ÉCOLE POLYTECHNIQUE DE MONTREAL

INF3610 : Laboratoire 1

Introduction à µC-II/OS

Guy Bois

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

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

II- 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!

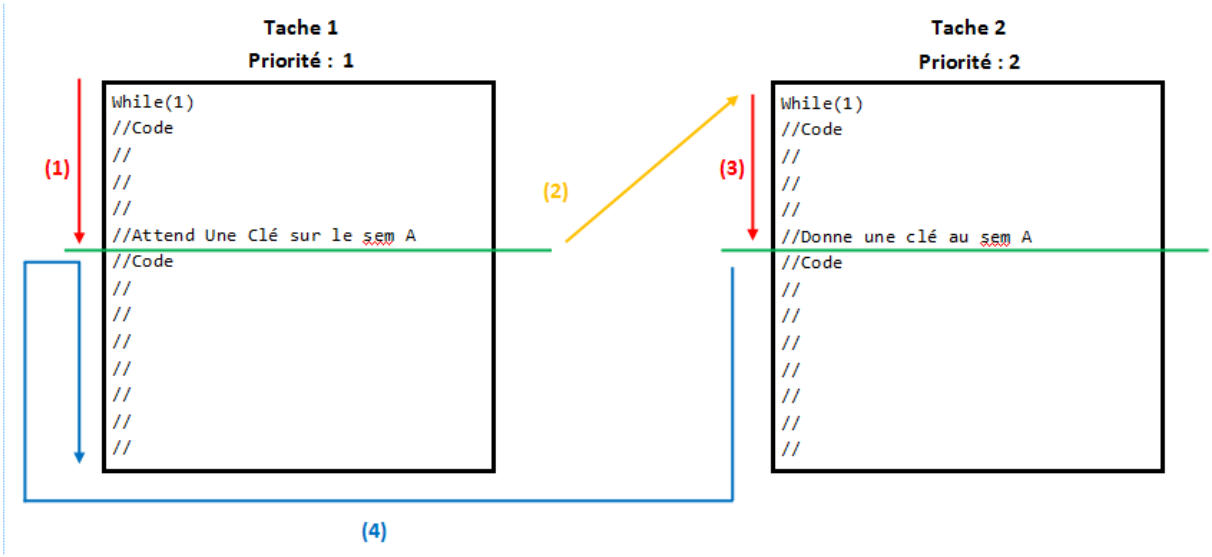
III- Éléments de synchronisation, d'exclusion et variables partagées

a. Sémaphores

Nous savons maintenant créer des tâches et les organiser les unes par rapport aux autres en fonction de leur priorité. Toutefois, µC fournit un mécanisme offrant des possibilités de synchronisation aux taches auxquelles on a assigner des priorités. Ce mécanisme se nomme sémaphore. 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*).

Revenons sur l'utilisation des ces barrières. Deux des utilisations les plus classiques sont les rendez-vous unilatéraux (Figure 1) et les rendez-vous bilatéraux (Figure 2)[1]. 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.



[1] Dans ces deux cas d'utilisation, le nombre de clés est de 0 ou 1, mais nous verrons en classe des cas d'utilisation avec plusieurs clés, qu'on nomme le sémaphore compteur.

Figure 1. Rendez-vous unilatéral

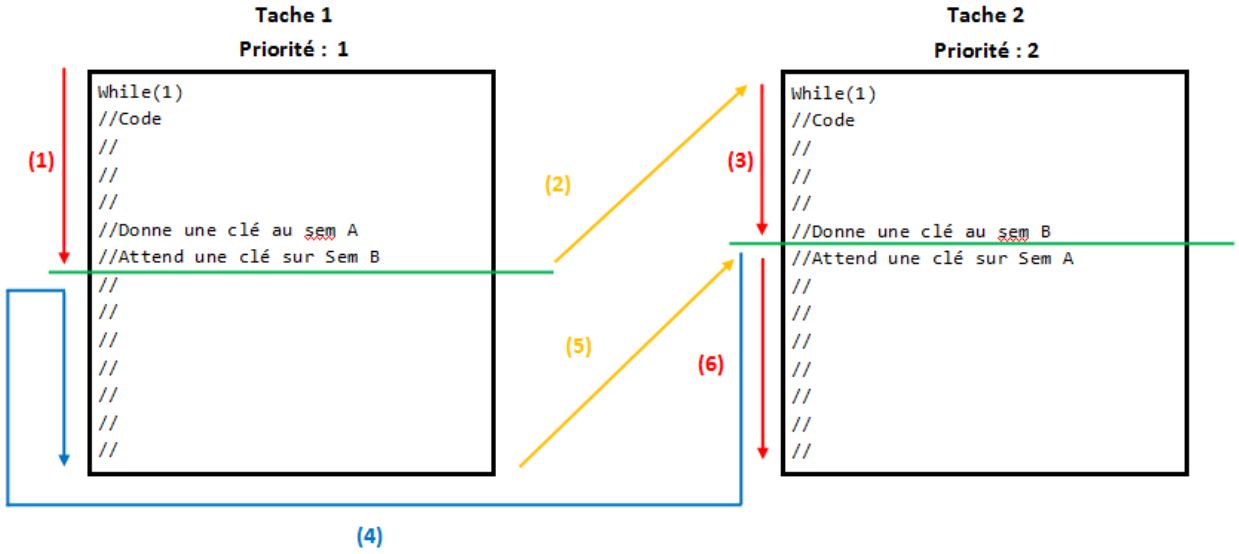


Figure 2. Rendez-vous bilatéral

a. Mutex (Mutual Exclusion)

Un autre service (mécanisme) 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.

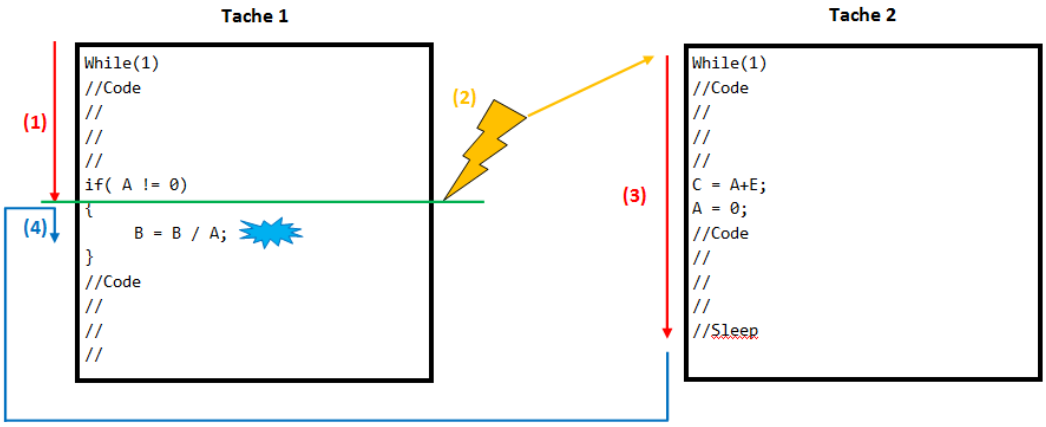


Figure 3. Corruption de données

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.

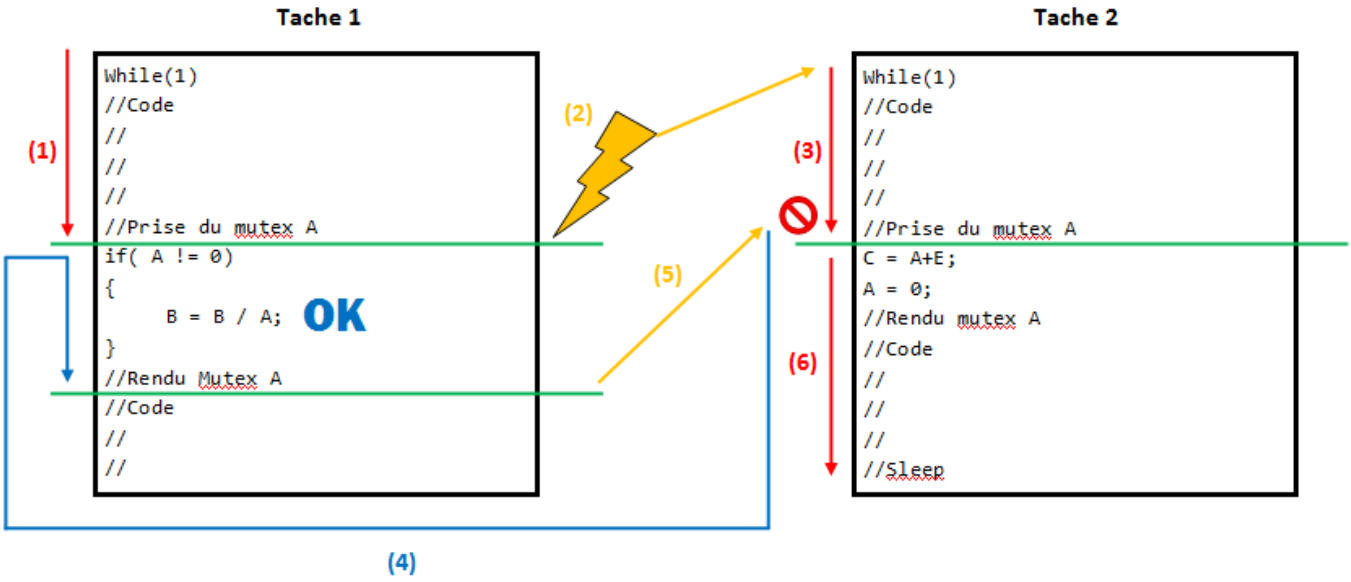


Figure 4. Corruption de données évitée grâce au mutex

b. 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:

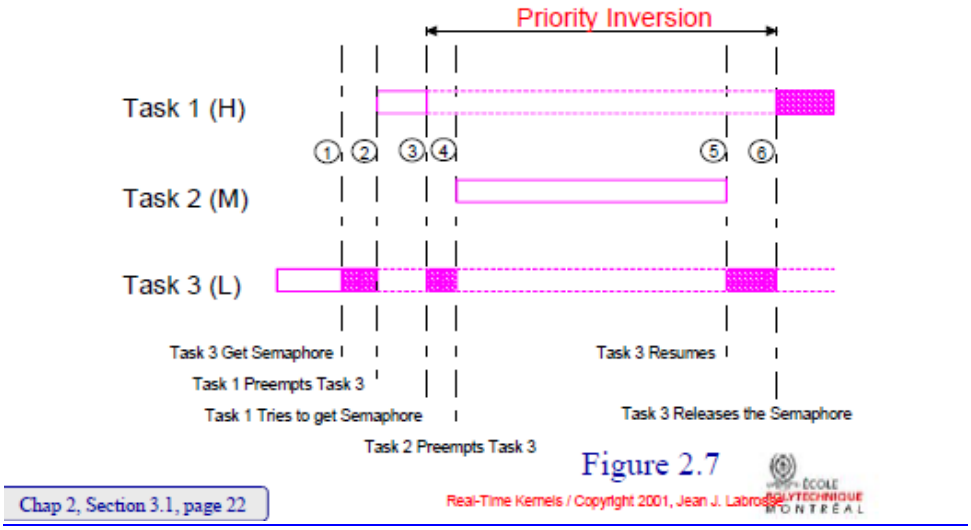


Figure 5. Phénomène d'inversion de priorité

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 un 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 ont été inroduits en classe (et seront approfondis aux cours des prochaines semaines) et ne devraient pas être rencontré 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.

b. 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).

c. 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*.