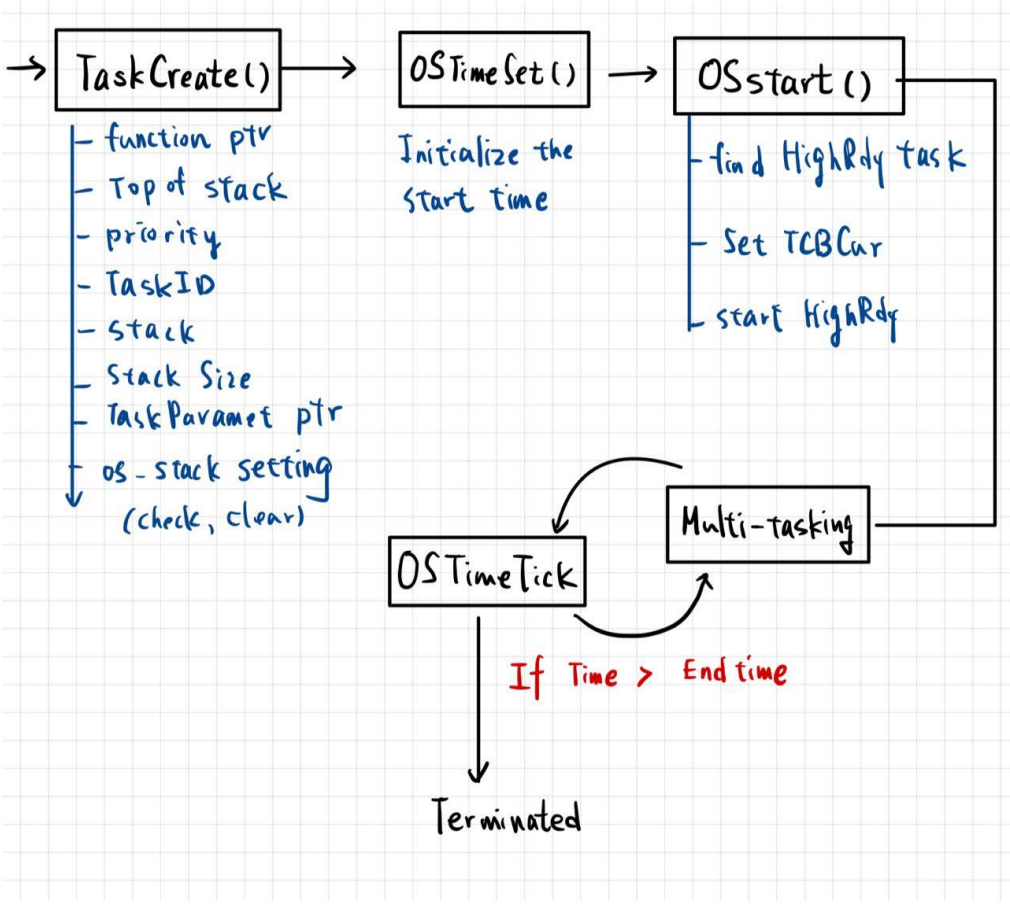
(a.)



- **TaskCreateExe():**
  Create a task, and set parameters of task. The task stack, the priority, TaskID, and any task-specific parameters. The OS also sets up the task's stack (cleaning or checking it) so that it can be safely used by the new task.
- **OSTimeSet():**
  Initializes the system start time (often by setting the tick counter to 0).
- **OSStart()**
  Finds the highest-priority ready task and sets up its Task Control Block (TCB). Begins multitasking by starting the scheduler, which will give the CPU to the highest-priority task that is ready to run.
- **OSTimeTick()**
  A periodic interrupt (tick) that updates the system's internal time and checks if any tasks' delays have expired. If a task's wait time is over, it becomes ready, and the scheduler may switch to that task if its priority is higher than the current one. Finally, if system time exceeds a defined end time, the system will be terminated.

```
if (OSRunning == OS_TRUE) {
    /*setting the end time for the os*/
    if (OSTimeGet() > SYSTEM_END_TIME) {
        OSRunning = OS_FALSE;
        exit(0);
    }
    /*Setting the end time for the OS*/
```

- Task1() and Task2():
  Both all these two tasks will use OSTimeDly to wait a certain time continuously.

```
void task1(void* p_arg) {
    task_para_set* task_data;
    task_data = p_arg;
    while (1)
    {
        OSTimeDly(task_data->TaskPeriodic);
    }
}
```

```
void task2(void* p_arg) {
    task_para_set* task_data;
    task_data = p_arg;
    while (1)
    {
        OSTimeDly(task_data->TaskPeriodic);
    }
}
```

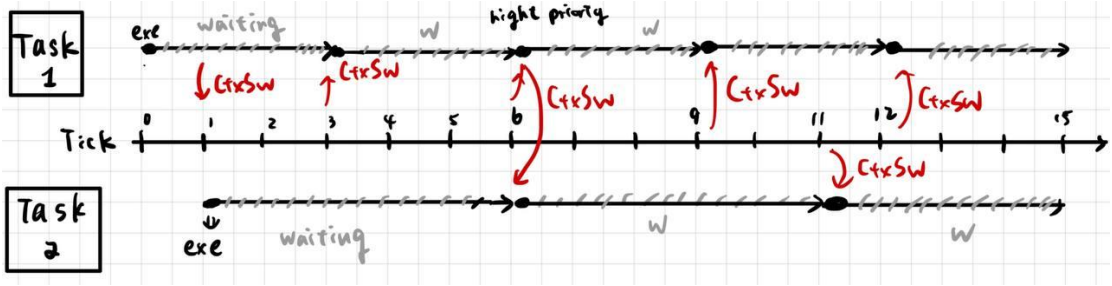
- **OSTimeDly()** will set this task to waiting state, and waiting about its periodic tick time. It will make sure it call by task, update the state of task and clean the ready table.

```
void  OSTimeDly (INT32U ticks)
{
    INT8U      y;
#if OS_CRITICAL_METHOD == 3u                    /* Allocate storage for CPU status register           */
    OS_CPU_SR  cpu_sr = 0u;
#endif



    if (OSIntNesting > 0u) {                    /* See if trying to call from an ISR                  */
        return;
    }
    if (OSLockNesting > 0u) {                   /* See if called with scheduler locked                */
        return;
    }
    if (ticks > 0u) {                           /* 0 means no delay!                                  */
        OS_ENTER_CRITICAL();
        y             =  OSTCBCur->OSTCBY;      /* Delay current task                                 */
        OSRdyTbl[y] &= (OS_PRIO)~OSTCBCur->OSTCBBitX;
        OS_TRACE_TASK_SUSPENDED(OSTCBCur);
        if (OSRdyTbl[y] == 0u) {
            OSRdyGrp &= (OS_PRIO)~OSTCBCur->OSTCBBitY;
        }
        OSTCBCur->OSTCBDly = ticks;             /* Load ticks in TCB                                  */
        OS_TRACE_TASK_DLY(ticks);
        OS_EXIT_CRITICAL();
        OS_Sched();                             /* Find next task to run!                             */
    }
}
```

- **Context Switch flow:**



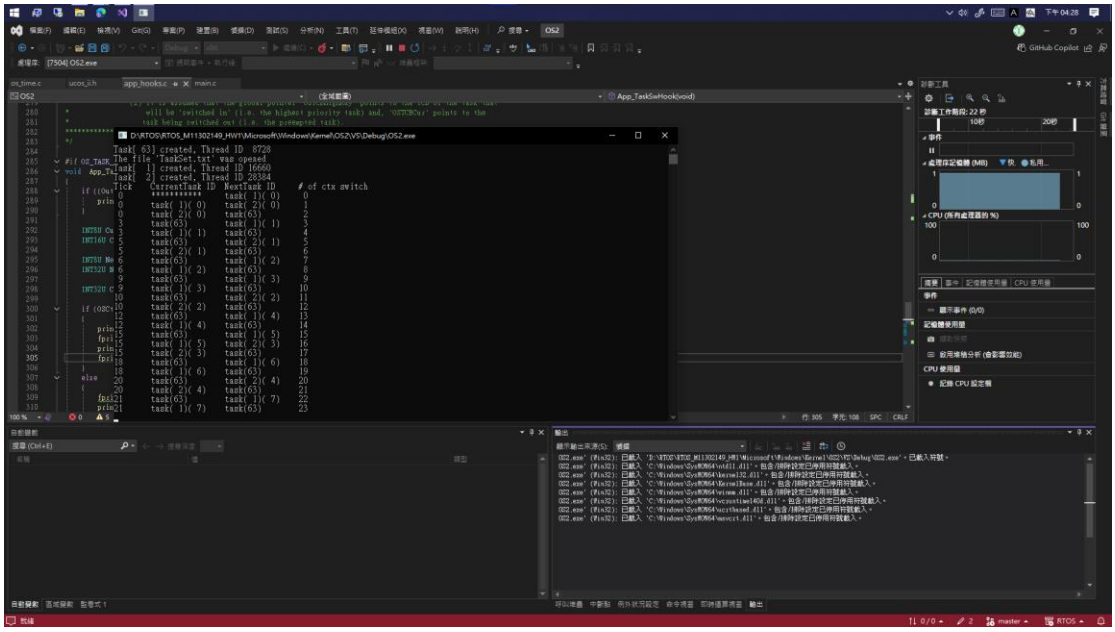
In this scenario, Task 1 has the highest priority (priority 1) and a period of 3 ticks, while Task 2 has a lower priority (priority 2) and a period of 5 ticks. Because each task's actual execution time is much shorter than a single tick, they quickly complete

their work and then wait for the remainder of their period (OSTImeDly). After delay, it will be ready again.

I’ve illustrated 15 ticks of execution flow to show how context switches occur. Whenever neither Task 1 nor Task 2 is ready, the system automatically runs the idle task (priority 63). Note that at tick 6, both Task 1 and Task 2 become ready simultaneously; however, Task 1 has a higher priority and therefore runs first.

The diagram above demonstrates how task-level context switching is managed under these conditions in this lab.

(b.)



Output.txt - 記事本

檔案(F) 編輯(E) 格式(O) 檢視(V) 說明

```
Tick    CurrentTask ID  NextTask ID     # of ctx switch
0       ***********     task( 1)( 0)     0
0       task( 1)( 0)    task( 2)( 0)     1
0       task( 2)( 0)    task(63)         2
3       task(63)        task( 1)( 1)     3
3       task( 1)( 1)    task(63)         4
5       task(63)        task( 2)( 1)     5
5       task( 2)( 1)    task(63)         6
6       task(63)        task( 1)( 2)     7
6       task( 1)( 2)    task(63)         8
9       task(63)        task( 1)( 3)     9
9       task( 1)( 3)    task(63)        10
10      task(63)        task( 2)( 2)    11
10      task( 2)( 2)    task(63)        12
12      task(63)        task( 1)( 4)    13
12      task( 1)( 4)    task(63)        14
15      task(63)        task( 1)( 5)    15
15      task( 1)( 5)    task( 2)( 3)    16
15      task( 2)( 3)    task(63)        17
18      task(63)        task( 1)( 6)    18
18      task( 1)( 6)    task(63)        19
20      task(63)        task( 2)( 4)    20
20      task( 2)( 4)    task(63)        21
21      task(63)        task( 1)( 7)    22
21      task( 1)( 7)    task(63)        23
24      task(63)        task( 1)( 8)    24
24      task( 1)( 8)    task(63)        25
25      task(63)        task( 2)( 5)    26
25      task( 2)( 5)    task(63)        27
27      task(63)        task( 1)( 9)    28
27      task( 1)( 9)    task(63)        29
30      task(63)        task( 1)(10)    30
30      task( 1)(10)    task( 2)( 6)    31
30      task( 2)( 6)    task(63)        32
```

(c.)

Firstly, I modified the structure “task_para_set”, adding a unsigned integer variable ExCounter (Executed counter) to record the times task be run.

```
/*Task structure*/
typedef struct task_para_set {
    INT16U TaskID;
    INT16U TaskArriveTime;
    INT16U TaskExecutionTime;
    INT16U TaskPeriodic;
    INT16U TaskNumber;
    INT16U TaskPriority;
    INT16U ExCounter;
}task_para_set;
int TASK_NUMBER;
/*Task structure*/
```

The TaskSet :

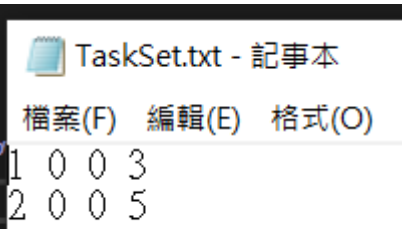
	ID / Priority	ArriveTime	Exe Time	Period
Task1	1	0	0	3
Task2	2	0	0	5

It reads the TaskSet.txt, set on the parameter of task and then initialize the executed counter to 0.

```
while (ptr != NULL)
{
    TaskInfo[i] = atoi(ptr);
    ptr = strtok_s(NULL, " ", &pTmp);
    /*printf("Info: %d\n", task_inf[i]);*/
    if (i == 0) {
        TASK_NUMBER++;
        TaskParameter[j].TaskID = TaskInfo[i];
        TaskParameter[j].TaskPriority = 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];

    i++;
}
TaskParameter[j].ExCounter = 0u;

j++;
}
fclose(fp);
```



After set on the parameter, I modify the App_TaskSwHook() in *app_hook.c.* This function is called when a task switch is performed. It open output.txt, and print the ticktime, taskID, job number and the number of context switch.

```
void  App_TaskSwHook (void)
{
    if ((Output_err = fopen_s(&Output_fp, "./Output.txt", "a")) != 0) {
        printf("Error open Output.txt!\n");
    }

    INT8U CurrentTaskID = TaskParameter[OSPrioCur-1].TaskID;
    INT16U CurrentTaskCtr = TaskParameter[OSPrioCur - 1].ExCounter;

    INT8U NextTaskID = TaskParameter[OSPrioHighRdy-1].TaskID;
    INT32U NextTaskCtr = TaskParameter[OSPrioHighRdy - 1].ExCounter;

    INT32U CurrentTick = OSTimeGet();

    if (OSCtxSwCtr == 0)
    {
        printf("Tick\tCurrentTask ID\tNextTask ID\t# of ctx switch\n");
        fprintf(Output_fp, "Tick\tCurrentTask ID\tNextTask ID\t# of ctx switch\n");
        printf("%2d\t**********\ttask(%2d)(%2d)\t 0\n", CurrentTick ,NextTaskID, NextTaskCtr);
        fprintf(Output_fp, "%2d\t**********\ttask(%2d)(%2d)\t 0\n", CurrentTick, NextTaskID, NextTaskCtr);
    }
```

```
else
{
    fprintf(Output_fp, "%2d\t", OSTimeGet());
    printf("%2d\t", CurrentTick);
    if (OSPrioCur == OS_TASK_IDLE_PRIO)  // if idle
    {
        printf("task(%2d)\t", OS_TASK_IDLE_PRIO);
        fprintf(Output_fp, "task(%d)\t", OS_TASK_IDLE_PRIO);
    }
    else //current
    {
        printf("task(%2d)(%2d)\t", CurrentTaskID, CurrentTaskCtr);
        fprintf(Output_fp, "task(%2d)(%2d)\t", CurrentTaskID, CurrentTaskCtr);
        TaskParameter[OSPrioCur - 1].ExCounter++;
    }
    if (OSPrioHighRdy == OS_TASK_IDLE_PRIO)  // if idle
    {
        printf("task(%2d)\t", OS_TASK_IDLE_PRIO);
        fprintf(Output_fp, "task(%2d)\t", OS_TASK_IDLE_PRIO);
    }
    else  // next
    {
        printf("task(%2d)(%2d)\t", NextTaskID, NextTaskCtr);
        fprintf(Output_fp, "task(%2d)(%2d)\t", NextTaskID, NextTaskCtr);
    }
    printf("%2d\n", OSCtxSwCtr);  // CtxSw count
    fprintf(Output_fp, "%2d\n", OSCtxSwCtr);
}


fclose(Output_fp);
```

If the task-level context switch is not performed first time and it is not idle task, it increases the executed time counter with the current task.