Q16

```
#define     lecteur                 3       // Priorité au lecteur
#define         redacteur           4

OS_MUTEX  access, ecrit;
OS_ERR err;
OSMutexCreate(&access, "access", &err);
OSMutexCreate(&ecrit, "ecrit", &err);
int nb_lecteurs = 0;

// Création des tâches
.
.
void lecteur(*pdata) {
while(1) {
OSMutexPend(&access, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
++ nb_lecteurs ;
if (nb_lecteurs =  1)  OSMutexPend(&ecrit, 0, OS_OPT_PEND_BLOCKING, &ts,&err);
OSMutexPost(&access, OS_OPT_POST_NONE, &err);
read;
OSMutexPend(&access, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
-- nb_lecteurs ;
if (nb_lecteurs =  0)  OSMutexPost(&ecrit, OS_OPT_POST_NONE, &err);
OSMutexPost(&access, OS_OPT_POST_NONE, &err);
}

void redacteur (*pdata) {
while(1) {
OSMutexPend(&ecrit, 0, OS_OPT_PEND_BLOCKING, &ts,&err);
write;
OSMutexPost(&ecrit, OS_OPT_POST_NONE, &err);
}
```

Q25

a) Les interruptions sont toujours plus prioritaires que les tâches. Ensuite, la tâche de calcul sera plus prioritaire que la tâche d'affichage car sa fréquence d'exécution et son importance sont plus élevées.

b) Les périodes d'acquisition durent 5 ms et il y a une acquisition toutes les 20µs donc on acquière au total 250 points (5 / (20E-3)). Durant une période d'acquisition l'ISR occupe le CPU 50% du temps ce qui laisse donc au total 2.5ms pour la tâche de calcul. En 2.5ms elle peut traiter 100 points (2.5 / (25E-3)). Il lui reste donc 150 points à traiter durant la période de non-acquisition, ce qui lui prend donc 3.75ms.

La taille minimum du buffer de calcul devrait donc être de 150 points, par sécurité on choisira une taille de 300 points (au cas où le ISR demanderait un peu plus que 2.5 ms car ce n'est jamais complètement exact!).

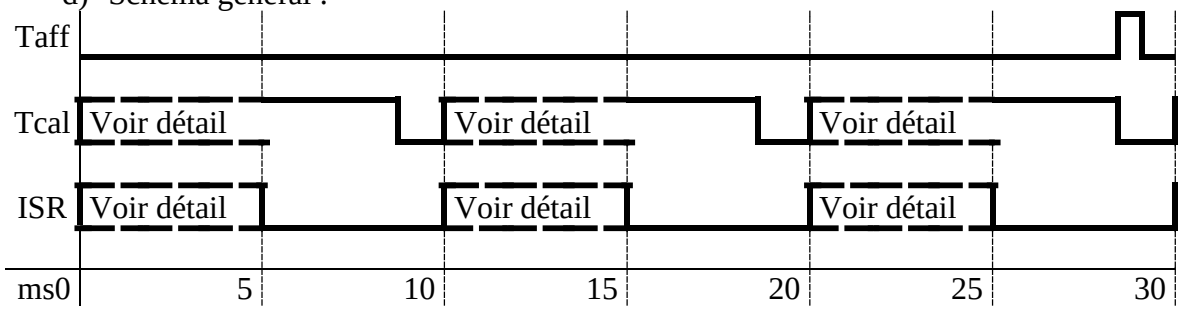
Le buffer d'affichage devant contenir trois séries d'acquisition, il sera donc de 750 points. On peut aussi doubler pour être bien certain de rien perdre…

c) Durant 30 ms le CPU réalise les tâches suivantes:

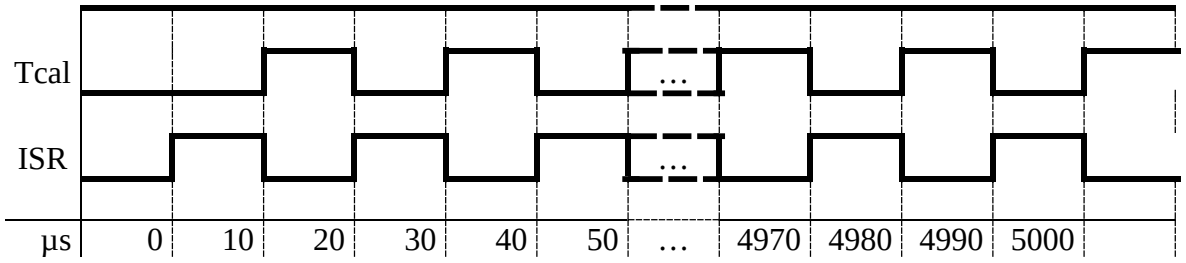
1) 7.5 ms pour les interruptions (3 streams de 2.5ms)
2) 18.75 ms pour le calcul (3 streams de 250 points * 25us = 18.75 ms)
   Ou encore 3 * 100 points = 7.5ms durant les 3 streams + 3 * 150 points = 3 * 3.75ms = 11.25ms après les streams (via le fifo) pour un grand total de 18.75 ms pour 3 streams
3) 1 ms pour l'affichage

Au total ça fait 27.25 ms de traitement sur 30 ms donc 90.83% de pourcentage d'occupation du CPU

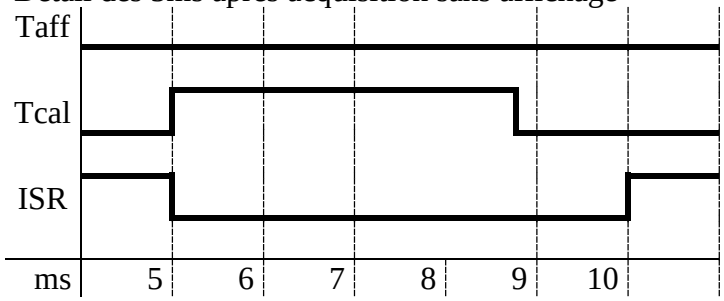
d) Schéma général :



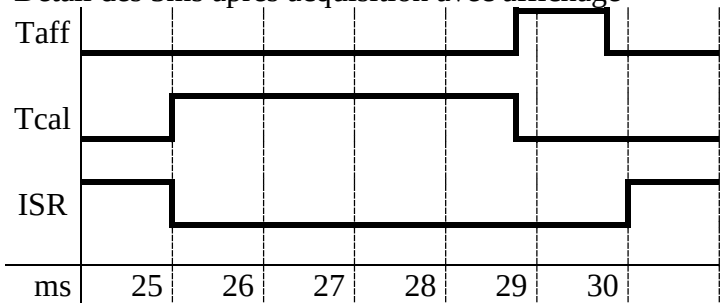
Détail des 5ms d'acquisition



Détail des 5ms après acquisition sans affichage



Détail des 5ms après acquisition avec affichage

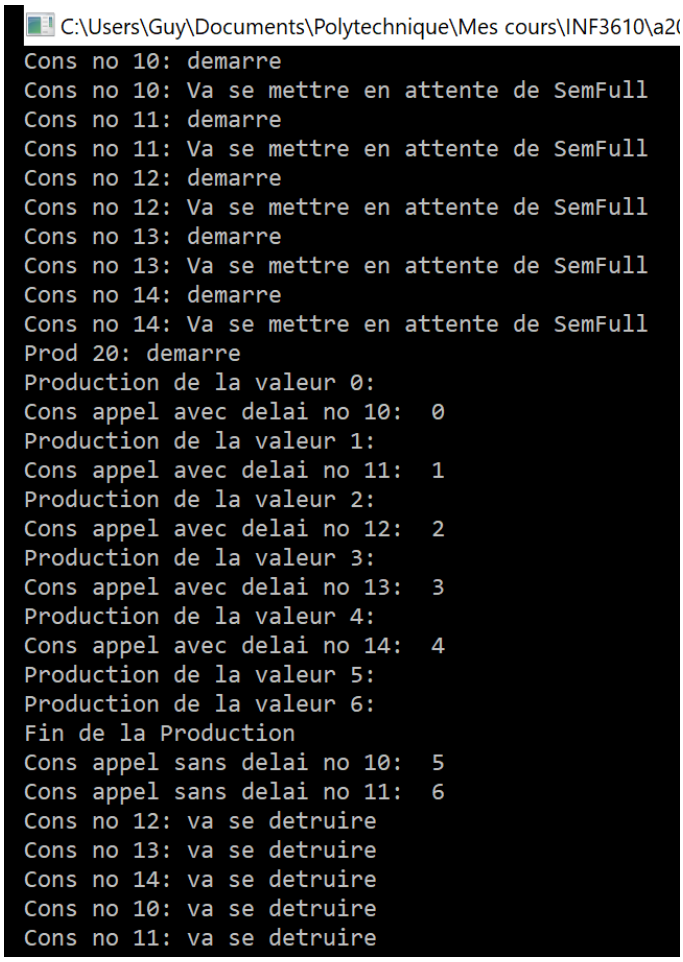


Q26

```
OS_SEM  S1, S2, S3;
S1 = OSSemCreate(0);    // Ici on simplifie les paramètres, on veut juste montrer que le
                        //sémaphore est initialisé à 0.
S2 = OSSemCreate(0);
S3 = OSSemCreate(0);

// Création des tâches
.
.
.
// Pour améliorer la lisibilité, on a simplifier les paramétres des appels de fcts
Void T1()
{
  while(1)
  {
    U();
    OSSemPost(S1);
    OSSemPend(S3);
    X();
    OSSemPost(S1);
    OSSemPend(S3);
  }
}

Void T2()
{
  while(1)
  {
    OSSemPend(S1);
    V();
    OSSemPost(S2);
    OSSemPend(S1);
    Y();
    OSSemPost(S2);
  }
}

Void T3()
{
  while(1)
  {
    OSSemPend(S2);
    W();
    OSSemPost(S3);
    OSSemPend(S2);
    Z();
    OSSemPost(S3);
  }
}
```

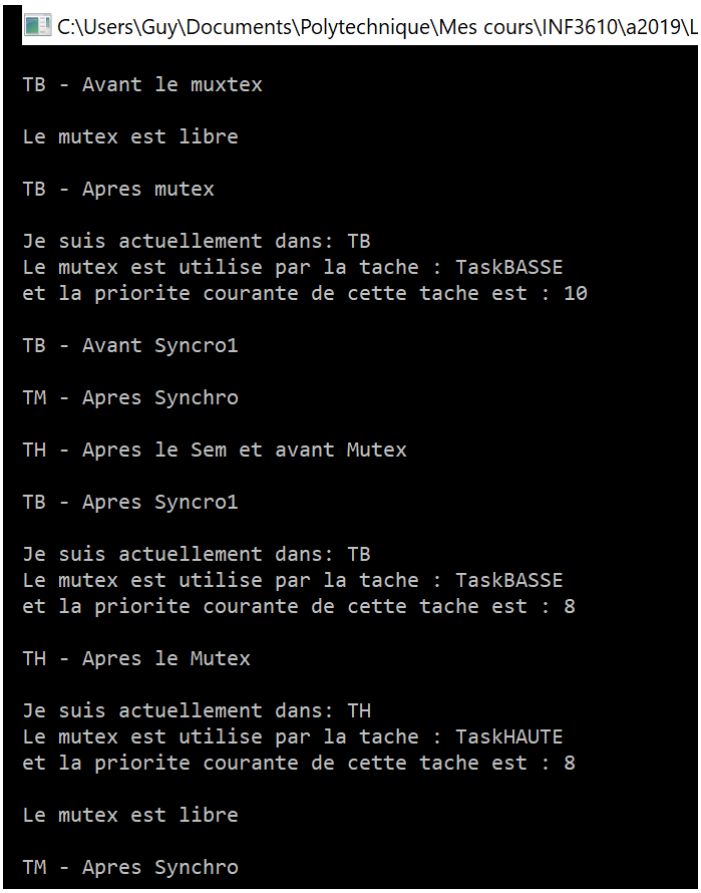
Q28

On a analysé ce code en classe. Vous pouvez l'exécuter dans les exemples de code sur le web (chap. 2). Il s'agit du fichier *1 producteur n consommateurs.c.* La trace donne :



Q29

On a analysé ce code en classe. Vous pouvez l'exécuter dans les exemples de code sur le web (chap. 2). Il s'agit du fichier mutex.c. La trace donne :



Q30

1)

1 Le A/D émet une interruption vers le PIC (contrôleur d'interruption)
2 Le PIC reçoit l'interruption et la transmet au processeur
3 Le processeur interrompt la tâche en cours d'exécution en sauvant son adresse d'exécution (PC) sur le dessus de la pile
4 L'état de la tâche passe à interrompue
5 Le processeur interroge le PIC pour connaître le numéro de l'interruption
6 Le processeur entre dans la table des vecteurs d'interruption à la ligne donné par le PIC
7 Le processeur saute jusqu'au code de la routine d'interruption (ISR)(grâce au vecteur trouvé dans la table)
8 Les interruption sont permises à nouveau (INTA=1) sur le PIC
9 L'ISR va lire la donné sur le A/D
10 L'adresse de sur la pile (+ 1) est remis dans le PC pour que la tâche en cours poursuive son exécution.
11 L'état de la tâche passe à exécuté

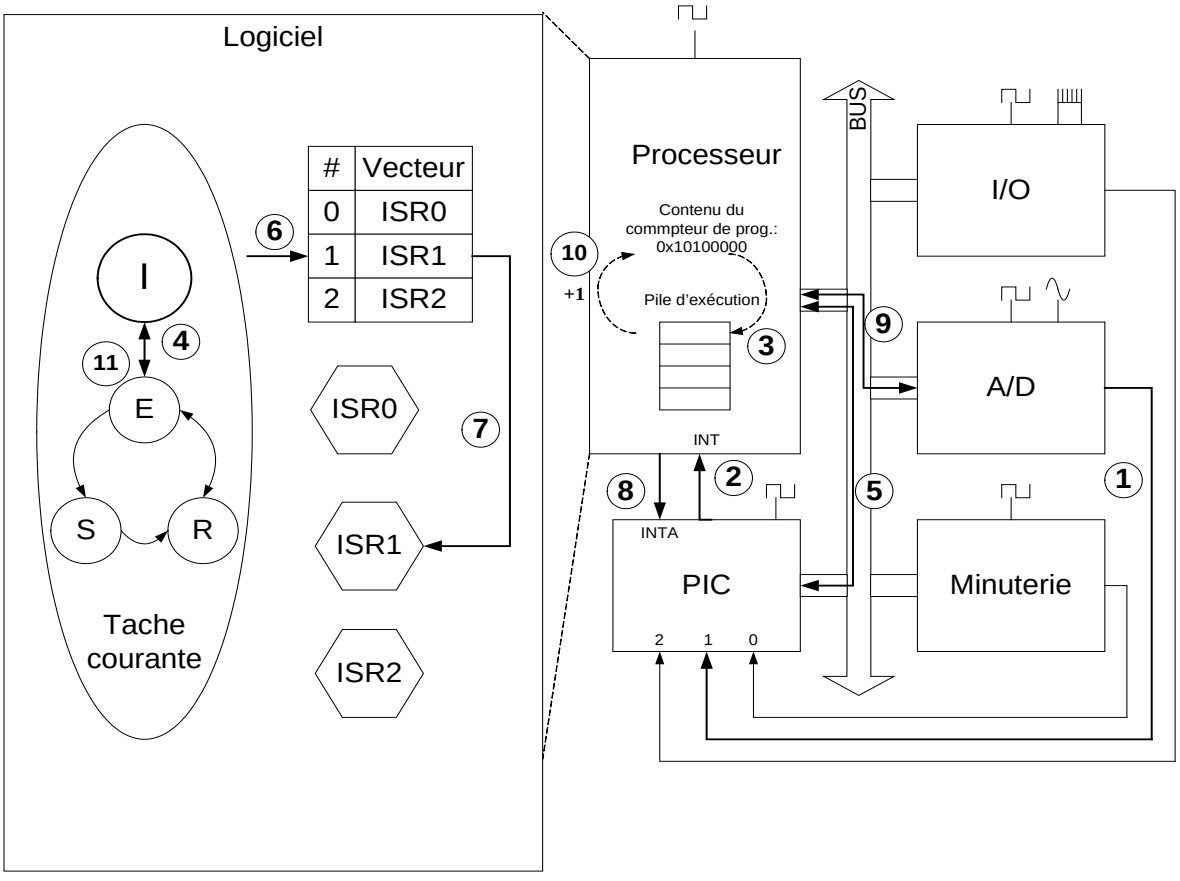
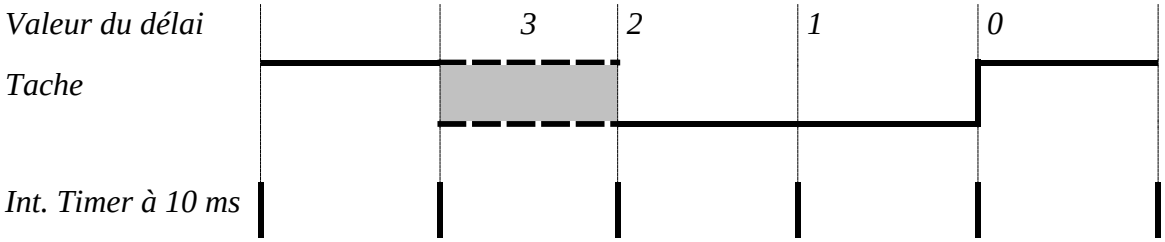


Figure 30.1 avec numéros

2)

22 / 10 = 2.2 Le délais sera donc de 3 tick (la politique d'arrondi du système au supérieur). Ensuite il y a une incertitude de 1 tick dépendent de l'instant où on demande le délais par rapport à la période du timer.

Admettons que le délai est été arrondi à 3 ticks :



La zone grise représente le moment ou la tache fait appel au délai.

Ici le délai de 22 ms a été arrondi à 3 tick soit 30 ms, puis s'ajoute a cela une incertitude de 1 tick : donc le délais pourra aller de 20 ms á 30 ms.

Figure 30.2 Diagramme complété

Q31

a)

Le timer est décrémenté avec une fréquence de 2 MHz et il génère une interruption à chaque passage par zéro. Sa valeur d'initialisation est donc égale :

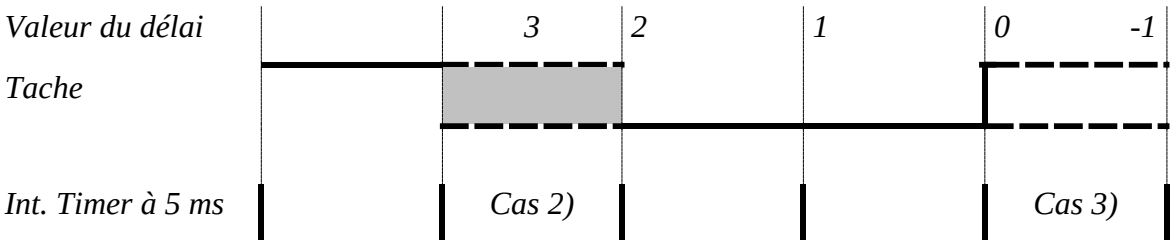
$C = f_{clock} * T_{tick} = 2 * 10 \wedge 6 * 5 * 10 \wedge -3 =$ ***10 000***

b)

12 / 5 = 2.4

Il existe 3 sources contribuant à l'incertitude :

1) *Le délai sera de 2 ou 3 ticks suivant la politique d'arrondi du système.*
2) *Ensuite il y a une incertitude de 1 tick dépendent de l'instant où on demande le délai par rapport à la période du timer. Admettons que le délai est été arrondi à 3 ticks :*



La zone grise représente le moment ou la tache fait appel au délai.

Ici le délai de 12 ms a été arrondi à 3 tick soit 15 ms, puis s'ajoute a cela une incertitude de 1 tick : donc le délai pourra aller de 10 ms á 15 ms.

(dans le cas ou il serait arrondis à 2 ticks, le délai serait compris entre 5 et 10 ms)

3) *On peut finalement ajouter une incertitude de 1 tick dépendent de l'instant où on reprend l'exécution par rapport à la période du timer. On assume qu'on va reprendre entre 0 et -1 tick (-5ms).*

Conclusion : Le pire cas serait qu'on commence à 15ms et qu'on revient en exécution à -5ms soit un délai total de 20 ms pour une erreur de 8ms.

c)

La façon la plus rapide d'augmenté la précision de ce délai est de prendre une fréquence d'horloge qui est un multiple commun de 12 afin qu'il n'y ait plus d'arrondi. D'autre part en diminuant la période des interruptions du timer, on augmentera la précision des délais (incertitude de 1 tick).

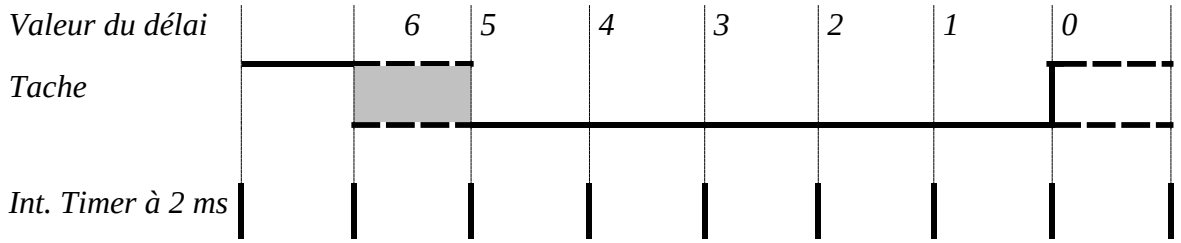
On peut prendre comme période du timer 4, 3, 2 ou 1 ms qui sont des sous multiple de 12.

Une période de 4 ms nous éviterait l'erreur d'arrondi, mais nous ferait gagner peut en précision final : 12 ms / 4 = délais de 3 ticks (2 à 3)= 8 à 12 ms pour l'incertitude de départ (cas 2) et 0 à -4 ms pour le redémarrage (cas 3) donc une erreur maximum (en absolu) de 4 ms. (Notez ici, que la 3e incertitude vient simplement diminuer l'erreur de 4 ms, en prolongeant le délai. On peut donc dire que lorsqu'il s'agit d'un multiple, la 3e incertitude peut être omise.)

Une période de 3 ou 2 ms est un choix plus judicieux

Avec 3 : 12 ms / 3 = délais de 4 ticks (3 à 4) = 9 à 12 ms pour un maximum de 3 ms d'incertitude

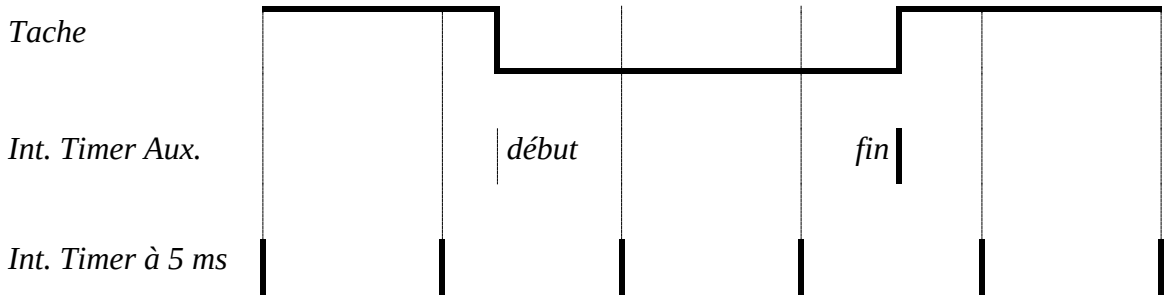
Avec 2 : 12 ms / 2 = délais de 6 ticks (5 à 6) = 10 à 12 ms pour un maximum de 2 ms d'incertitude



Une période de 1ms ou moins apporterait encore plus de précision, mais saturait le système car il y aurait trop d'interruption du timer à traiter.

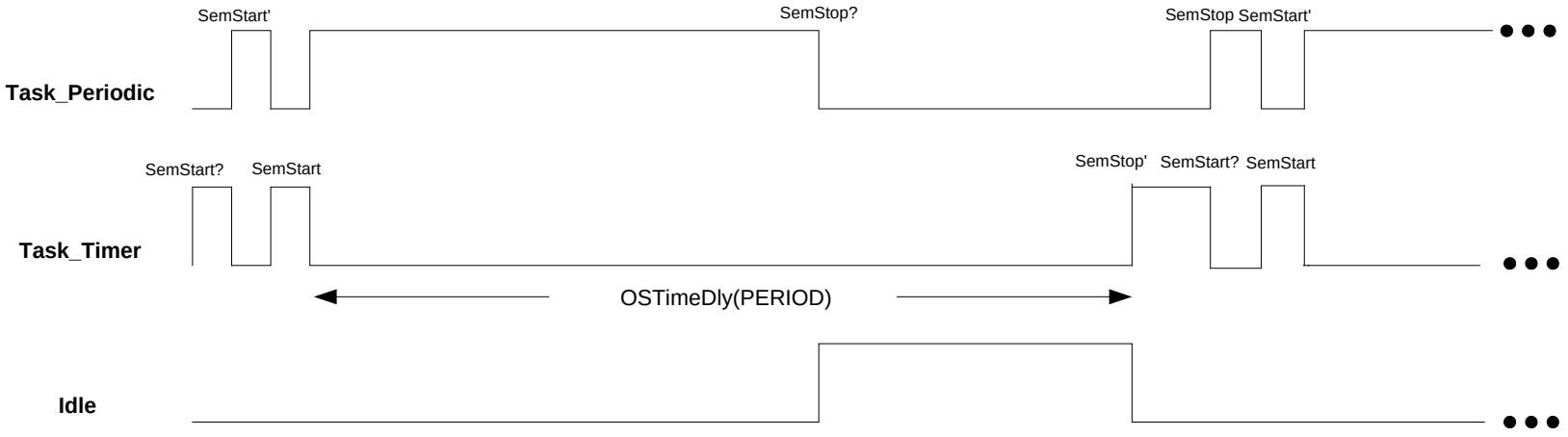
d)

La meilleure solution pour obtenir un délai de 12 ms avec le plus d'exactitude possible est d'utiliser un timer auxiliaire. On écrira une fonction spécifique pour demander un délai avec ce timer. Cette fonction programmera le timer auxiliaire avec une valeur correspondant au temps à décompter, lancera le timer auxiliaire, puis suspendra la tâche. Lorsque le délai expira, le timer auxiliaire émettra une interruption qui sera traitée par une ISR spécifique. Cette ISR réactivera la tache qui avait demandé le délai. De cette façon le délai durera exactement le temps voulu et commencera exactement au moment où il est demandé.



Notez aussi que dans le cas où on veut le faire de manière périodique sous uC on pourrait imaginer une fonction *OSTimeDlyAux* qui utiliserait un code similaire à celui du no 41.

Q32



Question (discuté en classe): quel avantage apporte l'ordonnancement proposé dans le no 32 par à rapport à la façon de faire suivante :

```
Task_Periodic (void)
{
   for (;;) {
     OSTimeDly(500,OS_OPT_TIME_DLY, &err);
         …
// Code de durée variable
         ….
   }
}
```

Voir aussi page suivante

Même question avec :

```
Task_Periodic (void)
{
   for (;;) {
      OSTimeDly(500,OS_OPT_TIME_PERIODIC, &err);
          ….
// Code de durée variable
          ….
   }
}
```

Q33 Voir le code sur le web (chap. 2). Il s'agit du fichier *minuterie.c*

Q34

a)

Event 1

- Tâche A : passe de DORMANT à READY
- Tâche B : passe de DORMANT à READY
- L'ordonnanceur est appelé et comme la Tâche B a la plus haute priorité cette dernière passe de READY à EXECUTE

Event 2

- Puisque S1 est à 0, la Tâche B passe de RUNNING à PENDING. La fonction système OS_PendListRemove() est exécutée et elle retire la Tâche B de OSPrioTbl (mise à 0). Puis B est mis dans la structure d'attente OS_PEND_OBJ de S1 (liste doublement chainée avec TailPtr et HeadPtr) avec l'appel à la fonction système OS_PendListInsertPrio(). L'ordonnanceur OSSched() est appelé et la prochaine tâche la plus propriétaire, la Tâche A, passe de READY à RUNNING.

Event 3

- La Tâche A exécute OSMutexPend() et puisque M1 est à 1 la Tâche A reste dans l'état RUNNING.

Event 4

- Une interruption externe survient. La Tâche A passe de RUNNING à ISR et la routine d'interruption de service (ISR) est exécutée.

Event 5

- ISR exécute OSSemPost(), ce qui fait passer la Tâche B de PENDING à READY.
- La fonction système OS_PendListRemove() est appelée et sélectionne dans OS_PEND_OBJ de S1 la tâche la plus prioritaire (pointeur sur HeadPtr) qui est B
- On exécute alors la fonction système OS_PrioInsert() qui remet B actif dans OSPrioTbl[].

OSCtxSW et OSIntCtxSW

Event 6

- Avant de terminer, ISR exécute OSIntExit qui trouve que la tâche B a la plus haute priorité.
- Tâche A : passe de ISR à READY
- Tâche B : passe de PENDING à RUNNING

Event 7

- OSMutexPend() de B est executée mais cette fois M1 est à 0.
- Tâche B : passe de RUNNING à PENDING.
- La fonction système OS_PendListRemove() est exécutée et elle retire la Tâche B de OSPrioTbl (mise à 0). Puis B est mis dans la structure d'attente OS_PEND_OBJ de M1 avec l'appel à la fonction système OS_PendListInsertPrio(). L'ordonnanceur OSSched() est appelé et la prochaine tâche la plus propriétaire, la Tâche A, passe de READY à RUNNING.
- La tâche A hérite de la priorité de M1 qui est 3 (i.e. que temporairement on va créer une deuxième tâche de priorité 3 avec OSChangePrio(&Tache_A, 4, &err)).
- En terminant, l'ordonnanceur est appelé et il détermine que la tâche A comme la plus haute priorité.
- Tâche A : passe de READY à RUNNING

Event 8

- OSMutexPost() est exécutée et M1 est maintenant à 1.
- La fonction système OS_PendListRemove() est appelée et sélectionne dans OS_PEND_OBJ de M1 la tâche la plus prioritaire (pointeur sur HeadPtr) qui est B.
- On exécute alors la fonction système OS_PrioInsert() qui remet B actif dans OSPrioTbl[].
- B passe de PENDING à READY.
- La priorité de Tâche A passe de 3 à 4 (OSChangePrio(&Tache_A, 3, &err)).)
- En terminant, l'ordonnanceur est appelé et il détermine que la tâche B a la plus haute priorité.
- Tâche A : passe de RUNNING à READY
- Tâche B : passe de READY à RUNNING

b)

À l'évènement 3 on aurait hérité de la priorité 3, mais ça n'aurait rien changé car l'interruption 4 surviendrait quand même.

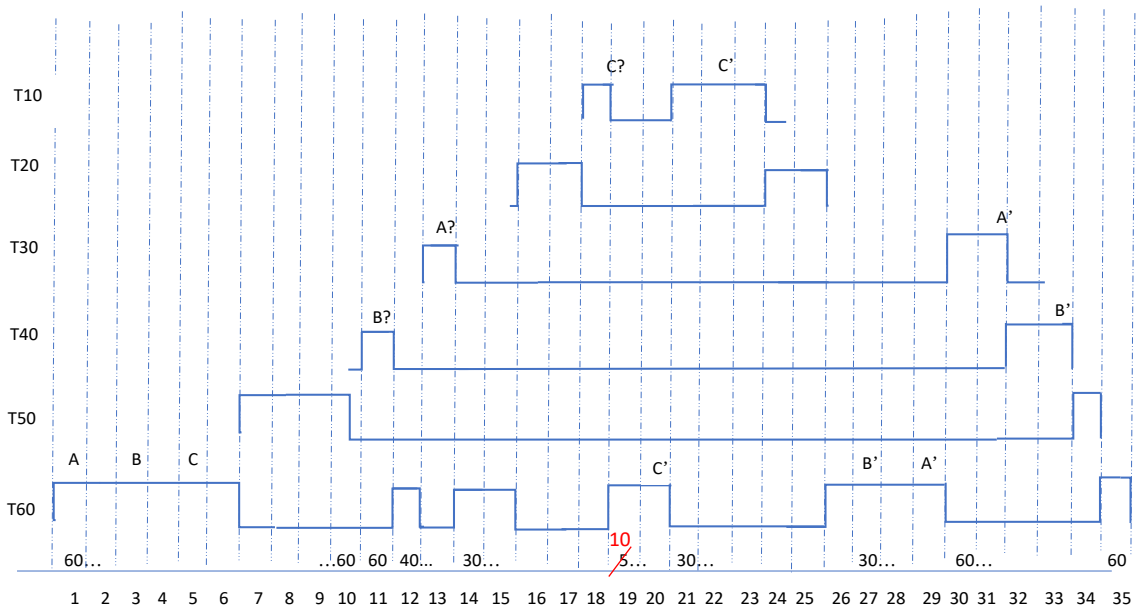
À l'évènement 6, on aurait sélectionné A comme tâche la plus prioritaire. Donc A se serait aller à OSMutexPost(M1) tout de suite.

À l'évènement 7 on aurait pas fait d'héritage car ça a été fait en 3.

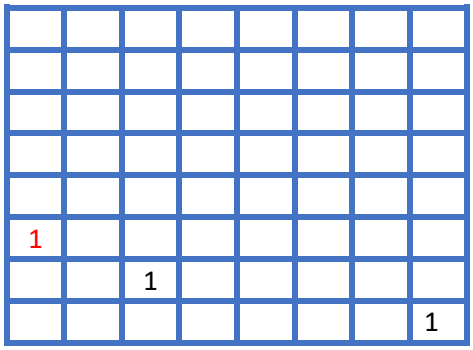
Bref, la comportement est le même, mais la séquence est quelque peu modifiée.

Q35

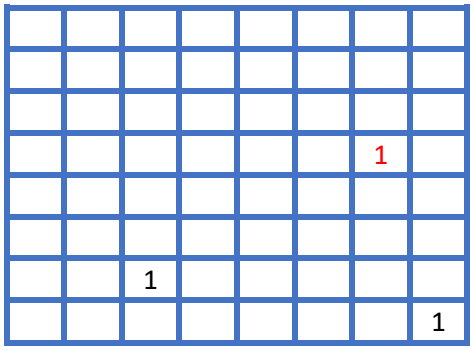
a)


T10
T20
T30
T40
T50
T60
C?
C'
A?
A'
B?
B'
A
B
C
C'
B'
A'
60…
…60
60
40…
30…
10
5…
30…
30…
60…
60
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

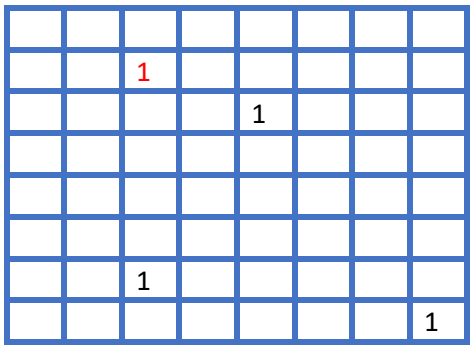
b)



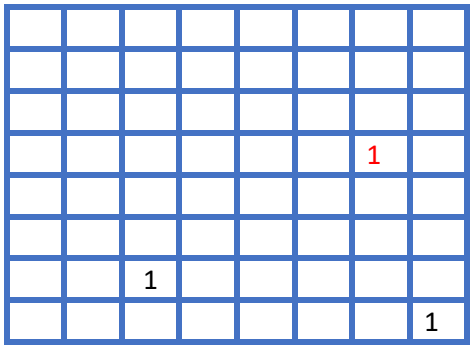
Tick 12 : La tâche T40 est en attente dans OS_PEND_OBJ du mutex B mais la T60 a passer à la priorité 40 (héritage de T40) via OSChangePrio(). De plus on dit que T40 et T50 sont bloquées (attention on dit bloquée par opposition à préemptée car ici T60 ne peut avoir de préemption sur T50).



Tick 14 : La tâche T30 est en attente dans OS_PEND_OBJ du mutex A mais la T60 a passer à la priorité 30 (héritage de T30) via OSChangePrio(). De plus T30, T40 et T50 sont dites bloquées.

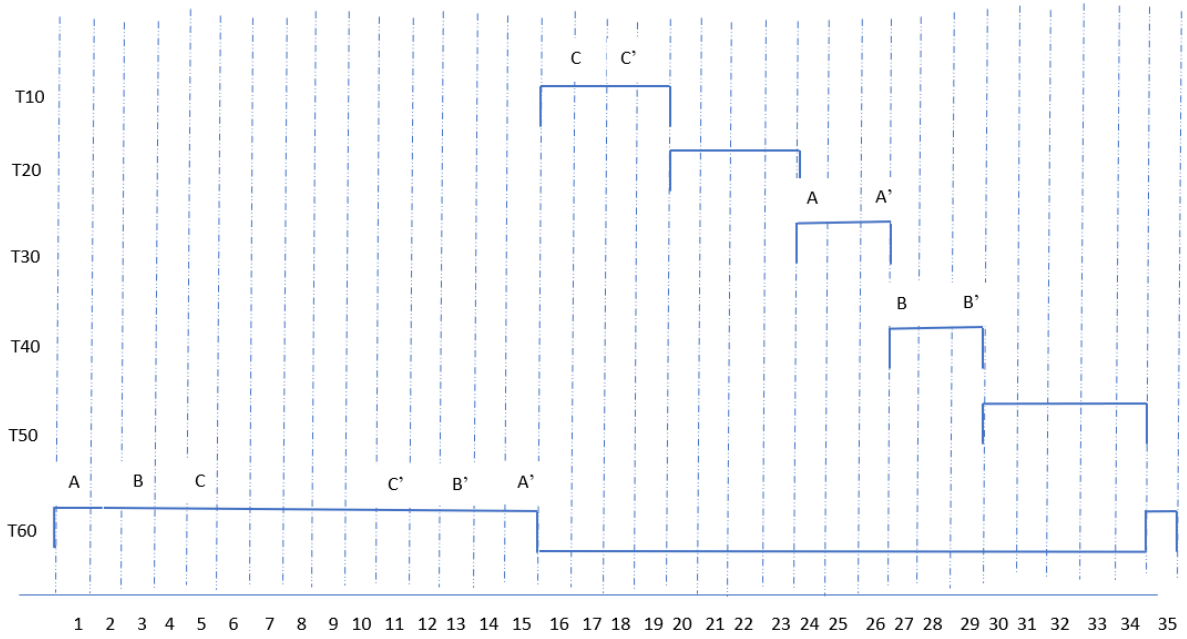


Tick 19 : La tâche T10 est en attente dans OS_PEND_OBJ du mutex C mais la T60 a passer à à la priorité 10 (héritage de T10) via OSChangePrio(). Ici on dit aussi que T10, T20, T30, T40 et T50 sont bloquées.



Tick 28 : La tâche T40 est en attente dans OS_PEND_OBJ du mutex B mais la T60 a passer à T40 via OSChangePrio(). De plus T30, T40 et T50 sont bloquées.

c)



T1 passe de 6 à 4 ticks.

Q36

Ceci a été expliqué dans le cours no 3 pp. 32 à 50. Voir aussi au besoin, pp. 146 à 167 du user manuel.

Q37

Ceci a été expliqué dans le cours no 5 pp. 4 à 9. Voir aussi au besoin, pp. 202 à 204 du user manuel.

Q38

E.g. lorsqu'on enlève désactive un bit de OSPrioTbl[] (passage de 1 à 0) suite à un passage de Running à Pending c'est parce qu'on insère dans la structure d'attente de OS_PEND_OBJ et vise versa quand quand on passe de Pending à Ready c'est parce qu'on retire de la structure d'attente de OS_PEND_OBJ. Des exemples sont illustrés dans le no 33 par exemple pour les events 2 et 5, respectivemment.

Q39

a) À l'instant t, les tâche 11, 45 et 63 sont actives. Les tâched T10 et T41 sont en attente du sémaphore S1.

b) Il faut ici expliquer que la fonction OSPrioHighest est appelé et va exécuter le 3 étapes de la p. 41 Cours no 3. En résumé, la ligne 0 (on va de 0 à 7) égal 0 donc on incrémente p_tbl. À la 1e ligne la fonction CPU_CntLeadZeos va trouver le 1er bit à 1 à la position 3 (en partant à 0 de la gauche), ce qui fait (8 * 1) + 3 = 11. OSCtxSw va donc mettre la tâche 11 sur le CPU si ce n'est pas déjà le cas.

c) Ici il faut décrire l'évènement comme au no 34 par exemple l'évènement 2 en l'appliquant au TCB de la tâche 11. Il faut aussi mentionner que T11 s'ajoute entre T10 et T45 déjà en attente (voir la figure de la page 38).

   Une fois la tâche 11 mis en attente on va appeler l'ordonnanceur : la fonction OSPrioHighest est appelé et va incrémenter p_tbl jusqu'à la ligne 5 en partant de 0. À la 6e ligne la fonction CPU_CntLeadZeos va trouver le 1er bit à 1 à la position 5 (en partant à 0 de la gauche), ce qui fait (8 * 5) + 5 = 45. OSCtxSw va donc mettre la tâche 45 sur le CPU en remplacement de 11.

d) Il faut décrire l'évènement comme au no 34 par exemple l'évènement 5 en l'appliquant au TCB de la tâche 10 (car on prend la tête de liste i.e. la tâche la plus prioritaire).

   Une fois la tâche 10 mis en attente on va appeler l'ordonnanceur : la fonction OSPrioHighest est appelé et va incrémenter p_tbl jusqu'à la ligne 1 en partant de 0. À la 2e ligne la fonction CPU_CntLeadZeos va trouver le 1er bit à 1 à la position 2 (en partant à 0 de la gauche), ce qui fait (8 * 1) + 2 = 10. OSCtxSw va donc mettre la tâche 10 sur le CPU en remplacement de 45.