PRIORITY INVERSION PROBLEM IN RTOS

• A priority is assigned to each task -- the more important the task, the higher the priority given to it.
• The application designer is responsible for deciding what priority each task gets.
• STATIC PRIORITIES -- task priorities are static when the priority of each task does not change during the application’s execution -- all the tasks and their timing constraints are known at compile time and each task gets a fixed priority at compile time.
• DYNAMIC PRIORITIES -- Task priorities are dynamic if the priority of tasks can be changed during the application’s execution -- each task can change its priority at run time. This feature is desirable to have in real-time kernels to avoid priority inversions.

PRIORITY INVERSION PROBLEM --

• Priority inversion is a situation in which a low-priority task executes while a higher priority task waits on it due to resource contentions.
  ○ Priority inversion is a problem in real-time system and occurs mostly when you use a real-time kernel.
  ○ May be caused by semaphore usage, device conflicts, bad design of interrupt handlers, poor programming and system design.
Example 1

Task 1 has a higher priority than Task 2, which in turn has a higher priority than Task 3.

- (1) Task 1 and Task 2 are both waiting for an event to occur and Task 3 is executing. (2) At some point, Task 3 acquires a semaphore, which the task needs before it can access a shared resource. (3) Task 3 performs some operations on the acquired resource. (4) The event for which Task 1 was waiting occurs, and thus the kernel suspends Task 3 and starts executing Task 1 because Task 1 has a higher priority. (5) Task 1 executes for a while until it also wants to access the resource at (6) (i.e. it attempts to get the semaphore that Task 3 owns). Because Task 3 owns the resource, Task 1 is placed in a list of tasks waiting for the kernel to free the semaphore. (7) Task 3 resumes and continues execution until it is preempted by Task 2 at (8) because the event for which Task 2 was waiting occurred. (9) Task 2 handles the event for which it was waiting, and when it’s done, the kernel relinquishes the CPU back to Task 3 at (10). During (11) Task 3 finishes working with the resource and releases the semaphore at (12). At this point, the kernel knows that a higher priority task is waiting for the semaphore and performs a context switch to resume Task 1. During (13) Task 1 has the semaphore and can access the shared resource.

- In this scenario the priority of Task 1 has been virtually reduced to that of Task 3.
Three tasks T1, T2 and T3 have decreasing priorities (T1 has the highest priority) and T1 and T3 share some data or resource that require exclusive access, while T2 does not interact with either of the other tasks. Access to the critical section is done through the P (WAIT, PEND) and V (SIGNAL, POST) operations on semaphore S.

Consider the following execution scenario -- the tasks are preemptable and the release times (the time when the tasks start executing) of the 3 tasks are:
- T1: t2; T2: t4; and T3: t0

A priority inversion is said to occur between time interval [t3, t6] during which the highest priority task T1 has been unduly prevented from execution by a medium-priority task.

Note that the blocking of T1 during the periods [t3, t4] and [t5, t6] by T3 which has the lock, is preferable to maintain the integrity of the shared resource while blocking due to T2 is not preferred since it can result in an unbounded or excessive blocking.
PRIORITY INHERITANCE PROTOCOL (PIP)

• The problem of priority inversion in real-time systems has been studied intensively for both fixed-priority and dynamic-priority scheduling.

• **One result is:** the **priority inheritance protocol** that offers a simple solution to the problem of *unbounded priority inversion*.

• In the PIP the priority of tasks are dynamically changed so that the priority of any task in a critical region gets the priority of the highest task pending on that same critical region. In particular when a task T blocks one or more higher-priority tasks, it temporarily inherits the highest priority of the blocked tasks.

**HIGHLIGHTS OF THE PIP**

• The highest-priority task T gives up the processor whenever it seeks to lock the semaphore guarding a critical section that is already locked by some other task.

• If a task T1 is blocked by T2 and , T1 > T2 (i.e., T1 has precedence over task T2), task T2 inherits the priority of T1 as long as it blocks T1.
  - When T2 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 T3 blocks T2, which blocks T1, (with T1 > T2 > T3) then T3 inherits the priority of T1 via T2.
Thus, in Example 2, T3 priority would be temporarily raised to that of T1 at time $t_3$ => the preemption of T3 at $t_4$ by T2 is prevented

At time $t_5$, T3 reverts to its original priority and T2 gets to execute only after T1 completes its computations
uC/OS-II solve the priority inversion problem by providing the mutex objects that implement the PIP protocol.

(1) Task 3 executes. (2) As with Example 1, Task 3 is running but, this time, acquires a mutual exclusion semaphore (mutex) to access a shared resource. (3) - (4) Task 3 accesses the resource and then is preempted by Task 1. (5) – (6) Task 1 executes and tries to obtain the mutex. The kernel sees that Task 3 has the mutex and knows that Task 3 has a lower priority than Task 1. In this case, the kernel raises the priority of Task 3 to the same level as Task 1 (mutex priority which is greater than T1 priority). (7) The kernel places Task 1 in mutex wait list and then resumes execution of Task 3 so that this task can continue with the resource. (8) When Task 3 is done with the resource, it releases the mutex. At this point, the kernel reduces the priority of Task 3 to its original value and looks in the mutex waiting list to see if a task is waiting for th mutex. The kernel sees that Task 1 is waiting and gives it the mutex. (9) Task 1 is now free to access the resource. (10) – (11) When Task 1 is done executing, the medium priority task (i.e. Task 2) gets the CPU. Note that Task 2 could have been ready to run any time between points (3) and (10) without affecting the outcome. Some level of priority inversion cannot be avoided but far less is present than in the previous scenario.
DEADLOCK

• A deadlock also called a deadly embrace, is a situation in which two tasks are each unknowingly waiting for resources held by the other.
  ○ Assume task T1 has exclusive access to resource R1 and task T2 has exclusive access to resource R2. If T1 needs exclusive access to R2 and T2 needs exclusive access to R1, neither task can continue – they are deadlocked.

• (1) The simplest way to avoid deadlock is for tasks to:
  ○ acquire all resources before proceeding,
  ○ acquire the resources in the same order, and release resources in the reverse order. ...Q1?

• Most kernels allow you to specify a timeout when acquiring a semaphore so the deadlock can be broken. ...Q2?

• Q1: How do the techniques presented at (1) solve the deadlock problem?
  A. Tasks don’t lock when they are in the wait state; In the example if we acquire, release resources in order R1, R2. This makes the tasks acquire and release the resources in the same order and as a result the tasks can’t deadlock.

• Q2: How the deadlock is broken by having the timeout specified, and what are the implications of this method?
  A: If the semaphore is not available within a certain amount of time, the task requesting the resource resumes execution. Some form of error code must be returned to the task to notify it that a timeout occurred. A return error code prevents the task from thinking it has obtained the resource. Deadlocks generally occur in multitasking systems, not in embedded systems.
PROBLEMS WITH THE PIP

- PIP does not prevent deadlock. In fact, PIP can cause deadlock or multiple blocking. It also cannot prevent other problems induced by semaphores.
  - Ex. Consider the following sequence with T1 > T2:
    
    T1: Lock S1; Lock S2; Unlock S2; Unlock S1
    T2: Lock S2; Lock S1; Unlock S1; Unlock S2
  - Here two semaphores are used in a nested fashion, but reverse order. Although the deadlock does not depend on the PIP (it is caused by an erroneous use of a semaphore), the PIP does not prevent the problem.

- Priority Ceiling Protocol (PCP) solves some of these problems by imposing a total ordering of the S access.

TRANSITIVE EXAMPLE: the PIP is dynamic and the priority promotion for a task during PIP is transitive but deadlock can take place -- 3 tasks share a common resource

- In the transitive example above deadlock situation can take place -- MP-task can hold some additional resources required by HP-task. HP-task can also acquire some other resources needed by MP before HP-task is blocked.
- When LP task releases the resource and HP task immediately gets to run, it is deadlocked with the MP-task.
- **Therefore, the PIP protocol does not eliminate deadlock.**
PRIORITY CEILING PROTOCOL (PCP)

• 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.
  ○ To achieve this, 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, T, 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 T.

• In the PCP, the priority of every task is known, as are the resources required by every task.

• For a given resource the priority ceiling is the highest priority of all possible tasks that might require the resource.
  ○ Example 1: R is required by 4 tasks (T1 of priority 4, T2 of priority 9, T3 of priority 10 and T4 of priority 8). As a result the priority ceiling of R is 4.

• The current priority ceiling for a running system at any given time is the highest priority ceiling of all of resources in use at that time
  ○ Example 2: A system has 4 resources. R1: PC=4, R2: PC=9, R3: PC=10; R4: PC=8. As a result the current priority ceiling of the system is 4.
PRIORITY CEILING PROTOCOL RULES

• The following rules apply when a task T requests a resource R:
  ○ (1) If R is in use, T is blocked,
  ○ (2) If R is free and if the priority of T is higher than the current priority ceiling, R is allocated to T,
  ○ (3) If the current priority ceiling belongs to one of the resources that T currently holds, R is allocated to T, and otherwise T is blocked,
  ○ (4) The task that blocks T inherits T’s priority if it is higher and executes at this priority until it releases every resource whose priority ceiling is higher than or equal to T’s priority. The task then returns to its previous priority.

• In PCP a requesting task can be blocked for one of 3 causes.
  1. The first cause is when the resource is currently in use, which is direct resource contention blocking, and is the result of rule (1).
  2. The second cause is when the blocking task has inherited a higher priority and its current execution priority is higher than that of the requesting task. This cause is priority inheritance blocking and is the result of rule (4).
  3. A task can be blocked when its priority is lower than the current priority ceiling even when the requested resource is free. This cause is priority ceiling blocking and is a direct consequence of the “otherwise” clause of rule (3). Rule (3) prevents a task from blocking itself if it holds a resource that has defined the current priority ceiling.
PCP CHARACTERISTICS

- The PCP has 3 characteristics:
  1. A requesting task can be blocked by only one task; therefore, the blocking interval is at most the duration of the critical section,
  2. Transitive blocking never occurs under the PCP,
  3. Deadlock never occurs under the PCP.

- One of the deadlock preventions strategies is to impose ordering on the resources.

- The resource ordering can be realized by using the priority ceilings of the resources. Rule (2) says if the priority of T is higher than the current priority ceiling, T does not require any resources that are in use.
  - This issue occurs because otherwise the current priority ceiling would be either equal to or higher than the priority of T, which implies that tasks with a priority higher than T’s do not require the resources currently in use.
  - Consequently, none of the tasks that are holding resources in use can inherit a higher priority, preempt task T, and then request a resource that T holds. This feature prevents the circular-wait condition. This feature is also why deadlock cannot occur when using the PCP as an access control protocol.
  - The same induction process shows that the condition in which a task blocks another task but is in turn blocked by a third task, transitive blocking, does not occur under the PCP.
Scenario:

- T2 executes and holds a lock on S2;
- T1 is initiated:
  - T1 will be blocked from entering S1?
  - A: P(T1) is not strictly greater than the PC(S2) = P(T1);

<table>
<thead>
<tr>
<th>Critical Section</th>
<th>Accessed by</th>
<th>Priority Ceiling</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>T1, T2</td>
<td>P(T1)</td>
</tr>
<tr>
<td>S2</td>
<td>T1, T2, T3</td>
<td>P(T1)</td>
</tr>
<tr>
<td>S3</td>
<td>T3</td>
<td>P(T3)</td>
</tr>
<tr>
<td>S4</td>
<td>T2, T3</td>
<td>P(T2)</td>
</tr>
</tbody>
</table>

T1, T2, T3 (decreasing priorities) with the following sequence of op:

- T1: Lock S1; Unlock S1
- T2: Lock S1; Lock S2; Unlock S2; Unlock S1
- T3: Lock S2; Unlock S2; Semaphores ceiling priorities for S1 and S2 are P(T1) and P(T2), respectively.

- Suppose that T3 starts executing first, locks the semaphore S2 at time t1 and enters the critical section.
- At time t2, T2 starts executing, preempts T3, and attempts to lock semaphore S1 at time t3. At this time, T2 is suspended because its priority is not higher than priority ceiling of semaphore S2 (it is equal only), currently locked by T3.
- Task T3 temporarily inherits the priority of T2 and resumes execution.
- At time t4, T1 enters, preempts T3, and executes until t5, where it tries to lock S1. Note that T1 is allowed to lock S1 at time t5, as its priority is greater than the priority ceiling of all the semaphores currently being locked (in this case, it is compared with S2). Task T1 completes its execution at t6 and lets T3 execute to completion at t7. Task T2 is then allowed to lock S1, and subsequently S2, and completes at t8.