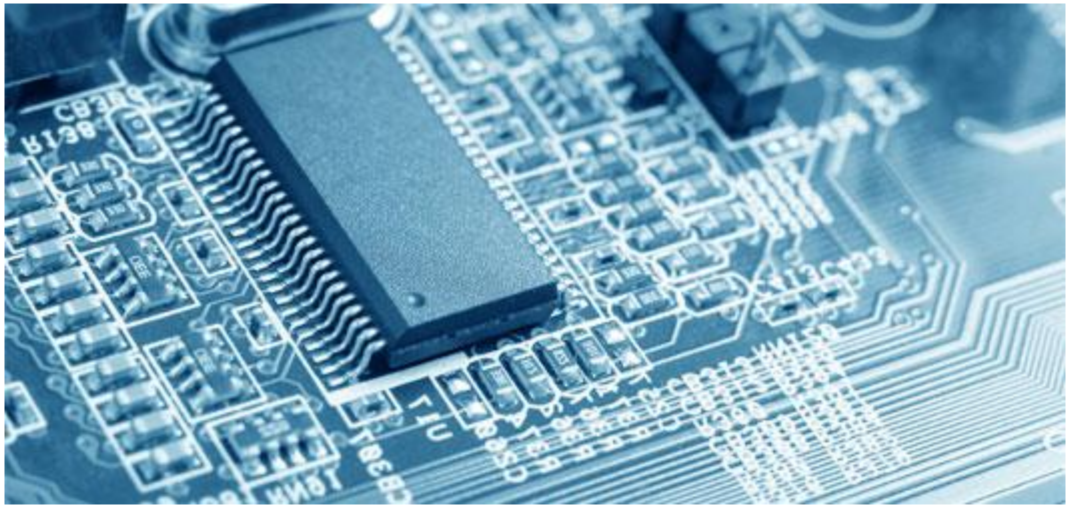
NAZARBAYEV UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY



ROBT305 - Embedded Systems

Lecture 12 – Deadlocks, Priority Inversion

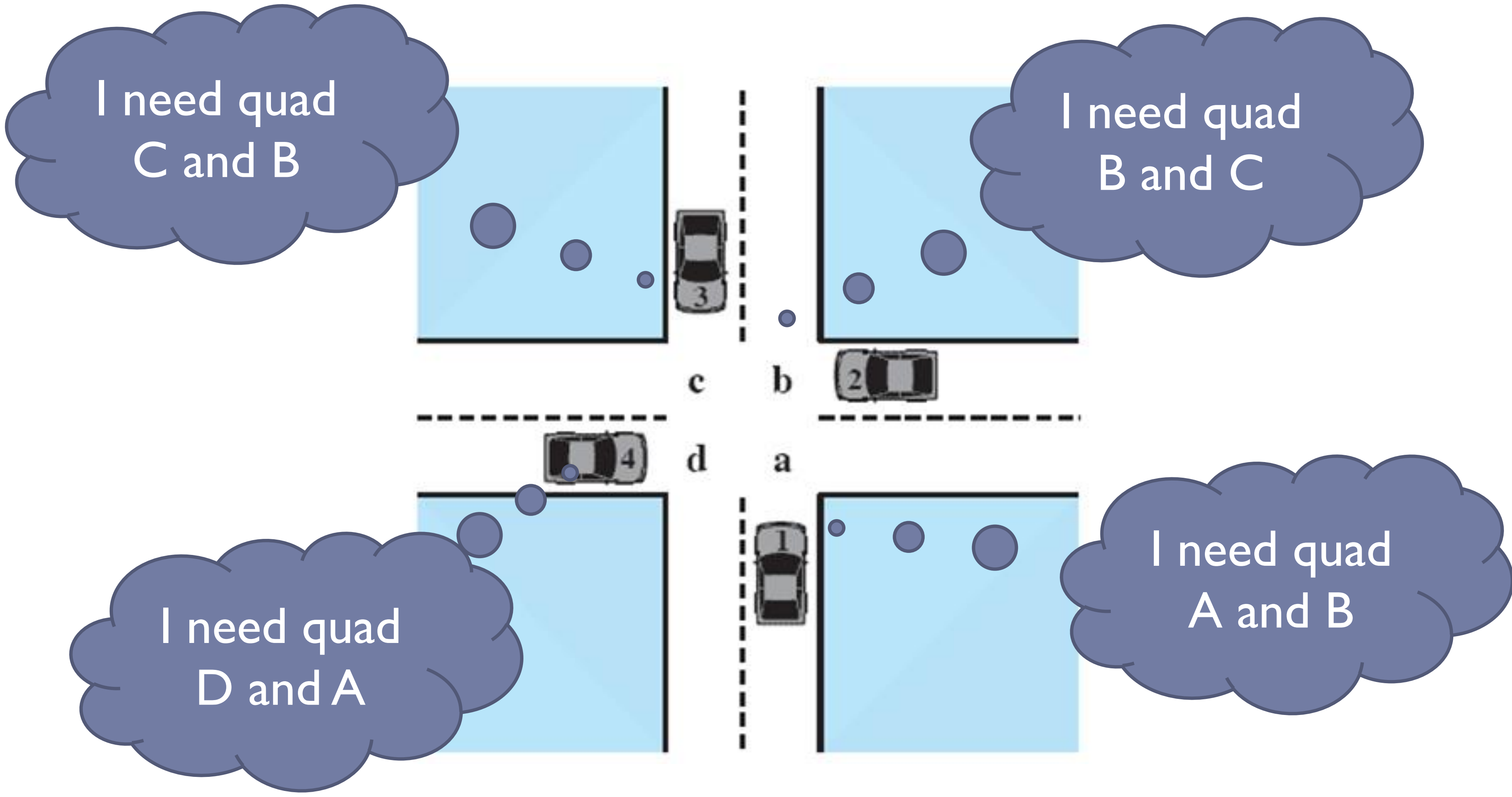
20 October, 2015

Deadlock

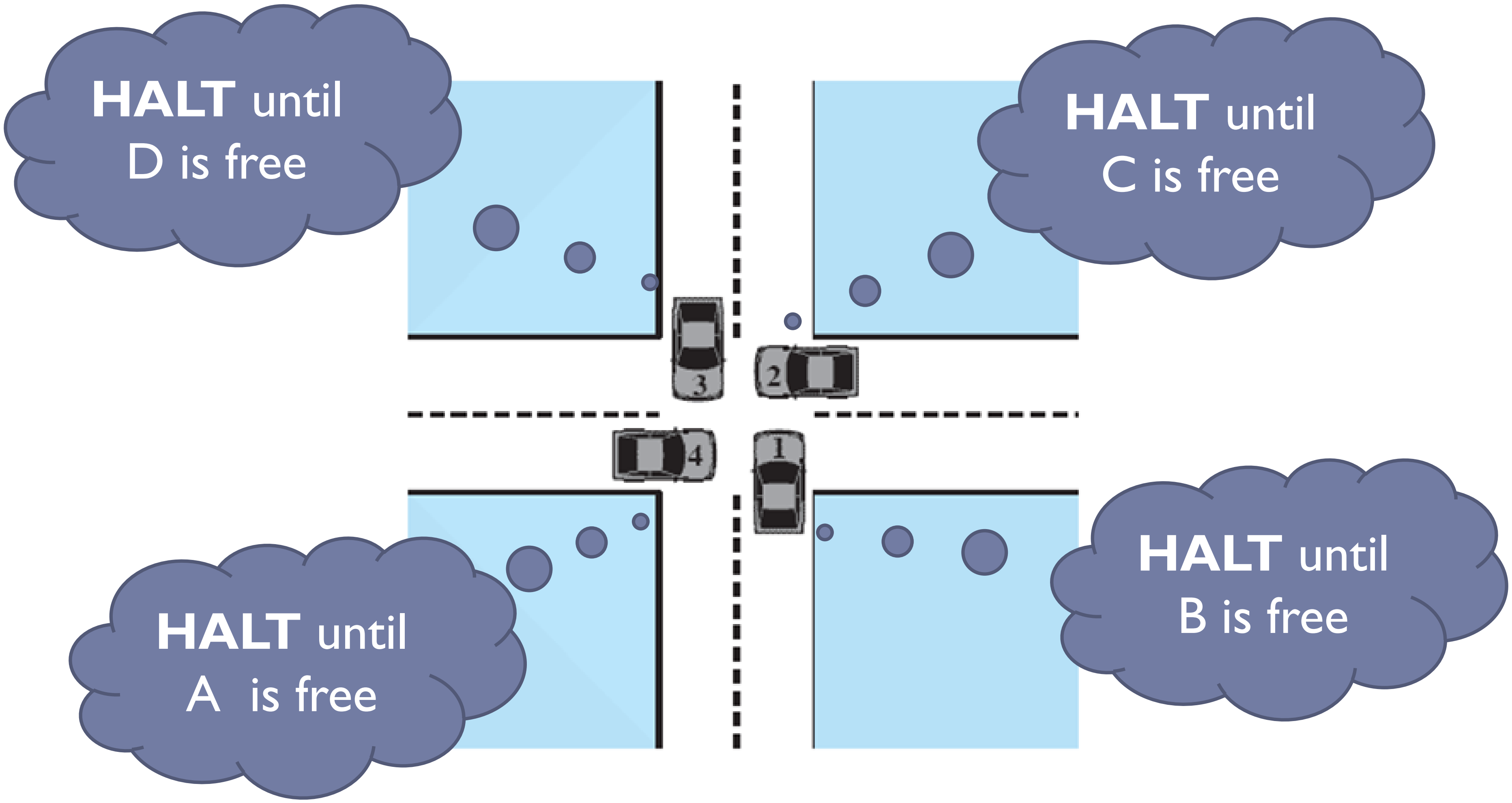
- The permanent blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent
- No efficient solution

Potential Deadlock

What if all four cars wish to go straight through the intersection?



Actual Deadlock



Deadlock Example

```
/* Task_A */
...
wait(&s1); /* wait for resource 1 */
...    /* use resource 1 */
wait(&s2); /* wait for resource 2 */
deadlock here
...    /* use resource 2 */
signal(&s2); /* release resource 2 */
signal(&s1); /* release resource 1 */
...

/* Task_B */
...
wait(&s2); /* wait for resource 2 */
...    /* use resource 2 */
wait(&s1); /* wait for resource 1 */
deadlock here
...    /* use resource 1 */
signal(&s1); /* release resource 1 */
signal(&s2); /* release resource 2 */
...
```

- Task A and Task B require resources 1 and 2
- Task A possesses resource 1 and waits for recourse 2
- Task B possesses recourse 2 and waits for recourse 1

Conditions for Deadlock

Mutual Exclusion	Hold-and-Wait	Nonpreemption	Circular Wait
• only one task may use a resource at a time (communication channels, disk drives, printers etc.)	• a task may hold allocated resources while awaiting assignment of others	• no resource can be forcibly removed from a task holding it	• a closed chain of tasks exists, such that each process holds at least one resource needed by the next task in the chain

Eliminating any one of the four necessary conditions will prevent deadlock from occurring.

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a task requests a resource, it does not hold any other resources
  - Require task to request and be allocated all its resources before it begins execution, or allow task to request resources only when the task has none allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention

- **Nonpreemption –**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Deadlock Prevention

- **Circular Wait**

One way to eliminate circular wait is to impose an explicit ordering on the resources and to force all the tasks to request all resources above the number of the lowest one needed

Device numbering scheme to eliminate the circular wait condition

Device	Number
disk	1
printer	2
motor control	3
monitor	4

$\downarrow$ rank

- If task needs to use the printer, it will be assigned the printer, motor control and monitor.
- If other task request the monitor only, it will have to wait until the three resources are released

Detect and Recover

- Detect deadlock with watchdog timers and external monitors.
- Can ignore possibility of deadlock in non-critical systems.
- System reset may be viable in non-critical systems.
- In some cases rollback to a pre-deadlock state can be performed
- However, this may lead to a recurrent deadlock, and operations such as writing to certain files or devices cannot be rolled back easily.

Avoiding Deadlocks when Using Semaphores

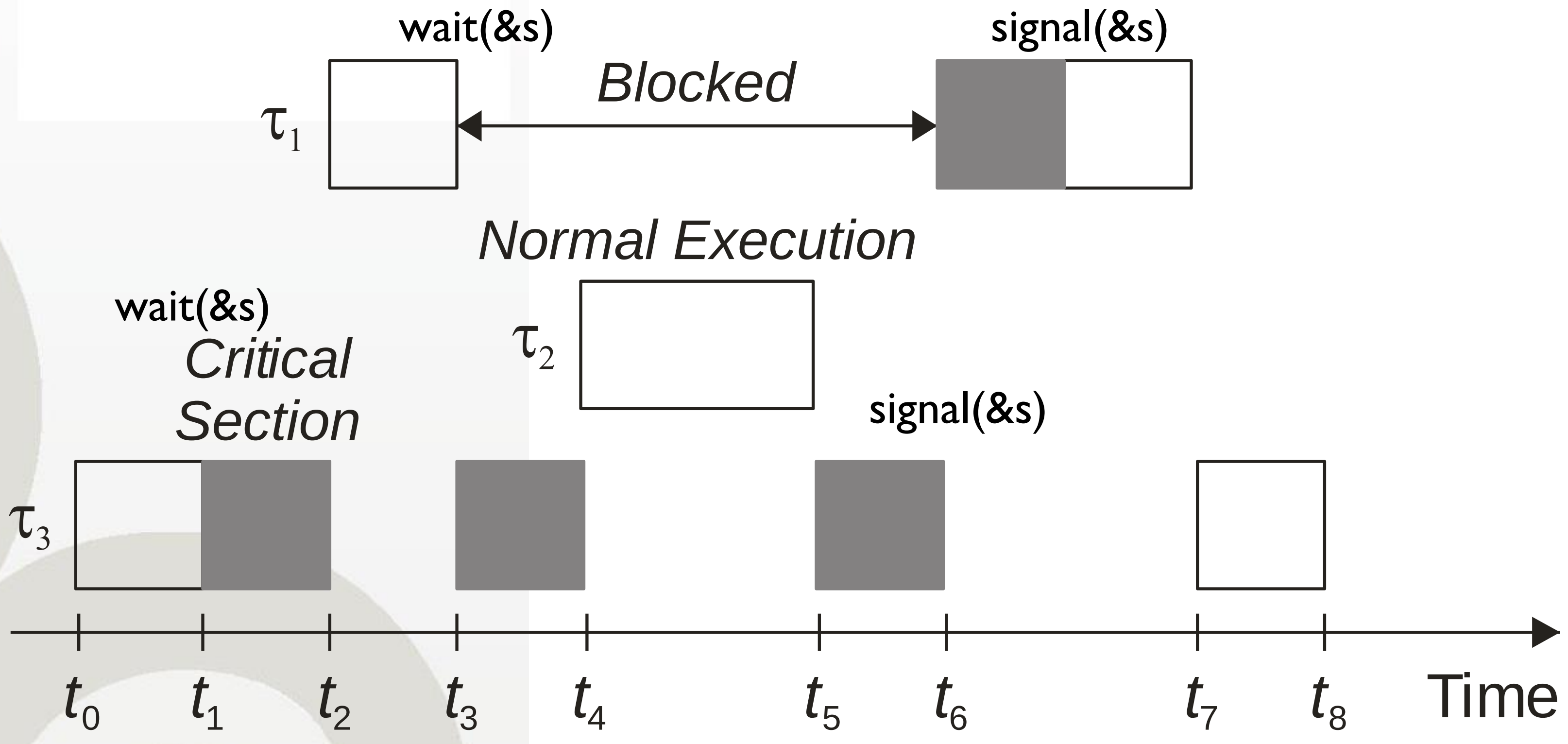
- Best way to deal with deadlock is to avoid it.
- If semaphores protecting critical resources are implemented with time-outs, then deadlocking cannot occur. But starvation of one or more tasks is possible.
- The following approach is recommended can help avoid deadlock when using semaphores protecting critical regions:
  - Minimize the number of critical regions and their length.
  - All tasks must release any lock as soon as possible.
  - Do not suspend any task while it controls a critical region
  - All critical regions must be 100% error free.
  - Do not lock devices in interrupt handlers.
  - Always perform validity checks on pointers used within critical regions. Pointer errors are common in certain languages, like C, and can lead to serious problems within the critical regions.

Priority Inversion

- A **priority inversion** is said to occur in the case when a lower priority task blocks a higher-priority one.
- May be caused by semaphore usage, device conflicts, bad design of interrupt handlers, poor programming and system design.

A Typical Priority-Inversion Scenario

- Tasks τ_1, τ_2 and τ_3 have following priorities: $\tau_1 > \tau_2 > \tau_3$
- Tasks τ_1 and τ_3 share some data or resource with exclusive access



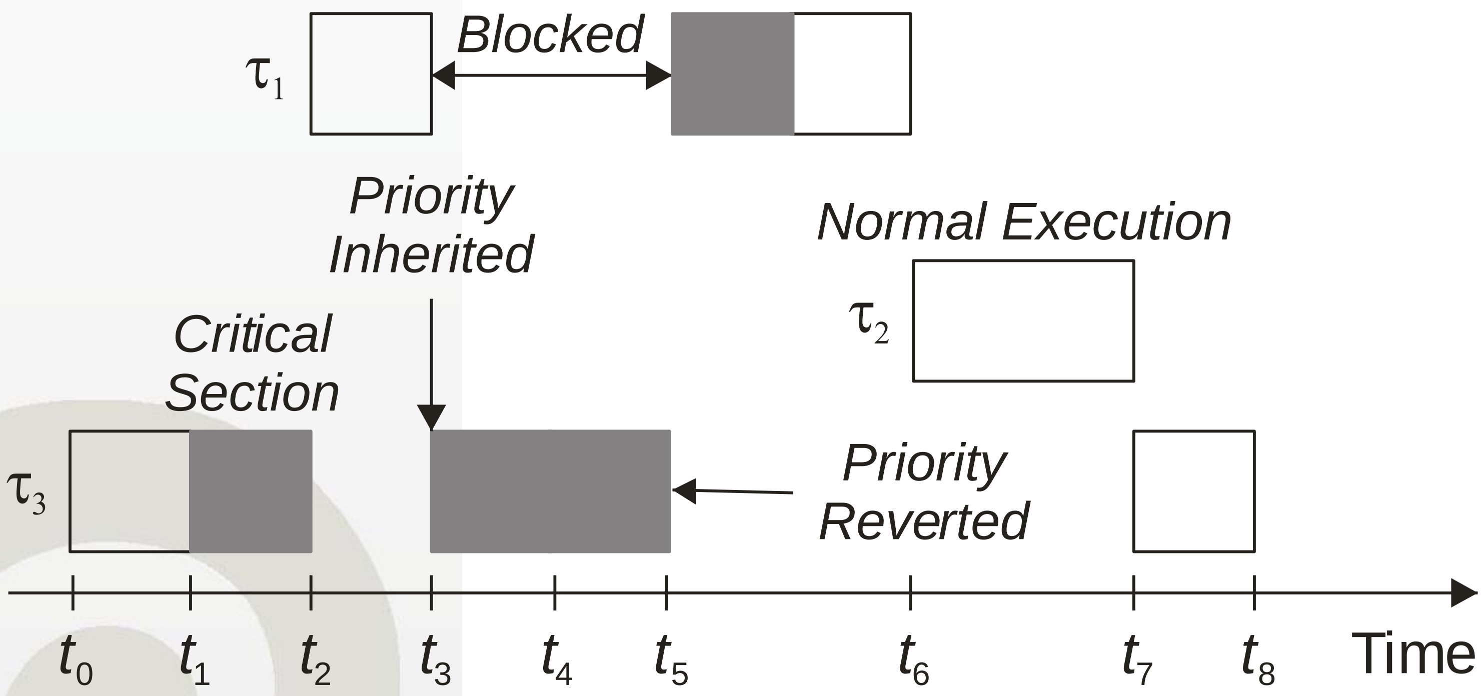
A priority inversion is said to occur within time interval [t4, t5] when task τ_2 preempted higher priority task τ_1

Priority Inheritance Protocol

- A solution to the priority inversion problem such that when a task blocks one or more higher-priority tasks, it temporarily inherits the highest priority of the blocked tasks – the **priority inheritance protocol**
- The highest-priority task τ_1 quits the processor whenever it seeks to lock the semaphore guarding a critical section that is already locked by some other job.
- If a task τ_1 is blocked by τ_2 and $\tau_1 > \tau_2$, task τ_2 inherits the priority of τ_1 as long as it blocks τ_1. When τ_2 exits the critical section that caused the block, it reverts to the priority it had when it entered that section.
- Priority inheritance is transitive. If τ_3 blocks τ_2, which blocks τ_1, then τ_3 inherits the priority of τ_1 via τ_2.

Priority-Inheritance Protocol

- The priority of τ_3 would be temporarily raised to that of τ_1 at time t3, thereby preventing τ_2 from preempting it at time t4.
- The priority of τ_3 reverts to its original at time t5, and τ_2 is executed only after completing τ_1

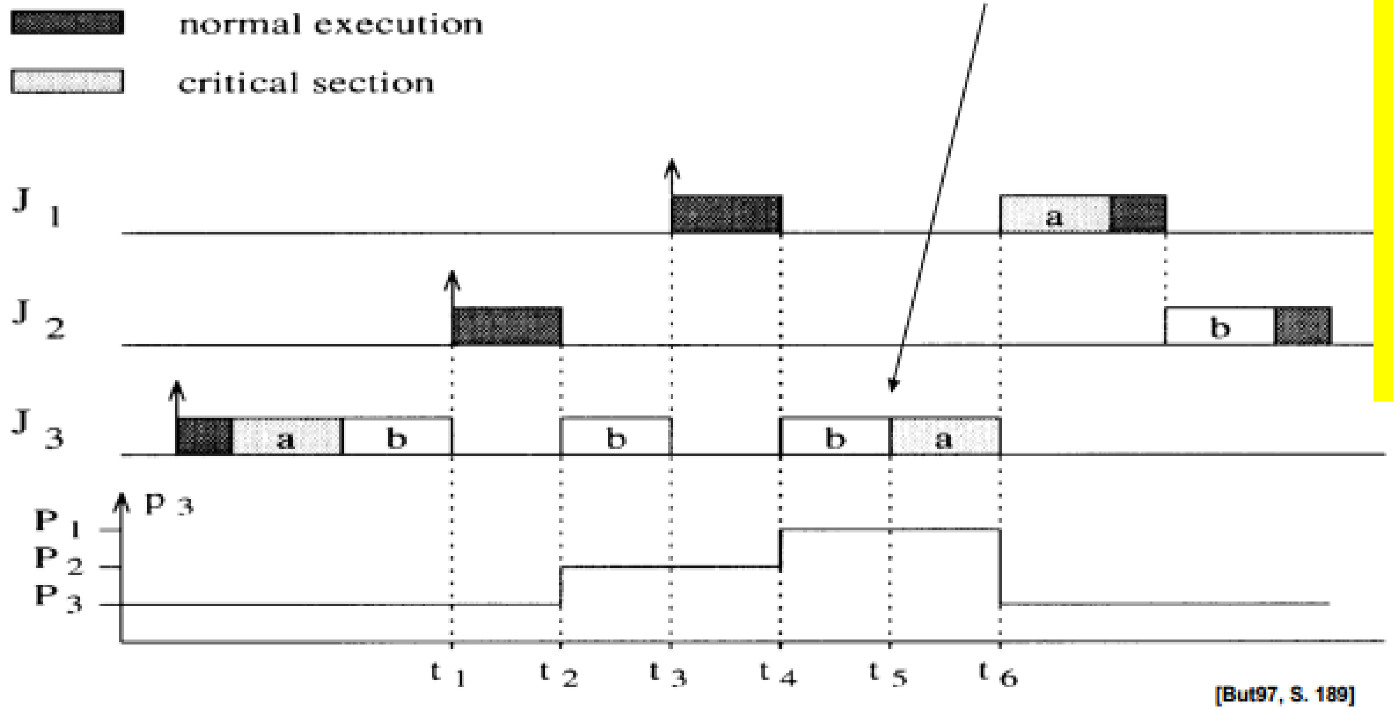


Priority-Inheritance Protocol

- J_1 requests resource **a**, J_2 requests resource **b**

- Example with nested critical sections:

priority does not change

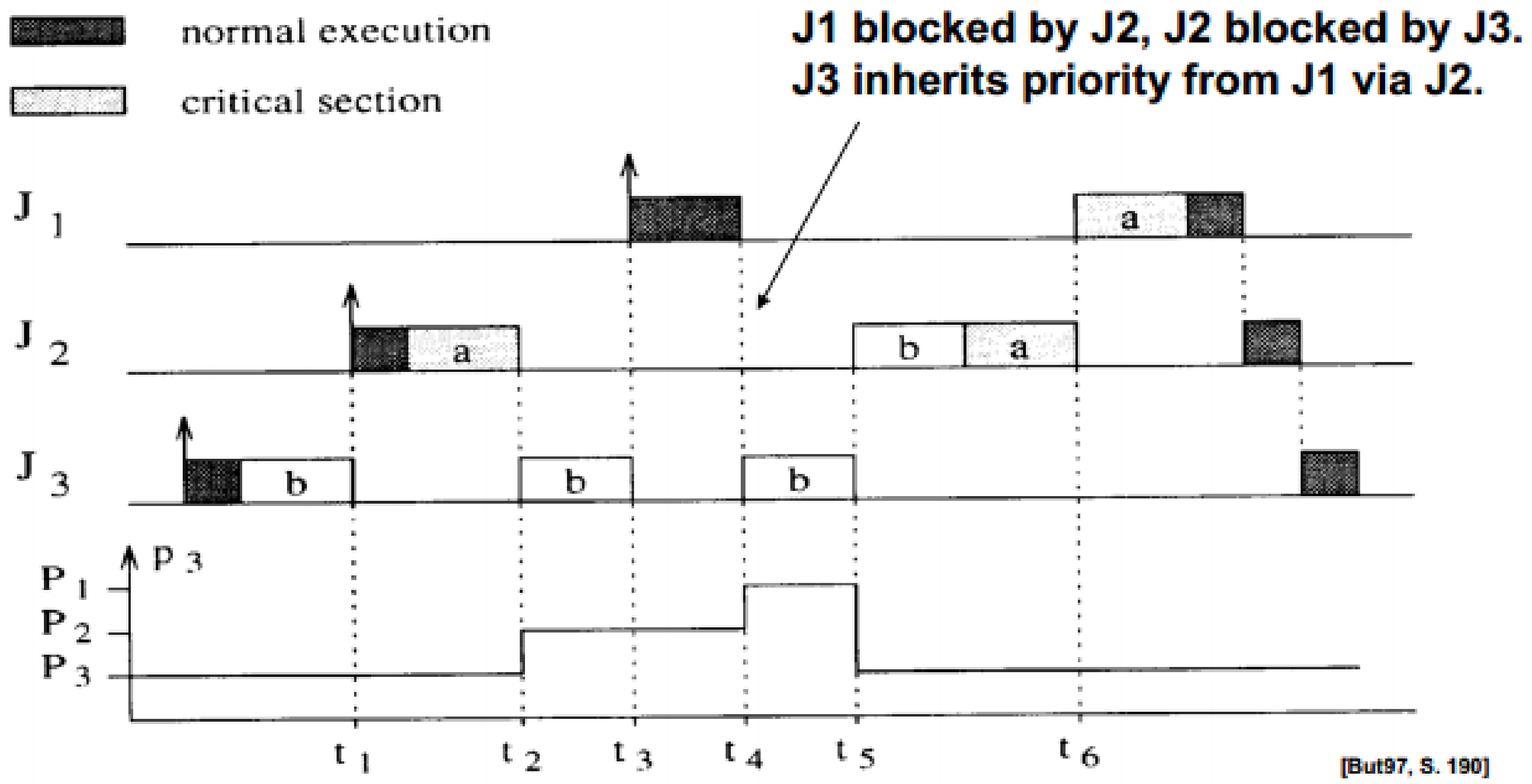


Task J3:

```
…
wait(&s_a)
wait(&s_b)
…
signal(&s_b)
signal(&s_a)
…
```

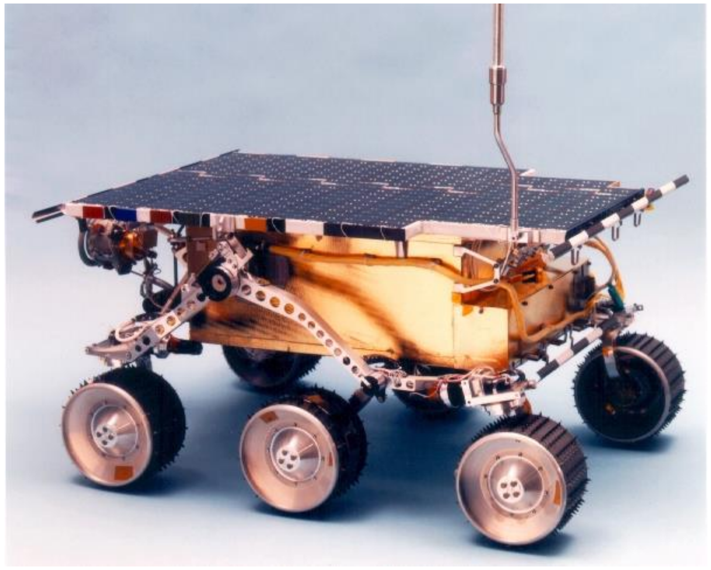
Priority-Inheritance Protocol

- Example of transitive priority inheritance:



Priority Inheritance Protocol Example: MARS Pathfinder problem

- Priority inversion occurred in 1997 in NASA's Mars Pathfinder Space mission's Sojourner rover vehicle.
- MIL-STD-1553B information bus manager was synchronized with mutex locks.
- A meteorological data gathering task that was of low priority and low frequency blocked a communications task that was of higher priority and higher frequency.
- This infrequent scenario caused the system to reset.



Priority Inheritance Protocol Example: MARS Pathfinder problem

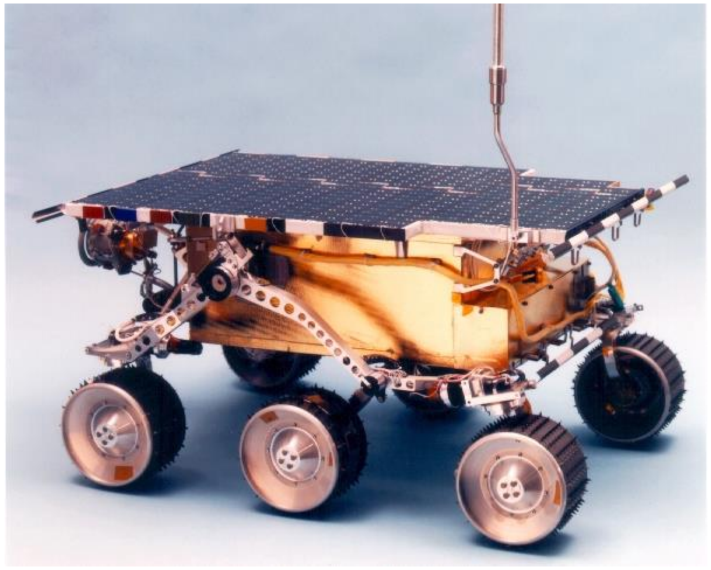
- VxWorks RTOS provides preemptive priority scheduling of threads.
- Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.
- The **meteorological data** gathering task ran as an infrequent, **low 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**

Priority Inheritance Protocol Example: MARS Pathfinder problem

Most of the time this combination worked fine. 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. This scenario is a classic case of priority inversion

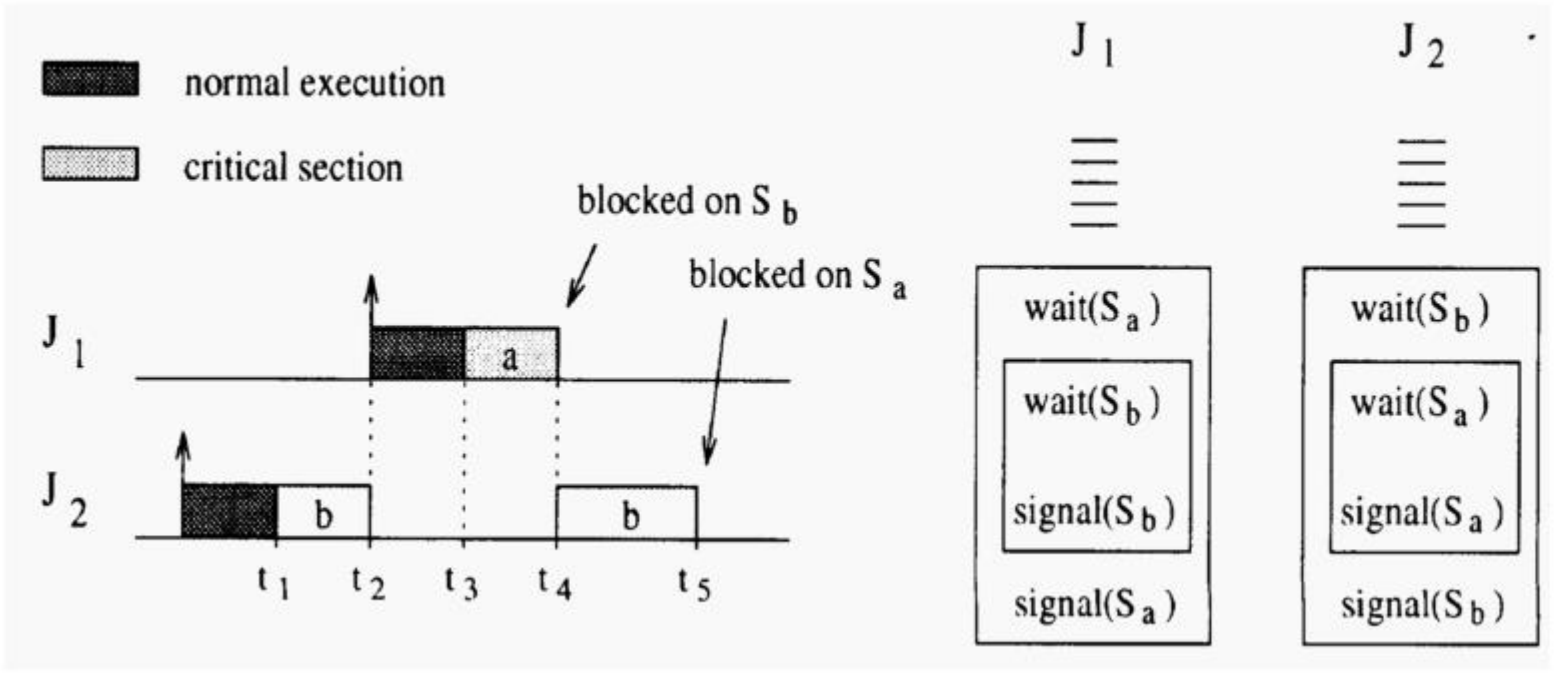
Priority Inheritance Protocol Example: MARS Pathfinder problem

- The VxWorks RTOS allows priority inheritance to be set to “on”. When the software was shipped, it was set to “off”
- Fortunately, the problem was diagnosed in ground based testing and remotely corrected by re-enabling the priority inheritance mechanism, while the Pathfinder was already on the Mars



Priority Inheritance Protocol

- PIP does not prevent deadlock
- Problem: ***Deadlock***



J_1

```
wait(S_a)
    wait(S_b)

    signal(S_b)
signal(S_a)
```

J_2

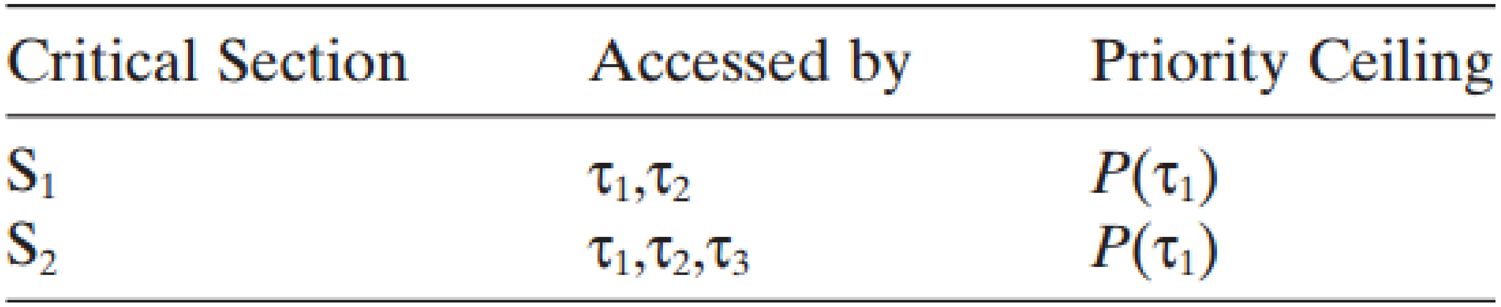
```
wait(S_b)
    wait(S_a)

    signal(S_a)
signal(S_b)
```

Priority Ceiling Protocol

- The **Priority Ceiling Protocol** extends the Priority Inheritance Protocol through chained blocking in such a way that no task can enter a critical section in a way that leads to blocking it.
- Each resource is assigned a priority (the priority ceiling) equal to the priority of the highest priority task that can use it.
- The Priority Ceiling Protocol is the same as the Priority Inheritance Protocol, except that a task τ_i can also be blocked from entering a critical section if there exists any semaphore currently held by some other task whose priority ceiling is greater than or equal to the priority of τ_i

Priority-Ceiling Protocol: Example



Critical Section	Accessed by	Priority Ceiling
S_1	τ_1, τ_2	$P(\tau_1)$
S_2	τ_1, τ_2, τ_3	$P(\tau_1)$

- ❑ Suppose τ_2 currently holds a lock on section S_2, and τ_1 is initiated.
- ❑ Task τ_1 will be blocked from entering section S_1, because its priority is not greater than the priority ceiling of the section S_2.

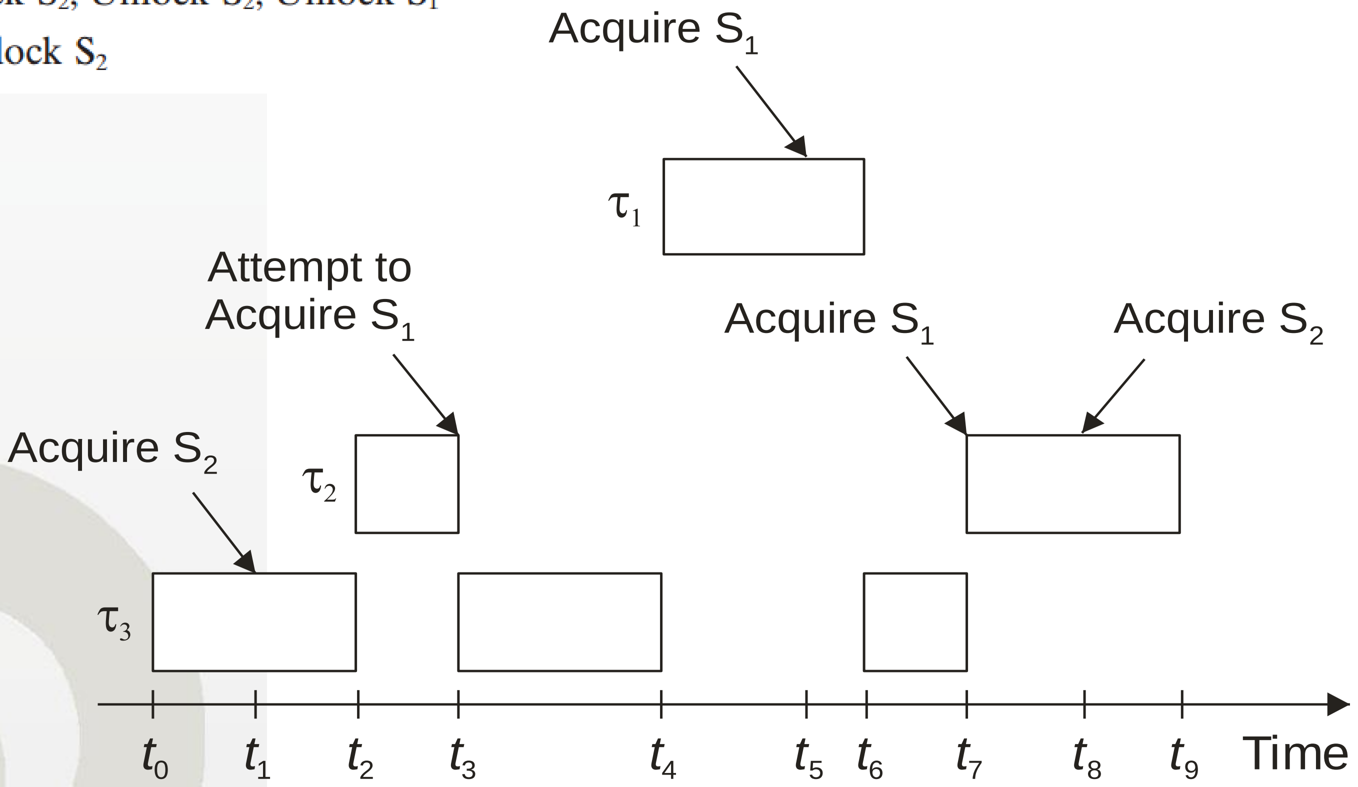
Priority-Ceiling Protocol

Consider the three tasks with the following sequence of lock – unlock operations, and having decreasing priorities ($\tau_1 > \tau_2 > \tau_3$)

τ_1: Lock S_1; Unlock S_1

τ_2: Lock S_1; Lock S_2; Unlock S_2; Unlock S_1

τ_3: Lock S_2; Unlock S_2



Any Questions?