[PART I] NPCS Implementation

- The screenshot result (with the given format) of the two task sets. (Time tick 0-100) (10%)

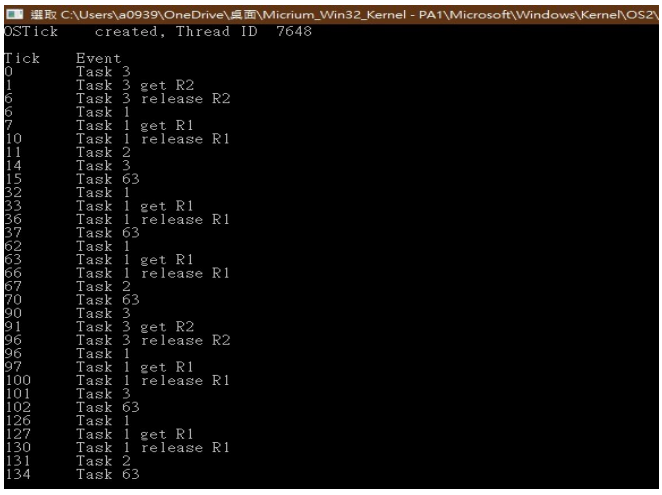


Fig 1. NPCS Task Set1 ={task1(2,5,30), task2(3,3,60), task3(0,7,90)}

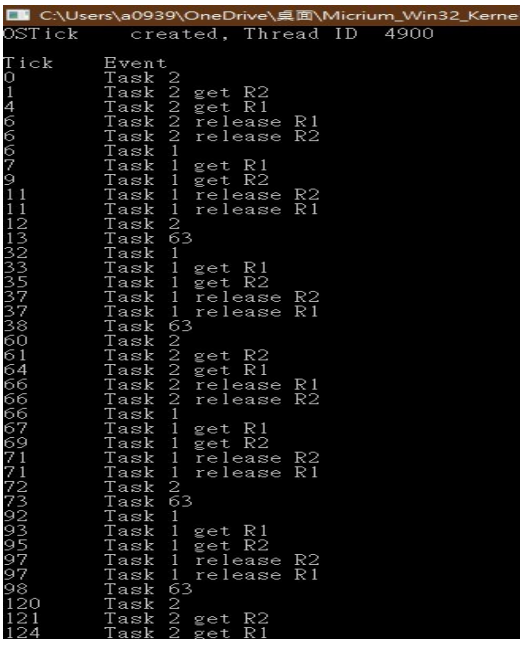


Fig 2. NPCS Task Set2 = {task1(2,6,30), task2(0,7,60)}

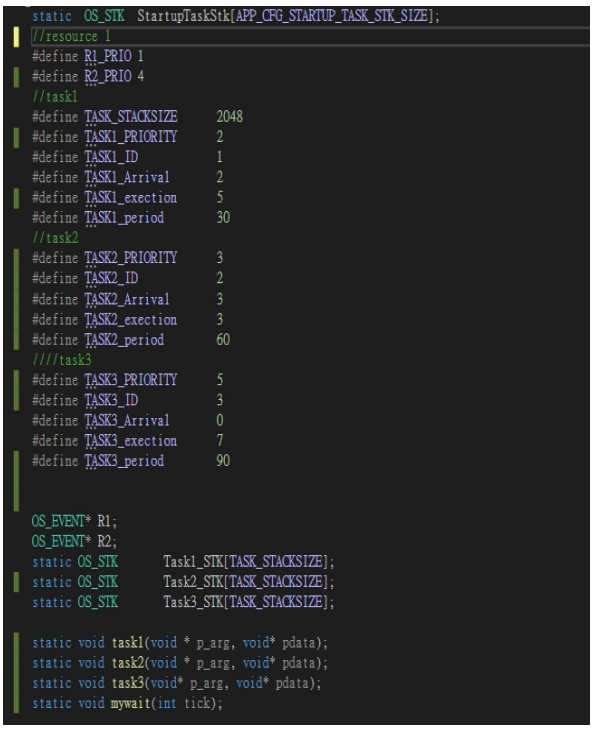
- A report that describes your implementation, including scheduling results of two task sets, modified functions, data structure, etc. (please **ATTACH** the screenshot of the code and **MARK** the modified part)

由於第一部分跟第二部分得程式都寫在同一個裡面，在說明第二部分會使用到這邊的圖片說明。

```
INT32U          execution;
INT32U          response;
INT32U          execution_frequency;
INT32U          task_arrival;
```

Fig 3. OS_TCB基本設定

一開始先設定task的資料結構，如Fig 3.所示。



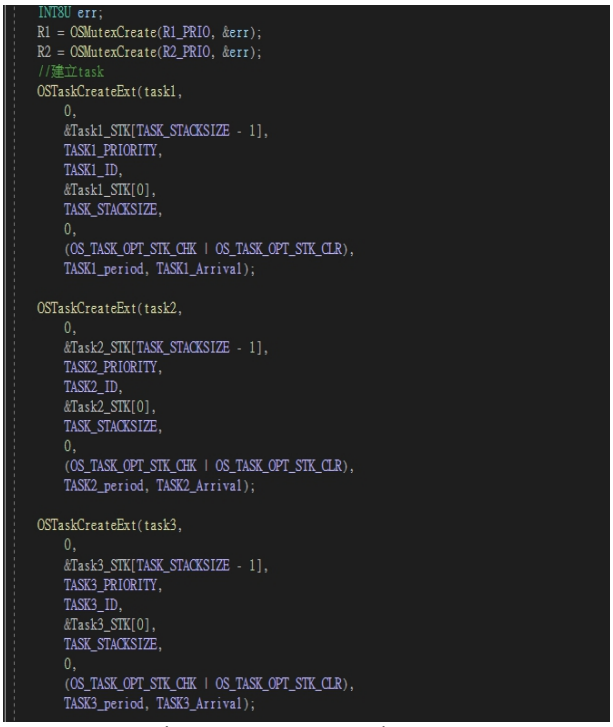
```
static  OS_STK  StartupTaskStk[APP_CFG_STARTUP_TASK_STK_SIZE];
//resource 1
#define R1_PRIO 1
#define R2_PRIO 4
//task1
#define TASK_STACKSIZE      2048
#define TASK1_PRIORITY      2
#define TASK1_ID            1
#define TASK1_Arrival       2
#define TASK1_exection      5
#define TASK1_period        30
//task2
#define TASK2_PRIORITY      3
#define TASK2_ID            2
#define TASK2_Arrival       3
#define TASK2_exection      3
#define TASK2_period        60
////task3
#define TASK3_PRIORITY      5
#define TASK3_ID            3
#define TASK3_Arrival       0
#define TASK3_exection      7
#define TASK3_period        90


OS_EVENT* R1;
OS_EVENT* R2;
static OS_STK       Task1_STK[TASK_STACKSIZE];
static OS_STK       Task2_STK[TASK_STACKSIZE];
static OS_STK       Task3_STK[TASK_STACKSIZE];

static void task1(void * p_arg, void* pdata);
static void task2(void * p_arg, void* pdata);
static void task3(void* p_arg, void* pdata);
static void mywait(int tick);
```

Fig 4. Resource & task setting

然後設定task和resource的基本定義，以及會用到的function，如Fig 4.所示。



```
INT8U err;
R1 = OSMutexCreate(R1_PRIO, &err);
R2 = OSMutexCreate(R2_PRIO, &err);
//建立task
OSTaskCreateExt(task1,
    0,
    &Task1_STK[TASK_STACKSIZE - 1],
    TASK1_PRIORITY,
    TASK1_ID,
    &Task1_STK[0],
    TASK_STACKSIZE,
    0,
    (OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR),
    TASK1_period, TASK1_Arrival);

OSTaskCreateExt(task2,
    0,
    &Task2_STK[TASK_STACKSIZE - 1],
    TASK2_PRIORITY,
    TASK2_ID,
    &Task2_STK[0],
    TASK_STACKSIZE,
    0,
    (OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR),
    TASK2_period, TASK2_Arrival);

OSTaskCreateExt(task3,
    0,
    &Task3_STK[TASK_STACKSIZE - 1],
    TASK3_PRIORITY,
    TASK3_ID,
    &Task3_STK[0],
    TASK_STACKSIZE,
    0,
    (OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR),
    TASK3_period, TASK3_Arrival);
```

Fig 5. Resource & task create

Resource建立的方式與原本task的方式不太一樣，不必用到OSMutexCreate去建立，如Fig 5.所示。

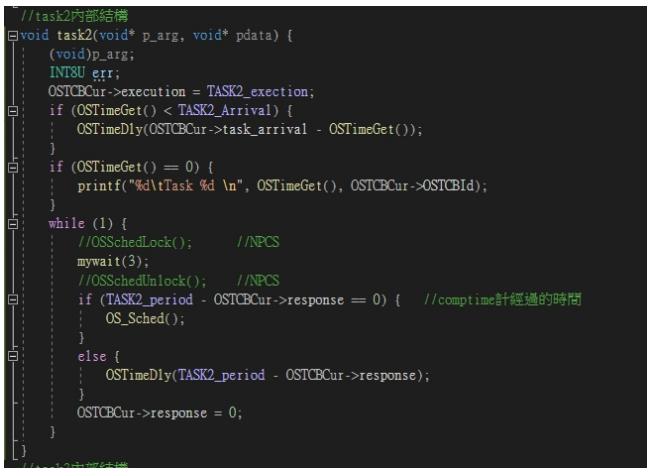
```
//task1內部結構
void task1(void* p_arg, void*pdata) {
    (void)p_arg;
    INT8U err;
    OSTCBCur->execution = TASK1_exection;
    if (OSTimeGet() < TASK1_Arrival) {
        OSTimeDly(OSTCBCur->task_arrival - OSTimeGet());
    }
    if (OSTimeGet() == 0) {
        printf("%d\tTask %d \n", OSTimeGet(), OSTCBCur->OSTCBId);
    }
    while (1) {
        mywait(1);
        //OSSchedLock();      //NPCS
        printf("%d\tTask %d get R1",OSTimeGet(), OSTCBCur->OSTCBId);
        OSMutexPend(R1,0,&err);
        mywait(3);
       /* printf("%d\tTask %d get R2", OSTimeGet(), OSTCBCur->OSTCBId);
        OSMutexPend(R2, 0, &err);
        mywait(2);*/
        /*printf("%d\tTask %d release R2", OSTimeGet(), OSTCBCur->OSTCBId);
        OSMutexPost(R2);*/
        printf("%d\tTask %d release R1", OSTimeGet(), OSTCBCur->OSTCBId);
        OSMutexPost(R1);
        //OSSchedUnlock();    //NPCS
        OS_Sched();
        mywait(1);
        int time = OSTCBCur->response;
        OSTCBCur->response = 0;          //清除經過時間

        if (TASK1_period - time == 0) {
            OS_Sched();
        }
        else {
            OSTimeDly(TASK1_period - time);
        }

    }
}
```

Fig 6. 有包含resource的task

這次有分成兩種不同的方式(CPP、NPCS)，在NPCS中，我使用到OSSchedlock去鎖住正在使用的resource防止其他task來占用，而Fig 6. 的NPCS code由註解的//NPCS 到 //NPCS之間，就如上述所說，防止當我正在使用資源時，被其他task搶佔。



```
//task2內部結構
void task2(void* p_arg, void* pdata) {
    (void)p_arg;
    INT8U err;
    OSTCBCur->execution = TASK2_exection;
    if (OSTimeGet() < TASK2_Arrival) {
        OSTimeDly(OSTCBCur->task_arrival - OSTimeGet());
    }
    if (OSTimeGet() == 0) {
        printf("%d\tTask %d \n", OSTimeGet(), OSTCBCur->OSTCBId);
    }
    while (1) {
        //OSSchedLock();      //NPCS
        mywait(3);
        //OSSchedUnlock();    //NPCS
        if (TASK2_period - OSTCBCur->response == 0) {   //comptime計經過的時間
            OS_Sched();
        }
        else {
            OSTimeDly(TASK2_period - OSTCBCur->response);
        }
        OSTCBCur->response = 0;
    }
}
```

Fig 7. 沒有包含resource的task

Fig 6. 是講說這個task有使用資源的情況，Fig 7.是指沒有使用資源的情況，因為沒有使用，所以也沒有使用OSSchedlock去鎖住此task。

```
//模擬執行時間
void mywait(int tick) {

#if OS_CRITICAL_METHOD==3;
    OS_CPU_SR cpu_sr = 0;
#endif
    while (1)
    {
        if (tick <= OSTCBCur->execution_frequency)
            break;
    }
    OS_ENTER_CRITICAL();
    OSTCBCur->CompTime += OSTCBCur->execution_frequency;
    OSTCBCur->execution_frequency = 0;
    OS_EXIT_CRITICAL();

}
```

Fig 8. 模擬使用時間的function

Fig 8. 為模擬使用時間的function，因為exeuction_frequency是task裡的資料結構的參數，而每次進來做的次數是分開計算，因此做完時都要進行重製。

```
//判斷有ready卻沒執行的task，每一個tick會持續增加response，以及deadline，此功能在OSTimeTick設定
for (INT8U y = 0; y < 8; y++) {
    for (INT8U x = 0; x < 8; x++) {
        if ((OSRdyTbl[y] & (INT8U)1 << x) == (INT8U)1 << x) {   //確認是否有ready
            INT8U prio = (y << 3u) + x;
            if (OSTCBPrioTbl[prio] != OSTCBCur && prio != OS_TASK_IDLE_PRIO) {
                if (OSTimeGet() > OSTCBPrioTbl[prio]->task_arrival) {
                    OSTCBPrioTbl[prio]->response += 1;
                    if (OSTCBPrioTbl[prio]->response > OSTCBPrioTbl[prio]->Deadline) {
                        printf("%d\t MissDealine\t Task(%d)(%d)\t\t----------\n", OSTimeGet(),
                            OSTCBPrioTbl[prio]->OSTCBId, OSTCBPrioTbl[prio]->job_ID);
                        OSTCBPrioTbl[prio]->response = 0;
                        OSTCBPrioTbl[prio]->contextswitch = 0;

                    }
                }

            }
        }
    }
}
if (OSPrioCur != OS_LOWEST_PRIO) {
    OSTCBCur->response += 1;
    OSTCBCur->execution_frequency += 1;     //經過一個ticktime ++執行時間
}
```

Fig 9. TimeTick

在Fig 9.所示，只要task的arrival時間有到達，上面就會去找已經有ready但沒有執行的task的response++，下面則是除了response++，還有執行次數++，以及missdeadline的判斷。

```
//OSIntExit設定
//判斷是否為最高prio
for (INT8U y = 0; y < 8; y++) {
    for (INT8U x = 0; x < 8; x++) {
        if ((OSRdyTbl[y] & (INT8U)1 << x) == (INT8U)1 << x) {   //確認是否有ready
            INT8U prio = (y << 3u) + x;
            if (OSTCBPrioTbl[prio] != OSTCBCur && prio != OS_TASK_IDLE_PRIO) {
                if (OSTimeGet() >= OSTCBPrioTbl[prio]->task_arrival) {
                    if (OSTCBCur->OSTCBPrio > OSTCBPrioTbl[prio]->OSTCBPrio)
                        c = 1;
                }

            }
        }
    }
}
if (c == 0) { //判斷是否目前task為最高prio
    OS_TRACE_ISR_EXIT();
}
else if (OSTCBCur->execution_frequency == OSTCBCur->execution && OSPrioCur != OS_TASK_IDLE_PRIO ) {
    //printf("OSTCBCurId: %d OSTCBCur->execution_frequency %d , OSTCBCur->execution %d \n", OSTCBCur->OSTCBId,OSTCBCur->execution_frequency, OSTCBCur->execution);
    OS_TRACE_ISR_EXIT();
}
else {
    OS_SchedNew();
    OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
    if (OSPrioHighRdy != OSPrioCur )
    {                           /* No Ctx Sw if current task is highest rdy */
#if OS_TASK_PROFILE_EN > 0u
        OSTCBHighRdy->OSTCBCtxSwCtr++;          /* Inc. # of context switches to this task  */
#endif
        OSCtxSwCtr++;                           /* Keep track of the number of ctx switches */

#if OS_TASK_CREATE_EXT_EN > 0u
#if defined(OS_TLS_TBL_SIZE) && (OS_TLS_TBL_SIZE > 0u)
        OS_TLS_TaskSw();
#endif
#endif

        OS_TRACE_ISR_EXIT_TO_SCHEDULER();
        //OSIntExit，被中斷完之後，顯示要進去的task資訊
        if (OSTimeGet() > OSTCBCur->task_arrival) {
            printf("%d\tTask %d \n", OSTimeGet(), OSTCBHighRdy->OSTCBId);

        }
        OSIntCtxSw();                           /* Perform interrupt level ctx switch     */
```

Fig 10. OSIntExit

這邊除了顯示，每次被中斷的後，下一個task的顯示，還包括會去看說，目前task是否為最高priority，如果是就跳出，如果不是就去找最高權限的priority。

```
//當進到OS_Sched裡，判斷下個task的顯示
if (OSTimeGet() >= OSTCBCur->task_arrival) {
    //OSTCBCur->CompTime = OSTimeGet();
    //OSTCBHighRdy->contextswitch += 1;
    //OSTCBCur->contextswitch += 1;
    if (OSPrioHighRdy == 63) {
        printf("%d\tTask 63 \n", OSTimeGet());
    }
    else {
        printf("%d\tTask %d \n", OSTimeGet(), OSTCBHighRdy->OSTCBId);
    }
    //OSTCBCur->contextswitch = 0;
    OSTCBCur->execution_frequency = 0;
    //OSTCBCur->response = 0;
    //OSTCBCur->job_ID += 1;
}
```

Fig 11. OS_Sched

這邊跟上面一樣，都是顯示下一個最高權限的priority顯示，由於跟以往的RMS有點不太一樣，有些清除的動作，都改到其他地方。

[PART II] CPP Implementation

- The screenshot result (with the given format) of the two task sets. (Time tick 0-100) (10%)

```
選取 C:\Users\a0939\OneDrive\桌面\Micrium_Win32_Kernel - PA1\Microsoft\W
OSTick    created, Thread ID 17768

Tick    Event                   Prio_Inheritance
0       Task 3
1       Task 3 get R2           5->4
2       Task 1
3       Task 1 get R1           2->1
6       Task 1 release R1       1->2
7       Task 2
10      Task 3
14      Task 3 release R2       4->5
15      Task 63
32      Task 1
33      Task 1 get R1           2->1
36      Task 1 release R1       1->2
37      Task 63
62      Task 1
63      Task 1 get R1           2->1
66      Task 1 release R1       1->2
67      Task 2
70      Task 63
90      Task 3
91      Task 3 get R2           5->4
92      Task 1
93      Task 1 get R1           2->1
96      Task 1 release R1       1->2
97      Task 3
101     Task 3 release R2       4->5
102     Task 63
122     Task 1
123     Task 1 get R1           2->1
126     Task 1 release R1       1->2
127     Task 2
130     Task 63
152     Task 1
153     Task 1 get R1           2->1
156     Task 1 release R1       1->2
157     Task 63
```

Fig 12. CPP Task Set1 ={task1(2,5,30), task2(3,3,60), task3(0,7,90)}

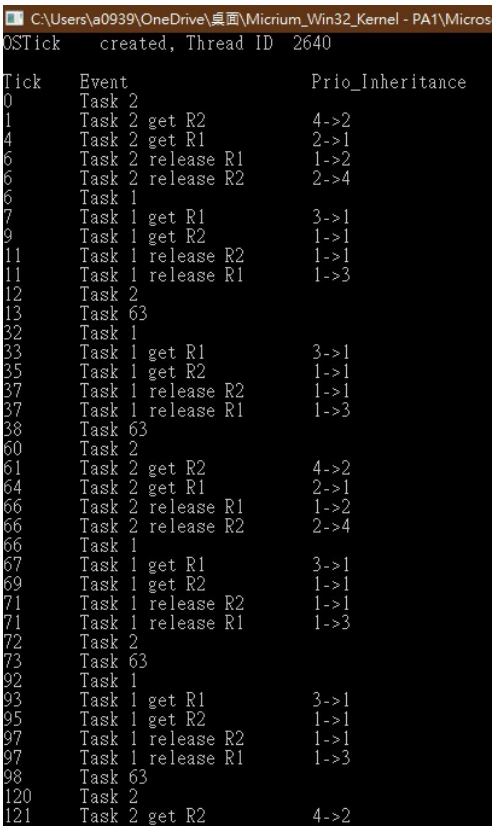


Fig 13. CPP Task Set2 = {task1(2,6,30), task2(0,7,60)}

- A report that describes your implementation, including scheduling results of two task sets, modified functions, data structure, etc. (please **ATTACH** the screenshot of the code and **MARK** the modified part).

基本的task、resource與Fig 4. 到 Fig 6.基本上一樣，依照每提的要求會有一些變化，但這邊我在設定R1、R2的priority有一些變化，如Fig 12.我的priority就設定R1=1、R2＝4，是因為要對照使用這個資源的最高權限的task，但在做Fig 13.時因為兩個task都有用到相同的resource因此，我在設定R1和R2都必須高於兩個task，但由於OS2不能有相同的priority，因此我才設R1=1、R2=2，結果就如Fig 13.所示，task的內部結構，如Fig 6. 和 Fig 7.所示，只差別在於沒有使用到OSSchedlock，模擬的時間function如Fig 8. 所示，ostimetick、os_sched、osintexit，的寫法如Fig 9.、Fig 10.、Fig 11.所示。比較不一樣的地方在於下面兩張圖(Fig 14. 、Fig 15.)，作為CPP的重要判斷方式，並在撰寫繼承的過程，以及release的priority的過程。

```
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 owni
    pevent->OSEventPtr  = (void *)OSTCBCur;            /*      Point to owning task'

    //判斷繼承的prio大或小，在OSMutexPend設定
    if (OSTCBCur->OSTCBPrio > pcp) {
        printf("\t\t%d->%d\n", OSTCBCur->OSTCBPrio, pcp);
        OSTCBCur->OSTCBPrio = pcp;
    }
    else {
        pcp = OSTCBCur->OSTCBPrio;
        printf("\t\t%d->%d\n", OSTCBCur->OSTCBPrio, pcp);
    }
```

Fig 14 . OSMutexPend

```
OS_ENTER_CRITICAL();
pcp  = (INT8U)(pevent->OSEventCnt >> 8u);          /* Get priority ceiling priority of mutex        */
prio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);  /* Get owner's original priority      */
//釋放資源並回到上的狀態，在OSMutexPost設定
printf("\t%d->%d\n", OSTCBCur->OSTCBPrio, prio);
```

Fig 15 . OSMutexPost

[PART III] Performance Analysis

- Compare the scheduling behaviors between NPCS and CPP with the results of PART I and PART II.

  在NPCS中，是只要我task正在使用資源時，其他task都沒辦法中斷他，除非他將資源釋放掉，其他task才有辦法去中斷他。在CPP而言，他會先判斷這些資源的最高權限，每次task使用這個資源會去繼承這個資源的priority(除非資源priority比task還低就不繼承)，但是有別於NPCS是即使你使用他但是其他比priority高的task搶佔(但必須高於不能等於)，還是會發生，但如果發生像Fig 13.的問題，最高權限的task都有使用到這些資源的情況下，理論上，他的行為會跟NPCS很像，因為另一個task要大過他的權限，但是R1、R2都是最高權限，因此沒被搶佔，除非他把資源釋放才可以。總結一下，NPCS正在使用資源誰都不能搶佔，CPP是只要高於他的task還是會發生搶佔。

- Explain how NPCS and CPP avoid the deadlock problem.
  NPCS 當有在使用到資源時，會把中斷的功能關掉，當資源釋放，再把中斷功能開啟。
  CPP 資源 ceiling 取決於最高的 task 優先權，如果發生中斷時，要中斷的 task 必須高於目前的 ceiling 才可以中斷，否則無法中斷，就可以避免 deadlock 的問題。