Ecole Nationale
Supérieure
de l'Electronique
et de ses Applications
ENSEA

# Noyaux Temps-réel

## Les sémaphores

**Laurent Fiack**
**Bureau D212/D060 – laurent.fiack@ensea.fr**

# Sémaphores binaires

# Exemple : Traitements en interruptions

```c
#include "main.h"
#include "usart.h"
#include "tim.h"
#include "gpio.h"

uint8_t input_char;

/* New character on USART1 */
void USART1_IRQHandler(void)
{
    process_char(input_char);

    HAL_UART_Receive_IT(&huart1, &input_char, 1);
    HAL_UART_IRQHandler(&huart1);
}

/* Every millisecond or something */
void TIM1_IRQHandler(void)
{
    process_something();

    HAL_TIM_IRQHandler(&htim1);
}
```

```c
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();

    HAL_UART_Receive_IT(&huart1, &input_char, 1);
    HAL_TIM_Base_Start_IT(&tim1);

    for(;;)
    {
    }
}
```

- Et si les traitements sont longs ?

# Exemple : Traitements délégués dans une superboucle

```c
#include "main.h"
#include "usart.h"
#include "tim.h"
#include "gpio.h"

uint8_t input_char;

uint8_t usart1_char_available = 0; /* New! */
uint8_t tim1_overflowed = 0;       /* New! */

/* New character on USART1 */
void USART1_IRQHandler(void)
{
    usart1_char_available = 1; /* New! */

    HAL_UART_Receive_IT(&huart1, &input_char, 1);
    HAL_UART_IRQHandler(&huart1);
}

/* Every millisecond or something */
void TIM1_IRQHandler(void)
{
    tim1_overflowed = 1;       /* New! */

    HAL_TIM_IRQHandler(&htim1);
}
```

```c
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();

    HAL_UART_Receive_IT(&huart1, &input_char, 1);
    HAL_TIM_Base_Start_IT(&tim1);

    for(;;)
    {
        if (usart1_char_available)     /* New! */
        {
            process_char(input_char); /* New! */
        }
        if (tim1_overflowed)           /* New! */
        {
            process_something();       /* New! */
        }
    }
}
```

- Et les priorités ?

# Exemple : Traitements délégués dans des tâches (1/2)

```c
#include "main.h"
#include "usart.h"
#include "tim.h"
#include "gpio.h"

#include "FreeRTOS.h" /* New! */

uint8_t input_char;

uint8_t usart1_char_available = 0;
uint8_t tim1_overflowed = 0;

/* New character on USART1 */
void USART1_IRQHandler(void)
{
    usart1_char_available = 1;

    HAL_UART_Receive_IT(&huart1, &input_char, 1);
    HAL_UART_IRQHandler(&huart1);
}

/* Every millisecond or something */
void TIM1_IRQHandler(void)
{
    tim1_overflowed = 1;

    HAL_TIM_IRQHandler(&htim1);
}
```

## New !

```c
void task_usart1(void * unused)
{
    HAL_UART_Receive_IT(&huart1, &input_char, 1);

    for(;;)
    {
        if (usart1_char_available)
        {
            process_char(input_char);
        }
    }
}

void task_tim1(void * unused)
{
    HAL_TIM_Base_Start_IT(&tim1);

    for(;;)
    {
        if (tim1_overflowed)
        {
            process_something();
        }
    }
}
```

# Exemple : Traitements délégués dans des tâches (2/2)

```c
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();

    xTaskCreate(task_usart1, "USART1", 256, NULL, 1, NULL); /* New! */
    xTaskCreate(task_tim1, "TIM1", 256, NULL, 2, NULL);     /* New! */

    vTaskStartScheduler();
}
```

- Quel est le problème ?

# Exemple : La bonne méthode, avec des sémaphores (1/2)

```c
#include "main.h"
#include "usart.h"
#include "tim.h"
#include "gpio.h"
#include "FreeRTOS.h"

uint8_t input_char;

SemaphoreHandle_t sem_usart1; /* New! */
SemaphoreHandle_t sem_tim1;   /* New! */

/* New character on USART1 */
void USART1_IRQHandler(void)
{
    xSemaphoreGive(sem_usart1); // Attention! Illégal

    HAL_UART_Receive_IT(&huart1, &input_char, 1);
    HAL_UART_IRQHandler(&huart1);
}

/* Every millisecond or something */
void TIM1_IRQHandler(void)
{
    xSemaphoreGive(sem_tim1);   // Attention! Illégal

    HAL_TIM_IRQHandler(&htim1);
}
```

```c
void task_usart1(void * unused)
{
    HAL_UART_Receive_IT(&huart1, &input_char, 1);

    for(;;)
    {
        xSemaphoreTake(sem_usart1, portMAX_DELAY); /* New! */
        process_char(input_char);
    }
}

void task_tim1(void * unused)
{
    HAL_TIM_Base_Start_IT(&tim1);

    for(;;)
    {
        xSemaphoreTake(sem_tim1, portMAX_DELAY);   /* New! */
        process_something();
    }
}
```

# Exemple : La bonne méthode, avec des sémaphores (2/2)

```c
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();

    sem_usart1 = xSemaphoreCreateBinary(); /* New! */
    sem_tim1 = xSemaphoreCreateBinary();   /* New! */

    xTaskCreate(task_usart1, "USART1", 256, NULL, 1, NULL);
    xTaskCreate(task_tim1, "TIM1", 256, NULL, 2, NULL);

    vTaskStartScheduler();
}
```

## En bref

- Les tâches sont dans l'état **BLOCKED**
- Elles deviennent **READY** grâce aux interruptions
- Si le timer est déclenché pendant le traitement de l'USART1 :
  - La tâche préempte l'UART
- Si un charactère arrive pendant le traitement du TIM1 :
  - L'interruption est traitée
  - Mais le timer continue
  - Le charactère est traité après

# Syntaxe : Création d'un sémaphore

- Création d'un sémaphore

```
SemaphoreHandle_t xSemaphoreCreateBinary(void);
```

  - Paramètre de retour : `SemaphoreHandle_t`
    - Handle pour manipuler le sémaphore
    - NULL en cas d'erreur
- Destruction d'un sémaphore

```
void xSemaphoreDelete(SemaphoreHandle_t xSemaphore);
```

  - `xSemaphore` : Handle du sémaphore à détruire

# Syntaxe : Manipulation d'un sémaphore

- Obtention d'un sémaphore

```
BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);
```

  - `xSemaphore` : Handle du sémaphore à prendre
  - `xTicksToWait` : Timeout avant de débloquer la tâche
    - **Attention !** Le sémaphore n'est pas pris
    - Penser à faire de la gestion d'erreur
  - Paramètre de retour :
    - pdTRUE si le sémaphore est obtenu
    - pdFALSE si le sémaphore n'est pas disponible après le timeout
- Libération d'un sémaphore

```
BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);
```

  - `xSemaphore` : Handle du sémaphore à donner
  - Paramètre de retour :
    - pdTRUE si le sémaphore est libéré
    - pdFALSE en cas d'échec

# Dans une interruption

- Premier constat :
  - Une interruption peut survenir à n'importe quel moment
  - Lors de l'exécution de n'importe quelle tâche
  - Elle n'est pas liée à une tâche
- Conséquences :
  - Interruption dans un contexte particulier
  - Utilisation de la pile système
  - Et donc pas la pile de la tâche en cours d'exécution
- Deuxième constat :
  - Take et Give appellent le scheduler
  - Le scheduler peut effectuer un changement de tâche
  - Et modifie donc le contexte
- Conséquences :
  - Sauvegarde du contexte de l'interruption
  - Pas de la tâche active
  - Graves déconvenues !

# Solution

## Le problème

```c
/* New character on USART1 */
void USART1_IRQHandler(void)
{
    xSemaphoreGive(sem_usart1); // Attention! Illégal

    HAL_UART_Receive_IT(&huart1, &input_char, 1);
    HAL_UART_IRQHandler(&huart1);
}
```

## La solution

```c
/* New character on USART1 */
void USART1_IRQHandler(void)
{
    BaseType_t higher_priority_task_woken = pdFALSE;

    xSemaphoreGiveFromISR(sem_usart1, &higher_priority_task_woken);

    HAL_UART_Receive_IT(&huart1, &input_char, 1);
    HAL_UART_IRQHandler(&huart1);

    portYIELD_FROM_ISR(higher_priority_task_woken);
}
```

# Syntaxe : Manipulation d'un sémaphore en interruption

- Libération d'un sémaphore

```
BaseType_t xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore,
       BaseType_t *pxHigherPriorityTaskWoken);
```

  - xSemaphore : Handle du sémaphore à donner
  - pxHigherPriorityTaskWoken est une sortie, et vaut pdTRUE si une tâche plus prioritaire est réveillée, pdFALSE sinon.
  - Paramètre de retour :
    - pdTRUE si le sémaphore est libéré
    - pdFALSE en cas d'échec

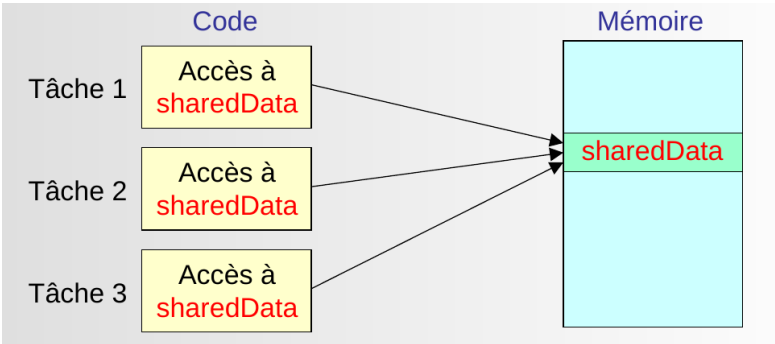
- Appel manuel du scheduler

```
portYIELD_FROM_ISR(BaseType_t pxHigherPriorityTaskWoken);
```

  - Appelle le scheduler si pxHigherPriorityTaskWoken vaut pdTRUE

# Sémaphores mutex

# Cas d'utilisation : Mémoire partagée




- `SharedData` est une simple variable globale
  - Ou un tableau, ou une structure
- L'accès multiple doit être protégé pour éviter les incohérences

# Ressource et section critique

## Ressource critique

- Variable (tableau, structure) ou périphérique commun à plusieurs tâches
- Une seule tâche peut y accéder à un instant donné
  - On parle d'**exclusion mutuelle.**

## Section critique

- Aussi appelé *Région* ou *Zone* critique
- Partie du code qui accède à la ressource critique

# Protection par blocage des interruptions

```
fonction() {
    ...
    taskENTER_CRITICAL();
    ...
    ... /* région critique ne pouvant pas être interrompue */
    ...
    taskEXIT_CRITICAL();
    ...
}
```

- Interdit toutes les interruptions pendant tout le traitement de la région critique
- Peu recommandable pour une application temps-réel

# Protection en bloquant la préemption

```
fonction() {
    ...
    vTaskSuspendAll();
    ...
    ... /* région critique ne pouvant pas être interrompue */
    ...
    xTaskResumeAll();
    ...
}
```

- N'empêche pas les interruptions
- Mais empêche le changement de tâches
- Une tâche plus prioritaire ne peut pas s'exécuter
- À éviter également

# Protection en utilisant un sémaphore Mutex

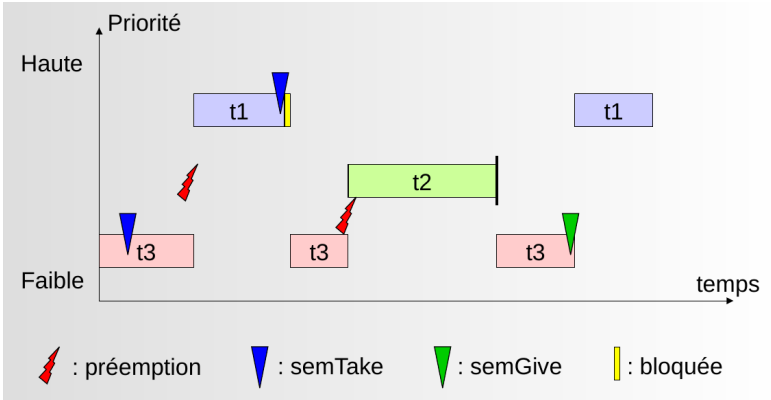
```
fonction() {
    ...
    xSemaphoreTake(sem_handle, SEM_DELAY);
    ...
    ... /* région critique ne pouvant pas être interrompue */
    ...
    xSemaphoreGive(sem_handle);
    ...
}
```

- N'empêche pas les interruptions
- N'empêche pas la préemption
- Une tâche plus prioritaire peut s'exécuter
- Elle bloque si elle essaie de prendre le sémaphore
- C'est la bonne solution !

# Sémaphore d'exclusion mutuelle (ou sémaphore Mutex)

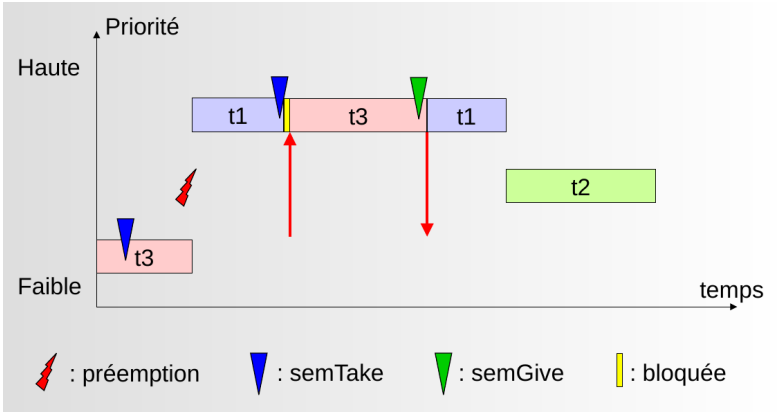
- C'est un sémaphore binaire spécialisé pour l'exclusion mutuelle
- Il possède trois caractéristiques supplémentaires :
  - Inversion de priorité
  - Protection contre la destruction
    - La tâche ayant pris le sémaphore est protégée contre la destruction
  - Accès récursif (Seulement les RecursiveMutex)
- De plus :
  - Il ne doit être utilisé que pour l'exclusion mutuelle
  - Il est plein par défaut
  - Il ne peut être libéré que par la tâche qui l'a pris
  - Il ne peut pas être pris par un serveur d'interruption

# L'inversion de priorité : Le problème



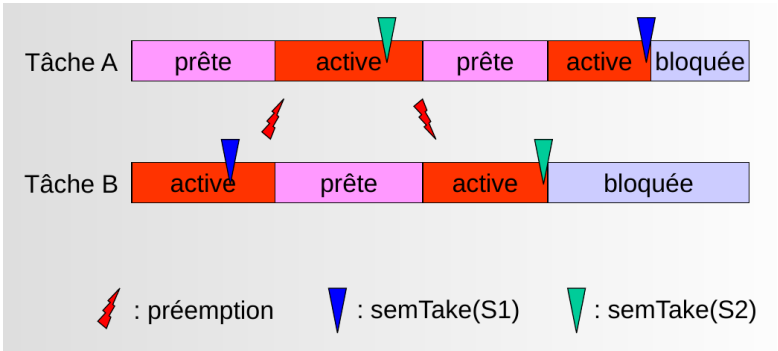

- Il faudrait que t1 ne soit pas bloquée plus longtemps que le temps nécessaire à t3 pour libérer le sémaphore

# L'inversion de priorité en action



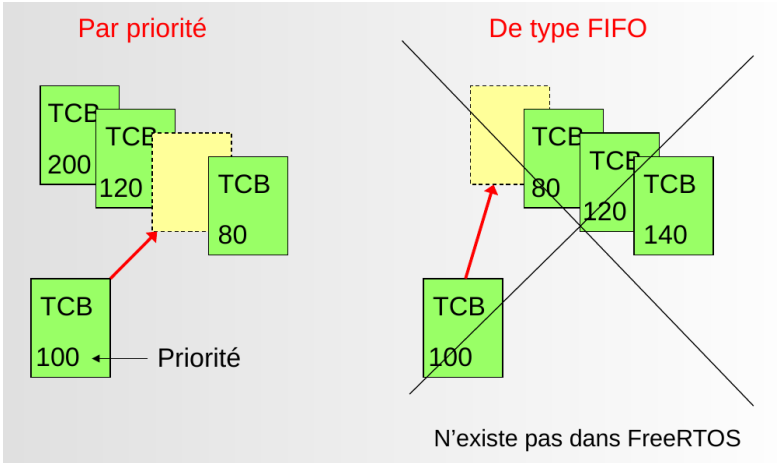

- t1 n'est plus bloquée par t2

# Problème d'interbloquage (Deadlock)




- Solution : accéder aux sémaphores dans le même ordre

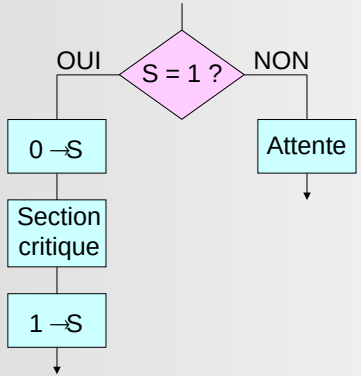
# Gestion de la file d'attente des tâches




- Si plusieurs tâches sont en attente sur un mutex :
- La tâche la plus prioritaire est débloquée en premier

- Implémentation d'un sémaphore

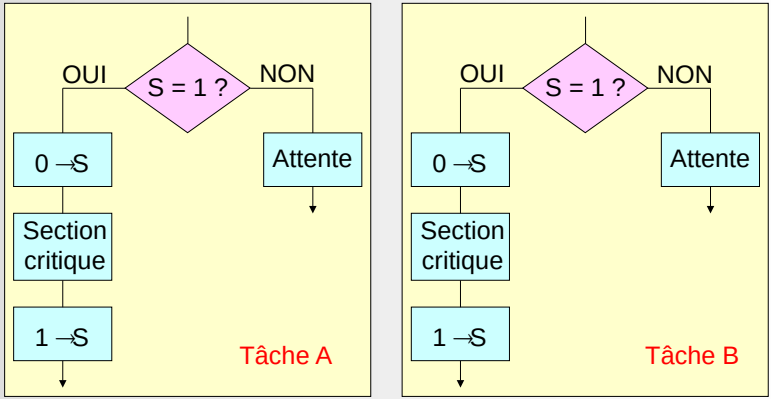
Par un simple indicateur :



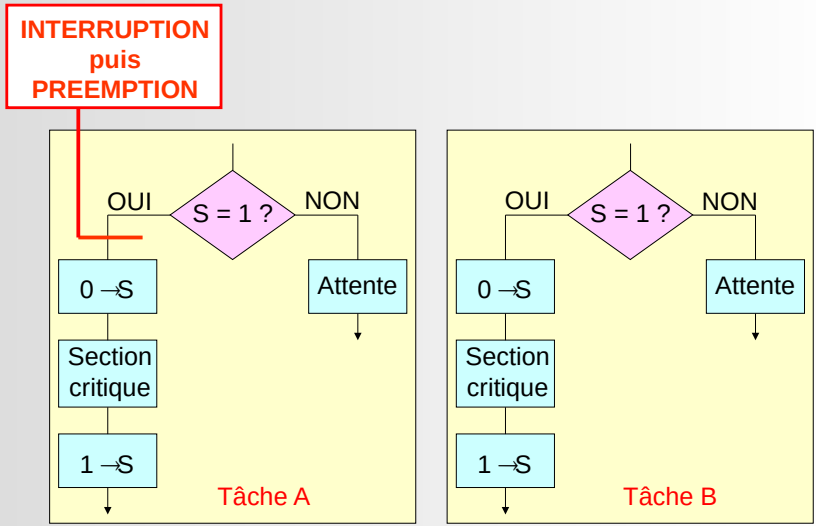

S = 1 : sémaphore libre (plein)
S = 0 : sémaphore pris (vide)

Mais mauvaise solution !

Soient deux tâches accédant à la ressource critique :

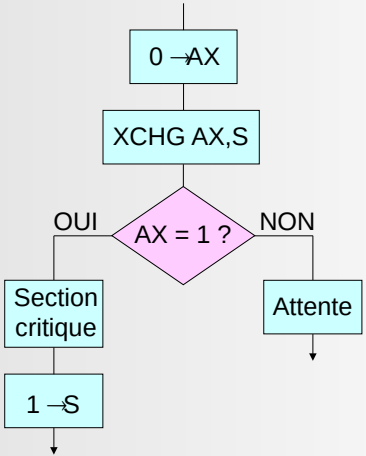



Si la tâche A est préemptée par la tâche B après le test du flag S,




le sémaphore serait pris en même temps par les deux tâches !

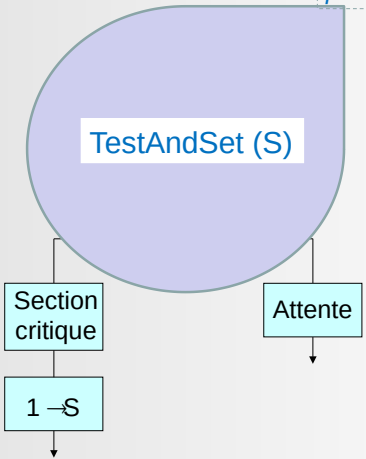
Bonne solution :




XCHG est une instruction "atomique"
SWP sur ARM

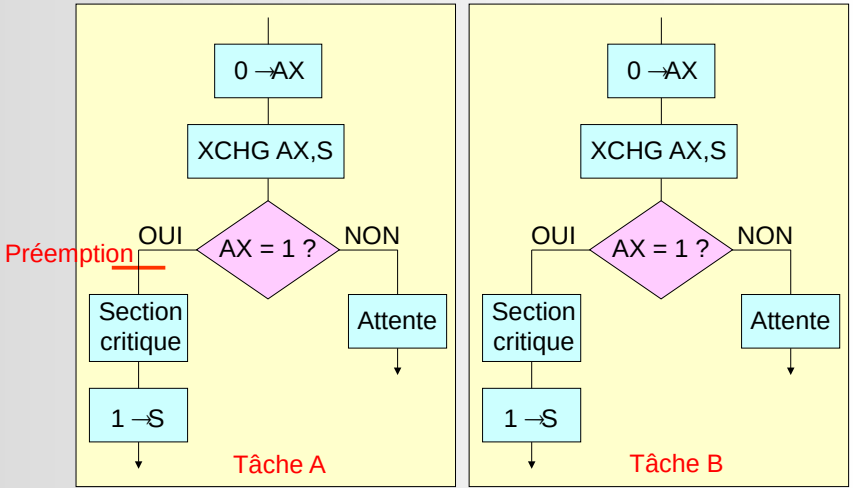
Bonne solution :

*Instruction atomique pour tester puis écrire S*




XCHG est une instruction "atomique"
SWP sur ARM

Fonctionnement avec deux tâches :



# Sémaphore à comptes

- Stocke un nombre entier positif ou nul

```
SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount,
        UBaseType_t uxInitialCount);
```

  - `uxMaxCount` : Valeur maximale du sémaphore
  - `uxInitialCount` : Valeur du sémaphore à la création
- Deux cas d'utilisation
  - Compteur de requêtes
  - Gestion de ressources