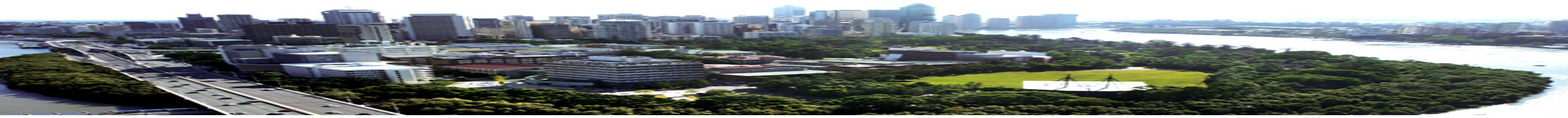
ENB 350 Real-time Computer based systems

Lecture 11 – AVOIDING UNBOUNDED PRIORITY INVERSION

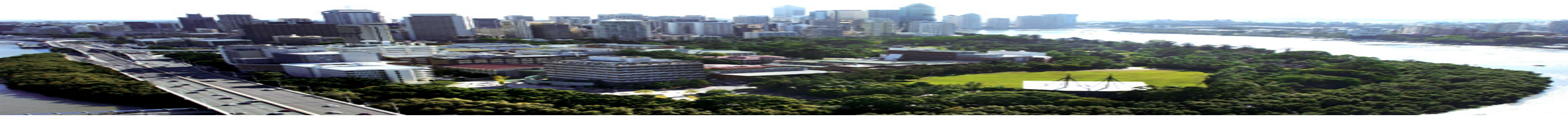
Prof. V. Chandran



Contents

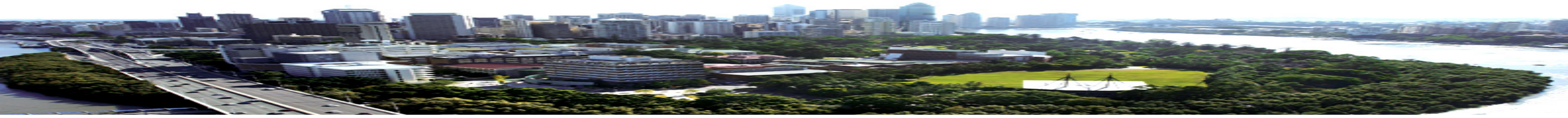
- Priority Inversion problem
- Priority Inheritance Protocol
- Priority Ceiling Protocol
- Deadlock
- Mutex Support in Micro C/OS-II
- Demonstration - priority inversion and mutex

These demonstrations will not be posted on Blackboard as they are part solutions to lab exercises.



Priority Inversion

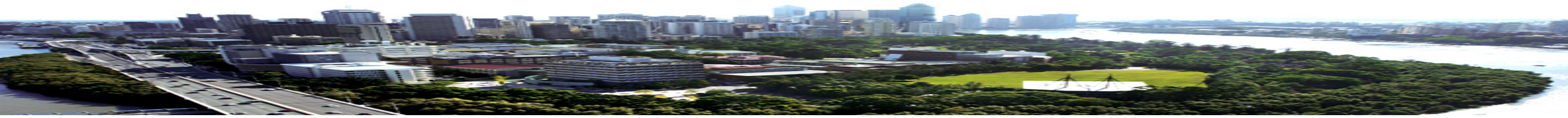
Priority inversion is a situation where a high priority task is prevented from running by a lower priority task   because it has to wait for a resource being held by a lower priority task.



Bounded / Unbounded

- bounded -> lasts a short period of time
  unbounded -> potentially indefinite

- For unbounded priority inversion to occur there must be at least 3 tasks.



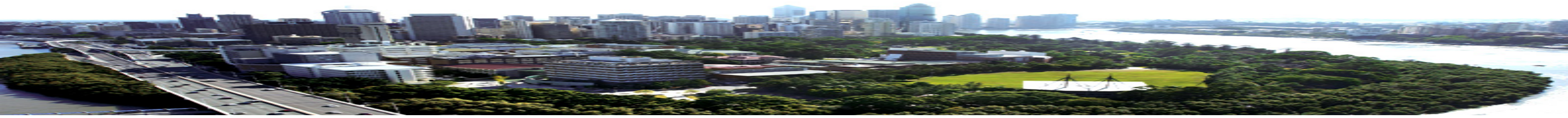
Priority inversion example

Task A (High priority)

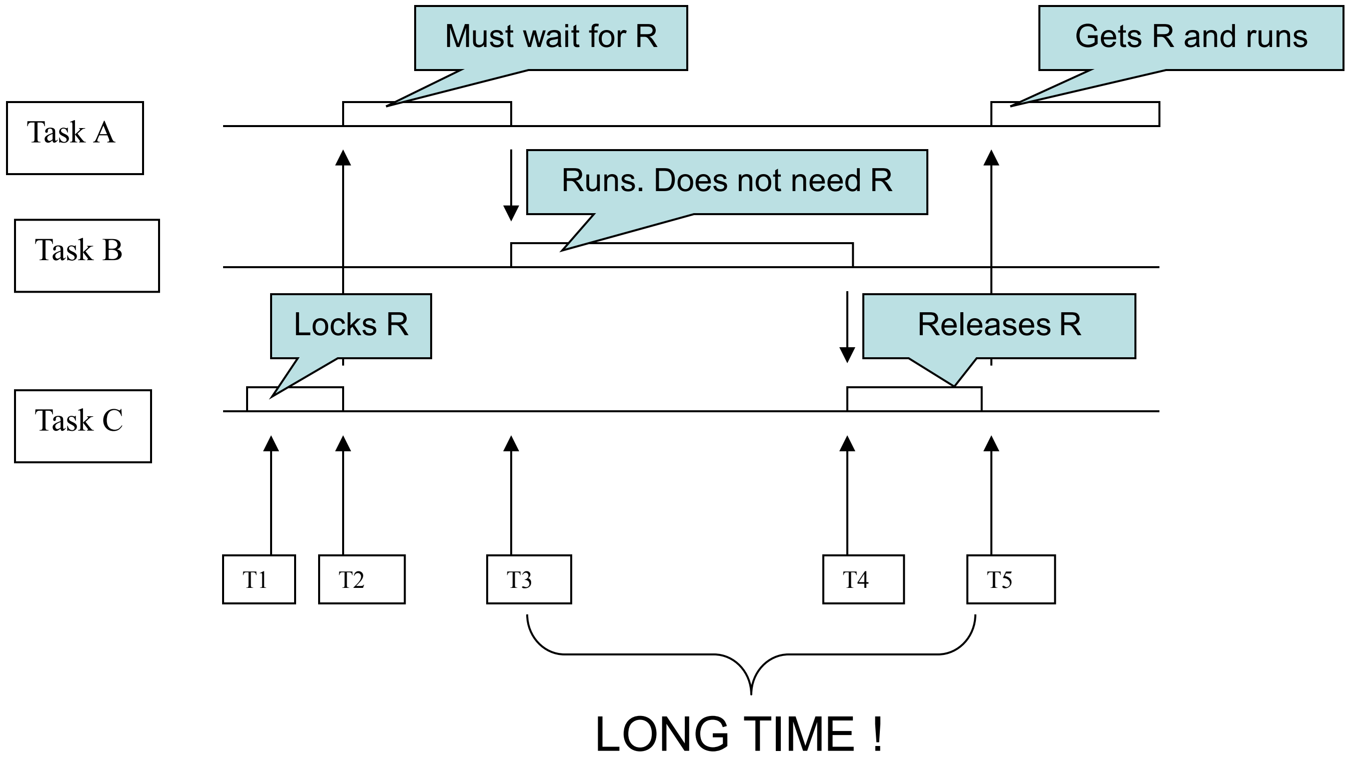
Task B (Medium)

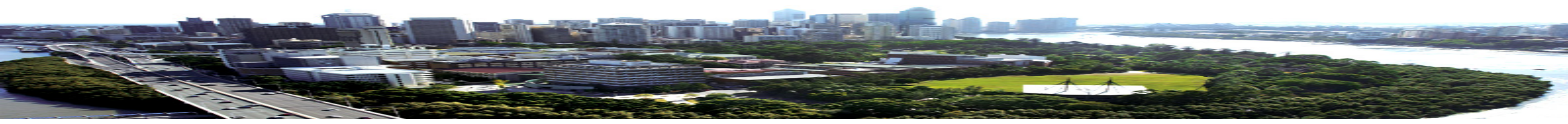
Task C (Low)

- Resource R shared by A and C protected by a semaphore is currently held by C. A must wait for R.
- B becomes ready and runs, pre-empts C.
- B can run for a long time and C never gets a chance to relinquish the resource lock and awaken A until it is too late.



Priority inversion example

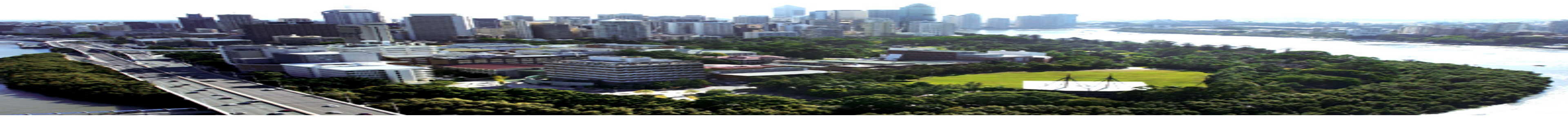




Mars Pathfinder case

- In 1997 the priority inversion problem occurred in NASA Mars Pathfinder mission's Sojourner rover vehicle used to explore the surface of Mars.

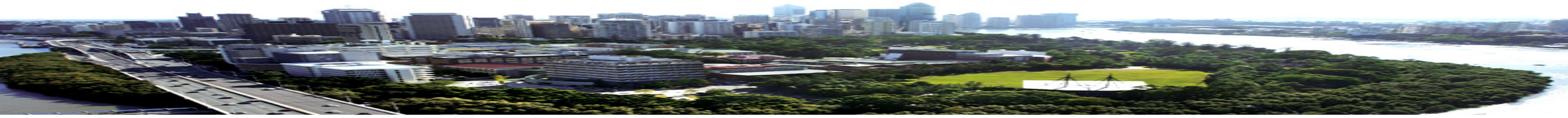
- Three threads (among others): a high priority bus management thread designed to run quickly and frequently, a medium priority communication thread that ran for a much longer time and a low priority meteorological thread that ran infrequently.



Mars Pathfinder case

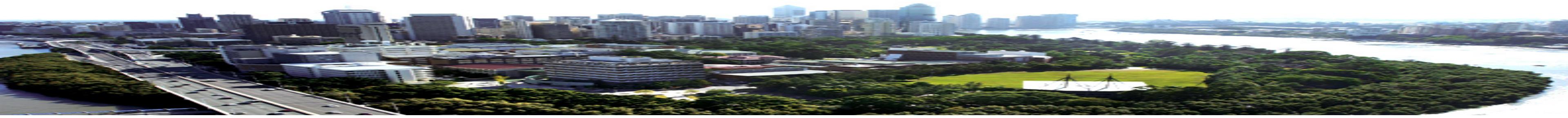
- The meteorological thread required the (shared) bus to publish data. A binary semaphore was used. The meteorological thread acquired it and the bus manager had to wait. The meteorological thread was pre-empted by the communications thread and the bus manager had to wait longer – causing a missed deadline, alarm and system hardware reset.

- The problem was diagnosed in ground based testing and remotely corrected by re-enabling the priority inheritance mechanism of a RTOS that was used.

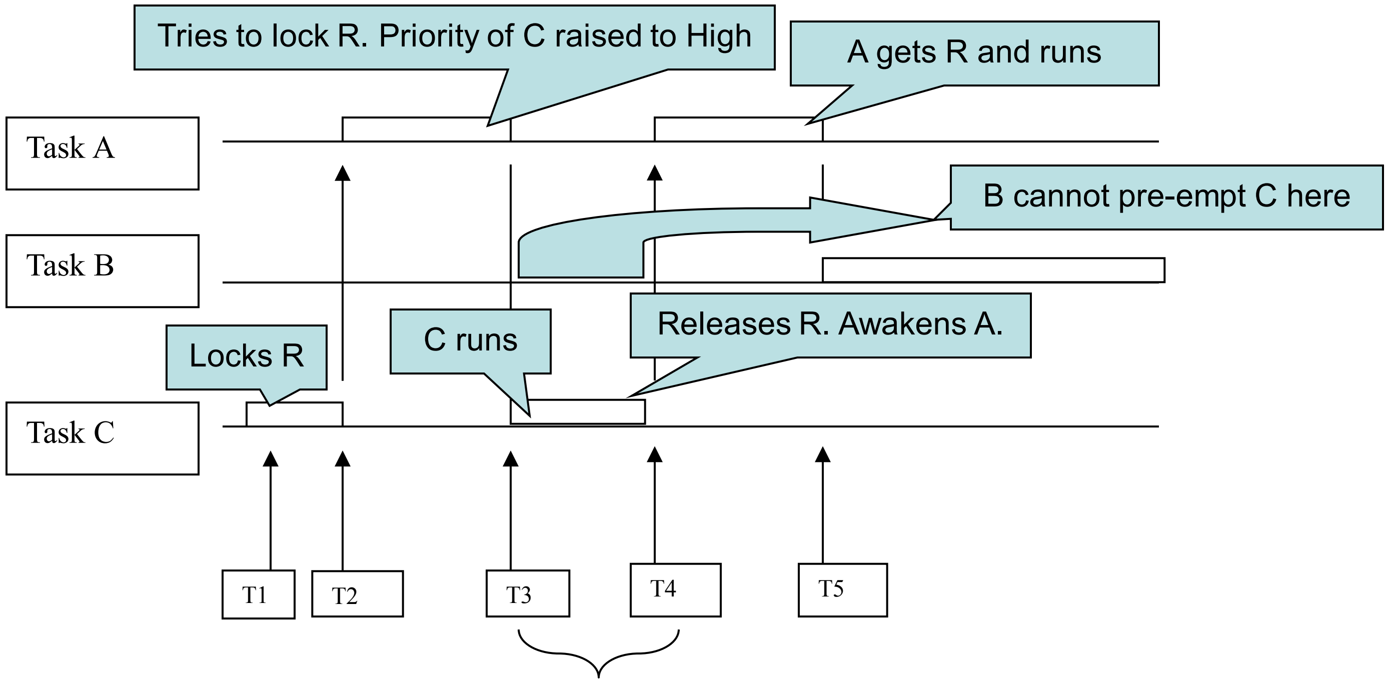


Priority Inheritance protocol

- A simple solution
- Priorities of tasks are dynamically changed
- A task in a critical section inherits the priority of the highest task pending on that critical region.



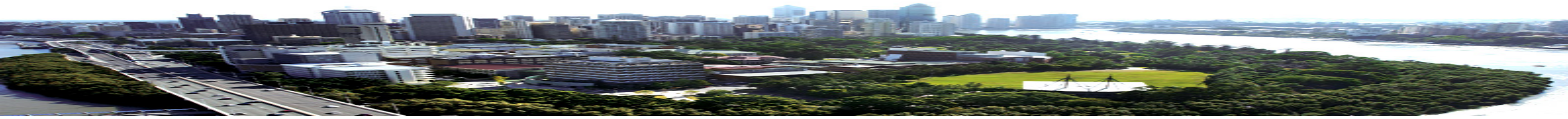
Priority Inheritance example



SHORT CRITICAL SECTION

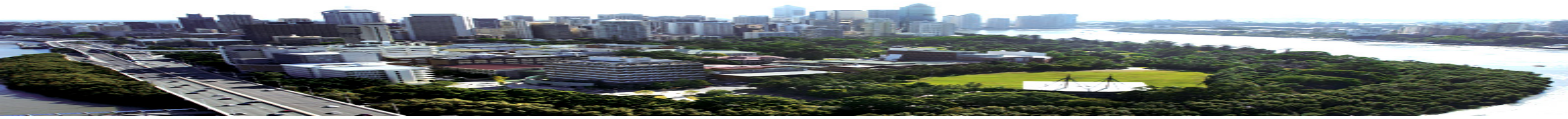
Protocol highlights

- The highest priority task relinquishes the processor whenever it seeks to unlock the semaphore guarding a critical section that is already locked.
- If a task P1 is blocked by P2 and P1 has precedence over P2, P2 inherits the priority of P1 as long as it blocks P1. When P2 exits the critical section, it reverts to the priority it had when it entered the critical section.
- Priority inheritance is transitive. If P1 blocks for resource held by P2 and P2 blocks for P3, the P3 inherits the priority of P1 via P2.

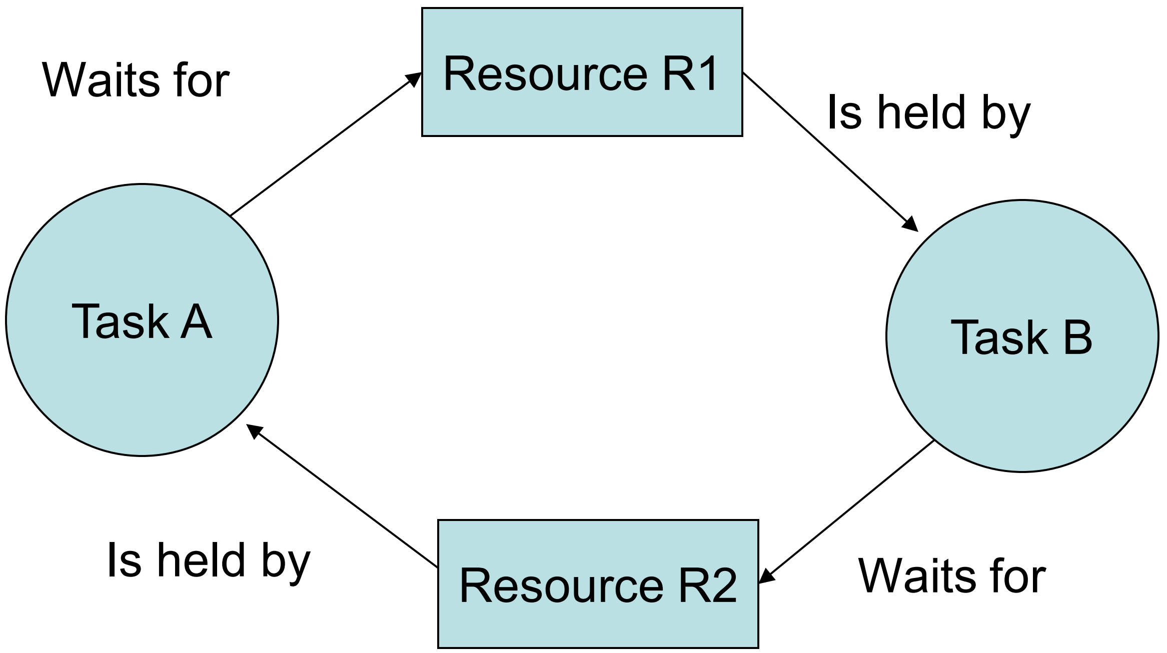


Problems

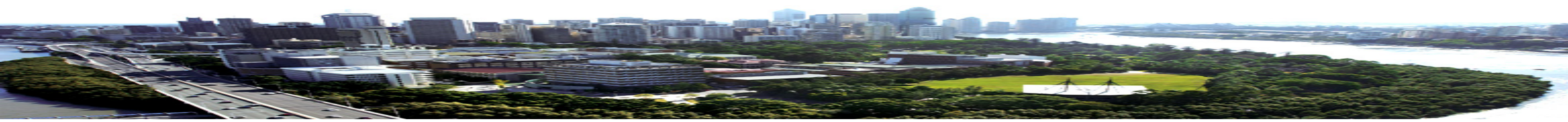
- Priority inheritance does not prevent deadlock.
- Chains of blocking may be formed and the blocking duration can become substantial.



Deadlock

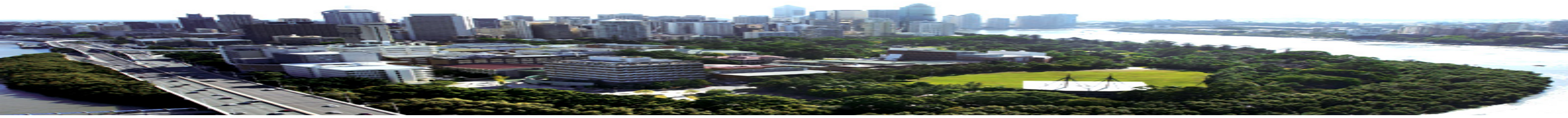


Neither Task will be awakened



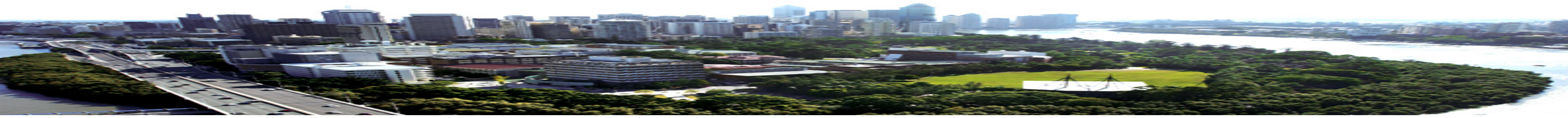
Priority inheritance protocol

- Transparent to the application
- Adds moderate complexity to the real-time kernel when it is supported
- Keeping track of all the threads that have acquired a semaphore is inefficient



Priority inheritance protocol

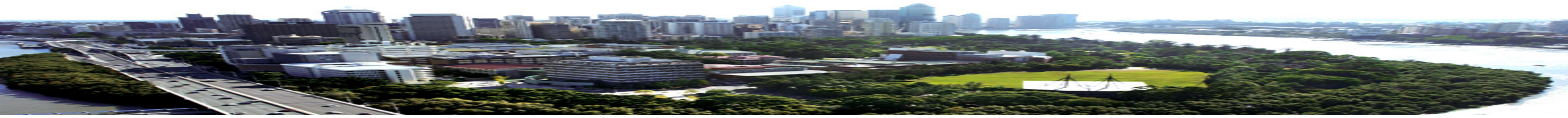
- Attractive when nesting of locks is not going to occur
- Average performance is good because there is no penalty when locks are not contended
- Deadlock can be avoided by other means



Priority Ceiling Protocol

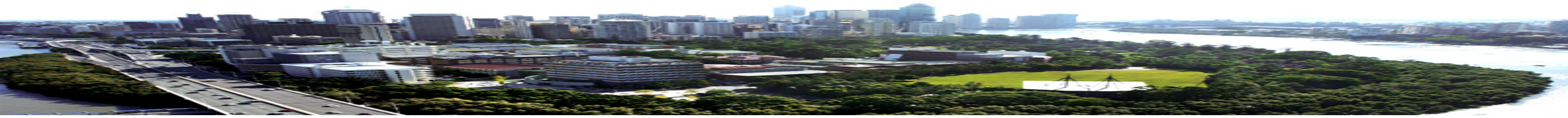
- Avoids unbounded priority inversion and mutual deadlock due to wrong nesting of critical sections.

- Each resource is assigned a priority ceiling
- Priority ceiling = the priority of the highest task that can use this resource.



Priority Ceiling Protocol

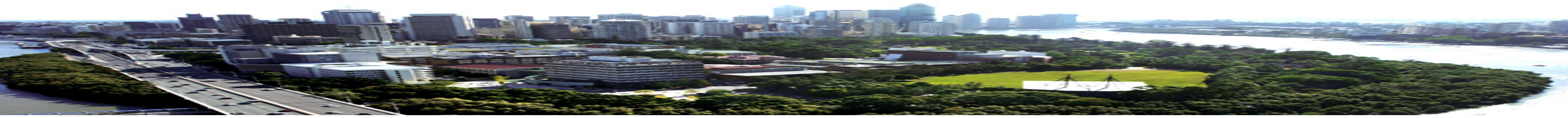
- A task can be blocked from entering a critical section if there exists any semaphore currently held by some other task whose priority ceiling is greater than or equal to the priority of this task.
- Originally proposed by Sha, Rajkumar and Lahoczky – but the algorithm is difficult to implement and not available on commercial RTOS.



Immediate Ceiling Priority Protocol

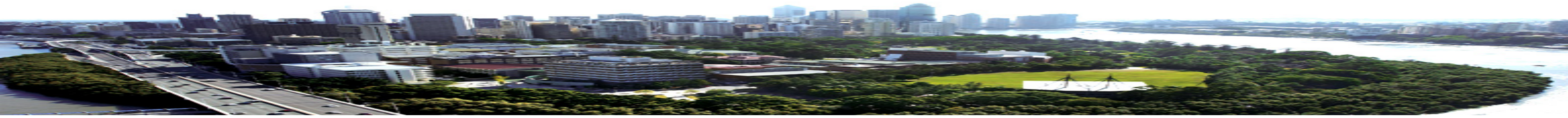
- The priority of a task that locks a resource is immediately raised to the priority ceiling of the resource. (Also called the priority ceiling emulation protocol or highest locker protocol)

- No task that needs the resource will therefore get scheduled. Deadlock is prevented. However, a lot of unnecessary blocking may occur.



Immediate Ceiling Priority Protocol

- Can be implemented within the application even if not provided by the kernel.
- Priority ceiling must be pre-determined for each resource (mutex). Static analysis is necessary.
- Supported by most real-time kernels
- Worst case performance is good
- Average performance may suffer.
- Unwanted blocking may increase the jitter or variability in the period of periodic tasks



Example:

3 Tasks

Task A : Lock S1; Unlock S1;

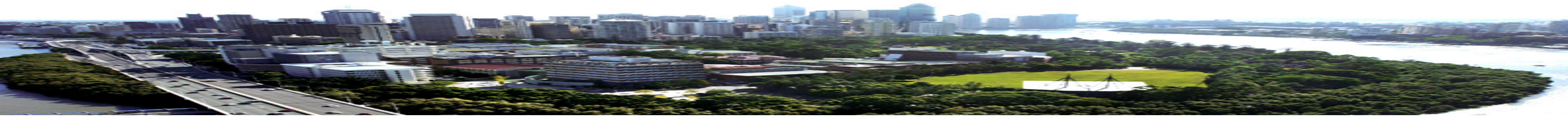
Task B: Lock S1; Lock S2; Unlock S2; Unlock S1;

Task C: Lock S2; Unlock S2;

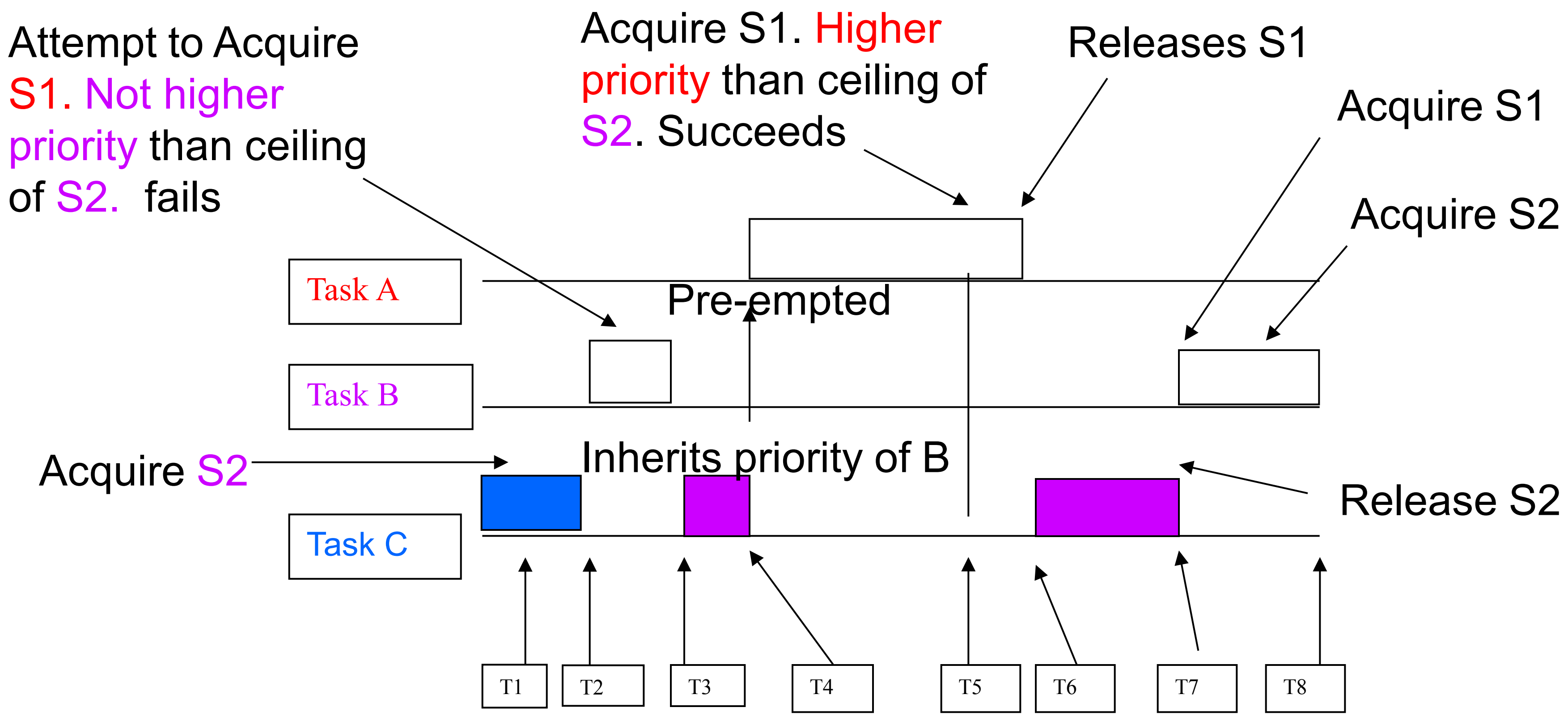
Priorities are PA, PB and PC

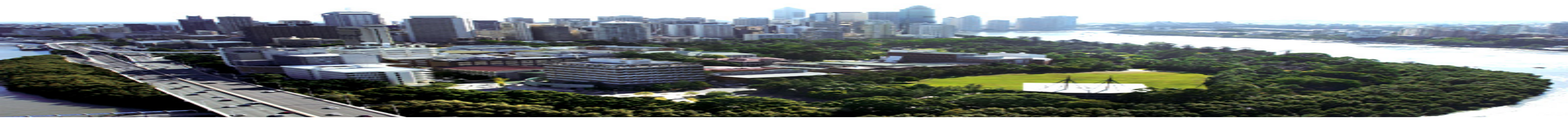
Priority ceilings of S1 and S2 are PA and PB, respectively.

Task B attempts to acquire semaphores S1 and S2 in that order. Measure to prevent deadlock.



Priority ceiling example


Attempt to Acquire S1. Not higher priority than ceiling of S2. fails
Acquire S1. Higher priority than ceiling of S2. Succeeds
Releases S1
Acquire S1
Acquire S2
Task A
Pre-empted
Task B
Acquire S2
Inherits priority of B
Release S2
Task C
T1
T2
T3
T4
T5
T6
T7
T8



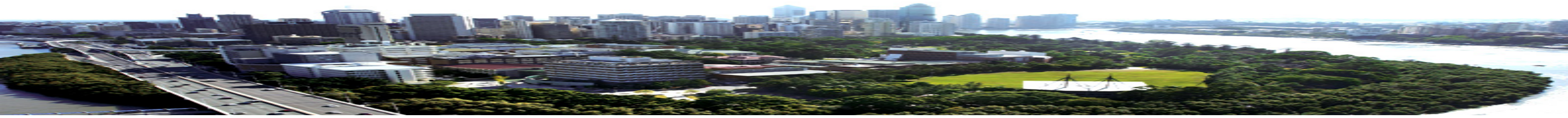
Explanation

Suppose Task C executes first and locks the semaphore S2 at time T1 and enters critical section.

At time T2, Task B starts executing, pre-empts Task C and attempts to lock semaphore S1 at time T3. At this time, Task B is suspended because its priority is not higher than the priority ceiling of semaphore S2, currently locked by Task C.

Task C temporarily inherits the priority of Task B and resumes execution.

(from Laplante, “Real-time Systems Design”.)



Explanation

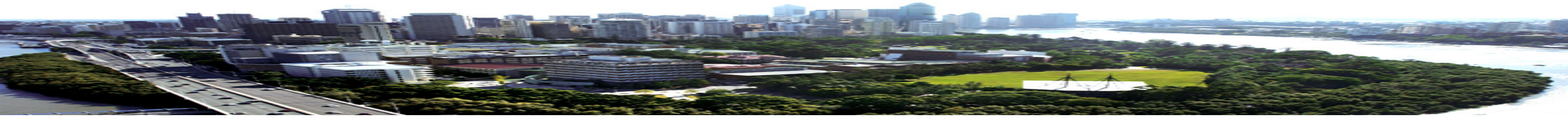
At time T4, task A enters, pre-empts Task C, and executes until time T5 where it tries to lock S1.

Task A is allowed to lock S1 at time T5 as its priority is greater than the priority ceiling of all the semaphores currently being locked.

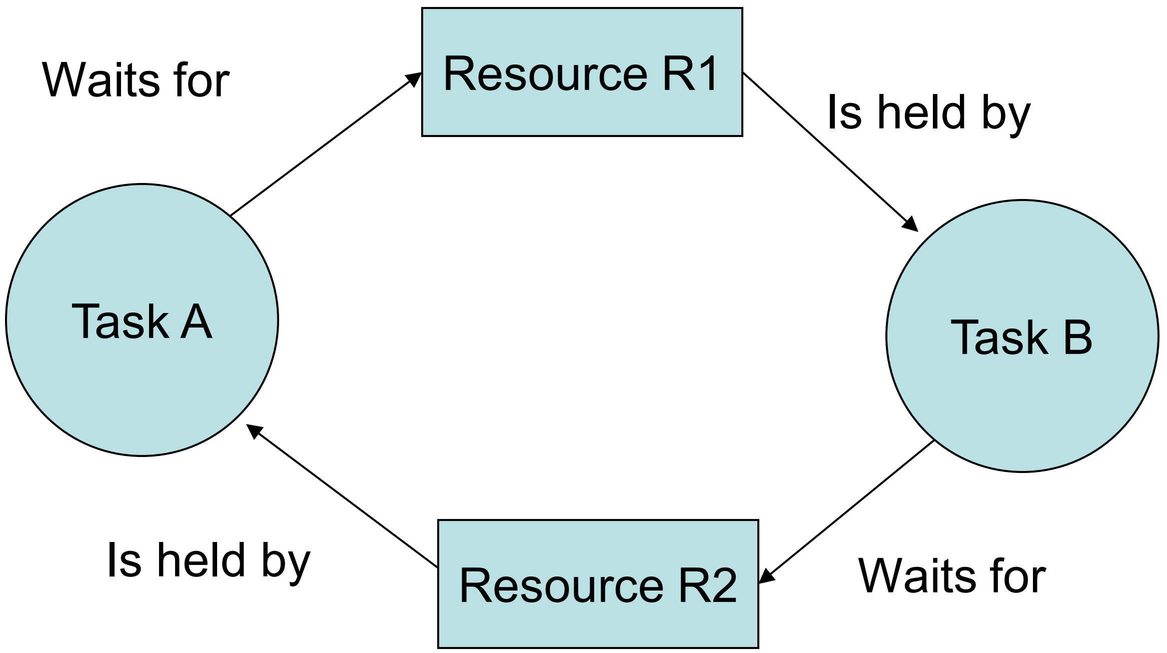
Task A completes its execution at T6 and lets Task C execute to completion at T7.

Task B is then allowed to lock S1 and subsequently S2 and completes at T8.

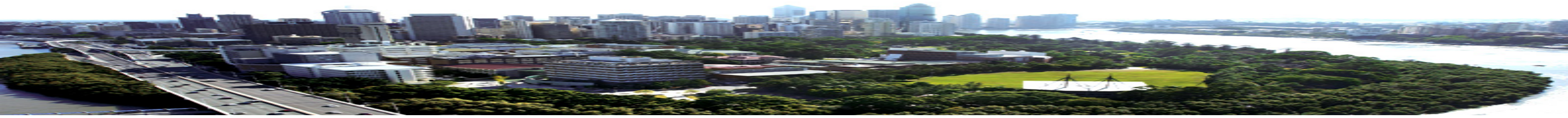
(from Phillip Laplante, “Real-time Systems Design”.)



Deadlock



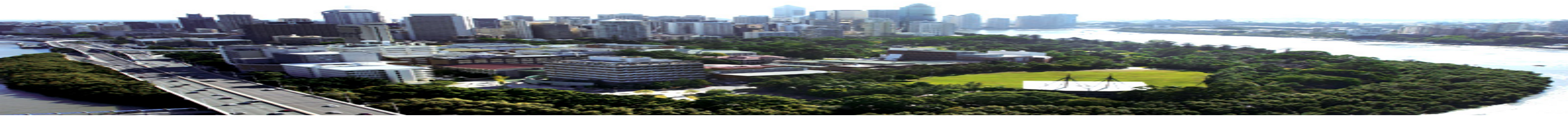
Neither Task will be awakened



Necessary conditions for deadlock

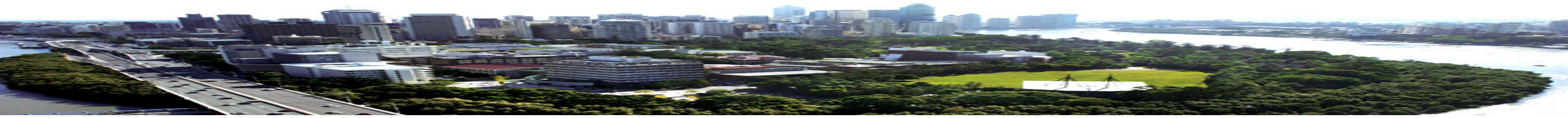
- Mutual exclusion
- Circular wait
- Hold and wait
- No pre-emption

- Eliminating any one will prevent deadlock



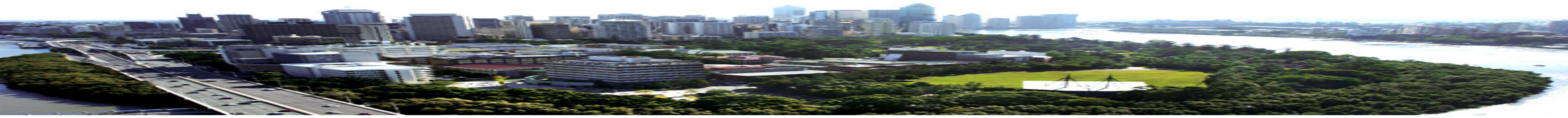
Resource ordering

- Impose an explicit ordering on the resources
- Force tasks to acquire all resources above the lowest one needed.
- Example Disk = 1, printer = 2, motor control = 3, Monitor = 4. If a task wants the printer, it will request and be assigned the printer, the motor control and the monitor.
- Circular wait is eliminated, but tasks may be starved of CPU time.



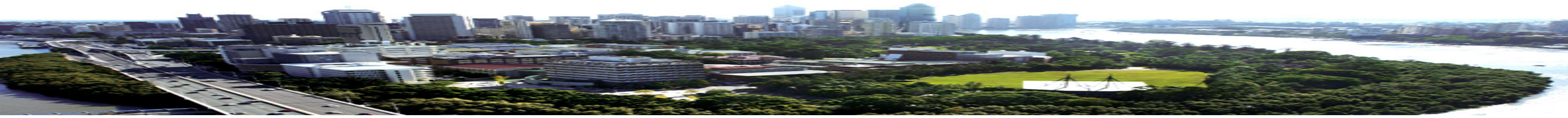
Other methods

- Mutual exclusion can be eliminated using spooling. Then only one task actually accesses the resource on behalf of the other tasks that want to.
- Allocate all required resources to a task at the same time. Can lead to starvation in other tasks.
- Use a deadlock avoidance algorithm such as the Banker's algorithm. (We won't cover this)
- Detect deadlock with a watchdog timer and have a recovery procedure (We won't cover this)

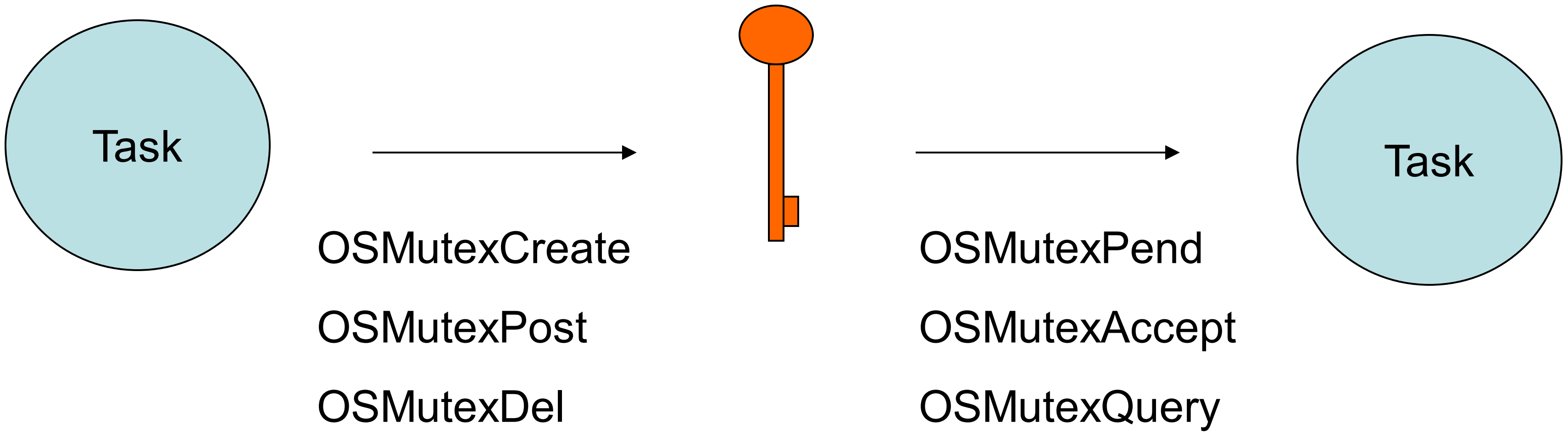


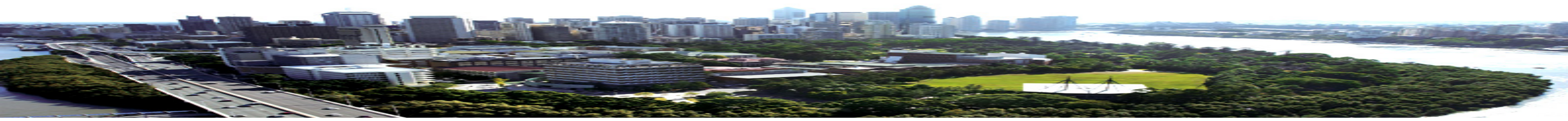
Mutex : ucos-ii support

- Mutex implementation in ucos-ii incorporates priority inheritance to reduce priority inversion
- Since ucos-ii priorities have to be unique, the implementation must work around this.
- It is the priority ceiling emulation protocol (or ICPP) that is implemented.



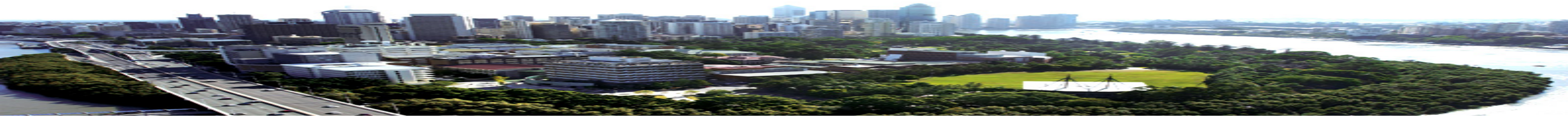
Mutex related kernel calls





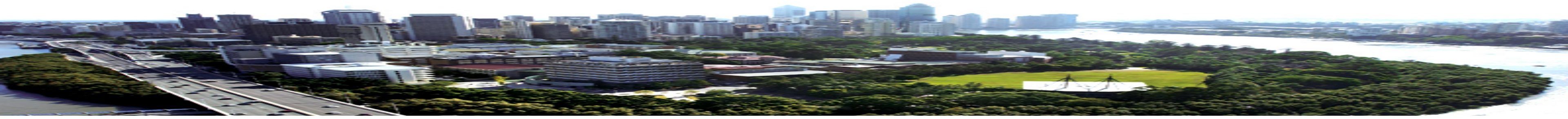
Mutex : some details

- Each mutex is associated with a Priority Inheritance Priority (PIP)
- The value of the mutex is initialized to 1 implying that the resource is available
- If a mutex is already allocated, the calling task is put to sleep. Before that, the PIP, priority of the owner and a pointer to its TCB are extracted. The priority of the owner is raised to PIP.
- This allows the owner to release the mutex sooner.



Mutex : some details

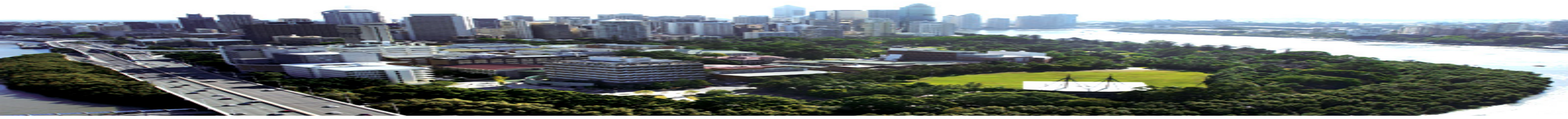
- When a mutex is released, OSMutexPost checks to see if the owner's priority had been raised to PIP. If so, it is restored to its original priority and then removed from the ready list at PIP and placed in the ready list at its original priority.
- The highest priority task waiting on that mutex is then made ready to run. The priority of the new owner is saved. OS_Sched() is then called.



Mutex : ucos-ii support

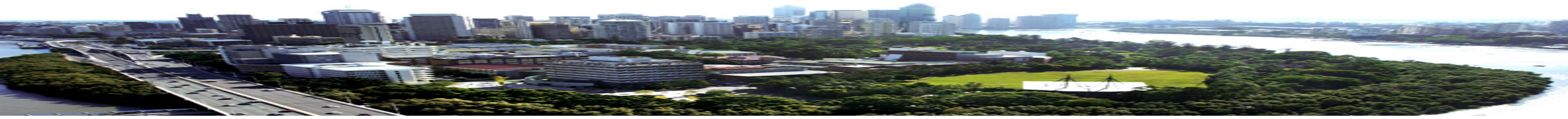
- Three tasks need to access a resource protected by a Mutex
- Task A (priority = 10)
- Task B (priority = 15)
- Task C (priority = 20)

- An unused priority (say 9) is reserved as the priority inheritance priority (PIP). This is higher than the highest priority for tasks waiting on that resource.



Mutex usage scenario

- Task C runs and locks the mutex.
- Task A pre-empts C and tries to acquire the mutex. It fails and has to wait, but the priority of Task C is then raised to the PIP(9). (It could be raised at the time of successful locking itself).
- Task B may become ready but cannot pre-empt Task C
- Task C executes its critical section quickly and releases the mutex. OSMutexPost will reduce its priority and also note the higher priority task A waiting and thus context switch to Task A
- Task A now runs without the risk of priority inversion (by Task B having run before it).



Tips for protecting resources

- Avoid using locks if possible and use non-locking atomic operations.
- Locks must held for the shortest time possible when acquired.
- Use built-in features of real-time kernels to simplify implementation.