B10332024 四電資四 江承祐

PART 1 NPCS Implementation

1. Implementation Description:

The main objective of this project is to implement non-preemptible critical section within our earliest deadline first (EDF) scheduler. After observing the structure of this code, I decided to implement my code within ***OSMutexPend and OSMutexPost.*** Both of these functions are responsible for deciding which resources to use, so I decided to rewrite the code within this function.

```
void  OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *perr)
{
    INT8U      pip;                                /* Priority Inheritance Priority (PIP)      */
    INT8U      mprio;                              /* Mutex owner priority                     */
    BOOLEAN    rdy;                                /* Flag indicating task was ready           */
    OS_TCB    *ptcb;
    OS_EVENT  *pevent2;
    INT8U      y;
#if OS_CRITICAL_METHOD == 3                        /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr = 0;
#endif


INT8U  OSMutexPost (OS_EVENT *pevent)
{
    INT8U      pip;                                /* Priority inheritance priority            */
    INT8U      prio;
#if OS_CRITICAL_METHOD == 3                        /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr = 0;
#endif
```

(1) Non-Premptible Section

In a non-preemptible critical section implementation, the scheduler never allows a context switch whenever there are resources in use. To implement this, I decided to use the function OSSchedLock to control when to disable context switches.

I implemented OSSchedLock() within OSMutexPend and OSSchedUnlock() within OSMutexPost. By doing this I disable context switches whenever we grab a resource and enable context switches again whenever we let go of the resource.

OSMutexPend:

```
                                                           /* Is Mutex available?                      */
    if ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
        pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;       /* Yes, Acquire the resource                */
        pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;         /*      Save priority of owning task        */
        pevent->OSEventPtr  = (void *)OSTCBCur;            /*      Point to owning task's OS_TCB       */
        OSSchedLock();
        if (OSTCBCur->OSTCBPrio <= pip) {                  /*      PIP 'must' have a SMALLER prio ...  */
            OS_EXIT_CRITICAL();                            /*      ... than current task!              */
            *perr = OS_ERR_PIP_LOWER;
        } else {
            OS_EXIT_CRITICAL();
            *perr = OS_ERR_NONE;
        }
        return;
    }
```

OSMutexPost:

```
            return (OS_ERR_PIP_LOWER);
        } else {
            OS_EXIT_CRITICAL();
            OS_Sched();                          /*      Find highest priority task ready to run  */
            return (OS_ERR_NONE);
        }
    }
    OSSchedUnlock();
    pevent->OSEventCnt |= OS_MUTEX_AVAILABLE;    /* No,  Mutex is now available                   */
    pevent->OSEventPtr  = (void *)0;
    OS_EXIT_CRITICAL();
    return (OS_ERR_NONE);
}
/*$PAGE*/
/*
```

(2) Task Simulation

The tasks we are trying to simulate can be divided into four main parts:

(a) Task Awaiting time (Arrival Time)
(b) Task Computation Time (CPU running time)
(c) Task using Resource 1 time
(d) Task using Resource 2 time

a: We simulate the task awaiting time by delaying the task briefly before it starts running the infinity while loop. After the task's arrival time is up, we start running the task periodically.

b: We simulate the CPU running time the same way we simulated task computing in our previous projects. We put the remaining time in another while loop and subtract from the OSTCBCur->RemainingTime within OSTimeTick to simulate task computing.

c & d: Because our tasks cannot be preempted when we are using resources, we do not have to simulate the task the same way as our CPU running time. Instead, we implemented a "wait" function within the application layer to simulate waiting for a certain amount of time.

```
void task2(void* pdata)
{
  INT8U prio=TASK2_PRIORITY;
  OS_TCB *ptcb= OSTCBPrioTbl[prio];
  ptcb->REMAINING_TIME     = ptcb->compute;
  INT8U err;
  OSTimeDly(4);                  (a) Task Awaiting Time (Arrival Time)
  while (1)
  {

   ptcb->TASK_ACTUAL_START_TIME = OSTimeGet();
    printf("Time %d\t\t Task 2\n",OSTime);
    while( 0 < ptcb->REMAINING_TIME){
                                                  (b) Task Computation Time (CPU running time)
    }
    printf("Time %d\t\t Task 2 get R2\n",OSTime);
    OSMutexPend(R2,0,&err);                          (d) Task Using Resource 2 Time
    wait(2);
    printf("Time %d\t\t Task 2 get R1\n",OSTime);
    OSMutexPend(R1,0,&err);                          (c) Task Using Resource 1 Time
    wait(4);
    printf("Time %d\t\t Task 2 release R1\n",OSTime);
    OSMutexPost(R1);
    printf("Time %d\t\t Task 2 release R2\n",OSTime);
    OSMutexPost(R2);

    ptcb->RESPONSE_TIME = OSTimeGet() - ptcb->TASK_SHOULD_START_TIME;
    int todelay = ptcb->period - ptcb->RESPONSE_TIME;
    ptcb->TASK_SHOULD_START_TIME = ptcb->period + ptcb->TASK_SHOULD_START_TIME;
    ptcb->DEADLINE = ptcb->period + ptcb->DEADLINE;
    ptcb->REMAINING_TIME     = ptcb->compute;
    OSTimeDly(todelay);
    if(todelay==0){
       OS_Sched();
    }
```

The wait function used to simulate resource usage time

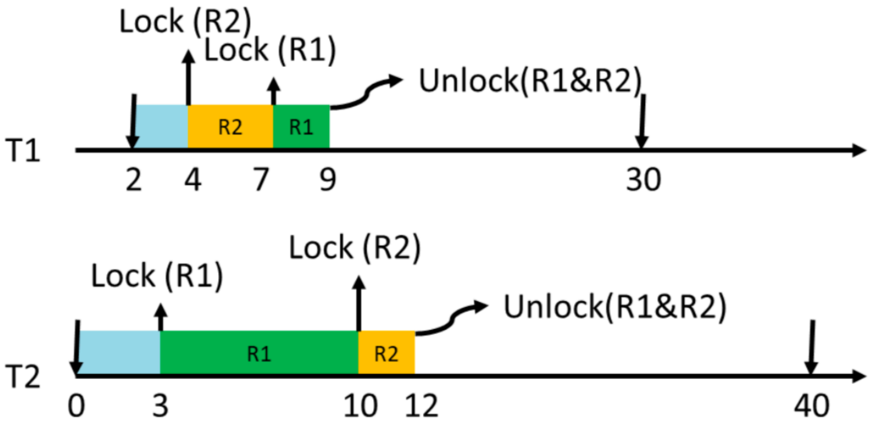
```
void wait(int tick)
{
     #if OS_CRITICAL_METHOD == 3
         OS_CPU_SR  cpu_sr = 0;
     #endif
     int now,exit;
     OS_ENTER_CRITICAL();
     now=OSTimeGet();
     exit=now+tick;
     OS_EXIT_CRITICAL();
     while(1){
         if(exit<=OSTimeGet())
             break;
     }
}
```

2. Simulation Results

(1) Task Set 1 (Arrival Time, Period):

Task1: (2,28) CPU: 2 R2: 3 R1:2

Task2: (0,40) CPU: 3 R1: 7 R2:2

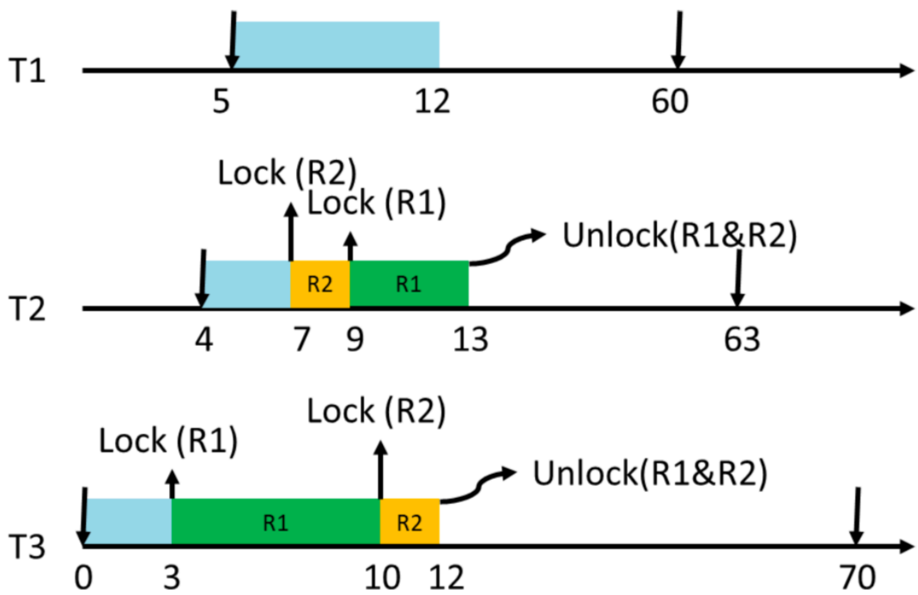


Time	Event
Time 0	Task 2
Time 2	Task 1
Time 4	Task 1 get R2
Time 7	Task 1 get R1
Time 9	Task 1 release R1
Time 9	Task 1 release R2
Time 10	Task 2 get R1
Time 17	Task 2 get R2
Time 19	Task 2 release R2
Time 19	Task 2 release R1
Time 30	Task 1
Time 32	Task 1 get R2
Time 35	Task 1 get R1
Time 37	Task 1 release R1
Time 37	Task 1 release R2
Time 40	Task 2
Time 43	Task 2 get R1
Time 50	Task 2 get R2
Time 52	Task 2 release R2
Time 52	Task 2 release R1
Time 58	Task 1
Time 60	Task 1 get R2
Time 63	Task 1 get R1
Time 65	Task 1 release R1
Time 65	Task 1 release R2
Time 80	Task 2
Time 83	Task 2 get R1
Time 90	Task 2 get R2
Time 92	Task 2 release R2
Time 92	Task 2 release R1
Time 92	Task 1

(2) Task Set 2 (Arrival Time, Period):

Task1: (5,55) CPU: 7

Task2: (4,59) CPU: 3 R2: 2 R1:4
Task3: (0,70) CPU: 3 R1: 7 R1:2



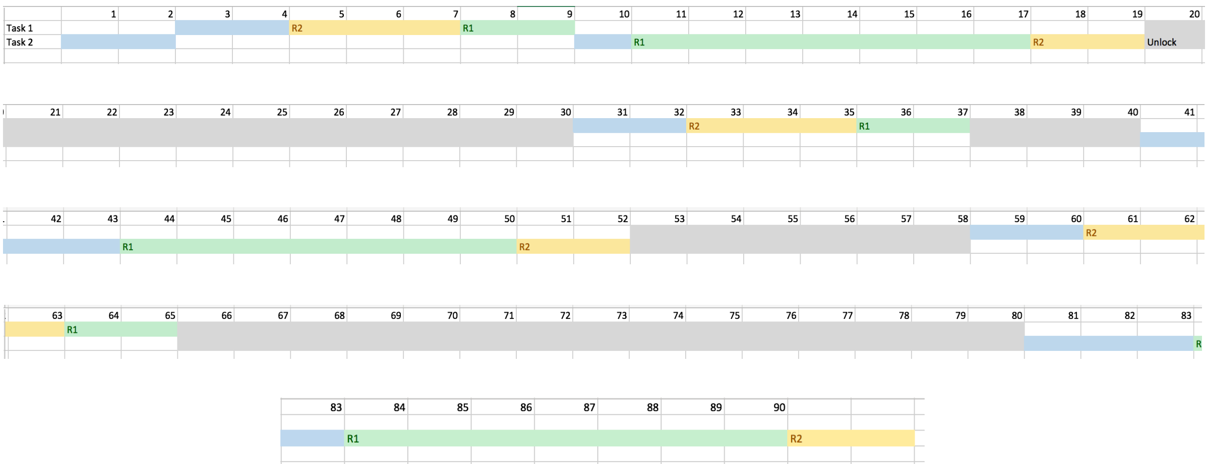
Time	Event
Time 0	Task 3
Time 3	Task 3 get R1
Time 10	Task 3 get R2
Time 12	Task 3 release R2
Time 12	Task 3 release R1
Time 12	Task 1
Time 19	Task 2
Time 22	Task 2 get R2
Time 24	Task 2 get R1
Time 28	Task 2 release R1
Time 28	Task 2 release R2
Time 60	Task 1
Time 67	Task 2
Time 70	Task 2 get R2
Time 72	Task 2 get R1
Time 76	Task 2 release R1
Time 76	Task 2 release R2
Time 76	Task 3
Time 79	Task 3 get R1
Time 86	Task 3 get R2
Time 88	Task 3 release R2
Time 88	Task 3 release R1

3. Schedulability Analysis

(1) Task Set 1:

Because the periods and computation times for both of these tasks are optimal, the only time they interfere with each other is every 120 ticks. Within our output, the only time that an interference happens is when Task 1 arrives before Task 2 finishes (At Time 2). Even though this is a NPCS protocol, because Task 2 has not locked up any resources, Task 1 can preempt Task 2 because it has an earlier deadline. In this task set, the earliest deadline task truly finishes first, while the later deadline task is finished later.

Time	Event
Time 0	Task 2
Time 2	Task 1
Time 4	Task 1 get R2
Time 7	Task 1 get R1
Time 9	Task 1 release R1
Time 9	Task 1 release R2



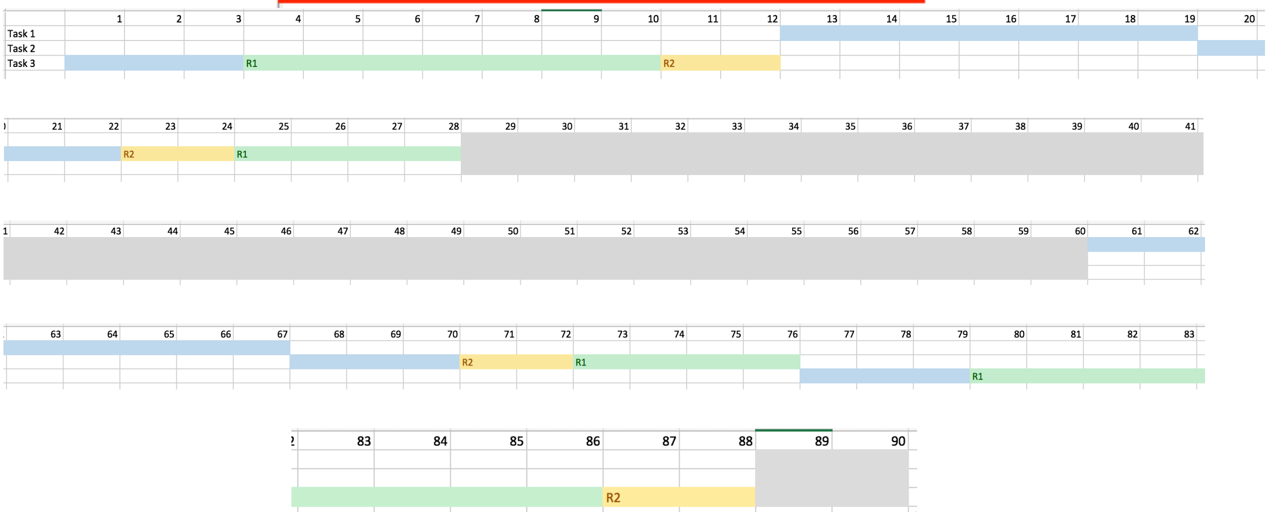
(2) Task Set 2:

Task 1 and Task 2 both arrive when Task 3 is computing. Task 1 arrives at Time 5, and Task 2 arrives at Time 4. Both of these tasks have a nearer deadline than Task 3. Normally in an EDF scheduler Task 3 would be preempted by the earlier deadline tasks, Task 1 and Task 2. However, Task 3 is currently using resources so preemption is disabled. Task 1 runs after Task 3 is finished using its resources, and Task 2 runs after Task 1 (Task 1 has an earlier deadline). Even thought this is a EDF scheduler, the NPCS results in a less responsive result for the earlier deadline tasks.

Time	Event
Time 0	Task 3
Time 3	Task 3 get R1
Time 10	Task 3 get R2
Time 12	Task 3 release R2
Time 12	Task 3 release R1
Time 12	Task 1
Time 19	Task 2

At Time 61-88, the tasks obey the rules of the EDF scheduler and run the task with the earlier deadline. There are no resource interferences within this time span.

```
Time 28         Task 2 release R2
Time 60         Task 1
Time 67         Task 2
Time 70         Task 2 get R2
Time 72         Task 2 get R1
Time 76         Task 2 release R1
Time 76         Task 2 release R2
Time 76         Task 3
Time 79         Task 3 get R1
Time 86         Task 3 get R2
Time 88         Task 3 release R2
Time 88         Task 3 release R1
```



PART 2 SRP Implementation

1. Implementation Description:

The main objective of this project is to implement the Stack-Resource Policy.
I decided to implement my code within ***OSTimeTick, OSMutexPend and OSMutexPost.***

(1) Variables

First, I implemented the variables I needed within the OS_EVENT and OS_TCB structures.

(a) OS_EVENT:

I implemented a new "ceiling" variable to save the system ceiling of resources.

```
typedef struct os_event {
    INT8U    OSEventType;                    /* Type of event control block (see OS_EVENT_TYPE_xxxx)    */
    void    *OSEventPtr;                     /* Pointer to message or queue structure                   */
    INT16U   OSEventCnt;                     /* Semaphore Count (not used if other EVENT type)          */
    INT8U    ceiling; //System ceiling for the resource
```

(b) OS_TCB:

I implemented a new variable to save the original deadline of the task. The deadline will be changed/inherited often in a SRP setting so we have to save the original deadline.

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;          /*
    INT32U             task_id;
    INT32U             task_times;
    INT32U             period;
    INT32U             compute;
    INT32U             computing;
    INT32U             TASK_SHOULD_START_TIME;
    INT32U             TASK_ACTUAL_START_TIME;
    INT32U             REMAINING_TIME;
    INT32U             RESPONSE_TIME;
    INT32U             ARRIVAL_TIME;
    INT32U             DEADLINE;
    INT32U             ORIGINAL_DEADLINE;
```

(c) Global Variables:

```
//System ceiling
OS_EXT  INT8U           SYSTEM_CEILING;
//Save the previous system ceiling, if system ceiling is updated twice
OS_EXT  INT8U           PREV_SYSTEM_CEILING;
//Save the original task priority that was holding the resource
OS_EXT  INT8U           HOLDING_RESOURCE_PRIORITY;
```

I. **SYSTEM_CEILING**: to save the current system ceiling
II. **PREV_SYSTEM_CEILING**: to save the previous system ceiling

III. **HOLDING_RESOURCE_PRIORITY**: to save the priority of the task currently holding the resource

(2) New Implementations

(a) OSTimeTick:

In OSTimeTick, I had to add an if statement to check whether or not the current task has to inherit a new task's deadline.

We use HOLDING_RESOURCE_PRIORITY to check if there are resources being held. We also check if the arriving task's priority is higher than the SYSTEM_CEILING's priority.

If all of these are valid, we allow the current task to inherit the arriving task's deadline and continue running the current task.

```
//Only inherit the deadline if the system ceiling is lower and if the original task is holding resources
if(OSTCBCur->OSTCBPrio != OS_TASK_IDLE_PRIO){
    if(OSTime == ptcb->TASK_SHOULD_START_TIME && HOLDING_RESOURCE_PRIORITY!=0 &&
    ptcb->task_id!=OSTCBCur->task_id && SYSTEM_CEILING<=(ptcb->OSTCBPrio%10)){
        if(OSTCBCur->DEADLINE > ptcb->DEADLINE){
            OSTCBCur->DEADLINE = ptcb->DEADLINE;
            printf("Time %-2d\t\t Task %-2d%-10s %-3d\n",OSTime,OSTCBCur->task_id,"inherit deadline       deadline",OSTCB
        }
    }

}
```

(b) OSMutexPend:

In OSMutexPend, we simply change the previous system ceiling and system ceiling according the current values. If SYSTEM_CEILING is "0", we can directly change the value. If SYSTEM_CEILING is larger than "0", we have to adjust the previous system ceiling to the old system ceiling and change the current system ceiling into the new one.

```
if ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {

    //if this is not zero, this means there are other resoures not let go of yet, so we record the old ceiling
    if(SYSTEM_CEILING != 0){
        PREV_SYSTEM_CEILING = SYSTEM_CEILING;
    }
    //change ceiling only if it is higher than the original or if system ceiling is not set
    if( SYSTEM_CEILING < pevent->ceiling || SYSTEM_CEILING == 0){
        SYSTEM_CEILING = pevent->ceiling;
    }
```

(c) OSMutexPost:

In OSMutexPost, we do two things.

First we change the current task's deadline back to its original deadline. Next, we change the system ceiling into the previous system ceiling. If PREV_SYSTEM_CEILING is "0", that means we can change the current system ceiling directly back to "0".

```
//restore the deadline to the original deadline
if(OSTCBCur->DEADLINE != OSTCBCur->ORIGINAL_DEADLINE){
    OSTCBCur->DEADLINE = OSTCBCur->ORIGINAL_DEADLINE;
}
//restore the system ceiling to the previous ceiling, if no previous, that means we can safely restore it back to zero
if(PREV_SYSTEM_CEILING != 0){
    SYSTEM_CEILING=PREV_SYSTEM_CEILING;
    PREV_SYSTEM_CEILING = 0;
}
else{
    SYSTEM_CEILING=0;
}
```

(3) Modifications to EDF

I had to modify the EDF scheduler I previously implemented in ***OSSchedNew***. Besides checking which task has earlier deadline, we also have to check if the current system ceiling has a value. If it doesn't we can just check for the earliest deadline. If there is a system ceiling, we have to check if the task has both an earlier deadline and a higher priority than the system ceiling before we can schedule it.

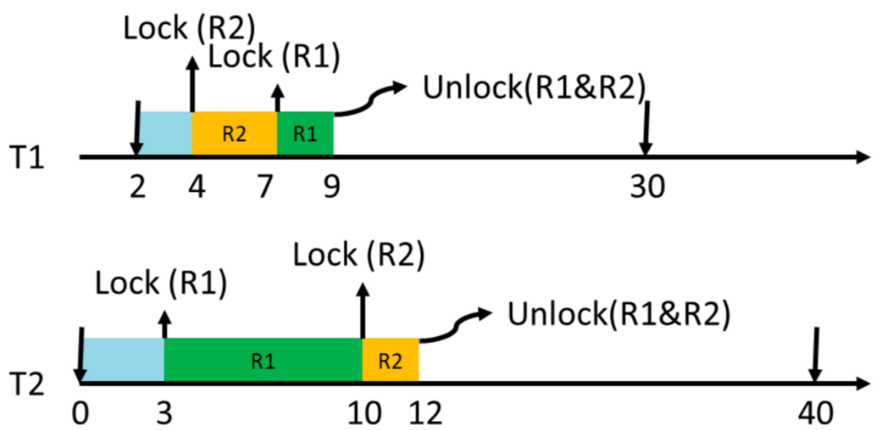
```
//Slightly modify the original EDF algorithm to check the system ceiling
while (ptcb->OSTCBPrio != OS_TASK_IDLE_PRIO) {
    if(ptcb->DEADLINE > 0 && ptcb->OSTCBDly==0 && OSTimeGet()!=ptcb->DEADLINE){
        if(SYSTEM_CEILING!=0){
            if((ptcb->OSTCBPrio%10)<SYSTEM_CEILING || HOLDING_RESOURCE_PRIORITY == ptcb->OSTCBPrio){
                if(ptcb->DEADLINE <= nearest){
                    i=ptcb->OSTCBPrio;
                    nearest=ptcb->DEADLINE;
                }
            }
        }
        else{
            if(ptcb->DEADLINE <= nearest){
                i=ptcb->OSTCBPrio;
                nearest=ptcb->DEADLINE;
            }
        }
    }
    ptcb = ptcb->OSTCBNext;
}
if(OSPrioHighRdy != OS_TASK_IDLE_PRIO){
    OSPrioHighRdy = i;
}
```

2. Simulation Results

(1) Task Set 1 (Arrival Time, Period):

Task1: (2,28) CPU: 2 R2: 3 R1:2

Task2: (0,40) CPU: 3 R1: 7 R2:2



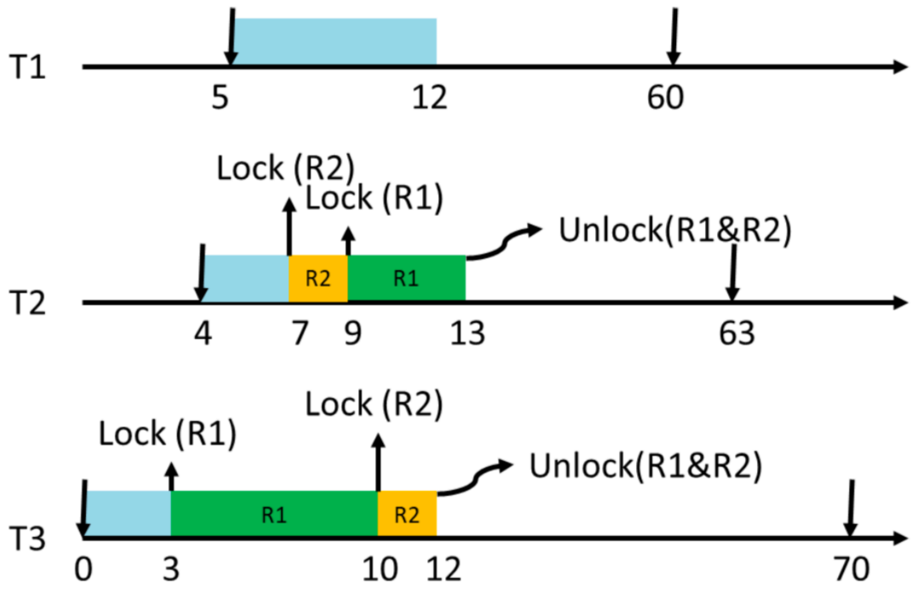
Time	Event	System Ceiling
Time 0	Task 2	
Time 2	Task 1	
Time 4	Task 1 get R2	1
Time 7	Task 1 get R1	1
Time 9	Task 1 release R1	1
Time 9	Task 1 release R2	0
Time 10	Task 2 get R1	1
Time 17	Task 2 get R2	1
Time 19	Task 2 release R2	1
Time 19	Task 2 release R1	0
Time 30	Task 1	
Time 32	Task 1 get R2	1
Time 35	Task 1 get R1	1
Time 37	Task 1 release R1	1
Time 37	Task 1 release R2	0
Time 40	Task 2	
Time 43	Task 2 get R1	1
Time 50	Task 2 get R2	1
Time 52	Task 2 release R2	1
Time 52	Task 2 release R1	0
Time 58	Task 1	
Time 60	Task 1 get R2	1
Time 63	Task 1 get R1	1
Time 65	Task 1 release R1	1
Time 65	Task 1 release R2	0
Time 80	Task 2	
Time 83	Task 2 get R1	1
Time 86	Task 2 inherit deadline	deadline 114
Time 90	Task 2 get R2	1
Time 92	Task 2 release R2	1
Time 92	Task 2 release R1	0

(2) Task Set 2 (Arrival Time, Period):

Task1: (5,55) CPU: 7

Task2: (4,59) CPU: 3 R2: 2 R1:4

Task3: (0,70) CPU: 3 R1: 7 R1:2



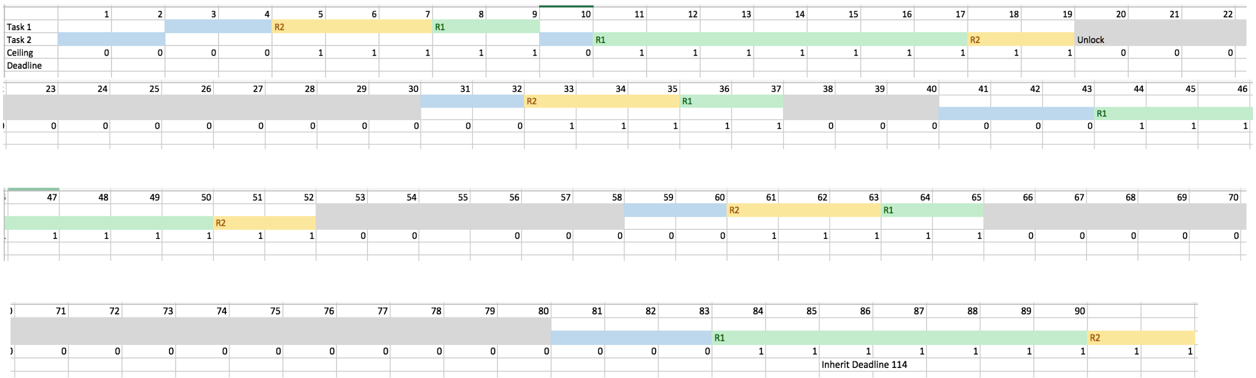
Time	Event	System Ceiling
Time 0	Task 3	
Time 3	Task 3 get R1	2
Time 4	Task 3 inherit deadline	deadline 63
Time 5	Task 1	
Time 17	Task 3 get R2	2
Time 19	Task 3 release R2	2
Time 19	Task 3 release R1	0
Time 19	Task 2	
Time 22	Task 2 get R2	2
Time 24	Task 2 get R1	2
Time 28	Task 2 release R1	2
Time 28	Task 2 release R2	0
Time 60	Task 1	
Time 67	Task 2	
Time 70	Task 2 get R2	2
Time 72	Task 2 get R1	2
Time 76	Task 2 release R1	2
Time 76	Task 2 release R2	0
Time 76	Task 3	
Time 79	Task 3 get R1	2
Time 86	Task 3 get R2	2
Time 88	Task 3 release R2	2
Time 88	Task 3 release R1	0

3. Schedulability Analysis

(1) Task Set 1:

At Time 2, Task 2 has not used any resources yet so it is pre-emptible by Task 1.

At Time 86, Task 1 arrives and has an earlier deadline than Task 2. However, Task 2 is currently using resources and Task 1 does not have a higher preemption level than the system ceiling. Due to these circumstances, Task 2 has inherit Task 1's deadline of 114.



(2) Task Set 2:

At Time 4, Task 3 is already using resources so the system ceiling is raised to 2. Since Task 2's preemption level is not high enough and cannot be scheduled, Task 3 inherits its deadline.
However, at Time 5 Task 1's preemption level is higher than the system ceiling. Task 1 preempts Task 3. We do not have to be afraid of deadlocks because Task 1 has no resource conflicts with Task 3. After Task 1 is finished, we return back to Task 3 and finish running it.

4. Experience/Impression

Although implementing the NPCS protocol was quite easy, the SRT implementation was quite challenging. In SRT, deadlines for tasks are constantly changing and numerous situations have to be considered. The system ceiling is constantly changing and we have to consider whether or not tasks can be run according to the current system ceiling. Despite this, I overcame numerous obstacles to finish this project. Over the past few projects I have attained a solid grasp of how to program certain protocols and methods within a simple OS system. However, most of these protocols are only rudimentary and there are many more types of protocols within the world of real time operating systems. I hope I can use the skills I have learned through programming these projects to implement even more intermediate protocols in the future.