PA3

M11202117 呂長恩

Part 1 NPCS

```c
/*  End time for the simulation */
#define SYSTEM_END_TIME 100

/*  Input file  */
FILE* fp;
#define INPUT_FILE_NAME "./TaskSet.txt"
#define OUTPUT_FILE_NAME "./Output.txt"
#define MAX  20        // Task maximum number
#define INFO 10        // information of task
/*  Input file  */

/*  Output file */
FILE* Output_fp;
errno_t Output_err;
/*  Output file */

/*  Task Structure  */
typedef struct task_para_set {
    INT16U TaskID;
    INT16U TaskArriveTime;
    INT16U TaskExecutionTime;
    INT16U TaskPeriodic;
    INT16U TaskNumber;
    INT16U TaskPriority;
    INT16U R1LockTime;
    INT16U R1UnLockTime;
    INT16U R2LockTime;
    INT16U R2UnLockTime;
} task_para_set;
```

File: `ucos_ii.h`

- Adjusted the system end time to 100 seconds.

- Adjusted the the number of information of a task to 10.
- Added the 4 parameters in the `task_para_set` structure, recording lock time and unlock time of R1 and R2.

```c
while (!feof(fp))
{
    i = 0;
    memset(str, 0, sizeof(str));
    fgets(str, sizeof(str) - 1, fp);
    ptr = strtok_s(str, " ", &pTmp); // partition string by " "
    while (ptr != NULL)
    {
        TaskInfo[i] = atoi(ptr);
        ptr = strtok_s(NULL, " ", &pTmp);

        if (i == 0) {
            TASK_NUMBER++;
            TaskParameter[j].TaskID = TaskInfo[i];
        }
        else if (i == 1)
            TaskParameter[j].TaskArriveTime = TaskInfo[i];
        else if (i == 2)
            TaskParameter[j].TaskExecutionTime = TaskInfo[i];
        else if (i == 3) {
            TaskParameter[j].TaskPeriodic = TaskInfo[i];
            TaskParameter[j].TaskPriority = TaskInfo[i];
        }
        else if (i == 4)
            TaskParameter[j].R1LockTime = TaskInfo[i];
        else if (i == 5)
            TaskParameter[j].R1UnLockTime = TaskInfo[i];
        else if (i == 6)
            TaskParameter[j].R2LockTime = TaskInfo[i];
        else if (i == 7)
            TaskParameter[j].R2UnLockTime = TaskInfo[i];
        i++;
    }
    j++;
}
fclose(fp);
```

File: `app_hooks.c`

- Added code to read the lock and inlock data of R1 and R2 from TaskSet.txt.

```c
#define OS_SCHED_LOCK_EN            1u   /* Include code for OSSchedLock() and OSSchedUnlock()        */

#define OS_TICK_STEP_EN             1u   /* Enable tick stepping feature for uC/OS-View               */
#define OS_TICKS_PER_SEC          100u   /* Set the number of ticks in one second                     */

#define OS_TLS_TBL_SIZE             5u   /* Size of Thread-Local Storage Table                        */
```

File: `os_cfg_r.h`

- Adjusted the number of ticks per second to 100.

```c
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;           /* Pointer to current top of stack

#if OS_TASK_CREATE_EXT_EN > 0u
    void            *OSTCBExtPtr;           /* Pointer to user definable data for TCB extension
    OS_STK          *OSTCBStkBottom;        /* Pointer to bottom of stack
    INT32U           OSTCBStkSize;          /* Size of task stack (in number of stack elements)
    INT16U           OSTCBOpt;              /* Task options as passed by OSTaskCreateExt()
    INT16U           OSTCBId;               /* Task ID (0..65535)
    INT8U            OSTCBExecuTime;        /* Task execution time
    INT8U            OSTCBExecuTimeCtr;     /* Task execution time for counting
    INT8U            OSTCBArriTime;         /* Task arrive time
    INT8U            OSTCBPeriod;           /* The period of task
    INT8U            R1RelatLockTime;       /* The related lock time to resource1
    INT8U            R1RelatUnLockTime;     /* The related unlock time to resource1
    INT8U            R2RelatLockTime;       /* The related lock time to resource2
    INT8U            R2RelatUnLockTime;     /* The related unlock time to resource2
    INT8U            R1UnLockTime;          /* The really unlock time to resource1
    INT8U            R2UnLockTime;          /* The really unlock time to resource2
    INT8U            R1LockFlag;            /* When flaged, R1 is used by this task
    INT8U            R2LockFlag;            /* When flaged, R2 is used by this task
    INT16U           OSTCBBlockingTime;     /* The counter of blocking time
#endif
```

File: `ucos_ii.h`

- Added new parameters to record usage information of R1 and R2 for each task.
- Recorded blocking time.

```c
        if (prio != OS_TASK_IDLE_PRIO) {
            ptcb->OSTCBArriTime         = TaskParameter[id - 1].TaskArriveTime;       /* Store arrive time                */
            ptcb->OSTCBExecuTime        = TaskParameter[id - 1].TaskExecutionTime;    /* Store execution time             */
            ptcb->OSTCBExecuTimeCtr     = TaskParameter[id - 1].TaskExecutionTime;    /* Store execution time to count    */
            ptcb->OSTCBPeriod           = TaskParameter[id - 1].TaskPeriodic;
            if (TaskParameter[id - 1].R1LockTime == TaskParameter[id - 1].R1UnLockTime)
            {
                /* if task doesn't use R1 */
                /* set the lock and unlock time to 140, let the task never use this resource until the system end */
                ptcb->R1RelatLockTime = 140;
                ptcb->R1RelatUnLockTime = 140;
                ptcb->R1UnLockTime = 140;
                ptcb->R1LockFlag = 0;
            }
            else
            {
                ptcb->R1RelatLockTime = TaskParameter[id - 1].R1LockTime;
                ptcb->R1RelatUnLockTime = TaskParameter[id - 1].R1UnLockTime;
                ptcb->R1LockFlag = 0;
            }
            if (TaskParameter[id - 1].R2LockTime == TaskParameter[id - 1].R2UnLockTime)
            {
                /* if task doesn't use R2 */
                /* set the lock and unlock time to 140, let the task never use this resource until the system end */
                ptcb->R2RelatLockTime = 140;
                ptcb->R2RelatUnLockTime = 140;
                ptcb->R2UnLockTime = 140;
                ptcb->R2LockFlag = 0;
            }
            else
            {
                ptcb->R2RelatLockTime = TaskParameter[id - 1].R2LockTime;
                ptcb->R2RelatUnLockTime = TaskParameter[id - 1].R2UnLockTime;
                ptcb->R2LockFlag = 0;
            }
        }
```

File: `os_core.c` → `OS_TCBInit()`

- Added a decision code to check whether the task will use R1 and R2. if it does not use the resource, set the lock and unlock time to 140, ensuring the task never acquires the resource until the system ends.

Implementation

> **NPCS is mainly implemented through a NPCS lock**. when the lock is locked, it can not schedule tasks until the locked is released. The lock will be opened when the resource is released by the holding task. The task will not be preempted in this duration.

```c
void  OSStart (void)
{
    OS_TCB *ptcb;
    OSNPCSLock = 0; /* initial state */
    if (OSRunning == OS_FALSE) {
        OSTimeSet(0);                                   /*Set OS Start Time is 0*/
        fopen_s(&Output_fp, "./Output.txt", "a");
        ptcb = OSTCBList;
        while (ptcb->OSTCBPrio != OS_TASK_IDLE_PRIO) {
            if (ptcb->OSTCBArriTime == OSTimeGet()) { // if task arrives
                ptcb->OSTCBArriTime = OSTimeGet();
                OSRdyGrp               |= ptcb->OSTCBBitY; /* Make ready                          */
                OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
            }
            ptcb = ptcb->OSTCBNext;
        }
        OS_SchedNew();                                  /* Find highest priority's task priority number   */
        OSPrioCur     = OSPrioHighRdy;
        OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy]; /* Point to highest priority task ready to run    */
        OSTCBCur      = OSTCBHighRdy;
```

```c
        if (OSTCBCur->R1RelatLockTime == 0)
        {
            /* Task uses R1 at 0 second */
            OSNPCSLock = 1;                                                 /* Lock the NPCS lock */
            OSTCBCur->R1LockFlag = 1;                                       /* Flag the R1 flag of the highest task */
            OSTCBCur->R1UnLockTime = OSTimeGet() + OSTCBCur->R1RelatUnLockTime;     /* Compute the R1 unlock time of the highest task */
            printf("%3d\t LockResource\t task(%2d)(%2d)\t\t R1\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);
            fprintf(Output_fp, "%3d\t LockResource\t task(%2d)(%2d)\t\t R1\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);

        }
        if (OSTCBCur->R2RelatLockTime == 0)
        {
            /* Tesk uses R2 at 0 second */
            OSNPCSLock = 1;                                                 /* Lock the NPCS lock*/
            OSTCBCur->R2LockFlag = 1;                                       /* Flag the R2 flag of the highest task */
            OSTCBCur->R2UnLockTime = OSTimeGet() + OSTCBCur->R2RelatUnLockTime;     /* Compute the R2 unlock time of the highest task */
            printf("%3d\t LockResource\t task(%2d)(%2d)\t\t R2\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);
            fprintf(Output_fp, "%3d\t LockResource\t task(%2d)(%2d)\t\t R2\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);

        }
        OSStartHighRdy();                          /* Execute target specific code to start task     */
    }
}
```

File: `os_core.c` → `OSStart()`

- First, set the status of `OSNPCSLock` to 0, indicating the lock is open.

- Second, check whether the highest-priority task uses the R1 or R2 at 0 seconds. If the task uses either, lock `OSNPCSLock` and flag the resource flag of the task.

```c
static  void  OS_SchedNew (void)                    /* Find the highest priority task */
{
#if OS_LOWEST_PRIO <= 63u                           /* See if we support up to 64 tasks                   */
    INT8U   y;

    if (OSNPCSLock == 0)
    {
        /* if npcs is lock, it can not schedule tasks */
        y = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy = (INT8U)((y << 3u) + OSUnMapTbl[OSRdyTbl[y]]);
    }
#else                                               /* We support up to 256 tasks                         */
    INT8U     y;
    OS_PRIO  *ptbl;


    if ((OSRdyGrp & 0xFFu) != 0u) {
        y = OSUnMapTbl[OSRdyGrp & 0xFFu];
    } else {
        y = OSUnMapTbl[(OS_PRIO)(OSRdyGrp >> 8u) & 0xFFu] + 8u;
    }
    ptbl = &OSRdyTbl[y];
    if ((*ptbl & 0xFFu) != 0u) {
        OSPrioHighRdy = (INT8U)((y << 4u) + OSUnMapTbl[(*ptbl & 0xFFu)]);
    } else {
        OSPrioHighRdy = (INT8U)((y << 4u) + OSUnMapTbl[(OS_PRIO)(*ptbl >> 8u) & 0xFFu] + 8u);
    }
#endif
}
```

File: `os_core.c` → `OS_SchedNew()`

- If the `OSNPCSLock` equals to 1, indicating NPCS lock is held by a task. During this time, the scheduler can not schedule new tasks.

```c
/* Resource Request */
if (OSTCBCur->OSTCBExecuTime - OSTCBCur->OSTCBExecuTimeCtr == OSTCBCur->R1RelatLockTime)
{
    /* Request R1 */
    OSNPCSLock = 1;     /* Lock the NPCS lock */
    OSTCBCur->R1LockFlag = 1;   /* Flag the R1 flag of the task */
    OSTCBCur->R1UnLockTime = OSTimeGet() + (OSTCBCur->R1RelatUnLockTime - OSTCBCur->R1RelatLockTime); /* compute the R1 unlock time of the task*/
    printf("%3d\t LockResource\t task(%2d)(%2d)\t\tR1\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);
    fprintf(Output_fp, "%3d\t LockResource\t task(%2d)(%2d)\t\tR1\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);

}
if (OSTCBCur->OSTCBExecuTime - OSTCBCur->OSTCBExecuTimeCtr == OSTCBCur->R2RelatLockTime)
{
    /* Request R2 */
    OSNPCSLock = 1;     /* Lock the NPCS lock */
    OSTCBCur->R2LockFlag = 1;   /* Flag the R2 flag of the task */
    OSTCBCur->R2UnLockTime = OSTimeGet() + (OSTCBCur->R2RelatUnLockTime - OSTCBCur->R2RelatLockTime); /* Compute the R2 unlock time of the task */
    printf("%3d\t LockResource\t task(%2d)(%2d)\t\tR2\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);
    fprintf(Output_fp, "%3d\t LockResource\t task(%2d)(%2d)\t\tR2\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);
}
```

File: `os_core.c` → `OSTimeTick()`

- Request R1
    - If the current execution time of the task substracted from the total execution time equals to the `R1RelatLockTime`, it indicates the task needs to acquire R1

at that specific time.
    - After obtaining R1, the current task locks the NPCS lock and flags its R1 flag.
    - It computes the R1 unlock time and saves it.
- Request R2
    - If the current execution time of the task substracted from the total execution time equals to the `R1RelatLockTime`, it indicates the task needs to acquire R2 at that specific time.
    - After obtaining R2, the current task locks the NPCS lock and flags its R2 flag.
    - It computes the R2 unlock time and saves it.
- No task can preemptively interrupt the current task until the NPCS is unlocked, even if a new task has a higher priority.

```c
/* Resource Release */
if (OSTCBCur->R1UnLockTime == OSTimeGet())
{
    /* Release R1 */
    OSTCBCur->R1LockFlag = 0;   /* Unflag the R1 flag of the task, it indicates the R1 is released */
    printf("%3d\t UnlockResource\t task(%2d)(%2d)\t\tR1\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);
    fprintf(Output_fp, "%3d\t UnlockResource\t task(%2d)(%2d)\t\tR1\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);

}
if (OSTCBCur->R2UnLockTime == OSTimeGet())
{
    /* Release R2 */
    OSTCBCur->R2LockFlag = 0;   /* Unflag the R2 flag of the task, it indicates the R2 is released */
    printf("%3d\t UnlockResource\t task(%2d)(%2d)\t\tR2\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);
    fprintf(Output_fp, "%3d\t UnlockResource\t task(%2d)(%2d)\t\tR2\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);
}
```

File: `os_core.c` → `OSTimeTick()`

- Release R1
    - If the current time tick equals to the `R1UnlockTime`, it is the time to release R1.
    - Set the `R1LockFlag` to 0, indicating that R1 is released by the current task.
- Release R2
    - If the current time tick equals to the `R2UnlockTime`, it is the time to release R2.
    - Set the `R2LockFlag` to 0, indicating that R2 is released by the current task.
- When releasing either R1 or R2, the NPCS lock can not be unlocked immediately. For instance, if releasing R1, and R2 has not reached the unlock time, that NPCS must remain locked even if R1 is released.

```c
    /* Release NPCS Lock */
    if ((OSTCBCur->R1LockFlag | OSTCBCur->R2LockFlag) == 0)
    {
        /* if either R1 or R2 flag equals to 1, the NPCS lock must remain locked */
        /* uc/OS-ii releases the NPCS lock only if the flag of both R1 and R2 equal to 0 */
        OSNPCSLock = 0;
    }
```

File: `os_core.c` → `OSTimeTick()`

- uc/OS-ii releases the NPCS lock only if the flag of both R1 and R2 equal to 0.
- It indicates that R1 and R2 must be released before the NPCS lock is unlocked.

```c
        /* Blocking Time */
        if ((ptcb->OSTCBPrio < OSTCBCur->OSTCBPrio) && (ptcb->OSTCBExecuTimeCtr > 0) && (OSNPCSLock == 1))
        {
            /* Increment the blocking time of the task */
            ptcb->OSTCBBlockingTime++;
        }
```

File: `os_core.c` → `OSTimeTick()`

- A task is blocked under the following conditions:
    - The priority of the current task is lower than the task.
    - The task has not completed.
    - The NPCS lock is set to 1, indicating the NPCS lock is locked.
- The blocking time represents that how long the higher priority task has been impeded by lower priority task.

```c
(OSTCBCur->OSTCBPrio != OS_TASK_IDLE_PRIO) {
 if (OSTCBCur->OSTCBExecuTimeCtr == 0) {
    if (OSTCBHighRdy->OSTCBPrio == OS_TASK_IDLE_PRIO) {
        printf("%3d\t Completion\t task(%2d)(%2d)\t\t", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);
        printf("task(%2d)\t\t", OSTCBHighRdy->OSTCBPrio);
        printf("%2d\t\t", OSTimeGet() - OSTCBCur->OSTCBArriTime); /*Response Time*/
        printf("%2d\t\t", OSTCBCur->OSTCBBlockingTime); /*Blocking Time*/
        printf("%2d\n", OSTimeGet() - OSTCBCur->OSTCBArriTime - OSTCBCur->OSTCBBlockingTime - OSTCBCur->OSTCBExecuTime); /*Preemptive Time*/

        fprintf(Output_fp, "%3d\t Completion\t task(%2d)(%2d)\t\t", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);
        fprintf(Output_fp, "task(%2d)\t\t", OSTCBHighRdy->OSTCBPrio);
        fprintf(Output_fp, "%2d\t\t", OSTimeGet() - OSTCBCur->OSTCBArriTime); /*Response Time*/
        fprintf(Output_fp, "%2d\t\t", OSTCBCur->OSTCBBlockingTime); /*Blocking Time*/
        fprintf(Output_fp, "%2d\n", OSTimeGet() - OSTCBCur->OSTCBArriTime - OSTCBCur->OSTCBBlockingTime - OSTCBCur->OSTCBExecuTime); /*Preemptive Time*/
    }
    else {
        printf("%3d\t Completion\t task(%2d)(%2d)\t\t", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);
        printf("task(%2d)(%2d)\t\t", OSTCBHighRdy->OSTCBId, OSTCBHighRdy->OSTCBCtxSwCtr);
        printf("%2d\t\t", OSTimeGet() - OSTCBCur->OSTCBArriTime); /* Response Time */
        printf("%2d\t\t", OSTCBCur->OSTCBBlockingTime); /*Blocking Time*/
        printf("%2d\n", OSTimeGet() - OSTCBCur->OSTCBArriTime - OSTCBCur->OSTCBBlockingTime - OSTCBCur->OSTCBExecuTime); /*Preemptive Time*/

        fprintf(Output_fp, "%3d\t Completion\t task(%2d)(%2d)\t\t", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr);
        fprintf(Output_fp, "task(%2d)(%2d)\t\t", OSTCBHighRdy->OSTCBId, OSTCBHighRdy->OSTCBCtxSwCtr);
        fprintf(Output_fp, "%2d\t\t", OSTimeGet() - OSTCBCur->OSTCBArriTime); /* Response Time */
        fprintf(Output_fp, "%2d\t\t", OSTCBCur->OSTCBBlockingTime); /*Blocking Time*/
        fprintf(Output_fp, "%2d\n", OSTimeGet() - OSTCBCur->OSTCBArriTime - OSTCBCur->OSTCBBlockingTime - OSTCBCur->OSTCBExecuTime); /*Preemptive Time*/


    }
```

File: `os_core.c` → `OSIntExit()`

- Print out the blocking time information.
- The preemptive time is calculated as folloes:

$$PreemptiveTime = ResponseTime - BlockingTime$$

- Preemptive Time is the duration of the higher priority task interrupts the lower task.
- Blocking Time is the duration of the lower priority task impeded the higher priority task.

Part 2 CPP

```
/* Resource Index */
#define R1_idx 1
#define R2_idx 2
```

- File: `ucos_ii.h`
- Declare the R1 index is 1.
- Declare the R2 index is 2.

```
*********************************************************************************************
*                                          GLOBAL VARIABLES
*********************************************************************************************
*/

OS_EXT  INT32U           OSCtxSwCtr;               /* Counter of number of context switches
OS_EXT  INT8U            resumeCurrTCB;
OS_EXT  INT8U            R1_ceiling;               /* R1 ceiling */
OS_EXT  INT8U            R2_ceiling;               /* R2 ceiling */
```

- File: `ucos_ii.h`
- Declare the global variables `R1_ceiling` and `R2_ceiling`

Implementation

> CPP is mainly implemented by changing the priority of the task. When a task utilizes resource, it inherits the priority of that resource. This mechanism prevents the higher priority task from interrupting lower priority task. It ensures that tasks utilizing the resource do not interfere with each other, but it also guarantees that certain tasks not utilizing the resource are eligible for interruption without impacting synchronization issues.

```c
void  OSStart (void)
{
    OS_TCB *ptcb;
    R1_ceiling = OS_LOWEST_PRIO;
    R2_ceiling = OS_LOWEST_PRIO;

    if (OSRunning == OS_FALSE) {
        OSTimeSet(0);                                    /*Set OS Start Time is 0*/
        fopen_s(&Output_fp, "./Output.txt", "a");
        ptcb = OSTCBList;
        while (ptcb->OSTCBPrio != OS_TASK_IDLE_PRIO) {
            /* if task arrives */
            if (ptcb->OSTCBArriTime == OSTimeGet()) { // if task arrives
                ptcb->OSTCBArriTime = OSTimeGet();
                OSRdyGrp               |= ptcb->OSTCBBitY; /* Make ready
                OSRdyTbl[ptcb->OSTCBY]  |= ptcb->OSTCBBitX;
            }
            /* Resource Ceiling */
            computeResourceCeiling(ptcb);
            ptcb = ptcb->OSTCBNext;
        }
```

File: `os_core.c` → `OSStart()`

- Intially, uC/OS-ii will traverse all the task to identify those that utilize R1 and R2. Subsequently, it will calculate the ceiling value of R1 and R2.

```c
void  computeResourceCeiling (OS_TCB* ptcb)
{
    INT8U temp_R1;
    INT8U temp_R2;

    if (ptcb->R1RelatLockTime < 140) /* R1 used */
    {
        temp_R1 = ptcb->OrigPrio - R1_idx;
        R1_ceiling = (temp_R1 < R1_ceiling) ? temp_R1 : R1_ceiling;
        /* Identify the higher priority tasks among those that utilize R1 */
    }

    if (ptcb->R2RelatLockTime < 140) /* R2 used */
    {
        temp_R2 = ptcb->OrigPrio - R2_idx;
        R2_ceiling = (temp_R2 < R2_ceiling) ? temp_R2 : R2_ceiling;
        /* Identify the higher priority tasks among those that utilize R2 */
    }
}
```

File: `os_core.c` → `computeResourceCeiling()`

- If `R1RelatLockTime` is smaller than 140, it indicates that the task utilizes R1. Thus, it will consider which value has higher priority. `R1_ceiling` will contain the higher value.
- If `R2RelatLockTime` is smaller than 140, it indicates that the task utilizes R2. Thus, it will consider which value has higher priority. `R2_ceiling` will contain the higher value.

```c
        if (OSTCBCur->R1RelatLockTime == 0)
        {
            /* using resource1 */
            OSTCBCur->R1LockFlag = 1;  /* Set the R1LcokFlag to 1 */
            OSTCBCur->R1UnLockTime = OSTCBCur->R1RelatUnLockTime - OSTCBCur->R1RelatLockTime;   /* Set the R1UnLockTime */
            if (R1_ceiling < OSTCBCur->OSTCBPrio)
            {
                /* if R1_ceiling has higher priority, then update the current task's priority */
                OSTaskChangePrio(OSPrioCur, R1_ceiling);
            }
            printf("%3d\t LockResource\t task(%2d)(%2d)\t\tR1\t\t %2d to %2d\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr,
                OSTCBCur->OrigPrio, OSTCBCur->OSTCBPrio);
            fprintf(Output_fp, "%3d\t LockResource\t task(%2d)(%2d)\t\tR1\t\t %2d to %2d\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr,
                OSTCBCur->OrigPrio, OSTCBCur->OSTCBPrio);
        }

        if (OSTCBCur->R2RelatLockTime == 0)
        {
            /* using resource2 */
            OSTCBCur->R2LockFlag = 1;   /* Set the R2LcokFlag to 1 */
            OSTCBCur->R2UnLockTime = OSTCBCur->R2RelatUnLockTime - OSTCBCur->R2RelatLockTime;   /* Set the R2UnLockTime */
            if (R2_ceiling < OSTCBCur->OSTCBPrio)
            {
                /* if R2_ceiling has higher priority, then update the current task's priority */
                OSTaskChangePrio(OSPrioCur, R1_ceiling);
            }
            printf("%3d\t LockResource\t task(%2d)(%2d)\t\tR2\t\t %2d to %2d\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr,
                OSTCBCur->OrigPrio, OSTCBCur->OSTCBPrio);
            fprintf(Output_fp, "%3d\t LockResource\t task(%2d)(%2d)\t\tR2\t\t %2d to %2d\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr,
                OSTCBCur->OrigPrio, OSTCBCur->OSTCBPrio);
        }
```

File: `os_core.c` → `OSStart()`

- Identify whether the highest-priority task uses the R1 or R2 at 0 seconds.
- If the task uses either,

- Set the resource flag of the task to 1.
- Set the resource unlock time to $RelatUnlockTime - RelatLockTime$
- Identify if the priority of the current task is greater than the R1 or R2 ceiling value. If so, change the current task's priority to the higher one.

```c
/* Resource Release */
if ((--OSTCBCur->R1UnLockTime == 0u) && (OSTCBCur->R1LockFlag == 1))
{
    /* Release R1 */
    OSTCBCur->R1LockFlag = 0;
    if (OSTCBCur->R2LockFlag == 1)
    {
        OSTaskChangePrio(OSTCBCur->OSTCBPrio, R2_ceiling);
    }
    else
    {
        OSTaskChangePrio(OSTCBCur->OSTCBPrio, OSTCBCur->OrigPrio);
    }

    printf("%3d\t UnlockResource\t task(%2d)(%2d)\t\tR1\t\t %2d to %2d\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr,
        temp_prio, OSTCBCur->OSTCBPrio);
    fprintf(Output_fp, "%3d\t UnlockResource\t task(%2d)(%2d)\t\tR1\t\t %2d to %2d\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr,
        temp_prio, OSTCBCur->OSTCBPrio);

}
if ((--OSTCBCur->R2UnLockTime == 0u) && (OSTCBCur->R2LockFlag == 1))
{
    /* Release R2 */
    OSTCBCur->R2LockFlag = 0;

    if (OSTCBCur->R1LockFlag == 1)
    {
        OSTaskChangePrio(OSTCBCur->OSTCBPrio, R1_ceiling);
    }
    else
    {
        OSTaskChangePrio(OSTCBCur->OSTCBPrio, OSTCBCur->OrigPrio);
    }

    printf("%3d\t UnlockResource\t task(%2d)(%2d)\t\tR2\t\t %2d to %2d\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr,
        temp_prio, OSTCBCur->OSTCBPrio);
    fprintf(Output_fp, "%3d\t UnlockResource\t task(%2d)(%2d)\t\tR2\t\t %2d to %2d\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr,
        temp_prio, OSTCBCur->OSTCBPrio);

}
```

File: `os_core.c` → `OSTimeTick()`

- R1 Release:
  - Identify whether R1 reaches the unlock time and whether the R1 lock flag equals to 1. If so, it is time to release R1.
  - Identify whether the current task utilizes the R2 at the same time. If so, change the priority to the priority of R2. if not, change the priority to the original priority of the task.
- R2 Release:
  - Identify whether R2 reaches the unlock time and whether the R2 lock flag equals to 1. If so, it is time to release R2.
  - Identify whether the current task utilizes the R1 at the same time. If so, change the priority to the priority of R1. if not, change the priority to the original priority of the task.

```c
/* Resource Request */
if (OSTCBCur->OSTCBExecuTime - OSTCBCur->OSTCBExecuTimeCtr == OSTCBCur->R1RelatLockTime)
{
    /* Request R1 */
    OSTCBCur->R1LockFlag = 1;   /* Set the R1LcokFlag to 1 */
    OSTCBCur->R1UnLockTime = OSTCBCur->R1RelatUnLockTime - OSTCBCur->R1RelatLockTime;   /* Set the R1UnLockTime */
    if (R1_ceiling < OSTCBCur->OSTCBPrio)
    {
        /* if R1_ceiling has higher priority, then update the current task's priority */
        OSTaskChangePrio(OSPrioCur, R1_ceiling);
    }
    printf("%3d\t LockResource\t task(%2d)(%2d)\t\tR1\t\t %2d to %2d\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr,
        temp_prio, OSTCBCur->OSTCBPrio);
    fprintf(Output_fp, "%3d\t LockResource\t task(%2d)(%2d)\t\tR1\t\t %2d to %2d\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr,
        temp_prio, OSTCBCur->OSTCBPrio);

}
if (OSTCBCur->OSTCBExecuTime - OSTCBCur->OSTCBExecuTimeCtr == OSTCBCur->R2RelatLockTime)
{
    /* Request R2 */
    OSTCBCur->R2LockFlag = 1;   /* Set the R2LcokFlag to 1 */
    OSTCBCur->R2UnLockTime = OSTCBCur->R2RelatUnLockTime - OSTCBCur->R2RelatLockTime;   /* Set the R2UnLockTime */
    if (R2_ceiling < OSTCBCur->OSTCBPrio)
    {
        /* if R2_ceiling has higher priority, then update the current task's priority */
        OSTaskChangePrio(OSPrioCur, R2_ceiling);
    }
    printf("%3d\t LockResource\t task(%2d)(%2d)\t\tR2\t\t %2d to %2d\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr,
        temp_prio, OSTCBCur->OSTCBPrio);
    fprintf(Output_fp, "%3d\t LockResource\t task(%2d)(%2d)\t\tR2\t\t %2d to %2d\n", OSTimeGet(), OSTCBCur->OSTCBId, OSTCBCur->OSTCBCtxSwCtr,
        temp_prio, OSTCBCur->OSTCBPrio);

}
```

File: `os_core.c` → `OSTimeTick()`

- R1 Request:
    - Set the resource flag of the task to 1.
    - Set the resource unlock time to $RelatUnlockTime - RelatLockTime$
    - Identify which one has higher priority between R1_ceiling and the priority of the current task. If R1_ceiling has a higher value, then change the priority to R1_ceiling.
- R2 Request:
    - Set the resource flag of the task to 1.
    - Set the resource unlock time to $RelatUnlockTime - RelatLockTime$
    - Identify which one has higher priority between R2_ceiling and the priority of the current task. If R2_ceiling has a higher value, then change the priority to R2_ceiling.

```c
/* Blocking Time */
if ((ptcb->OrigPrio < OSTCBCur->OrigPrio) && (ptcb->OSTCBExecuTimeCtr > 0) && (ptcb->OSTCBArriTime < OSTimeGet()))
{
    ptcb->OSTCBBlockingTime++;
}
```

File: `os_core.c` → `OSTimeTick()`

- A task is blocked under the following conditions:

    - The priority of the `ptcb` task is lower than the execution task.
    - The `ptcb` task has not completed.
    - The `ptcb` task has arrived.
- The blocking time represents that how long the higher priority task has been impeded by lower priority task.

The parts not mentioned are set up the same as the NPCS.

Part 3 Performance Analysis

Scheduling Behaviors between NPCS and CPP

- *NPCS* employs a lock to block the scheduler. Even if there is a higher priority task emerges, it cannot interrupt the ongoing task within this duration. However, some **higher tasks that do not utilize this resource will still be affected**. **They can not interrupt the lower priority task, even through the interruption of the lower priority task would not cause synchronization issues**. NPCS would have more blocking time than CPP.
- In *CPP*, avoiding synchronization issues is achieved by elevating the priority instead of using locks. CPP first calculates the highest priority among all tasks that utilize a particular resource and determines the appropriate priority for that resource. Therefore, when the resource is only utilized by tasks with lower priorities, the resource's priority remains relatively low. In this scenario, **when a higher-priority task arrives, it can preemptively interrupt to execute the higher-priority task. It would allow CPP to have less blocking time than NPCS.**

Blocking Time vs. Preemption Time

- *Blocking Time* refers to the period during which a higher-priority task, upon arrival, is unable to execute immediately. Instead, it must wait until a lower-priority task releases the necessary resources, leading to a waiting period before the higher-priority task can take over execution.
- *Preemption Time* refers to the duration when the currently executing task is interrupted by a higher-priority task, preventing it from continuing its execution. The time spent waiting for the higher-priority task to complete its execution before the original task can resume is known as preemption time.

Deadlock Problem

- In *NPCS*, once a task acquires a resource, it cannot be preempted by other tasks until it releases that resource. Therefore, there is no scenario where two tasks are mutually waiting for each other's resources, and this eliminates the possibility of a deadlock.
- In *CPP*, the OS first calculates all the tasks that will use a particular resource and sets its priority higher than all of them. When a task acquires the resource, its priority is elevated to the priority of that resource. At this point, no other task needs to use the resource with a priority higher than the acquired task. Consequently, the OS avoids the occurrence of a deadlock where two tasks need access to resources held by each other.