**Exercise 9 - Scheduling**

**Properties**

**Task 1:**

1. **Why do we assign priorities to tasks?** On concurrent (not parallel) systems, a priority scheduling is used to define that some tasks (threads) are more important/critic than others, so the first ones should not be stopped/ delayed by the second ones.
2. **What features must a scheduler have for it to be usable for real-time systems?** From each task/thread, the scheduler should know: the priority level (in case this algorithm works with static priority levels), what kind of shared resource they are going to use, how much time the thread needs to be executed and at what moment they start executing.

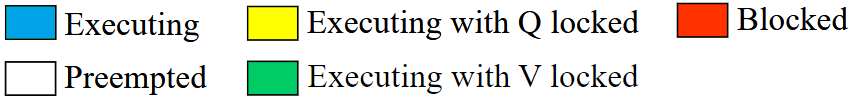
**Inversion and inheritance**

| **Task** | **Priority** | **Execution sequence** | **Release time** |
| --- | --- | --- | --- |
| A | 3 | E Q V E | 4 |
| B | 2 | E V V E E E | 2 |
| C | 1 (lowest) | E Q Q Q E | 0 (in which clock cycle starts) |

* E : Executing (no resource locked)
* Q : Executing with resource Q locked
* V : Executing with resource V locked

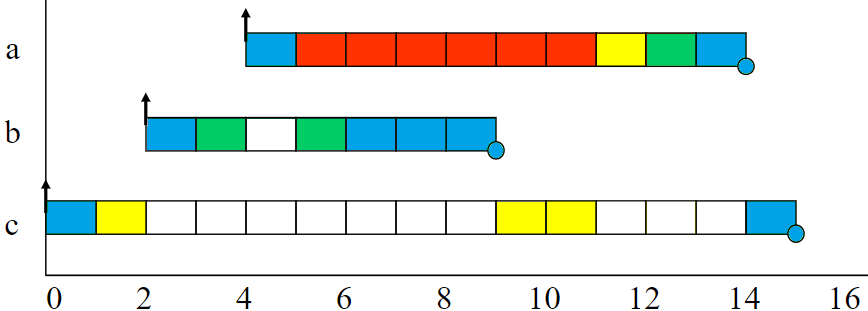
**Task 2: Draw Gantt charts to show how the former task set:**

The tasks stated below in both Gantt charts have the following colour coding:



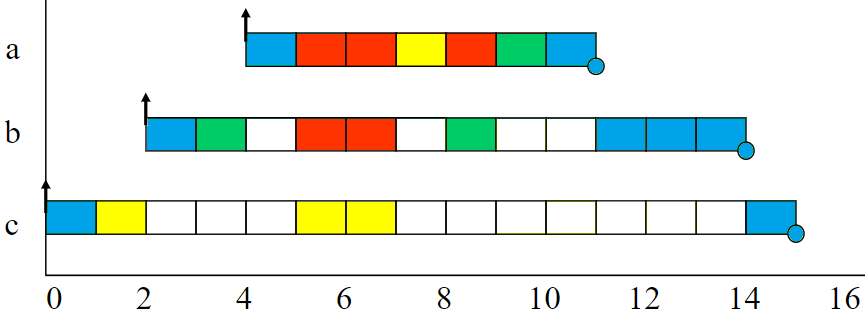
1. **Without priority inheritance**

Without any correction technique, the requested execution sequence presents the priority inversion problem (explained in the next task).



1. **With priority inheritance**

Priority Inheritance Protocol is a technique to manage priority level of different tasks dynamically. This protocol raises the priority of a task, if it holds a shared resource being requested by a higher priority task, to its same priority level.



NOTE: As it can be seen above, lower-priority tasks must wait if the higher-priority ones start executing. However, **in the particular case that a higher-priority task wants to execute the SAME task that a lower priority one is already doing, AND it is a SHARED-RESOURCE task, it will have to wait until finished**.

**Task 3: Explain:**

1. **What is priority inversion?** In a concurrent system with three tasks (A, B and C, being A most important and C least one), in case C starts a process with a shared resource (which means the process must finish before being used by another task), and later A needs to do the same process, A will have to wait. Before that happens, task B wants to do a different process, and as it has more priority than C (and it does not need the same shared resource), task A would stop until B task finishes and then C does so.  
   **What is unbounded priority inversion?** When the priority inversion scenario occurs, it would be more precisely called bounded priority inversion if this situation holds for a determined amount of time. On the contrary, if the amount of time that A will have to wait is undetermined, then the problem is called unbounded priority inversion.
2. **Does priority inheritance avoid deadlocks?** One way to fix the previous situation is with the Priority Inheritance Protocol. Using this protocol, for example, B could hold some resources required by A, and A could also hold some other resources needed by B before A blocks. When C releases the resource and A immediately gets to run, it would be deadlocked with B. Therefore, priority inheritance protocol does NOT eliminate deadlock.

**Utilization and response time**

**Task set 2:**

| **Task** | **Period (T)** | **Exec. Time (C)** |
| --- | --- | --- |
| a | 50 | 15 |
| b | 30 | 10 |
| C | 20 | 5 |

Note: For complete the formulas (below), the total number of tasks is N=3, the current task is “i” and the next one is “j”.

**Task 4:**

1. **There are a number of assumptions/conditions that must be true for the utilization and response time tests to be usable (the "simple task model). What are these assumptions? Comment how realistic they are**

The assumptions that allow to do the utilization and response time test are the following. Most of them are realistic, but some could become difficult.

* All tasks are periodic, with known periods.
* The tasks are completely independent of each other.
* All system's overheads, context-switching times and so on are ignored (i.e, assumed to have zero cost).
* All tasks have a deadline equal to their period (that is, each task must complete before it is next released).
* All tasks have a fixed worst-case execution time.

1. **Perform the utilization test for the task set. Is the task set schedulable?** As seen below, the task set is not schedulable because the utilization test fails.
2. **Perform response-time analysis for the task set. Is the task set schedulable? If you got different results than in 2), explain why.**

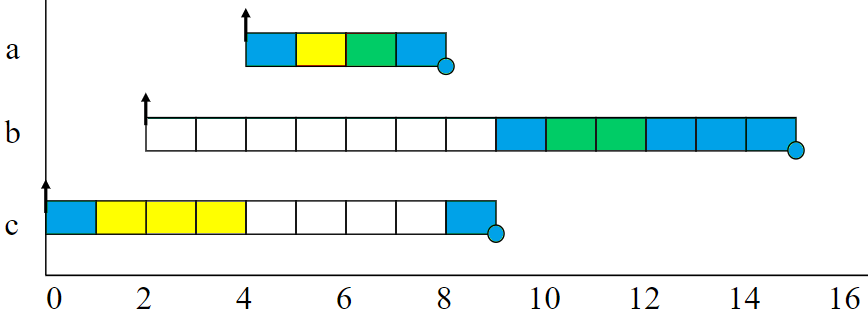
|  |  |  |
| --- | --- | --- |
| **Task c** | **Task b** | **Task a** |
|  |  |  |

From tasks **c**, **b** and **a**, it can be set that the task set is schedulable. The reason why results from task 2 and task 4 are in different is because the utilization test is sufficient, but not necessary. However, the response-time analysis is both sufficient and necessary.

1. **(Optional) Draw a Gantt chart to show how the task set executes using rate monotonic priority assignment, and verify that your conclusions are correct.**

In the Rate Monotonic Priority assignment, each task is assigned a (unique) priority based on its period: the shorter the period, the higher the priority.

In the previous example, assuming all processes last same, the execution sequence determines **a** has highest priority, then **c** and finally **b** the lowest priority. Release time is kept the same. The new Gantt chart would be:



This assignment is optimal in the sense that if any task set can be scheduled (using pre-emptive priority-based scheduling) with a fixed-priority assignment scheme, then the given task set can also be scheduled with a rate monotonic assignment scheme.

**Formulas**

|  |  |
| --- | --- |
| Utilization test:  C:\Users\Cristian\Desktop\eqn-utilization.png | Response-time: C:\Users\Cristian\Desktop\eqn-responsetime.png |