Carleton
UNIVERSITY

Department of Systems and Computer Engineering

SYSC 5701 Fall 2017

Operating System Methods for Real-Time Applications

Project Report

By

Group 11

Hemant Gupta (101062246)

Anurag Das (101089268)

Submitted to

Dr. Gregory Franks

Submitted to Dr. Gregory Franks in partial fulfillment of the requirements for the course SYSC 5701

INDEX

Executive Summary

The aim of this project is to implement the Priority Ceiling Protocol (PCP) for the locking and unlocking of resources on RavenOS. The PCP behaviour is achieved by creating a new type of RavenOS objects called Server1 (PCPSemaphore1) and Server2 (PCPSemaphore2). Server1 and Server2 are like the existing RavenOS Semaphores, except that they assume that binary semaphores are being used and has assigned unique identification and priority. Furthermore, they support PCP behaviour associated with PCP_Semaphore_Wait and PCP_Semaphore_Release using PCP allocation and deallocation rules. Hence, locking a resource is done by calling PCP_Semaphore_Wait, and unlocking a resource is done by calling PCP_Semaphore_Release. Test application is developed to get the desired output showing PCP behaviour. Timing analysis of overhead introduced due to new data structures and PCP allocation and deallocation of resources in RavenOS is also done after obtaining the output of test application and comparison is done with normal semaphore behaviour from Assignment 2.

Key Words: Priority Ceiling Protocol, RavenOS, Semaphore, RTOS overhead.

Introduction

In this project we are going to implement Priority Ceiling Protocol and will calculate the overhead between PCP Semaphores and normal binary semaphores.

We modified the semaphore structure i.e. **ocSemaphoreDef_t** so that it can also be used for **PCP semaphore resources (Server 1 and Server 2**) by adding three more variable **resource_id** (unique ID), **priority** (Ceiling Priority of PCP resource) and **thread_info** (Pointer to the thread info holding the PCP resource). We allocate the value of priority and resource id during creation of PCP resources as we already know them.

We have defined a macro i.e. **MAX_PCP_RESOURCE** (current value is 2) it will give us the count of PCP resource going to used in the system. As it's also already mentioned in system.

We also take two global variables. One is a pointer to the current ceiling priority of the system i.e. **CurrentCeilingPCPsemaphore** by using this pointer we can access priority of PCPsemaphore i.e**. current ceiling priority of the system at time t** and its **thread_info gives info about thread holding current ceiling resource**.

Second one is **PCP_resource_list [MAX_PCP_RESOURCE]**. This is array of pointer of active PCP resources. This variable is of static size which is equivalent to max PCP resources. It's a one to one mapping we directly fill the PCP resource address using its id as an method of direct location to avoid overhead of searching every time.

We have define four new function. **First, PCP_allocation** (LOC =11) in this function we implement the rules for PCP semaphore allocation if resource is free. **Second, PCP_deallocation** (LOC=14) in this function we implement deallocation rules like searching new ceiling resource on getting free. **Third, PCP_Semaphore_Wait**(LOC=26) in this function we used to acquire a PCP resource and call PCP allocation if resource is free and if resource not free raise priority of thread holding current ceiling resource and put thread in blocked queue. **Fourth, PCP_Semaphore_Release** (LOC=20) in this function we release the PCP semaphore by calling function PCP_deallocation.

In this Project, we are working with five tasks and two PCP resources such that task5 has the lowest priority and task 1 has the highest in the same order. Two PCP resource has ceiling priority equivalent to maximum priority of task acquiring the resource in our case for Server1 it is Task2 priority (i.e. osPriorityAboveNormal) and for Server 2 it is Task 1 priority (i.e. osPriorityHigh).

We release the task in the order of increasing priority using four binary semaphores which are initialized by 0. Order is Task5 released first and then Task4 and so on till Task1 which is last. We follow basic rule of allocation, if task if free and its priority higher then current ceiling priority at that time then allocate resource or if task free and task asking for resource is the thread holding the current ceiling priority resource then also assign the resource. Otherwise, raise the priority of task holding ceiling resource to the task asking for resource.

We are also going to check the overhead taken by PCP resource in allocation and deallocation and in case of context switch during wait and release and going to compare it with the binary semaphore overhead calculation.

RavenOS: Modifications

Threads Info:

Thread 1: Priority = osPriorityHigh i.e. 2

Thread 2: Priority = osPriorityAboveNormal i.e. 1

Thread 3: Priority = osPriorityNormal i.e. 0

Thread 4: Priority = osPriorityBelowNormal i.e. -1

Thread 5: Priority = osPriorityLow i.e. -2

Semaphore Info:

Sem1:- Used to block Thread 1

Sem2:- Used to block Thread 2

Sem3:- Used to block Thread 3

Sem4:- Used to block Thread 4

PCP Resource Info:

Server 1:- Ceiling Priority = Max(Priority Thread 2, Priority Thread 5, Priority Thread 4)

= Priority Thread 2 = osPriorityAboveNormal i.e. 1

Server 1:- Ceiling Priority = Max(Priority Thread 1, Priority Thread 4)

= Priority Thread 1 = osPriorityHigh i.e. 2

Structure Modified:

File Name: RavenOS.h

osSemaphoreDef_t - Semaphore Definition structure contains setup information for a semaphore.

Three new field added:-

a. **Priority**: Its gives the Ceiling priority of the PCP resource. Not used in case of basic semaphore. Filled at the time of PCP resource creation.

b. **Thread_info**: Pointer of the thread holding the resource. Updated when thread acquire a Resource. And Assign to NULL when thread release the resource.

c. **Resource_id**: Unique Identification for the PCP resource. Assigned at the time of resource creation.

```
/// Semaphore Definition structure contains setup information for a semaphore.
/// \note CAN BE CHANGED: \b os_semaphore_def is implementation specific in every CMSIS-RTOS.
typedef struct os_semaphore_def  {
  // def structure modified to eliminate additional scb ... memory management now done in application
  os_tcb_p                  blocked_q_h;   ///< head pointer for list of threads blocked on this semaphore
  uint32_t                  blocked_q_cnt; ///< indicated how many threads are blocked on this semaphore
  int32_t                   ownCount;      ///< number of tokens for this semaphore
  osPriority                priority;      ///Added- <priority of PCP resource
  osThreadId                thread_info;   ///Added- <Info of thread holding the resource
  uint32_t                  resource_id;   ///Added- <unique Id of the resource
  osSemaphoreId             next;          ///< link into semaphore list
} osSemaphoreDef_t;
```

Macro Added:

File Name: semaphores.c

PCP_MAX_RESOURCE: Value of Max PCP resource in system. Update according to number of resources.

```
//=========MACRO================
///MAX number of PCP Semaphore Resources
/// Must be updated for more Servers
#define PCP_MAX_RESOURCE  2
```

Global Variables:

File Name: semaphores.c

Two global variables are defined whose scope are within the file semaphore.c file and cannot be accessed beyond that.

a. **CurrentCeilingPCPsemaphore**: Pointer to the Current Ceiling PCP resourse of the system at time T. Updated when every time a task tries to acquire a PCP resource or releases a PCP resource. Default initialize it with NULL. Its Priority is current ceiling priority of system and its thread_info gives info about thread holding current ceiling resource.

b. **PCP_resource_list[MAX_RESOURCE]:** List of the Active PCP resources which are currently in locked state (i.e. Not Free). It is a static list and one to one map with the PCP resource id.

```
//pointer to current PCP ceiling resource.Its Priority is current ceiling priority of
// of system at time t and its thread_info gives info about thread holding current ceiling resource.
osSemaphoreId CurrentCeilingPCPsemaphore = NULL;
osSemaphoreId PCP_resource_list[PCP_MAX_RESOURCE]={0}; //Active PCP resource List
```

Function Definitions: **File Name:- Semaphore.c**

Added 4 new functions to implement the PCP allocation and deallocation rules.

1.PCP allocation (LOC =11): Rules for the Assignment of the PCP resource are present in this function.This function get called from PCP_Semaphore_Wait when resource is free i.e. available. If when current thread priority higher then ceiling priority of system or when current thread asking for resource is the thread holding ceiling priority of system resource then

A. Store thread info in the semaphore structure(thread_info)
B. update CurrentCeilingPCPSemaphore which give current ceiling priority of system.
C. Store PCP resource info in the Active PCP resource list.
D. Decrease PCP resource count by 1.

Else return resource cannot be allocation.

```
/// Priority Ceiling Protocol Allocation Rules when resource is free.
/// \param[in]     semaphore_id  semaphore object referenced with \ref osSemaphoreCreate.
/// \param[in]     millisec      timeout value or 0 in case of not willing to wait.
/// \return number of available tokens, or -1 in case of incorrect parameters.
/// \note MUST REMAIN UNCHANGED: \b osSemaphoreWait shall be consistent in every CMSIS-RTOS.
int32_t PCP_allocation (osSemaphoreId semaphore_id)
{
  osThreadId curr_th = osThreadGetId(); //Current thread in execution
  //when current thread priority higher then ceiling priority of system or
  //when current thread is the thread holding ceiling priority of system resource then
  if ((CurrentCeilingPCPsemaphore == NULL) || (curr_th->priority > CurrentCeilingPCPsemaphore->priority) ||
      (CurrentCeilingPCPsemaphore->thread_info == curr_th))
  {
    // update ceiling Current Ceiling PCP resource of system if priority is greater
    if ((CurrentCeilingPCPsemaphore == NULL) ||(semaphore_id->priority > CurrentCeilingPCPsemaphore->priority))
    {
      CurrentCeilingPCPsemaphore = semaphore_id;
    }
    //Store thread info in the semaphore structure
     semaphore_id->thread_info =  curr_th;
     //store resource info in the Active PCP resource list
     PCP_resource_list[semaphore_id->resource_id - 1] =  semaphore_id;
     // semaphore is free -> take semaphore
     // TWP the returned value is not really reliable by the time the released thread runs
     --(semaphore_id->ownCount);
    return (semaphore_id->ownCount);
  }
   //otherwise return No resource will be allocated
   return (-1);
}
```

2. **PCP_deallocation(LOC =14):** Rules for the releasing of the PCP resource are present in this function. This function called from PCP_Semaphore_Release.

A. Increase the PCP resource count by one.
B. Update the priority of the thread releasing the resource to original priority.
C. Clear the resource info from the Active PCP resource list.
D. Update the CurrentCeilingPCPSemaphore info by choosing the next available highest priority resource from Active PCP resource list if PCP resource for release equal to current ceiling PCP resource.
E. Clear the thread info from the semaphore structure.

```
////// Priority Ceiling Protocol Deallocation Rule.
/// \param[in]     semaphore_id  semaphore object referenced with \ref osSemaphoreCreate.
/// \return status code that indicates the execution status of the function.
/// \note MUST REMAIN UNCHANGED: \b PCP_deallocation shall be consistent in every CMSIS-RTOS.
void PCP_deallocation(osSemaphoreId semaphore_id)
{
  osThreadId curr_th = osThreadGetId();//current thread info
  uint32_t cnt = 0, id = 0; //Maintain count and resource id
  osPriority max_priority = osPriorityIdle;//used to search highest priority ceiling resource
  semaphore_id->ownCount++;  // increment semaphore's current ownCount
  //updating Priority of current thread to its original priority
  curr_th->priority = curr_th->tpriority;
  semaphore_id->thread_info = NULL;
  //Clearing the resource from the Active PCP resource List
    PCP_resource_list[semaphore_id->resource_id - 1] =  NULL;
  //if the PCP Resource is equal to current ceiling priority resource
  if(CurrentCeilingPCPsemaphore == semaphore_id)
  {
    CurrentCeilingPCPsemaphore = NULL; //CLearing Current PCP Resource info
    //Now locating the id of Active PCP semaphore which has highest ceiling peiority
    for(cnt=0; cnt < PCP_MAX_RESOURCE; cnt++)
    { //if list is not NULL and its priority higher then max_priority
      if ((PCP_resource_list[cnt]!= NULL) && (PCP_resource_list[cnt]->priority > max_priority))
      {
        id = cnt; //storing resource id
        max_priority = PCP_resource_list[cnt]->priority; //updating max_priority
      }
    }
    CurrentCeilingPCPsemaphore = PCP_resource_list[id]; //update Current Ceiling Priority Resource
  }
}
```

3.PCP_Semaphore_Wait (LOC = 26):- Function to lock a PCP resource.

A. When resource is free call **function PCP_Allocation** for PCP rules for resource allocation.
B. If PCP resource allocation return error or if resource is not available
   Then raise the priority of the thread holding the ceiling resource of the system to the Current thread asking for the resource.
C. Put the thread in blocking queue and invoke the scheduler.
D. In case once resource get freed, call PCP resource allocation for assigning the resource with PCP rules.

```
/// Wait until a PCP Semaphore token becomes available.
/// \param[in]     semaphore_id  semaphore object referenced with \ref osSemaphoreCreate.
/// \param[in]     millisec      timeout value or 0 in case of not willing to wait.
/// \return number of available tokens, or -1 in case of incorrect parameters.
/// \note MUST REMAIN UNCHANGED: \b osSemaphoreWait shall be consistent in every CMSIS-RTOS.
int32_t PCP_Semaphore_Wait (osSemaphoreId semaphore_id, uint32_t millisec)
{
  osThreadId curr_th = osThreadGetId();
  int32_t retVal = 0;
  // semaphore does not exist
  if ( semaphore_id == NULL )
  {
    return (-1);
  }
```

```
    //PCP resource is free
    if ( semaphore_id->ownCount  > 0 )
    {
      //Implementation of PCP Allocation Rules
      retVal = PCP_allocation(semaphore_id);
      if(-1 != retVal)
      {
        return(retVal);
      }
    }
    //Updating the priority of thread holding the resource with ceiling
    //priority when resource is not available or thread has lower priority
    //then the ceiling priority of system.
    if(CurrentCeilingPCPsemaphore->thread_info->priority < curr_th->priority)
    {
      CurrentCeilingPCPsemaphore->thread_info->priority = curr_th->priority;
      osRemoveRTRQ (CurrentCeilingPCPsemaphore->thread_info);
      osInsertRTRQ (CurrentCeilingPCPsemaphore->thread_info );
    }
  // if still here, then semaphore locked ... need to block calling thread!
  // add thread to the semaphore queue, mark thread as blocked and invoke scheduler
  // set thread to blocked state and manipulate RTRQ and blocked Q before yield
  curr_th->status = TH_BLOCKED;
  // remove from RTRQ
  osRemoveRTRQ( curr_task );
  // put in blocked Q
    if (os_InsertThreadInSemaphoreBlockedQ(curr_th, semaphore_id, osWaitForever, osWaitForever) != osOK)
    {
      return (-1); // something went wrong and cannot block on this semaphore   TWP :-(   whaaatt?
    }
  //have modified RTRQ and semaphore blocked Q
  // curr_task is no longer at head of RTRQ ... invoke scheduler to do context switch
  os_KernelInvokeScheduler ();
  // will eventually resume here when thread released from semaphore ...
  if ( semaphore_id->ownCount  <= 0 )
  {
     return (0);
  }
  else
  {    //Implementation of PCP Allocation Rules when Server is free
       retVal = PCP_allocation(semaphore_id);
       if(-1 != retVal)
       {
         return(retVal);
       }
       else
       {
         return(0);
       }
  }
}
```

4.PCP_Semaphore_Release(LOC =20):- Function to release the locked PCP semaphore resource.

A. Call PCP deallocation function to apply all PCP deallocation rules.
B. Unblock the thread with highest priority in the semaphore blocked queue.
C. Put the thread in ready to run queue
D. Invoke the scheduler.

```
/// Release a PCP Resource token.
/// \param[in]     semaphore_id  semaphore object referenced with \ref osSemaphoreCreate.
/// \return status code that indicates the execution status of the function.
/// \note MUST REMAIN UNCHANGED: \b osSemaphoreRelease shall be consistent in every CMSIS-RTOS.
osStatus PCP_Semaphore_Release (osSemaphoreId semaphore_id)
{
  osThreadId th_p;    // Active if releasing a thread
  osThreadId curr_th = osThreadGetId();
  if ( semaphore_id == NULL )
  { // semaphore does not exist
    return osErrorParameter;
  }
  PCP_deallocation(semaphore_id);   //PCP Deallocation Rules
  //Schedule Decision the is made due to change in priority of process
  osRemoveRTRQ (curr_th);
  osInsertRTRQ( curr_th );

  // any threads blocked on semaphore?
  if (semaphore_id->blocked_q_cnt == 0)
  {   // EASY case:  no threads blocked on this semaphore
    return osOK;
  }
  // unblock the thread with highest priority waiting in the semaphore queue
  th_p = osRemoveFromHeadOfBlockedQ( semaphore_id );
  // change state to RUNNING
  //TWPV6: removed TH_READY state ... only using TH_RUNNING
  th_p->status = TH_RUNNING;
  // put the thread in the RTRQ
  osInsertRTRQ( th_p  );
  // RTRQ has changed - invoke scheduler for context switch if necessary
  os_KernelInvokeScheduler ();
  return osOK;
}
```

New Added Files

File:- PCP_resource1.c **Function Name: Init_PCP_Semaphore1**

Update resource id: 1 **Priority: osPriorityAboveNormal (i.e. 1)**

File:- PCP_resource2.c **Function Name: Init_PCP_Semaphore2**

Update resource id: 2 **Priority: osPriorityHigh (i.e. 2)**

Both files contain the function for creating and deleting the PCP resource. During initialization we assign the resource priority (i.e. priority) and unique id (i.e. resource_id) to the PCP Resource Semaphore 1 (Server 1) and PCP Semaphore 2 (Server2). We create the PCP Resource as binary semaphore.

We create the PCP Resource as binary semaphore.

```
int Init_PCP_Semaphore1 (void)
{  //TWPV1 create with count = 0
  Server1 = osSemaphoreCreate(osSemaphore(PCP_Semaphore1), 1);
  if (!Server1)  {
    return (-1);
  }
  //Setting the Server1 Priority
  Server1->priority = osPriorityAboveNormal;
  //Setting unique id for server
  Server1->resource_id = 1;
  printf("created pce_Sem1\n\r");
  return(0);
}
```

```
int Init_PCP_Semaphore2 (void)
{  //TWPV1 create with count = 0
  Server2 = osSemaphoreCreate(osSemaphore(PCP_Semaphore2), 1);
  if (!Server2)  {
    return (-1);
  }
  //Setting the Server2 Priority
  Server2->priority = osPriorityHigh;
  // Setting Server 2 Unique ID
  Server2->resource_id = 2;
  printf("created Server2\n\r");
  return(0);
}
```

Test Application

Client-Server Model:

- Server1 Ceiling Priority = Task 2 Priority (i.e. Above Normal = 1)
- Server 2 Ceiling Priority = Task 1 Priority (i.e. High = 2)
- Task 1 (Priority = High = 2) locks Server 2
- Task 2 (Priority = Above Normal = 1) locks Server 1
- Task 3 (Priority = Normal = 0) did not lock any Server.
- Task 4 (Priority = Below Normal = -1) lock Server 2 and then Server 1 and release Server 1 and then Server 2.
- Task 5(Priority = Low = -2) lock Server 1.
- Task 1 has the highest priority and Task 5 has the lowest priority.

Assumption:

- Task 5 will execute first, to achieve this we are using 4 different semaphore (initialize with count 0) to block the 4 high priority tasks then Task 5 to release task in order of Task5 to Task 1.

PCP Resource Info:

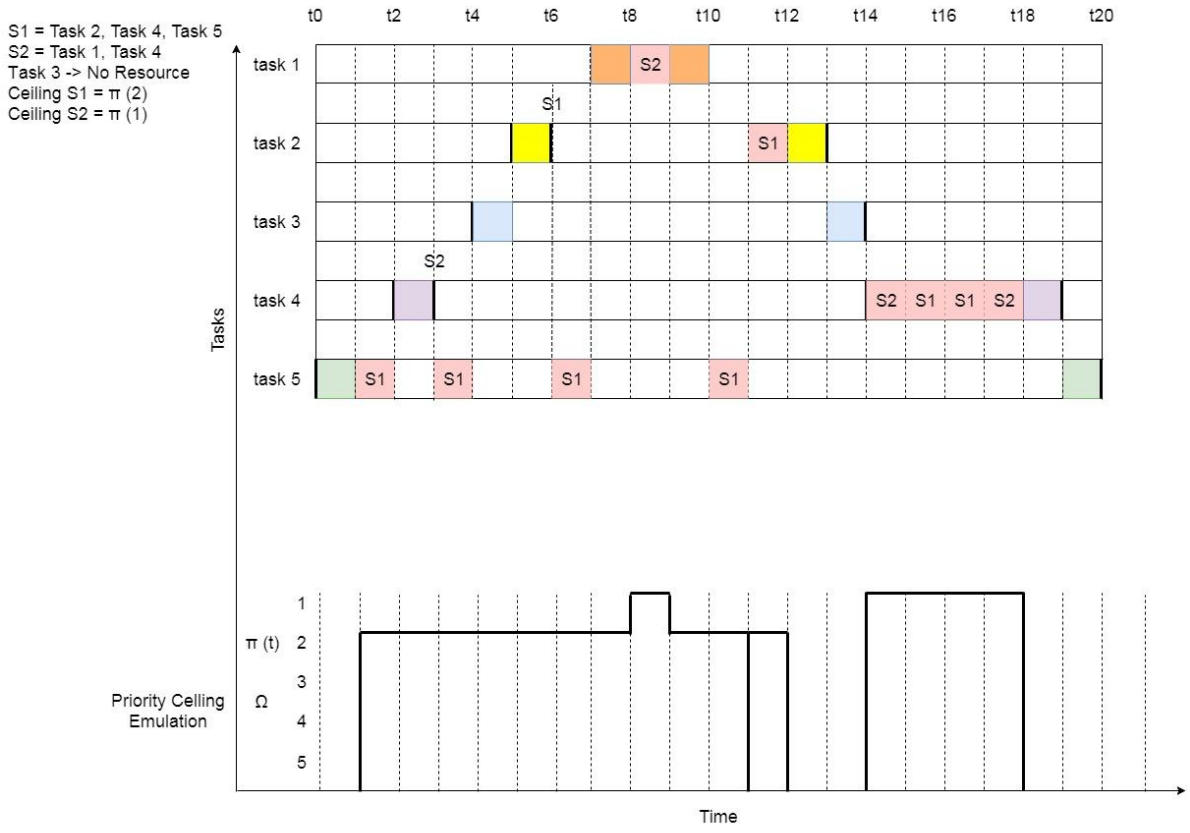
Server 1:- Ceiling Priority = Max(Priority Thread 2, Priority Thread 5, Priority Thread 4)

= Priority Thread 2 = osPriorityAboveNormal i.e. 1

Server 1:- Ceiling Priority = Max(Priority Thread 1, Priority Thread 4)\

= Priority Thread 1 = osPriorityHigh i.e. 2

Priority Ceiling Protocol-Timing Diagram:

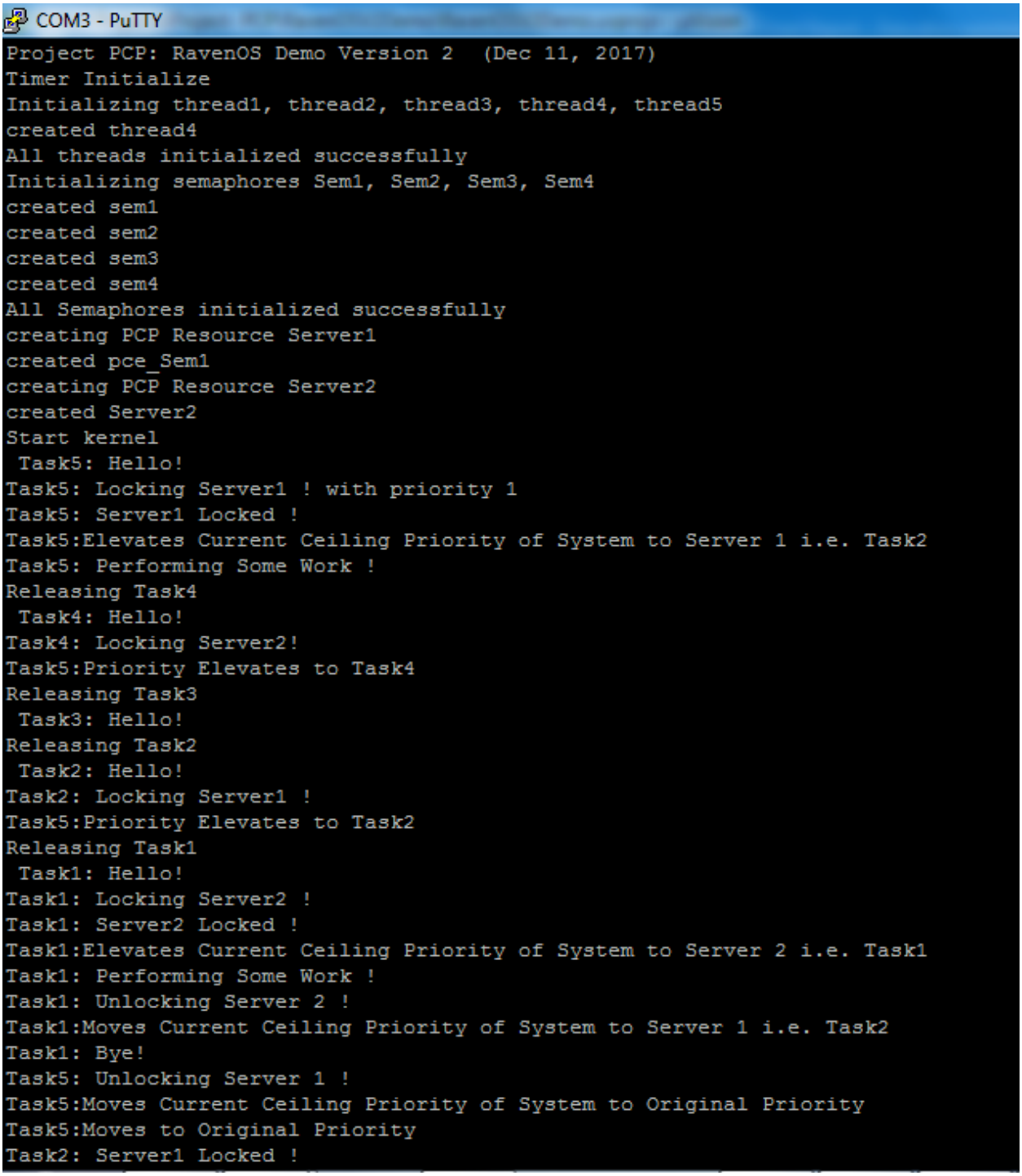


Execution steps according to Timing Diagram:

1. At t0, Task 5 starts execution. And at t1, Task5 tried to lock Server 1. and locked Server 1 and kernel elevates Current Ceiling Priority of System to Server1 i.e. Task2
2. At t2, now release Task4(Blocked on Semaphore4)
3. At t3, Task4 tries to lock Server 2 but unable to lock because Task4 has priority lower then current ceiling priority of the system. Kernel raise Task5 priority to Task4 and Task5 start execution.
4. At t4, Task3(Blocked on Semaphore3) releases from Task5.
5. At t5, now release Task2(Blocked on Semaphore2) from Task 3.
6. At t6, Task2 tries to lock Server 1 but directly blocked by Task 5. Kernel now raise Task5 priority equivalent to Task2.
7. At t7, now release Task 1(Blocked on Semaphore 1) from Task5, Kernel schedule this Task 1 as it has higher priority (i.e. High) then the current ceiling priority of system (i.e. Above Normal) at t5. Task 1 start running.
8. At t8, Task 1 tries to lock Server 2. As Server 2 (Priority = High) is free resource, therefore Task1 locked Server 1 and ceiling priority of system changes to High (i.e. Priority of Server 2) by the kernel.
9. At t9, Task1 releases Server 2. Now kernel update the current ceiling priority of system to ceiling priority of Server1 locked by Task5(i.e. Above Normal).
10. At t10, Task1 complete its execution. Now kernel schedule next high priority task in the queue i.e. Task 5 start execution.
11. At t11, Task5 releases resource Server 1. And all resources are free. Now kernel changes current ceiling priority of system to idle as no resources are locked and also priority of Task5 move to its original priority (i.e. Low) and kernel schedule next high priority task, Task2.
12. At t11, Task2, locked resource Server 1 as its free and kernel moves current ceiling priority of system to Above Normal.
13. At t12, Task 2 releases Server 1. Now kernel updates current ceiling priority of system to original priority.
14. At t13, Task2 complete its execution. Now kernel schedule next high priority task in the queue i.e. Task 3 start execution.
15. At t14, Task3 complete its execution. Now kernel schedule next high priority task in the queue i.e. Task 4 start execution.
16. At t14, Task4 tried to lock Server 2. As Server2 is free resource, and Task4 has higher priority than current ceiling priority of system at t15, then Task 4 locked the Server 1. And kernel updates ceiling priority of system to High.
17. At t15, Now Task4 tried to lock Server 1. As Server1 is free resource, and Task4 has priority lower then current ceiling priority of system but it is the current task holding the other resource (i.e. Server 2) therefore Task 4 locked the Server 1.
18. At t17, Task 4 releases Server 1.
19. At t18, Task 4 releases Server 2. And all resources are free. And kernel updates current ceiling priority of system moves to idle.
20. At t19, Task4 complete its execution. Now kernel schedule next high priority task in the queue i.e. Task 5 start execution.
21. At t20, Task 5 completes its execution.

Results

Priority Ceiling Protocol-Console Output:

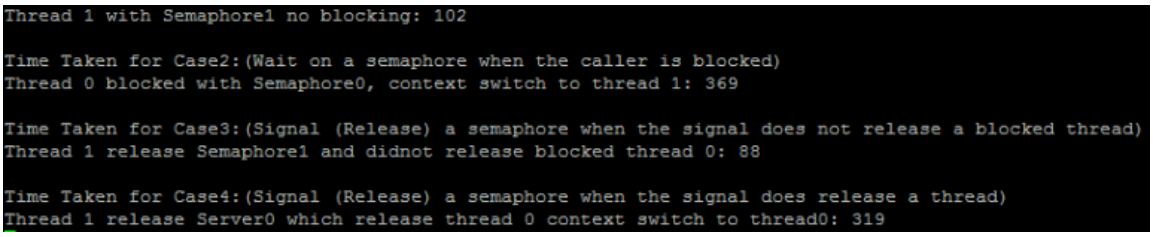


```
Task2:Elevates Current Ceiling Priority of System to Server 1 i.e. Task2
Task2: Performing Some Work !
Task2: Unlocking Server 1 !
Task2:Moves Current Ceiling Priority of System to Original Priority
Task2: Bye!
Task3: Performing Some Work !
Task3: Bye!
Task4: Server2 Locked !
Task4:Elevates Current Ceiling Priority of System to Server 2 i.e. Task1
Task4: Performing Some Work !
Task4: Locking Server1!
Task4: Server1 Locked !
Task4: Performing Some Work !
Task4: Unlocking Server 1 !
Task4: Unlocking Server 2 !
Task4:Moves Current Ceiling Priority of System to Original Priority
Task4: Bye!
Task5: Bye!

Time Taken for Case1:(Wait on a PCP semaphore when the caller is not blocked)
Thread 5 with Server1 no blocking: 141

Time Taken for Case2:(Wait on a semaphore when the caller is blocked)
Thread 2 blocked with Server1, context switch to thread 5: 512

Time Taken for Case3:(Signal (Release) a semaphore when the signal does not release a blocked thread)
Thread 2 release Server1 and didnot release blocked thread 4: 242

Time Taken for Case4:(Signal (Release) a semaphore when the signal does release a thread)
Thread 5 release Server1 which release thread 2 context switch to thread2: 639

Time Taken for Case5:(Wait on semaphore when caller is block even when resource is available)
Thread 4 blocked with Server2, thread 4 context switch to thread5: 553
```

From the results we can observe the ideal PCP implementation.

Analysis RTOS Overhead Study

Project timing output is in result section and Assignment 2 Output below:



- **Case 1: Wait on a semaphore when caller is not blocked**

Reason: Time Taken Normal Semaphore = 102 Ticks Time Taken PCP Semaphore = 141 ticks

In case of normal semaphore allocation if resource is available we allocate the resource. On the other hand in PCP, we need to check the allocation rule whether thread priority is higher then ceiling priority or not. And after that we need to update the CurrentCeilingPCPsemaphore and store thread info in PCP resource structure and store the PCP resource in Used PCP resource list. Then return the PCP semaphore successfully.

- **Case 2: Wait on a semaphore when caller is blocked**

Reason: Time Taken Normal Semaphore = 369 Ticks Time Taken PCP Semaphore = 512 ticks

In case of normal semaphore allocation if resource is not available we update blocked semaphore queue and put the thread in blocked queue and invoke scheduler which cause context switch. On the other hand, in case of PCP we have perform few more extra tasks which can be contained in two scenarios. If the resource is not available, then we have to increase the priority of thread holding ceiling resource to the thread asking for resource. And second case even when resource is available then we need to check whether it fulfill PCP resource allocation rules or not. If not then to increase the priority of thread holding ceiling resource to the thread asking for resource then put current thread in blocked queue and invoke scheduler. In case of PCP we have to perform extra task therefore timing is high.

- **Case 3: Signal a semaphore when signal did not release a blocked task**

Reason: Time Taken Normal Semaphore = 88 Ticks Time Taken PCP Semaphore = 242 ticks

In case of normal semaphore release we increase semaphore count and check if there is any thread blocked on semaphore in this case “NO” and function continue. On the other hand in case of PCP we have to do same task as normal semaphore but extra task that we perform is we need to modify the used PCP semaphore resource list. And then we need to search the next highest ceiling priority in the resource list and update the structure of CurrentCeilingPCPsemaphore and update priority of current thread to original priority. This is an extra task and there for cause extra number of ticks.

- **Case 4: Signal a semaphore when signal release a blocked task**

Reason: Time Taken Normal Semaphore = 319 Ticks Time Taken PCP Semaphore = 639 ticks

In case of normal semaphore release we increase semaphore count and check if there is any thread blocked on semaphore in this case “YES”. Then we invoke the scheduler which causes context switch. On the other hand in case of PCP we have to do same task as normal semaphore but extra task that we perform is we need to modify the used PCP semaphore resource list. And then we need to search the next highest ceiling priority in the resource list and update the structure of CurrentCeilingPCPsemaphore and update priority of current thread to original priority. This is an extra task and there for cause extra number of ticks

- **Case 5: Wait on a semaphore when caller is blocked even when resource is available**

Reason: Time Taken Normal Semaphore = Not Applicable Time Taken PCP Semaphore = 553 ticks

Case is not applicable for normal semaphore allocation. On the other hand, in case of PCP we have perform few more extra tasks which can be contained in two scenarios. If the resource is available, then we check the thread priority is higher then current ceiling priority or thread asking for resource is the one holding current ceiling priority resource. If both cases fail then we increase the priority of thread holding ceiling resource to the thread asking for resource. If not then to increase the priority of thread holding current ceiling resource to the thread asking for resource then put current thread in blocked queue and invoke scheduler. In case of PCP we perform extra tasks therefore timing is high.

Conclusions

PCP was successfully implemented between five tasks sharing two servers of varying priorities. Both allocation and deallocation rules are working fine. It can easily be enhance for more servers by modifying macro MAX_PCP_RESOURCE value in semaphores.c file. PCP semaphore functionality is achieved by adding few new variables and updating semaphore structure and a code change of 70 LOC. We added two functions PCP_allocationa nd PCP_deallocation which contains rules of PCP resource handing. And these functions called from PCP_Semaphore_Wait and PCP_Semaphore_Release which work on basic semaphores rules and maintain blocked and ready to run queue.

Overhead, in case of normal semaphore allocation and deallocation we simply check whether resource is available or not if yes then allocate it if not then blocked the thread. And in case of deallocation we increase the count and invoke the scheduler to run the task in ready to run queue.

On the other hand, in case of PCP semaphore allocation we have perform few more extra tasks which can be contained in three scenarios. First if the resource is not available, we need to update the Current Ceiling PCP semaphore info and store thread info in PCP resource structure and store the PCP resource in active PCP resource list. Then return the PCP semaphore successfully. Second, if resource is not available then we increase the priority of thread holding ceiling resource to the thread asking for resource. And third case is only applicable for PCP semaphore even when resource is available then we need to check whether it fulfill PCP resource allocation rules i.e. thread asking for resource has priority higher then current ceiling priority of system or is the same thread holding the resource. If not then to increase the priority of thread holding ceiling resource to the thread asking for resource then put current thread in blocked queue and invoke scheduler.

In case of PCP resource release we have to perform extra task therefore timing is high. During deallocation, we need to modify the used PCP semaphore resource list. And then we need to search the next highest ceiling priority in the resource list and update the structure of CurrentCeilingPCPsemaphore and update priority of current thread to original priority. This extra task cause extra number of ticks.

Due to these extra validations and maintaining of resources PCP has higher overhead then normal Semaphores. But it has much more benefits that we conclude from our project that

1. No Deadlock in PCP.
2. No chain blocking.
3. Task blocks in at most one critical section.
4. And once acquire a resource, all resource will be available when requested.