

### Project : The laboratory intelligent vehicle with several sensors

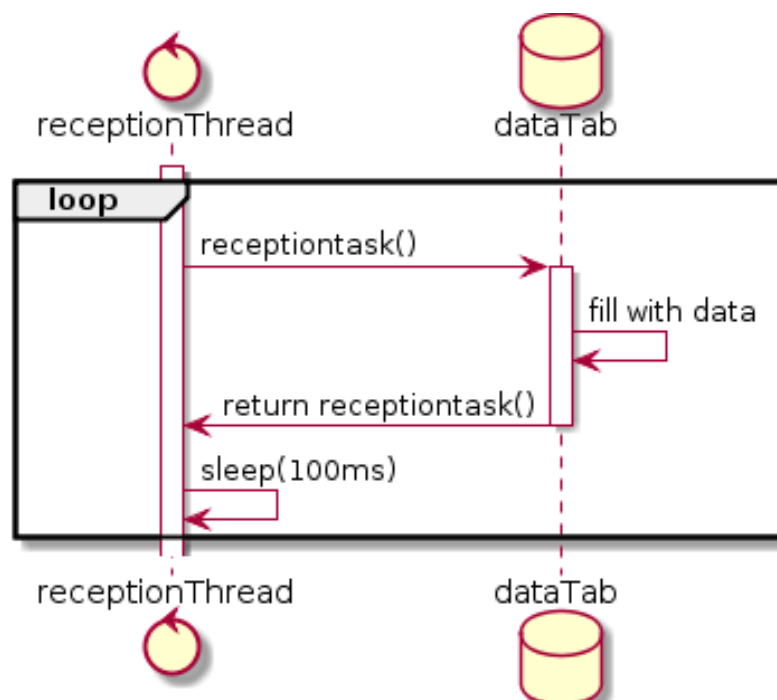
#### 1. description UML

Le projet consiste en la simulation du travail d'un véhicule intelligent. Nous devons simuler la réception de données par un capteur, le traitement de ces données et l'affichage des données importantes.

Pour ce projet, nous avons besoin de 3 threads, qui sont indépendants (en principe!). Ils effectuent donc chacun leur travail et se mettent en pause avant de recommencer périodiquement. Toute la difficulté est de faire que chaque thread s'exécute sans gêner l'exécution des autres. La synchronisation est décrite dans le point suivant.

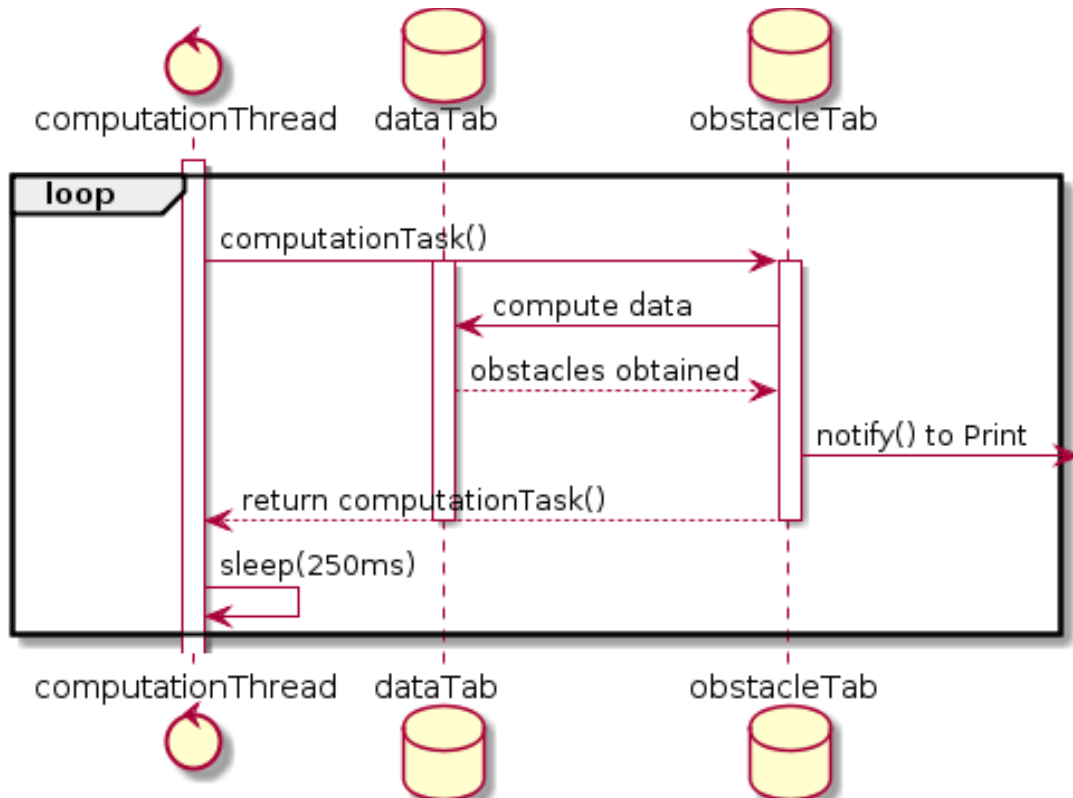
Tout d'abord, décrivons les différentes tâches :

#### Reception Task :



La tâche de réception remplit un tableau avec des valeurs aléatoires. Ce tableau est également utilisé par d'autres tâches. Quand le tableau est rempli, le processus de réception se met au repos pendant 100ms avant de recommencer le même travail.

### Computation Task :

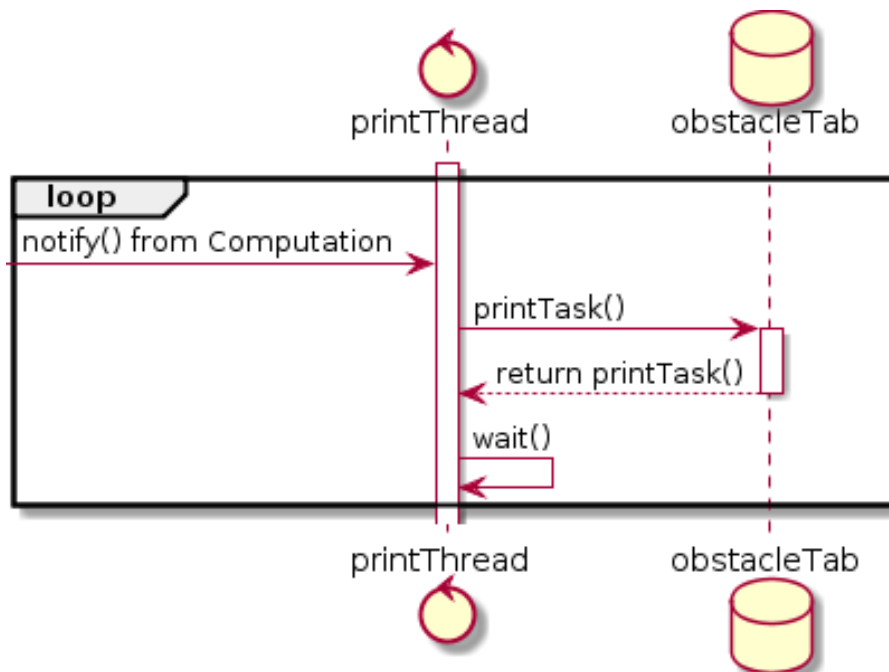


Dans cette tâche, le travail consiste en un parcours du tableau précédemment rempli et la récupération de certaines données dans un nouveau tableau "obstacle".

Quand le traitement est terminé, la tâche avertit le processus d'affichage que de nouvelles données sont disponibles. Le processus se met ensuite en pause pendant 250ms avant de recommencer.

### Print Task :

Cette tâche permet d'afficher le tableau "obstacle" qui a été traité par le processus précédent. A chaque fois que la tâche "computation" a fini son travail, la tâche "affichage" doit être lancée. Pour se faire, elle se met en attente d'un signal provenant du processus précédent. Quand le signal est reçu, le thread parcourt le tableau d'obstacles et affiche chaque élément.



## 2. Problème de synchronisation

Il y a plusieurs points qui concernent les problèmes de synchronisation dans ce projet.

Le premier concerne les données générées par le processus "Reception". Ces dernières sont également utilisées par le processus "Computation". ces données sont représentées par le tableau "data" dans mon programme. Il est donc nécessaire de contrôler l'accès à cette ressource. Pour se faire, un mutex a été utilisé. Ainsi lorsqu'un processus est en train d'utiliser ou de remplir ce tableau, le processus qui souhaite accéder à la ressource est mis en attente jusqu'à ce que mutex soit libéré. Et le second processus bloque alors à son tour l'accès à la ressource.

(Une condition a également été mise en place pour que le processus "computation" ne traite pas deux fois les mêmes données. Ainsi, si le tableau a déjà été traité, "computation" est mis en pause et le tableau est libéré pour que le processus "reception" puisse le remplir avec de nouvelles données.)

Le deuxième point concerne les données issue du processus "computation". Il s'agit du tableau "obstacle" qui contient tous les points dont la distance est supérieure à 7m. Ce tableau est partagé avec le processus "print" qui est chargé d'afficher le tableau. Un mutex a également été utilisé. De cette manière aucun des deux processus ne peut pas accéder à la ressources si elle est utilisée par l'autre processus. Mais dans ce cas, le problème est moins important car le processus "print" est appelé par "computation" quand ce dernier à fini son travail. Mais par sécurité, la ressource est tout de même verrouillée.

Ce qui nous amène au troisième point : la synchronisation de ces deux dernier processus ("computation" et "print"). "print" doit être lancé à chaque fois que le processus "computation" a terminé sa tâche. Pour se faire, on lance de thread « print » au départ, puis il est mis en pause. Et à chaque fois que le processus "computation" a fini son travail, il envoi un signal pour la tâche "print soit réveillée.

### 3. Comparaison entre les 3 implémentations

On peut comparer les 3 implémentations en terme de performance dans le temps.

L'implémentation séquentielle nous permet de remarquer l'intérêt des application multi-threadées. En effet, dans cette implémentation, nous n'avons qu'un seul thread et chaque tâche est exécutée l'une après l'autre. Donc pendant que chaque tâche est mise en pause, les autres attendent également. Il y a une perte de temps importante à ce niveau. En revanche dans cette implémentation, il n'y a pas besoin de mécanismes de synchronisation. Chaque tâche effectuant son travail quand son tour est arrivé, il n'y a jamais 2 processus qui tentent d'accéder à une ressource en même temps.

L'implémentation à base de thread simples permet un gain de performance important par rapport à l'implémentation précédente. En effet, chaque tâche est indépendante et travaille donc indépendamment des deux autres. Quand une tâche est au repos, les deux autres peuvent travailler. il n'y a donc presque plus d'attente pour chaque thread.

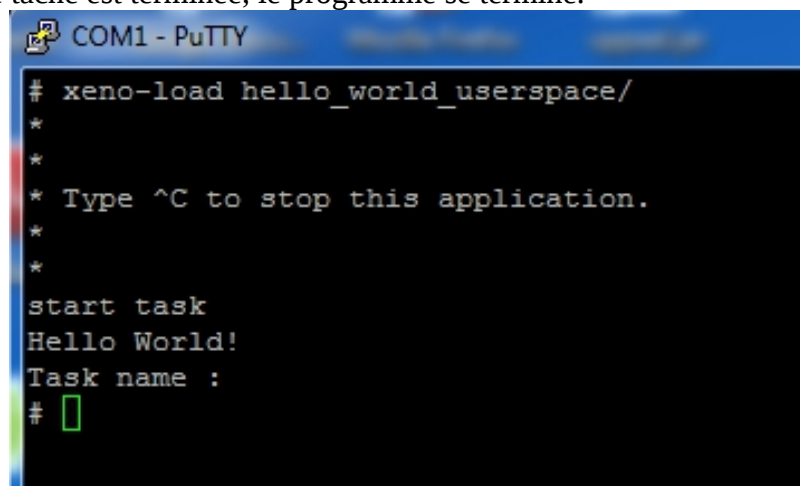
L'implémentation basée sur les tâches xenomai (thread temps réel) apporte encore un gain de performance. Car ces tâches sont gérées par le noyau temps réel de la carte. L'ordonnanceur utilisé n'est plus le même et du fait des contraintes temps réel, la synchronisation entre les tâches en est encore améliorée.

En quelques chiffres :

- séquentiel : sur 10s, 7735 exécutions de chaque tâche
- thread simple : 19770 receptions, 12199 computations, 11935 prints
- tâches xenomai : les tâches ne se lançaient pas sur ma carte à cause d'un problème de segmentation dont on ne connaît pas la cause.

### 4. Exercices

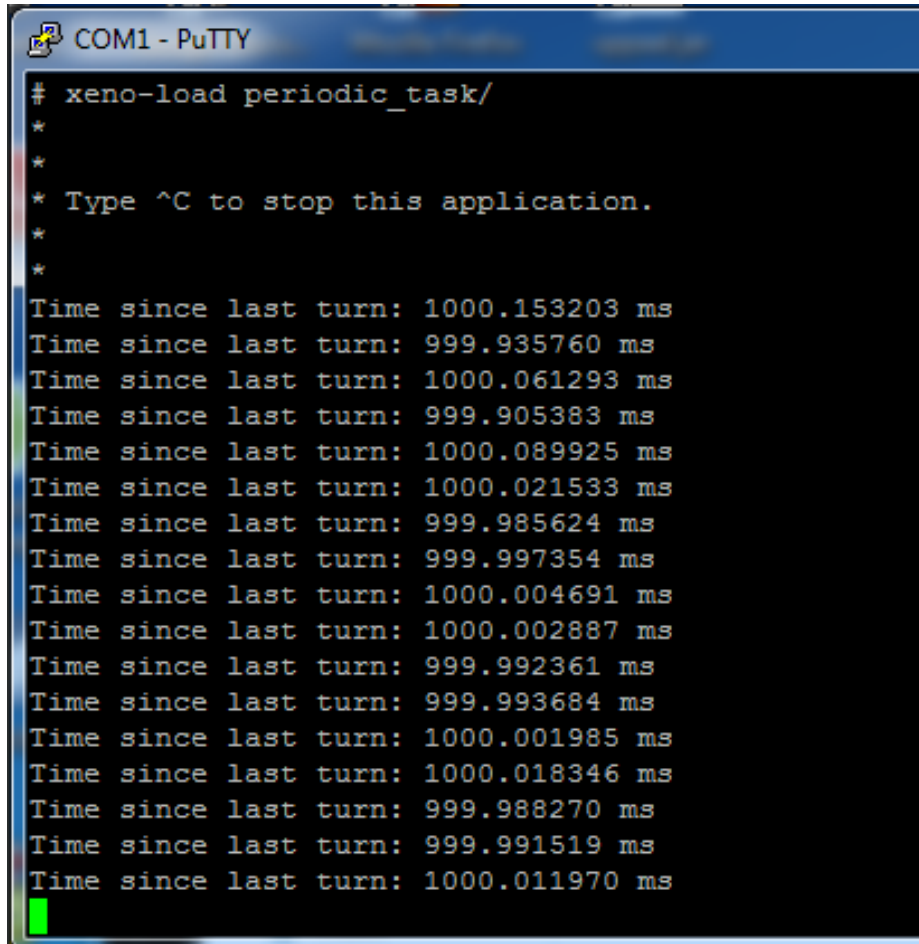
1. Dans ce programme, le processus principal crée une tâche, lui donne un nom et la lance. A l'exécution, la tâche affiche « hello world » et ensuite son nom. Quand la nouvelle tâche est terminée, le programme se termine.



```
# xeno-load hello_world_userspace/
*
*
* Type ^C to stop this application.
*
*
start task
Hello World!
Task name :
#
```

2. Ce programme crée une nouvelle tâche mais cette dernière est périodique. Dans cette tâche, on récupère la valeur actuelle du timer et on la compare à la dernière lecture. Cela permet de vérifier si la période qui a été définie fait exactement 1000ms. Lors de l'exécution, on remarque que la période n'est pas rigoureusement respectée alors qu'il s'agit tout de même d'une tâche « temps réel ».

L'écart varie de +0.152ms à -0.094ms.



```
# xeno-load periodic_task/
*
*
* Type ^C to stop this application.
*
*
Time since last turn: 1000.153203 ms
Time since last turn: 999.935760 ms
Time since last turn: 1000.061293 ms
Time since last turn: 999.905383 ms
Time since last turn: 1000.089925 ms
Time since last turn: 1000.021533 ms
Time since last turn: 999.985624 ms
Time since last turn: 999.997354 ms
Time since last turn: 1000.004691 ms
Time since last turn: 1000.002887 ms
Time since last turn: 999.992361 ms
Time since last turn: 999.993684 ms
Time since last turn: 1000.001985 ms
Time since last turn: 1000.018346 ms
Time since last turn: 999.988270 ms
Time since last turn: 999.991519 ms
Time since last turn: 1000.011970 ms
```

3. Dans ce troisième programme nous lançons 2 tâches qui vont incrémenter ou décrémenter une variable globale au programme.

Dans la première implémentation, on remarque que quand une tâche est lancée, elle effectue tous les tours de boucles et, seulement quand elle a fini, donne la main à la seconde tâche.

(voir capture d'écran ci-dessous)

La seconde implémentation, un sémaphore binaire a été mis en place pour qu'à chaque fois que la variable globale est incrémentée, la tâche est mise en attente pour que la seconde tâche puisse décrémenter la variable, et réciproquement. Ainsi, la variable ne prend plus que les valeurs 0 et 1.

```
~
# sh c.sh
# cd /usr/local/bin
# chmod a+x task_sync/hello_world_userspace
# xeno-load task_sync/
*
*
* Type ^C to stop this application.
*
*
I am taskOne and global = 1.....
I am taskOne and global = 2.....
I am taskOne and global = 3.....
I am taskOne and global = 4.....
I am taskOne and global = 5.....
I am taskOne and global = 6.....
I am taskOne and global = 7.....
I am taskOne and global = 8.....
I am taskOne and global = 9.....
I am taskOne and global = 10.....
I am taskTwo and global = 9-----
I am taskTwo and global = 8-----
I am taskTwo and global = 7-----
I am taskTwo and global = 6-----
I am taskTwo and global = 5-----
I am taskTwo and global = 4-----
I am taskTwo and global = 3-----
I am taskTwo and global = 2-----
I am taskTwo and global = 1-----
I am taskTwo and global = 0-----
```