



國立台灣科技大學

嵌入式作業系統實作

指導教授：陳雅淑教授

嵌入式作業系統實作 Embedded OS Implementation

Project 03

班 級 ： 電機碩一
學 生 ： 江昱霖
學 號 ： M10307431

I. 系統環境設定(system.h)

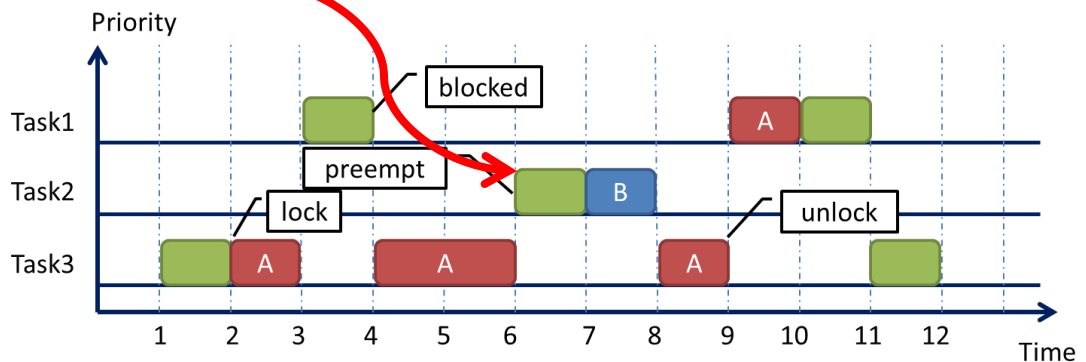
- 關閉系統常駐任務 Statistic Task (不需執行此任務)

```
#define OS_DEBUG_EN 1
#define OS_SCHED_LOCK_EN 1
#define OS_TASK_STAT_EN 0
#define OS_TASK_STAT_STK_CHK_EN 1
#define OS_TICK_STEP_EN 1
```

II.SRP (Stack Resource Policy) Implement

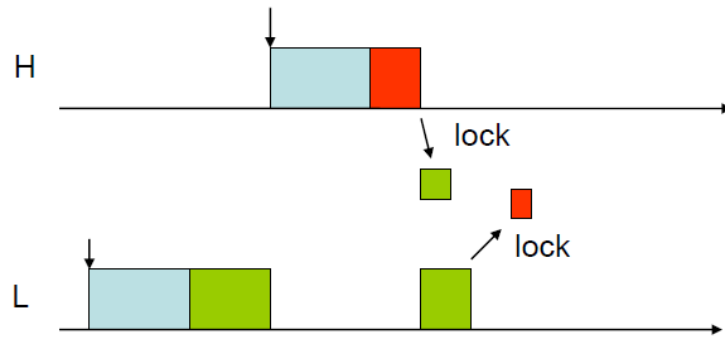
由於系統採用最早截止時間優先演算法 EDF(Earliest Deadline First)是非常著名的實時排程演算法，一種優先搶占式的排程法，能夠保證當前運行最高優先權的任務。然而在實際運用上，任務間的資源共享，佔有資源往往會造成低優先權任務長時間佔有或阻塞高優先任務，高優先權任務反而需要等待低優先權任務執行完畢，這可能會造成系統的不確定性甚至崩潰，這即是優先權反轉的問題。

● Priority Inversion



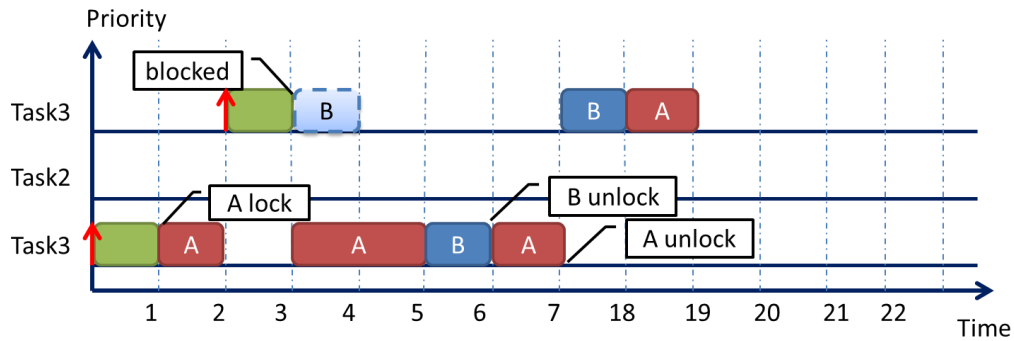
為解決反轉問題，發展出了 PIP(Priority Inherit Protocol)，當有高優先權也需要取得資源時，將持有資源的低優先權任務去繼承高優先權任務的優先權，避免中優先權任務搶佔的可能性，減少阻塞等待的時間。但此方式無法解決 Deadlock 的問題。當雙方任務分別持有資源尚未釋放，又同時需要取得對方資源時，就會陷入雙方互相等待對方釋放資源的死胡同，這就是 Deadlock 問題。

● Deadlock



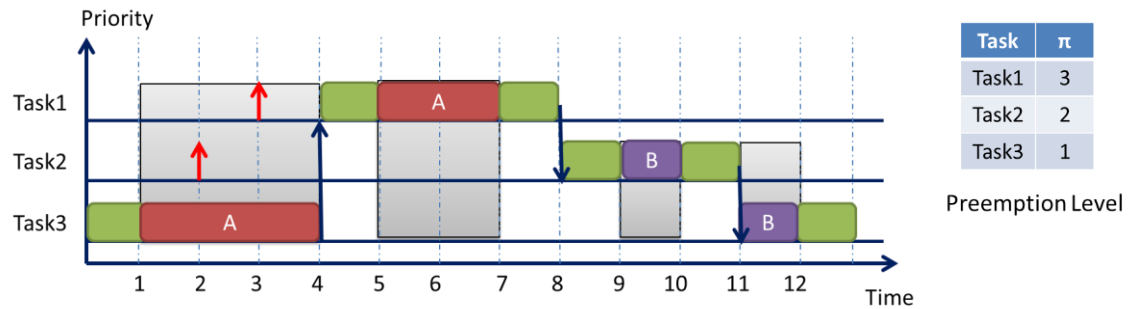
因此進一步發展出了 PCP(Priority Ceiling Protocol)，當任務在拿取資源時需確認是否能夠一次拿取全部的所需資源，否則就可能陷入 deadlock 的情況，每個資源會依據被任務存取的優先權高低訂定出 Ceiling，用來保證任務存取的優先性，避免多個任務分別持有又互相等待的情況。

● PCP



由於 PCP 是以任務的優先權來訂定 Ceiling，因此適用在固定優先權的 RM 上，而動態優先權的 EDF 則無法適用。所以發展出了 SRP (Stack Resource Policy)，根據任務的週期訂定每個任務的 Preemption Level，其代表的意義就如同 PCP 中的 Priority 一樣，任務對於資源存取的優先順序，只是在 EDF 中任務的優先權會依據時間不同而改變，所以 PCP 才無法適用，SRP 則是依據持有資源任務的 Preemption Level 來訂定系統的 Ceiling，確保在存取資源時能夠一次取得所需資源而不被他人阻塞甚至造成鎖死。且 PCP 是在存取資源時檢查 Ceiling 與自身權限的關係，而 SRP 則是在任務開始執行時就檢查權限，如此一個任務至多就是 2 次的 Context Switch，優化了任務在存取資源時才發現權限不足需要切換任務的情況，能夠省去許多不必要的任務切換。

● SRP



為實作 SRP 於 uCOS II 上，我們可以使用 Mutex 這個功能來做資源存取的保護與控管，由於 uCOS II 不支援多個任務同時擁有相同優先權，因此我們必須做繼承優先權的定義與管理。Mutex 本身即具有優先權繼承的功能，當某一資源已被低優先權持有，且有高優先權也需存取此資源時，Mutex 會將低優先權任務的優先權繼承為使用者定義的優先權級別，達到優先權繼承的概念，確保任務不會被中斷並保護資源的使用。

● 任務參數創建及初始化設定

由於 SRP 需使用到繼承的部分，而系統是建立在 EDF 排程下，所以需要將原始的截止時間記錄起來，以便在釋放資源時回復原優先權的使用。且加入了 Preemption Level 用來定義該任務對資源的權限。因此在 TCB 下加入截止時間與 SRP 的相關參數及修改函式達到任務的設定及初始化。

1. 修改 TCB (ucos_ii.h)，加入執行時間、週期、截止時間、原始截止時間、搶占等級以及持有資源旗標，用來計算當前任務狀態及顯示的相關資訊。

```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;           /* Pointer to task stack */
    INT16U          ExecTime;               /* Exec time */
    INT16U          Exec;                   /* Exec flag */
    INT16U          Period;                 /* Period */
    INT16U          Deadline;               /* Deadline */
    INT16U          Org_Deadline;           /* original deadline */
    INT16U          Preempt_lv;             /* preemption level */
    INT8U           GetRes;                 /* get resource ? */
}
```

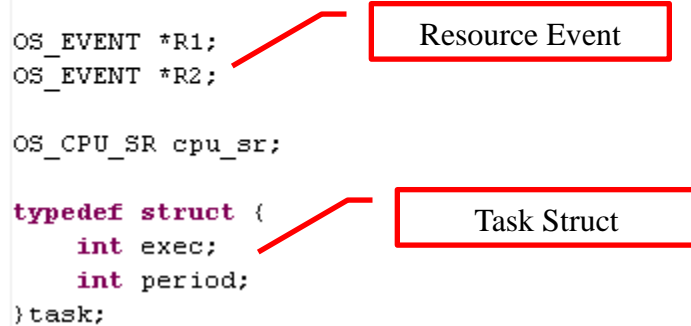
2. 定義優先權參數及創建任務參數資料結構 task，作為創建任務的傳入參數，用來初始化任務的設定，並建立資源的事件結構。

```
/* Definition of Task Priorities */
#define R1_PRIO          1
#define R2_PRIO          2
#define TASK1_PRIORITY   11
#define TASK2_PRIORITY   12
#define TASK3_PRIORITY   13

OS_EVENT *R1;
OS_EVENT *R2;

OS_CPU_SR cpu_sr;

typedef struct {
    int exec;
    int period;
}task;
```



3. 定義執行時間函式，用來模擬任務執行。

```
void mywait(int tick)
{
    int now, exit;
    OS_ENTER_CRITICAL();
    now = OSTimeGet();
    exit = now + tick;
    OS_EXIT_CRITICAL();
    while(1) {
        if(exit <= OSTimeGet())
            break;
    }
}
```

4. 定義任務函式，以無限迴圈來表示週期性的任務，並利用先前建立的等待函式來模擬任務的執行時間，使用 Mutex 來實現任務間共享資源的控管，用 delay 函式來模擬任務等待下個週期的到達。

```
void task1(task* pdata)
{
    int end;
    int delay;
    INT8U err;
    int arrival = 3;
    OSTCBCur->Period = pdata->period;
    OSTCBCur->Exec = pdata->exec;
    OSTCBCur->ExecTime = pdata->exec;
    OSTCBCur->Deadline = pdata->period + arrival;
    OSTCBCur->Org_Deadline = pdata->period + arrival;
    OSTimeDly(arrival);
    while (1)
    {
        printf("\t%d\tTask_1\n", OSTimeGet());
        mywait(OSTCBCur->Exec); // CPU time
        printf("\t%d\tTask_1 get R2\t", OSTimeGet());
        OSMutexPend(R2, 0, &err);
        mywait(5); // R2
        printf("\t%d\tTask_1 get R1\t", OSTimeGet());
        OSMutexPend(R1, 0, &err);
        mywait(3); // R1
        printf("\t%d\tTask_1 release R1", OSTimeGet());
        OSMutexPost(R1);
        mywait(0); // R2
        printf("\t%d\tTask_1 release R2", OSTimeGet());
        OSMutexPost(R2);
        end = OSTimeGet();
        delay = OSTCBCur->Deadline-end; // 1e
        OSTCBCur->Deadline += OSTCBCur->Period;
        OSTCBCur->Org_Deadline = OSTCBCur->Deadline;
        OSTimeDly(delay);
    }
}
```

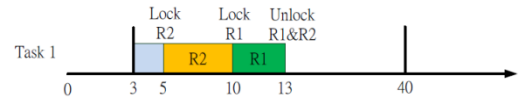
任務參數初始化

到達時間設定

任務執行迴圈

拿取資源

釋放資源



5. 創建 Mutex 來保護資源，並傳入該資源被存取的優先權限。

```
R1 = OSMutexCreate(R1_PRIO, &err);
R2 = OSMutexCreate(R2_PRIO, &err);
```

6. 創建任務，傳入先前加入的任務資料結構，並修改該函式傳入截止時間做為初始化設定。

```
OSTaskCreateExt(task1,
                &t1,
                (void *)&task1_stk[TASK_STACKSIZE-1],
                TASK1_PRIORITY,
                TASK1_PRIORITY,
                task1_stk,
                TASK_STACKSIZE,
                NULL,
                0,
                t1.period // deadline
                );
```

修改創建任務函式(ucos ii.h\os_task.c)，並在 OS_TCBinit(os_core.c) 中做截止時間的初始。

```
#if OS_TASK_CREATE_EXT_EN > 0
INT8U OSTaskCreateExt (void (*task)(void *p_arg),
                      void *p_arg,
                      OS_STK *ptos,
                      INT8U prio,
                      INT16U id,
                      OS_STK *pbos,
                      INT32U stk_size,
                      void *pext,
                      INT16U opt,
                      INT32U deadline);

INT8U OS_TCBInit (INT8U prio, OS_STK *ptos, OS_STK *pbos, INT16U id, INT32U stk_size,
{
    OS_TCB *ptcb;
    #if OS_CRITICAL_METHOD == 3 /* Allocate storage for CPU
        OS_CPU_SR cpu_sr = 0;
    #endif

    ptcb->Deadline = deadline; // Deadline Initialization
```

● uC/OS ii Kernel 修改

基於 EDF 排程的系統中，SRP 會依據 Preemption Level 與 Ceiling 來控管資源的使用，優先權繼承的部分 Mutex 已經有提供基本的操作了，還有截止時間的繼承是我們所需要實作的。因此在任務有異動(加入/完成)時就必須判斷目前的就緒任務中是否有截止時間較早的任務需要被繼承其截止時間。必須 OSMutexPend 以及 OSMutexPost 中來調整 Ceiling，並在 Scheduler 中加入 Ceiling 的判斷，以便正確的調度任務。

1. 首先宣告全域變數 Ceiling，反映目前資源使用的優先權高低。(ucosii.h)，並在系統初始時設定為 0。

```
*****  
*                                                                 GLOBAL VARIABLES  
*****  
*/  
OS_EXT  INT8U   Ceiling;|  
OS_EXT  INT32U          OSTxSwCtr;          /* Counter  
  
void OSInit (void)  
{  
    Ceiling = 0;  
}                                     (os_core.c)
```

2. 在 EDF 排程下，需要隨時判斷是否有高優先權的工作就緒，為防止被搶占須完成截止時間繼承的步驟，使低優先權任務能夠完整完成資源的使用，並不被搶占，達到 SRP 的基本目的。在 TimeTick 中去判斷是否有截止時間較早的工作就緒，若有則繼承其截止時間，使其在 EDF 排程下達到 SRP 不被搶佔的目的。

```
    if(ptcb->Deadline < min_deadline && ptcb->OSTCBDly == 0){  
        min_deadline = ptcb->Deadline;  
        //printf("%d\n",ptcb->OSTCBPrio);  
    }  
    ptcb = ptcb->OSTCBNext;          /* Point at next  
    OS_EXIT_CRITICAL();  
}  
//printf("%d\n",min_deadline);  
if(OSTCBCur->Deadline != min_deadline && OSTCBCur->GetRes == 1){  
    OSTCBCur->Deadline = min_deadline; // inherit deadline  
    printf("\t\t%d\tTask_%d inherit deadline\t\t\tdeadline %d\n",OSTime
```

搜尋較短 deadline

Deadline 繼承

3. 在 MutexPend 時，將 Ceiling 提升為 3 (最高為 3)，並判斷是否持有多個資源，並將其記錄在 ECB 中，若是持有多個資源意味在 Post 時，只有在釋放最後一個資源時才能將 Ceiling 降為 0，其他都將維持 3 表示手上仍擁有資源尚未釋放。(os_mutex.c)

```
OS_ENTER_CRITICAL();
pip = (INT8U) (pevent->OSEventCnt >> 8);

if ((INT8U) (pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_
    pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;
    pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;
    pevent->OSEventPtr = (void *) OSTCBCur;

    if (Ceiling == 3)
        pevent->chain = 1; // pend multi-resources
    else
        pevent->chain = 0; // first pend
    Ceiling = 3;           // raise system ceiling
    OSTCBCur->GetRes = 1;  // Get Resources
    printf("\t%d\n", Ceiling);
```

持有多資源

4. 在 MutexPost 時，依多資源持有情況來決定是否要將 Ceiling 降為 0，因先前的截止時間繼承關係，釋放資源時須將截止時間還原，並更新資源持有的情況。

```
if (pevent->chain == 0) { // Release all resources, Reset Ceiling
    Ceiling = 0;
    OSTCBCur->Deadline = OSTCBCur->Org_Deadline; // Restore Deadline
    OSTCBCur->GetRes = 0;
    OSTCBCur->ExecTime = 0;
}
printf("\t%d\n", Ceiling);
```

Deadline 還原

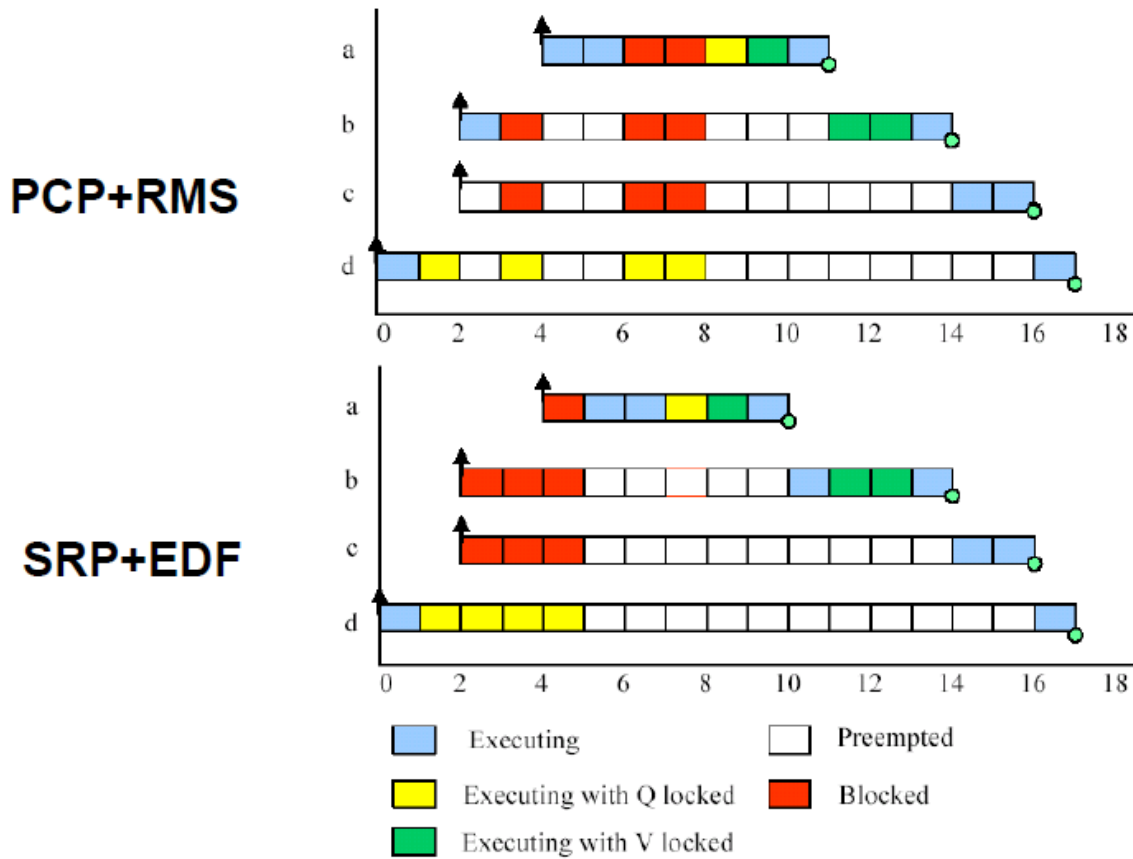
5. 為實現 EDF 排程，修改 Scheduler 尋找最高優先權的部分，以最早截止時間為優先(OS_SchedNew)，搜尋就緒任務中截止時間最早的任務優先排程，若截止時間相同，則讓周期較長的任務優先執行，會有較好的響應時間及較少的 context switch 次數，同時須考慮 SRP 的 Ceiling，判斷 Preemption Level 是否有大於 Ceiling。

```
INT8U UrgentPrio = OS_LOWEST_PRIO; // the most urgent priority */
INT16U UrgentDeadline = 65535u;
INT8U i=0;
while (i < OS_LOWEST_PRIO) { // search all
    if (OSTCBPrioTbl[i] != (OS_TCB*)0 && OSTCBPrioTbl[i] != OS_TCB_RESERVED) { //
        //printf("%d, ", OSTCBPrioTbl[i]->OSTCBPrio);
        if (OSTCBPrioTbl[i]->OSTCBDly==0 && (OSTCBPrioTbl[i]->Preempt_lv > Ceiling || OSTCBCur->OSTCBPri

            if ((OSTCBPrioTbl[i]->Deadline < UrgentDeadline) || (OSTCBPrioTbl[i]->Deadline == UrgentDeadli
                UrgentDeadline = OSTCBPrioTbl[i]->Deadline;
                UrgentPrio = OSTCBPrioTbl[i]->OSTCBPrio;
            }
        }
        i++;
    }
}
OSPrioHighRdy = UrgentPrio;
```

Ceiling 判斷

● 問題與討論



由於 PCP 與 SRP 的任務只會被 block 一次，因當持有資源低優先權任務釋放資源後，高優先權能夠被排程馬上執行而不會被其他低優先全的任務 block，所以 PCP 與 SRP 的 block time 是相同的，唯一不同在於任務檢查權限的時間點，PCP 在拿取資源時才會檢查權限，而 SRP 則是在任務就緒時就檢查，因此任務能夠完整的執行，中途不被中斷，一個任務至多只會有 2 次的 Context Switch，而 PCP 可能會有 2 次以上的任務切換而造成額外的 Overhead。由上圖的例子中可以看到不管是 PCP 或 SRP 其 blocking time 是一樣的，只是時間點不同，而 PCP 中可以看出由於是在存取資源時才檢查權限，發現權限不足時又會任務切換至原任務執行等待資源釋放，所以造成了許多不必要的 context switch，但是我們可以看出 PCP 多工任務的平行度是比 SRP 好的。

● 結果(Time Tick 0 - 70)

1. Task set 1 模擬:

===== SRP Task Set 1 { t1(3,37) , t2(0,45) } (Arrival,Period) =====			
Current Time	Event	System Ceiling	
0	Task_2		
2	Task_2 get R1	3	
3	Task_2 inherit deadline		deadline 40
7	Task_2 get R2	3	
12	Task_2 release R2	3	
12	Task_2 release R1	0	
12	Task_1		
14	Task_1 get R2	3	
19	Task_1 get R1	3	
22	Task_1 release R1	3	
22	Task_1 release R2	0	
40	Task_1		
42	Task_1 get R2	3	
47	Task_1 get R1	3	
50	Task_1 release R1	3	
50	Task_1 release R2	0	
50	Task_2		
52	Task_2 get R1	3	
57	Task_2 get R2	3	
62	Task_2 release R2	3	
62	Task_2 release R1	0	
77	Task_1		
79	Task_1 get R2	3	
84	Task_1 get R1	3	

2. Task set 2 模拟:

===== SRP Task Set 2 (t1(9,60), t2(3,70), t3(0,80)) (Arrival,Period) =====				
Current Time	Event	System Ceiling		
0	Task_3			
2	Task_3 get R1	3		
3	Task_3 inherit deadline		deadline 73	
7	Task_3 get R2	3		
9	Task_3 inherit deadline		deadline 69	
12	Task_3 release R2	3		
12	Task_3 release R1	0		
12	Task_1			
16	Task_1 get R2	3		
20	Task_1 get R1	3		
23	Task_1 release R1	3		
23	Task_1 release R2	0		
23	Task_2			
24	Task_2 get R2	3		
27	Task_2 release R2	0		
69	Task_1			
73	Task_1 get R2	3		
77	Task_1 get R1	3		
80	Task_1 release R1	3		
80	Task_1 release R2	0		
80	Task_2			
81	Task_2 get R2	3		

● Schedulability Analysis

1. Task set 1

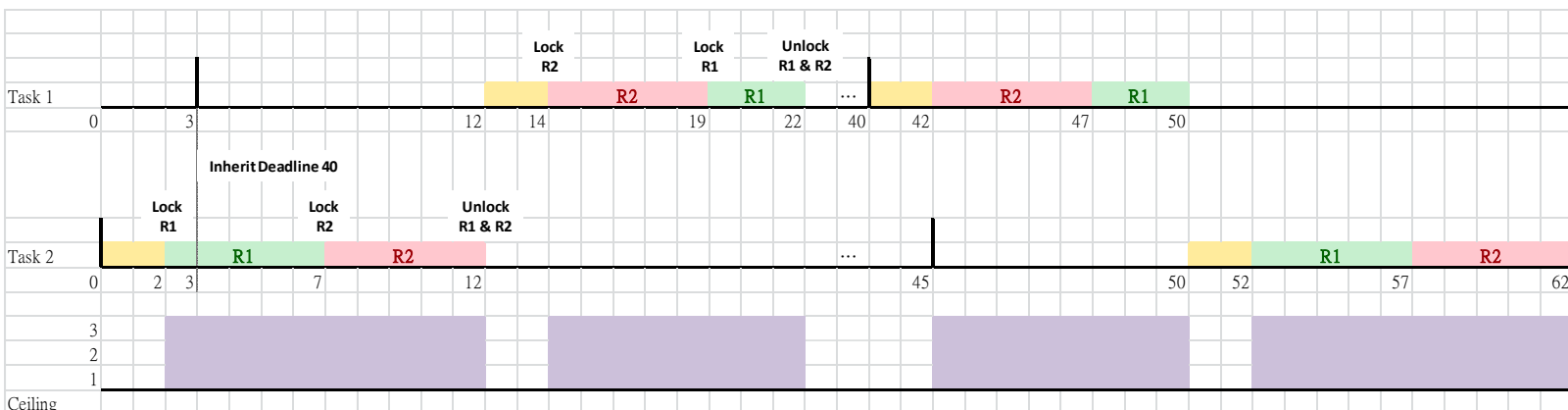
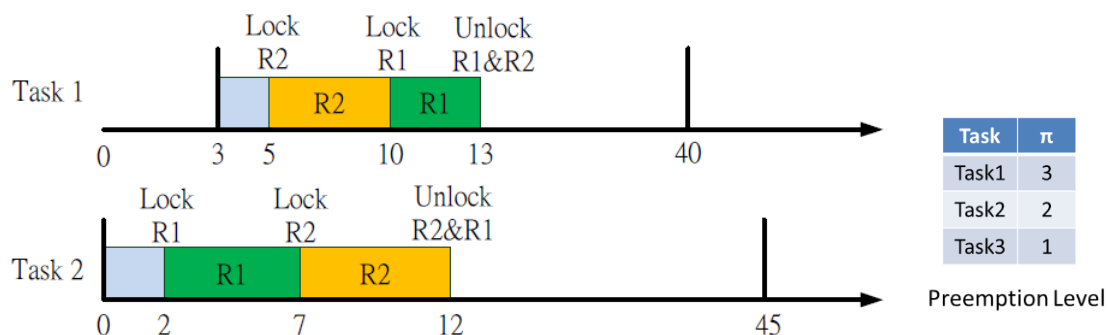


圖 1 (a) Input (b) Output

Task set 1 即是典型的 Deadlock，task 2 先持有了 R1，接著 task 1 抵達並持有了 R2，然後 task 1 又要拿取 R1，但 R1 已被 task2 所持有，當返回 task 2 執行又叫拿取 R2，接下來就陷入雙方互等對方釋放資源的情況。使用 SRP 來解決資源共享的問題時，由上圖可以知道 task 2 先持有了 R1，Ceiling 上升到了 3(最高)，當 $t=3$ 時，task 1 抵達，此時發現 task 1 的 preemption level 並無高於系統 Ceiling，因此無法搶占執行，同時 task 2 會去繼承 task 1 的 deadline，由於 task 1 無法搶占資源，所以 task 2 能夠一次存取所有所需資源到結束，接著會調度高優先權的任務執行，也就不會發生分別持有資源而互等對方釋放資源造成崩貴的情況。

2. Task set 2

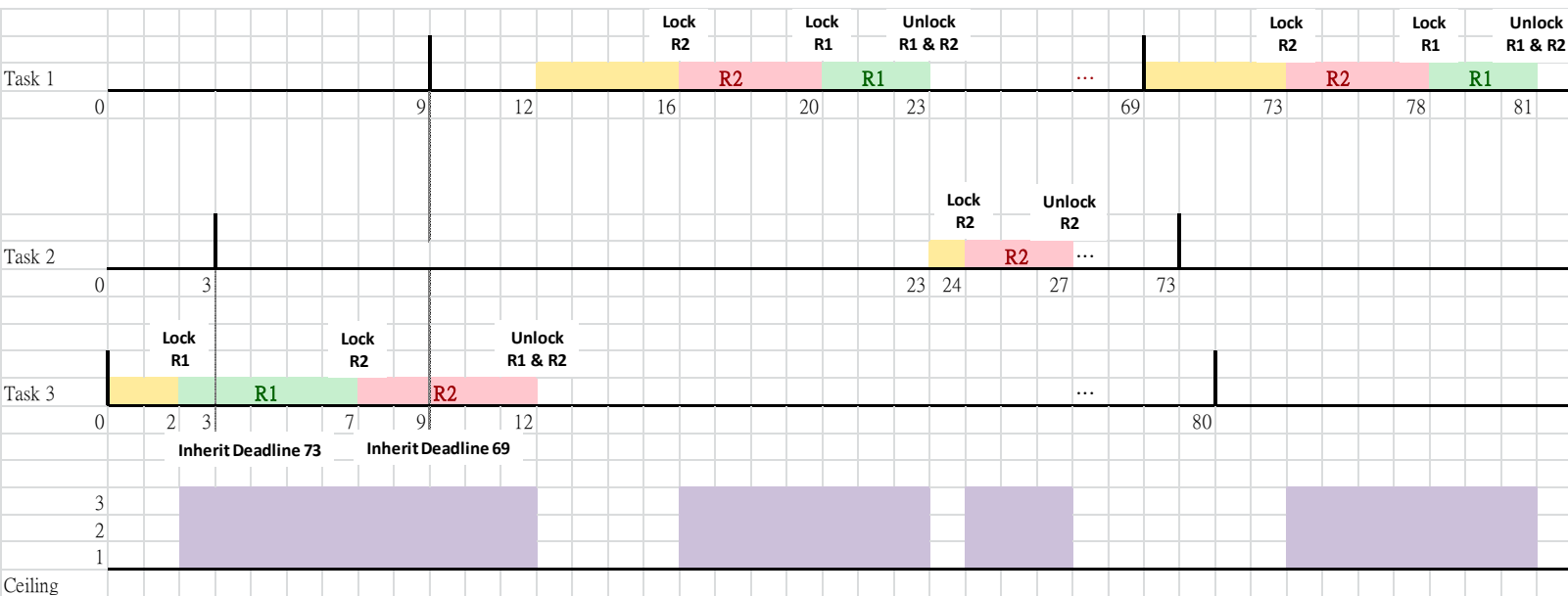
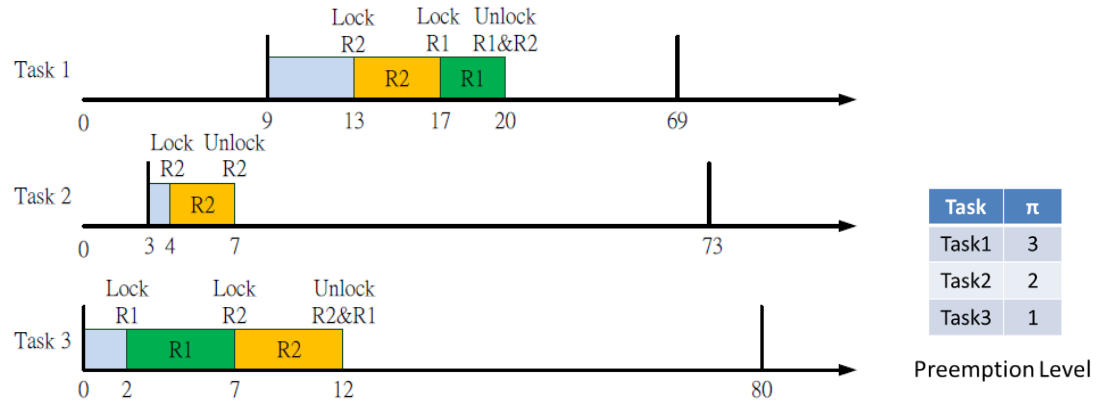


圖 2 (a) Input (b) Output

Task set 2 可能造成 chain blocking 的情況，由最低優先權任務先抵達並持有資源，在釋放資源前，較高優先權的任務繼續抵達並也要求存取資源，所以導致最高優先權任務抵達要求存取資源時，發現須等較低優先權任務釋放，而中間的任務又等最低優先權任務釋放資源，如此一個等一個，環環相扣，如同鎖鏈一般，造成高優先權任務需要等每個低優先權任務解開才能夠存取。SPR 能夠解決 chain blocking 的問題，如上圖所示，由於 task 1 來的太慢或是 task 2 執行時間不夠長所以看不出有 chain blocking 的狀況，若是 task 1 緊接在 task 2 後抵達，且執行時間能夠剛好配合就有可能造成 chain blocking 情況，而這個例子，剛好不會有這個問題，而是 task 1 與 task 3 的 deadlock 問題。由 SPR 中 Ceiling 對資源的權限控管，如同 Task set 1 的分析，能夠避免掉上述的那些問題。

III. 心得

本次project主要是實作SRP的部分，練習使用相關工具來達到對資源共享的保護。在程式開發過程中，一定有使用過全域變數，因此就能了解到對與有使用到此變數的程式部分，必須保持該變數的完整及正確性，否則會造成錯誤的運算影響後續的程式部分，甚至會導致系統崩潰，而全域變數就是共享資源的一種例子，因此對於資源的控管是相當重要的。而課堂中介紹了幾種相關的操作函式，為實作出SRP我們須使用Mutex來達到對資源的保護。Mutex本身就有了優先權繼承的概念，因此在這部分不用去修改也少了很多麻煩，只需針對Ceiling及Deadline的部份去做控制，並且在之前改的EDF Scheduler下，加入Ceiling與Preemption Level的比較來控管任務的優先順序。整體來說，有了之前的基礎，這次的Project難度反而是最簡單的，過程中只有遇到了小小問題，系統在運行當中會被Idle任務中斷，後來發現是Scheduler沒加入Ceiling的判斷以及相關的初始。透過程式的撰寫以及實際SRP，對於課堂上的理論有更深刻的理解及內化，也了解到資源共享對一個作業系統來說是非常重要的，沒有好的方案來控管資源，就如同錯誤的政策比貪汙可怕，對後續的發展有嚴重的影響。