Q16. Soit le problème du lecteur/rédacteur avec les contraintes suivantes:

- Plusieurs lecteurs peuvent lire simultanément les données;
- Les rédacteurs s'excluent mutuellement et
- Les lecteurs et les rédacteurs s'excluent mutuellement.

En considérant que **les lecteurs ont priorité sur les rédacteurs** (i.e. les rédacteurs sont exécutés lorsqu'il n'y a plus de lecteurs), réalisez en langage C sous uC (avec les mécanismes de synchronisation appropriés) les fonctions *lecteur* et *redacteur* qui appellent *read* et *write* respectivement.

N.B. - Vous n'avez pas à vous soucier de l'initialisation du programme principal mais si vous utilisez des sémaphores, dites à quelles valeurs elles sont initialisées.
- L'accès à la mémoire par le lecteur est une fonction pré-définie nommée *read* et *write* pour le rédacteur. Vous n'avez qu'à faire les appels.

Q24

Soit la tâche de priorité 11 en uC/OS-III.

Supposons que la tâche 11 fait un appel à OSTimeDly(3). La variable OSTCBDly du TCB correspondant sera alors mise à 3 et la tâche sera suspendue. En principe sans *jitter*, au bout de 3 ticks cette même variable OSTCBDly sera mise à 0 via OSTimeTick et la tâche redeviendra prête. Sachant que seules les tâches 5, 12, 14 et 45 sont prêtes au moment de mettre OSTCBDly à 0:

a) Démonterez clairement les différentes étapes dans OSTimeTick qui rendront la tâche 11 prête à être exécutée.

b) Toujours dans OSTimeTick, et plus particulièrement lors de l'appel à OSIntExit, démontrez les différentes étapes qui feront que 5 sera la prochaine tâche à être exécutée.

Q25 Un système de contrôle temps réel est composé des éléments suivants : un convertisseur A/D, un CPU et un écran d'affichage. Le convertisseur A/D génère une valeur de 8 bits toutes les 20 µs et en avertit le CPU par l'intermédiaire d'une interruption. Une ISR s'exécute alors en 10µs et transfert la donnée dans un buffer de calcul. L'acquisition se fait de manière discontinue : durant 5 ms le A/D génère des points comme décrit ci-dessus et durant 5 ms il ne fait rien, puis il recommence etc. Sur le CPU une tâche de calcul traite les valeurs une par une : elle en prend une dans le buffer de calcul, effectue un calcul avec et la place dans le buffer d'affichage. Cela lui prend 25 µs par valeur. Enfin une tâche d'affichage de période 30 ms et qui s'exécute en 1 ms, vide au complet le *buffer* d'affichage pour le mettre à l'écran.

a) En considérant que le système fonctionne sous µC, quel choix d'affection de priorité feriez-vous?
b) Calculer la taille minimum des buffers de calcul et d'affichage toute en assurant le bon fonctionnement du système. Quelle taille choisiriez-vous?
c) Calculer le pourcentage d'utilisation du CPU pour 30 ms.
d) Faites le diagramme d'ordonnancement du système pour 30 ms. (vous pouvez couper les zones répétitives)

Q26 Synchronisation entre les tâches

Soit les 6 fonctions suivantes funcU(), funcV(), funcW(), funcX(), funcY(), funcZ() devant toujours être exécutées dans ce même ordre (U, V, W, X, Y, Z et ainsi de suite) et 3 thread T1(), T2() et T3(). Le thread T1() appelle les fonctions funcU() et funcX(), le thread T2() les fonctions funcV() et funcY() et le thread T3() les fonction funcW() et funcZ(). Écrivez le code des thread T1(), T2() et T3() en veillant à ce que l'ordre d'appel des fonction (U, V, W, X, Y, Z) soit respecté.

Q28

Soit le code uC/OS-III suivant sur 4 pages. Faire la trace d'exécution de ce code.

```
#include  <cpu.h>
#include  <lib_mem.h>
#include  <os.h>

#include  "os_app_hooks.h"
#include  "app_cfg.h"

/*
*********************************************************************************************************
*                                                CONSTANTS
*********************************************************************************************************
*/

#define                 TASK_STK_SIZE    8192                // Size of each task's stacks (# of WORDs)
#define                 N                 256


#define          TASK_CONS1_PRIO      10 // Application tasks priorities
#define          TASK_CONS2_PRIO      11
#define          TASK_CONS3_PRIO      12
#define          TASK_CONS4_PRIO      13
#define          TASK_CONS5_PRIO      14

#define          TASK_PROD_PRIO       20


/*
*********************************************************************************************************
*                                                VARIABLES
*********************************************************************************************************
*/

CPU_STK           TaskCons1Stk[TASK_STK_SIZE];
CPU_STK           TaskCons2Stk[TASK_STK_SIZE];
CPU_STK           TaskCons3Stk[TASK_STK_SIZE];
CPU_STK           TaskCons4Stk[TASK_STK_SIZE];
CPU_STK           TaskCons5Stk[TASK_STK_SIZE];

CPU_STK           TaskProdStk[TASK_STK_SIZE];

OS_TCB            TaskCons1TCB;
OS_TCB            TaskCons2TCB;
OS_TCB            TaskCons3TCB;
OS_TCB            TaskCons4TCB;
OS_TCB            TaskCons5TCB;

OS_TCB            TaskProdTCB;

OS_MUTEX Mutex, Mutex_done;
OS_SEM SemFull;
OS_SEM SemEmpty;

int buffer[N];

int add, rem = 0;

int Producer_done = 0;

/*
*********************************************************************************************************
*                                            FUNCTION PROTOTYPES
*********************************************************************************************************
*/

void              TaskCons(void *data);              // Function prototypes of tasks
void              TaskProd(void *data);
```

Suite page suivante

```c
/*
*********************************************************************************************************
*                                                MAIN
*********************************************************************************************************
*/

int main (void)
{

 OS_ERR err;
                                                                // InitializeuC/OS-II
 CPU_IntInit();

 Mem_Init();                                                    // Initialize Memory Managment Module
 CPU_IntDis();                                                  // Disable all Interrupts
 CPU_Init();                                                    // Initialize the uC/CPU services
 OSInit(&err);

 OSMutexCreate(&Mutex, "Mutex", &err);
 OSMutexCreate(&Mutex_done, "Mutex_done", &err);

 OSSemCreate(&SemFull, "SemFull", 0, &err);
 OSSemCreate(&SemEmpty, "SemEmpty", N, &err);

 OSTaskCreate(&TaskCons1TCB,"TaskCons1", TaskCons, (void *) 0, TASK_CONS1_PRIO,
    &TaskCons1Stk[0u], TASK_STK_SIZE/2, TASK_STK_SIZE, 20, 0, (void *) 0,
    (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &err);
 OSTaskCreate(&TaskCons2TCB,"TaskCons2", TaskCons, (void *) 0, TASK_CONS2_PRIO,
    &TaskCons2Stk[0u], TASK_STK_SIZE/2, TASK_STK_SIZE, 20, 0, (void *) 0,
    (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &err);
 OSTaskCreate(&TaskCons3TCB,"TaskCons3", TaskCons, (void *) 0, TASK_CONS3_PRIO,
    &TaskCons3Stk[0u], TASK_STK_SIZE/2, TASK_STK_SIZE, 20, 0, (void *) 0,
    (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &err);
 OSTaskCreate(&TaskCons4TCB,"TaskCons4", TaskCons, (void *) 0, TASK_CONS4_PRIO,
    &TaskCons4Stk[0u], TASK_STK_SIZE/2, TASK_STK_SIZE, 20, 0, (void *) 0,
    (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &err);
 OSTaskCreate(&TaskCons5TCB,"TaskCons5", TaskCons, (void *) 0, TASK_CONS5_PRIO,
    &TaskCons5Stk[0u], TASK_STK_SIZE/2, TASK_STK_SIZE, 20, 0, (void *) 0,
    (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &err);
 OSTaskCreate(&TaskProdTCB,"TaskProd", TaskProd, (void *) 0, TASK_PROD_PRIO,
    &TaskProdStk[0u], TASK_STK_SIZE/2, TASK_STK_SIZE, 20, 0, (void *) 0,
    (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &err);


 OSStart(&err);

 return 1;                                          // Start multitasking

}
```

Suite page suivante

```
121  void  TaskCons (void *pdata)
122  {
123     OS_ERR err;
124     CPU_TS ts;
125
126     int out;
127
128     pdata = pdata;
129     printf("Cons no %d: demarre\n",OSPrioCur);
130     while(1){
131
132        OSMutexPend(&Mutex_done, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
133        if (Producer_done) {                                  // Le producteur a-t-il terminé sa production
134           OSMutexPost(&Mutex_done, OS_OPT_POST_NONE, &err);
135           OSSemPend(&SemFull,0, OS_OPT_PEND_NON_BLOCKING, &ts, &err);  // On prend soin de ne pas resté bloqué
136           if (err == OS_ERR_PEND_WOULD_BLOCK)  {
137           printf("Cons no %d: va se detruire\n",OSPrioCur);
138           OSTaskDel((OS_TCB *) 0, &err);                     // Si il n'y a plus rien à consommer
139                                                              // c'est la fin!
140           }
141           else  {
142              OSMutexPend(&Mutex, 0, OS_OPT_PEND_BLOCKING, &ts, &err); // sinon on consomme
143              out = buffer[rem++];                                     //memcpy
144              if(rem>=N)
145                 rem =0;
146              OSMutexPost(&Mutex, OS_OPT_POST_NONE, &err);
147              OSSemPost(&SemEmpty, OS_OPT_POST_1, &err );
148              printf("Cons appel sans delai no %d:  %d\n",OSPrioCur, out);
149              OSTimeDly(10, OS_OPT_TIME_DLY, &err );
150           }
151        }
152        else {                                                // Tout fonctionne normalement
153           OSMutexPost(&Mutex_done, OS_OPT_POST_NONE, &err);
154           printf("Cons no %d: Va se mettre en attente de SemFull\n",OSPrioCur);
155           OSSemPend(&SemFull,10, OS_OPT_PEND_BLOCKING, &ts, &err);    // Pour éviter de rester pris on attend 10 ticks
156                                                              // P.S. 10 ticks est un choix, ça aurait pu être moins ou plus
157           if (err != OS_ERR_TIMEOUT) {
158              OSMutexPend(&Mutex, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
159              out = buffer[rem++];
160              if(rem>=N)
161                 rem =0;
162              OSMutexPost(&Mutex, OS_OPT_POST_NONE, &err);
163              OSSemPost(&SemEmpty, OS_OPT_POST_1, &err );
164              printf("Cons appel avec delai no %d:  %d\n",OSPrioCur, out);
165              OSTimeDly(10, OS_OPT_TIME_DLY, &err);
166           }
167        }
168
169     }
170  }
```

Suite page suivante

```
172
173     void  TaskProd (void *data)
174     {
175        OS_ERR err;
176         CPU_TS ts;
177
178        int i, in=0;
179        data = data;
180
181  printf("Prod %d: demarre\n",OSPrioCur);
182
183        while(1){
184
185           for (i = 1; i <= 7; i++) {
186
187              OSSemPend(&SemEmpty, 0, OS_OPT_PEND_BLOCKING, &ts, &err); // Acquire semaphore
188              OSMutexPend(&Mutex, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
189              buffer[add++] = in++;           //memcpy
190              if(add>=N)
191                 add =0;
192              printf("Production de la valeur %d:\n",(in-1));
193              OSMutexPost(&Mutex, OS_OPT_POST_NONE, &err);            // Release semaphore
194              OSSemPost(&SemFull, OS_OPT_POST_1, &err );
195           }
196           printf("Fin de la Production\n");
197           OSMutexPend(&Mutex_done, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
198           Producer_done = 1;
199           OSMutexPost(&Mutex_done, OS_OPT_POST_NONE, &err);
200           OSTaskDel((OS_TCB *) 0, &err);
201        }
202     }
```

Q 29

Soit le code uC/OS-III suivant sur 3 pages. Faire la trace d'exécution de ce code.

```
#include  <cpu.h>
#include  <lib_mem.h>
#include  <os.h>

#include  "os_app_hooks.h"
#include  "app_cfg.h"

/*
*********************************************************************************************************
*                                                CONSTANTS
*********************************************************************************************************
*/

#define         TASK_STK_SIZE    8192               // Size of each task's stacks (# of WORDs)


#define         TASK_BASSE_PRIO     10               // Application tasks priorities
#define         TASK_HAUTE_PRIO     8
#define         TASK_MILIEU_PRIO    9


/*
*********************************************************************************************************
*                                                VARIABLES
*********************************************************************************************************
*/

CPU_STK         TaskBASSEStk[TASK_STK_SIZE];        // Startup    task stack
CPU_STK         TaskHAUTEStk[TASK_STK_SIZE];
CPU_STK         TaskMILIEUStk[TASK_STK_SIZE];

OS_TCB          TaskBASSETCB;
OS_TCB          TaskHAUTETCB;
OS_TCB          TaskMILIEUTCB;

  /* This disables all code until the next "#endif" */

OS_MUTEX        Mutex;
OS_SEM          Sem;            // le semaphore emule le reveil de la tache la plus prioritaire
OS_SEM          Synchro;        // idem


/*
*********************************************************************************************************
*                                          FUNCTION PROTOTYPES
*********************************************************************************************************
*/

void            TaskBASSE(void *data);              // Function prototypes of tasks
void            TaskduMilieu(void *data);
void            TaskHAUTE(void *data);
void            PrintOwnerofMutexanditsPriority(char *str, OS_MUTEX  *p_mutex);
```

Suite page suivante

```c
/*
*********************************************************************************************************
*                                                MAIN
*********************************************************************************************************
*/

int main (void)
{

    OS_ERR  err;
    CPU_IntInit();

    Mem_Init();                                                 // Initialize Memory Managment Module
    CPU_IntDis();                                               // Disable all Interrupts
    CPU_Init();                                                 // Initialize the uC/CPU services

    OSInit(&err);

    OSMutexCreate(&Mutex, "Mutex", &err);
    OSSemCreate(&Sem, "Sem", 0, &err);
    OSSemCreate(&Synchro, "Synchro", 0, &err);

    OSTaskCreate(&TaskBASSETCB,"TaskBASSE", TaskBASSE, (void *) 0, TASK_BASSE_PRIO,
       &TaskBASSEStk[0u], TASK_STK_SIZE/2, TASK_STK_SIZE, 20, 0, (void *) 0,
       (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &err);

    OSTaskCreate(&TaskMILIEUTCB,"TaskMILIEU", TaskduMilieu, (void *) 0, TASK_MILIEU_PRIO,
       &TaskMILIEUStk[0u], TASK_STK_SIZE/2, TASK_STK_SIZE, 20, 0, (void *) 0,
       (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &err);

    OSTaskCreate(&TaskHAUTETCB,"TaskHAUTE", TaskHAUTE, (void *) 0, TASK_HAUTE_PRIO,
       &TaskHAUTEStk[0u], TASK_STK_SIZE/2, TASK_STK_SIZE, 20, 0, (void *) 0,
       (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), &err);


    OSStart(&err);
    return 1;                                           // Start multitasking
}

/*
*********************************************************************************************************
*                                              STARTUP TASK
*********************************************************************************************************
*/

void  TaskBASSE (void *data)
//Task Low
{
    OS_ERR  err;
    CPU_TS ts;
    while(1)
    {
        printf("\nTB - Avant le muxtex\n");
        PrintOwnerofMutexanditsPriority("TB", &Mutex);
        OSMutexPend(&Mutex, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
        printf("\nTB - Apres mutex\n");
        PrintOwnerofMutexanditsPriority("TB", &Mutex);
        printf("\nTB - Avant Syncro1\n");
        OSSemPost(&Synchro, OS_OPT_POST_1, &err);
        printf("\nTB - Apres Syncro1\n");
        PrintOwnerofMutexanditsPriority("TB", &Mutex);
        OSMutexPost(&Mutex, OS_OPT_POST_NONE, &err);
        printf("TB - Apres mutex\n");
        OSTimeDlyHMSM(1,0,0,0, OS_OPT_TIME_DLY, &err);
    }
}
```

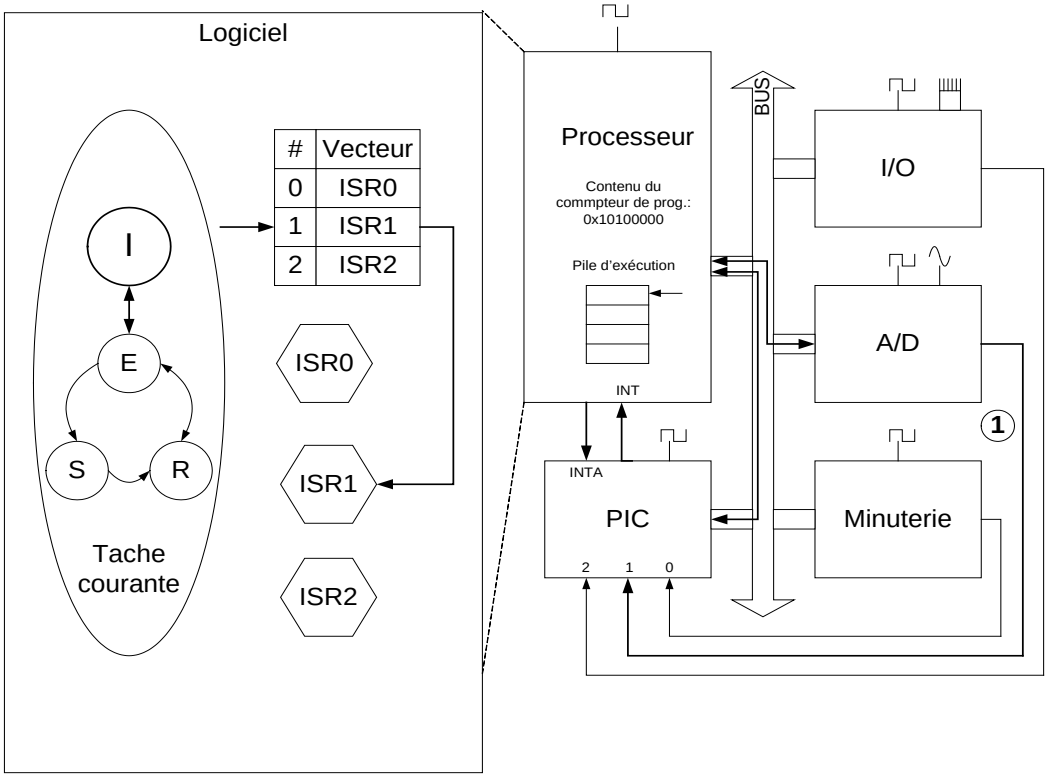
Suite page suivante

```
123  void TaskduMilieu (void *data)
124  {
125      OS_ERR  err;
126      CPU_TS ts;
127
128      while(1)
129      {
130          OSSemPend(&Synchro,0, OS_OPT_PEND_BLOCKING, &ts, &err);
131          printf("\nTM - Apres Synchro\n");
132          OSSemPost(&Sem, OS_OPT_POST_1, &err);
133          printf("\nTM - Apres Synchro\n");
134          OSTimeDlyHMSM(1,0,0,0, OS_OPT_TIME_DLY, &err);
135      }
136  }
137
138  void TaskHAUTE(void *data)
139  //Task high
140  {
141      OS_ERR  err;
142      CPU_TS ts;
143
144      int cnt = 0;
145      while(1)
146      {
147          OSSemPend(&Sem,0, OS_OPT_PEND_BLOCKING, &ts, &err);
148          printf("\nTH - Apres le Sem et avant Mutex\n");
149          OSMutexPend(&Mutex, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
150          printf("\nTH - Apres le Mutex\n");
151          PrintOwnerofMutexanditsPriority("TH", &Mutex);
152          OSMutexPost(&Mutex, OS_OPT_POST_NONE, &err);
153          PrintOwnerofMutexanditsPriority("TH", &Mutex);
154          OSTimeDlyHMSM(1,0,0,0, OS_OPT_TIME_DLY, &err);
155      }
156
157
158  }
159
160  void PrintOwnerofMutexanditsPriority(char *str, OS_MUTEX  *p_mutex)
161  {
162          if (p_mutex->OwnerNestingCtr != 0u){                    /* Resource available?
163          printf("\nJe suis actuellement dans: %s\n", str);
164          printf("Le mutex est utilise par la tache : %s\n", p_mutex->OwnerTCBPtr->NamePtr);
165          printf("et la priorite courante de cette tache est : %d\n", p_mutex->OwnerTCBPtr->Prio);
166          }
167          else printf("\nLe mutex est libre\n");
168  }
```

Q30

Soit un système d'acquisition de données en temps réel dont le schéma se trouve à la figure 36.1 (page suivante). Ce système est composé d'un microprocesseur, d'un contrôleur d'interruption (*PIC*), d'une minuterie (timer), d'un convertisseur analogique/numérique (A/D) et d'une interface d'entrée/sortie (I/O). Un système d'exploitation temps réel ordonnançant plusieurs tâches est exécuté par le microprocesseur. Le convertisseur analogique numérique fonctionne par interruptions et utilise l'interruption 1. La routine d'interruption (*ISR*) qui lui est associée est chargée de lire la donnée sur le convertisseur.

1) Complétez le schéma de la figure 36.1 vous devez expliquer dans l'ordre chacune des étapes du cycle d'interruption du A/D. Plus précisément, pour chaque étape inscrivez un numéro sur la figure 36.1 et expliquez ce qui se passe durant cette étape.



1. Le convertisseur analogique / numérique envoie une interruption sur la broche 1 du PIC (contrôleur d'interruption)
2. …

À compléter…

Figure 30.1

2) Dans ce même système, la minuterie émet une interruption à toutes les 10 ms et la tâche la plus prioritaire demande un délai de 22 ms. Expliquez les deux imprécisions liées à ce délai. Complétez le diagramme de la figure 36.2 en faisant la trace d'exécution mettant en évidence ces imprécisions. (le temps d'exécution de l'*ISR* de la minuterie est considéré comme négligeable).

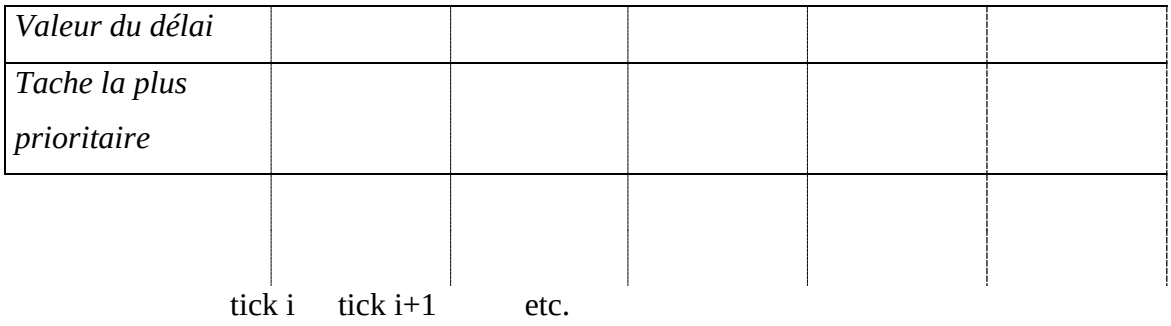


Figure 30.2 Diagramme à compléter

Q31

Soit le système utilisant un RTOS multitâche et composé entre autres d'un CPU PIII, d'un compteur/timer 32 bits, d'un contrôleur d'interruption et d'autres périphériques. Le CPU fonctionne à une fréquence de 500 MHz et le compteur/timer est raccordé à un signal d'horloge à 2MHz.

a) Calculer la valeur à laquelle le compteur du timer doit être initialisé par le RTOS pour qu'il délivre des interruptions avec une période de 5 ms.

b) Dans une des tâches du système, on fait appel à un délai de 12 ms. Sachant que le timer est réglé à 5 ms, quelle sera la précision de ce délai? Expliquez et faites un schéma de l'exécution.

c) On souhaiterait améliorer la précision pour ce délai de 12 ms. Sans ajouter de matériel, quelle solution facile pouvez vous proposer? Quelle conséquence a-t-elle sur l'ensemble du système, quelle sont les limitations? Expliquez et faites un schéma de l'exécution.

d) Toujours pour ce délai de 12 ms, proposez la solution donnant la meilleure précision (vous êtes libre d'ajouter du matériel). Décrivez son fonctionnement et son implémentation en quelques lignes.

Q32

Faites la trace d'exécution pour un cycle du code ci-après (considérez qu'il n'y a que ces 2 tâches dans le système)

Ensuite, expliquez en quelques lignes ce que fait ce code.

```
#define TASK_PERIOD_PRIO  11    // Priorité de TachePeriodic
#define TASK_TIMER_PRIO   10    // Priorité de TacheTimer
...
#define PERIOD 500 // Période de TachePeriodic en tick d'horloge
...

OSEVENT *SemStart;
OSEVENT *SemStop;


void main(void)
{
      ...


      SemStart = OSSemCreate(0);
      SemStop = OSSemCreate(0);


      ...
}


void TachePeriodic(void* p)
{
      ...


      while(1)
      {
            OSSemPost(SemStart);


            ...
            //Code de durée variable (<PERIOD) a rendre périodique
            ...
```

```
            OSSemPend(SemStop,0,&err);
      }
}

void TacheTimer(void* p)
{
      while(1)
      {
            OSSemPend(SemStart,0,&err);
            OSTimeDly(PERIOD);
            OSSemPost(SemStop);
      }
}
```

Q34

À partir de la figure 34.1, nous avons 8 évènements étiquetés de 1 à 8. Considérez que *Tache_A* a une priorité de 4, alors que *Tache_B* a une priorité de 3 et que le mécanisme héritage de priorité est utilisé.

a) Pour chaque évènement, vous devez :

1) Donnez l'état de la tâche correspondante (aidez-vous au besoin de la figure 34.2, page suivante);
2) Les détails internes de ce qui se passe dans le noyau lorsque l'interruption se produit et que les fonctions *pend* et post sont appelées; par exemple décrire ce qui se passe au niveau des structures et des appels systèmes.
3) S'il y a lieu, les changements de priorité lorsque l'ordonnanceur est appelé ou lorsque le mécanisme héritage de priorité est appliqué.

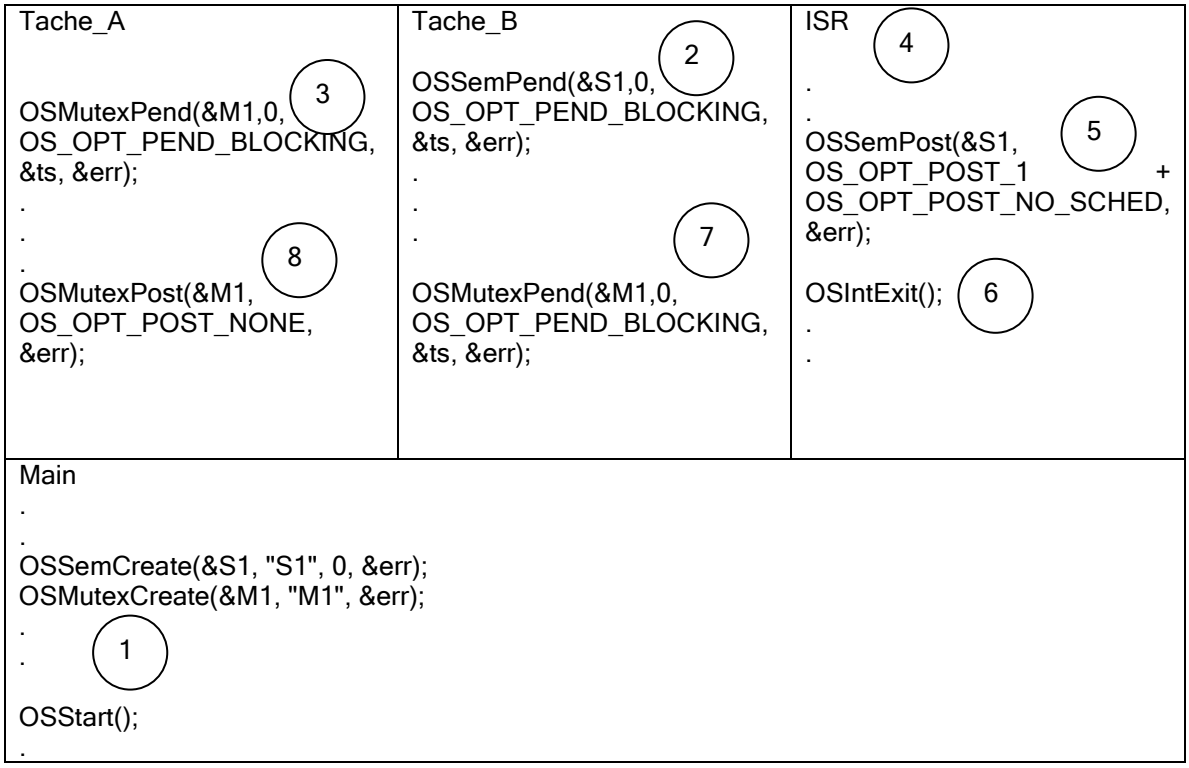


Figure 34.1

b) En considérant maintenant le mécanisme ICPP, donnez les différences par rapport à a) (juste les différences).

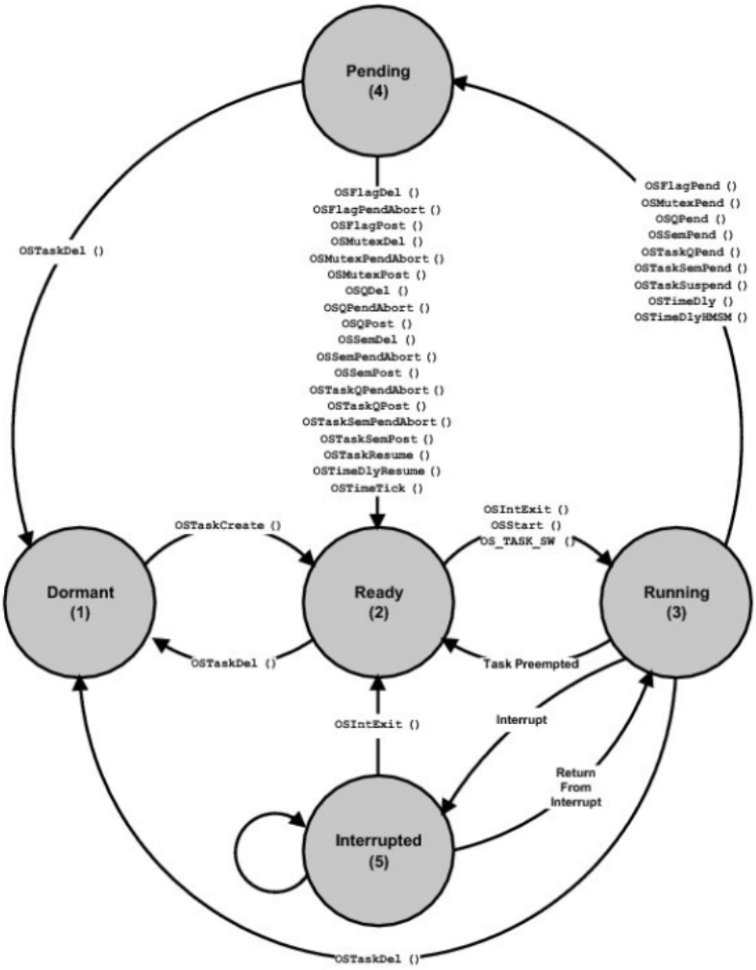

Pending
(4)
OSFlagDel ()
OSFlagPendAbort ()
OSFlagPost ()
OSMutexDel ()
OSMutexPendAbort ()
OSMutexPost ()
OSQDel ()
OSQPendAbort ()
OSQPost ()
OSSemDel ()
OSSemPendAbort ()
OSSemPost ()
OSTaskQPendAbort ()
OSTaskQPost ()
OSTaskSemPendAbort ()
OSTaskSemPost ()
OSTaskResume ()
OSTimeDlyResume ()
OSTimeTick ()
OSTaskDel ()
OSFlagPend ()
OSMutexPend ()
OSQPend ()
OSSemPend ()
OSTaskQPend ()
OSTaskSemPend ()
OSTaskSuspend ()
OSTimeDly ()
OSTimeDlyHMSM ()
OSTaskCreate ()
OSIntExit ()
OSStart ()
OS_TASK_SW ()
Dormant
(1)
Ready
(2)
Running
(3)
OSTaskDel ()
Task Preempted
OSIntExit ()
Interrupt
Return
From
Interrupt
Interrupted
(5)
OSTaskDel ()

Figure 34.2

Q35

Soit la trace d'exécution de la figure 35.1 pour 6 tâches µC. Considérez que T10, T20, T30, T40, T50 et T60 ne peuvent démarrent qu'à partir des ticks 18, 16, 13, 11, 7 et 1 respectivement.

a) Pour chaque tick d'horloge (1 à 35), donnez la priorité de la tâche 60. Indiquez-la directement en dessous de la tâche T60 à la figure 3.1.

b) Remplissez la table OSRdyTbl pour les ticks 12, 14, 19 et 28 (donc 4 photos de OSRdyTbl à différents instants). Considérez ici que uC/OS-III a créée une table de 64 tâches lors de OSInit().

c) Complétez la trace de la figure 35.2 en considérant cette fois le protocole ICPP. Si c'est le cas, indiquez les améliorations pour T10 au niveau temps de blocage (par rapport à celui obtenu à la figure 3.1).

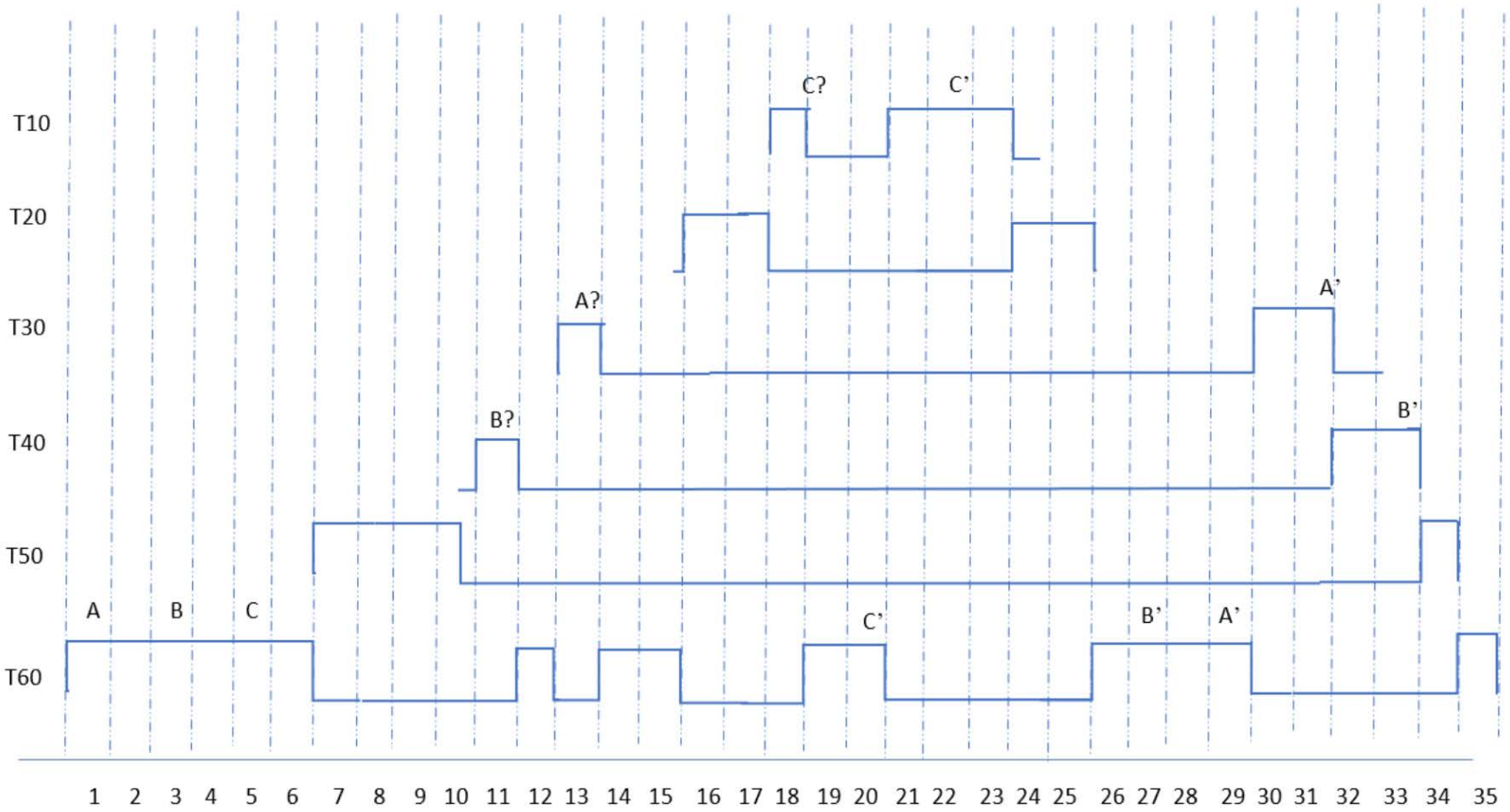

T10
T20
T30
T40
T50
T60
C?
C'
A?
A'
B?
B'
A
B
C
C'
B'
A'
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

Figure 35.1

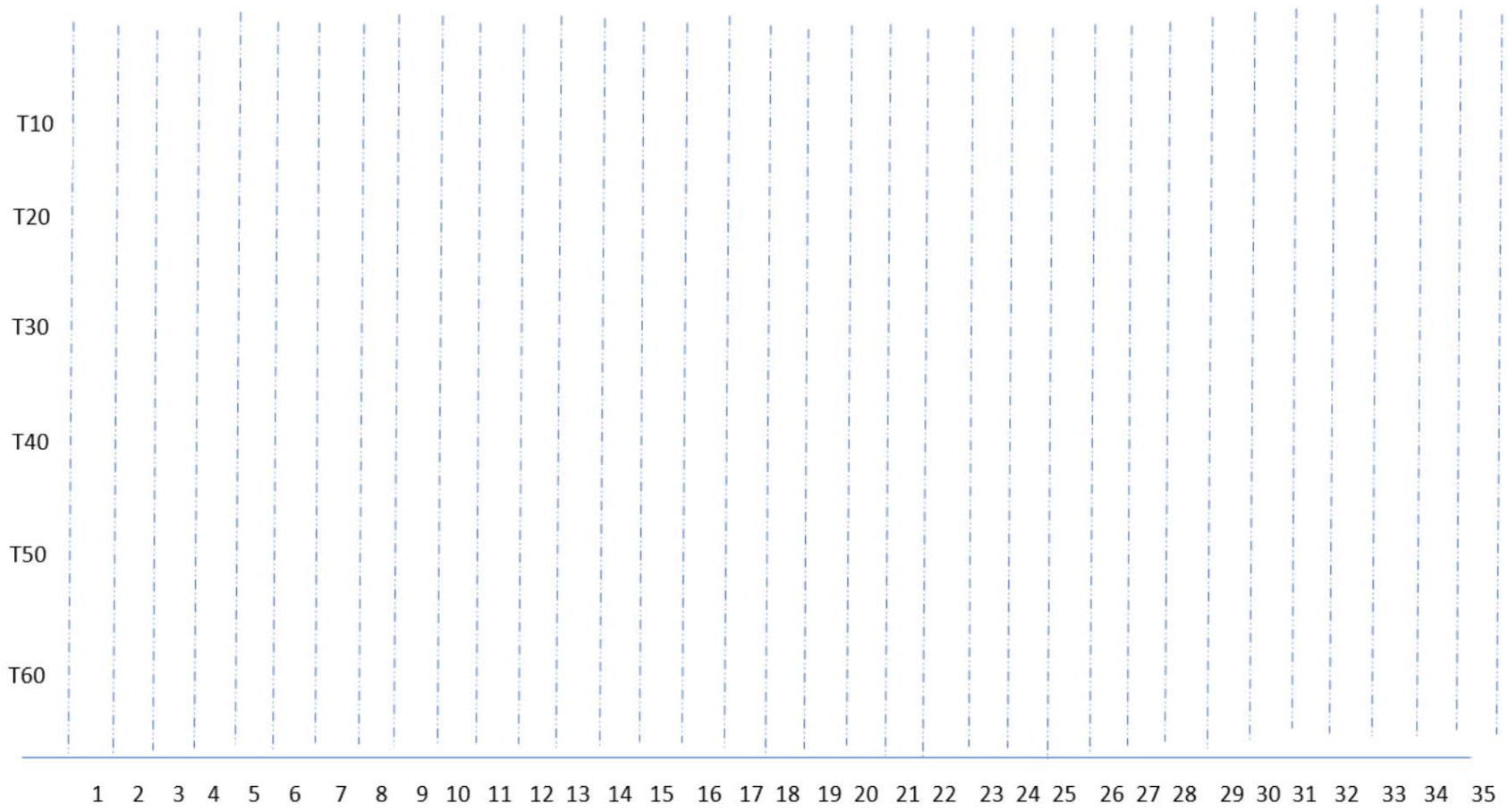


Figure 35.2 À compléter en considérant ICPP

Q38

Expliquez comment les structures des question 37 et 38 se complète lors de l'exécution.

Q39

Soit l'état d'un système à un instant *t* sous µC tel que décrit à la Figure de la page suivante :

a) Décrivez l'état des différentes tâches de ce système à l'instant *t*.

b) On suppose que OSSched() est appelé à l'instant *t*, décrivez comment OSSched() détermine la tâche la plus prioritaire qui sera exécutée.

c) On suppose que la tâche trouvée en b) fait appel à *OSSemPend(S1, 0, &err)*. Dans un ordre chronologique, décrivez la suite d'évènements **en précisant tous les changements** qui conduisent aux mises à jour des structures de données et déterminez la tâche la plus prioritaire.

d) On suppose que la tâche trouvée en c) fait appel à *OSSemPost(S1, 0, &err)*. Dans un ordre chronologique, décrivez la suite d'évènements **en précisant tous les changements** qui conduisent aux mises à jour des structures de données et déterminez la tâche la plus prioritaire.

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1

OSRdyTbl