ECOLE POLYTECHNIQUE DE MONTREAL

INF3610 : Laboratoire 1 Partie 1

Introduction à µC-II/OS

Arnaud Desaulty

9/4/2015

I- Introduction

µC-II/OS est un système d'exploitation temps réel préemptif. En tant que tel, il permet entre autre :

- La création de multiples tâches possédant chacune une priorité, un ensemble de registres processeurs ainsi qu'une portion de la pile
- La préemption de tâches de priorité inférieures
- La gestion d'événements asynchrones par des routines d'interruption

L'intérêt d'un tel système repose sur plusieurs avantages :

+ Les événements critiques sont gérés le plus rapidement possible
+ Le processus de développement est simplifié par la séparation des différentes fonctionnalités en tâches
+ La possibilité d'étendre son logiciel sans changement majeur de celui-ci
+ La disponibilité de nombreux services (sémaphores, files, mailboxes, etc...) permettant un meilleur usage des ressources

Le but de ce laboratoire est de se familiariser avec les différents services mis à votre disposition par µC-II. A la fin de ce laboratoire, vous aurez une vue d'ensemble de comment utiliser un système d'exploitation temps réel. Les connaissances acquises vous serviront aussi lors de la deuxième partie du laboratoire qui sera plus complexe et applicative.

Sans plus tarder, entrons dans les différentes caractéristiques de µC.

ATTENTION

Légende utilisée au cours de ce laboratoire :

- **Le texte en gras** représente des éléments de cours qui doivent être compris pour répondre aux exercices
- Les lignes précédées d'une lettre minuscule (a.) représentent la progression conseillée dans l'exercice
- **Le texte en vert** représente les questions auxquelles vous devrez répondre textuellement dans vos rapports
- **Le texte en rouge** représente des consignes à suivre pour assurer le bon fonctionnement des exercices. **Un non-respect de ces consignes entrainera des pertes de points sévères.**
- Le texte bleu souligné représente des liens vers les ressources disponibles sur Moodle. Il suffit de Ctrl+click sur ce texte pour y accéder.

II- Exercices

1- Création de tâches, priorités et démarrage de l'OS

MicroC organise son code sous forme de tâches. Chaque tâche maintient son propre set de registre et son état. Ainsi toutes ces tâches sont en concurrence pour le temps processeur. Contrairement à un OS classique ou chaque tâche est alloué une slice de temps pour avancer dans son exécution, MicroC possède un ordonnanceur qui a pour but de sélectionner quelle tâche va s'exécuter.

Pour l'aider dans sa décision l'utilisateur doit définir des priorités à assigner à ses tâches. Ainsi, dans µC, si plusieurs tâches sont en concurrences pour le temps processeur, l'ordonnanceur choisira la tâche la plus prioritaire (dans MicroC cela correspond à la priorité la plus proche de 0).

Une fois qu'une tâche a acquis le droit d'exécution, elle s'exécute sans discontinuer jusqu'à ce qu'elle décide de se mettre en pause pour une raison ou une autre (pause de la tâche, attente d'une variable) ou bien qu'une interruption survient.

L'ordonnanceur doit alors choisir quelle est la nouvelle tâche la plus prioritaire prête à s'exécuter puis lui donne le droit de s'exécuter.

A noter que l'ordonnanceur n'est appelé pour décider de la nouvelle tâche à exécuter qu'à des moments bien précis (par exemple lorsqu'une tâche sort d'une pause ou bien qu'une variable est libérée). Si aucun éléments de ce genre n'est présent, l'OS peut très bien bloquer sur une tâche qu'il va exécuter à l'infini !

Exercice 1

Dans ce premier exercice, vous allez devoir utiliser les fonctions de création de tâches pour créer les tâches puis démarrer l'OS. Le but final est d'obtenir cette trace :

```
End of task creation !
Task priorities
are an
important
feature
of MicroC-II !
```

a. Utilisez la fonction `OSInit()` avant d'utiliser n'importe quel autre service de µC
b. Créez les tâches dans la fonction main à l'aide de la fonction `OSTaskCreate()`. Vous trouverez les informations sur les paramètres à fournir à cette fonction en cliquant sur le nom de la fonction.
c. Une fois toutes les tâches créées, utilisez la fonction `OSStart()` pour démarrer l'OS
d. Faites tourner votre programme une première fois. **Qu'observez-vous ?**
e. Modifiez le code jusqu'à obtenir la trace attendue

Veuillez respecter les consignes suivantes lors de cet exercice :

- **Ne pas modifier le code des tâches pour cet exercice**
- **Attention à gérer les cas d'erreurs de manière sommaire (si vous rencontrez une erreur, imprimez un message d'erreur. Pas la peine de faire un message personnalisé en fonction de l'erreur)**

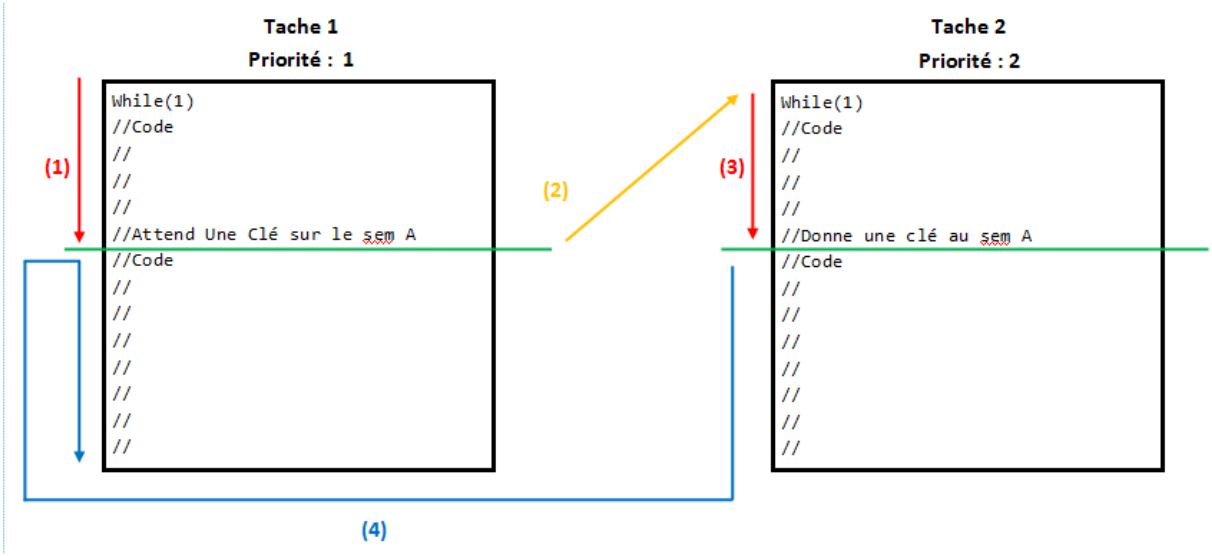
2- Éléments de synchronisation, d'exclusion et variables partagées

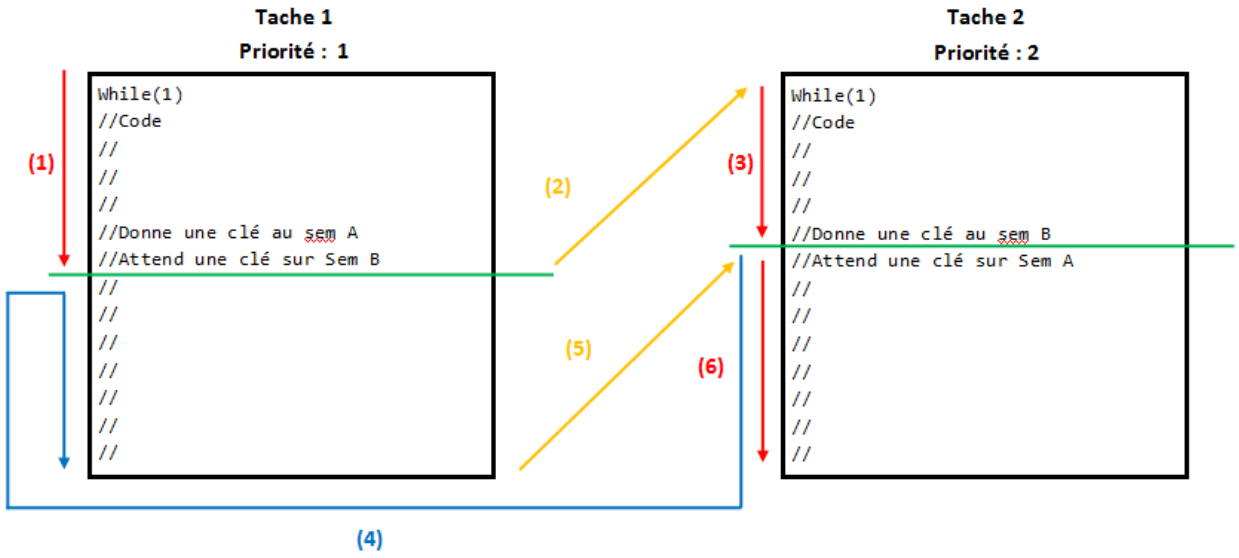
a. Sémaphores

Nous savons maintenant créer des tâches et les organiser les unes par rapport aux autres par rapport à leur priorité. Toutefois, µC fournit un mécanisme offrant plus de possibilités qui vient s'ajouter par-dessus le système de priorité : Les sémaphores. On pourrait assimiler les sémaphores à des barrières. Lorsque votre code rencontre un sémaphore, s'il reste des clés, le sémaphore est passé et l'exécution continue normalement. Sinon, l'exécution de la tâche s'arrête temporairement, jusqu'à ce que ce sémaphore reçoive une clé supplémentaire.

Les fonctions à utiliser sur ces sémaphores sont nombreuses mais les trois principales sont OSSemCreate() qui permet de créer un sémaphore et de définir le nombres de clés présentes au début, OSSemPend() qui agit comme une barrière qui ne peut être passée que s'il reste une clé au moins dans le sémaphore (une clé sera alors consommée) et OSSemPost() qui permet de rajouter une clé dans un sémaphore. Il existe d'autres fonctions affiliés aux sémaphores que vous pouvez découvrir en allant dans le manuel utilisateur (toutes ces fonctions commencent par OSSem*).

Mais à quoi peuvent bien servir ces barrières ? Deux des utilisations les plus classiques sont les rendez-vous unilatéraux et les rendez-vous bilatéraux. Pour le premier, il s'agit tout simplement d'avoir une tâche qui attend sur un sémaphore à 0 clés, puis de libérer une clé depuis une autre tâche, ce qui aura pour effet de permettre à la tâche bloqué de s'exécuter. Cela peut être utile pour forcer des tâches à s'exécuter en séquence. Dans le cas du rendez-vous bilatéral, la situation est un peu plus complexe. Une première tâche A libère une clé sur un sémaphore 1 puis attend sur un sémaphore 2, alors qu'une seconde tâche fait l'inverse : elle libère une clé sur le sémaphore 2 puis attend sur le sémaphore 1. Cela aura pour effet de forcer les deux tâches à commencer ou terminer un segment de code au même moment.



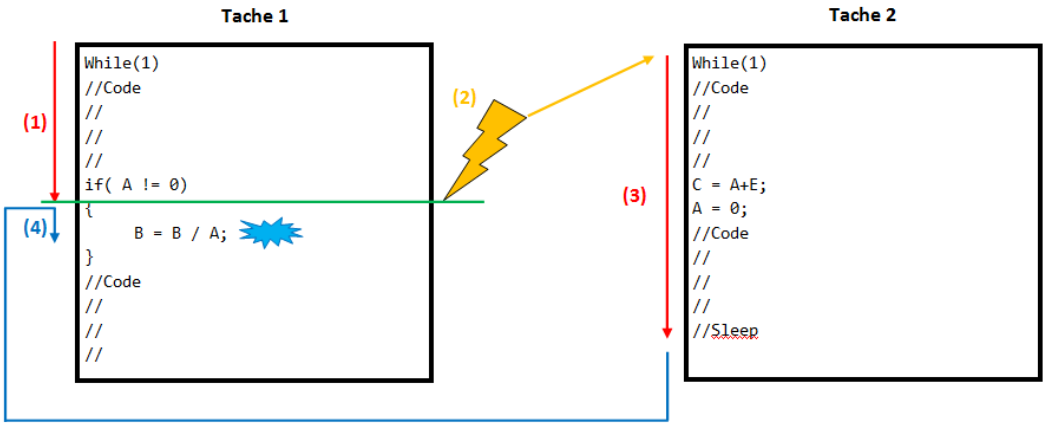


b. Mutex (Mutual Exclusion)

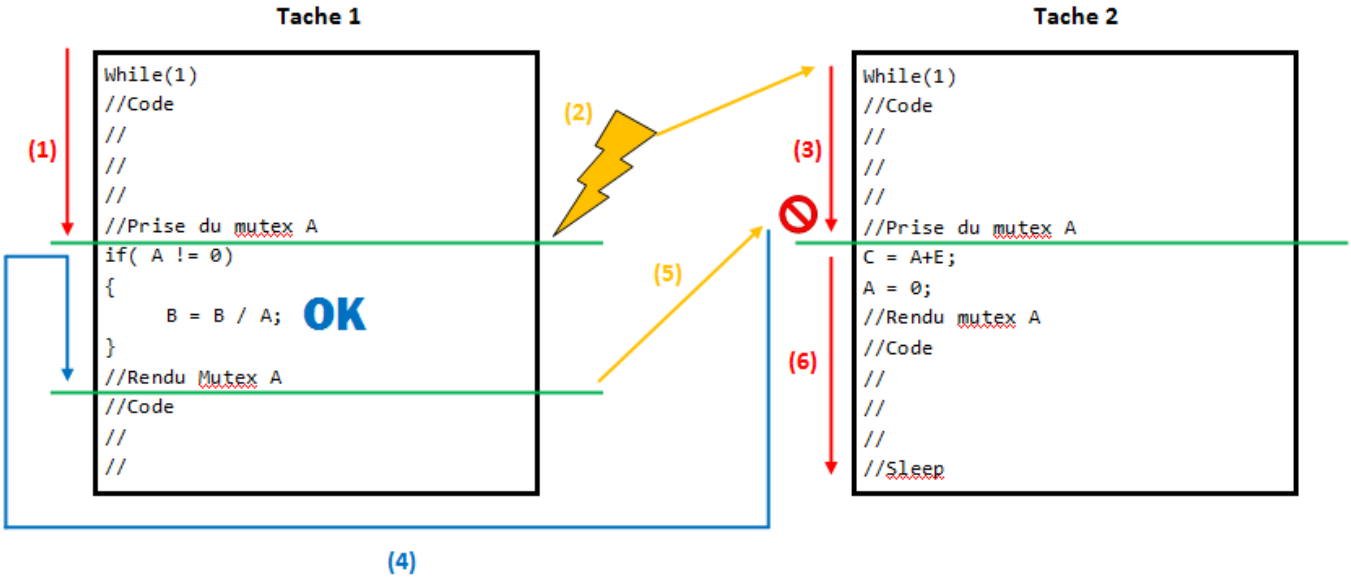
Un autre service proposé par l'OS porte le nom glamour de mutex, pour exclusion mutuelle. Le mutex possède un fonctionnement similaire au sémaphore, mais son utilisation est différente. Tout d'abord, un mutex ne peut pas posséder plus d'une clé. De plus, le mutex est toujours créé avec une clé de disponible au démarrage. Ces deux différences font du mutex un élément de blocage.

Comme pour le sémaphore les fonctions principales du mutex sont OSMutexCreate(), OSMutexPost() et OSMutexPend(). Toutes les fonctions relatives aux mutex sont répertoriées dans le manuel utilisateur avec le préfixe OSMutex*.

Comment se servir de ces mutex ? L'utilisation unique du mutex est de protéger des segments de code d'une intrusion qui pourrait corrompre le segment de code. Pour expliquer cela, un exemple : Imaginons que deux tâches différentes travaillent sur une même variable A. La première tâche fait un test sur A (1) pour vérifier que cette variable est différente de 0 (pour effectuer une division), et valide le test. Une interruption survient (2) et réveille la seconde tâche qui est plus prioritaire. Celle-ci fait des opérations sur A et la rend égale à zéro (3). La seconde tâche ayant terminé son travail se rendort et l'ordonnanceur rend la main à la première tâche. Celle-ci reprend où elle s'était arrêtée (4), c'est à dire après le test. Elle tente alors de diviser par A mais fait planter le programme à cause d'une division par zéro.



Cela aurait pu être évité en utilisant un mutex. Avant de faire son test sur A la première tâche essaye d'acquérir le mutex en faisant OSMutexPend() (1). Puisque le mutex possède une clé de base, cela fonctionne. L'interruption survient (2) et la seconde tâche démarre. Toutefois, elle aussi, avant de faire son traitement sur A, essaie d'acquérir le mutex (OSMutexPend()) (3). Cette fois-ci, cela n'est pas possible puisque la première tâche possède déjà l'unique clé. La seconde tâche rend donc la main à la première (4), qui peut terminer sa division sans problème. A la fin de cette section de code, la première tâche relâche la clé à l'aide de OSMutexPost(). Cela permet à la seconde tâche, qui attendait cette clé de passer en exécution (5). A la fin de son traitement sur A, cette tâcher devra elle aussi libérer la clé (6). Ces deux sections, entourées par la paire OSMutexPend()/OSMutexPost(), sont appelés sections critiques et doivent être mutuellement exclusives pour éviter des problèmes de corruption de données, d'où l'appellation mutex.



Exercice 2

Le code fourni dans l'exercice 2 est très simple. La tâche 1 et la tâche 2 ont un message à imprimer. Ce message est envoyé caractère par caractère à une troisième tâche qui a pour but d'afficher ces messages. Entre chaque caractère, une petite attente de 3 ou 2 ticks de l'OS est effectué pour simuler les délais de traitement respectifs des tâches 1 et 2. Une fois un temps limite passé, la tâche 3 imprime tout ce qui est dans son buffer.

Pour cet exercice, encore une fois, le but est d'obtenir la trace suivante :

```
End of task creation !
TACHE 1 @ 1 : Debut tache 1.
T1 : i = 0
TACHE 2 @ 1 : Debut tache 2.
TACHE 3 @ 1 : Debut tache 3.
T1 : i = 1
T1 : i = 2
T1 : i = 3
T1 : i = 4
T1 : i = 5
T1 : i = 6
T1 : i = 7
T1 : i = 8
T1 : i = 9
T1 : i = 10
TACHE 1 @ 34 : Suspension.
T2 : i = 0
T2 : i = 1
T2 : i = 2
T2 : i = 3
T2 : i = 4
T2 : i = 5
T2 : i = 6
T2 : i = 7
T2 : i = 8
T2 : i = 9
T2 : i = 10
T2 : i = 11
T2 : i = 12
T2 : i = 13
T2 : i = 14
T2 : i = 15
TACHE 2 @ 66 : Suspension.
TACHE 3 @ 71 : Fin du delay .
I am task 1 and i am task 2
```

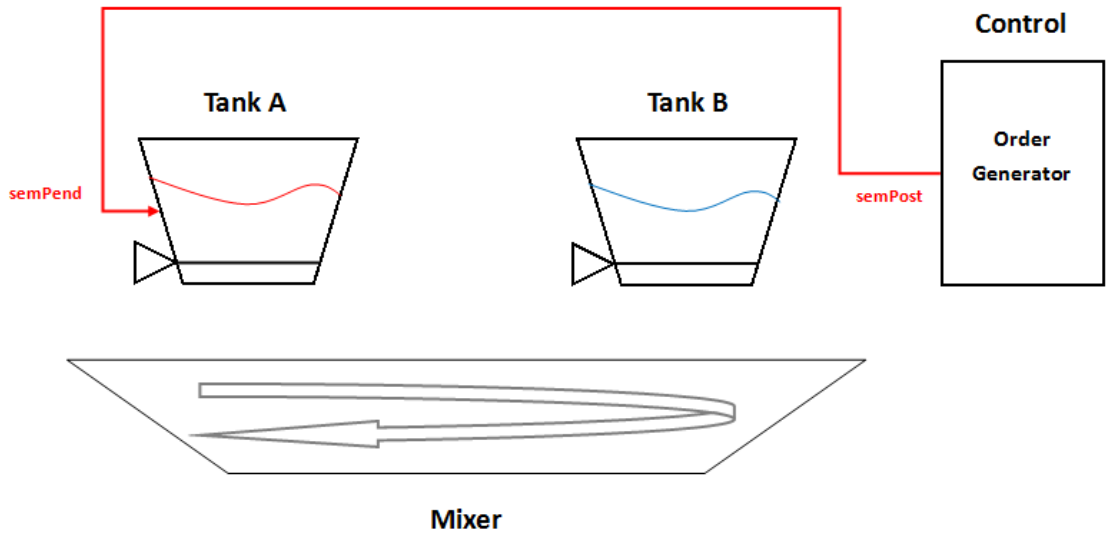
a. Créez les tâches 1, 2 et 3 en modifiant le main.
b. Faites fonctionner le code et observez la trace obtenue. **Quel est le problème ?**
c. Modifiez le code des tâches 1 et 2 à l'aide d'un des éléments vu ci-dessus dans la section 2 afin de corriger le comportement du code.

Veuillez respecter les consignes suivantes lors de cet exercice :

- **Ne pas modifier les priorités des tâches**
- **Ne pas modifier le comportement de la tâche printer**
- **Attention à gérer les cas d'erreurs de manière sommaire (si vous rencontrez une erreur, imprimez un message d'erreur. Pas la peine de faire un message personnalisé en fonction de l'erreur)**

Exercice 3

Le code de l'exercice 3 modélise un système temps réel de mélange de peinture tel que décrit sur le schéma suivant :



Le module de contrôle génère des commandes de peintures à des intervalles de temps aléatoires et le reste du système doit répondre à ces commandes le plus vite possible.

Le flot de commande est tel que décrit ci-après :

-Dès qu'une commande est prête, le control libère une clé sur le sémaphore qu'attend le réservoir A (créant ainsi un rendez-vous unilatéral). Le réservoir A et le réservoir B doivent alors démarrer en même temps **si le mixer a fini la commande précédente** (protocole à définir à l'aide des éléments décrits dans la section 2). Les distributions des réservoirs A et B sont représentées par des délais respectifs de 300 et 700 cycles (ces délais sont créés à l'aide de la fonction **OSTimeDly()**).

-Une fois que les deux réservoirs ont terminés de distribuer leur pigment, le mixer se réveille et commence le mélangeage. Une fois le mélangeage terminé, la cuve du mixer est évacué et le cycle peut recommencer. Le malaxage ainsi que l'évacuation sont simulés par un délai de 1000 cycles.

a. Créez les tâches controller, tank_a, tank_b et mixer dans le main ainsi que le premier sémaphore sem_controller_to_tank_A.
b. Rajoutez les éléments de synchronisation nécessaires au bon fonctionnement du système au cours des dix commandes du contrôleur.

Veuillez respecter les consignes suivantes lors de cet exercice :

- **Ne pas modifier les priorités des tâches**
- **Ne pas créer plus de 5 sémaphores**
- **Aucun sémaphore à part le sémaphore sem_controller_to_tank_A n'est autorisé à accumuler plus d'une clé au cours de l'exécution**
- **Attention à gérer les cas d'erreurs de manière sommaire (si vous rencontrez une erreur, imprimez un message d'erreur. Pas la peine de faire un message personnalisé en fonction de l'erreur)**

3- Éléments de communication

Il est parfois nécessaire dans un système temps réel, de passer des informations d'une tâche à une autre sans passer par une variable partagé. Dans ce but, MicroC propose plusieurs éléments permettant de communiquer d'une tâche à l'autre. Nous allons voir l'un des plus utilisé, la file de communication.

Une file de communication permet le stockage et la distribution de message d'une tâche productrice à une tâche consommatrice. Son fonctionnement est simple. Vous devez d'abord créer la file en spécifiant la taille de celle-ci à l'aide de OSQCreate() . Vous aurez donc besoin d'un espace alloué statiquement ou dynamiquement de la même taille que votre file. Il est important de noter que la file véhicule des pointeurs sur void.

Une fois la file créée, vous pouvez utiliser une des nombreuses fonctions disponible sur cette file. Les plus utilisées étant OSQPost() qui permet de poster un message, et OSQPend() qui bloque la tâche jusqu'à ce qu'un message soit disponible dans la file. On notera aussi l'existence de OSQAccept(), qui teste si un message est dans la file mais qui ne bloque pas si jamais aucun message n'est présent. Le reste des fonctions est disponible sous le préfixe OSQ*.

Exercice 4

Le code de l'exercice 4 est une reprise du code de l'exercice 3. Toutefois, notre contrôleur est maintenant capable de prendre des commandes personnalisées. Il génère donc aléatoirement le temps de distribution de A et B ainsi que le temps de mélange du mixer.

a. Reprenez votre code de l'exercice 3 et utilisez des files afin d'acheminer les informations générées dans le contrôleur vers les tâches impliquées. **Quelle est la taille minimale des files pour être certains de ne perdre aucune commande lors de l'exécution des tâches.**

Veuillez respecter les consignes suivantes lors de cet exercice :

- **Ne pas modifier les priorités des tâches**
- **Ne pas utiliser plus de 3 files**
- **Attention à ne pas dupliquer les synchronisations (les files peuvent servir d'éléments de synchronisation dans certains cas)**

4 - Questions supplémentaires

a. **D'après le fonctionnement d'un sémaphore et d'un mutex, est-il possible de modéliser un mutex à l'aide d'un sémaphore ? Expliquez votre choix.**
b. **Donnez une brève appréciation de ce laboratoire (temps nécéssaire, difficulté, etc…)**

III- Barème et rendu

A l'issue de ce laboratoire vous devrez remettre sur moodle, une fois par groupe de 2, une archive respectant la convention INF3610Lab1Part1_matricule1_matricule2.zip contenant :

- Dans un dossier src, le code de vos 4 fichiers exo1.c, exo2.c, exo3.c et exo4.c
- A la racine, un bref rapport contenant les réponses aux questions du laboratoire

Vous devez rendre ce laboratoire au plus tard la veille du prochain laboratoire à minuit (soit 2 semaines après le premier laboratoire)

Barème	
Exécution du code	
Exo1	/3
Exo2	/3
Exo3	/5
Exo4	/4
Réponse aux questions	
Exo1 d.	/1
Exo2 b.	/1
Exo4 a.	/1
Questions supplémentaires a.	/2
Questions supplémentaires b.	/0
Respect des consignes	
Entraine des points négatifs (peut aussi invalider les points d'un exercice)	
TOTAL	/20

IV- Conclusion

Au cours de ce laboratoire, vous aurez eu l'occasion de vous familiariser avec l'OS temps réel MicroC. Ce premier laboratoire vous permettra d'être plus à l'aise avec les services fournis par cet OS lors du laboratoire 2, qui sera plus conséquent.