Question 1 (3 pts) Répondez par vrai ou par faux et justifiez votre réponse en quelques lignes.

a) (.6 pt) Pour partager une section critique entre une tâche et un ISR ou entre 2 ISRs, la meilleure méthode d'exclusion mutuelle est de désactiver les interruptions avant d'accéder à la section de code critique et de les réactiver une fois sorti de la section de code critique.

Vrai car bien que l'option d'utiliser un mutex est la meilleure, dans un ISR cela est impossible à cause du blocage possible.

b) (.6 pt) Dans µCOS-III tous les changements de contexte se font avec la fonction *OSCtxSw()*.

Faux car pour le retour de(s) ISR(s) OSIntExit() utilise OSIntCtxSw.

c) (.6 pt) Une fonction *OSSemPend()* qui ne bloque pas lors de son appel (e.g. lorsque le sémaphore > 0) aura le même comportement qu'un appel à OSSemPend(&Sem,0, OS_OPT_PEND_NON_BLOCKING, &ts, &err).

Vrai si on exclue le traitement du message au retour de OSSemPend(&Sem,0, OS_OPT_PEND_NON_BLOCKING, &ts, &err) i.e. OS_ERR_PEND_WOULD_BLOCK et faux sinon. Il fallait bien justifier…

d) (.6 pt) La latence d'interruption est identique que le système soit en mode préemptif ou non préemptif.

Faux le mode préemptif est légèrement plus long à cause de l'appel à OSIntEnter() qui doit compter le nombre d'interruptions en cour.

e) (.6 pt) Le pseudo code de OSIntExit() est le suivant :

i. *désactive les interruptions*
ii. *décrémente la variable globale OSIntNesting*
iii. *si OSIntNesting égal 0, détermine la tâche "prête à rouler" la plus prioritaire en appelant OSPrioGetHighest()*
iv. *réactive les interruptions*

v. *On appelle OSCtxSw pour le changement de contexte*

- *Faux car à l'étape v on doit appeler OSIntCtxSw.*

Question 1 (3 points) En vrac

Pour chaque sous-question, expliquez en quelques lignes les différences entre:

a) Une *interruption logicielle* et une *interruption matérielle* (e.g. périphérique)

Interruption logicielle quand une tâche bloque (SemPend, MutexPend, etc.) donc interne

Interruption matérielle provient d'un périphérique externe.

b) Un *mutex* et un *sémaphore binaire*

Mutex pour l'exclusion mutuelle pour section critique avec héritage de priorité
Sémaphore binaire pour la synchronisation i.e. RDV

c) Un *flag* et un *sémaphore binaire*?

Un flag peut contenir plusieurs sémaphores binaires dans un même mot avec option consume ou pas quand on appel OSFlagPend.

Sémaphore binaire demande 1 mot donc moins compact et consomme toujours quand on appel OSSemPend

d) La fonction *OSQPend* et *OSTaskQPend*

OSQPend() lecture dans un FIFO externe pouvant être être partagée par plusieurs producteurs et consommateurs.
OSTaskQPend() lecture dans un FIFO de TCB pour une tâche données pouvant être être partagée par plusieurs producteurs, mais un seul consommateur.

e) La fonction *OSTimeDly* et un *watchdog timer*

Utilisée par une tâche pour réaliser un délai. À la fin du délai la tâche est à nouveau active.

Watchdog timer réalise un délai et quand celui arrive à expiration, il démarre une fonction de call back associée à une tâche spécifique (une seule et toujours la même).

f) *OSTimeDly(1000,OS_OPT_TIME_DLY, &err)* et
OSTimeDly(1000,OS_OPT_TIME_PERIODIC, &err)

Option OS_OPT_TIME_DLY cumule le délai mais aussi le temps d'exécution lorsqu'utilisée pour ordonnancer une tâche périodique.

Option OS_OPT_PERIODIC cumule uniquement le délai lorsqu'utilisée pour ordonnancer une tâche périodique.

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 1 (4 points) En vrac : vrai ou faux avec justification

a) (.5 pt) Dans les centrales nucléaires, le temps de réaction à un évènement (qui peut se traduire par une latence d'interruption) est généralement de l'ordre des millisecondes. Linux peut donc être utilisé car il ne s'agit pas d'un système temps réel.

Faux car on a besoin de déterminisme et contrainte dure

b) (.5 pt) Soit un convertisseur analogique-numérique qui fonctionne en mode périodique de manière très précise : un résultat de conversion est disponible à toutes les 1 ms exactement. L'utilisation d'un *watchdog* en uC/OS-III plutôt qu'une interruption externe est alors tout aussi efficace et ce peu importe la charge du système (c.-à-d. le nombre de tâches).

Faux car le watchdog de uC/OS-III utilise une tache lors du déclenchement de la fonction de call back. Or la priorité d'une tâches est toujours inférieure à celle d'un ISR.

c) (.5 pt) On put utiliser un mutex uC/OS-III pour faire une synchronisation pourvu que la valeur initiale demandée du sémaphore soit de 1.

Faux on ne peut séparer OSMutexPend() et OSMutexPost, i.e. fonctionne par pair

d) (.5 pt) Il est préférable de demander à une tâche de se supprimer elle-même plutôt que de la supprimer directement.

Vrai ça permet d'éviter les situation de deadlock si la tâche est en plein milieu d'un mutex, elle sort de la section critique et se termine ensuite.

e) (.5 pt) Une *pile d'exécution* contient des données d'une tâche alors que le TCB (Task Control Block) sert à enregistrer des informations au sujet des tâches actives dans un programme.

Faux c'est l'inverse

f) (.5 pt) Un des désavantages de uC/OS-III est d'être limité à 64 tâches.

Faux, les structure de données de uC/OS-III (e.g. OSPrioTbl) sont extensibles à 128, 256 et même 512 tâches.

(suite page suivante)

g) (1 pt) Dans le code de la figure suivante vue en classe, si au lieu d'utiliser un sémaphore pour réaliser le rendez-vous entre *TaskDevFifo* et *TaskDrivFifo* on utilisait le flag de uC/OS-III, il ne serait alors plus nécessaire d'avoir une fonction dont le rôle est d'empêcher *TaskDrivFifo* de lire plusieurs fois une même valeur (c'est-à-dire équivalent à *OSSemPend(&Irq1, 0, OS_OPT_PEND_NON_BLOCKING, &ts, &err)*.

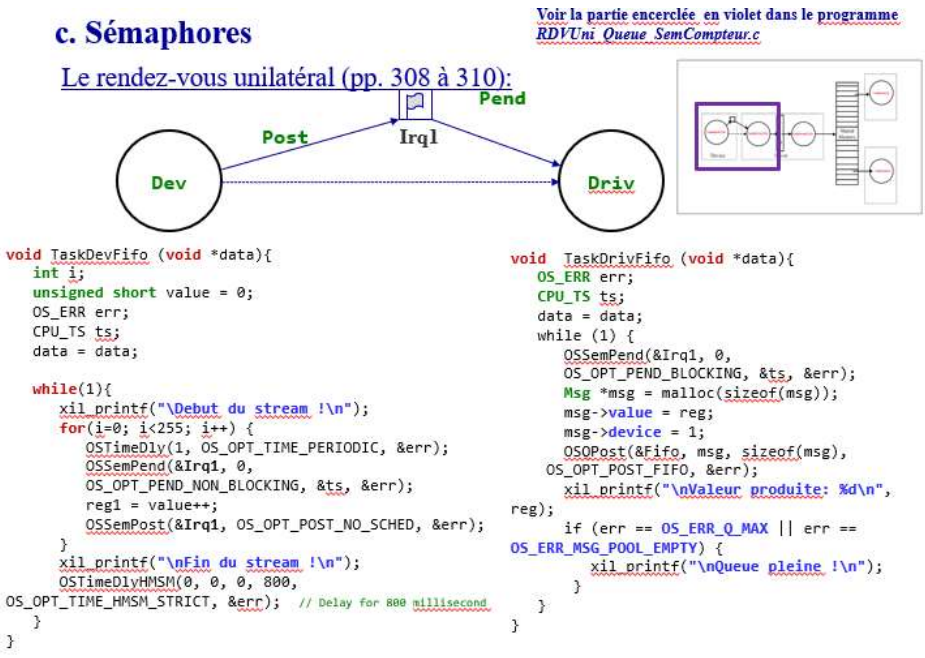

Figure 1.1 pour la question 1g.

Vrai car par définition si dans le OSFlagPost de TaskDevFifo on fait un OS_OPT_POST_FLAG_SET on met le bit à 1 et si on le refait plus tard une 2e fois (sans qu'il y est eu consommation de TaskDrivFifo avec OSFlagPend) le bit reste 1. Autrement dit c'est comme si on avait mod 1. N.B. Il ne faut pas oublier de faire un consume dans le OSFlagPend…

En résumé:

- *Paramètre du côté du producteur TaskDevFifo :*

OS_OPT_POST_FLAG_SET_ALL

- *Paramètre du côté du consommateur TaskDrivFifo :*

OS_OPT_PEND_FLAG_SET_ALL + OS_OPT_PEND_BLOCKING + OS_OPT_PEND_FLAG_CONSUME

Question 1 (3.5 points) En vrac : vrai ou faux avec justification

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.

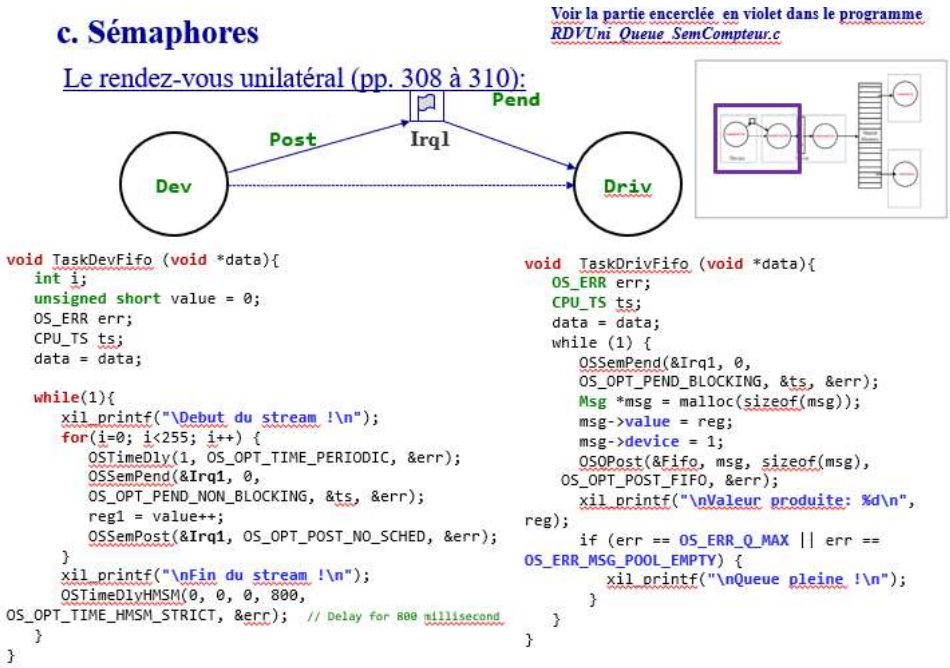

Figure 1.1 pour la question 1c.

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 1 (3.5 points) En vrac : vrai ou faux avec justification

a) (.6 pt) Lorsqu'on compare le protocole héritage de priorité avec celui de ICPP, on remarque qu'en général ICPP minimise davantage les préemptions (c.-à-d. le fait qu'une tâche puisse être arrêtée par une tâche plus prioritaire qu'elle-même).

Faux, on dit que ICPP minimise davantage le temps de blocage par rapport à l'héritage de priorité mais pas le temps des interférences. Si vous enlevez complètement le temps des blocages il restera toujours le temps des interférences. Notez aussi que ICPP minimise les changements de contexte par rapport à l'héritage de priorité mais il y a une différence entre les changements de contexte et les interférences.

b) (.6 pt) La période d'horloge du processeur est en général une valeur 100,000 à 1 million de fois plus petites que le pas d'ordonnancement (tick), ce qui implique que de manière générale que le tick d'horloge est de l'ordre des *ms*, la période la minuterie est en *us* et la fréquence du processeur est en *ns* ou moins.

Vrai, ça été dit en classe, en général des ticks de l'ordre de la us et moins ont tendance à provoquer trop d'interruptions de la minuterie. Donc pour avoir des ticks dans l'ordre de la ms (1000 Hz) on doit avoir une minuterie qui va plus vite donc dans les MHz. Finalement, on a le processeur qui lui roule en GHz.

c) (.6 pt) Soit les 2 appels de fonctions suivantes :

OSSemPend(&Sem, 0, OS_OPT_PEND_BLOCKING, &ts, &err); **(1)**

OSSemPend(&Sem, 0, OS_OPT_PEND_NON_BLOCKING, &ts, &err); **(2)**

Lorsque (1) ne bloque pas lors de son appel (p. ex. sémaphore > 0), il aura alors le même comportement que (2).

Vrai et faux (j'ai accepté les 2 en autant que la justification était appropriée).

Pourquoi vrai : même comportement au sens non bloquant si sem > 0.

Pourquoi faux : si on tient compte du fait qu'il y a une différence avec la gestion d'erreur pour (2)

d) (.6 pt) Lorsqu'une ressource est partagée entre deux sous-routines d'interruption, le mécanisme le plus approprié est de désactiver l'ordonnanceur avant chaque accès à la ressource.

Faux, dans le contexte de 2 ISRs il faut avoir recours à la désactivation des interruptions

e) (1.1 point) Soit une application composée de 4 tâches T4 à T1 (en ordre décroissant de priorité) utilisant trois mutex M1, M2 et M3 :

	M1	M2	M3
T4	5	2	0
T3	0	7	1
T2	6	5	0
T1	7	8	9

Dans le cas de l'utilisation de l'utilisation de ICPP, le temps maximal de blocage pour chaque tâche est : B4 = 8, B3 = 9, B2 = 9, B1 = 0.

Si on se concentre sur M1 pour T4, on a 2 sources de blocage B4 pour T4

-T2 peut bloquer T4 par 6 ticks

-T1 peut bloquer T4 par 7 ticks

-Comme T2 et T1 ne peuvent utiliser le mutex au même moment, on prend le maximum

Conclusion pour M1 : il contribue au blocage de T4 pour un maximum de max(6, 7) = 7 ticks

Si on se concentre sur M2 pour T4, on a 3 sources de blocage B4 pour T4

-T3 peut bloquer T4 par 7 ticks

-T2 peut bloquer T4 par 5 ticks

-T1 peut bloquer T4 par 8 ticks

Conclusion pour M2: il contribue au blocage B4 pour un maximum de max(7, 5 ,8) = 8 ticks

Comme M4 n'utilise pas M3, le blocage de T4 ne vient que de M1 et M2. Or, comme on parle de ICPP on prend le maximum entre les 2 blocages calculés ci-haut:

Conclusion : B4 = max(7,8) = 8 OK

De la même manière que pour B1, calculons B3:

B3 = max(max(5, 8), max(9)) = 9 donc OK

De la même manière que pour B1, calculons B2:

B2 = max(max(7), max(8)) = 8 différent de 9 donc la réponse est fausse

P.S. T1 ne peut jamais bloqué donc T1 = 0 donc OK.

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.