POLYTECHNIQUE MONTRÉAL

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**	**202301**
Professeur	***Local***	***Téléphone***
Guy Bois	**M-5105**	**5944**

Jour	*Date*	*Durée*	*Heures*
Jeudi	**16 mars 2023**	**2h30**	**14h00 à 16h30**

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 **15** pages **(excluant cette page)**

- La pondération de cet examen est de **25** **%**
- Répondez au no 2 directement sur le questionnaire
- Pour les no 1), 3), 4) et 5), répondez dans le cahier de réponse.
- Vous devez donc remettre le cahier de réponse et le questionnaire

Solution

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

Question 1 (4 points) En vrac : vrai ou faux avec <u>justifications</u>

1. (.8 pt) Soit le recours à la *désactivation des interruptions pour garantir la protection d'une ressource partagée*. Celle-ci fonctionne dans les 3 situations suivantes : 1) entre deux ISR, 2) entre deux tâches et 3) entre un ISR et une tâche.

   Faux à cause de 2). Si on désactive les interruptions, on peut quand même passer d'une tache à l'autre (p.e. cas ou une tâche s'arrête dans une section critique sur un pend de fifo vide et l'ordonnanceur passe alors la main à une autre tâche).

2. (.8 pt) Le flag uC/OS-III pourrait être utilisé et joué le même rôle que le mutex uC/OS-III afin d'éviter la corruption de données, mais il n'offrirait alors pas l'héritage de priorité.

   Vrai car OSFlagPost(&F1, TASK_B_RDY, OS_OPT_POST_FLAG_SET + OS_OPT_POST_NO_SCHED, &err; et OSFlagPend(&F1, TASK_B_RDY, 0, OS_OPT_PEND_FLAG_SET_ALL + OS_OPT_PEND_BLOCKING + OS_OPT_PEND_FLAG_CONSUME, &ts, &err) peuvent jouer le même rôle qu'un semaphore. Or le semaphore peut aussi être utilisé pour la gestion de sections citiques mais sans héritage de priorité.

3. (.8 pt) Soit I_i le temps d'interférence (ou de préemption) d'une tâche T_i et B_i le temps de blocage d'une tâche T_i. On peut alors affirmer que I_i est toujours égal à 0 lorsque T_i est la tâche la plus prioritaire du système et $B_i = 0$ lorsque T_i est la tâche la moins prioritaire du système.

   Vrai - Preuve par l'absurde : si T_i est interférée, c'est qu'il existe une tâche plus prioritaire, or par définition T_i est la tâche plus prioritaire. Donc $I_i = 0$. De même, si Ti est bloquée, c'est qu'il existe une tâche moins prioritaire, or par définition Ti est la tâche la moins prioritaire. Donc $B_i = 0$.

4. (.8 pt) La fonction *OSIntExit()* appelle l'ordonnanceur uC/OS-III, et au besoin, procède au changement de contexte à la fin d'une interruption logicielle et d'une interruption matérielle.

   Faux c'est OSSched() qui s'occupe des interruptions logicielles pas OSintExit()

5. (.8 pt) Soit une application composée de 4 tâches Task_1 à Task_4 en ordre décroissant de priorité et utilisant trois mutex M1, M2 et M3 :

	M1	M2	M3
Task_1	5	2	0
Task_2	0	7	1
Task_3	6	5	0
Task_4	7	8	9

Dans le cas de l'utilisation de ICPP, le temps maximal de blocage B_i pour chacune des 4 tâches est : B1 = 8, B2 = 9, B3 =**8** et B4 = 0.

Faux car

B1 = max (max(6,7), max(7, 5, 8)) = 8

B2 = max(max(6, 7), mx(5,8), max(9)) = 9

B3 = max(max(7), max(8), max(9)) = ***9 ≠ 8***

B4 = 0

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

N.B. on fait ici l'hypothèse que ICPP est disponible sous uC/OS-III

Soit le schéma de la figure 2.1 une trace d'exécution avec uC/OS-III selon la convention établie en classe :

- *B?* équivaut à un *Pend* qui bloque une tâche et la met en attente,
- *B* équivaut à sortir de l'attente et obtenir le mutex et
- *B'* équivaut à un *Post*.

De plus, les tâches Task_5 à Task_1 démarrent dans l'ordre 0, 1, 3, 4 et 5 respectivement.

a) (.5 pt) Dans ce même schéma, quel protocole utilise-t-on pour minimiser l'inversion de priorité? Justifiez directement sur la figure 2.1.

b) (1 pt) Directement sur la figure 2.1, indiquez les priorités de Task_4 et Task_5 pour chaque tick, et ce pour la durée de leur exécution.

c) (1 pt) Complétez directement sur la figure 2.2 (page 5), les tables *OS_Prio_Tbl* au début des ticks 5, 6 et 8. Expliquez.

d) (1.5 pt) Si en a) vous avez répondu *protocole héritage de priorité*, complétez la figure 2.3 pour le *protocole ICPP* et si en a) vous avez choisi le *protocole ICPP*, complétez la figure 2.3 pour le *protocole héritage de priorité*. Indiquez également les priorités de Task_4 et Task_5 pour chaque tick, et ce pour la durée de leur exécution.

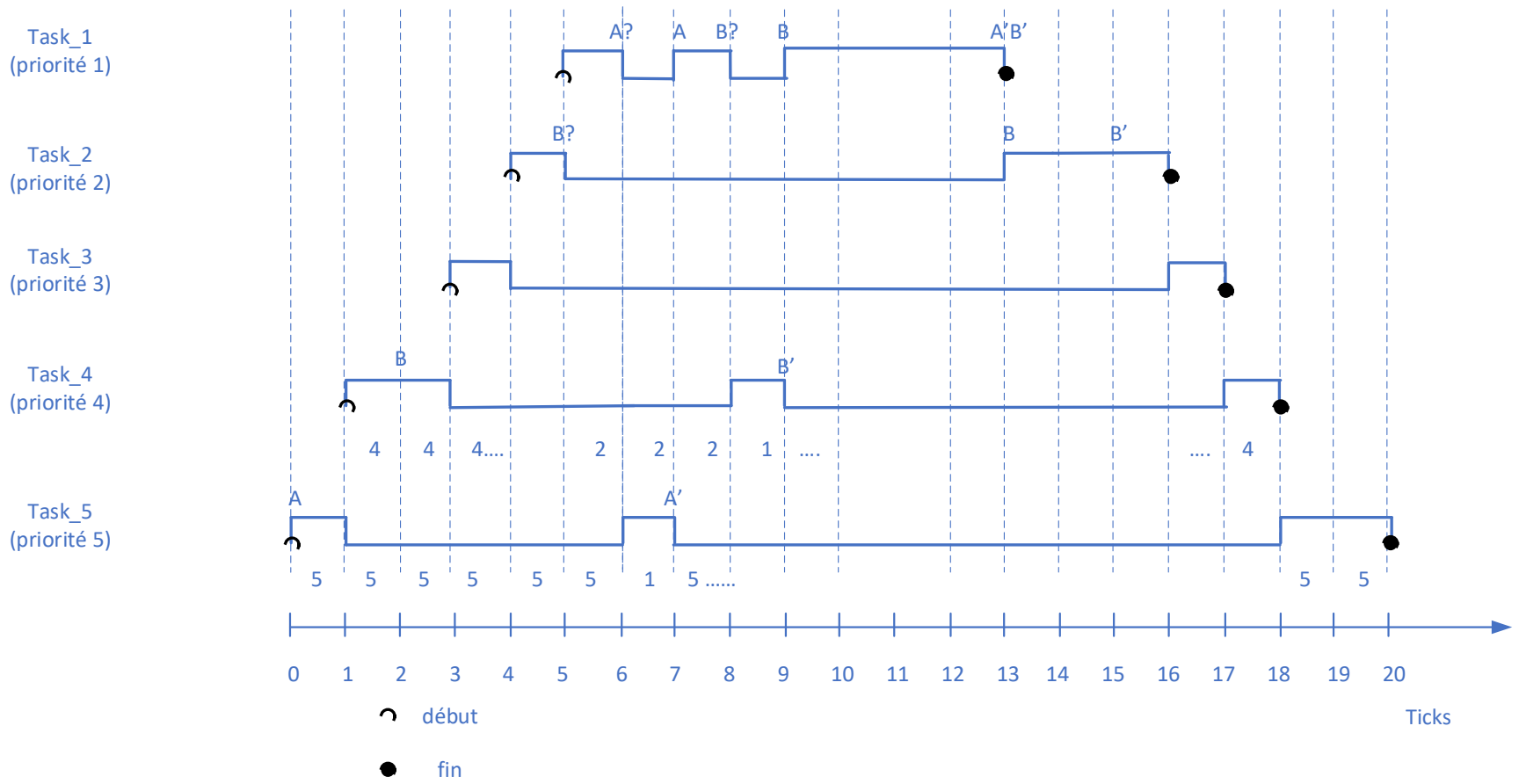
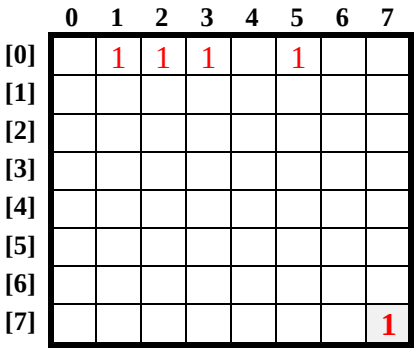


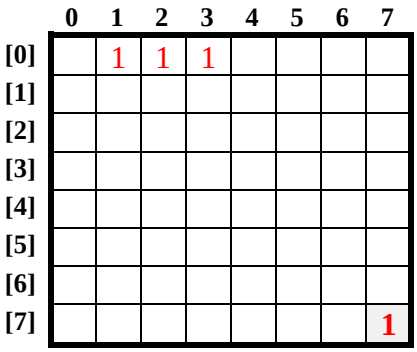
Figure 2.1 À compléter pour les no 2b) et 2c)

a) On utilise le protocole héritage de priorité car T5, une fois dans la section critique A, attend vraiment que T1 demande de rentrer dans A avant de recevoir la priorité de ce dernier, puis poursuivre son exécutio



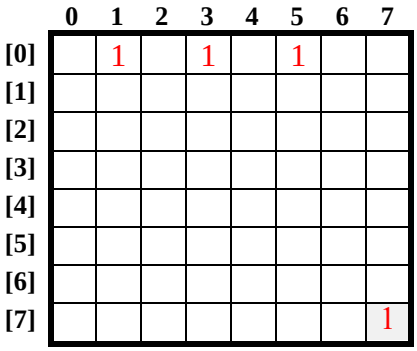
OSPrioTbl[]

Début du tick 5
(N.B. 1 3 4 5 si pas héritage de priorité 2 en b)



OSPrioTbl[]

Début du tick 6
(N.B. 1 3 4 si pas héritage de priorité 2 en b)



OSPrioTbl[]

Début du tick 8

Figure 2.2 À compléter pour le no 2d)

N.B. Un espace vide correspondra à une tâche non prête alors qu'un 1 correspondra à une tâche prête (ready)

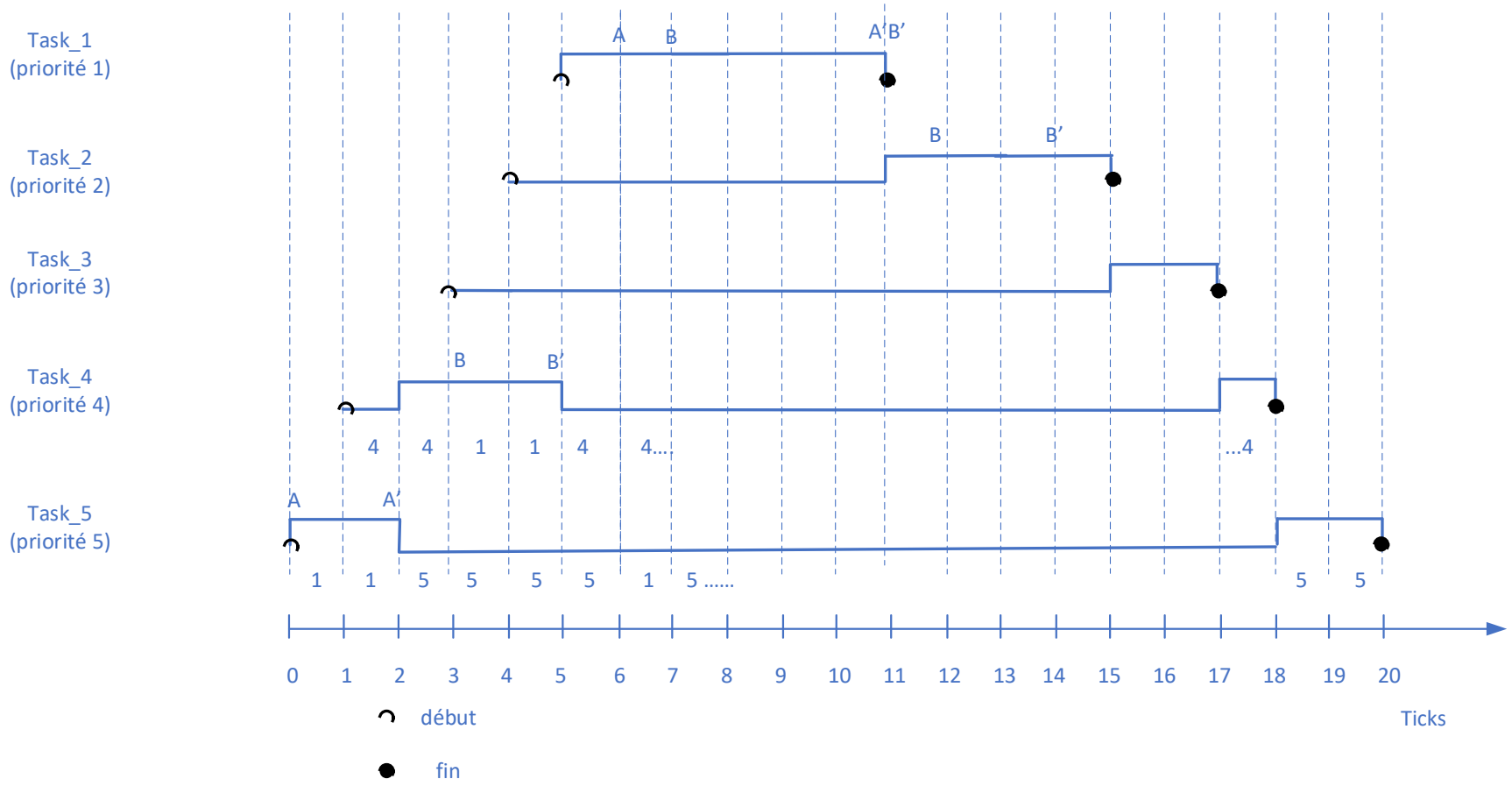


Figure 2.3 À compléter pour le no 2e)

Question 3 (4 points) Analyse de code avec uC/OS-III

À partir de la figure 3.1 de la page suivante, 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 l'héritage de priorité est utilisé.

Pour chaque évènement, vous devez :

1) (1 pt) Donnez l'état de la tâche correspondante (aidez-vous au besoin de la figure 4.2, page 9);
2) (2 pt) 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 système (au besoin, consultez l'Annexe).
3) (1 pt) S'il y a lieu, les changements de contexte lorsque l'ordonnanceur est appelé ou lorsque le mécanisme d'héritage de priorité est appliqué.

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 le masque TASK_B_RDY de F1 est à 0, la Tâche B passe de RUNNING à PENDING. La fonction système OS_PrioRemove() 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 F1 (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 OSFlagPost(), 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 F1 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[].*

- *On appelle pas l'ordonnanceur car on utilise l'option OS_OPT_POST_NO_SCHED puisqu'on sait que OSintExit le fera.*

Event 6

- *Avant de terminer, ISR exécute OSIntExit qui décrémente OSIntNesting, fait un appel à l'ordonnanceur (OSPrioGetHighest) et détermine que la tâche B a la plus haute priorité, puis finalement fait un changement de contexte avec OSIntCtxtSw.*
- *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_PrioRemove() 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*

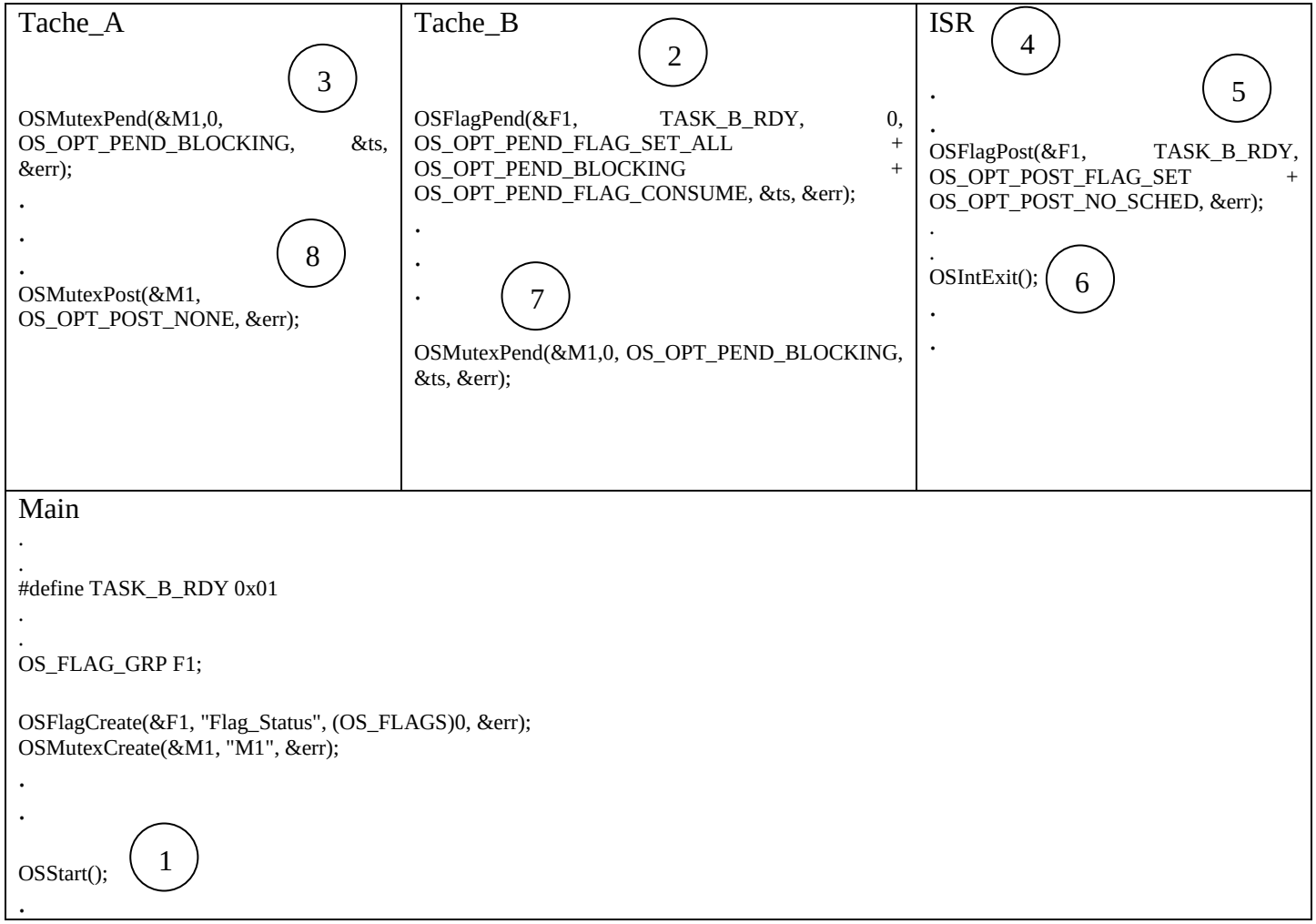

Tache_A
3
OSMutexPend(&M1,0,
OS_OPT_PEND_BLOCKING, &ts,
&err);
.
.
.
8
OSMutexPost(&M1,
OS_OPT_POST_NONE, &err);
Tache_B
2
OSFlagPend(&F1, TASK_B_RDY, 0,
OS_OPT_PEND_FLAG_SET_ALL +
OS_OPT_PEND_BLOCKING +
OS_OPT_PEND_FLAG_CONSUME, &ts, &err);
.
.
.
7
OSMutexPend(&M1,0, OS_OPT_PEND_BLOCKING,
&ts, &err);
ISR
4
.
.
5
OSFlagPost(&F1, TASK_B_RDY,
OS_OPT_POST_FLAG_SET +
OS_OPT_POST_NO_SCHED, &err);
.
.
OSIntExit();
6
.
.
Main
.
.
#define TASK_B_RDY 0x01
.
.
OS_FLAG_GRP F1;
OSFlagCreate(&F1, "Flag_Status", (OS_FLAGS)0, &err);
OSMutexCreate(&M1, "M1", &err);
.
.
OSStart();
1
.

Figure 3.1

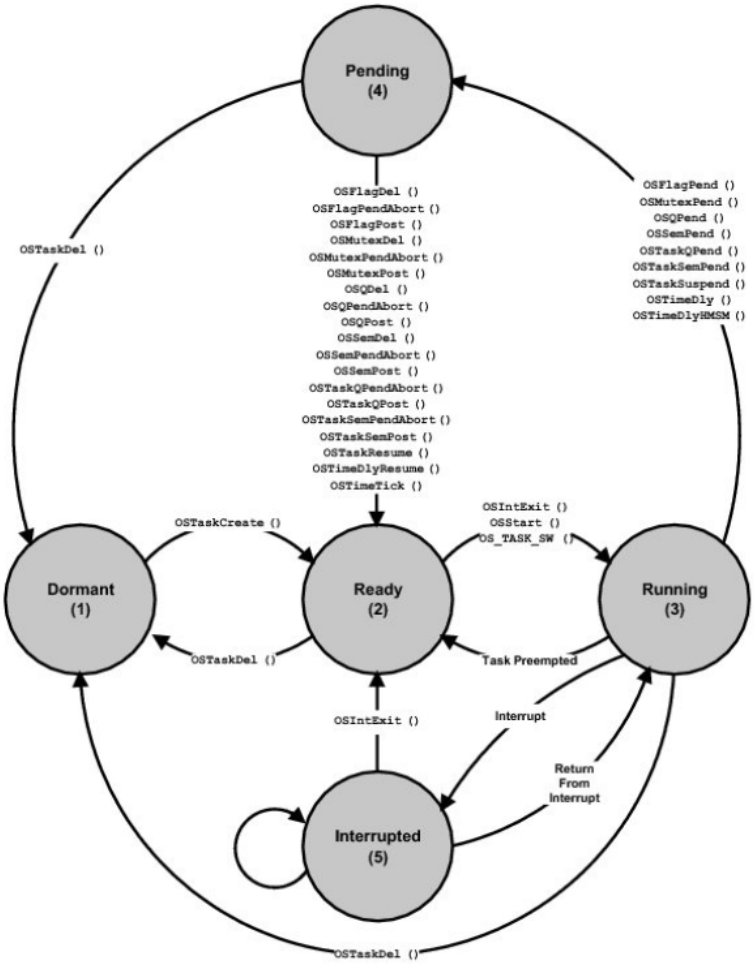

Pending
(4)
OSFlagDel ()
OSFlagPendAbort ()
OSFlagPost ()
OSMutexDel ()
OSMutexPendAbort ()
OSMutexPost ()
OSQDel ()
OSQPendAbort ()
OSQPost ()
OSSemDel ()
OSSemPendAbort ()
OSSemPost ()
OSTaskQPendAbort ()
OSTaskQPost ()
OSTaskSemPendAbort ()
OSTaskSemPost ()
OSTaskResume ()
OSTimeDlyResume ()
OSTimeTick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

Figure 4.2

Question 4 (4 points) Délai et minuterie

Soit le système sur puce similaire à votre Pynq-Z2 composé entre autres d'un ARM A9 à 2 cœurs sur lequel est directement disponibles un compteur 32 bits et un contrôleur d'interruption. Le CPU fonctionne à une fréquence de 600 MHz et le compteur du ARM est raccordé à un signal d'horloge à 100 MHz. De plus, un FPGA peut communiquer avec le CPU à travers un bus et des signaux d'interruptions IRQ.

Finalement à l'aide des outils Vivado et SDK on utilise uC/OS-III pour rouler un système temps réel.

a) (0.5 pt) Calculer la valeur à laquelle le compteur doit être initialisé par uC/OS-III pour qu'il délivre des interruptions via *OSTimeTick* avec une fréquence de 200 Hz.

200 Hz fait 5 ms comme période de tick. Donc .005 sec diviser par $1/100x10^6$ va donner 500000.

b) (1 pt) Définir en quelques lignes la fonctionnalité de *OSTimeTick*.

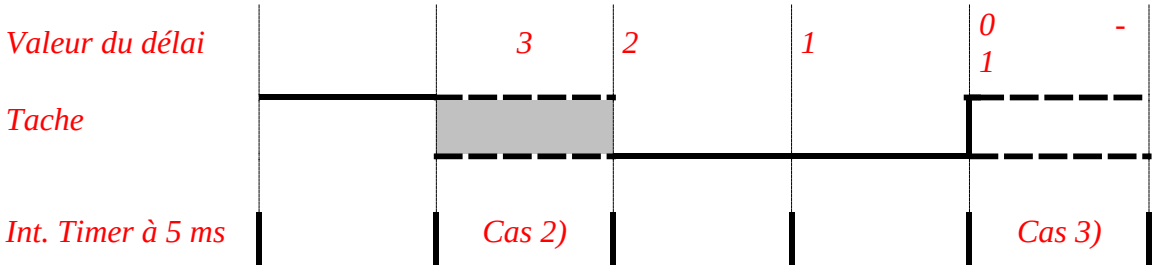
Appelez à chaque fin de tick (ouvent ISR0), fais 2 choses principales :

1) *Incrémente le nombre de ticks obtenu depuis le début de l'exécution (valeur retourner quand on fait OSTimeGet)*
2) *Pour chaque tâche en attente d'un délai (p.e. OSTimeDly) on décrémente le délai. Si ce dernier est à 0 pour une tâche donnée, on remet active (ready) la tâche (via OSPendListRmove et OSPrioInsert).*

c) (1.5 pt) Dans une des tâches du système, on fait appel à un délai de 12 ms via OSTimeDlyHMSM(0, 0, 0, 12, OS_OPT_TIME_HMSM_STRICT, &err). Toujours avec la fréquence de 200 Hz des interruptions via OSTimeTick, quelle sera la précision de ce délai? Expliquez et faites un schéma expliquant les différentes sources d'incertitude.

Il existe 3 sources contribuant à l'incertitude :

1) *Le délai sera de 2 ou 3 ticks suivant la politique d'arrondi du système car 12/5 = 2.4.*
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.

d) (1pt) Toujours pour ce délai de 12 ms, proposez une solution pour améliorer la précision. Expliquez ce que vous feriez et d'où viendrait le gain en précision par rapport à c).

Hint : Cette solution peut être logiciel et/ou matériel. Autrement dit, vous pouvez modifier les paramètres du BSP de uC et/ou utilisez le FPGA.

Solution logicielle : s'assurer que le tick divise parfaitement le délai p.e. 12/2 ou 12/1, etc. Peut être changé dans le BSP dans SDK.

Solution matérielle : ajout d'une minuterie auxiliaire (genre fittimer) qui va donner un délai exacte de 12 ms. Demande de modifier e design dans Vivado.

Question 5 (4 points) Laboratoire no 1 parties 1 et 2

a) À propos de la comparaison des délais pour vider les fifos à 1000 Hz et 2000 Hz (Q2, partie 1) :

a.1) (.7 pt) Le résultat aurait-il été différent de celui que vous avez obtenu si au lieu d'utiliser des valeurs *WAITFORComputing* différentes, vous aviez utilisé la même valeur à 1000 Hz et 2000 Hz (p.e., la 5 ticks)? Expliquez.

Oui surement car soit 5 ticks, pour 1000 Hz ça fait 5 ms d'attente active alors que pour 2000Hz ça fait 2.5 ms d'attente active. La tâche à 2000Hz va pouvoir récupérer ce temps pour vider les FIFOs plus rapidement..

a.2) (.7 pt) Si vous aviez pu raffiner davantage votre recherche binaire à la question 2 de la partie 1, par exemple à 1 ms près au lieu de 125 ms, aurait-on eu selon vous un plus petit délai à 1000 Hz ou à 2000 Hz? Expliquez.

Je n'ai pas eu le temp de le tester, mais je reste convaincu que le délai aurait été plus petit à 2000Hz. Ceci parce la précision (no 4c) est meilleure à 2000Hz.Donc les tâhes retrouvent leur exécution plus rapidement.

Ceci dit, j'acceptais les 2 (mais un ou l'autre) à condition qu'il y avait une justification raisonnable.

b) (.7 pt) Expliquez le rôle de *OS_OPT_POST_NO_SCHED* dans

OSFlagPost(&RouterStatus, TASK_STOP_RDY, OS_OPT_POST_FLAG_SET + OS_OPT_POST_NO_SCHED, &perr) dans *fit_timer_isr1*.

On veut pas d'appel à l'ordonnanceur (OSPrioGetHighest) si parfois une tâche devenait active (sort de la liste d'attente).

Plus précisément que se passerait-il si on l'avait omis? Expliquez.

On ralentie le ISR (mais ça fonctionne pareil) et de toute façon on sait que l'ordonnanceur sera appelé dans OSIntExit.

c) (.7 pt) Expliquez le rôle de *OS_OPT_PEND_FLAG_CONSUME* dans :

OSFlagPend(&RouterStatus, TASK_STATS_PRINT, 0, OS_OPT_PEND_FLAG_SET_ALL + OS_OPT_PEND_BLOCKING + OS_OPT_PEND_FLAG_CONSUME, &ts, &err) de TaskStats

On attend ici que le bit passe à 1, une fois que c'est fait il est remis à 0. Un peu comme un sémaphore qui un fois le pend exécuté remet le sémaphore à 0.

Plus précisément que se passerait-il si on l'avait omis? Expliquez.

TaskStats se serait exécuté en boucle car c'est la tâche la plus prioritaire…

d) (1.2 pts) Décrivez 3 fonctionnalités réalisées par la tâche OSIntExit().

Gestion du OSIntNesting
Si OSIntNesting == 0 on ***fait appel à l'ordonnanceur (OSPrioGetHighest)*** *puis on fait un appel pour un* ***changement de contexte (OSIntCtxSw)****.*

Annexe

```
void  OSMutexPend (OS_MUTEX  *p_mutex,
                   OS_TICK    timeout,
                   OS_OPT     opt,
                   CPU_TS    *p_ts,
                   OS_ERR    *p_err)
```

opt
determines whether the user wants to block if the mutex is not available or not. This argument must be set to either:
OS_OPT_PEND_BLOCKING, or
OS_OPT_PEND_NON_BLOCKING

```
void  OSMutexPost (OS_MUTEX  *p_mutex,
                   OS_OPT     opt,
                   OS_ERR    *p_err);
```

opt
OS_OPT_POST_NONE
No special option selected.
OS_OPT_POST_NO_SCHED
Do not call the scheduler after the post, therefore the caller is resumed even if the mutex was posted and tasks of higher priority are waiting for the mutex.
Use this option if the task calling OSMutexPost() will be doing additional posts, if the user does not want to reschedule until all is complete, and multiple posts should take effect simultaneously.

```
void  OSFlagCreate (OS_FLAG_GRP  *p_grp,
                    CPU_CHAR     *p_name,
                    OS_FLAGS      flags,
                    OS_ERR       *p_err)
```

Arguments
p_grp
This is a pointer to an event flag group that must be allocated in the application. The user will need to declare a “global” variable as shown, and pass a pointer to this variable to OSFlagCreate():
OS_FLAG_GRP MyEventFlag;
p_name
This is a pointer to an ASCII string used for the name of the event flag group. The name can be displayed by debuggers or by µC/Probe.
flags
This contains the initial value of the flags to store in the event flag group. Typically, you would set all flags to 0 events correspond to set bits and all 1s if events correspond to cleared bits.

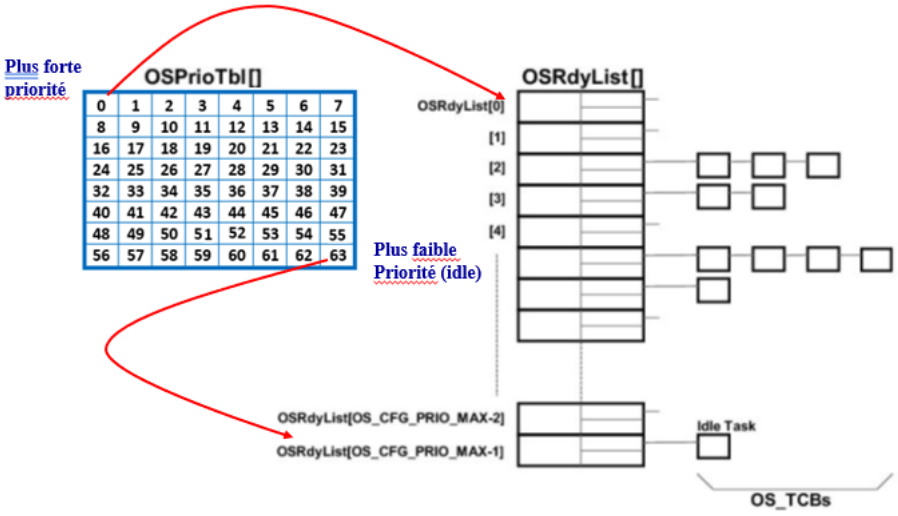
```
OS_FLAGS  OSFlagPost (OS_FLAG_GRP  *p_grp,
                      OS_FLAGS      flags,
                      OS_OPT        opt,
                      OS_ERR       *p_err)
```

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.

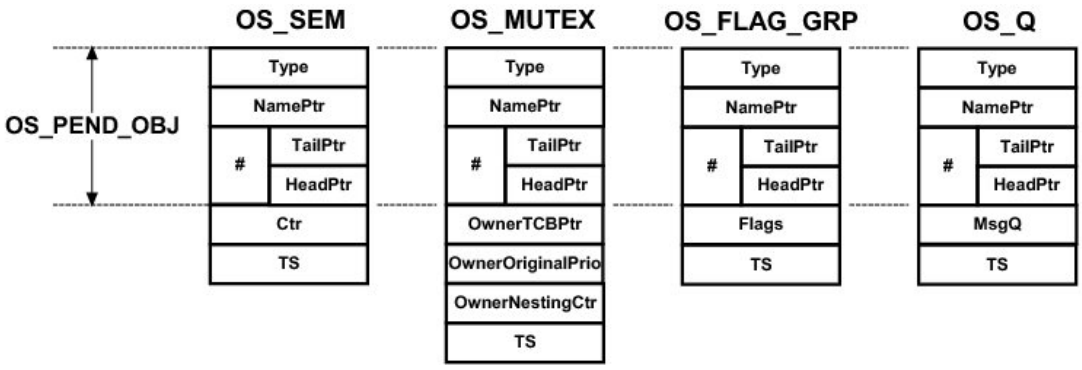
```
OS_FLAGS  OSFlagPend (OS_FLAG_GRP  *p_grp,
                      OS_FLAGS     flags,
                      OS_TICK      timeout,
                      OS_OPT       opt,
                      CPU_TS      *p_ts,
                      OS_ERR      *p_err)
```

opt

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



Function	Description
OS_PrioGetHighest()	Find the highest priority level
OS_PrioInsert()	Set bit corresponding to priority level in the bitmap table
OS_PrioRemove()	Clear bit corresponding to priority level in the bitmap table



Function	Description
`OS_PendListChangePrio()`	Change the priority of a task in a pend list
`OS_PendListInit()`	Initialize a pend list
`OS_PendListInsertPrio()`	Insert a task in priority order in the pend list
`OS_PendListRemove()`	Remove a task from a pend list

Formules du blocage:

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

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

Où:

1. ***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.**
2. **Lorsque qu'on a plusieurs usage(k,i) = 1 pour un même k, on prend le maximum**
3. **CS(*k*) le temps maximum requis pour passer au travers la section critique *k*.**
4. ***K* étant le nombre total de ressources (mutex)**

- **Avec ICPP on a:**

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

Où:

1. **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.**
2. **Lorsque qu'on a plusieurs usage(k,i) = 1 pour un même k, on prend le maximum**
3. **CS(k) le temps maximum requis pour passer au travers la section critique k.**
4. **K étant le nombre total de ressources (mutex)**

En résumé

Étapes d'une interruption du lab 1

- On s'intéresse ici à une interruption du PL au PS i.e. entre un périphériq
  (fit_timer_0, fit_timer_1 ou axi_gpio_0) et le Cortex A9 Core 1