1. **Executive Summary:**

In this project, we tried to implement **Priority Ceiling Protocol** in **Raven OS by** setting ceiling priority value for the shared resources which will be equal to highest priority of all the tasks needing to use that resource. This helps to prevent unbound priority inversions happening in real time system. It also prevents mutual deadlocks happening in the system. It raised priority of tasks when certain condition appears. Thus, it requires a scheduler and the protocol does a dynamic priority scheduling.

The priority ceiling emulation that we had implemented in the previous assignment has the problem that it will delay higher priority process even though there is no conflict. On the other hand, there is chain blocking and Deadlock chances in Disable Pre-emption and Priority inheritance.

Priority ceiling protocol takes better features of these methods and combine them.

* Priority Ceiling Protocol sets ceiling priority for semaphores
* Inheritance of the priority happens only when there is a conflict

The constraint which is imposed is that it should not impose overhead on the system when we implement the protocol. Our design will be focused on reducing the overhead on RavenOS due to the dynamic priority scheduling.

To implement the priority ceiling protocol behaviour, we are using five threads and two semaphores for protecting critical sections. After creating the resources(Semaphores), its ceiling priority value is calculated. Based on this, priorities are assigned dynamically when incoming threads request for the resource. To show the behaviour of priority ceiling protocol, threads are released in increasing order of priority. Our assumption is that once a resource is held by a thread, when a new thread comes to take the resource it will be blocked. The priority of blocked thread will be assigned to the low priority thread. A thread can take a resource only when its priority is greater than the ceiling priority and at it will not access any shared resource. And if a thread which already acquired a resource request a new resource, it will be provided. So once a thread acquires a resource, all the resources needed will be available for it when requested.

1. **Introduction**

Access to resource should be managed by semaphores or monitors to provide mutual exclusion. However, during scheduling there is chances of Unbound Priority Inversion where a high priority thread is blocked from execution by a lower priority task. The good method to solve this issue will be dynamically adjusting priorities. Priorities should be increased when a higher priority thread request for same resource depending on the ceiling priorities and should be reduced while a thread releases a resource.

In priority ceiling protocol, a job can acquire a resource only when its priority is higher than the ceiling value of all other resource which are held. In this way, the blocking is bounded, and no chain blocking occurs. Once a thread gets first resource, it will get all the resources when it requests.

For implementing the protocol, we require a list of threads that a semaphore uses which will help to set the ceiling priorities.

1. **Raven OS**

* main.c – main.c file is modified to create new threads so that we have 5 threads to show the behaviour according to slide 44. We have included additional semaphores to block the execution of remaining threads that have priority higher than thread 5.
* osObject.h – We have added resc\_ceilingS1, resc\_ceilingS2, CSC variables added to store the ceiling priority values. Function prototypes of findmax\_sem1, findmax\_sem2 are added.
* semaphores.c – Following function definitions are added to semaphores.c. findmax\_sem1() – To find ceiling priority of Semaphore 1

findmax\_sem2() – To find ceiling priority of Semaphore 2

change\_task\_priority () – To change the priority of a task when there is chance of unbounded priority inversion.

changetooriginalpriority() – To change the priority of task to its original priority when it releases a semaphore.

* RavenOS.h – To keep track of the threads using a resource we have added variable to **os\_semaphore\_def** so that we can store the list of threads which will use the resource.

When a semaphore is created, we will store the value of the threads which will use it. Maximum priority can be calculated by calling findmax\_sem1() and findmax\_sem2 functions.

1. **Test Application**

For this Project, we have created 5 threads 1, 2, 3, 4 and 5 in the order of decreasing priority. Therefore, thread 1 has the highest priority and thread 5 has the lowest priority. In addition to that, we have also created two semaphores i.e. S1 and S2. S1 is being shared by thread 2, thread 4 and thread 5, while S2 is being shared by thread 4 and thread 1.

To demonstrate the Priority Ceiling Protocol behavior, it is required that thread 5 will be executed first and rest of the threads will wake up after certain period. As we do not have API’s in current RavenOs code to put threads into sleep, we have included additional semaphores to block the execution of remaining threads that have priority higher than thread 5. These high priority threads will be unblocked by thread5 during its execution.

Test application achieves the desired example Startup behavior as follows:

1. We have developed two functions changetaskpriority () which will dynamically change the priority of thread to the priority of the thread requesting the resource and changetooriginalpriority() that will change the thread priority back to normal after releasing the resource. These two functions are called in from Semaphorewait and Semaphorerelease API present in semaphore.c file.
2. To find maximum ceiling values, we are using two function findmax\_sem1() and findmax\_sem2() for Sem1 and Sem2. We are maintaining a list of threads for each semaphore and from that calculate the ceiling values. Once the function call is made, ceiling value of resource S1 is assigned as thread 2 priority and resource S2 is assigned as thread1 priority.

3) Thread 5 would be executed first and acquires the resource S1. It starts executing its critical section.

4) While thread 5 is being carried out, thread 4 will be activated. But thread 4 will not be executed because the ceiling priority of S1 thread 5 is using is priority of thread 2 which is greater than that of thread 5. Thread 5 acquires the priority of thread 4 will be blocked.

5) Thread 3 will be activated now. As it does not require any resource for execution, it will start to execute.

6) Thread 2 will be activated now which will pre-empt thread 3. Since thread 5 is holding S1, thread 2 is blocked and thread 5 acquires priority of thread 2 and resumes execution.

7) Thread 1 will be activated now. It will preempt thread 5 as its priority is higher than the priority of thread 5. Thread 1 will capture and release the resource S2 and complete its execution.

6) After completion of thread1, execution control would jump to thread 5. As soon as thread 5 releases resource S1, its priority will change back to its original priority by the function, changetooriginalpriority ().

7) Control will now be transferred to thread 2 as its priority is higher as compared to thread 3 and thread 4. Thread 2 will complete its execution.

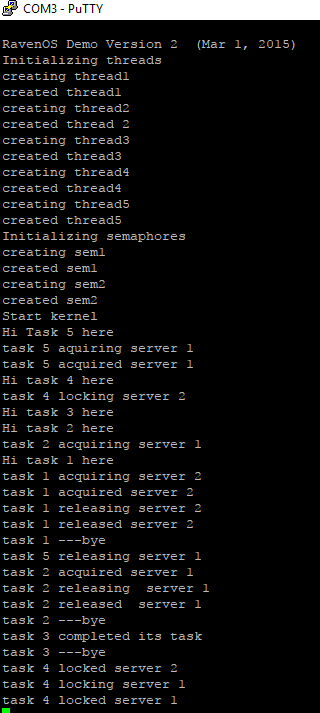
8) Control will now be transferred to thread 3 as the priority of thread 3 is more than the priority of thread 4.

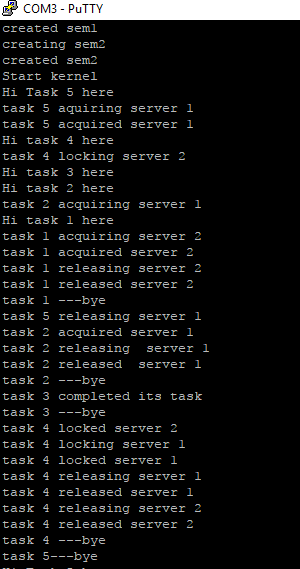
8) As soon as the execution of thread 3 is completed its execution, the control would jump to thread 4. Thread 4 will acquire resource S2 and S1. Thread 4 will release the resources after it completes its execution.

9) Finally, control will be given to thread 5 and it will complete its execution.

1. **Result**

Raven OS was modified and we where able to see the below behaviour.

****

****

1. **Analysis of Overheads**

The general-purpose hardware timer is utilized to accomplish the test objective i.e. to measure the latency of Raven OS for Priority Ceiling Protocol.

For measuring the execution time:

* Find the point in kernel code where thread 5 and thread 1 is going to start and end its wait semaphore services.
* For thread 2 and thread 4 find the point where it tries to acquire semaphore till the time thread 5 resumes execution.
* Timer functions are inserted at the beginning and at the end of each service

1. **Conclusion**

We where able to implement priority ceiling protocol according to slide 44 in Priority Ceiling Protocol lecture. The overhead introduced in the system due to the protocol is analyzed and was found to be small.