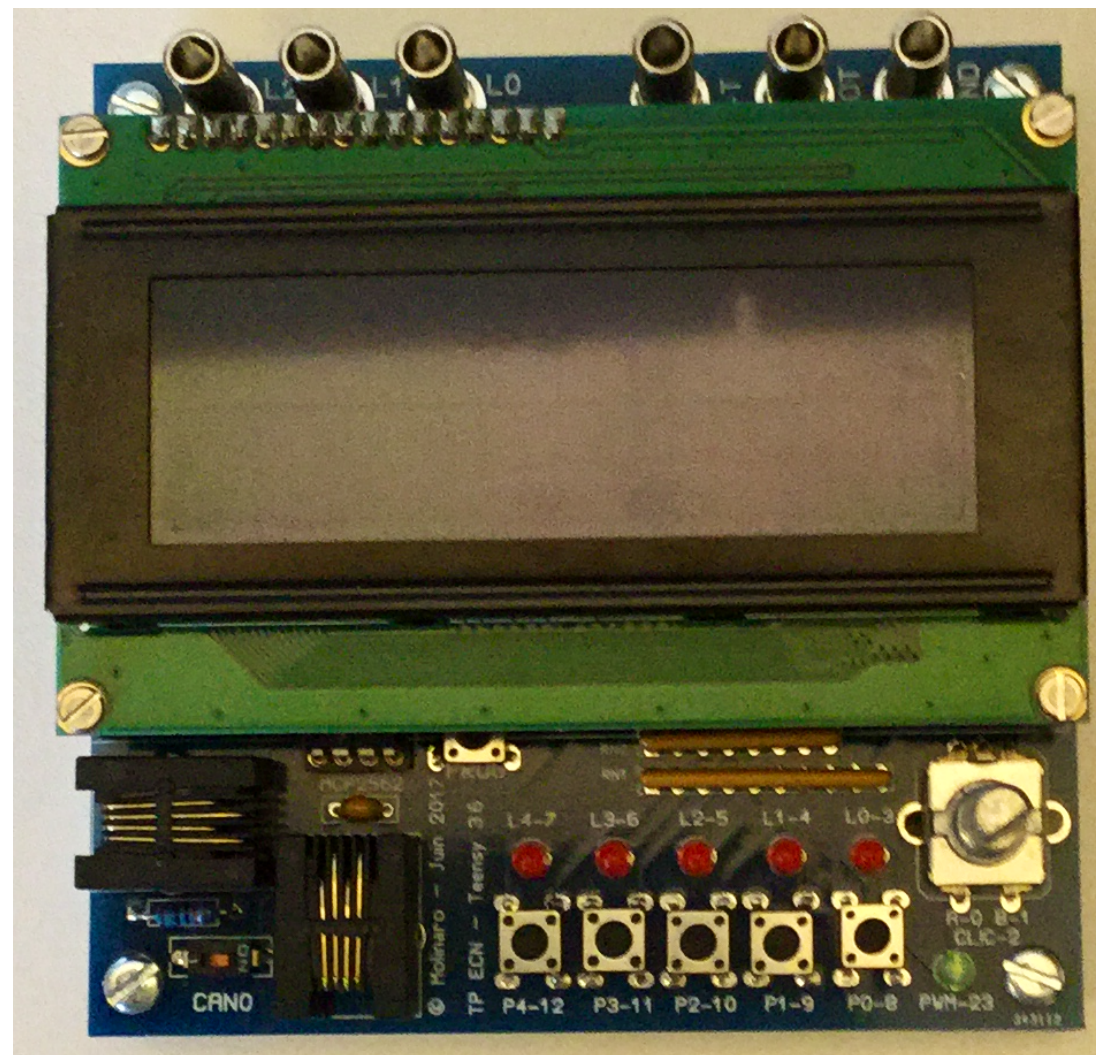


Temps Réel



Programme *15-synchronization-tools*

Description de cette étape

Cette étape est consacrée aux outils de synchronisation. On peut classer les primitives bloquantes des outils de synchronisation en trois niveaux :

- le blocage simple d'une tâche : par exemple **s.P()** ; c'est ce qui est présenté dans cette étape ;
- le blocage d'une tâche jusqu'à une échéance : par exemple **s.P_until(*échéance*)**, présenté dans l'étape suivante ;
- le blocage en garde : par exemple **[s.P() -> | s2.P() ->]**, présenté dans la dernière étape.

Deux fonctions de base vont être présentées dans cette étape : **kernel_blockRunningTaskInList** et **irq_makeTaskReadyFromList** : elles permettent d'exprimer la plupart des outils de synchronisation.

Dupliquer le projet de l'étape précédente et renommez-le par exemple **15-synchronization**.

La fonction `kernel_blockRunningTaskInList`

Cette fonction bloque la tâche en cours et insère son descripteur dans la liste passée en argument ; elle est appellable en mode **KERNEL** :

```
void kernel_blockRunningTaskInList (KERNEL_MODE_ TaskList & ioWaitingList) {  
    XTR_ASSERT_NON_NULL_POINTER (gRunningTaskControlBlockPtr) ;  
    //--- Insert in task list  
    ioWaitingList.enterTask (MODE_ gRunningTaskControlBlockPtr) ;  
    //--- Block task  
    kernel_makeNoTaskRunning (MODE) ;  
}
```

Insérer cette fonction dans **xtr.cpp**, et son prototype dans **xtr.h**.

La fonction `irq_makeTaskReadyFromList`

Cette fonction, appellable en mode **IRQ**, agit sur la liste de tâches passée en argument :

- si la liste est vide, la valeur **false** est renvoyée par la fonction ;
- si la liste n'est pas vide, la tâche la plus prioritaire en est retirée, cette tâche est rendue prête, et la valeur **true** est renvoyée par la fonction.

```
bool irq_makeTaskReadyFromList (IRQ_MODE_ TaskList & ioWaitingList) {  
    TaskControlBlock * taskPtr = ioWaitingList.removeFirstTask (MODE) ;  
    const bool found = taskPtr != nullptr ;  
    if (found) {  
        kernel_makeTaskReady (MODE_ taskPtr) ;  
    }  
    return found ;  
}
```

Insérer cette fonction dans **xtr.cpp**, et son prototype dans **xtr.h**.

Le sémaphore de Dijkstra (1/5)

On trouve dans la littérature de nombreuses implémentations du sémaphore de Dijkstra, par exemple :

Un sémaphore de Dijkstra est constitué :

- d'une variable **e** positive, négative ou nulle ;
- une liste **l** de tâches.

Initialisation

e prend une valeur ≥ 0
l est la liste vide

Primitive P

```
e := e - 1 ;  
si e < 0 alors  
    Bloquer la tâche en cours dans l  
    Appeler l'ordonnanceur  
finsi
```

Primitive V

```
e := e + 1  
si e  $\geq 0$  alors  
    une tâche est retirée de l  
    cette tâche est rendue prête  
    Appeler l'ordonnanceur  
finsi
```

Un sémaphore de Dijkstra est constitué :

- d'une variable **e** positive ou nulle ;
- une liste **l** de tâches.

Initialisation

e prend une valeur ≥ 0
l est la liste vide

Primitive P

```
si e == 0 alors  
    Bloquer la tâche en cours dans l  
    Appeler l'ordonnanceur  
sinon  
    e := e - 1 ;  
finsi
```

Primitive V

```
si l est vide alors  
    e := e + 1  
sinon  
    une tâche est retirée de l  
    cette tâche est rendue prête  
    Appeler l'ordonnanceur  
finsi
```

Le sémaphore de Dijkstra (2/5) : déclaration de la classe

On choisit la seconde implémentation, car elle permet facilement l'extension à l'attente temporelle et aux commandes gardées. En effet, le blocage d'une tâche lors d'un appel de **P** n'a pas d'autre effet de bord que l'insertion dans la liste des tâches bloquées.

Le sémaphore est déclaré comme une classe C++. Les primitives doivent être définies comme des services.

```
#include "task-list--32-tasks.h"

class Semaphore {
//--- Properties
    protected: TaskList mWaitingTaskList ;
    protected: uint32_t mValue ;

//--- Constructor
    public: Semaphore (const uint32_t inInitialValue) ;

//--- V
//$service semaphore.V
    public: void V (USER_MODE) asm ("semaphore.V") ;
    public: void sys_V (IRQ_MODE) asm ("service.semaphore.V") ;

//--- P
//$service semaphore.P
    public: void P (USER_MODE) asm ("semaphore.P") ;
    public: void sys_P (KERNEL_MODE) asm ("service.semaphore.P") ;

//--- No copy
    private: Semaphore (const Semaphore &) ;
    private: Semaphore & operator = (const Semaphore &) ;
};
```

Un sémaphore de Dijkstra est constitué :

- d'une variable **e** positive ou nulle ;
- une liste **l** de tâches.

Le sémaphore de Dijkstra (3/5) : initialisation

Initialisation

e prend une valeur ≥ 0
l est la liste vide

Le constructeur effectue l'initialisation :

```
Semaphore::Semaphore (const uint32_t inInitialValue) :  
mWaitingTaskList (),  
mValue (inInitialValue) {  
}
```

Le sémaphore de Dijkstra (4/5) : primitive P

Primitive P

```
si e == 0 alors
    Bloquer la tâche en cours dans l
    Appeler l'ordonnanceur
sinon
    e := e - 1 ;
finsi
```

La méthode **sys_P** implémente la primitive **P** du sémaphore :

```
void Semaphore::sys_P (KERNEL_MODE) {
    ..... à vous d'écrire l'implémentation .....
}
```

Notez que l'ordonnanceur n'est pas explicitement appelé. En effet, **sys_P** est appelé par la méthode **P** via le *svc handler* qui se charge lui-même d'appeler la fonction **kernel.select.task.to.run**.

Les annotations de mode garantissent qu'une routine d'interruption n'appellera ni **P** ni **sys_P**.

Le sémaphore de Dijkstra (5/5) : primitive V

Primitive V

```
si l est vide alors  
    e := e + 1  
sinon  
    une tâche est retirée de l  
    cette tâche est rendue prête  
    Appeler l'ordonnanceur  
finsi
```

La méthode **sys_V** implémente la primitive **V** du sémaphore :

```
void Semaphore::sys_V (IRQ_MODE) {  
    ..... à vous d'écrire l'implémentation .....  
}
```

Notez que l'ordonnanceur n'est pas explicitement appelé. En effet :

- soit **sys_V** est appelé par la méthode **V** via le *svc handler* qui se charge lui-même d'appeler la fonction **kernel.select.task.to.run** ;
- soit **sys_V** est appelé directement par une routine d'interruption, qui doit être déclarée en mode **IRQ**, ce qui provoquera l'appel de la fonction **kernel.select.task.to.run**.

Les annotations de mode garantissent qu'une routine d'interruption n'appellera pas **V** mais **sys_V**.

Travail à faire

Écrire le code du sémaphore dans des fichiers **Semaphore.h** et **Semaphore.cpp**.

Ajouter un sémaphore d'exclusion mutuelle dans **lcd.cpp** de façon qu'un caractère soit écrit de façon indivisible.

Attention, dans le code de **lcd.cpp**, on ne demande pas à ce qu'une séquence de caractères soit écrite de façon indivisible. Par exemple, si deux tâches écrivent deux nombres en parallèle, les deux séquences seront entrelacées.

On peut à titre d'information voir comment est géré le même problème sur les ordinateurs de bureau (voir pages suivantes).

Un exemple d'utilisation des sémaphores (1/2)

À exécuter sur votre ordinateur, et non pas sur la carte de TP

```
#include <iostream>
#include <thread>

//-----*
static const int NOMBRE_THREADS = 10 ;
//-----*

static void codeThread (int tid) {
    std::cout << "thread " << tid << std::endl ;
}

//-----*

int main (void) {
    //--- Déclaration des threads
    std::thread t [NOMBRE_THREADS] ;
    //--- Démarrage des threads
    for (int i=0 ; i<NOMBRE_THREADS ; i++) {
        t[i] = std::thread (codeThread, i) ;
    }
    //--- Message
    std::cout << "main\n";
    //--- Attente de la fin de l'exécution des threads
    for (int i=0 ; i<NOMBRE_THREADS ; i++) {
        t[i].join () ;
    }
    //---
    return 0;
}
```

Compilation : `g++ main.cpp -o main`

Lire attentivement le programme ci-contre, et le faire tourner sur votre ordinateur de bureau.

11 threads se déroulent en parallèle, chacun d'eux affiche un message.

Voici le résultat de deux exécutions :

```
main
thread 9
thrthttttttethhhhhhahrrrrrrrdreeeeeeee eaaaaaaa1addddddd
d                23074568
```

```
mttatthttttthtihrhhhhhrhnrerrrrrrer
eeaeaeaeaeadaaaaadadd dddd d 9      8 03
25164
7
```

Les caractères sont corrects, mais les affichages sont entremêlés.

Note : sur votre plateforme, vous pouvez être amené à utiliser les options de compilation suivantes :

- `-std=c++11` (les threads sont définis à partir du C++ 11) ;
- `-lpthread` (édition des liens avec la librairie libpthread).

Un exemple d'utilisation des sémaphores (2/2)

À exécuter sur votre ordinateur, et non pas sur la carte de TP

```
#include <iostream>
#include <thread>

//-----*

static const int NOMBRE_THREADS = 10 ;
static std::mutex semaphore ; // Sémaphore initialisé à 1

//-----*

static void codeThread (int tid) {
    semaphore.lock () ; // P(semaphore)
    std::cout << "thread " << tid << std::endl ;
    semaphore.unlock () ; // V (semaphore)
}

//-----*

int main (void) {
    //--- Déclaration des threads
    std::thread t [NOMBRE_THREADS] ;
    //--- Démarrage des threads
    for (int i=0 ; i<NOMBRE_THREADS ; i++) {
        t[i] = std::thread (codeThread, i) ;
    }
    //--- Message
    semaphore.lock () ; // P(semaphore)
    std::cout << "main\n";
    semaphore.unlock () ; // V (semaphore)
    //--- Attente de la fin de l'exécution des threads
    for (int i=0 ; i<NOMBRE_THREADS ; i++) {
        t[i].join () ;
    }
    //---
    return 0;
}
```

Compilation: g++ main.cpp -o main

On ajoute maintenant un sémaphore d'exclusion mutuelle (en bleu).

Voici le résultat de deux exécutions :

| | |
|----------|----------|
| main | thread 2 |
| thread 0 | main |
| thread 7 | thread 4 |
| thread 8 | thread 8 |
| thread 6 | thread 9 |
| thread 9 | thread 5 |
| thread 1 | thread 0 |
| thread 3 | thread 1 |
| thread 4 | thread 6 |
| thread 2 | thread 3 |
| thread 5 | thread 7 |

Les affichages sont corrects, l'ordre peut varier d'une exécution à une autre.

Note : sur votre plateforme, vous pouvez être amené à utiliser les options de compilation suivantes :

- `-std=c++11` (les threads sont définis à partir du C++ 11);
- `-lpthread` (édition des liens avec la librairie `libpthread`).

Quelques outils de synchronisation

L'évènement fugace

Un évènement fugace contient une liste de tâches bloquées.

Initialement, la liste des tâches bloquées est vide.

La primitive **wait** bloque inconditionnellement la tâche qui l'appelle.

La primitive **signal** libère toutes les tâches bloquées.

L'évènement mémorisé

Un évènement mémorisé contient une liste de tâches bloquées et un booléen.

Initialement, la liste des tâches bloquées est vide et le booléen est faux ou vrai.

La primitive **wait** :

- si le booléen est vrai, il est mis à faux ;
- si il est faux, la tâche appelante est bloquée.

La primitive **signal** :

- si la liste des tâches bloquées n'est pas vide, toutes les tâches bloquées sont rendues prêtes ;
- si la liste des tâches bloquées est vide, le booléen est mis à vrai.

Porte logicielle

Une porte logicielle contient une liste de tâches bloquées et un booléen, qui indique si la porte est ouverte ou fermée.

Initialement, la liste des tâches bloquées est vide et la porte est ouverte ou fermée.

La primitive **wait** :

- si la porte est ouverte, la tâche appelante passe sans être bloquée ;
- si la porte est fermée, la tâche appelante est bloquée.

La primitive **open** :

- si la porte est ouverte, aucune action, la liste des tâches bloquées est vide ;
- si la porte est fermée, elle est ouverte, et toutes les tâches bloquées sont rendues prêtes.

La primitive **close** :

- ferme la porte.

Rendez-vous : port de Silberschatz

Le *port de Silberschatz* est un outil de synchronisation basé sur le rendez-vous, très semblable à la synchronisation de CSP. Les différences sont :

- les commandes d'entrée et de sortie nomment un *port*, au lieu de nommer la tâche correspondante ;
- dans un premier temps, il n'y a pas de donnée transmise, c'est une synchronisation pure.

La commande d'entrée est notée **A?**, la commande de sortie **A!**.

Commande de sortie **A!** :

- si une ou plusieurs tâches ayant invoqué une commande d'entrée **A?** sur le même port sont bloquées alors :
 - ▶ une tâche bloquée est libérée ;
 - ▶ la commande de sortie n'est pas bloquante ;
- si aucune tâche ayant invoqué une commande d'entrée sur le même port n'est bloquée alors la commande de sortie est bloquante.

Commande d'entrée **A?** :

- si une ou plusieurs tâches ayant invoqué une commande de sortie **A!** sur le même port sont bloquées alors :
 - ▶ une tâche bloquée est libérée ;
 - ▶ la commande d'entrée n'est pas bloquante ;
- si aucune tâche ayant invoqué une commande de sortie sur le même port n'est bloquée alors la commande d'entrée est bloquante.

Port de Silberschatz avec transmission de données

Nous allons maintenant ajouter la transmission de données au *port de Silberschatz*. À titre d'exemple, nous considérerons que la donnée est de type `uint32_t`. Pour un type quelconque, utiliser un *template C++ de classe*.

Le problème est plus complexe qu'il n'y paraît. Il faut s'assurer de la synchronisation et que les recopies de données s'effectuent en exclusion mutuelle.

Le plus simple (à mon avis...) est définir une nouvelle classe qui possède comme propriétés un port de Silberchatz (sans donnée), et des sémaphores.