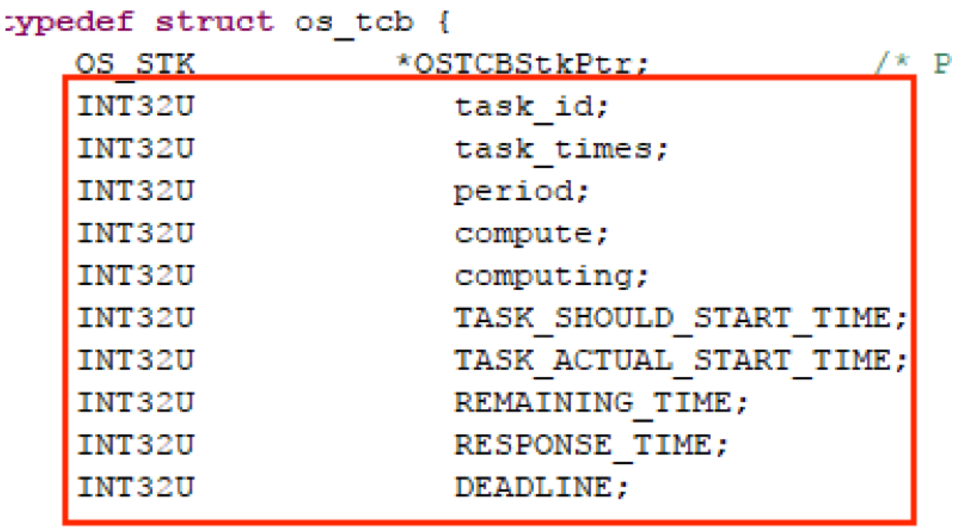
B10332024 四電資四 江承祐

1. Implementation Description:

The main objective of this project is to simulate tasks that run for both within and for certain period (c, p).

(1) Information

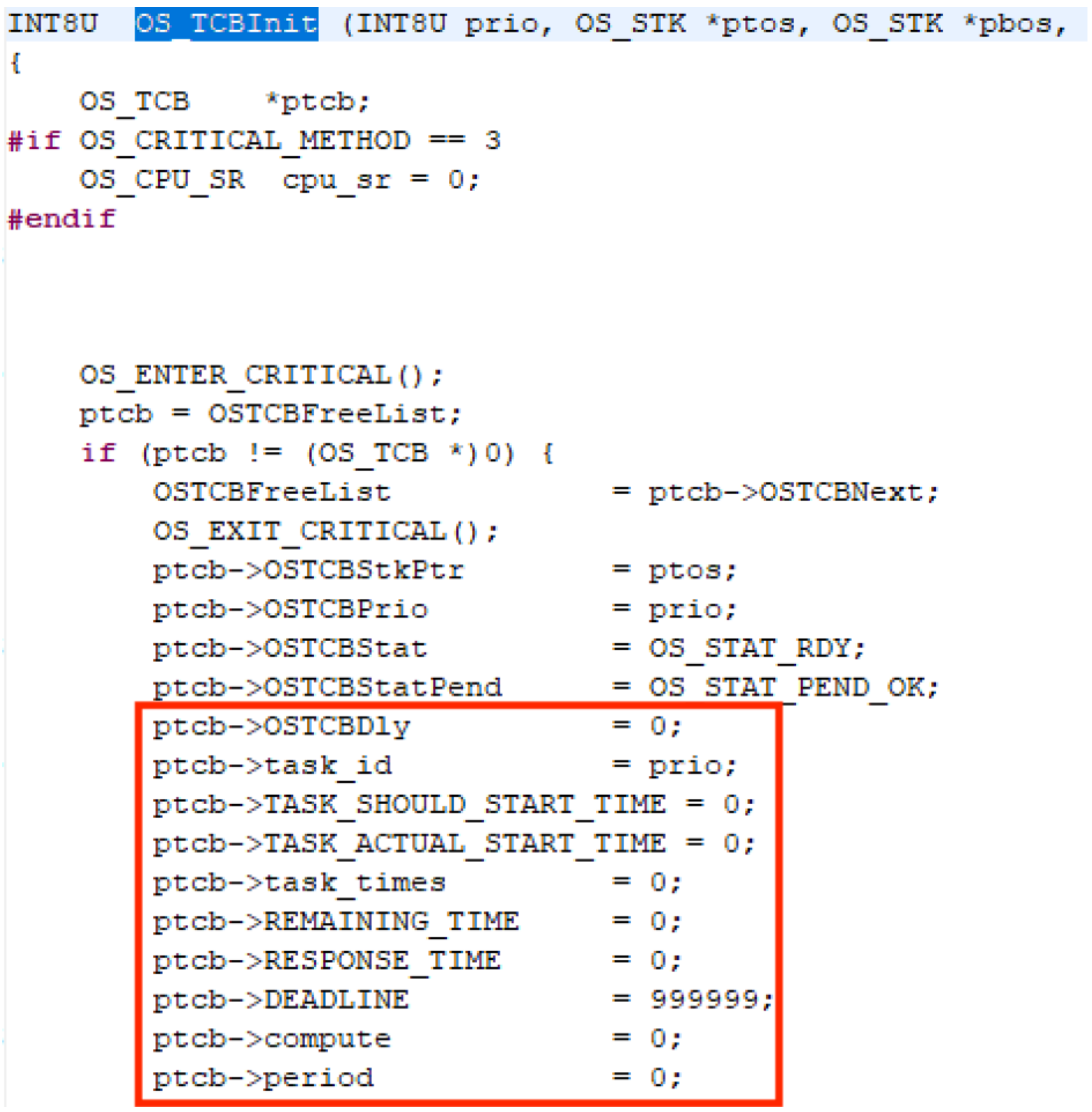
First, we set up the information we need for the Task simulation. We need to record the remaining time, response time, and actual start time, and the supposed starting time. We set this up in the structure OS_TCB.



```
typedef struct os_tcb {
    OS_STK          *OSTCBStkPtr;          /* P
    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              DEADLINE;
```

(1) Task_id is the ID number of the task
(2) Task_times is the number of time the task has computed
(3) Period is the period time of the task
(4) Compute is the computing time of the task
(5) Computing is a flag to determine if the task if currently computing or not
(6) TASK_SHOULD_START_TIME is the start of each period of the task
(7) TASK_ACTUAL_START_TIME is the time the task actually starts computing
(8) REMAINING_TIME is the remaining computation time the task has to compute
(9) RESPONSE_TIME is the difference between the time the task finishes computing and TASK_ACTUAL_START_TIME
(10) DEADLINE is the task's deadline time

After setting up, the task information should be initialized in OSTCBInit.



```
INT8U   OS_TCBInit (INT8U prio, OS_STK *ptos, OS_STK *pbos,
{
    OS_TCB    *ptcb;
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR  cpu_sr = 0;
#endif



    OS_ENTER_CRITICAL();
    ptcb = OSTCBFreeList;
    if (ptcb != (OS_TCB *)0) {
        OSTCBFreeList              = ptcb->OSTCBNext;
        OS_EXIT_CRITICAL();
        ptcb->OSTCBStkPtr          = ptos;
        ptcb->OSTCBPrio            = prio;
        ptcb->OSTCBStat            = OS_STAT_RDY;
        ptcb->OSTCBStatPend        = OS_STAT_PEND_OK;
        ptcb->OSTCBDly             = 0;
        ptcb->task_id              = prio;
        ptcb->TASK_SHOULD_START_TIME = 0;
        ptcb->TASK_ACTUAL_START_TIME = 0;
        ptcb->task_times           = 0;
        ptcb->REMAINING_TIME       = 0;
        ptcb->RESPONSE_TIME        = 0;
        ptcb->DEADLINE             = 999999;
        ptcb->compute              = 0;
        ptcb->period               = 0;
```

(2) Task Simulation

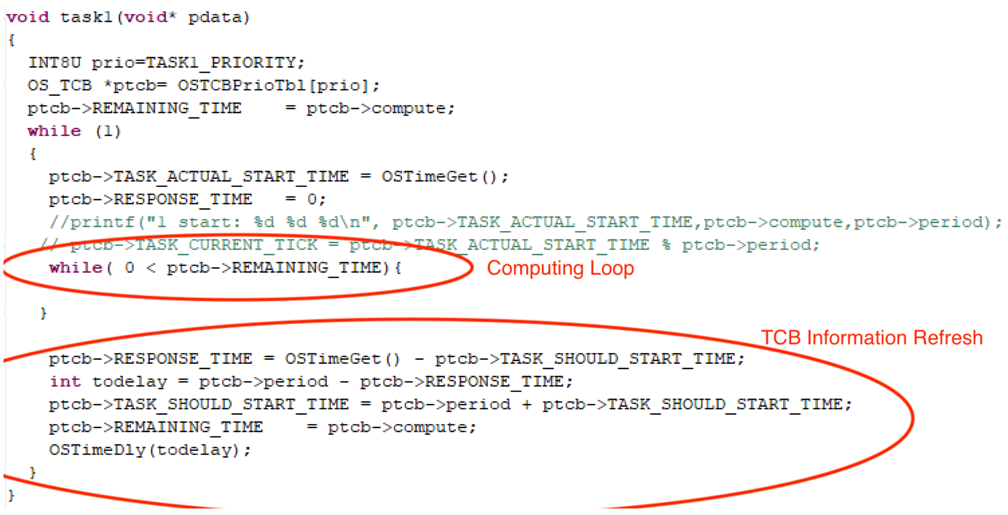
The main part where I simulated a task period was within the application layer.

First, we assign the correct TCB used for this task from the OSTCBPrioTbl.

We assign the remaining time of our task before truly starting.

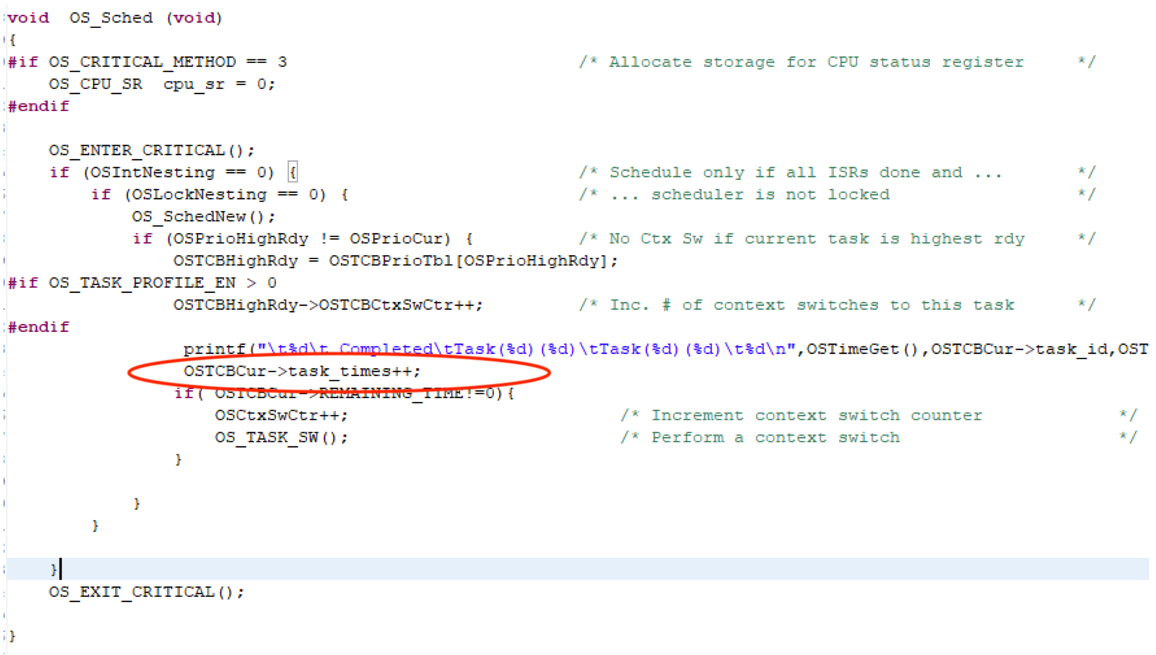
"While (0 < ptcb->REMAINING_TIME)" is the time spent on computing. After computing, we retrieve the current time and calculate the response time of the task.

We also reassign the information needed for the next period of this task. The task is then delayed until the start of the next period.



```
void task1(void* pdata)
{
  INT8U prio=TASK1_PRIORITY;
  OS_TCB *ptcb= OSTCBPrioTbl[prio];
  ptcb->REMAINING_TIME     = ptcb->compute;
  while (1)
  {
    ptcb->TASK_ACTUAL_START_TIME = OSTimeGet();
    ptcb->RESPONSE_TIME    = 0;
    //printf("1 start: %d %d %d\n", ptcb->TASK_ACTUAL_START_TIME,ptcb->compute,ptcb->period);
   // ptcb->TASK_CURRENT_TICK = ptcb->TASK_ACTUAL_START_TIME % ptcb->period;
    while( 0 < ptcb->REMAINING_TIME){

   }

    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->REMAINING_TIME     = ptcb->compute;
    OSTimeDly(todelay);
  }
}
```

Meanwhile, the task counter is updated after each period is done with its computation (within OSSched()). If the task counter is updated too early, the information of our task will be updated too fast for our output and will cause errors.

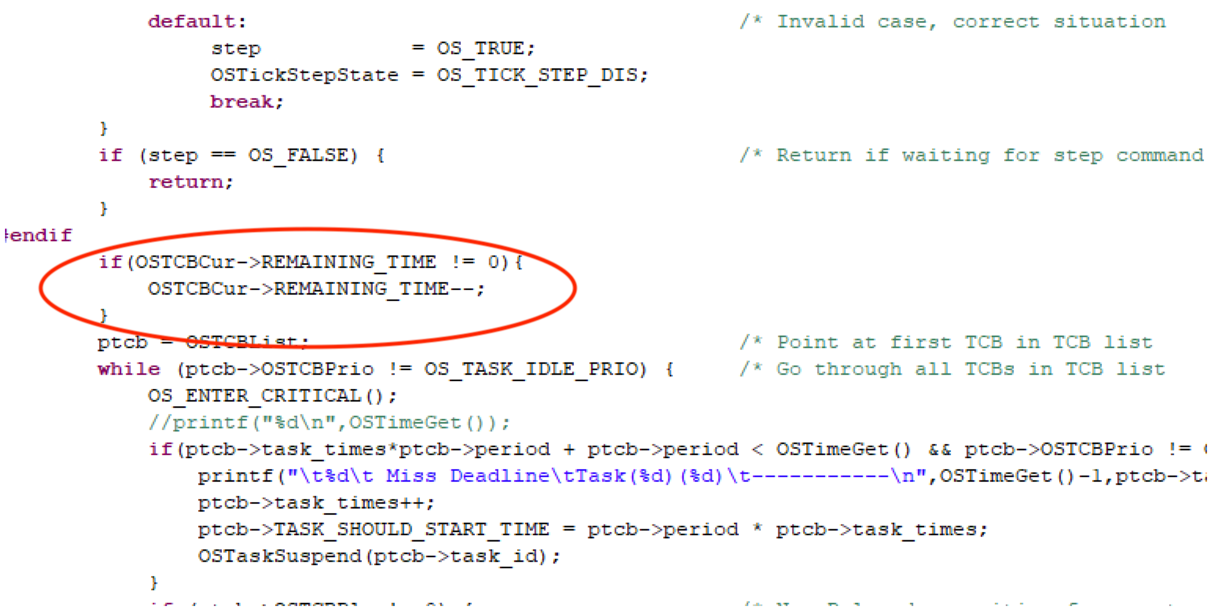


```
void  OS_Sched (void)
{
#if OS_CRITICAL_METHOD == 3                          /* Allocate storage for CPU status register     */
    OS_CPU_SR  cpu_sr = 0;
#endif

    OS_ENTER_CRITICAL();
    if (OSIntNesting == 0) {                         /* Schedule only if all ISRs done and ...       */
        if (OSLockNesting == 0) {                    /* ... scheduler is not locked                  */
            OS_SchedNew();
            if (OSPrioHighRdy != OSPrioCur) {        /* No Ctx Sw if current task is highest rdy     */
                OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
#if OS_TASK_PROFILE_EN > 0
                OSTCBHighRdy->OSTCBCtxSwCtr++;       /* Inc. # of context switches to this task      */
#endif
                 printf("\t%d\t Completed\tTask(%d)(%d)\tTask(%d)(%d)\t%d\n",OSTimeGet(),OSTCBCur->task_id,OST
                 OSTCBCur->task_times++;
                if( OSTCBCur->REMAINING_TIME!=0){
                    OSCtxSwCtr++;                        /* Increment context switch counter             */
                    OS_TASK_SW();                        /* Perform a context switch                     */
                }

            }
        }

    }
    OS_EXIT_CRITICAL();

}
```

(3) Task Computation and Deadline Misses

(a) Task Computation

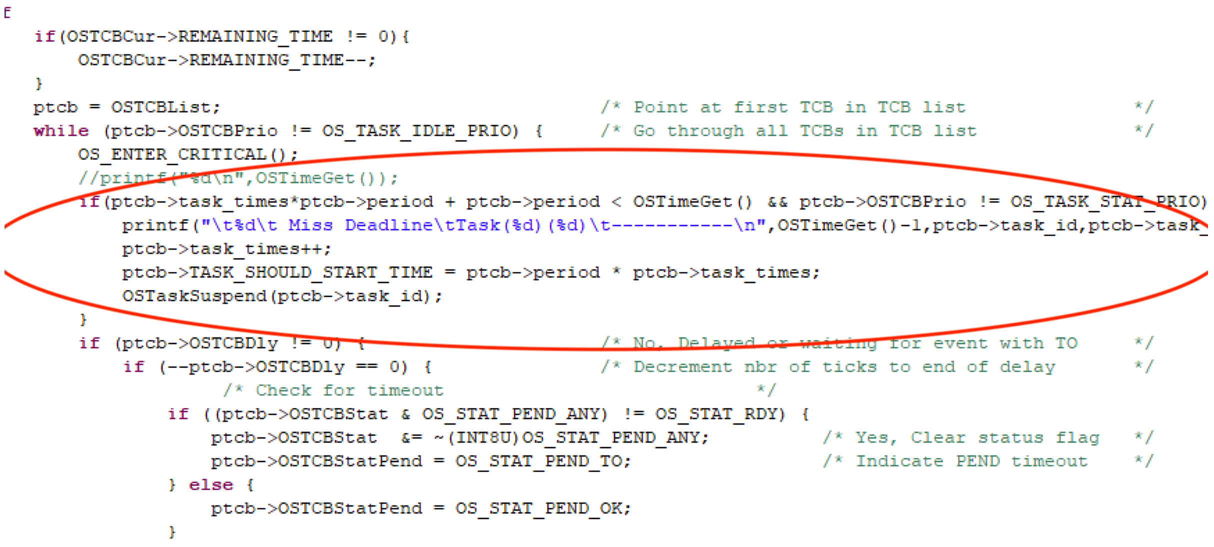
OSTimeTick is a task within the operating system that runs once per second. Here, we do the actual computing of the current task (OSTCBCur). If a task is currently running, the remaining time of that task lowers by one.



```
            default:                                      /* Invalid case, correct situation
                 step             = OS_TRUE;
                 OSTickStepState = OS_TICK_STEP_DIS;
                 break;
        }
        if (step == OS_FALSE) {                           /* Return if waiting for step command
            return;
        }
#endif
        if(OSTCBCur->REMAINING_TIME != 0){
            OSTCBCur->REMAINING_TIME--;
        }
        ptcb = OSTCBList;                                 /* Point at first TCB in TCB list
        while (ptcb->OSTCBPrio != OS_TASK_IDLE_PRIO) {    /* Go through all TCBs in TCB list
            OS_ENTER_CRITICAL();
            //printf("%d\n",OSTimeGet());
            if(ptcb->task_times*ptcb->period + ptcb->period < OSTimeGet() && ptcb->OSTCBPrio != 
                printf("\t%d\t Miss Deadline\tTask(%d)(%d)\t-----------\n",OSTimeGet()-1,ptcb->t
                ptcb->task_times++;
                ptcb->TASK_SHOULD_START_TIME = ptcb->period * ptcb->task_times;
                OSTaskSuspend(ptcb->task_id);
            }
```

(b) Deadline Misses

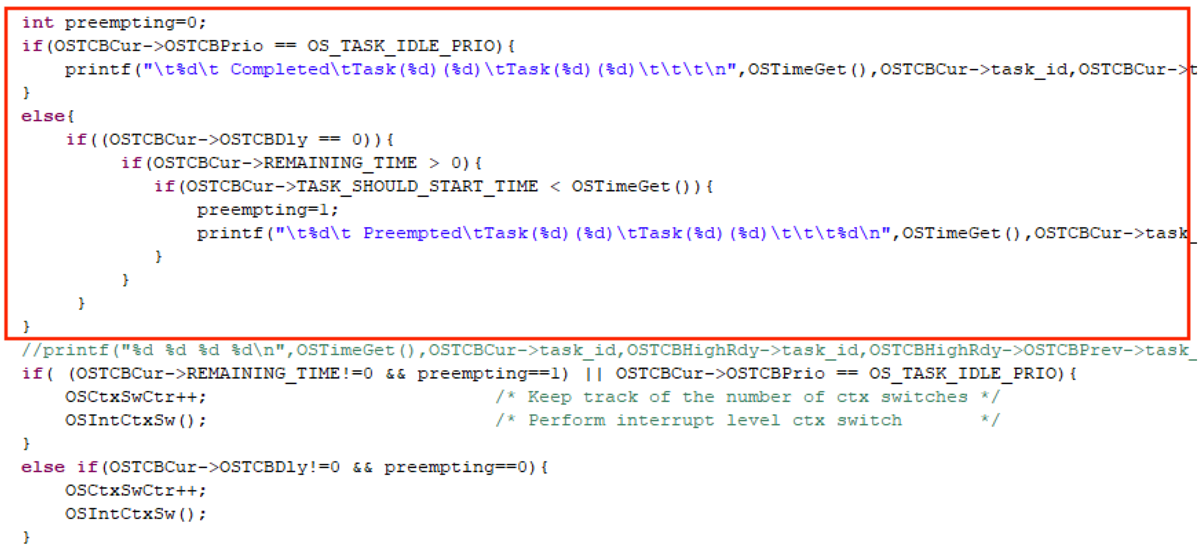
We also constantly check for **Deadline Misses** here. We calculate the supposed ending point of this period and if the current time exceeds that, it means the deadline has been missed. If the deadline is missed, we output the results and suspend the task for the rest of the program.



```
    if(OSTCBCur->REMAINING_TIME != 0){
        OSTCBCur->REMAINING_TIME--;
    }
    ptcb = OSTCBList;                                 /* Point at first TCB in TCB list         */
    while (ptcb->OSTCBPrio != OS_TASK_IDLE_PRIO) {    /* Go through all TCBs in TCB list        */
        OS_ENTER_CRITICAL();
        //printf("%d\n",OSTimeGet());
        if(ptcb->task_times*ptcb->period + ptcb->period < OSTimeGet() && ptcb->OSTCBPrio != OS_TASK_STAT_PRIO)
            printf("\t%d\t Miss Deadline\tTask(%d)(%d)\t-----------\n",OSTimeGet()-1,ptcb->task_id,ptcb->task_
            ptcb->task_times++;
            ptcb->TASK_SHOULD_START_TIME = ptcb->period * ptcb->task_times;
            OSTaskSuspend(ptcb->task_id);
        }
        if (ptcb->OSTCBDly != 0) {                    /* No, Delayed or waiting for event with TO */
            if (--ptcb->OSTCBDly == 0) {              /* Decrement nbr of ticks to end of delay  */
                    /* Check for timeout                             */
                if ((ptcb->OSTCBStat & OS_STAT_PEND_ANY) != OS_STAT_RDY) {
                    ptcb->OSTCBStat  &= ~(INT8U)OS_STAT_PEND_ANY;          /* Yes, Clear status flag  */
                    ptcb->OSTCBStatPend = OS_STAT_PEND_TO;                 /* Indicate PEND timeout   */
                } else {
                    ptcb->OSTCBStatPend = OS_STAT_PEND_OK;
                }
```

(4) Task Preemption

We check for preemption in the ISR Context Switcher, OSIntExit. Normally if a task is currently computing and suddenly goes through the ISR Context switcher, the task is getting ready for preemption to compute a higher priority task. We check if the task is currently running by checking if the REMAINING_TIME is larger than 0. If the task is running, it means it is being preempted and we output the preemption results into the console.

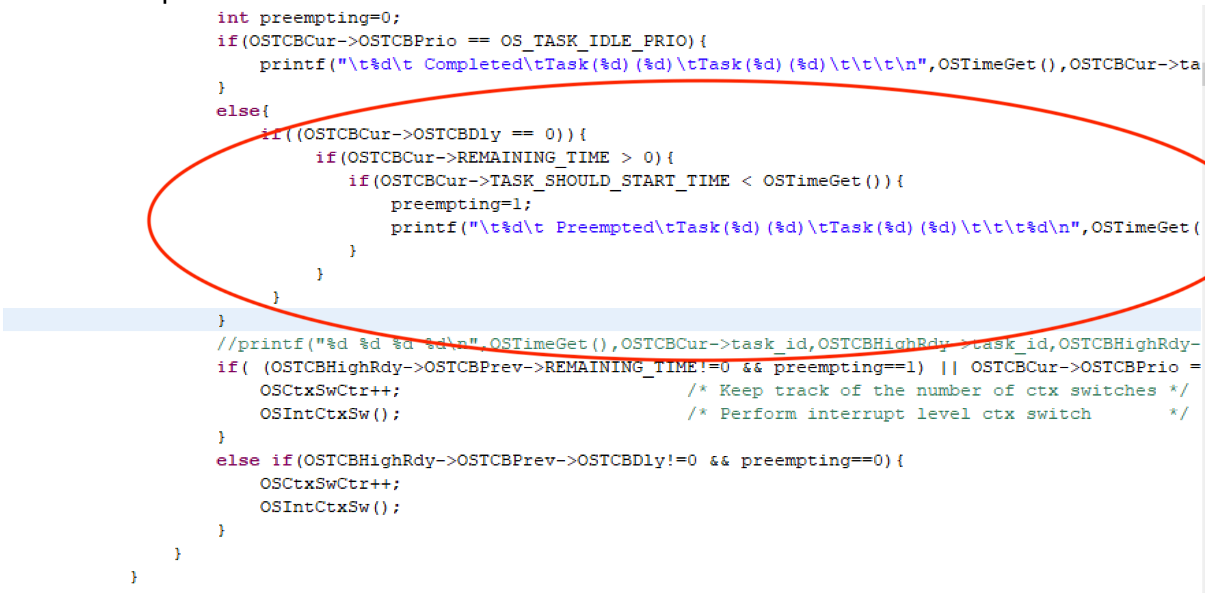


```
int preempting=0;
if(OSTCBCur->OSTCBPrio == OS_TASK_IDLE_PRIO){
    printf("\t%d\t Completed\tTask(%d)(%d)\tTask(%d)(%d)\t\t\t\n",OSTimeGet(),OSTCBCur->task_id,OSTCBCur->t
}
else{
    if((OSTCBCur->OSTCBDly == 0)){
         if(OSTCBCur->REMAINING_TIME > 0){
            if(OSTCBCur->TASK_SHOULD_START_TIME < OSTimeGet()){
                preempting=1;
                printf("\t%d\t Preempted\tTask(%d)(%d)\tTask(%d)(%d)\t\t\t%d\n",OSTimeGet(),OSTCBCur->task_
            }
         }
     }
}
//printf("%d %d %d %d\n",OSTimeGet(),OSTCBCur->task_id,OSTCBHighRdy->task_id,OSTCBHighRdy->OSTCBPrev->task_
if( (OSTCBCur->REMAINING_TIME!=0 && preempting==1) || OSTCBCur->OSTCBPrio == OS_TASK_IDLE_PRIO){
    OSCtxSwCtr++;                             /* Keep track of the number of ctx switches */
    OSIntCtxSw();                             /* Perform interrupt level ctx switch       */
}
else if(OSTCBCur->OSTCBDly!=0 && preempting==0){
    OSCtxSwCtr++;
    OSIntCtxSw();
}
```

(5) Task Completion

(a) Our task is completed

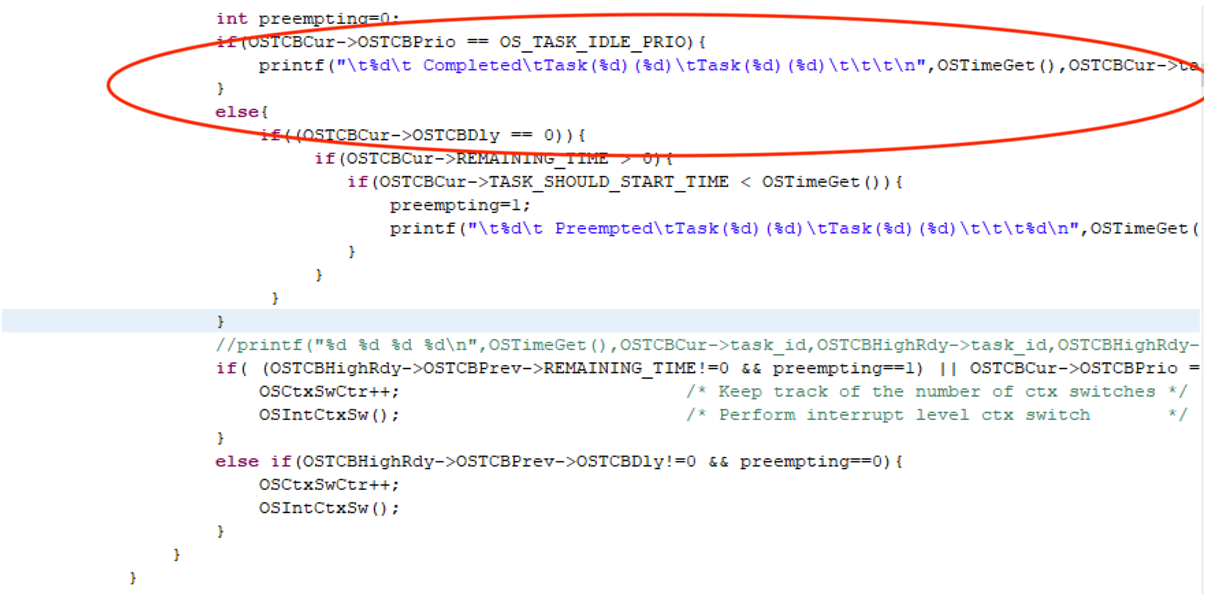
Next, we check for the completion of a task within the Task Switcher, OS_Sched().
Here we use a simple printf, as OS_Sched() is called only when the task is completed and about to be switched.



```
int preempting=0;
if(OSTCBCur->OSTCBPrio == OS_TASK_IDLE_PRIO){
    printf("\t%d\t Completed\tTask(%d)(%d)\tTask(%d)(%d)\t\t\t\n",OSTimeGet(),OSTCBCur->ta
}
else{
    if((OSTCBCur->OSTCBDly == 0)){
         if(OSTCBCur->REMAINING_TIME > 0){
            if(OSTCBCur->TASK_SHOULD_START_TIME < OSTimeGet()){
                preempting=1;
                printf("\t%d\t Preempted\tTask(%d)(%d)\tTask(%d)(%d)\t\t\t%d\n",OSTimeGet(
            }
         }
     }
}
//printf("%d %d %d %d\n",OSTimeGet(),OSTCBCur->task_id,OSTCBHighRdy->task_id,OSTCBHighRdy-
if( (OSTCBHighRdy->OSTCBPrev->REMAINING_TIME!=0 && preempting==1) || OSTCBCur->OSTCBPrio =
    OSCtxSwCtr++;                             /* Keep track of the number of ctx switches */
    OSIntCtxSw();                             /* Perform interrupt level ctx switch       */
}
else if(OSTCBHighRdy->OSTCBPrev->OSTCBDly!=0 && preempting==0){
    OSCtxSwCtr++;
    OSIntCtxSw();
}
        }
    }
```

(b) Idle task is completed

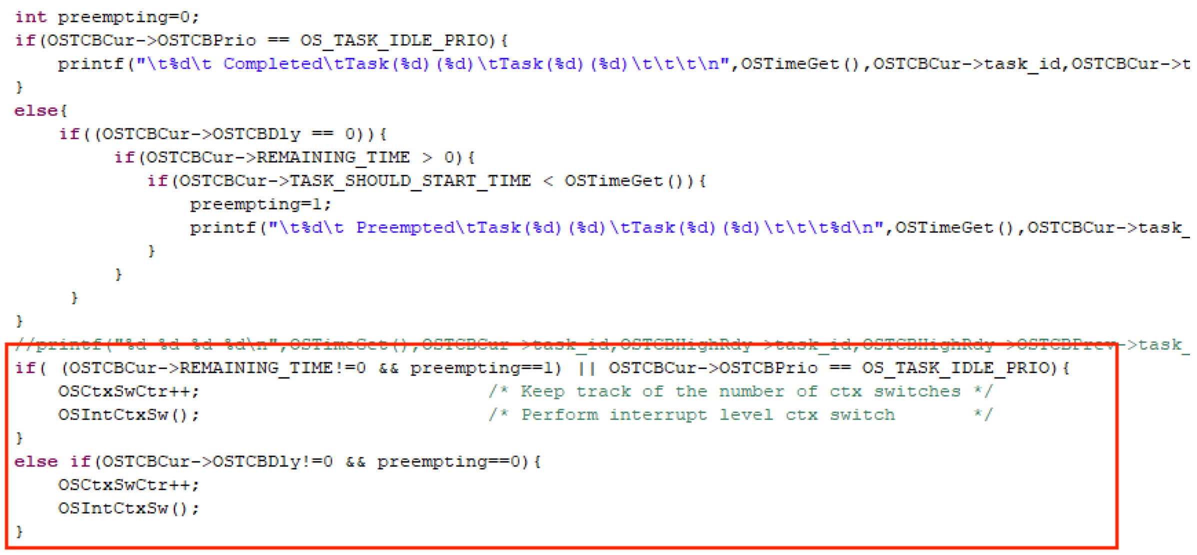
However, OS_Sched() is not called when an idle task (Priority 63) is about to end and start running a task. To fix this problem, we implement a “Completed Task” circumstance for the idle task within OSIntExit(). The reason this works is because a task goes through an interrupt (ISR) first before it starts to compute.



```c
            int preempting=0;
            if(OSTCBCur->OSTCBPrio == OS_TASK_IDLE_PRIO){
                printf("\t%d\t Completed\tTask(%d)(%d)\tTask(%d)(%d)\t\t\t\n",OSTimeGet(),OSTCBCur->ta
            }
            else{
                if((OSTCBCur->OSTCBDly == 0)){
                     if(OSTCBCur->REMAINING_TIME > 0){
                        if(OSTCBCur->TASK_SHOULD_START_TIME < OSTimeGet()){
                            preempting=1;
                            printf("\t%d\t Preempted\tTask(%d)(%d)\tTask(%d)(%d)\t\t\t%d\n",OSTimeGet(
                        }
                     }
                 }
            }
            //printf("%d %d %d %d\n",OSTimeGet(),OSTCBCur->task_id,OSTCBHighRdy->task_id,OSTCBHighRdy-
            if( (OSTCBHighRdy->OSTCBPrev->REMAINING_TIME!=0 && preempting==1) || OSTCBCur->OSTCBPrio =
                OSCtxSwCtr++;                         /* Keep track of the number of ctx switches */
                OSIntCtxSw();                         /* Perform interrupt level ctx switch       */
            }
            else if(OSTCBHighRdy->OSTCBPrev->OSTCBDly!=0 && preempting==0){
                OSCtxSwCtr++;
                OSIntCtxSw();
            }
        }
    }
```

(c) Task completion problems and fixes
Within our task simulation, our task is considered to be finished computing after the computing loop is finished. After computing is done, new information essential for the next period is inputted into the task TCB. However, if a higher priority task is about to run, a lower priority task is often preempted without the new information inputted. To fix this problem, we add an if to check if the new information has been successfully inputted into the task before the context switch in OSIntExit().

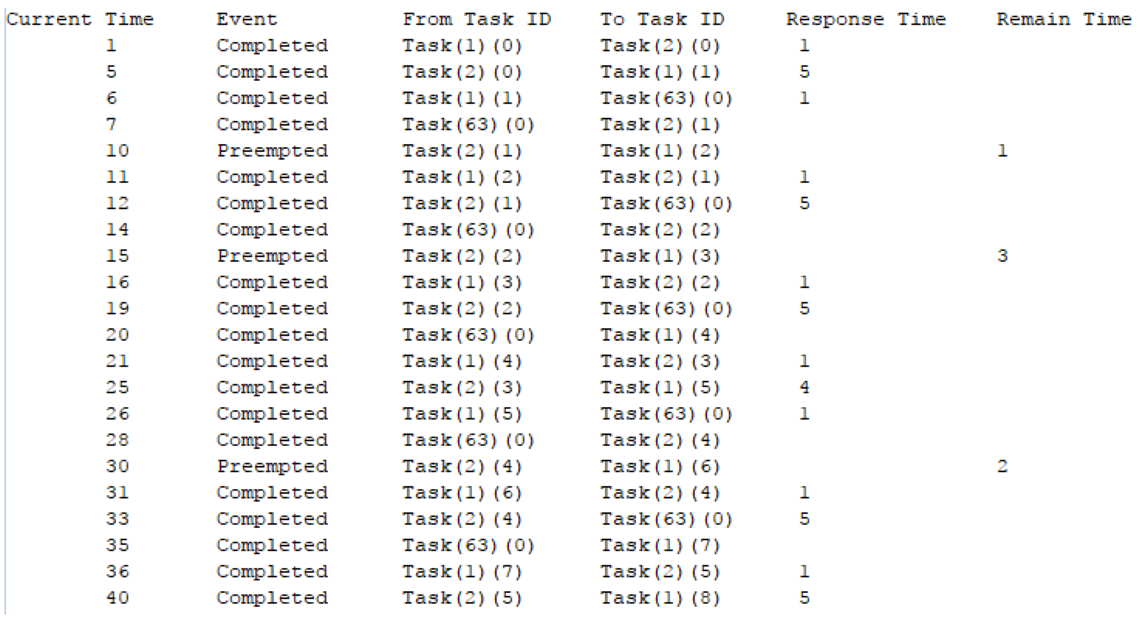
There are other times when the new information is successfully inputted, but the task is preempted before it can be delayed! To fix this, we implemented a check for preemption, and a check for the delay time. If the task is not preempting (previous task is completed), but the delay time is 0 (which it shouldn't be), we do not the task perform a context switch.



```c
int preempting=0;
if(OSTCBCur->OSTCBPrio == OS_TASK_IDLE_PRIO){
    printf("\t%d\t Completed\tTask(%d)(%d)\tTask(%d)(%d)\t\t\t\n",OSTimeGet(),OSTCBCur->task_id,OSTCBCur->t
}
else{
    if((OSTCBCur->OSTCBDly == 0)){
         if(OSTCBCur->REMAINING_TIME > 0){
            if(OSTCBCur->TASK_SHOULD_START_TIME < OSTimeGet()){
                preempting=1;
                printf("\t%d\t Preempted\tTask(%d)(%d)\tTask(%d)(%d)\t\t\t%d\n",OSTimeGet(),OSTCBCur->task_
            }
         }
     }
}
//printf("%d %d %d %d\n",OSTimeGet(),OSTCBCur->task_id,OSTCBHighRdy->task_id,OSTCBHighRdy->OSTCBPrev->task_
if( (OSTCBCur->REMAINING_TIME!=0 && preempting==1) || OSTCBCur->OSTCBPrio == OS_TASK_IDLE_PRIO){
    OSCtxSwCtr++;                         /* Keep track of the number of ctx switches */
    OSIntCtxSw();                         /* Perform interrupt level ctx switch       */
}
else if(OSTCBCur->OSTCBDly!=0 && preempting==0){
    OSCtxSwCtr++;
    OSIntCtxSw();
}
```

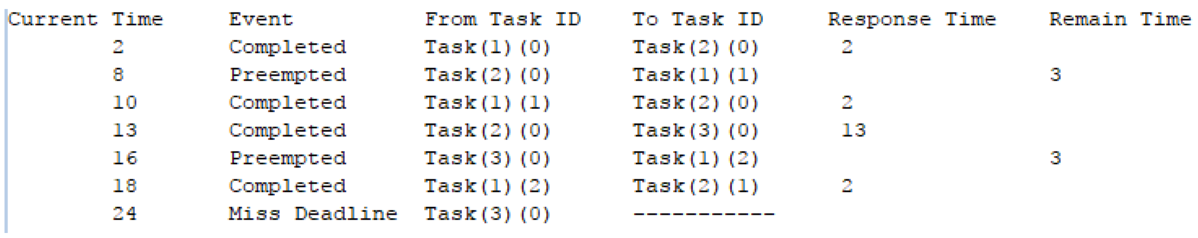
2. Simulation Result:

Task Set 1: 1(1,5) 2(4,7):



```
Current Time    Event        From Task ID   To Task ID    Response Time   Remain Time
        1       Completed    Task(1)(0)     Task(2)(0)     1
        5       Completed    Task(2)(0)     Task(1)(1)     5
        6       Completed    Task(1)(1)     Task(63)(0)    1
        7       Completed    Task(63)(0)    Task(2)(1)
        10      Preempted    Task(2)(1)     Task(1)(2)                     1
        11      Completed    Task(1)(2)     Task(2)(1)     1
        12      Completed    Task(2)(1)     Task(63)(0)    5
        14      Completed    Task(63)(0)    Task(2)(2)
        15      Preempted    Task(2)(2)     Task(1)(3)                     3
        16      Completed    Task(1)(3)     Task(2)(2)     1
        19      Completed    Task(2)(2)     Task(63)(0)    5
        20      Completed    Task(63)(0)    Task(1)(4)
        21      Completed    Task(1)(4)     Task(2)(3)     1
        25      Completed    Task(2)(3)     Task(1)(5)     4
        26      Completed    Task(1)(5)     Task(63)(0)    1
        28      Completed    Task(63)(0)    Task(2)(4)
        30      Preempted    Task(2)(4)     Task(1)(6)                     2
        31      Completed    Task(1)(6)     Task(2)(4)     1
        33      Completed    Task(2)(4)     Task(63)(0)    5
        35      Completed    Task(63)(0)    Task(1)(7)
        36      Completed    Task(1)(7)     Task(2)(5)     1
        40      Completed    Task(2)(5)     Task(1)(8)     5
```

Task Set 2: 1(2,8) 2(9,18) 3(6,24):



```
Current Time    Event          From Task ID   To Task ID     Response Time   Remain Time
        2       Completed      Task(1)(0)     Task(2)(0)      2
        8       Preempted      Task(2)(0)     Task(1)(1)                      3
        10      Completed      Task(1)(1)     Task(2)(0)      2
        13      Completed      Task(2)(0)     Task(3)(0)      13
        16      Preempted      Task(3)(0)     Task(1)(2)                      3
        18      Completed      Task(1)(2)     Task(2)(1)      2
        24      Miss Deadline  Task(3)(0)     -----------
```

Task 2 experiences a deadline at TimeTick 24.

3. Pros and Cons

Pros:

(1) The overall OS is more responsive as tasks that are deemed to be more important are processed first, resulting in a faster response time.

(2) More efficient and deadlines can be met easier if the time sets are arranged well enough.

(3) Infinite periods cannot block the system.

Cons:

(1) If the computing time and period for different tasks are not set well enough, a deadline miss could easily be invoked.

(2) If a high priority task is not delayed long enough or takes too long computing, low priority tasks could easily be led to starvation.

(3) A lot harder to implement than Non-Preemptive kernels.

4. When to use preemptive kernels?

In multitasking environments where the response time for certain resources is vital, preemptive kernels should be used. Infinite loops also rarely occur, and preemptive kernels force tasks to be switched after a certain time limit.
On the other hand, sometimes tasks cannot be halted in the middle of something important. Tasks results could be heavily effected if they are suddenly preempted by tasks using the same results. In these sort of cases, non-preemptive kernels should be used.

5. Experience:

This project mainly focuses on the concept of context switching and task simulating. Even though my understanding the process of context switching was clear, there were still many other hurdles I had to overcome. Knowing where to observe where context switching happened wasn't hard, but the process of task simulating was. Problems like knowing how to fix a task from being preempted too early hindered my progress immensely. This was the main problem within my code the first time I did this project.
Even though this project took up way more time than I expected, I learned a lot. Preemptive kernels are often seen within modern operating systems and learning how to successfully implement them will be essential to our future within coding. Luckily, our teacher has given us a chance to make up for our grades. My understanding of the code and what was to be implemented was more clear after a few hints from the instructor and our TA. I sincerely hope my newfound understanding of UC/OS can help me do my projects faster in the future.