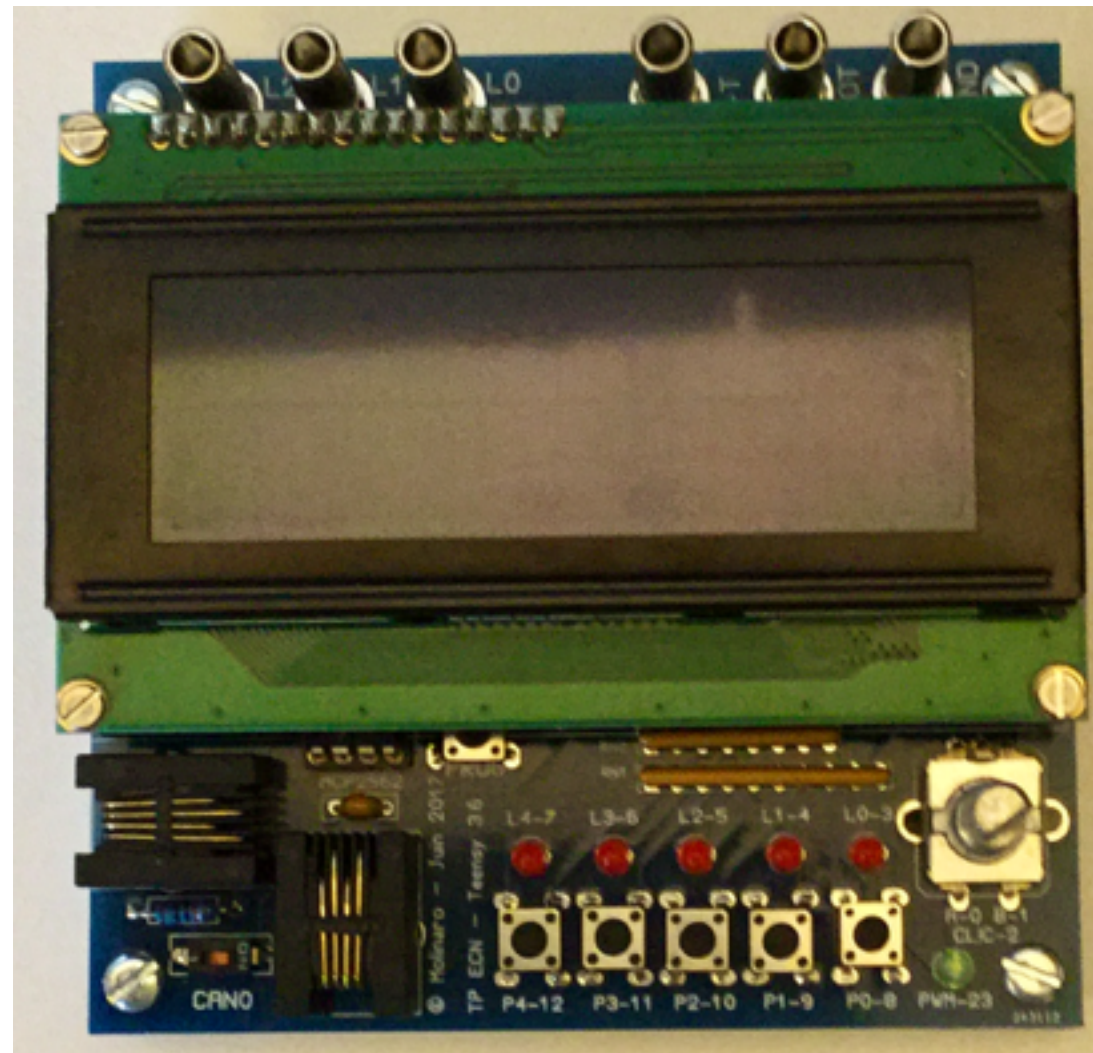


Temps Réel



Programme *15-synchronization-tools*

Description de cette étape

Cette étape est consacrée aux outils de synchronisation.

Deux fonctions vont être présentées ; 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.

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) {
    if (mValue == 0) {
        kernel_blockRunningTaskInList (MODE_ mWaitingTaskList) ;
    }else{
        mValue -= 1 ;
    }
}
```

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_P** implémente la primitive **P** du sémaphore :

```
void Semaphore::sys_V (IRQ_MODE) {
    const bool found = irq_makeTaskReadyFromList (MODE_ mWaitingTaskList) ;
    if (! found) {
        mValue += 1 ;
    }
}
```

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 ; celle-ci 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
thrtdtttttttethhhhhhahrrrrrrrdreeeeeeee eaaaaaaa1addddddd
d                23074568
```

```
mttatthtttttthtihrhhhhhrhnrerrrrrrer
eeaeaeaeaeadaaaaaadadd 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).

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 fugace : une implémentation

Déclaration

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

class Event {
//--- Properties
    protected: TaskList mWaitingTaskList ;

//--- Constructor
    public: Event (void) ;

//--- wait
//$service event.wait
    public: void wait (USER_MODE) asm ("event.wait") ;
    public: void sys_wait (KERNEL_MODE) asm ("service.event.wait") ;

//--- signal
//$service event.signal
    public: void signal (USER_MODE) asm ("event.signal") ;
    public: void sys_signal (IRQ_MODE) asm ("service.event.signal") ;

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

Implémentation

```
Event::Event (void) :
mWaitingTaskList () {
}

void Event::sys_wait (KERNEL_MODE) {
    kernel_blockRunningTaskInList (MODE_ mWaitingTaskList) ;
}

void Event::sys_signal (IRQ_MODE) {
    while (irq_makeTaskReadyFromList (MODE_ mWaitingTaskList)) {}
}
```

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.

L'évènement mémorisé : une implémentation (1/2)

Déclaration

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

class StoredEvent {
//--- Properties
    protected: TaskList mWaitingTaskList ;
    protected: bool mState ;

//--- Constructor
    public: StoredEvent (const bool inInitialState) ;

//--- wait
    // $service stored.event.wait
    public: void wait (USER_MODE) asm ("stored.event.wait") ;
    public: void sys_wait (KERNEL_MODE) asm ("service.stored.event.wait") ;

//--- signal
    // $service stored.event.signal
    public: void signal (USER_MODE) asm ("stored.event.signal") ;
    public: void sys_signal (IRQ_MODE) asm ("service.stored.event.signal") ;

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


L'évènement mémorisé : une implémentation (2/2)

Implémentation

```
StoredEvent::StoredEvent (const bool inInitialState) :  
mWaitingTaskList (),  
mState (inInitialState) {  
}  
  
void StoredEvent::sys_wait (KERNEL_MODE) {  
    if (mState) {  
        mState = false ;  
    }else{  
        kernel_blockRunningTaskInList (MODE_ mWaitingTaskList) ;  
    }  
}  
  
void StoredEvent::sys_signal (IRQ_MODE) {  
    const bool found = irq_makeTaskReadyFromList (MODE_ mWaitingTaskList) ;  
    if (! found) {  
        mState = true ;  
    }else{  
        while (irq_makeTaskReadyFromList (MODE_ mWaitingTaskList)) {}  
    }  
}
```

Porte logicielle

Un 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.

Porte logicielle : une implémentation (1/2)

Déclaration

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

class Gate {
//--- Properties
    protected: TaskList mWaitingTaskList ;
    protected: bool mIsOpen ;

//--- Constructor
    public: Gate (const bool inIsOpen) ;

//--- wait
//$service gate.wait
    public: void wait (USER_MODE) asm ("gate.wait") ;
    public: void sys_wait (KERNEL_MODE) asm ("service.gate.wait") ;

//--- open
//$service gate.open
    public: void open (USER_MODE) asm ("gate.open") ;
    public: void sys_open (IRQ_MODE) asm ("service.gate.open") ;

//--- close
//$service gate.close
    public: void close (USER_MODE) asm ("gate.close") ;
    public: void sys_close (IRQ_MODE) asm ("service.gate.close") ;

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

Porte logicielle : une implémentation (2/2)

Implémentation

```
Gate::Gate (const bool inIsOpen) :  
mWaitingTaskList (),  
mIsOpen (inIsOpen) {  
}  
  
void Gate::sys_wait (KERNEL_MODE) {  
    if (!mIsOpen) {  
        kernel_blockRunningTaskInList (MODE_ mWaitingTaskList) ;  
    }  
}  
  
void Gate::sys_open (IRQ_MODE) {  
    if (!mIsOpen) {  
        mIsOpen = true ;  
        while (irq_makeTaskReadyFromList (MODE_ mWaitingTaskList)) {}  
    }  
}  
  
void Gate::sys_close (IRQ_MODE) {  
    mIsOpen = false ;  
}
```