POLYTECHNIQUE MONTRÉAL

examen intra
(Solution)

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**	**202103**
Professeur	***Local***	***Téléphone***
Guy Bois	**M-4115**	**5944**

Jour	*Date*	*Durée*	*Heures*
Lundi	**14 mars 2022**	**2h30**	**8h50 à 11h20**

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 **12** pages **(excluant cette page)**
De plus un code de 4 pages à compléter pour le no 3 vous sera remis. N'oubliez pas d'y mettre votre nom et matricule.

- La pondération de cet examen est de **30** **%**
- Répondez directement sur le questionnaire pour le no 2 et sur les 4 pages de code pour le no 3. Pour le reste répondre dans le cahier de réponse.
- Remettre le cahier de réponse, le questionnaire et les 4 pages de code.
- Ne pas écrire en rouge
-

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>

a) (.5 pt) Lorsqu'une ressource est partagée entre deux tâches uC/OS-III, un mécanisme possible, mais inefficace, est de désactiver l'ordonnanceur avant chaque accès à la ressource.

Vrai, c'est possible, mais désactiver l'ordonnanceur fait en sorte que le système n'est plus préemptif car une tâche arrêtera son exécution seulement lorsqu'elle bloque elle-même. Le mécanisme le plus efficace est le mutex.

b) (.5 pt) Le blocage *Bi* d'une tâche *Ti* lors de l'utilisation d'un mécanisme d'héritage de priorité ou de ICPP est toujours 0 lorsque *Ti* a la priorité la plus faible du système.

Vrai. Supposons le contraire. Ça veut dire Bi > 0. Par définition du blocage ça veut dire qu'une tâche moins prioritaire à Ti (nommons la Tj) partage une section critique avec Ti et bloque Ti. Or Tj ne peut exister car Ti est la moins prioritaire.

c) (.5 pt) En uC/OS-III, on peut utiliser une queue de messages (Q) de profondeur 1 et ses fonctions pour créer un rendez-vous unilatéral.

Vrai. L'analogie (avec le sémaphore) est la suivane :
1) au lieu de créer un sémaphore on crée une queue de profondeur 1,
2) au lieu de faire OSSemPend(&Sem, 0, OS_OPT_PEND_NON_BLOCKING, &ts, &err), celui qui attend le rendez-vous fera un OSQPend(&Q, 0, OS_OPT_PEND_NON_BLOCKING, &msg_size, &ts, &err). (Le fifo étant vide la tâche bloquera.)
3) au lieu d'utiliser OSSemPost(&Sem, OS_OPT_POST_1, &err) la tâche qui fait (signal) le rendez-vous utilisera OSQPost(&Q, message, sizeof(messageQ),OS_OPT_POST_FIFO, &err) (ce qui fera débloquer la tâche en 2).

d) (.5 pt) Dans le code suivant de la figure 1.1, la fonction *OSSemPend(&**Irq1**, 0, OS_OPT_PEND_NON_BLOCKING, &ts, &err)* permet d'avoir l'équivalent d'un sémaphore binaire dont le rôle est d'empêcher *TaskDrivFifo* de lire plusieurs fois une même valeur.

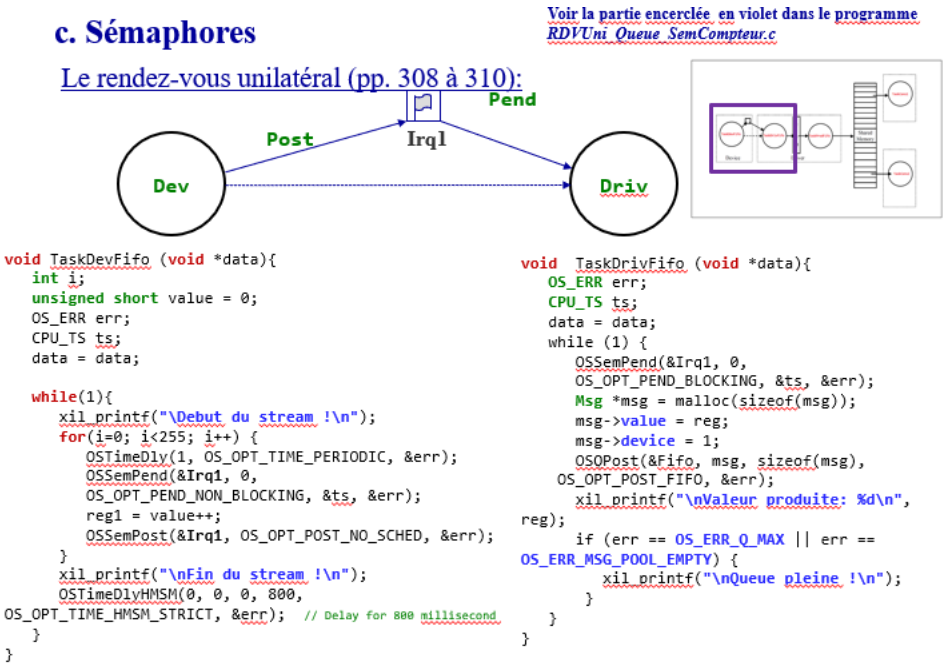


Figure 1.1 pour la question 1c.

Vrai. Ça été expliqué en classe. OSSemPend(&Irq1, 0, OS_OPT_PEND_NON_BLOCKING, &ts, &err) étant non bloquant à chaque fois qu'on fait un tour de boucle du for de TaskDevfifo, on décrémente le compteur de Irq1. Au maximum Irq1 sera à 1 (après OSemPost(&Irq1, OS_OPT_NO_SCHED, &err), TaskDrivFifo ne peut donc lire qu'une seule fois la même valeur.

e) (.5 pt) Une fois le délai écoulé d'un *OSTimeDly* la tâche appelante est re-activer par une interruption logicielle.

Faux. C'est une interruption matérielle (OSTimeTick) qui va remettre la tâche prête à rouler (ou en exécution). Comme je n'avais pas spécifier le terme re-activer (qui est pour moi équivalent à prête à rouler donc bit à 1 dans OSPrioTable), j'ai annuler ce numéro, mais ceux qui l'avait bon ont eu leur point.

Question 2 (4 points) Ordonnancement, héritage de priorité et ICPP

Considérez les 6 tâches suivantes T1 à T6 sous µC (Figure 2.1). La séquence d'exécution peut être interprétée comme suit:

Tâche	Priorité	Nombre de *ticks* en attente avant de démarrer.	Nombre de *ticks* à exécuter (Ci)	Séquence d'exécution
T1	2	10	7	EBBEAAA
T2	4	8	3	EEE
T3	6	5	5	EEAAA
T4	8	3	4	EBBE
T5	10	2	2	EE
T6	12	0	10	EBBBBBAAAA

Figure 2.1

À partir des spécifications de la Figure 2.1 :

a) (**2 pts**) Complétez la trace d'exécution de la Figure 2.2a (page 4) en considérant le **mécanisme héritage de priorité** pour prévenir l'inversion de priorité :

   i. Indiquez également la priorité de T6 (juste en dessous de T6 sur la fig. 2.2a) pour chaque tick d'horloge d'exécution.

   ii. Donnez également le temps de blocage T1 pratique (obtenu sur la fig. 2.2a) et comparez le avec la borne supérieure sur le temps de blocage utilisé pour l'héritage de priorité.

   iii. Remplissez la table *OSPrioTbl* (Fig 2.2b) juste après le tick 11.

b) (**2 pts**) Complétez la trace d'exécution de la Figure 2.3 (page 5) en considérant le **mécanisme ICPP** pour prévenir l'inversion de priorité :

   i. Indiquez également la priorité de T6 (juste en dessous de T6 sur la fig. 2.3a) pour chaque tick d'horloge d'exécution.

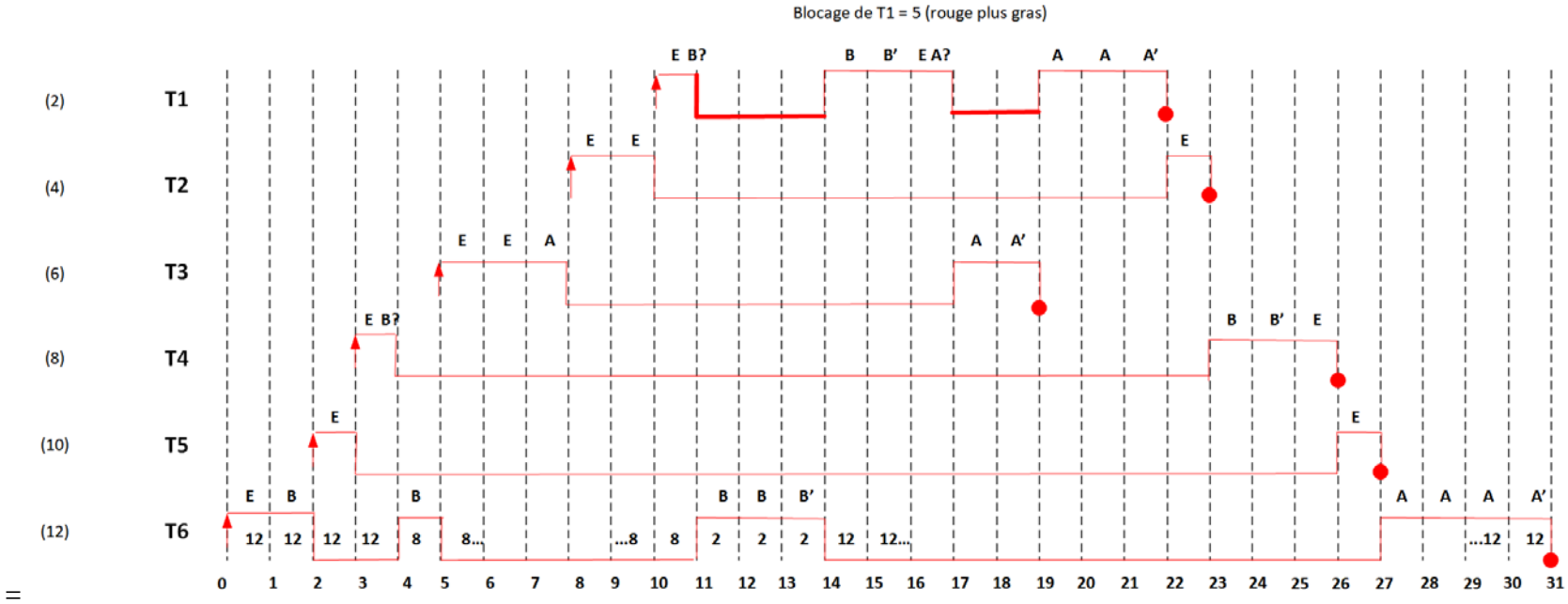
   ii. Donnez également le temps de blocage T1 pratique (obtenu sur la fig. 2.3a) et comparez le avec la borne supérieure sur le temps de blocage utilisé pour ICPP.

   iii. Remplissez la table *OSPrioTbl* (Fig 2.3b) juste après le tick 11.

N.B. Utilisez les symboles suivants pour compléter les figures 2.2 et 2.3 :

- *A? (or B? or C?) veut dire qu'une requête a été demandée (via OSMutexPend) pour accéder à la section critique, mais la section critique était occupée;*
- *A (or B or C) veut dire qu'une requête pour accéder à la section critique a été acceptée; et*
- *A' (or B' or C') indique que la tâche a terminé son exécution dans la section critique.*

Question 2) (Figures 2.2 à compléter)



=

a)

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

b) OSPrioTbl au Tick 11

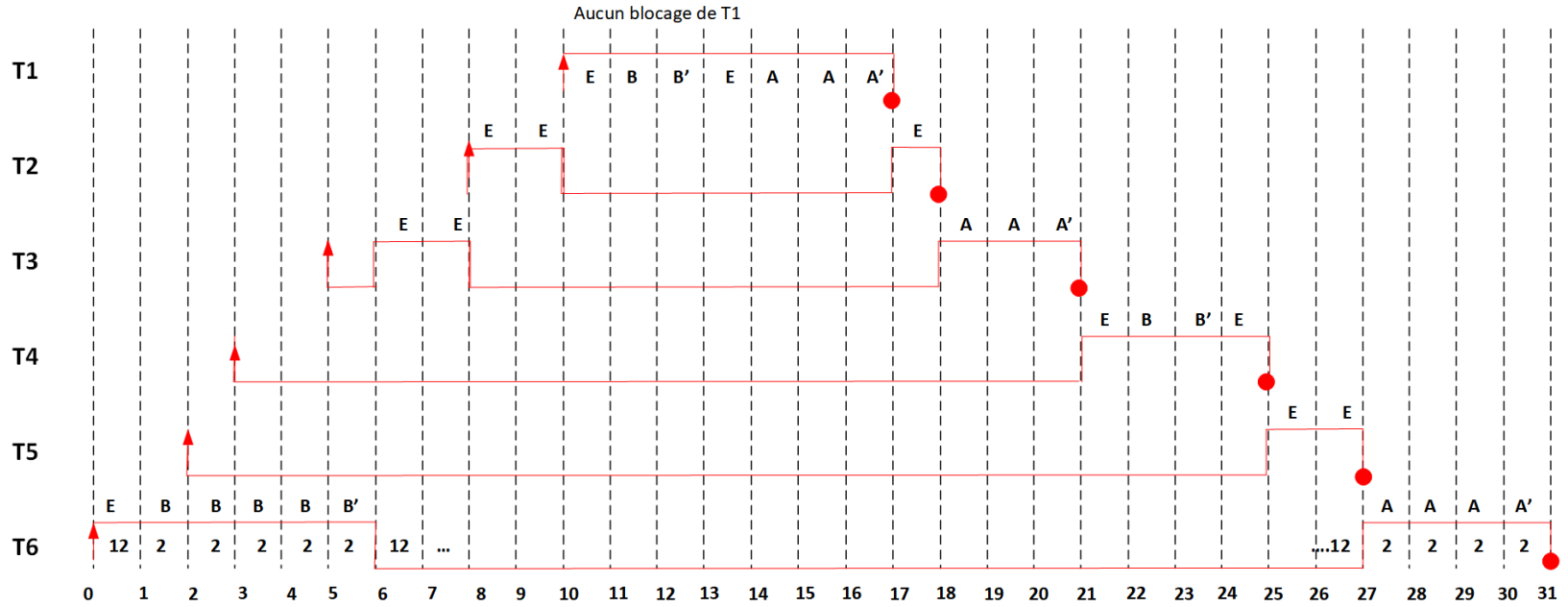
Blocage de T1 borne supérieure :

- *Venant de B*
  T1 peut être bloqué par T4 au maximum de 2 ticks
  T1 peut être bloqué par T6 au maximum de 5 ticks
  Par définition une seule tâche occupe B donc on prend le maximum soit 5

- *Venant de A*
  T1 peut être bloqué par T3 au maximum de 3 ticks
  T1 peut être bloqué par T6 au maximum de 4 ticks
  Par définition une seule tâche occupe B donc on prend le maximum soit 4

  Avec héritage de priorité on prend la somme de B et A soit 5 +4 = 9. Or sur le schéma on a un blocage de 5 (5 < 9).

Question 2) (Figures 2.3 à compléter)



b)

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

b) OSPrioTbl au Tick 11

Blocage de T1 borne supérieure :

- *Venant de B*
  T1 peut être bloqué par T4 au maximum de 2 ticks
  T1 peut être bloqué par T6 au maximum de 5 ticks
  Par définition une seule tâche occupe B donc on prend le maximum soit 5

- *Venant de A*
  T1 peut être bloqué par T3 au maximum de 3 ticks
  T1 peut être bloqué par T6 au maximum de 4 ticks
  Par définition une seule tâche occupe B donc on prend le maximum soit 4

  Avec héritage de priorité on prend le maximum de B et A soit 5 = 9. Or sur le schéma on a un blocage de 0 (0 < 5).

Question 3 (4 points) Synchronisation avec utilisation de Flags et utilisation d'une minuterie de type *watchdog*

Dans ce numéro pour lequel vous devrez compléter le code, les drapeaux d'évènements (*events flag*s) sont utilisés pour concevoir un sous-système de démarrage électronique d'une automobile. Le programme simplifié contient 3 tâches à synchroniser (*Task_Motor_Start*, *Task_Power_Up* et *Task_Stop*). De plus, on utilise un *OS_FLAG_GRP* à 3 bits mis à 1 via 3 ISRs de la manière suivante :

1) Bit *EVENT_KEY :* ce bit est mis à 1 par un ISR lié à un sensor qui détecte une clef. L'ISR fait alors l'authentification de la clef et fait un appel à *OSFlagPost(&CarStatus, EVENT_KEY, OS_OPT_POST_FLAG_SET + OS_OPT_POST_NO_SCHED, &err);*

2) Bit *EVENT_STARTER:* ce bit est mis à 1 par un ISR lié au système de démarrage (i.e. que le ISR rentre en fonction quand le conducteur appuie sur le démarreur). L'ISR fait alors appel à *OSFlagPost(&CarStatus, EVENT_STARTER, OS_OPT_POST_FLAG_SET + OS_OPT_POST_NO_SCHED, &err);*

3) Bit *EVENT_BRAKE:* ce bit est mis à 1 par un ISR lié au système de freinage (i.e. que le ISR entre en fonction quand le conducteur appuie sur la pédale). L'ISR fait alors appel à *OSFlagPost(&CarStatus, EVENT_BRAKE, OS_OPT_POST_FLAG_SET + OS_OPT_POST_NO_SCHED, &err);*

En ce qui concerne la fonctionnalité des différentes tâches, on a :

4) La tâche *Task_Motor_Start* se débloque (suite à un rendez-vous) lorsque les bits de *EVENT_KEY*, *EVENT_STARTER* et *EVENT_BRAKE* sont mis à 1. Ceci implique que le moteur démarre, les phares s'allument, les différents voyants du moteur s'affichent sur le tableau de bord (Figure 3.1) et il y a mise sous tension des appareils électriques (radio, chauffage, etc.). Notez aussi que cette tâche met la variable *CAR_ON* à vrai.

Figure 3.1 Exemples de voyants du moteur

5) La tâche *Task_Power_Up* se débloque (suite à un rendez-vous) lorsque les bits de *EVENT_KEY* et *EVENT_STARTER* sont mis à 1. Ceci qui implique que les phares s'allument, les différents voyants du moteur s'affichent sur le tableau de bord (Figure 3.1) et les accessoires (radio, chauffage, etc.) sont disponibles, mais le moteur ne démarre pas. Notez aussi que cette tâche met la variable *SYSTEM_ON* à vrai.

6) La tâche *Task_Stop* se débloque (suite à un rendez-vous) lorsque le moteur est en marche (variable globale MOTOR_ON) ou sous tension (variable globale SYSTEM_ON) et que le bit *EVENT_STARTER* est mis à 1. À ce moment-là, tout s'éteint sauf les phares qui restent allumés pendant 15 secondes à moins que *Task_Motor_Start* ou *Task_Power_Up* soient débloqués avant la fin de ces 15 secondes.

À partir du code de la figue 3.2, complétez le code:

a) Du bloc 1 qui initialise la minuterie *watchdog*
b) Du bloc 2 pour compléter le fonctionnement de *Task_Motor_Start*
c) Du bloc 3 pour compléter le fonctionnement de *Task_Power_Up*
d) Du bloc 4 pour compléter le fonctionnement de *Task_Stop*
e) De la fonction de rappel *StopCarFct*

N.B:

- *Pour les blocs 2 à 4 ça peut être du code à ajouter avant et/ou après le code déjà la.*
- *On suppose le tick d'horloge du système à 1000Hz et que la minuterie du watchdog est 100 fois plus lente*
- *La fonction de rappel du watchdog est plus prioritaire que Task_Motor_On, Task_Power_Up et TaskStop* et elle n'utilise aucun argument.
- *Quand OSTmrStart est appelé, il repart la minuterie au délai initial spécifié lors de la création de la minuterie.*
- *Un appel à OPSTmrStop lorsque la minuterie est déjà en arrêt n'a aucun effet.*

Une copie de la solution est en Annexe. Quelques commentaires :

1) *Je n'ai pas mis de mutex pour protéger CAR_ON et SYSTEM_ON car je ne l'avais pas demander (j'avais pas ça en tête quand j'ai fait le numéro). Mias à bien y penser ça prendrait les mutex. Bref, je n'ai pas enlever de points si il n'y avait pas de mutex et j'ai ajouté .5 pt à ceux qu'ils en avaient mis, en autant qu'ils étaient bien mis.*

2) *La bonne pratique était d'utiliser un consume dans les flags de Task Motor_Start et Task_Power_up. Beaucoup ne l'ont pas utilisé mais il fallait alors absolument*

faire un if (!CAR_ON) ou if (!SYSTEM_ON) pour éviter de redémarrer à chaque tour de boucles. D'ailleurs, avec le if ça reste une attente active ce qui pourrait créer la famine… Mais j'ai quand toutefois accepté le if mais souvenez-vous que ça prendrait un consume…

Question 4 (4 points) Ordonnancement périodique, délai et minuterie

On a vu en classe trois techniques d'ordonnancer des tâches en assurant pour chaque tâche un délai périodique.

a) (**2 pts**) Présenter **une de ces 3 techniques** et expliquer pourquoi il s'agit d'un délai périodique plutôt que relatif.

Je rappelle brièvement les 3 techniques vues (et répétées plusieurs fois) en classe :

1) *OSTimeDly(Period,OS_OPT_TIME_PERIODIC, &err);*
   et non OSTimeDly(Period,OS_OPT_TIME_DLY, &err);
   il fallait m'expliquer que le premier n'accumule pas le temps de calcul, ce que fait le délai relatif (réf. p. 13 du bloc 6, chap. 2)
2) *Utilisation d'un watchdog avec sémaphore dans la fonction de call back pour faire un rendez-vous pour rédémarrer une nouvelle période, comme le code présenté en classe et disponible sur le web (Minuterie Watchdog) (voir section Ressources et codes). Il fallait donc me donner les grandes lignes de cette configuration.*
3) *Avec une tâche supplémentaire et 2 sémaphores comme au no 32. Utile si on a pas l'option OS_OPT_TIME_PERIODIC et pas de watchdog.*

b) (**2 pts**) Soit un requis pour une application temps réel qui demande d'ordonnancer 3 tâches périodiques telle qu'illustrée à la figure 4.1. À l'aide de la technique choisie en a), expliquez comment mettre en œuvre ce requis. Plus précisément, indiquez ce que nécessiterait l'implémentation de la technique choisie en termes de ressources (e.g., minuteries, tâches, sémaphores, etc.) et de configuration (e.g., quels paramètres seraient passés à ces ressources).

Il fallait m'expliquer comment adapter une des 3 techniques au requis d'ordonnancement de la Fig. 4.1

Je l'ai fait pour la première décrite ci-haut. J'ai fait le code mais je n'en voulait évidemment pas temps. Ce code est disponible avec la solution.

	Period (T_i)	Temps de calcul (C_i)	Priorité (P_i)	Utilisation (U_i)
Task_1	90	40	3	0.5
Task_2	40	10	2	0.25
Task_3	20	5	1	0.25

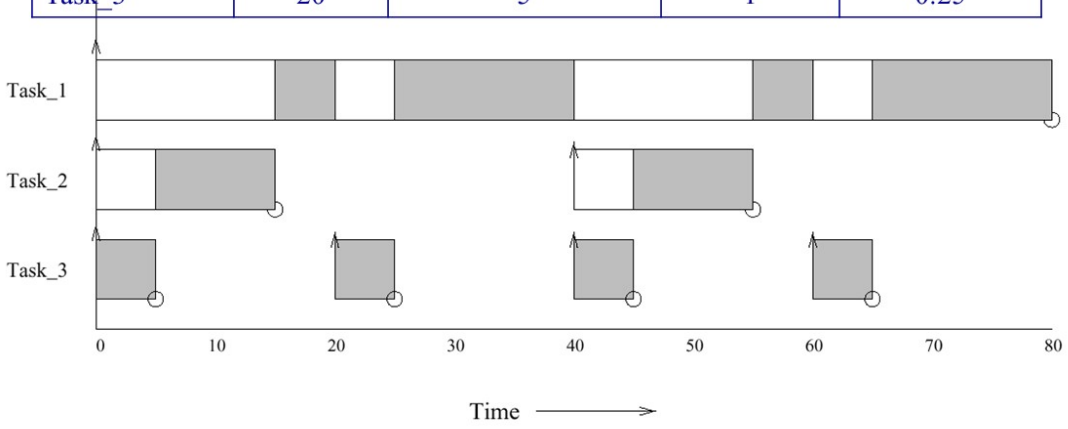


Figure 4.1

Question 5 (4.5 points) Interruptions sous uC/OS-III avec carte Zynq et Lab 1.

a) (1.5 pts) Soit la figure 5.1 qu'on a vue en classe :

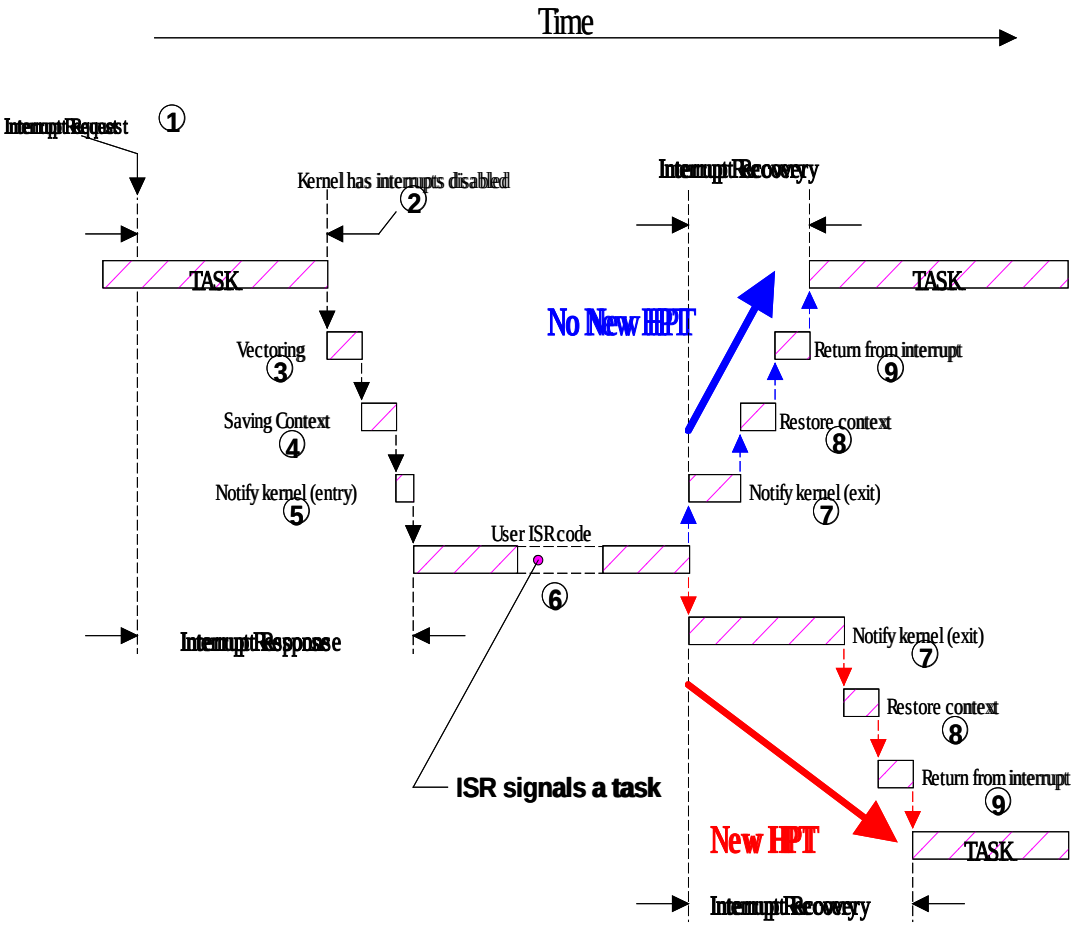


Figure 5.1

Expliquez brièvement ce que représente la figure 5.1 et indiquez à quel endroit sur la figure sont appelés : *OS_CPU_ARM_EXCEPT_IRQ()*, *uCOS_IntHandler()*, *XIntc_DeviceInterruptHandler()*, *OSIntExit()* et *OSIntCtxSw()*.

Il s'agissait du déroulement en mode préemptif d'une interruption matérielle.

- *OS_CPU_ARM_EXCEPT_IRQ() s'exécutait après 3 (la table des vecteurs fait un branchement à l'adresse de cette fonction.*
- *uCOS_IntHandler() est appelé dans le GIC donc dans 5*
- XIntc_DeviceInterruptHandler() est du code de driver sur FPGA donc appelé dans 6
- *OSIntExit()* permet de voir qu'on a pu d'interruption à traiter et d'appeler *OSPrioGetHighest() pour savoir si on va sur la branche du haut ou celle du bas. Donc appelé de 7.*

- *OSIntCtxSw() Une fois OSIntExit() complété et qu'on a bel et bien pu d'interruption, on peut procéder au changement de contexte. Donc appelé de 8.*

b) (1.5 pts) Comparez différences et similitudes au niveau fonctionnalité pour *OSFlagPend()*/*OSFlaPost()* et *OSSemPend()/OSSemPost()*.

Je voulais au moins 1 similitude et 1 différence.

Similitude :
Les deux permettent le rendez-vous unilatéral

Différences :
Flag peut supporter dans la même structure plusieurs n évènements alors que le sémaphore demandera n structures.
Autres différences :

- *FlagPost peut incrémenter ou décrémenter (resp. pour RDV et blocage) alors que Sempost ne permet que d'incrémenter.*
- *FlagPend peut consommer ou pas (resp. pour RDV une fois et RDV pour toujours).*

c) (.75 pts) Expliquez la différence(s) entre *OSIntExit()* et *OSSched()*

OSIntExit() est utilisé dans les interruptions matérielles

OSSched() est utilisé dans les interruptions logicielles (toutes les fonctions uC qui bloquent vont juste avant ce blocage appeler OSSched().

N.B. Les deux appellent OSPrioGetHighest() mais OSIntExit() n'appellent pas OSSched()

d) (.75 pt) Expliquez la différence(s) entre *OSCtxSw(), OSIntCtxSw()* et *OSStartHighRdy()*.

OSCtxSw() est un changement de contexte pour interruption logicielle (video dans bloc 5 du chap. 2).

OSIntCtxSw() est l'étape 8 de la question a) pour laquelle on charge le contexte de la tâche obtenue suite à OSPrioGetHighest() (branche du haut ou branche du bas sur la figue 5.1). N.B. Contrairement à OSCtxSw(), ici on n'a pas à retirer de tâches du CPU car cela a été fait à l'étape 4 de la Figure 5.1.

OSStartHighRdy() est le 1er changement de contexte juste après OSStart(). Il s'agit d'un cas spécial car on n'a pas à retirer de tâches du CPU.

Annexe

```
OS_SEM_CTR  OSSemPend (OS_SEM   *p_sem,
                       OS_TICK   timeout,
                       OS_OPT    opt,
                       CPU_TS   *p_ts,
                       OS_ERR   *p_err)
```

opt

OS_OPT_PEND_BLOCKING
to block the caller until the semaphore is available or a timeout occurs.

OS_OPT_PEND_NON_BLOCKING
If the semaphore is not available, OSSemPend() will not block but return to the caller with an appropriate error code.

```
OS_SEM_CTR  OSSemPost (OS_SEM  *p_sem,
                       OS_OPT   opt,
                       OS_ERR  *p_err)
```

opt

OS_OPT_POST_1
Post and ready only the highest-priority task waiting on the semaphore.

OS_OPT_POST_NO_SCHED
This option indicates that the caller does not want the scheduler to be called after the post. This option can be used in combination with one of the two previous options.
You should use this option if the task (or ISR) calling OSSemPost() will be doing additional posting and, the user does not want to reschedule until all done, and multiple posts are to take effect simultaneously.

```
void  OSTimeDly (OS_TICK   dly,
                 OS_OPT    opt,
                 OS_ERR   *p_err)
```

Arguments

dly

is the desired delay expressed in number of clock ticks. Depending on the value of the opt field, delays can be relative or absolute.

A relative delay means that the delay is started from the "current time + dly".

A periodic delay means the period (in number of ticks). µC/OS-III saves the current time + dly in .TickCtrPrev so the next time OSTimeDly() is called, we use .TickDlyPrev + dly.

An absolute delay means that the task will wake up when OSTickCtr reaches the value specified by dly.

opt

is used to indicate whether the delay is absolute or relative:

OS_OPT_TIME_DLY

Specifies a relative delay.

OS_OPT_TIME_TIMEOUT

Same as OS_OPT_TIME_DLY.

OS_OPT_TIME_PERIODIC

Specifies periodic mode.

OS_OPT_TIME_MATCH

Specifies that the task will wake up when OSTickCtr reaches the value specified by dly

```
OS_FLAGS  OSFlagPend (OS_FLAG_GRP  *p_grp,
                      OS_FLAGS     flags,
                      OS_TICK      timeout,
                      OS_OPT       opt,
                      CPU_TS       *p_ts,
                      OS_ERR       *p_err)
```

Arguments
p_grp is a pointer to the event flag group.

flags
is a bit pattern indicating which bit(s) (i.e., flags) to check. The bits wanted are specified by setting the corresponding bits in flags. If the application wants to wait for bits 0 and 1 to be set, specify 0x03. The same applies if you'd want to wait for the same 2 bits to be cleared (you'd still specify which bits by passing 0x03).

timeout
allows the task to resume execution if the desired flag(s) is (are) not received from the event flag group within the specified number of clock ticks. A timeout value of 0 indicates that the task wants to wait forever for the flag(s). The timeout value is not synchronized with the clock tick. The timeout count begins decrementing on the next clock tick, which could potentially occur immediately.

opt
specifies whether all bits are to be set/cleared or any of the bits are to be set/cleared. Here are the options:

OS_OPT_PEND_FLAG_CLR_ALL

If OS_CFG_FLAG_MODE_CLR_EN is set to DEF_ENABLED in os_cfg.h, check all bits in flags to be clear (0)

OS_OPT_PEND_FLAG_CLR_ANY

If OS_CFG_FLAG_MODE_CLR_EN is set to DEF_ENABLED in os_cfg.h, check any bit in flags to be clear (0)

OS_OPT_PEND_FLAG_SET_ALL

Check all bits in flags to be set (1)

OS_OPT_PEND_FLAG_SET_ANY

Check any bit in flags to be set (1)

The caller may also specify whether the flags are consumed by "adding" OS_OPT_PEND_FLAG_CONSUME to the opt argument. For example, to wait for any flag in a group and then clear the flags that satisfy the condition, you would set opt to:

OS_OPT_PEND_FLAG_SET_ANY + OS_OPT_PEND_FLAG_CONSUME

Finally, you can specify whether you want the caller to block if the flag(s) are available or not. You would then "add" the following options:

OS_OPT_PEND_BLOCKING
OS_OPT_PEND_NON_BLOCKING

Note that the timeout argument should be set to 0 when specifying OS_OPT_PEND_NON_BLOCKING, since the timeout value is irrelevant using this option. Having a non-zero value could simply confuse the reader of your code.

```
OS_FLAGS  OSFlagPost (OS_FLAG_GRP  *p_grp,
                      OS_FLAGS     flags,
                      OS_OPT       opt,
                      OS_ERR       *p_err)
```

Arguments

p_grp
is a pointer to the event flag group.
flags
specifies which bits to be set or cleared. If opt is OS_OPT_POST_FLAG_SET, each bit that is set in flags will set the corresponding bit in the event flag group. For example to set bits 0, 4, and 5, you would set flags to 0x31 (note that bit 0 is the least significant bit). If opt is OS_OPT_POST_FLAG_CLR, each bit that is set in flags will clear the corresponding bit in the event flag group. For example to clear bits 0, 4, and 5, you would specify flags as 0x31 (again, bit 0 is the least significant bit).
opt
indicates whether the flags are set (OS_OPT_POST_FLAG_SET) or cleared (OS_OPT_POST_FLAG_CLR).
The caller may also "add" OS_OPT_POST_NO_SCHED so that µC/OS-III will not call the scheduler after the post.

```
void  OSTmrCreate (OS_TMR              *p_tmr,
                   CPU_CHAR            *p_name,
                   OS_TICK             dly,
                   OS_TICK             period,
                   OS_OPT              opt,
                   OS_TMR_CALLBACK_PTR   p_callback,
                   void                *p_callback_arg,
                   OS_ERR              *p_err)
```

Arguments

dly
specifies the initial delay (specified in timer tick units) used by the timer (see drawing above). If the timer is configured for ONE-SHOT mode, this is the timeout used. If the timer is configured for PERIODIC mode, this is the timeout to wait before the timer enters periodic mode. The units of this time depends on how often the user will call OSTmrSignal() (see OSTimeTick()). If OSTmrSignal() is called every 1/10 of a second (i.e., OS_CFG_TMR_TASK_RATE_HZ set to 10), dly specifies the number of 1/10 of a second before the delay expires.
period
specifies the period repeated by the timer if configured for PERIODIC mode. You would set the "period" to 0 when using ONE-SHOT mode. The units of time depend on how often OSTmrSignal() is called. If OSTmrSignal() is called every 1/10 of a second (i.e., OS_CFG_TMR_TASK_RATE_HZ set to 10), the period specifies the number of 1/10 of a second before the timer repeats.
opt
is used to specify whether the timer is to be ONE-SHOT or PERIODIC:
OS_OPT_TMR_ONE_SHOT specifies ONE-SHOT mode
OS_OPT_TMR_PERIODIC specifies PERIODIC mode

```
CPU_BOOLEAN  OSTmrStop (OS_TMR  *p_tmr,
                        OS_OPT   opt,
                        void    *p_callback_arg,
                        OS_ERR  *p_err)
```

Pour simplifier, si vous désirez stopper la minuterie, faites simplement
OSTmrStop(&StopCar, OS_TMR_NONE, (void *)0, &err);

CPU_BOOLEAN OSTmrStart (OS_TMR *p_tmr,
OS_ERR *p_err)

Pour simplifier, si vous désirez démarrer la minuterie, faites simplement
OSTmrStart(&&StopCar, &err);

Héritage de priorité

- **Avec l'héritage de priorité on a:**

$$B_i = \sum_{k=1}^{K} usage(k,i)CS(k)$$

Où:

- ***usage*** **est une fonction 0/1: *usage*(*k*,*i*) = 1 si la ressource *k* est utilisé par au moins une tâche de priorité inférieure à *i*, et au moins une tâche de priorité supérieure ou égale à *i*, sinon 0.**
- **CS(*k*) le temps requis pour passer au travers la section critique *k*.**

- **Avec ICPP on a:**

$$B_i = \max_{k=1}^{K} usage(k,i)CS(k)$$

Où:

- ***usage*** **est une fonction 0/1: *usage*(*k*,1) = 1 si la ressource *k* est utilisé par au moins une tâche de priorité inférieure à *i*, et au moins une tâche de priorité supérieure ou égale à *i*, sinon 0.**
- **CS(*k*) le temps requis pour passer au travers la section critique *k*.**