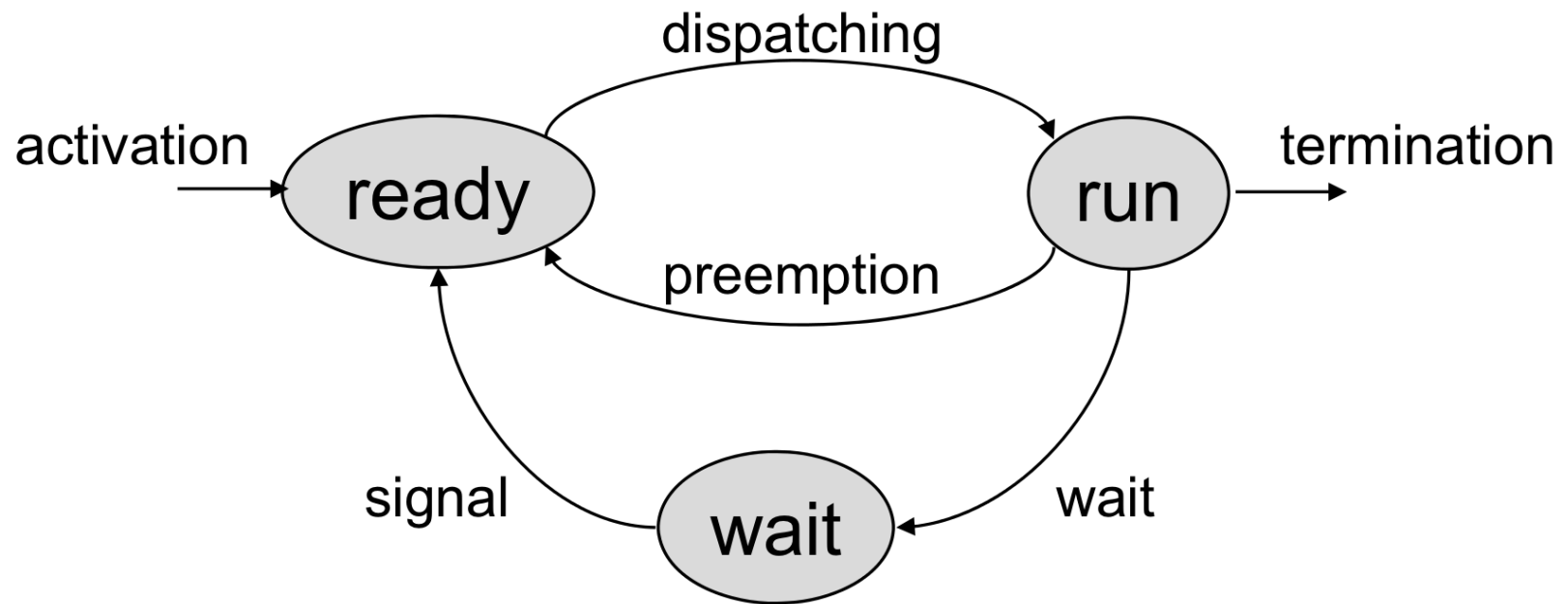


PCB: Process Control Block



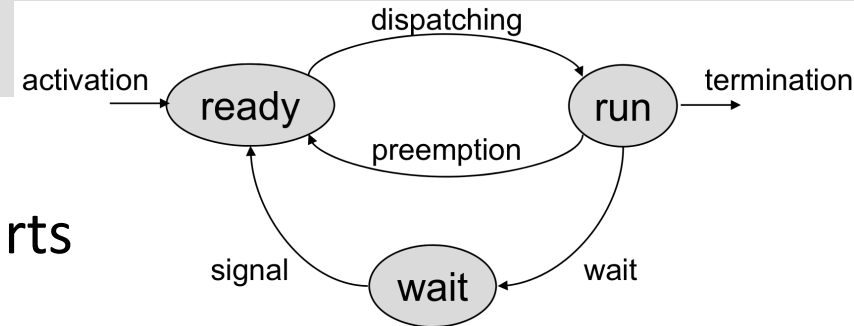
# Process States

- Minimal Set of Process States:





# Process states



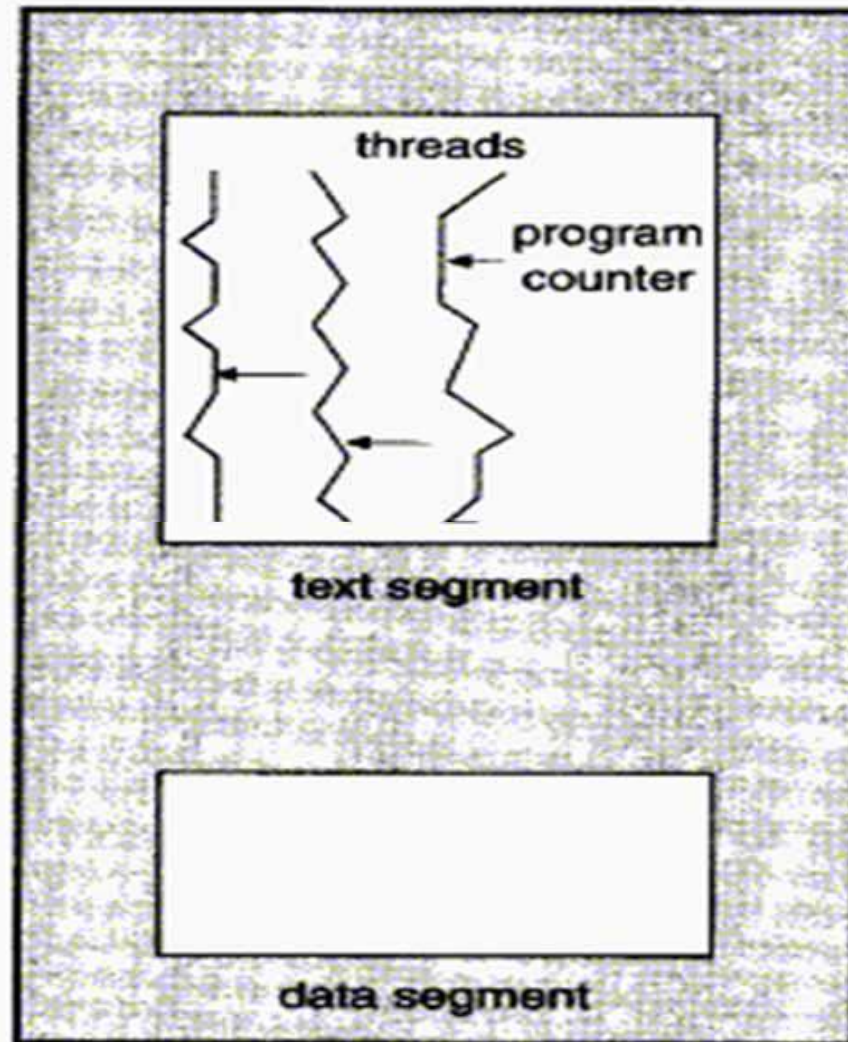
- **Run:**
  - A task enters this state as it starts executing on the processor
- **Ready:**
  - State of those tasks that are ready to execute but cannot be executed because the processor is assigned to another task.
- **Wait:**
  - A task enters this state when it executes a synchronization primitive to wait for an event, e.g., a wait primitive on a semaphore. In this case, the task is inserted in a queue associated with the semaphore. The task at the head is resumed when the semaphore is unlocked by a signal primitive. (Idle state can be a substate of wait)

# Threads

- A **thread** is an execution stream within the context of a thread state; e.g., a thread is a basic unit of CPU utilization.
- The key difference between **processes and threads**: multiple threads share parts of their state.
  - Typically **shared**: memory.
  - Typically **owned**: registers, stack.
- **Thread** advantages and characteristics
  - Faster to switch between threads; switching between user-level threads requires no major intervention by operating system.
  - Typically, an application will have a separate thread for each distinct activity.
  - Thread Control Block (TCB) stores information needed to manage and schedule a thread
    - E.g., pointer to the PCB



# Multiple Threads within a Process



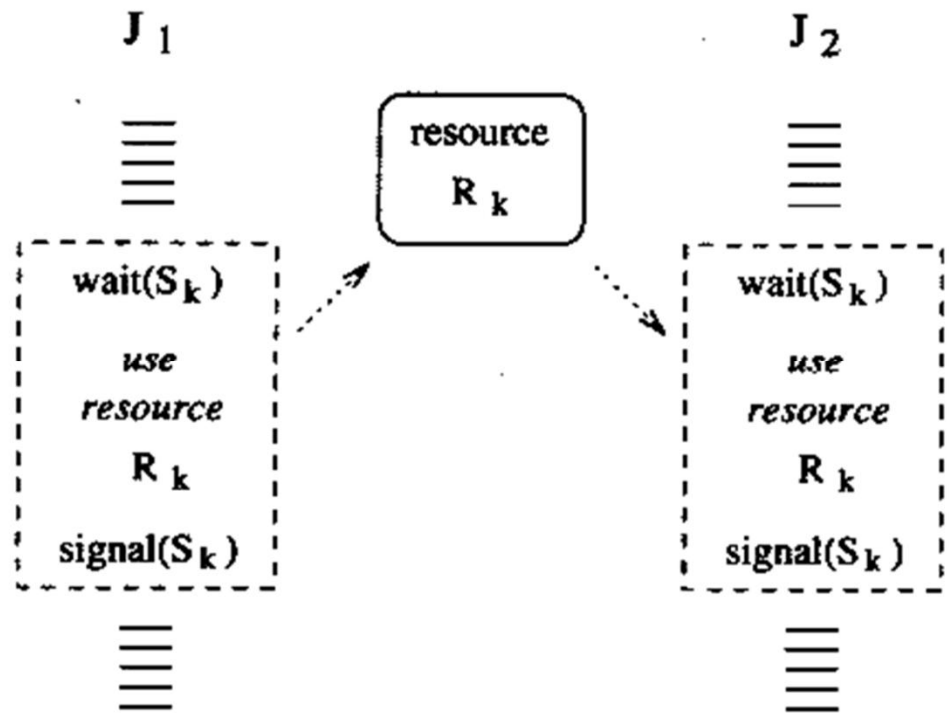
# Process Management

- Process synchronization:
  - In classical operating systems, synchronization and mutual exclusion is performed via semaphores and monitors.
  - In real-time OS, special semaphores and a deep integration into scheduling is necessary (priority inheritance protocols, ....).
- Further responsibilities:
  - Initializations of internal data structures (tables, queues, task description blocks, semaphores, ...)



# Communication Mechanisms

- Problem: the use of shared resources for implementing message passing schemes may cause priority inversion and blocking.



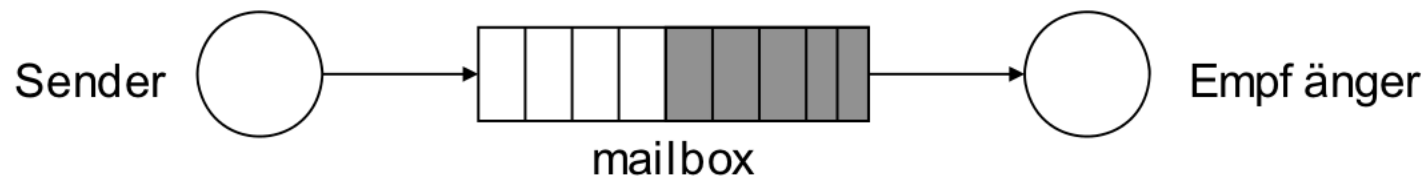
# Communication mechanisms

- Synchronous communication:
  - Whenever two tasks want to communicate they must be synchronized for a message transfer to take place (**rendezvous**)
  - They have to wait for each other.
  - **Problem** in case of dynamic real-time systems: Estimating the maximum blocking time for a process rendezvous.
  - In a **static** real-time environment, the problem can be solved off-line by transforming all synchronous interactions into precedence constraints.

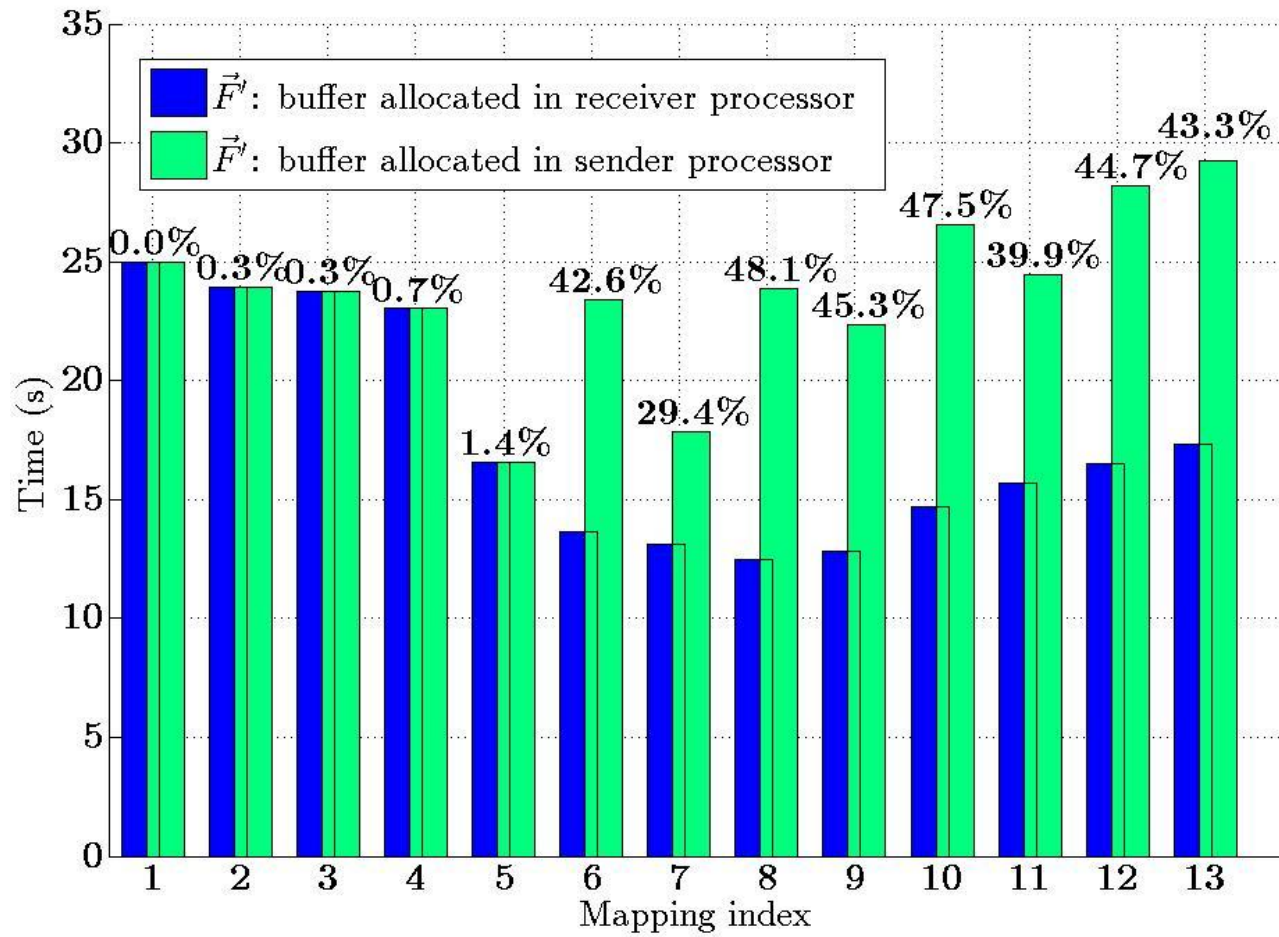


# Communication mechanisms

- Asynchronous communication:
  - Tasks do not have to wait for each other
  - The sender just deposits its message into a channel and continues its execution; similarly the receiver can directly access the message if at least a message has been deposited into the channel.
  - More suited for real-time systems than synchronous comm.
  - **Mailbox**: Shared memory buffer, FIFO-queue, basic operations are send and receive, usually has fixed capacity.
  - **Problem**: Blocking behavior if channel is full or empty; alternative approach is provided by cyclical asynchronous buffers.



# Impact of Buffer Allocation





# Class 1: Fast Proprietary Kernels

- Fast proprietary kernels

*For hard real-time systems, these kernels are questionable, because they are designed to be fast, rather than to be predictable in every respect*

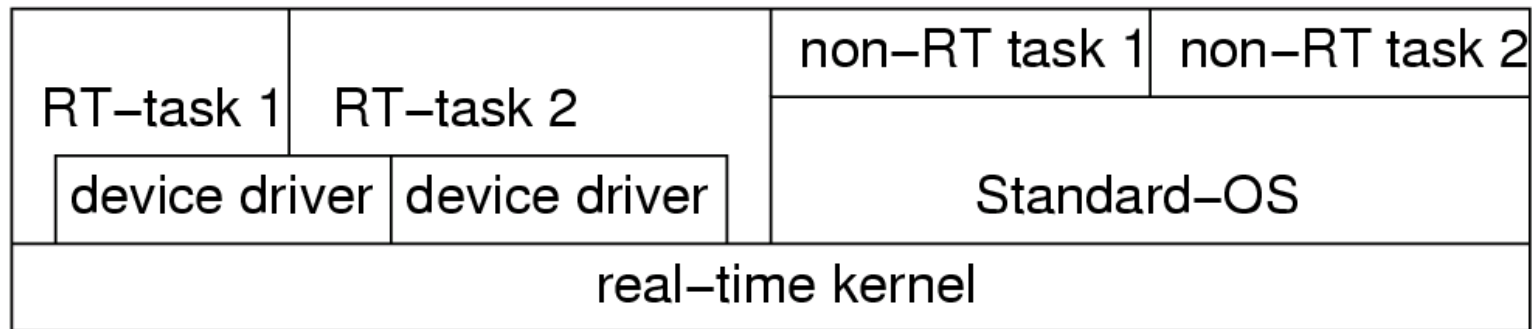
- Examples include

- QNX, PDOS, VCOS, VTRX32, VxWORKS.



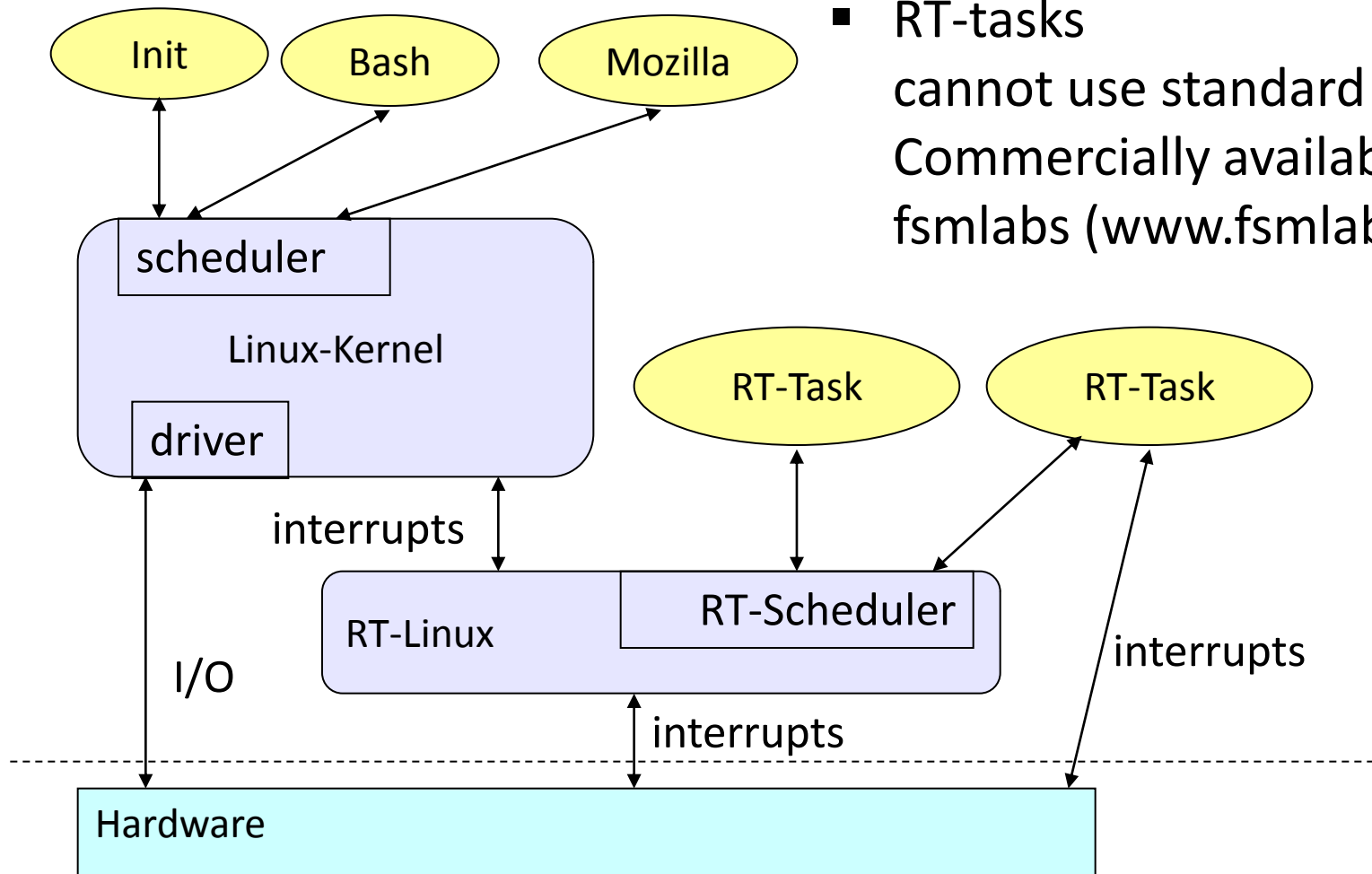
# Class 2: Extensions to Standard OSs

- Real-time extensions to standard OS:
  - Attempt to exploit comfortable main stream OS.
  - RT-kernel running all RT-tasks.
  - Standard-OS executed as one task.



- + Crash of standard-OS does not affect RT-tasks;
- RT-tasks cannot use Standard-OS services;  
less comfortable than expected

# Example RT Linux



- RT-tasks cannot use standard OS calls. Commercially available from fsm labs ([www.fsmlabs.com](http://www.fsmlabs.com))

# Class 3: Research Systems

- Research systems trying to avoid limitations:
  - Include MARS, Spring, MARUTI, Arts, Hartos, DARK, and Melody
  - More latest: FreeRTOS, RTEMS, PikeOS, OKL4, uC/OS-II ...
- Research issues:
  - low overhead memory protection,
  - temporal protection of computing resources
  - RTOSes for on-chip multiprocessors
  - quality of service (QoS) control (besides real-time constraints).



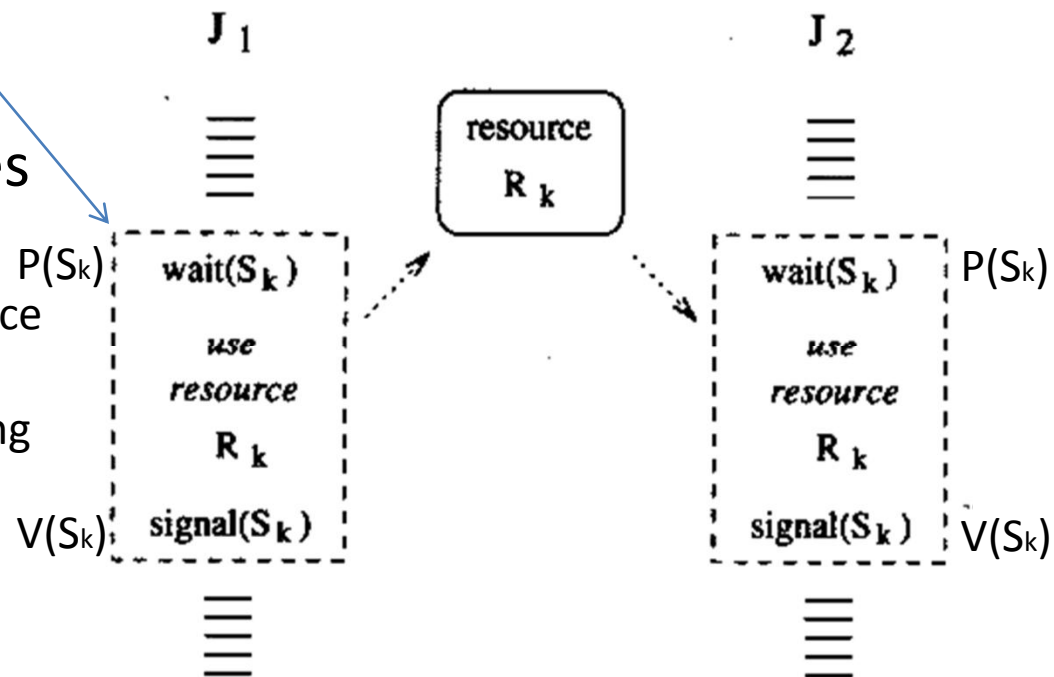
# Outline

- Embedded Operating System
- Resource Access Protocols



# Resource Sharing

- **Examples** of common resources: data structures, variables, main memory area, file, set of registers, I/O unit, ... .
- Many shared resources do not allow simultaneous accesses but require **mutual exclusion** (exclusive resources). A piece of code executed under mutual exclusion constraints is called a **critical section**.
- Can be guaranteed with semaphores  $S$  or mutexes

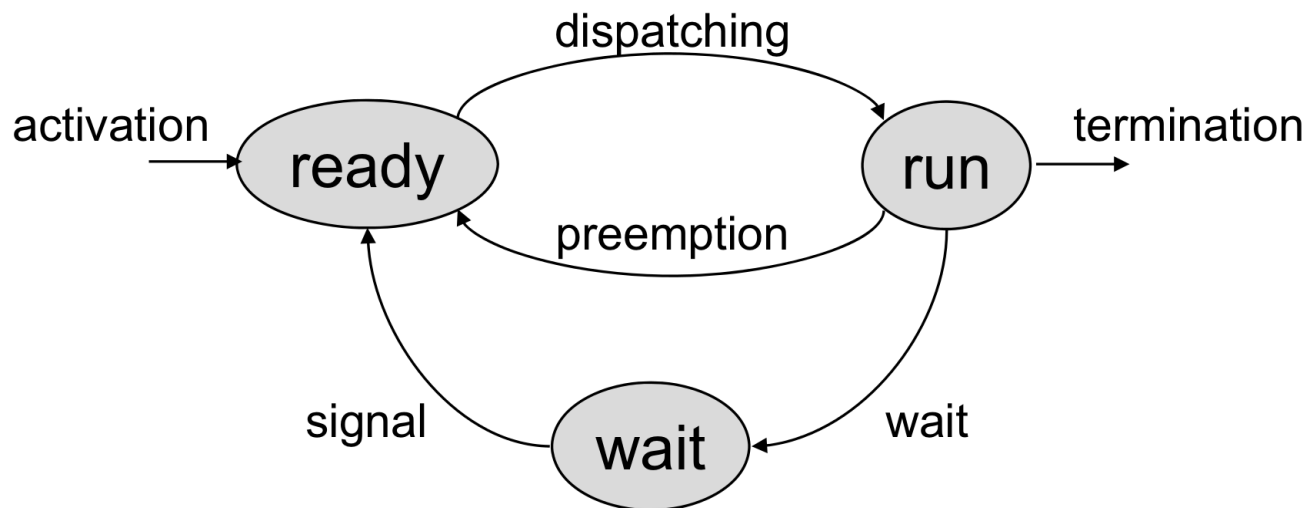


$P(S)$  checks semaphore to see if resource is available and if yes, sets  $S$  to "used". Uninterruptible operations! If no, calling task has to wait.

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

# Terms

- A task waiting for an exclusive resource is said to be **blocked** on that resource. Otherwise, it proceeds by entering the **critical section** and **holds** the resource. When a task leaves a critical section, the associated resource becomes **free**.
- Waiting state caused by resource constraints:



# Terms

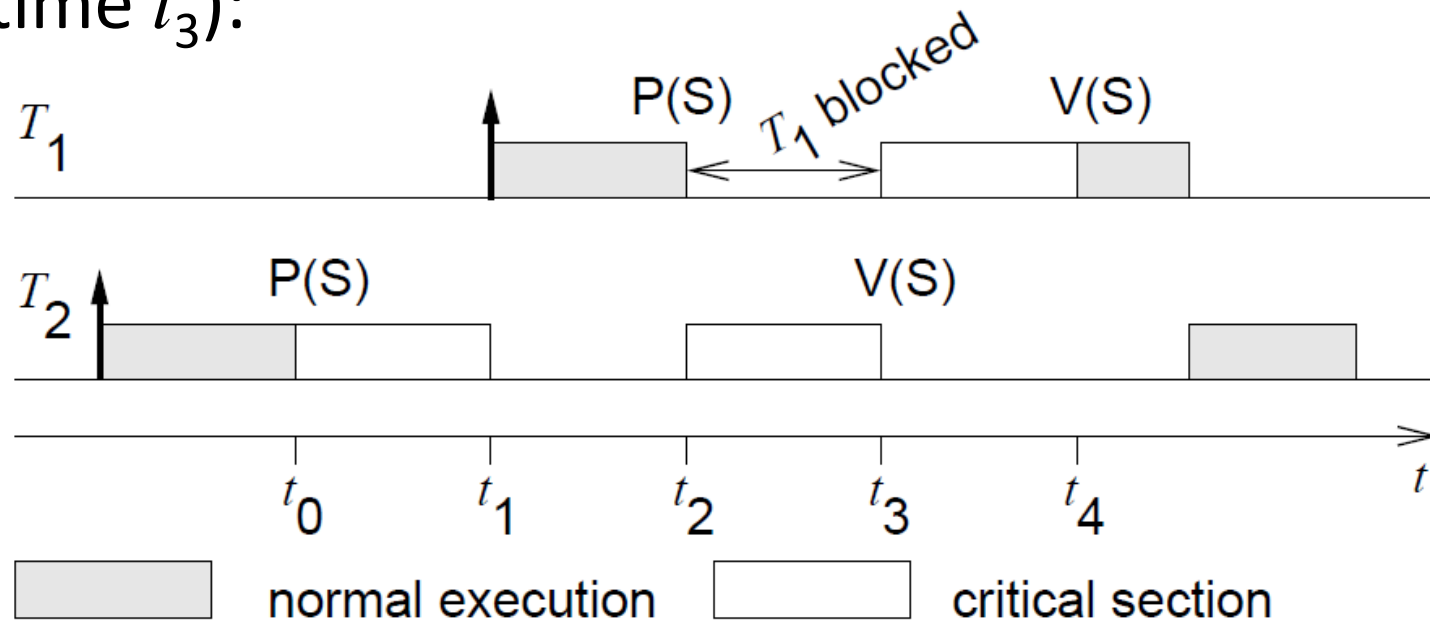
- Each **exclusive resource**  $R_i$  must be protected by a different **semaphore**  $S_i$  and each critical section operating on a resource must begin with a  $\text{wait}(S_i)$  primitive and end with a  $\text{signal}(S_i)$  primitive.
- All tasks blocked on the same resource are kept in a queue associated with the semaphore. When a running task executes a **wait** on a **locked semaphore**, it enters a **waiting state**, until another task executes a **signal** primitive that **unlocks the semaphore**.





# Priority Inversion (1)

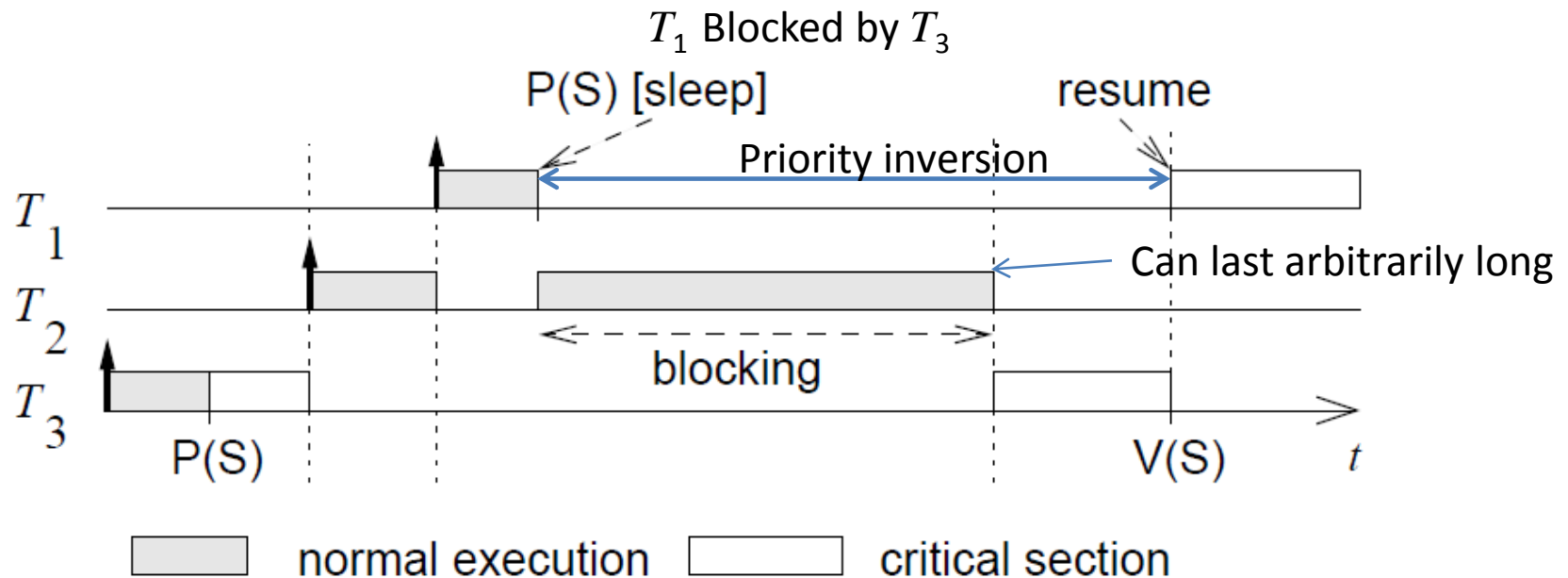
- Priority  $T_1$  assumed to be  $>$  than priority of  $T_2$ .
- If  $T_2$  requests exclusive access first (at  $t_0$ ),  $T_1$  has to wait until  $T_2$  releases the resource (at time  $t_3$ ):



For 2 tasks: blocking is bounded by the length of the critical section

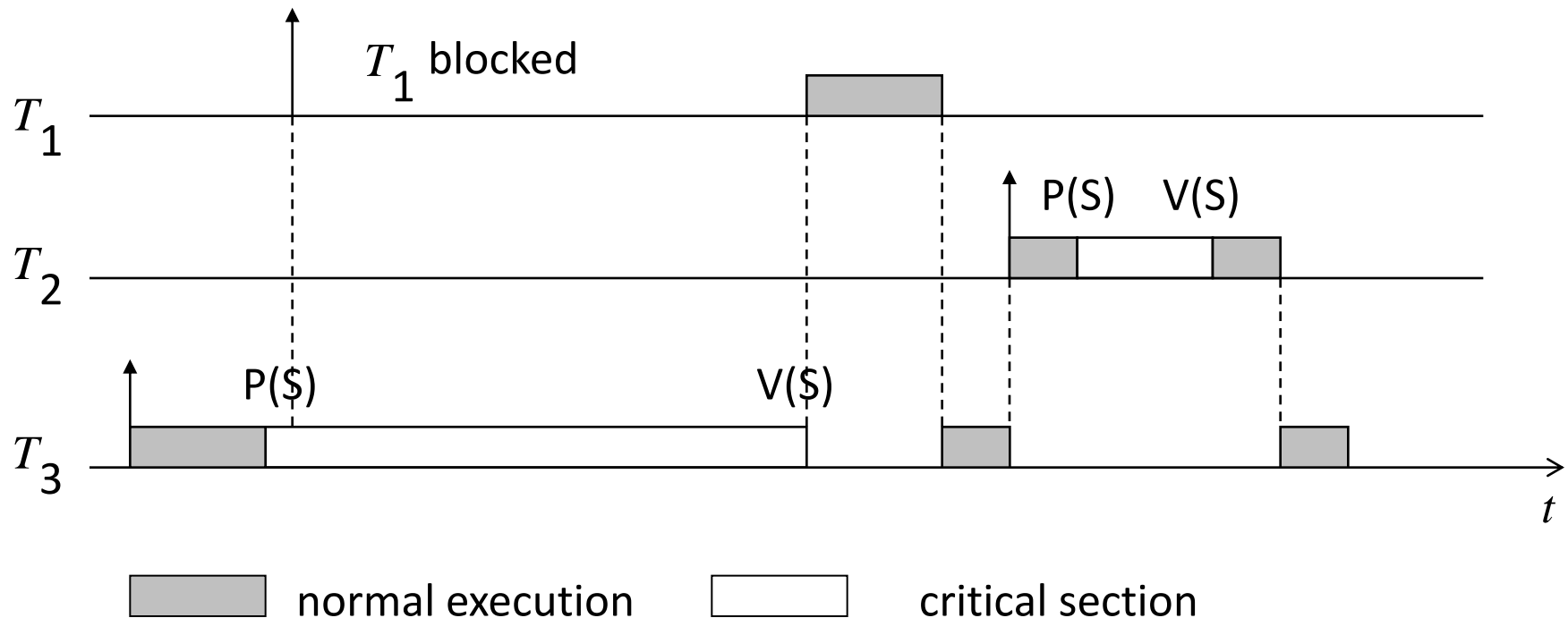
# Priority Inversion (2)

- Blocking with  $>2$  tasks can exceed the length of any critical section
  - 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.



# Solutions

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



# Resource Access Protocols

- Basic idea: Modify the priority of those tasks that cause blocking. When a task  $T_i$  blocks one or more higher priority tasks, it temporarily assumes a higher priority.
- Methods:
  - Priority Inheritance Protocol (PIP), for static priorities
  - Priority Ceiling Protocol (PCP), for static priorities
  - Stack Resource Policy (SRP),
    - For static and dynamic priorities
  - others ...

# Priority Inheritance Protocol (PIP)

- Assumptions:
  - $n$  tasks which cooperate through  $m$  shared resources; fixed priorities, all critical sections on a resource begin with a  $\text{wait}(S_i)$  and end with a  $\text{signal}(S_i)$  operation.
- Basic idea:
  - When a task  $T_i$  blocks one or more higher priority tasks, it temporarily assumes (inherits) the **highest** priority of the blocked tasks.
- Terms:
  - We distinguish a fixed **nominal priority**  $P_i$  and an **active priority**  $p_i$  larger or equal to  $P_i$ . Tasks  $T_1, \dots, T_n$  are ordered with respect to nominal priority where  $T_1$  has **highest priority**. Tasks do not suspend themselves.

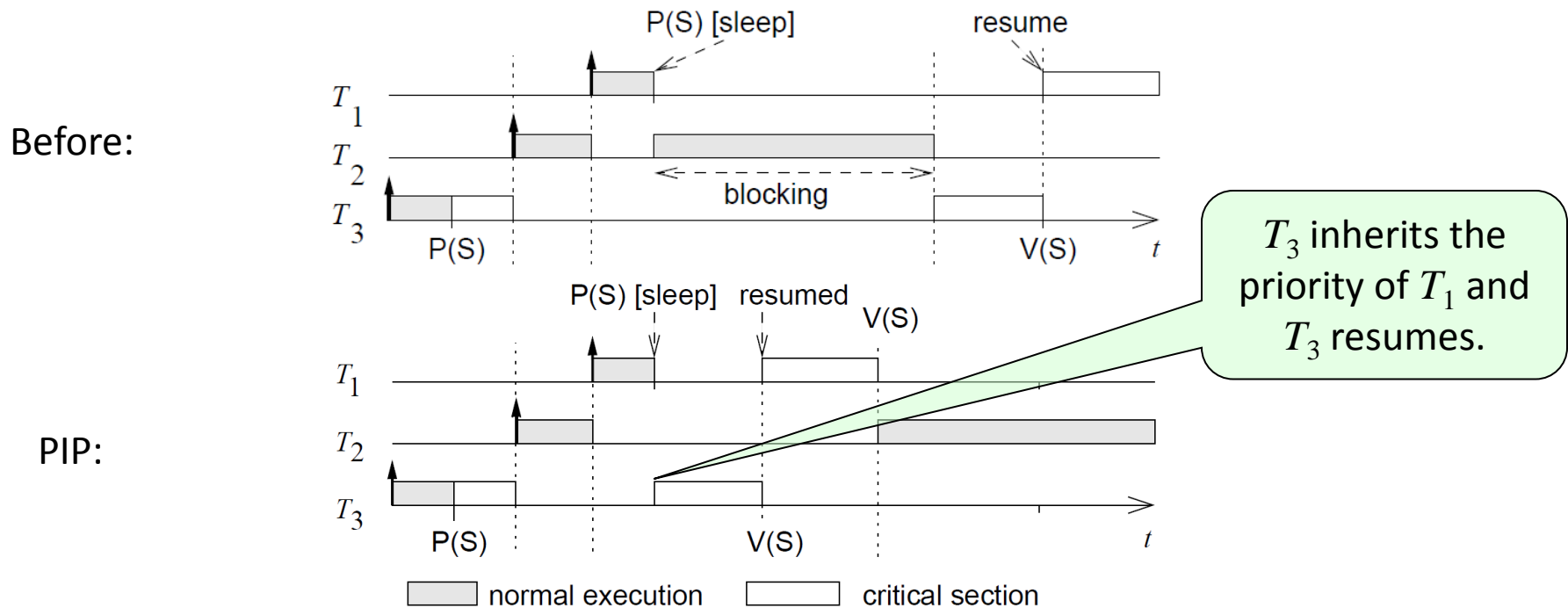
# PIP: Algorithm

- Tasks are scheduled according to their **active priorities**. Tasks with the same priorities are scheduled FCFS.
- When a task  $T_i$  tries to **enter a critical section** and the resource is blocked by a lower priority task, the task  $T_i$  is blocked. Otherwise it enters the critical section.
- When a task  $T_i$  is **blocked**, it transmits its active priority to the task  $T_k$  that holds the semaphore.  $T_k$  resumes and executes the rest of its critical section with a priority  $p_k = p_i$  (it **inherits** the priority of the highest priority of the tasks blocked by it).
- When  $T_k$  exits a critical section, it **unlocks** the semaphore and the highest priority task blocked on that semaphore is awakened. If no other tasks are blocked by  $T_k$ , then  $p_k$  is set to  $P_k$ , otherwise it is set to the highest priority of the tasks blocked by  $T_k$ .
- Priority inheritance is **transitive**, i.e., if 1 is blocked by 2 and 2 is blocked by 3, then 3 inherits the priority of 1 via 2.



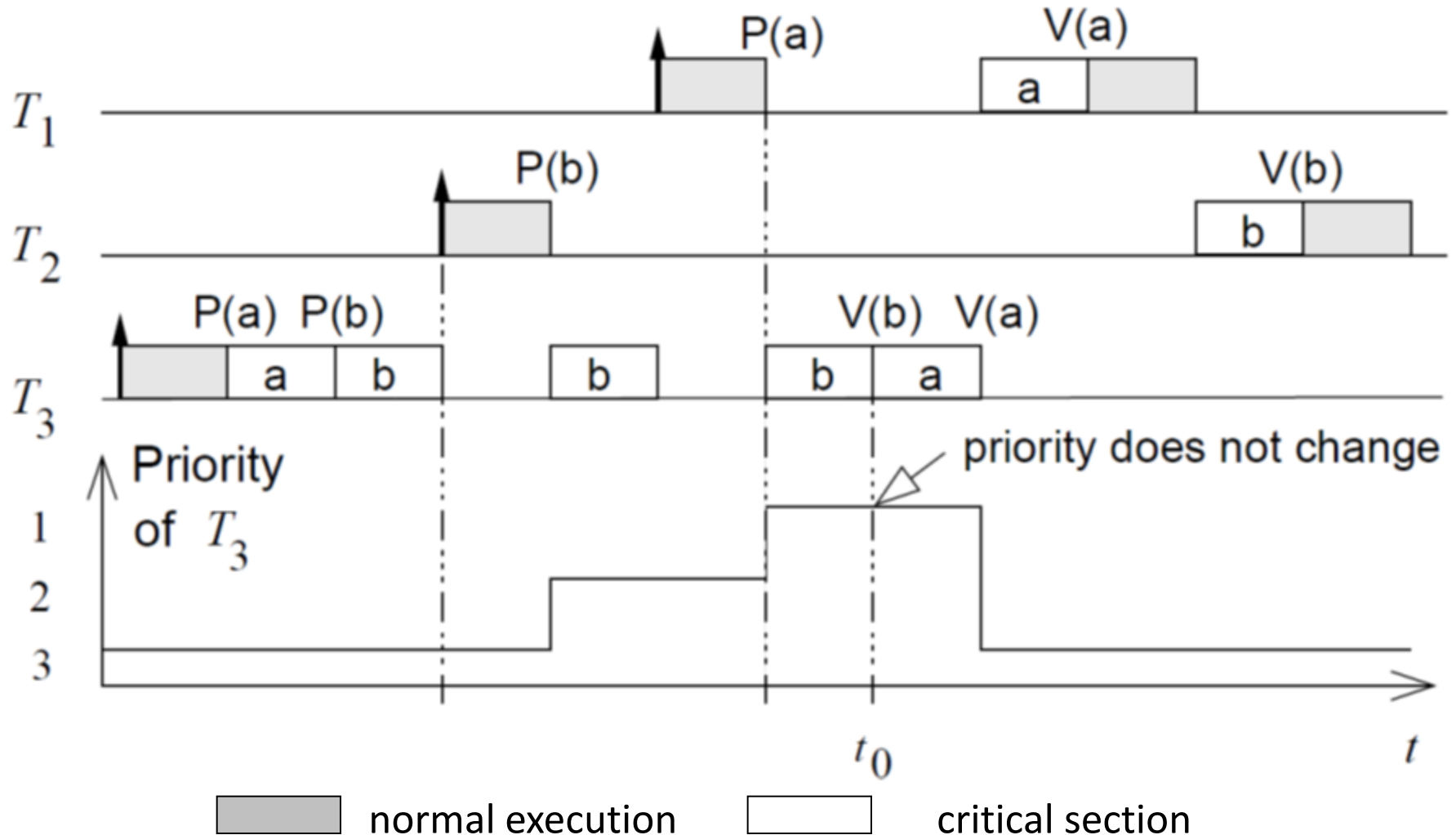
# Example

- How would priority inheritance affect our example with 3 tasks?



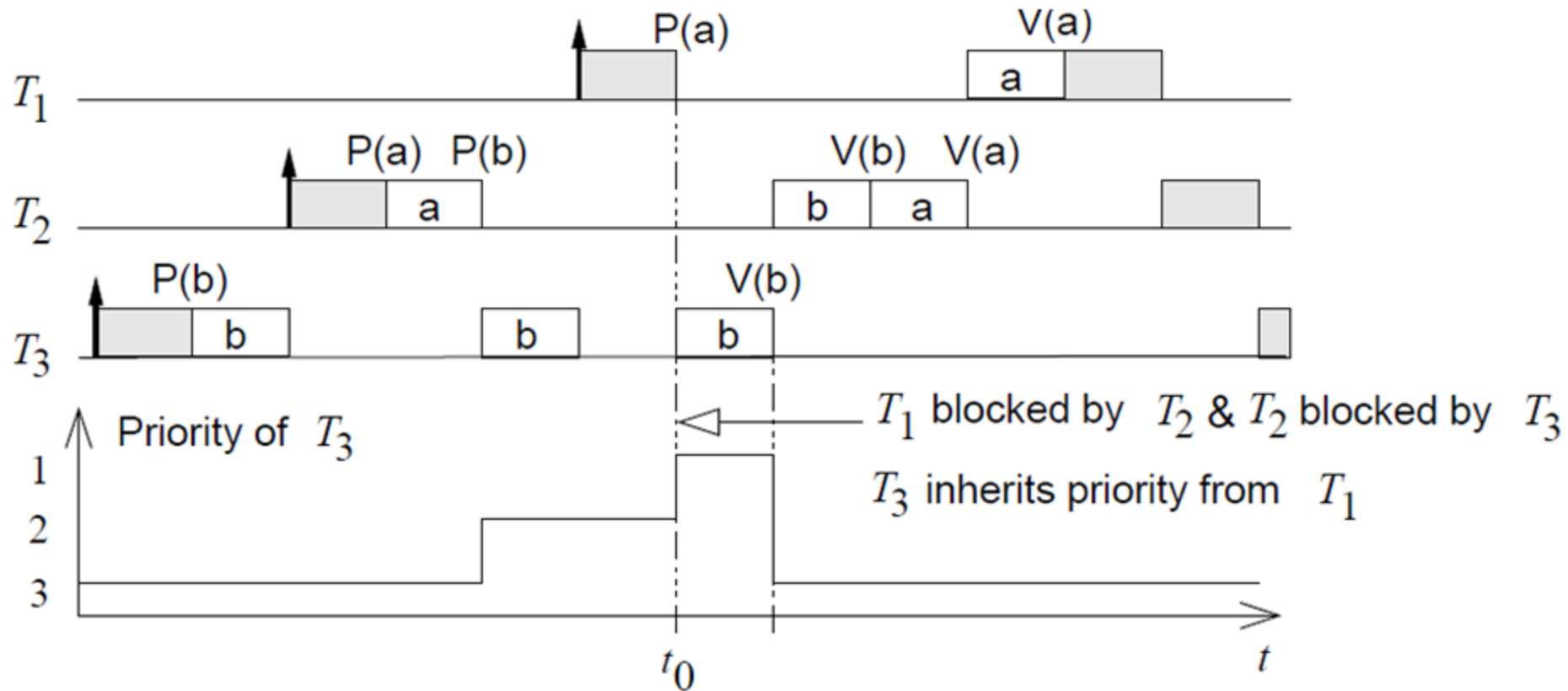
- Direct Blocking:** higher-priority task tries to acquire a resource held by a lower-priority task
- Push-through Blocking:** medium-priority task is blocked by a lower-priority. Task that has inherited a higher priority from a task it directly blocks

# Nested Critical Sections

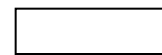




# Transitivity of Priority Inheritance

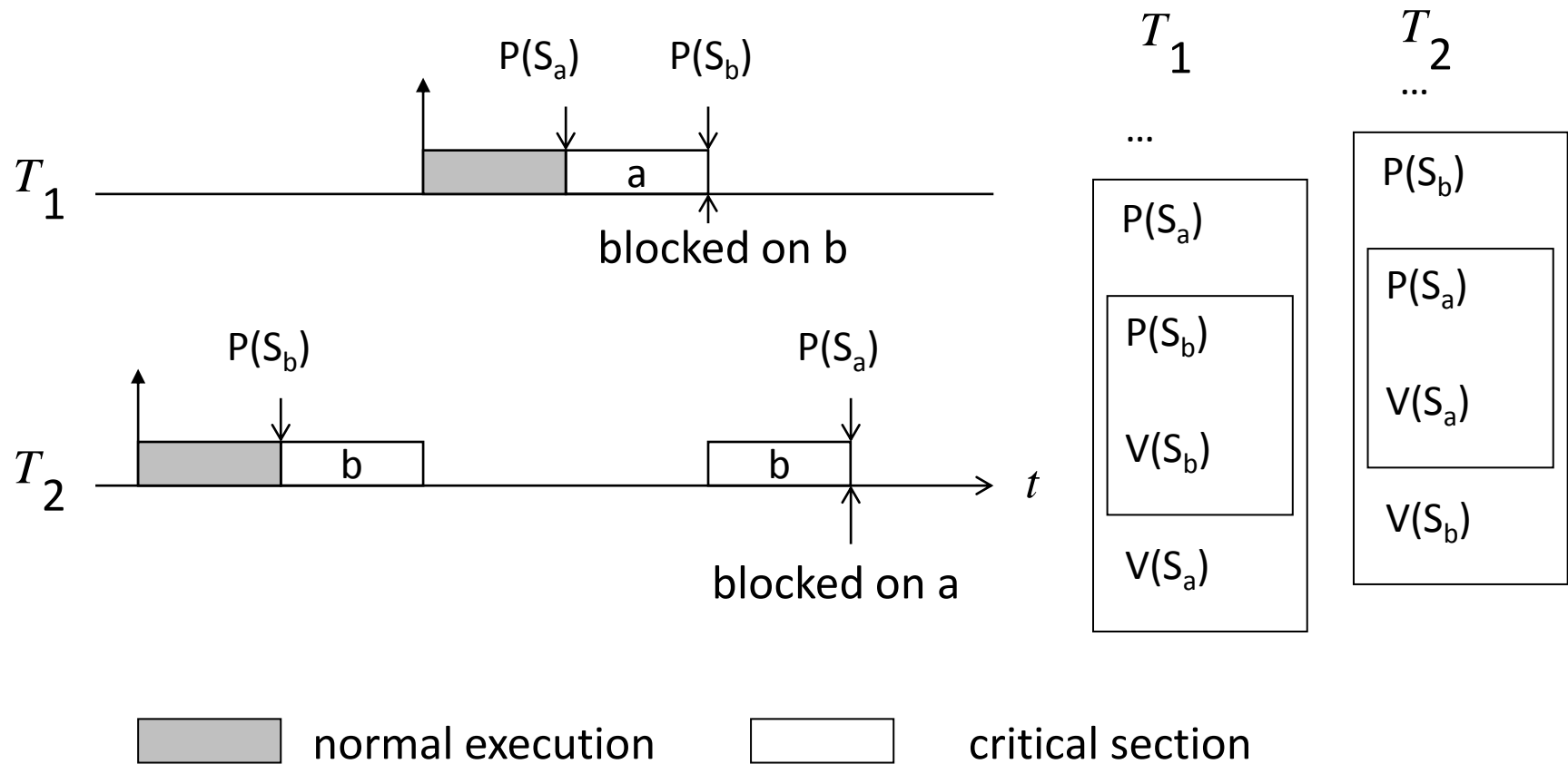


normal execution



critical section

# Deadlock is Possible

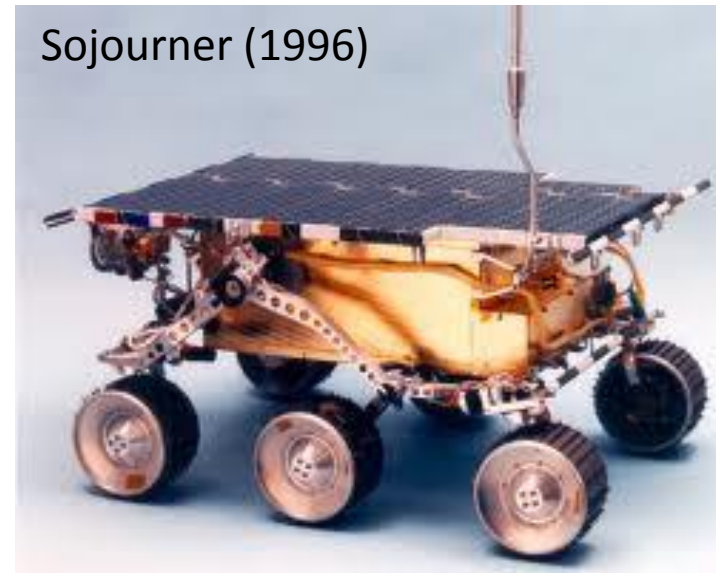


- Problem exists also when no priority inheritance is used

# The MARS Pathfinder problem (1)

- “But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once".” ...

Sojourner (1996)



[http://research.microsoft.com/~mbj/Mars\\_Pathfinder/Mars\\_Pathfinder.html](http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html)



# The MARS Pathfinder problem (2)

- “VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.”
- “Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft.”

A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).”

[http://research.microsoft.com/~mbj/Mars\\_Pathfinder/Mars\\_Pathfinder.html](http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html)



# The MARS Pathfinder problem (3)

- The meteorological data gathering task ran as an infrequent, low priority thread, ... When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. ..
- The spacecraft also contained a communications task that ran with medium priority.”



High priority: retrieval of data from shared memory

Medium priority: communications task

Low priority: thread collecting meteorological data

[http://research.microsoft.com/~mbj/Mars\\_Pathfinder/Mars\\_Pathfinder.html](http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html)



# The MARS Pathfinder problem (4)

- “... However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread.
- In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running.
- After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.”

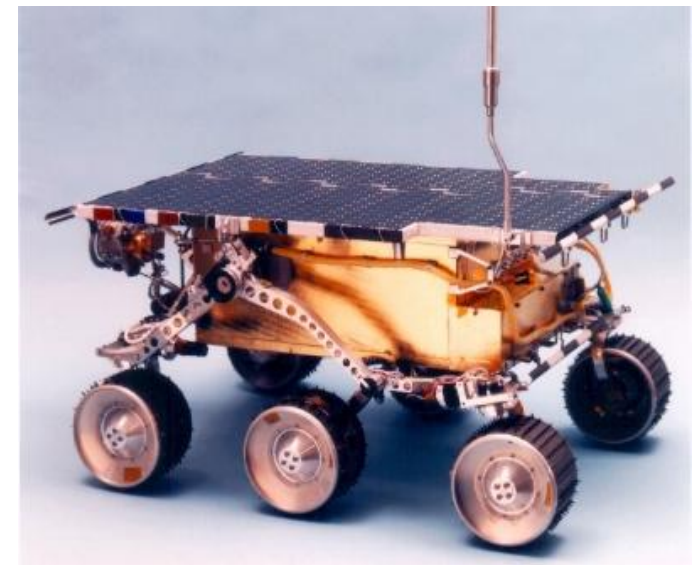
[http://research.microsoft.com/~mbj/Mars\\_Pathfinder/Mars\\_Pathfinder.html](http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html)



# Priority inversion on Mars

- Priority inheritance also solved the Mars 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”.

The problem on Mars was corrected by using the debugging facilities of VxWorks to change the flag to “on”, while the Pathfinder was already on the Mars [Jones, 1997].



# Remarks on PIP

- Possibly 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/](http://www.fsmlabs.com/resources/white_papers/priority-inheritance/)
- Finds application in ADA: During *rendez-vous*, task priority is set to the maximum.
- Protocol for fixed set of tasks: priority ceiling protocol.





# PIP → PCP

## ■ The Priority Inheritance Protocol (PIP)

- does not prevent deadlocks
- can lead to chained blocking
  - (Several lower priority tasks can block a higher priority task)
- and has inherent static priorities of tasks

## → The Priority Ceiling Protocol (PCP)

- avoids multiple blocking
- guarantees that, once a task has entered a critical section, it cannot be blocked by lower priority tasks until its completion.

Source: [http://www.ida.liu.se/~unmbo/RTS\\_CUGS\\_files/Lecture3.pdf](http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf)



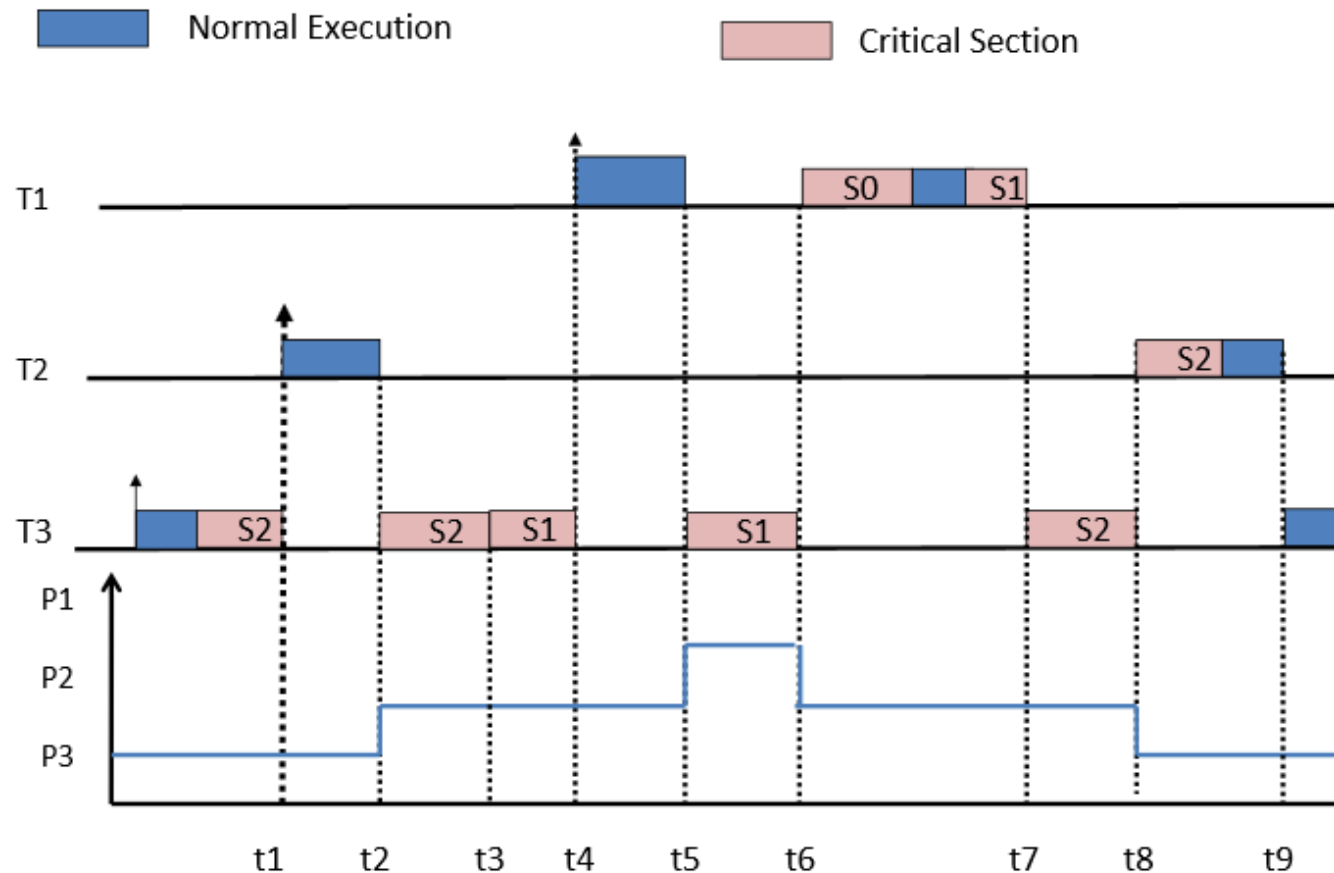
# PCP

- A task is not allowed to enter a critical section if there are already locked semaphores which could block it eventually
- Hence, once a task enters a critical section, it can not be blocked by lower priority tasks until its completion.
- This is achieved by assigning priority ceiling.
- Each semaphore  $S_k$  is assigned a **priority ceiling**  $C(S_k)$ . It is the priority of the highest priority task that can lock  $S_k$ .  
This is a static value.

Source: [http://www.ida.liu.se/~unmbo/RTS\\_CUGS\\_files/Lecture3.pdf](http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf)



# Priority Ceiling: Example



$C(S0=P1)$   $C(S1=P1)$   $C(S2=P2)$

Source: [http://www.ida.liu.se/~unmbo/RTS\\_CUGS\\_files/Lecture3.pdf](http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf)

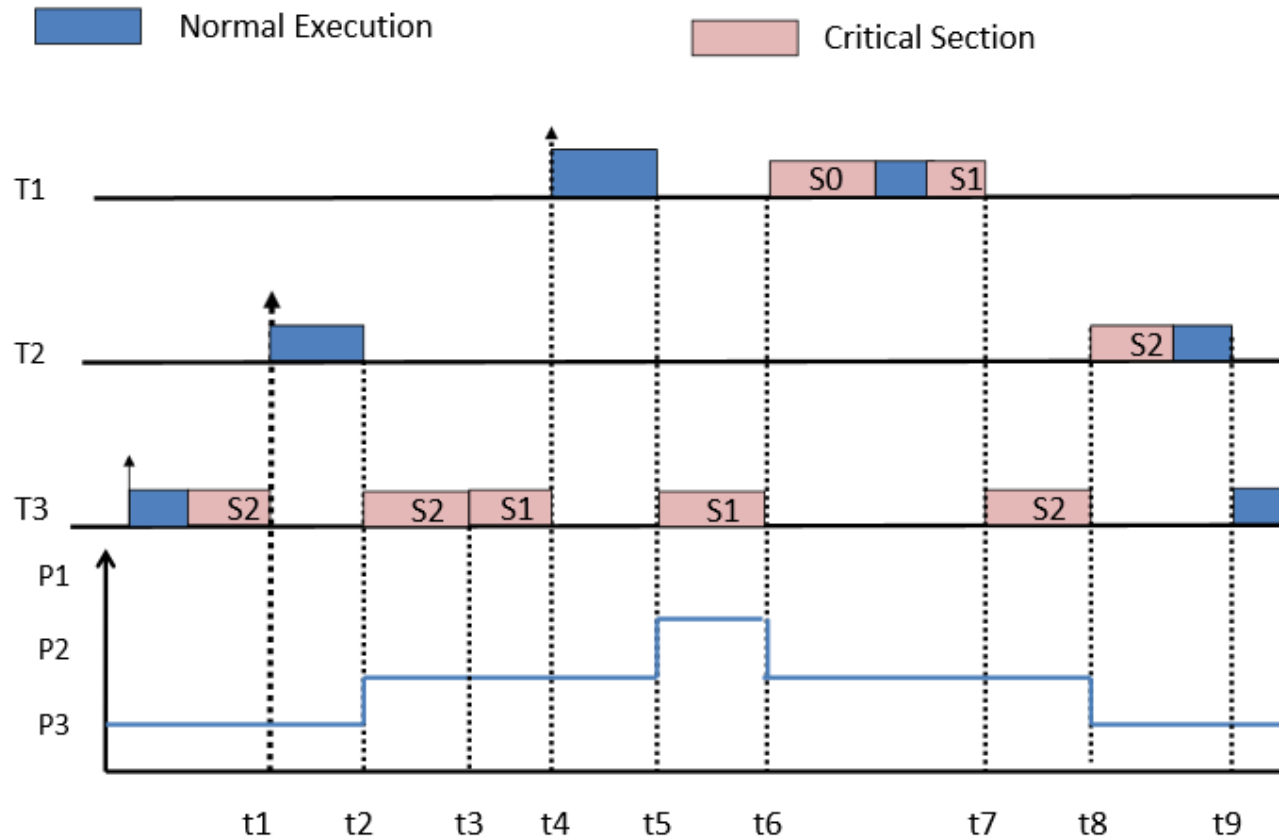


# PCP

- Suppose  $T$  is running and wants to lock semaphore  $S_k$ .
- $T$  is allowed to lock  $S_k$  only if priority of  $T >$  priority ceiling  $C(S^*)$  of the semaphore  $S^*$  where:
  - $S^*$  is the semaphore with the **highest** priority ceiling among all the semaphores which are currently locked by tasks other than  $T$ .
  - In this case,  $T$  is said to be blocked by the semaphore  $S^*$  (and the task currently holding  $S^*$ )
  - When  $T$  gets blocked by  $S^*$  then the priority of  $T$  is transmitted to the task that currently holds  $S^*$



# PCP: An Example

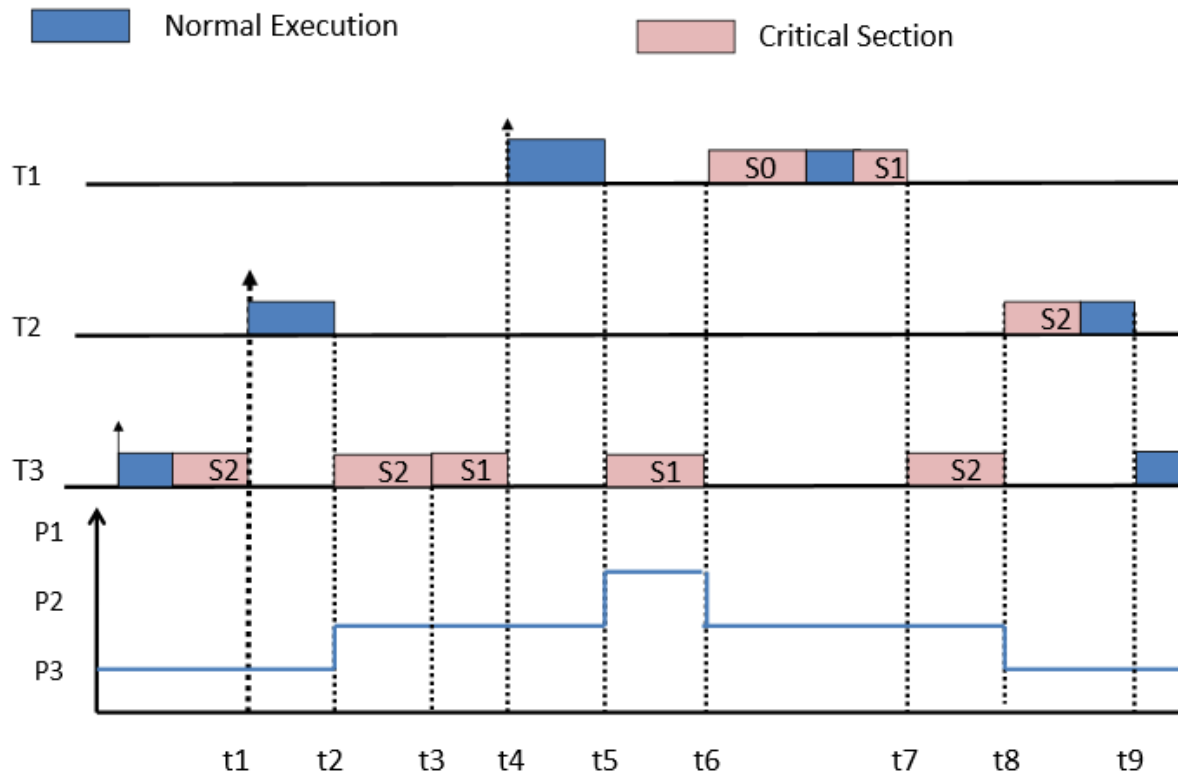


t2: T2 can not lock S2. Currently T3 is holding S2 and  $C(S2) = P2$  and the current priority of T2 is also P2

Source: [http://www.ida.liu.se/~unmbo/RTS\\_CUGS\\_files/Lecture3.pdf](http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf)



# PCP: An Example



t5 : T1 can not lock S0. Currently T3 is holding S2 and S1 and  $C(S1) = T1$  and the current priority of T1 is also P1. The (inherited) priority of T3 is now P1

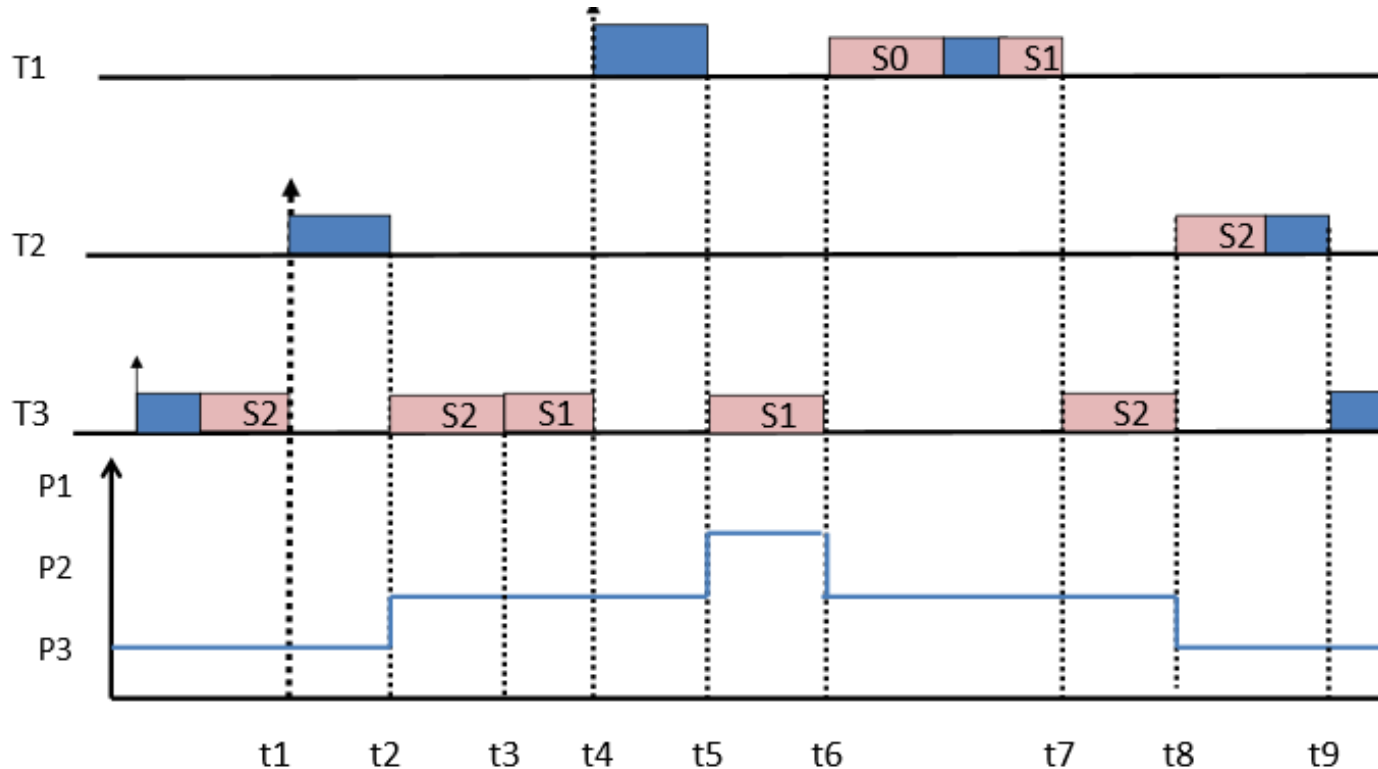
# PCP

- When  $T^*$  leaves a critical section guarded by  $S^*$  then it unlocks  $S^*$  and the highest priority task, if any, which is blocked by  $S^*$  is awakened
- The priority of  $T^*$  is set to the highest priority of the task that is blocked by some semaphore that  $T^*$  is still holding.  
If none, the priority of  $T^*$  is set to be its nominal one.

Source: [http://www.ida.liu.se/~unmbo/RTS\\_CUGS\\_files/Lecture3.pdf](http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf)



# PCP: Example



Source: [http://www.ida.liu.se/~unmbo/RTS\\_CUGS\\_files/Lecture3.pdf](http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf)

$t_6$  : T3 unlocks S1. It awakens T1. But T3s (inherited) priority is now only P2 while  $P1 > C(S2) = P2$ . So T1 preempts T3 and runs to completion.

$t_7$ : T3 resumes execution with priority P2

$t_8$  : T3 unlocks S2, goes back to its priority P3. T2 preempts T3, runs to completion

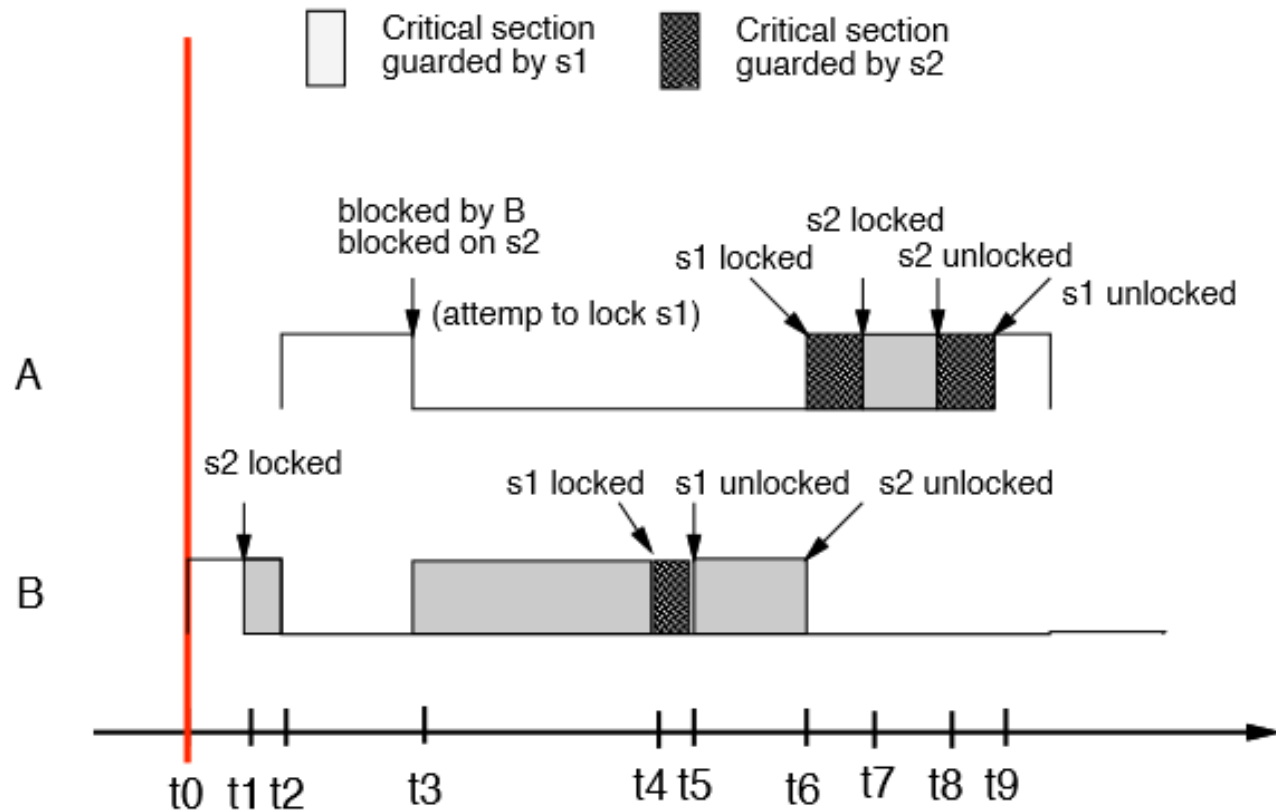




# PCP: Deadlock-Free Example

Task name	T	Priority
A	50	10
B	500	9

Task A	Task B
lock(s1)	lock(s2)
lock(s2)	lock(s1)
...	...
unlock(s1)	unlock(s1)
unlock(s2)	unlock(s1)



$$ceil(s_1) = 10, ceil(s_2) = 10$$

Source: Lund University, course EDA 040, <http://fileadmin.cs.lth.se/cs/Education/EDA040/lecture/RTP-F6b.pdf>



# PCP: Properties

- Deadlock free (only changing priorities)
- A given task  $i$  is delayed at most once by a lower priority task
- The delay is a function of the time taken to execute the critical section
- Certain variants as to when the priority is changed



# Extending PCP: Stack Resource Policy (SRP)

- SRP supports **dynamic priority scheduling**
- SRP blocks the task at the time it **attempts to preempt**.
- **Preemption level**  $l_i$  of task  $i$ : decreasing function of deadline (larger deadline  $\rightarrow$  easier to preempt) (Static)
- **Resource ceiling**: of a resource is the highest preemption level from among all tasks that may access that resource (Static)
- **System ceiling**: is the highest resource ceiling of all the resources which are currently blocked (dynamic, changes with resource accesses)

Source: [http://www.ida.liu.se/~unmbo/RTS\\_CUGS\\_files/Lecture3.pdf](http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf)



# SRP Policy

- A task can preempt another task if
  - it has the highest priority
  - and its preemption level is higher than the system ceiling
- A task is not allowed to start until the resources currently available are sufficient to meet the maximum requirement of every task that could preempt it.
- Why **Stack** Resource Policy? Tasks cannot be blocked by tasks with lower  $l_i$ , can resume only when the task completes. Tasks on the same  $l_i$  can share stack space. More tasks on the same  $l_i \rightarrow$  higher stack space saving.

Source: [http://www.ida.liu.se/~unmbo/RTS\\_CUGS\\_files/Lecture3.pdf](http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf)

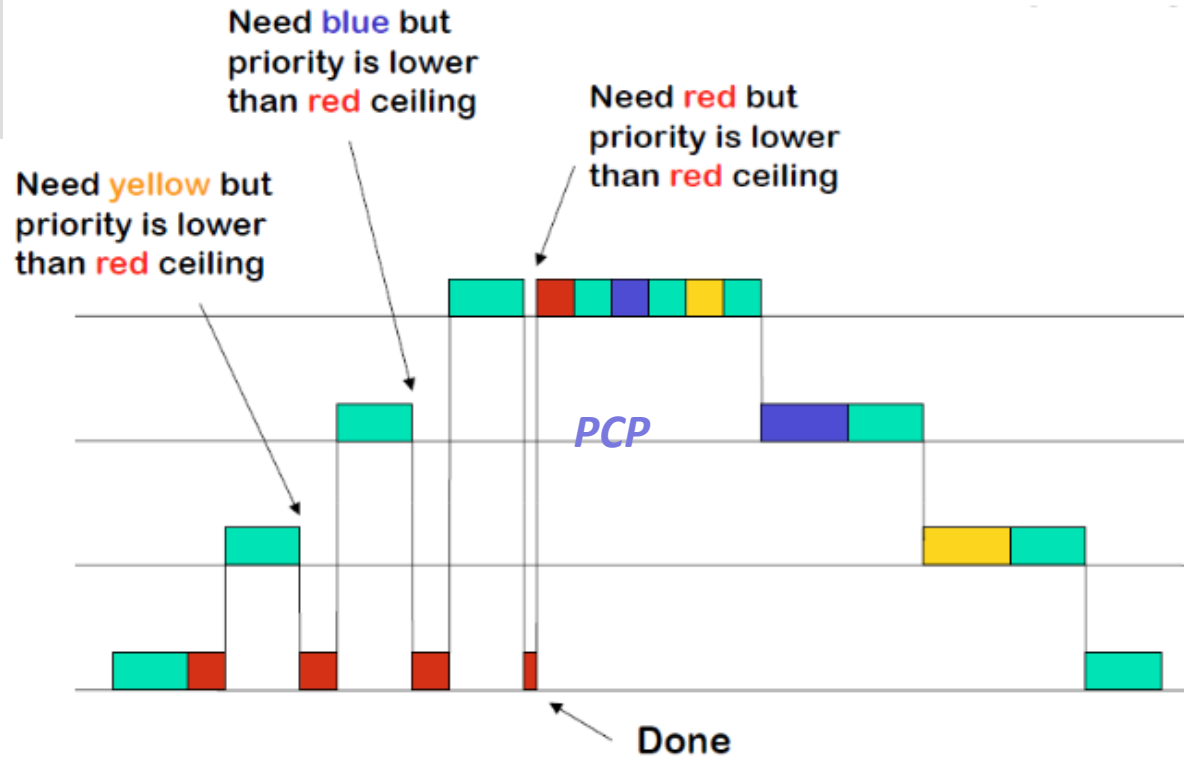
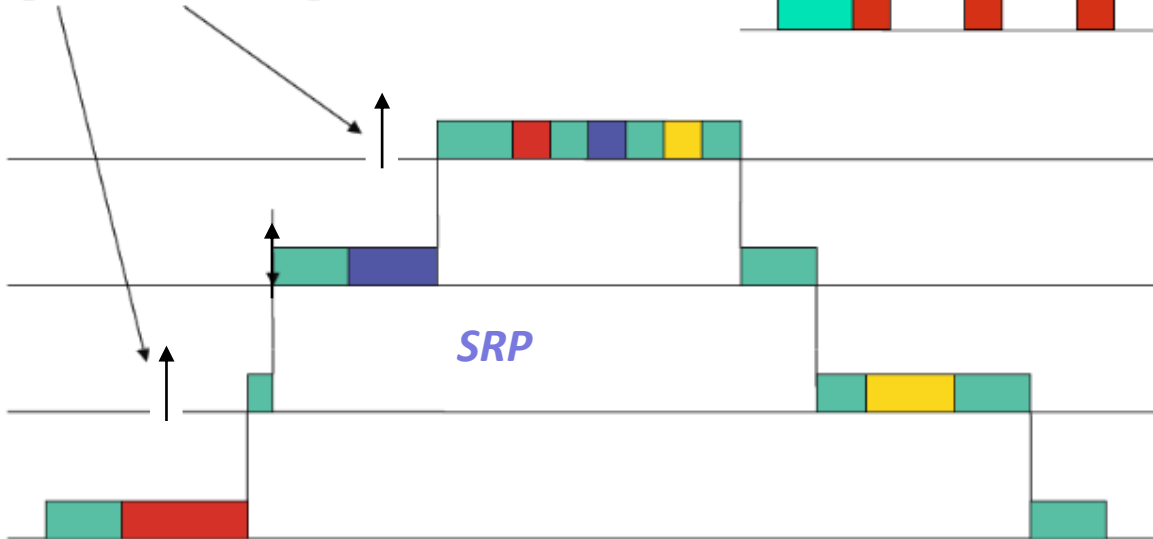


# SRP vs. PCP

■ a

Less preemptions for SRP

Can't preempt.  
Preemption level is not higher than ceiling.



Source: [http://www.ida.liu.se/~unmbo/RTS\\_CUGS\\_files/Lecture3.pdf](http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf)

