# École Polytechnique de Montréal

# Département de Génie Informatique

**INF3610**

**Hiver 2013**

**Laboratoire #1**

## Concepts de base avec uC

---

## 1. Objectifs

- Faire un premier contact avec μC
- Reproduire le problème d'inversion de priorité et comprendre la notion de temps de blocage par opposition à temps de préemption
- Faire la distinction entre *héritage de priorité* et protocole ICPP (*Immediate Ceiling Priority Protocol*)
- Faire l'ordonnancement d'une tâche μC périodique avec une minuterie
- Comprendre le concept de *watchdog* dans un système embarqué et implémenter une tâche réalisant une fonction simple de *watchdog*

## 2. Contexte

Ce travail se divisera donc en 2 parties :

### 2.1) Problème de l'inversion de priorité et les solutions pour l'éviter

Faire l'analyse du temps de réponse d'un système temps réel avec priorités statiques est une opération relativement simple tant que les tâches ne communiquent pas entre elles. Lorsque qu'une communication est requise (e.g. variable partagée) en exclusion mutuelle, des situations non souhaitées peuvent alors survenir. En effet, l'utilisation de mutex pour contrôler l'accès à une ressource partagée peut faire en sorte que l'ordre dans lequel les tâches devraient s'exécuter n'est pas forcément respecté. Ce phénomène lié à l'utilisation du mutex se nomme *inversion de priorité* et fait apparaitre la notion de *temps de blocage*.

Soit trois tâches, T1, T2 et T3, de priorité croissante (T3 étant la plus prioritaire). La figure 2 illustre le phénomène d'inversion de priorité. La tâche T1 s'exécute d'abord, entre dans sa section critique en obtenant le mutex A, puis subit une préemption par la tâche T2, plus prioritaire. T2 fait quelques cycles d'exécution avant de subir à son tour une préemption par la tâche T3. Il est toutefois impossible pour T3 d'entrer dans sa section critique puisque le mutex qu'elle désire est déjà utilisé par T1. T3 redonne ainsi le contrôle à T2 qui termine ainsi son exécution. Le contrôle revient alors à T1 qui termine sa section critique avant de redonner le contrôle à T3 puisque cette dernière peut enfin obtenir le mutex qu'elle désire. T3 termine son exécution pour ensuite redonner le contrôle à T1 qui terminera à son tour.

L'inversion de priorité découle du fait que la tâche T2 termine son exécution avant T3, malgré que cette dernière soit plus prioritaire. Ce genre de situation n'est pas désirable puisqu'une tâche de haute priorité doit terminer son exécution dans le délai le plus court possible.

À la figure 2, vous remarquerez aussi que la ligne en gras sur T3 représente le *temps de blocage*. Contrairement au temps de préemption pour lequel une tâche est arrêtée par une tâche de plus grande priorité, le temps de blocage est le temps par une tâche de plus faible priorité (effet non désirable et difficile à prédire).

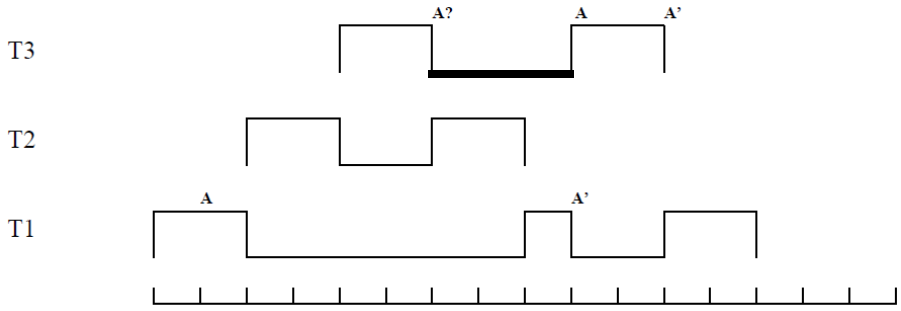


Figure 1. Inversion de priorité. A indique une demande (appel à OSSemPend) du mutex obtenu avec succès, A? indique également une demande du mutex mais sans succès (autrement dit l'appelant est bloqué) et finalement A' indique une libération du mutex (appel à OSSemPost).

## Héritage de priorité

Il existe plusieurs protocoles pour éviter l'inversion de priorité. Un algorithme fréquemment utilisé (cas de µC) est l'héritage de priorité. Lorsque T3 s'aperçoit qu'elle ne peut prendre possession du mutex, elle hausse la priorité de la tâche T1 à sa priorité (donc T1 hérite de la priorité de T3) pour que T1 termine sa section critique avant que T2 ait eu la chance de s'exécuter. Lorsque T1 termine sa SC, elle reprend sa priorité initiale et T3 peut ainsi s'exécuter. La figure 3 illustre graphiquement cette situation :

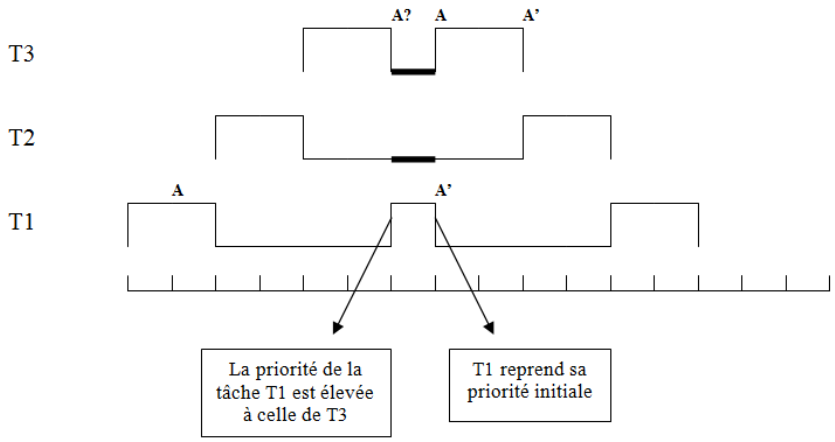


Figure 2 Héritage de priorité

T3 ne perd donc ici qu'un cycle, soit le temps que passe T1 dans sa section critique. Sur la figure 3, nous avons 2 temps de blocage (T3 mais aussi T2).

Un des inconvénients de cet algorithme est qu'il peut conduire à une situation de *deadlock* (interblocage) entre les tâches T1 et T3. En effet, imaginons le scénario où T3 obtient le mutex B avant de demander le mutex A et que ce même mutex B est aussi utilisé par T1. Nous pourrions obtenir le résultat illustré à la figure 4. Dans ces conditions, T1 et T3 sont prises à jamais dans l'attente d'un mutex. Pour éviter cette situation, il existe d'autres algorithmes pour prévenir l'interblocage. Dans ce qui suit nous en présentons un nommé ICPP.

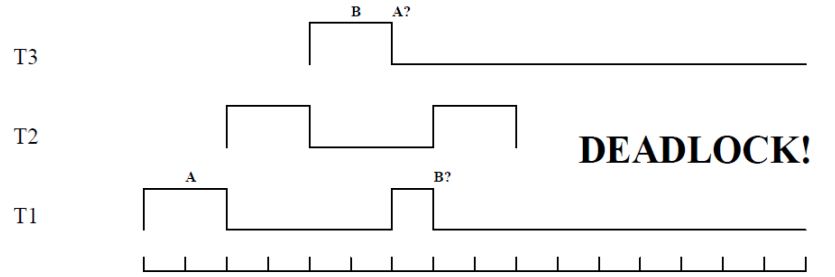


Figure 3 Interblocage causé par l'héritage de priorité

**<u>Protocole ICPP (Immediate Ceiling Priority Protocol)</u>**

Illustrons cet algorithme par cet exemple de la figure 5. Soit les tâches T1, T2 et T3, telles qu'utilisées plus tôt. La tâche T1 utilise dans son exécution les mutex A et B, tout comme le fait la tâche T3. Finalement, T2 n'utilise aucun mutex.

À la base, chacune des tâches possède une priorité statique (priorité attribuée lors de leur création). Par exemple, T1 : 15, T2 : 13, T3 : 11 (tout comme avec µC, plus la priorité est basse, plus la priorité est

grande). De plus, parmi toutes les tâches qui utilisent un mutex, ce dernier se voit assigner la plus haute priorité. Comme µC ne permet pas d'assigner deux fois la même priorité, nous dirons que les mutex ont pour une priorité définie comme étant supérieure aux priorités de toutes les tâches qui l'utilisent. Dans notre exemple, les mutex A et B pourraient ainsi avoir 10 et 9 comme priorité, respectivement.

Ainsi, T1 débute son exécution et s'approprie le mutex A. Lors de l'appropriation (OSMutexPend), la priorité de T1 est aussitôt élevée à celle du mutex qu'elle utilise (10 dans le cas présent). Ainsi, il est impossible pour T2 de s'exécuter lors de sa création même si, en théorie, elle est plus prioritaire que T1. Il en est de même pour T3. Lorsque T1 termine sa section critique en redonnant le mutex A, sa priorité est abaissée à sa valeur initiale (valeur statique : 15), ce qui permet à T3 puis T2 de s'exécuter. L'on évite ainsi toute possibilité d'interblocage.

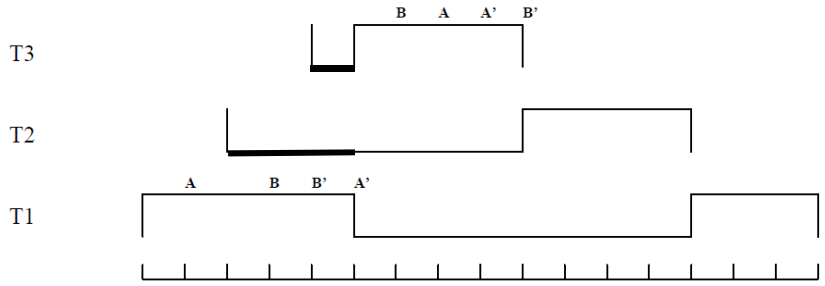


Figure 4 Algorithme ICPP

**2.2) Watchdog et tâches périodiques**

Un watchdog (en français chien de garde), est un circuit électronique ou un logiciel utilisé en électronique numérique pour s'assurer qu'un système (embarqué) ne reste pas bloqué à une étape particulière du traitement qu'il effectue. C'est une protection destinée généralement à redémarrer le système, si une action définie n'est pas exécutée dans un délai imparti. Pour nos besoins, ici le wtachdog très simple va s'occuper de détecter des situations d'interblocage créées par un ensemble de tâches périodiques.

**Tâches périodiques**

Les tâches pour des applications en temps réel sont très souvent des tâches périodiques de priorités distinctes. Le problème est le suivant: on désire exécuter une tâche à chaque *x* ticks d'horloge et on sait que cette même tâche a comme temps d'exécution maximale *y* ticks d'horloge (où $y < x$). Évidemment, il s'agit ici d'un temps maximal, ce qui veut dire que la tâche peut parfois prendre moins que *y* ticks pour s'exécuter (par exemple, imaginer une instruction conditionnelle de type *if then else* pour laquelle le passage dans la branche *then* demande *y* ticks alors que le passage dans la branche *else* demande *y*/2 ticks).

## 3. Travail à réaliser

**En ce qui concerne 2.1), vous devez :**

1. Compléter le pseudo code dans le fichier tp.c pour faire une implémentation complète en µC. Implémenter trois tâches **T1, T2, T3**. T1 est plus prioritaire que T2 et qui est elle-même plus prioritaire que T3. T3 sera la première à s'exécuter, T1 et T2 seront bloqué par un ou des sémaphores. Au départ, T3 rentre dans une section critique liée à R2 (*OSMutexPend(R2, …)*), puis débloque la tâche T2. T2 débloque tout simplement la tâche T1. Finalement, T1 rentre dans une section critique liée à R1 (*OSMutexPend(R1, …)*) et rentre immédiatement dans une section critique liée à R2 (*OSMutexPend(R2, …)*). La tâche T3 sera redémarrée et rentrera dans la section critique liée à R1 (*OSMutexPend(R1, …)*). Finalement, T1 et T3 relâchent R1 et R2.

2. Modifiez la valeur de `MUTEX_ICPP` dans le fichier os_mutex.c. Mettre à 1 pour activer ICPP et à 0 pour le désactiver. Regardez les changements de comportements du système. Le chargé vous posera peut-être des questions à ce sujet.

**En ce qui concerne 2.2, vous devez :**

1. Compléter le pseudo code dans le fichier tpPart2.c pour faire une implémentation complète en µC. Vous devez trouver une manière de simuler un code de durée variable (<PERIOD), autrement dit simulez une attente active. (Utilisez la fonction *OSTimeGet*).

2. Implémenter trois tâches périodiques ainsi qu'une tâche *timer* qui limitera l'exécution des trois tâches à 50 *ticks* cumulatifs. Par conséquent, soit un **système sous uC de 4 tâches T1, T2, T3 et TaskTimer**. T1 a une priorité de 10, T2 a une priorité de 20, T3 a une priorité de 30 et la tâche timer a une priorité à votre choix. T1 doit rentrer dans une section critique liée à R1 (*OSMutexPend(R1, …)*), puis tombe dans une attente non-active de 5 ticks. Après son réveil, T1 rentre dans une section critique liée à R2 (*OSMutexPend(R2, …)*). Finalement, elle relâche les deux mutexs. Pendant l'attente de T1, la tâche T2 démarre et rentre dans une attente active de 10 ticks et cette même attente diminue d'un tick à chaque exécution de la tâche. Finalement, T3 démarre puis rentre dans une section critique liée à R2 (*OSMutexPend(R2, …)*). Ensuite, elle rentre dans une section critique liée à R1 (*OSMutexPend(R1, …)*) puis tombe dans une attente active de 2 ticks. Finalement, T3 libére les deux mutexs.

3. Lorsque le délai de 50 ticks est terminé pour la tâche *Timer*, si un (ou plusieurs) Mutexs est toujours utilisés par une tâche, vous devez réinitialiser le système. Les mutexs, les sémaphores ainsi que les autres tâches devront être détruites et puis recréées. (Assurez-vous que les tâches ne détiennent plus de ressources avant de les détruire)

## 4. Évaluation

Vous disposez de 2 semaines pour effectuer ce laboratoire (1 séance de laboratoire seulement). Vous devez envoyer votre travail à votre chargé de laboratoire par courriel avant le jeudi 26 septembre 23hheures 59min. Vous devez remettre un fichier .zip ou .rar clairement identifier contenant uniquement les deux fichiers tp.c et tpPart2.c. Le nom du .zip doit être présenté sous la forme : Lab1_INF360_A13_Matricule1_Matricule2.zip. Au début de la première séance du TP2, le chargé posera deux questions que vous devrez répondre individuellement. Si vous avez des questions concernant le laboratoire, vous pouvez les poser par courriel à *jeff.falcon@polymtl.ca*.

## 5. Barème correction

| | |
|---|---|
| • Évaluation de la partie 2.1 | /2 |
| • Évaluation de la partie 2.2 | /4 |
| • Question #1 | /2 |
| • Question #2 | /2 |
| **TOTAL** | /10 |

**Bon travail!**