POLYTECHNIQUE MONTRÉAL

Solution
examen intra

INF3610

Sigle du cours

Identification de l'étudiant(e)		
Nom :	**Prénom :**	
Signature :	**Matricule :**	**Groupe : 1**

Sigle et titre du cours	*Groupe*	*Trimestre*
INF3610 – Systèmes embarqués	**Tous**	**202101**
Professeur	***Local***	***Téléphone***
Guy Bois	**M-4115**	**5944**

Jour	*Date*	*Durée*	*Heures*
Lundi	**15 mars 2021**	**3h00** (incluant remise)	**8h45 à 11h30**

Documentation	*Calculatrice*	
☐ Aucune	☐ Aucune	**Les cellulaires, agendas électroniques ou téléavertisseurs sont interdits.**
☒ Toute	☐ Toutes	
☐ Voir directives particulières	☒ Non programmable (AEP)	

Important

Cet examen contient **5** questions sur un total de **x** pages **(excluant cette page)**

- La pondération de cet examen est de **30** **%**
- **Vous devrez être présent sur Zoom à partir de 8h40 avec la caméra allumé**
- Les questions sont dans un fichier .docx et un .pdf qui vous parviendra par e-mail à 8h45
- Vous devez me remettre le .docx ou le .pdf rempli ou encore un .txt rempli
- Pour les tableaux à compléter, vous pouvez compléter directement avec le logiciel de dessin ou compléter à la main et retourner un scan ou photo jpeg.
- À partir de 11h30, vous aurez 15 mins pour transférer les documents sur Moodle (procédure similaire à celle d'une remise de devoir)

L'étudiant doit honorer l'engagement pris lors de la signature du code de conduit

Question 1 (3.5 points) En vrac : vrai ou faux avec <u>justification</u>

1. (.5 pt) La désactivation des interruptions augmente la latence d'interruption mais elle garantit toujours la protection d'une ressource partagée.

   Faux. C'est vrai que la désactivation des interruptions augmente la latence d'interruption, mais elle ne garantit la protection d'une ressource partagée qu'entre 2 ISRs ou 1 ISR et 1 tâche. Plus précisément, elle ne garantit la protection d'une ressource partagée entre 2 tâches.

2. (1 pt) La tâche la plus prioritaire d'une application en uC/OS-III ne peut jamais être préemptée par une autre tâche de la même application, alors que la tâche la moins prioritaire d'une application en uC/OS-III ne peut jamais être bloquée (suite à une inversion de priorité) par une autre tâche de la même application.

   Vrai. Une tâche qui est la plus prioritaire ne peut être préempté par une tâche plus prioritaire de la même application (par définition).
   D'autre part, la tâche la moins prioritaire ne peut être bloquée par une autre tâche de la même application avec laquelle elle partagerait un mutex puisque cette dernière se devrait d'être moins prioritaire (condition de blocage conduisant à l'inversion de prioprité). Encore une fois par définition ça ne peut pas arriver.

3. (1 pt) Nous avons vu en classe qu'il existe au moins 3 manières d'ordonnancer une tâche périodique sans retard relatif.

   Vrai :

   1. *OSTimeDly avec option OS_OPT_TIME_PERIODIC*
   2. *Une tâche timer comme dans le no 32*
   3. *Un watchdog timer (OS_TMR) avec option OS_OPT_TMR_PERIODIC*

4. (1 pt) Lors d'un appel de demande de délai avec *OSTimeDlyHMSM()*, il existe 2 sources contribuant à l'incertitude du déclenchement d'une tâche : 1) une incertitude de 1 tick qui dépend de l'instant où on demande le délai au cours de la période courante du *timer* et 2) une incertitude de 1 tick qui dépend de l'instant où on reprend l'exécution au cours de la période courante.

   Faux Il existe une autre source est d'avoir une durée de délai qui n'est pas multiple du tick d'horloge (intervalle entre deux ticks). uC/OS-III va alors arrondir. Voir no 31 des exercices à titre d'exemple.

Question 2 (4.5 points) Ordonnancement, héritage de priorité et ICPP

a) Soit 6 tâches T1, T2, T3, T4, T5 et T6 avec uC/OS-III ayant respectivement les priorités 2, 3, 6, 7, 8 et 9.

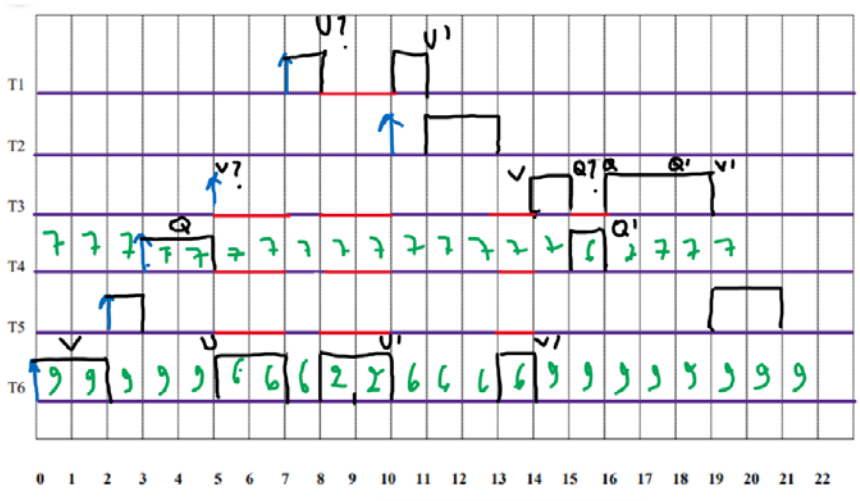
Tâche (T_i)	Numéro de la période de départ	Nombre de ticks d'exécution du thread	Séquence d'exécution (les parenthèses indiquent que la tâche possède deux mutex durant un même tick)
T1	7	2	EU
T2	10	2	EE
T3	5	4	V(VQ)(VQ)V
T4	3	3	EQQ
T5	2	3	EEE
T6	0	7	EV(VU)(VU)(VU)(VU)V

Table 2.1

(2 pts) Complétez la trace d'exécution de la figure 2.1 (page 5) en considérant le protocole supporté par uC/OS-III.

Vous devez :

- Pour chaque coup d'horloge de T4 et de T6, indiquez la priorité.
- Mettre en rouge la portion de l'exécution qui subit un blocage dû à l'inversion de priorité.
- Utilisé la convention vue en classe i.e., V? indique un *pend bloquant* et une mise en attente, V indique le mutex est obtenu (après attente ou sans attente) et V' indique un *post*.



b) Soit l'ensemble des tâches suivant exécuté sur un processeur embarqué:

Tâche	**Période (T_i)**	**Nombre de périodes d'exécution (C_i)**
T1	7	2
T2	11	3
T3	19	2
T4	21	3

Table 2.2

(1 pt) Proposez une assignation des priorités selon le Rate Monotonc Assignment (RMA) puis effectuez le test de *Liu and Layland*. Que peut-on en conclure ?

2/7 + 3/11 + 2/19 + 3/21 = 0,8065 = 80.7%

Or pour N=4 on a 75.7% (voir le tableau suivant tiré du cours no 2, p. 8):

N	Utilization Bound
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

On ne peut donc rien dire.

(1.5 pts) On désire approximer les valeurs de blocage B_i de la table 2.2 qui proviennent de l'utilisation de deux mutex A et B tel qu'indiqué à la table 2.3.

Tâche (t_i)	**Mutex utilisés (nombre de périodes)**
T1	A(1) B(1)
T2	--
T3	A(2) B(1)
T4	A(3)B(2)

Table 2.3

En considérant le protocole ICPP, calculez le blocage Bi de chaque tâche i.e. B1, B2, B3 et B4.

Pour B1 :

T1 peut être bloquée si T3 ou T4 (un ou l'autre) accède à A avant T1. Si c'est T3 alors T1 bloquera pour 2 ticks et si c'est T4 alors T1 bloquera de 3 ticks (on prend donc le ***maximum*** *entre 2 et* ***3****).*

T1 peut également être bloquée si T3 ou T4 (un ou l'autre) accède à B avant T1. Si c'est T3 alors T1 bloquera pour 1 tick et si c'est T4 alors T1 bloquera de 2 ticks (on prend donc le ***maximum*** *entre 1 et* ***2****).*

Donc si T3 ou T4 démarre avant T1 et que T3 ou T4 rentre dans A ou B (un ou l'autre) T3 ou T4 hérite de la priorité de T1, et T1 attendra au maximum 3 ticks (max(3,2)).

Pour B2 :

T2 peut être bloquée si T3 ou T4 (un ou l'autre) accède à A avant T1. Si c'est T3 alors T2 bloquera pour 2 ticks et si c'est T4 alors T2 bloquera de 3 ticks (on prend donc le ***maximum*** *entre 2 et* ***3****).*

T2 peut également être bloquée si T3 ou T4 (un ou l'autre) accède à B avant T1. Si c'est T3 alors T2 bloquera pour 1 tick et si c'est T4 alors T2 bloquera de 2 ticks (on prend donc le ***maximum*** *entre 1 et* ***2****).*

Donc si T3 ou T4 démarre avant T1 et que T3 ou T4 rentre dans A ou B (un ou l'autre) T3 ou T4 hérite de la priorité de T1, et T2 attendra au maximum 3 ticks (max (3,2)).

Pour B3 :

T3 peut être bloquée si T4 accède à A avant T1. T3 bloquera pour ***3*** *ticks.*

T3 peut également être bloquée si T4 accède à B avant T1. T3 bloquera de ***2*** *ticks.*

Donc si T4 démarre avant T1 et que T4 rentre dans A ou B (un ou l'autre), T4 hérite de la priorité de T1, et T3 attendra au maximum 3 ticks (max (3,2)).

Pour B4 :

La tâche la moins prioritaire ne peut être bloquée par une autre tâche de la même application avec laquelle elle partagerait un mutex puisque cette dernière se devrait d'être moins prioritaire (condition de blocage conduisant à l'inversion de prioprité). Encore une fois par définition ça ne peut pas arriver. Donc B4=0.

Question 3 (3.5 points) Analyse de code avec uC/OS-III

(1.5 pts) Soit le code de la figure 3.1, une version simplifiée de *OSSemPost()*. Expliquez le rôle de chaque bloc (1 à 3). Considérez que *CPU_CRITICAL_ENTER()* et *CPU_CRITICAL_EXIT()* désactive et réactive les interruptions respectivement.

```
OS_SEM_CTR  OSSemPost (OS_SEM  *p_sem,
                       OS_OPT   opt,
                       OS_ERR  *p_err)
{
    OS_SEM_CTR     ctr;
    OS_PEND_LIST  *p_pend_list;
    OS_TCB        *p_tcb;
    OS_TCB        *p_tcb_next;
    CPU_TS         ts;
    CPU_SR_ALLOC();

    .
    .
    .

    CPU_CRITICAL_ENTER();
    p_pend_list = &p_sem->PendList;
    if (p_pend_list->HeadPtr == (OS_TCB *)0) {            /*      Bloc 1        */

        p_sem->Ctr++;
        ctr       = p_sem->Ctr;

        CPU_CRITICAL_EXIT();
       *p_err     = OS_ERR_NONE;
        return (ctr);
    }

    p_tcb = p_pend_list->HeadPtr;                         /*      Bloc 2        */
    while (p_tcb != (OS_TCB *)0) {
        p_tcb_next = p_tcb->PendNextPtr;
        OS_Post((OS_PEND_OBJ *)((void *)p_sem),
                p_tcb,
                (void *)0,
                0u,
                ts);
        if ((opt & OS_OPT_POST_ALL) == 0u) {
            break;
        }
        p_tcb = p_tcb_next;
    }
    CPU_CRITICAL_EXIT();
    if ((opt & OS_OPT_POST_NO_SCHED) == 0u) {             /*      Bloc 3        */
        OSSched();
    }
   *p_err = OS_ERR_NONE;

    return (0u);
```

.5 pt par bloc

Bloc 1 :

On désactive les interruptions.
*On regarde si la liste d'attente est vide du sémaphore est vide (p_pend_list -> HeadPtr == (OSTCB *) 0)*
Si oui :
On augmente le compteur de la sémaphore.
On désactive les interruuptions.
On met indique qu'il n'y pas d'erreur et on retourne la nouvelle valeur du compteur à l'appelant.

Bloc 2 :

Il y a ici 2 possibilités :

1) il y avait un paramètre à l'appel de OSSemPost qui indiquait qu'on voulait que le post s'adresse à toutes les tâches en attente (OS_OPT_POST_ALL). Si c'est le cas, on passe à travers la liste d'attente et on enlève chaque tâche (i.e. son tcb) et on la remet prête dans OSPrioTbl.

2) Si on a pas le paramètre OS_OPT_POST_ALL, on remet uniquement la tâche en tête de liste d'attente comme prête dans OSPriotbl. (i.e. la + prioritaire).

Bloc 3 :

Si on n'a pas indiqué l'option OS_OPT_POST_NO_SCHED, L'ordonnanceur est appelé via OSSched.

(2 pts) Soit le code de la figure 3.2 de *OSIntExit()* porté sous ARM A9 (Cora Z7) Expliquez le rôle de chaque bloc (1 à 4). Considérez que *CPU_INT_DIS()* et *CPU_INT_EN()* désactive et réactive les interruptions respectivement.

```
1  void  OSIntExit (void)
2  {
3      CPU_INT_DIS();
4      OSIntNestingCtr--;
5      if (OSIntNestingCtr > 0u) {                          /* Bloc 1 */
6          OS_TRACE_ISR_EXIT();
7          CPU_INT_EN();
8          return;
9      }
10     if (OSSchedLockNestingCtr > 0u) {                    /* Bloc 2 */
11         CPU_INT_EN();
12         return;
13     }
14     OSPrioHighRdy   = OS_PrioGetHighest();               /* Bloc 3  Voir aussi plus bas */
15     OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;
16     if (OSTCBHighRdyPtr == OSTCBCurPtr) {
17         CPU_INT_EN();
18         return;
19     }
20
21     OSTaskCtxSwCtr++;                                    /* Bloc 4  */
22
23     CPU_INT_EN();
24
25 }
26
27 OS_PRIO  OS_PrioGetHighest (void)                       /* Bloc 3 */
28 {
29                 /* Optimize for    2x the word size nbr of priorities   */
30     if (OSPrioTbl[0] == 0u) {
31         return ((OS_PRIO)((OS_PRIO)CPU_CntLeadZeros(OSPrioTbl[1]) + (CPU_CFG_DATA_SIZE * 8u)));
32     } else {
33         return ((OS_PRIO)((OS_PRIO)CPU_CntLeadZeros(OSPrioTbl[0])));
34     }
35 }
```

.5 pt par bloc

Bloc 1 :

On désactive les interruptions.

On décrémente le compteur d'interruptions imbriquées. S'il y'a toujours des interruptions on appelle une trace (pour les logs), on réactive les interruptions puis on retourne traiter les interruptions.

Bloc 2 : Annulé (mais j'ai donné les points quand même à ceux qui l'avaient bons)

Si l'ordonnanceur est verrouillé on réactive les interruptions et on retourne (mode non préemptif).

Bloc 3 :

On cherche la tâche la plus prioritaire (voir ci-après) et on accède à son TCB. Si la tâche désigné est celle qui a été interompue, on a pas besoin de faire de changement de contexte donc on réactive les interruptions et on retourne.

Pour la recherche de la tâche la plus prioritaire (cas de 2 mots successifs de 32 bits) :

Si la première ligne est nulle, on cherche le 1er bit à 1ère de la deuxième ligne (via CPU_CntLeadZeros), sinon celui de la 1ère ligne (via CPU_CntLeadZeros).

Bloc 4

La prochaine étape après étant le changement de contexte, on incrémente ce nombre avant.

Question 4 (5 points) Gestion des tâches et des évènements

Soit un système sous uC/OS-III qui démarre (*OSStart()*). Ce système est composé d'un certains nombres de tâches qu'on identifie par Ti où i est le numéro de priorité (n.b. les tâches ont des priorités distinctes).

Dès le départ, on assiste aux 3 évènements suivants (début du tick 0) :

i. La tâche T10 (la seule tâche prête au départ), demande le mutex avec *OSMutexPend(&M1, 0, OS_OPT_PEND_BLOCKING, &ts, &err)* et l'obtient.
ii. Immédiatement après, T10 fait appel à *OSTimeDly(2, OS_OPT_TIME_DLY, &err).*
iii. Immédiatement après, la tâche T8 devient prête et demande le mutex avec *OSMutexPend(&M1, 0, OS_OPT_PEND_BLOCKING, &ts, &err).*

L'état du système après ces 3 évènements est illustré à la figure 4.

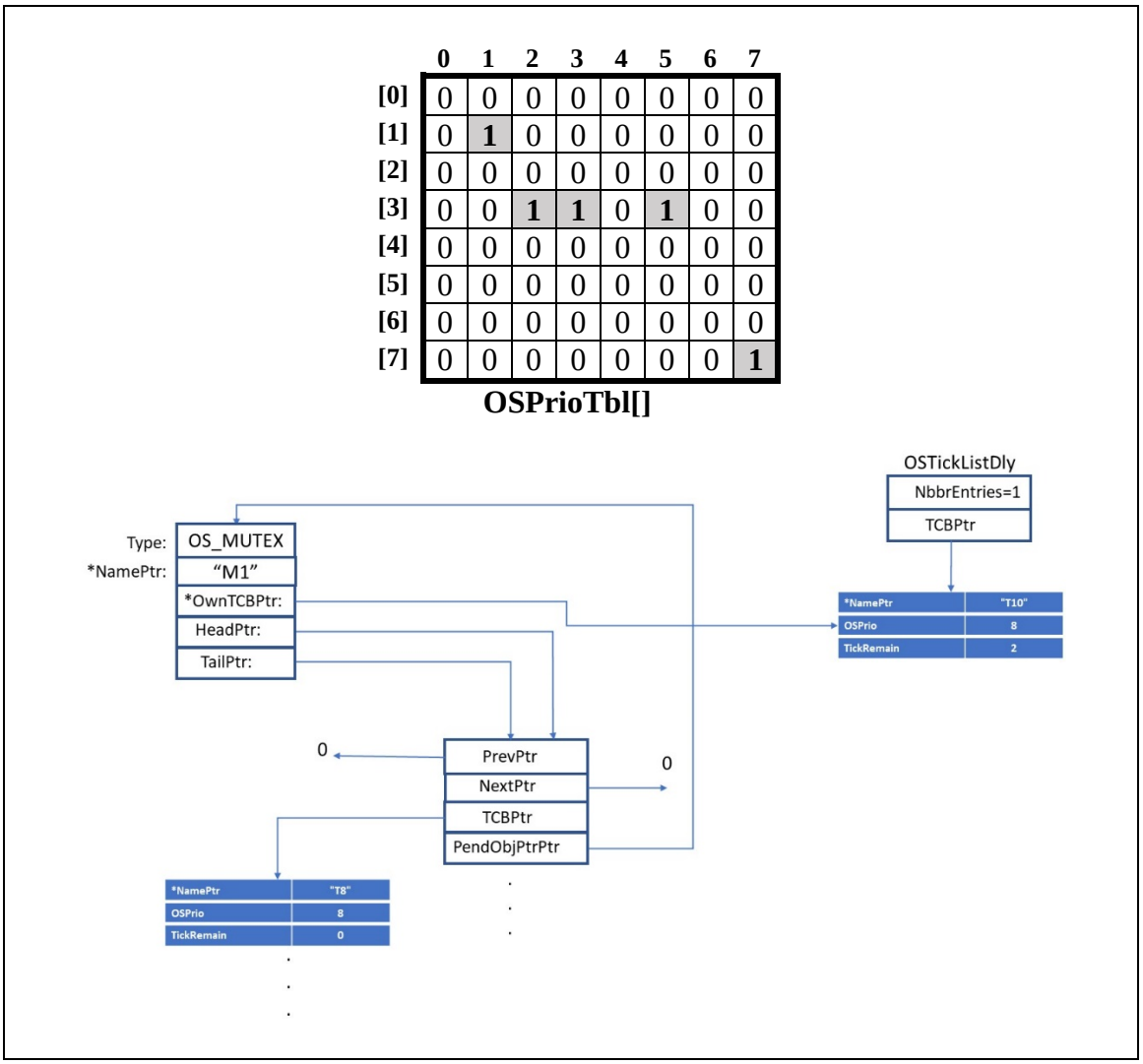


Figure 4.1

a) (.5 pt) Décrivez l'état du système à la Figure 4.1, c'est-à-dire les différentes tâches en cours (sous forme Ti) et leur état.
Hint : Il existe une fonction système OS_PendListChangePrio() qui permet de changer la priorité d'une tâche pendant qu'elle est en attente d'un délai.

T8 est en attente du mutex
T10 aussi mais également en attente d'un délai (T10 a hérité de la priorité de T8)
T9, T26, T27 et T29 sont prêtes
T63 est la tâche IDLE.

b) (.5 pt) Déterminez la tâche la plus prioritaire juste après l'évènement iii) ci-haut. Expliquez brièvement comment uC/OS-III détermine la prochaine tâche à être exécuter.

*La fonction OSPrioGetHighest() passe les lignes de 0 à 7. Pour chaque ligne, elle regarde si la ligne 0 = 0. Si oui, elle incrémente donc l'indice de 1 et passe à la ligne suivante. Sinon (ligne différente de 0). La fonction CPU_CntLeadZeros va trouver le 1er bit à 1 en partant de la colonne 0. Par conséquent, dans notre exemple, on aura une ligne non nul à la ligne 1 sur laquelle CPU_CntLeadZeros retournera la colonne 1. Ce qui fait (8*1)+1 = 9. La tâche la plus prioritaire est donc la tâche T9.*

c) (2 pts) Nous sommes toujours au tick no 0 et on suppose ensuite que la tâche trouvé en b) fait appel à *OSMutexPend(&M1, 2, OS_OPT_PEND_BLOCKING, &ts, &err)*. Ici vous devez indiquer dans un ordre chronologique ce qui se passe lors de cet appel i.e. :

- S'il y a lieu, les changements des états des tâches.
- Les détails internes de ce qui se passe dans le noyau lorsque que les fonctions *pend* et *post* sont appelées c'est-à-dire décrire ce qui se passe au niveau des structures et des appels système. Ici décrivez textuellement les changements qui devraient être apportés à la figure 4.
- S'il y a lieu, les changements de contexte (suite à l'appel de l'ordonnanceur), en indiquant la prochaine tâche à être exécuter.

On regarde si le mutex est libre, ce qui n'est pas le cas.

N.B. Le owner du mutex (T10 qui a une priorité de 8) est + prioritaire que T9, donc T9 n'hérite pas de la priorité de T10 !

Comme il y'a timeout on insère la tâche T9 dans la tick list avec OS_TickListInsert() et aussi on met T9 dans dans la liste d'attente OS_PEND_OBJ du mutex juste après T8 car elle est moins prioritaire (c'est une liste triée par ordre de priorité).

On retire T9 de la ReadyList en appelant OS_PrioRemove(), met 0 dans OSPrioTbl[1][1].

La tâche T9 passe donc de RUNNING à PENDING

On appelle ensuite OSSched(), qui appelle OS_PrioGetHighest().

La tâche la + prioritaire est T26. T26 passe dès lors de READY -> RUNNING

On appelle ensuite OSCtxSw(), il y'a changement de contexte et T26 devient la tâche courante en exécution.

d) (2 pts) La tâche la plus prioritaire trouvée en c) s'exécute jusqu'au tout début du tick no 2 où l'ISR de la minuterie s'exécute. Indiquez ce qui se passe au niveau de ce ISR no 2 de la minuterie ainsi que pour la fonction *OSIntExit()* qui termine ce ISR. À nouveau indiquez :

- S'il y a lieu, les changements des états des tâches.
- Les détails internes de ce qui se passe dans le noyau lorsque que les fonctions *pend* et *post* sont appelées c'est-à-dire décrire ce qui se passe au niveau des structures et des appels système. Ici décrivez textuellement les changements qui devraient être apportés à la figure 4.
- S'il y a lieu, les changements de contexte (suite à l'appel de l'ordonnanceur), en indiquant la prochaine tâche à être exécuter (ici retour de la tâche courante ou une nouvelle tâche).

L'ISR appelle OSTimeTick()

On regarde la OSTickListDly, et on enlève les deux tâches de cette liste T10 et T9 car leur TickRemain est rendu à 0.

T10 est directement mise dans la ReadyList et on appelle OS_PrioInsert(), met 1 dans OSPrioTbl[1][0], elle devient READY avec une priorité de 8

T9 attendait un timeout, elle est directement mise dans la ReadyList et on appelle OS_PrioInsert(), met 1 dans OSPrioTbl[1][1], elle devient READY avec sa priorité de 9.

Le traitement de OSTimeTick se termine.

On appelle OSIntExit() :

Désactive les interruptions, puis décrémente OSIntNestingCtr

Puisque OSIntNesting arrive à 0 ça veut dire qu'il n'y plus d'interruption,

On appelle OS_PrioGetHighest(). La tâche la + prioritaire est T10 (prio=8)

Tâche 10 passe de PENDING -> RUNNING

OSIntCtxSw() est appelée, On prend directement le contexte de T10 et on le met sur le CPU et elle prend le relai !

Question 5 (3.5 points) Laboratoire no 1

a) (1 pt) Expliquez le fonctionnement de la tâche *OS_StatTask*. Imaginez que vous l'expliquez à un collègue qui désire l'utiliser. Au besoin, utilisez le laboratoire no 1 pour illustrer votre propos.

La tâche OS_StatTask est une tâche périodique qui s'éxecute à la fréquence suivante OS_CFG_STAT_TASK_RATE_HZ. Pour l'activer, on doit mettre OS_CFG_STAT_TASK_EN à 1 et démarrer OSStatReset();

Elle permet d'avoir des statistiques telles que le pourcentage d'utilisation du CPU de chaque tâche, l'utilisation du FIFO depuis le dernier la dernière exécution ou encore le maximum d'utilisation des FIFOs depuis OSStatReset(), etc.

b) (.5 pt) J'ai dit en classe que le profilage fait par uC/OS-III est intrusif. Expliquez ce que je voulais dire exactement par cette affirmation ?

Intrusif *veut dire que la tâche OS_StatTask demande du temps CPU, ce qui peut venir fausser le temps d'exécution total d'un système utilisant OS_StatTask.*

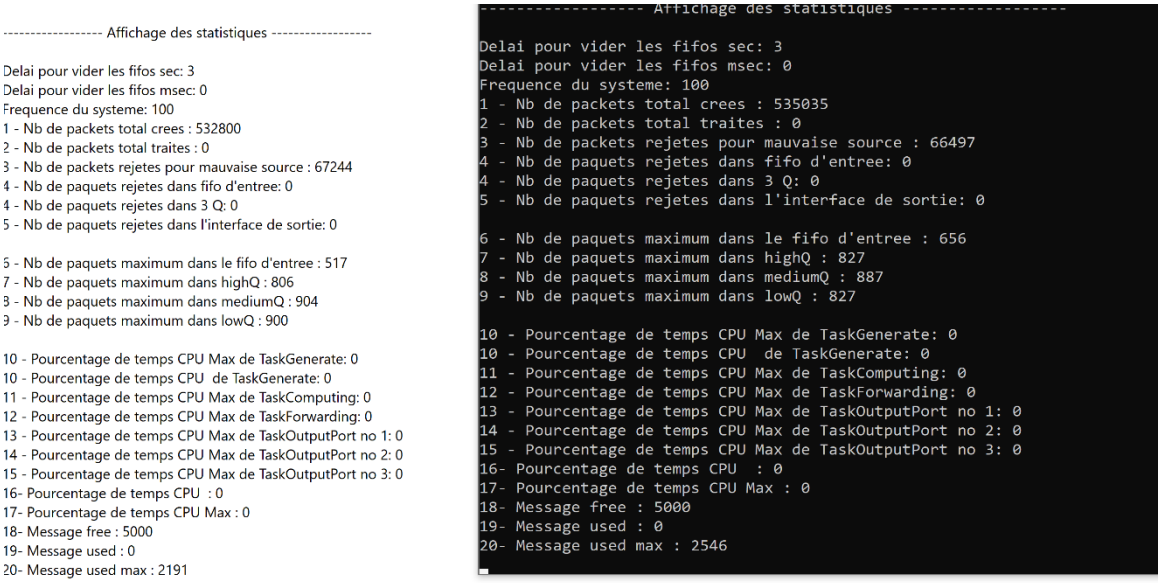
c) La figure 5.1 (page suivante) illustre des résultats que j'ai effectué dans la partie 1 et 2 du laboratoire no 1 (délai entre 2 rafales de 0 à 255 paquets).

c.1) (1 pt) En vous fiant aux différentes métriques imprimées à la figure 5.1, pourquoi ai-je décidé que ces délais de 250ms (~~SDK~~1000Hz) et de 3 sec (~~VS~~100Hz) sont acceptables ?

Simplement parce qu'aucune fifo n'a rejeté de paquets que leur utilisation est autour de 50%.

c.2) (1 pt) Pourquoi un délai de 250ms (SDK) vs 3 sec (VS) ? D'où provient cette différence ? Expliquez ***Annulé***

J'ai quand même donné une fraction de point à ceux qui m'ont dit que les opérations s'exécutaient plus vite à 1000Hz, mais la principale réponse est que la tâche TaskComputing s'exécute beaucoup plus rapidement puisque l'attente active est dix fois plus courte avec 1000Hz.



```
---------- Affichage des statistiques ----------

Delai pour vider les fifos sec: 3
Delai pour vider les fifos msec: 0
Frequence du systeme: 100
1 - Nb de packets total crees : 532800
2 - Nb de packets total traites : 0
3 - Nb de packets rejetes pour mauvaise source : 67244
4 - Nb de paquets rejetes dans fifo d'entree: 0
4 - Nb de paquets rejetes dans 3 Q: 0
5 - Nb de paquets rejetes dans l'interface de sortie: 0

6 - Nb de paquets maximum dans le fifo d'entree : 517
7 - Nb de paquets maximum dans highQ : 806
8 - Nb de paquets maximum dans mediumQ : 904
9 - Nb de paquets maximum dans lowQ : 900

10 - Pourcentage de temps CPU Max de TaskGenerate: 0
10 - Pourcentage de temps CPU  de TaskGenerate: 0
11 - Pourcentage de temps CPU Max de TaskComputing: 0
12 - Pourcentage de temps CPU Max de TaskForwarding: 0
13 - Pourcentage de temps CPU Max de TaskOutputPort no 1: 0
14 - Pourcentage de temps CPU Max de TaskOutputPort no 2: 0
15 - Pourcentage de temps CPU Max de TaskOutputPort no 3: 0
16- Pourcentage de temps CPU  : 0
17- Pourcentage de temps CPU Max : 0
18- Message free : 5000
19- Message used : 0
20- Message used max : 2191
```

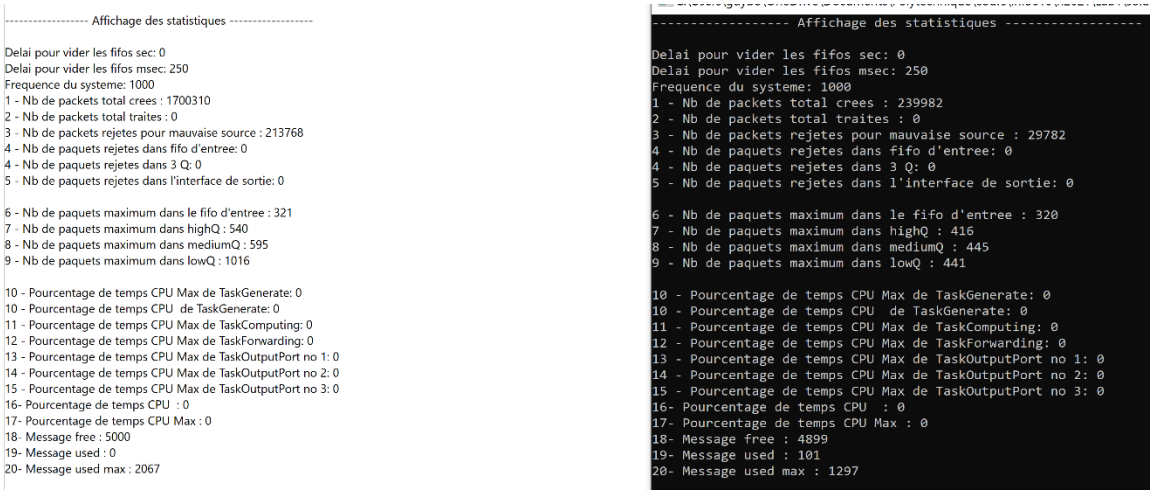
```
------------------ Affichage des statistiques ------------------

Delai pour vider les fifos sec: 3
Delai pour vider les fifos msec: 0
Frequence du systeme: 100
1 - Nb de packets total crees : 535035
2 - Nb de packets total traites : 0
3 - Nb de packets rejetes pour mauvaise source : 66497
4 - Nb de paquets rejetes dans fifo d'entree: 0
4 - Nb de paquets rejetes dans 3 Q: 0
5 - Nb de paquets rejetes dans l'interface de sortie: 0

6 - Nb de paquets maximum dans le fifo d'entree : 656
7 - Nb de paquets maximum dans highQ : 827
8 - Nb de paquets maximum dans mediumQ : 887
9 - Nb de paquets maximum dans lowQ : 827

10 - Pourcentage de temps CPU Max de TaskGenerate: 0
10 - Pourcentage de temps CPU  de TaskGenerate: 0
11 - Pourcentage de temps CPU Max de TaskComputing: 0
12 - Pourcentage de temps CPU Max de TaskForwarding: 0
13 - Pourcentage de temps CPU Max de TaskOutputPort no 1: 0
14 - Pourcentage de temps CPU Max de TaskOutputPort no 2: 0
15 - Pourcentage de temps CPU Max de TaskOutputPort no 3: 0
16- Pourcentage de temps CPU  : 0
17- Pourcentage de temps CPU Max : 0
18- Message free : 5000
19- Message used : 0
20- Message used max : 2546
```

a) À 100Hz (à gauche SDK et à droite VS)



```
---------- Affichage des statistiques ----------

Delai pour vider les fifos sec: 0
Delai pour vider les fifos msec: 250
Frequence du systeme: 1000
1 - Nb de packets total crees : 1700310
2 - Nb de packets total traites : 0
3 - Nb de packets rejetes pour mauvaise source : 213768
4 - Nb de paquets rejetes dans fifo d'entree: 0
4 - Nb de paquets rejetes dans 3 Q: 0
5 - Nb de paquets rejetes dans l'interface de sortie: 0

6 - Nb de paquets maximum dans le fifo d'entree : 321
7 - Nb de paquets maximum dans highQ : 540
8 - Nb de paquets maximum dans mediumQ : 595
9 - Nb de paquets maximum dans lowQ : 1016

10 - Pourcentage de temps CPU Max de TaskGenerate: 0
10 - Pourcentage de temps CPU  de TaskGenerate: 0
11 - Pourcentage de temps CPU Max de TaskComputing: 0
12 - Pourcentage de temps CPU Max de TaskForwarding: 0
13 - Pourcentage de temps CPU Max de TaskOutputPort no 1: 0
14 - Pourcentage de temps CPU Max de TaskOutputPort no 2: 0
15 - Pourcentage de temps CPU Max de TaskOutputPort no 3: 0
16- Pourcentage de temps CPU  : 0
17- Pourcentage de temps CPU Max : 0
18- Message free : 5000
19- Message used : 0
20- Message used max : 2067
```

```
------------------ Affichage des statistiques ------------------

Delai pour vider les fifos sec: 0
Delai pour vider les fifos msec: 250
Frequence du systeme: 1000
1 - Nb de packets total crees : 239982
2 - Nb de packets total traites : 0
3 - Nb de packets rejetes pour mauvaise source : 29782
4 - Nb de paquets rejetes dans fifo d'entree: 0
4 - Nb de paquets rejetes dans 3 Q: 0
5 - Nb de paquets rejetes dans l'interface de sortie: 0

6 - Nb de paquets maximum dans le fifo d'entree : 320
7 - Nb de paquets maximum dans highQ : 416
8 - Nb de paquets maximum dans mediumQ : 445
9 - Nb de paquets maximum dans lowQ : 441

10 - Pourcentage de temps CPU Max de TaskGenerate: 0
10 - Pourcentage de temps CPU  de TaskGenerate: 0
11 - Pourcentage de temps CPU Max de TaskComputing: 0
12 - Pourcentage de temps CPU Max de TaskForwarding: 0
13 - Pourcentage de temps CPU Max de TaskOutputPort no 1: 0
14 - Pourcentage de temps CPU Max de TaskOutputPort no 2: 0
15 - Pourcentage de temps CPU Max de TaskOutputPort no 3: 0
16- Pourcentage de temps CPU  : 0
17- Pourcentage de temps CPU Max : 0
18- Message free : 4899
19- Message used : 101
20- Message used max : 1297
```

a) À 1000Hz (à gauche SDK et à droite VS)