Objectifs : Acquérir les notions élémentaires de la programmation temps réel.
Mise en œuvre sur STM32 avec MBED-OS

Attention, ce TP est une approche d'un système temps réel sur microcontrôleur, il ne constitue pas un cours exaustif sur la technologie des systèmes d'exploitation temps réel (RTOS).
Il constitue néanmoins une approche suffisante pour les étudiants de BTS Systèmes Numériques et une première approche pour les élèves d'écoles d'ingénieurs

Matériel : Ce TP utilise une NUCLEO-F411RE, mais n'importe quelle autre carte NUCLEO convient.
Logiciel : MBED

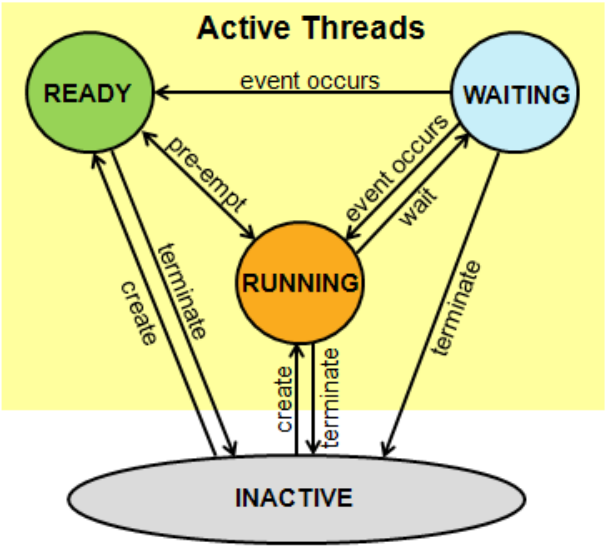
MBED utilise le système temps réel CMSIS-RTOS de KEIL. (Cortex Microcontroller Software Interface Standard – Real Time Operating System)

Un système d'exploitation temps réel (Real Time Operation System ou RTOS) est un système multi-tâches pour lequel le temps maximum entre un stimulus d'entrée et une réponse de sortie est précisément déterminé.
Exemple : le logiciel de gestion d'une voiture automatique demande s'il y a un obstacle à l'avant du véhicule. Un OS-Temps réel pourra garantir une réponse en moins de 10mS, un OS standard pourrait retarder la mesure d'un temps … indéterminé

Ressources OS-Temps réel : https://en.wikipedia.org/wiki/Real-time_operating_system

1 RTOS

Un RTOS permet de paralléliser les tâches. Une tâche est appelée « thread » (fil d'exécution). Le passage de l'exécution d'une tâche à l'autre est effectuée par le « scheduleur » du RTOS.
L'exécution peut être réellement parallèle dans le cas d'un processeur multi-cœurs. Dans le cas d'un processeur mono-cœur (comme le STM32) l'alternance des tâches est très rapide, ce qui donne l'impression que le processeur exécute plusieurs tâches en même temps. Suivant sont importance une tâche dispose d'une priorité et d'un temps d'exécution, lorsque ce temps est écoulé la tâche est suspendue, son état est préservée et la tâche suivant est executée.



2 Thread

Un Thread peut être dans les états suivants:

- **Running** : le thread en cours d'exécution. Un seul thread à la fois peut être dans cet état.
- **Ready** : les threads prêts à fonctionner. Une fois que le thread en running est terminé ou en waiting , le thread ready ayant la priorité la plus élevée devient le thread en running exécution.
- **Waiting** : les threads en attente d'un événement.
- **Inactive** : threads qui ne sont pas créés ou terminés. Ces threads ne consomment généralement aucune ressource système.

Signaux : Chaque Thread peut recevoir/emettre des signaux (create-terminate) et être averti d'événements:

Exemple de threads et de signaux :

```
#include "mbed.h"

Thread mon_thread;        // instanciation d'un thread
DigitalOut led(LED1);


void led_thread() {
    while (true) {
        Thread::signal_wait(0x1); // le thread attend le signal 0x1
        led = !led;
    }
}

int main (void) {
    mon_thread.start(led_thread);
    while (true) {
        wait(1);
        mon_thread.signal_set(0x1);
    }
}
```

led_thread est un fonction exécutée dans le thread.
Elle dispose d'une boucle sans fin, le thread ne se terminera donc jamais seul.
Dans la boucle sans fin le thread attend le signa 0x1 pour faire basculer la LED.
Le drapeau du signal est traité comme un événement et est automatiquement acquitté.

mon_thread.start(led_thread); Démarrage du thread, qui devra exécuter la fonction led_thread.

On voit ensuite une boucle sans fin qui envoie le signal 0x1 toutes les secondes.

Exemple de threads :
Cet exemple met en évidence la parallélisation des tâches , deux LEDs clignotent à des périodes différente dans deux boucles infinies.

```
#include "mbed.h"

DigitalOut led1(LED1);
DigitalOut led2(LED2);
Thread thread;

void led2_thread() {
    while (true) {
        led2 = !led2;
        wait(1);
    }
}

int main() {
    thread.start(led2_thread);
    while (true) {
        led1 = !led1;
        wait(0.5);
    }
}
```

Callback permet de transmettre des paramètres à un thread.signal
L'exemple ci dessous lance un thread avec en paramètre le port sur lequel se situe la LED

```
#include "mbed.h"

Thread thread;
DigitalOut led1(LED1);
volatile bool running = true;

// Blink function toggles the led in a long running loop
void blink(DigitalOut *led) {
    while (running) {
        *led = !*led;
        wait(1);
    }
}

// Spawns a thread to run blink for 5 seconds
int main() {
    thread.start(callback(blink, &led1));
    wait(5);
    running = false;
    thread.join();
}
```

Le thread s'execute tant que « running » est vrai, running devient faut après 5 secondes et se termine donc.
La méthode « join » attend la fin du thread.

Important : running est un booleen déclaré « volatile », ce qualificatif empêche le compilateur optimiser le code pour cette variable. En effet, à l'intérieur du thread, running est testé mais jamais modifié, sans le qualificatif volatil cela entraînerait lors la compilation la suppression du test.

Principales méthodes de la classe Thread :
https://os.mbed.com/docs/v5.8/mbed-os-api-doxy/classrtos_1_1_thread.html

start : Démarre un thread exécutant la fonction spécifiée.
terminate : Mettre fin à l'exécution d'un thread et le retirer des threads actifs
gettid : Récupère l'identifiant du thread en cours d'exécution.
join : Attendre la fin du thread
get_state : retourne l'etat du thread
set_priority : Définir la priorité d'un thread actif
signal_clr : efface un signal
signal_set : active un signal
signal_wait : Attendre qu'un ou plusieurs signaux deviennent actifs pour le fil en cours d'exécution.
wait : Attendre une période de temps spécifiée en millisecondes
wait_until : Attendre jusqu'à une heure spécifiée en millisecondes

2.1 Exercices :

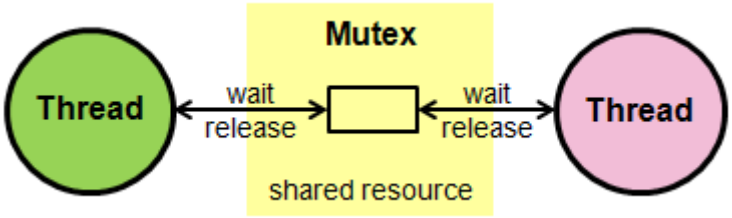
Réaliser un programme avec trois threads et trois LEDs:
Vous ferez clignoter les LEDs grâce aux threads avec des périodes respectives de 0,1s, 0,5s et 1s

Réaliser un programme avec deux threads :
Le premier lit en permanence un port analogique de votre choix et range le résultat dans une variable globale.
Le deuxième envoie cette variable sur le port série lors de l'appui sur le bouton utilisateur.

3 MUTEX

La classe Mutex permet de synchroniser les échanges d'informations entre threads.

Ressources :
https://fr.wikipedia.org/wiki/Exclusion_mutuelle
http://fr.cppreference.com/w/cpp/thread/mutex
https://os.mbed.com/docs/v5.8/reference/mutex.html



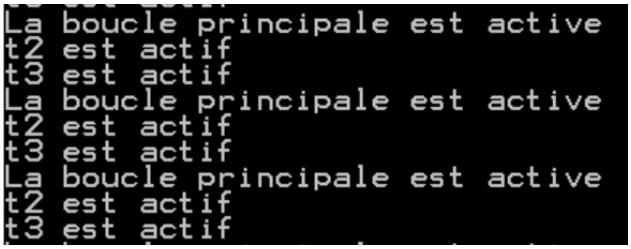
Méthodes de Mutex les plus utilisées :
lock : Attendre jusqu'à ce qu'un Mutex devienne disponible. Lock peut disposer d'un argument de temps
ex monMutex.lock(10) ; // le mutex attend 10mS avant de continuer
Si le Mutex est disponible, lock le rend indisponible.
unlock : Débloquez le mutex précédemment verrouillé par le même fil

Un exemple de l'intérêt (voir de l'obligation) d'utiliser des Mutex :
***Remarque** : printf n'est pas utilisé ici, printf agit sur la sortie standard (stdio) qui sur MBED-OS est par défaut protégée par un Mutex.*

Tester ce programme :

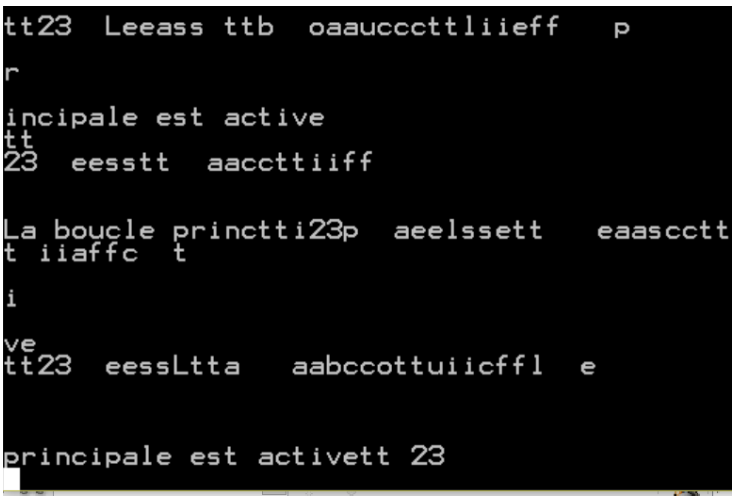
```
#include "mbed.h"

Serial pc(USBTX,USBRX);

Mutex test_mutex;
Thread t2;
Thread t3;

char mess1[]="La boucle principale est active \n\r";
char mess2[]="t2 est actif \n\r";
char mess3[]="t3 est actif \n\r";

void test_thread(char * message) {
char *p;
   while (true) {
          p=message;
          test_mutex.lock();
          while (*p) {putchar(*p++); wait(0.05);}
          test_mutex.unlock();
          wait(1);
   }
}

int main() {
   t2.start(callback(test_thread, mess2));
   t3.start(callback(test_thread, mess3));
   test_thread(mess1);
}
```

Le résultat doit être le suivant :



Commenter maintenant les instructions Mutex (en rouge)

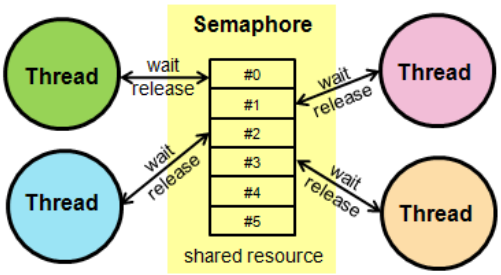
Le résultat ressemble à ceci :



Expliquer en quelques lignes ce qu'il se passe en l'absence des instructions Mutex

4 Semaphore

https://os.mbed.com/docs/v5.8/reference/queue.html



Le « Mutex » est en fait un « Semaphore » qui n'autorise d'un accès à la fois à une ressource. Le Semaphore peut autoriser N accès simultanés à une ressource.
L'instanciation est la suivante :
Semaphore monSemaphore(n) ; // n est le nombre d'accès autorisés
La méthode wait attend tant que le nombre d'accès possible est nul. (il existe des timeout), si le semaphore n'est pas null, il est décrémenté de 1.
La méthode release incrémente le sémaphore de 1.

Un exemple d'utilisation des sémaphores :

```
#include "mbed.h"

Serial pc(USBTX,USBRX);

Semaphore deux_slots(1);
Thread t2;
Thread t3;

char mess1[]="11111111111111111";
char mess2[]="22222222222222222";
char mess3[]="33333333333333333";

void test_thread(char *message) {
char *p;
   while (true) {
      p=message;
      deux_slots.wait();
      while (*p) {putchar(*p++); wait(0.05);}
      deux_slots.release();
   }
}

int main() {
   t2.start(callback(test_thread, mess2));
   t3.start(callback(test_thread, mess3));
   test_thread(mess1);
}
```

Le résultat ressemble au précédemment, un sémaphore de 1 a le même comportement qu'un Mutex.

Changer l'instanciation comme suit :
Semaphore deux_slots(2);
Maintenant deux threads sur les trois peuvent se partager la ressource. Voici le résultat

On voit très bien les alternances de deux threads actifs, 12 ou 13 ou 23 …