ÉCOLE POLYTECHNIQUE DE MONTREAL

INF3610 : Laboratoire 1

Introduction à µC-II/OS

Frédéric Fortier
Eva Terriault
avec portions théoriques par
Arnaud Desaulty

8/18/2017

I- Introduction

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

- 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, et
- 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, et
+ 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 du laboratoire 2 qui sera plus complexe et applicatif.

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+clic sur ce texte pour y accéder.

II- Exercices

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

µC organise son code sous forme de tâches. Chaque tâche maintient son propre ensemble de registres et son état. Ainsi, toutes ces tâches sont en concurrence pour l'accès au processeur. Contrairement à un OS classique ou chaque tâche est alloué une slice de temps pour avancer dans son exécution, µC 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 µC cela correspond à la priorité la plus basse, 0 étant la plus grande priorité).

Une fois qu'une tâche a acquis le droit d'exécution, elle s'exécute sans discontinuer jusqu'à ce qu'elle soit mise en pause pour une raison ou une autre (pause de la tâche, impossibilité d'accéder à une section critique, etc.) ou bien qu'une interruption survient.

L'ordonnanceur doit alors choisir la nouvelle tâche la plus prioritaire prête à s'exécuter, puis une fois déterminer ce même ordonnanceur donnera à cette tâche 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 lors d'une fin d'interruption de la minuterie, lorsqu'une tâche sort d'une pause, lorsque qu'une tâche sort d'une section critique, etc.). Si aucun élément de ce genre n'est présent, une même tâche pourrait s'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 :

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

a. Utilisez la fonction `OSInit()`[1] 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! 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)**

[1] Toutes les fonctions µC qui suivent dans ce laboratoire (i.e. commençant par OS) sont clairement définies dans le manuel de référence se trouvant sur le site web du cours (Section 2, sous-section 5, API de µC). Prenez le temps de télécharger le fichier pdf et de le regarder…

- **Faites attention lorsque vous passez les tableaux d' OS_STK lors de l'appel OSTaskCreate() : vous devez passer l'adresse du début du stack. Or, le stack évolue de l'adresse la plus haute à l'adresse la plus basse : vous devez donc passer la fin des tableaux, pas le début (des explications plus détaillées seront données pendant la période de laboratoire).**

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 pour ouvrir la barrière, 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 devenue disponible.

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 affiliées 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 (Figure 1) et les rendez-vous bilatéraux (Figure 2). Pour le premier, il s'agit tout simplement d'avoir une tâche qui attend sur un sémaphore à 0 clé, 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. Cette fonctionalité peut être utile dans certains procédés industriels.

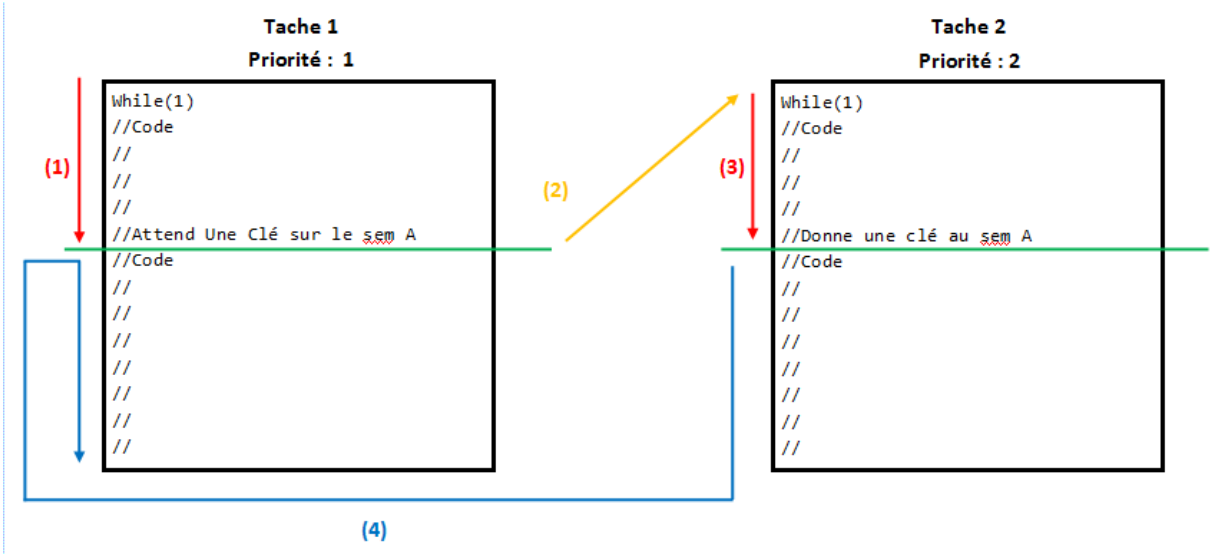


Figure 1. Rendez-vous unilatéral

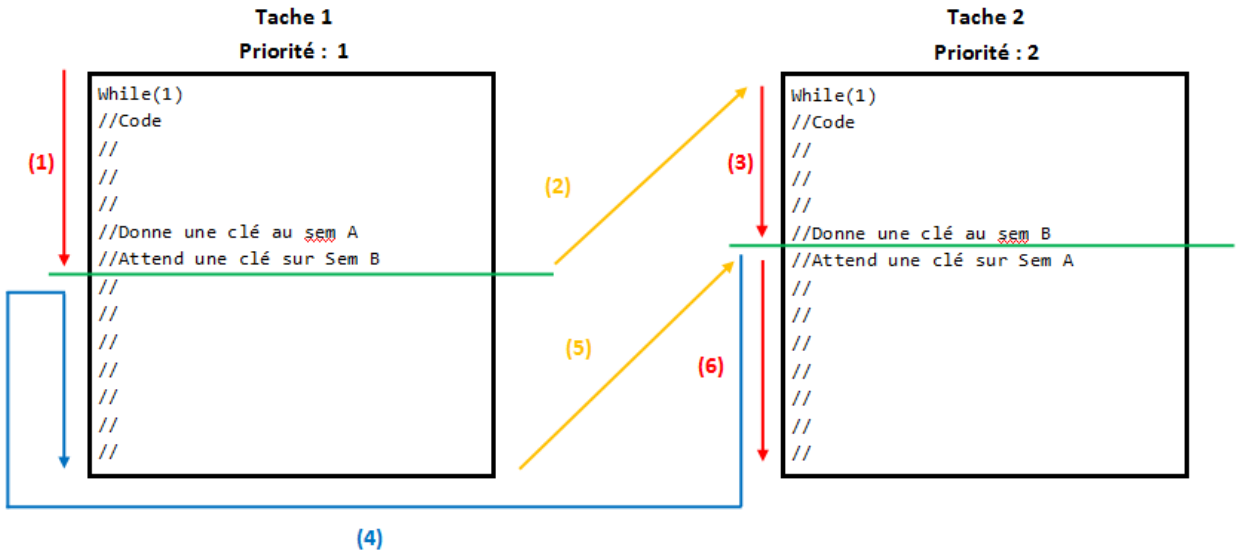


Figure 2. Rendez-vous bilatéral

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 les données d'un code d'exécution. Pour expliquer cela, un exemple (Figure 3): 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.

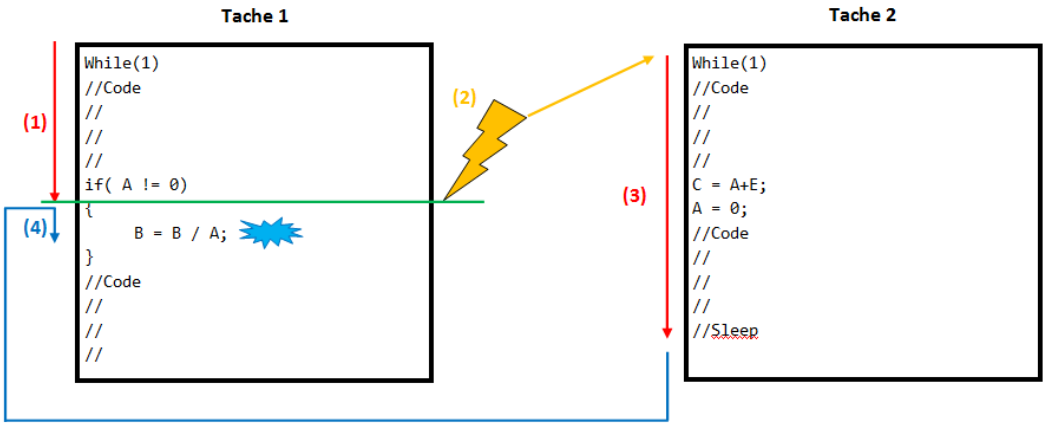


Figure 3. Corruption de données

Cela aurait pu être évité en utilisant un mutex (Figure 4). 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 tâche 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.

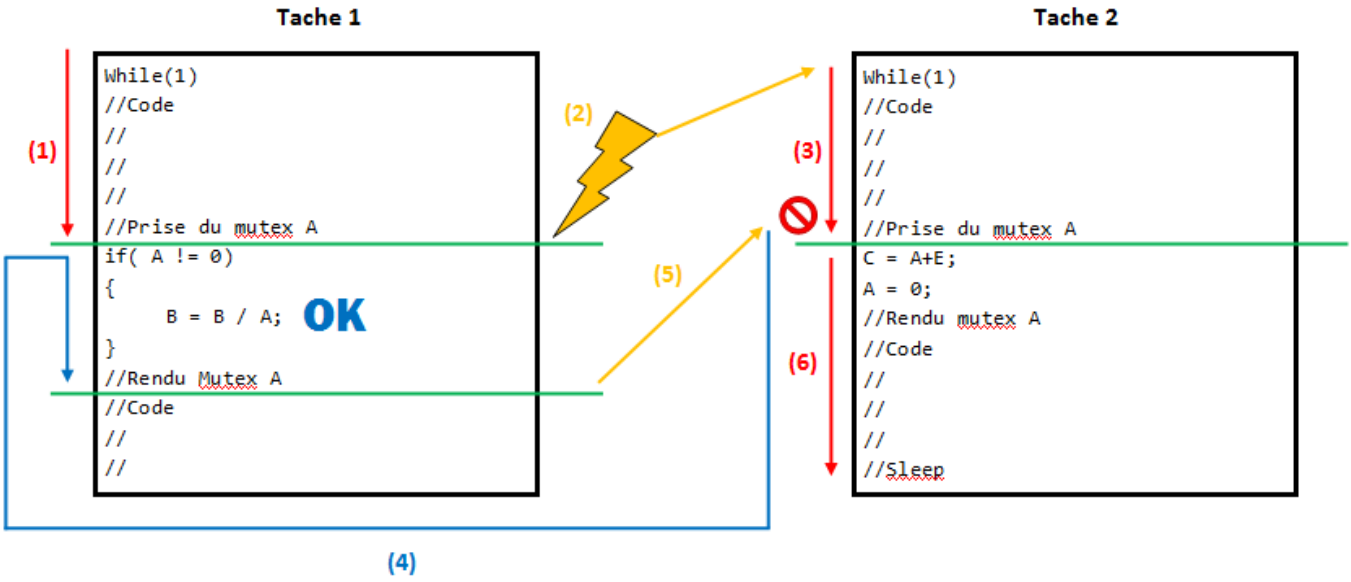


Figure 4. Corruption de données évitée grâce au mutex

c. Mutex vs. Sémaphore binaire

À première vue, il peut sembler ne pas y avoir de différence entre un mutex et un sémaphore dont on limiterait le nombre de clés à 1. Cependant, lorsque l'on regarde ces deux services de plus près, on remarque que OSMutexCreate(), contrairement à OSSemCreate(), prend en paramètre une priorité, un peu comme une tâche. Il en est ainsi dans le but de pallier au phénomène d'*inversion de priorités* (Figure 5), qui se produit lorsqu'une tâche de faible priorité possède un mutex et qu'elle se fait préempter par une autre tâche de priorité moyenne, empêchant la première de libérer le mutex qu'une troisième tâche de priorité plus haute voudrait acquérir et bloquant celle-ci:

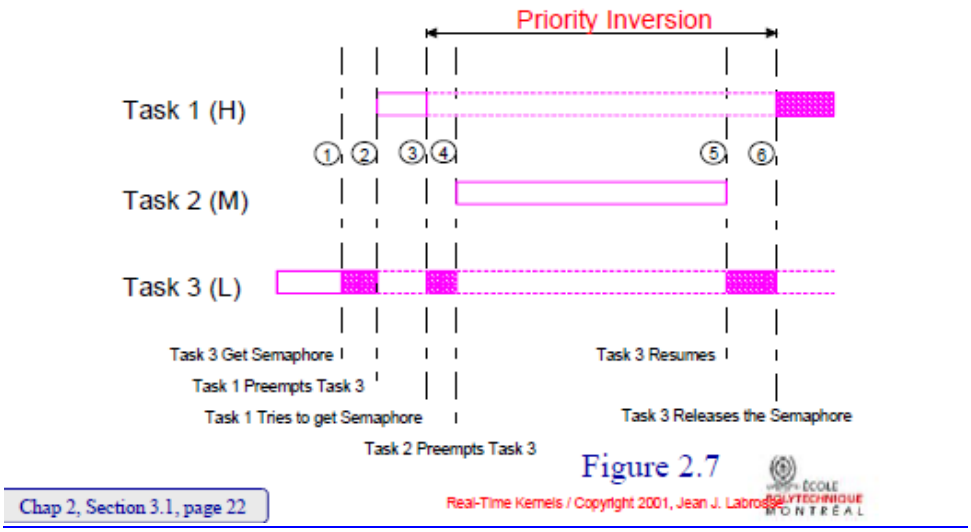


Figure 5. Phénomène d'inversion de priorité

µC-II règle le problème en augmentant temporairement la priorité de la tâche possédant le mutex vers la priorité assignée au mutex à sa création lorsqu'une tâche plus prioritaire que la première essaie aussi d'avoir le mutex, mais ne le fait pas pour une sémaphore, puisque celle-ci sert à la synchronisation et non à l'exclusion mutuelle.

Le problème d'inversion de priorités et les différents mécanismes d'*héritage de priorité* utilisés pour régler ce problème seront vus plus en détails en cours dans les prochaines semaines et ne devraient pas être un problème dans ce laboratoire. Cependant, comme cela reste une très bonne pratique d'utiliser les sémaphores et les mutex pour ce pourquoi ils sont conçus (soit respectivement la synchronisation et l'exclusion mutuelle), l'utilisation d'un mutex lorsqu'un sémaphore aurait été plus approprié (et vice versa) sera pénalisée.

Exercice 2a

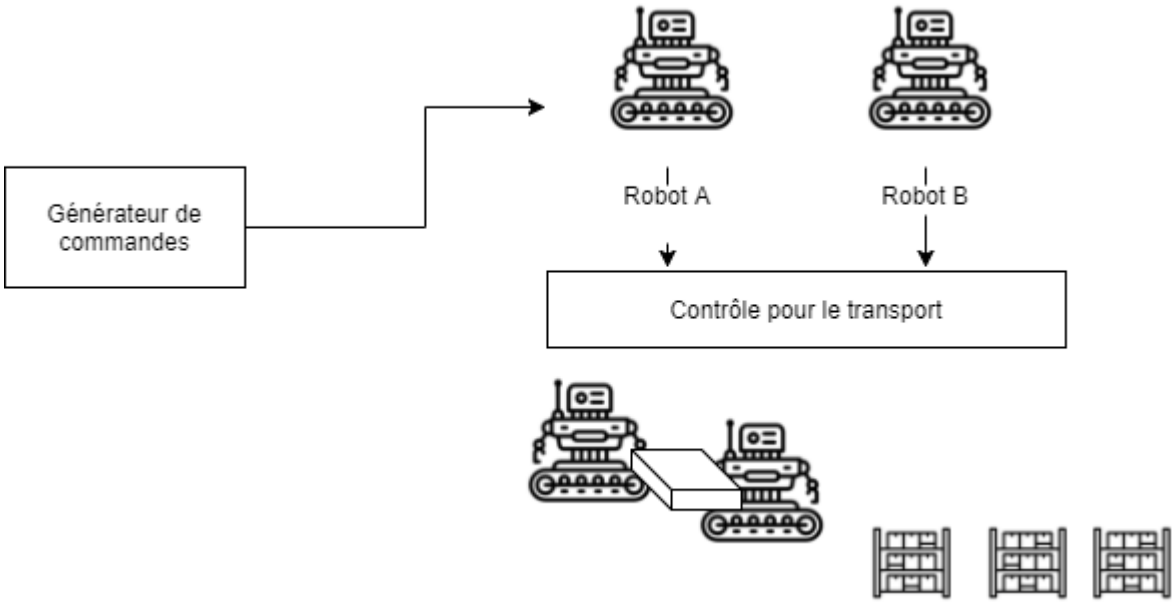
Le code de l'exercice 2 modélise un système temps réel de deux robots travaillant dans un centre de distribution. La tâche des deux robots est de transporter des objets à deux. Nous nous pencherons seulement sur la communication générale du système, sans toucher aux défis et subtilités qu'amène la synchronisation des robots lors du transport en tant que tel.

Le système peut avoir deux modes, soit le mode préparation et le mode transport.

- Mode préparation : Le robot se prépare à faire une livraison. Il doit donc se positionner correctement, s'assurer que la communication avec l'autre robot fonctionne et que tout est en place, etc.;
- Mode transport : Les robots effectuent la livraison en équipe.

Les robots reçoivent les commandes de livraison d'un contrôleur qui leur donne des directives à des intervalles de temps aléatoires. Les robots doivent répondre aux commandes le plus rapidement possible.

Puisque le transport à deux est une tâche délicate et qu'il est primordial de connaître l'état des deux robots en même temps, une tâche séparée s'occupe d'envoyer les directives de contrôle aux deux robots lors du transport.



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

- Dès qu'une commande est prête, le contrôleur libère une clé sur le sémaphore qu'attend le robot A (créant ainsi un rendez-vous unilatéral). Les robots A et B doivent alors se préparer (seulement si le transport de la commande précédent est fini bien sûr). Chaque robot ayant des tâches différentes lors du transport, leur temps de préparation diffère. Le robot A prend 40 cycles de OS à se préparer, et le robot B en prend 60 (ces délais sont modélisés à l'aide de la fonction OSTimeDly().

- Au début de chaque commande, juste avant le transport, chacun des robots doit mettre à jour le temps total de préparation du système. Chaque robot augmentera donc une variable à cette fin. Comme il est important de conserver cette information même en cas de panne de courant, le nombre de commande est enregistré sur une

mémoire externe avec un temps d'accès important, modélisé par les fonctions **readCurrentTotalPrepTime**() et **writeCurrentTotalPrepTime** () que vous devrez utiliser.

- Une fois que les deux robots ont terminé leurs préparations mutuelles, le contrôle du transport se réveille et commence à envoyer les commandes pour contrôler les robots lors du transport. Une fois le transport terminé, les robots reviennent à leur position initiale et le cycle peut recommencer. Le transport est simulé par un délai de 150 cycles.

a. Créez les tâches controller, prep_robot_a, prep_robot_b et transport dans le main ainsi que le premier sémaphore sem_controller_to_robot_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 modifier directement la variable total_prep_count.**
- **Différence entre total_prep_count and orderNumber : total_prep_count est le compteur de délais de commandes pour tous les robots du système (devrait valoir 1000 à la fin des 10 comandes), alors que orderNumber est le numéro de la commande (sera de 10 à la fin)**
- **Ne pas utiliser plus de 5 sémaphores**
- **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)**
- **Ne pas utiliser de drapeaux ou de files (vues plus loin)**
- **Vous ne pouvez pas interrompre le flot du contrôleur, par exemple avec un OSSemPend().**

3- Drapeaux d'événements

Il peut parfois être nécessaire d'exécuter ou de synchroniser une tâche en fonction de plus d'un sémaphore différent ou de vouloir synchroniser plus d'une tâche depuis le même sémaphore. Pour répondre à ce besoin sans avoir à écrire de quantités abondantes de code devenant de plus en plus difficiles à suivre en fonction du nombre de tâches et d'éléments de synchronisation impliqués, µC fournit la possibilité d'utiliser des drapeaux d'événements (ou « event flags » en bon français), groupés ensemble dans un « flag group ».

Créés par la fonction OSFlagCreate(), ils permettent à une tâche d'attendre sur le statut de un ou plusieurs événements différents à l'aide de OSFlagPend() et de les modifier à l'aide de OSFlagPost(). À la différence des sémaphores, le fait d'attendre sur un groupe de drapeaux ne modifie pas le statut de ceux-ci lorsque la condition d'attente est remplie (à moins de demander explicitement à OSFlagPend() de le faire).

µC représente les différents drapeaux d'un groupe comme étant simplement des bits différents d'un entier (voir l'exemple fourni dans la documentation de OSFlagPend() pour plus de détails).

Exercice 3

Le code de l'exercice 3 est une reprise du code de l'exercice 2. Toutefois, il faudra modifier un détail : tous les sémaphores seront remplacés par un seul groupe de drapeaux d'événements pour minimiser le nombre de synchronisations nécessaires (nb. vous pouvez modifier la priorité des tâches si cela vous aide, tant que la tâche contrôleur reste la plus prioritaire).

Notes :

-Une variable comptant la quantité de commandes en attente du contrôleur doit être utilisée pour ne pas manquer de commandes et être partagée entre le contrôleur et l'ordinateur s'occupant du transport, qui l'incrémentent/décrémentent respectivement, pour ne pas perdre de commandes.

-Vous pouvez modifier le statut des drapeaux depuis n'importe quelle tâche.

-En cas de problèmes de synchronisation, rappelez-vous que µC est un OS préemptif et donc qu'une tâche plus prioritaire non bloquée s'exécutera toujours avant une tâche moins prioritaire.

-Vous pourriez avoir des problèmes avec l'option OS_FLAG_CONSUME de la commande OSFlagPend()... Il est donc déconseillé de l'utiliser : faites plutôt un OSFlagPend() suivi d'un OSFlagPost().

Veuillez respecter les consignes suivantes lors de cet exercice :

- **Ne pas utiliser de files ou de sémaphores (vous pouvez utiliser des mutex à des fins d'exclusion mutuelle)**
- **Le nombre d'appels à OSFlagPend() (ou similaire comme OCFlagAccept()) est limité à 1 par tâche.**
- **Vous ne pouvez pas interrompre le flot du contrôleur, par exemple avec un OSSemPend.**

4- 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 partagée. Dans ce but, µC 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 messages 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 disponibles 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 4a

Le code de l'exercice 4a est une reprise du code de l'exercice 2. Toutefois, notre contrôleur est maintenant capable de prendre des commandes personnalisées. Il génère donc aléatoirement le temps de préparation des robots A et B ainsi que le temps du transport.

a. Reprenez votre code de l'exercice 2 et utilisez des files afin d'acheminer les informations générées dans le contrôleur vers les tâches impliquées.

Veuillez respecter les consignes suivantes lors de cet exercice :

- **Ne pas utiliser plus de 3 files (vous pouvez aussi utiliser des mutex ou des sémaphores si approprié)**

- **Attention à ne pas dupliquer les synchronisations (les files peuvent servir d'éléments de synchronisation dans certains cas)**
- **Vous ne pouvez pas interrompre le flot du contrôleur, par exemple avec un OSSemPend.**
- **N'oubliez pas de libérer la mémoire lorsque nécessaire**

Exercice 4b

Le code de l'exercice 4b est une reprise du code de l'exercice 4a et a un fonctionnement identique à celui-ci. Cependant, au lieu d'avoir deux fonctions séparées pour les réservoirs (prep_robot_A() et prep_robot_B()), vous ne devez définir qu'une fonction, **tout en conservant une tâche par robot** (le paramètre « pdata » de OSTaskCreate() devrait vous être très utile).

Veuillez respecter les consignes suivantes lors de cet exercice :

- **Mêmes consignes que l'exercice 4a**
- **Vous pouvez revoir vos méthodes de synchronisation entre tâches réservoir au besoin, utiliser plus ou moins de sémaphores, etc.**
- **Aucun code de départ n'est donné pour cet exercice : copiez exo4a.c vers exo4b.c lorsque vous aurez terminé 4a.**
- **Vous devez obtenir une trace d'exécution identique à celle de l'exercice 4a.**

5 - Questions supplémentaires

a. **Donnez un exemple de situation où il serait préférable d'utiliser un sémaphore plutôt que des drapeaux d'événements**
b. **Pour l'exercice 4a, quelle est la taille minimale des files pour être certain de ne perdre aucune commande lors de l'exécution des tâches? Attention, il ne s'agit pas ici de simplement indiquer le nombre de commandes totales pour chaque file.**
c. **Donnez une brève appréciation de ce laboratoire (temps nécessaire, 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 INF3610Lab1_matricule1_matricule2.zip contenant :

- Dans un dossier src, le code de vos 4 fichiers exo1.c, exo2.c, exo3.c, exo4a.c et exo4b.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	/2
Exo2	/5
Exo3	/4
Exo4a	/4
Exo4b	/2
Réponse aux questions	
Exo1 d.	/1
Question supplémentaire a	/1
Question supplémentaire b	/1
Question supplémentaire c	/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 µC. 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.