multitâche en pratique avec l'ESP32 (4)

Sémaphores binaires

Warren Gay (Canada)

En multitâche dans FreeRTOS, il faut souvent synchroniser les tâches. Le sémaphore binaire est l'un des moyens de synchronisation. Cet article explore le sémaphore binaire et l'illustre par une métaphore de couples de danseurs.

Une métaphore du sémaphore ?

Un sémaphore est une fonction de multitâche qui permet à un appelant de bloquer la poursuite de sa propre exécution jusqu'à ce que soit *donnée* la permission de continuer. L'appelant tente de *prendre* le sémaphore. Si celui-ci est déjà *pris*, la tâche appelante attend et reste donc bloquée. Imaginez un bal populaire à l'ancienne. Plusieurs garçons souhaitent danser avec une fille elle-même en train de danser avec son cavalier ; elle est *prise*, les cavaliers intéressés doivent attendre qu'elle soit prête à se *donner* à l'un d'eux. Cela nous montre bien les deux propriétés d'un sémaphore *binaire* a deux états : il est pris ou donné.

Comment le sémaphore protège-t-il ?

Un sémaphore lui-même ne peut pas contrôler l'accès à une ressource quelconque. Ce n'est que par un *accord* que la protection des ressources peut être mise en œuvre avec un sémaphore. Si les danseurs ne respectent pas le principe *pris-donné*, plusieurs garçons s'arracheront la *fille* et se battront pour l'obtenir. Le sémaphore fonctionne selon un protocole : l'accédant accepte d'utiliser toute ressource offerte uniquement s'il réussit à *prendre* le sémaphore. Il n'utilisera pas non plus la ressource avant d'avoir *pris* le prochain sémaphore disponible.

Délais d'attente pour prendre

La patience d'un garçon qui attend n'est pas illimitée. Après p. ex. dix minutes d'attente, il essaiera de trouver une autre fille avec qui danser. Les sémaphores binaires permettent à l'appelant de spécifier une telle limite de durée d'attente, exprimée en unités de temps du système (tics). Si une tentative de prendre un sémaphore dure plus longtemps que le nombre de tics spécifié, l'appel sera renvoyé avec une notification d'échec. On peut aussi enjoindre à FreeRTOS d'attendre indéfiniment que le sémaphore soit *donné* (comme attendrait un cavalier évincé). Il est possible aussi de ne spécifier aucune limite, auquel cas la tentative échoue immédiatement si le sémaphore ne peut être *pris*. Pour résumer, les sémaphores peuvent :

- bloquer pour toujours, jusqu'à ce que le sémaphore puisse être *pris* par l'appelant ;
- bloquer pour un temps déterminé, à la fin duquel l'opération se solde par un échec ;
- échouer immédiatement si le sémaphore est *pris*.

Propriété et dons

Lorsqu'un sémaphore est *pris*, on dit qu'il *appartient* à la tâche. Le garçon qui danse avec la fille la *possède* effectivement. Après avoir dansé avec elle, il peut la *donner* soit directement à un autre danseur soit en la libérant : il n'y a jamais d'attente. De même, lorsqu'il *donne* un sémaphore, il n'y a aucun délai : le *don* d'un sémaphore *possédé* est immédiat.

Le fait de *donner* un sémaphore n'implique pas nécessairement qu'il sera *pris* immédiatement par une autre tâche. Une fois que le garçon a fini de danser avec la fille, elle redevient disponible, mais il n'y a pas nécessairement d'autres preneurs intéressés. Les autres garçons peuvent avoir renoncé à attendre et trouvé entre-temps des partenaires de danse différents.

État initial

Puisque notre analogie du bal vieillot fonctionne, tirons-en profit : la fille est créée à l'état *pris* par ses parents. Ce n'est que lorsque ses parents l'autorisent à aller au bal qu'elle est *donnée* et devient disponible. Le sémaphore binaire de FreeRTOS est lui aussi créé à l'état *pris* et diffère en cela des *mutex* de Linux ou de Windows auxquels vous pouvez le comparer, car ils sont créés dans un état *donné*. Nous reviendrons sur le mutex de FreeRTOS dans un prochain article.

xSemaphoreBinaryCreate()

La création d'un sémaphore binaire dans FreeRTOS est simple :

```
SemaphoreHandle_t h;

h = xSemaphoreCreateBinary();
assert(h)
```

Il n'y a pas d'arguments à fournir, et une poignée (*handle*) est rendue au sémaphore binaire créé. Il est possible que le *handle* soit renvoyé sous la forme `nullptr` (`NULL`) en cas d'épuisement de la mémoire disponible. Il est recommandé de vérifier la valeur retournée (la macro `assert()` de `assert.h` a été utilisée ici) pour s'assurer que le sémaphore a été créé.

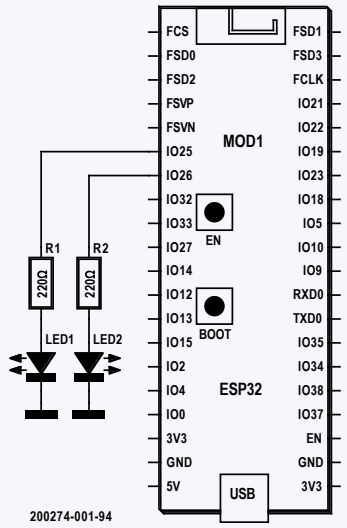

Figure 1. Schéma pour la démonstration de semaphores.ino.

xSemaphoreGive()

Donner un sémaphore est également simple :

```
SemaphoreHandle_t hMySemaphore ;
BaseType_t rc ; // code de retour
...
rc = xSemaphoreGive(hMySemaphore) ;
assert(rc == pdPASS) ;
```

L'opération *give* (= donner) peut échouer, en renvoyant `pdFAIL`, uniquement pour les raisons suivantes :
- poignée (`hMySemaphore`) invalide
- sémaphore déjà donné.

xSemaphoreTake()

Prendre un sémaphore est une opération potentiellement bloquante pour la tâche d'appel :

```
SemaphoreHandle_t hMySemaphore;
TickType_t wait = 30; // ticks
BaseType_t rc; // code de retour
```

Listage 1: Programme de démonstration semaphores.ino [1].

```
0001: // semaphores.ino
0002: // Practical ESP32 Multitasking
0003: // Binary Semaphores
0004:
0005: #define LED1_GPIO   25
0006: #define LED2_GPIO   26
0007:
0008: static SemaphoreHandle_t hsem;
0009:
0010: void led_task(void *argp) {
0011:   int led = (int)argp;
0012:   BaseType_t rc;
0013:
0014:   pinMode(led,OUTPUT);
0015:   digitalWrite(led,0);
0016:
0017:   for (;;) {
0018:     // First gain control of hsem
0019:     rc = xSemaphoreTake(hsem,portMAX_DELAY);
0020:     assert(rc == pdPASS);
0021:
0022:     for ( int x=0; x<6; ++x ) {
0023:       digitalWrite(led,digitalRead(led)^1);
0024:       delay(500);
0025:     }
0026:
0027:     rc = xSemaphoreGive(hsem);
0028:     assert(rc == pdPASS);
0029:   }
0030: }
0031:
0032: void setup() {
0033:   int app_cpu = xPortGetCoreID();
0034:   BaseType_t rc;  // Return code
0035:
0036:   hsem = xSemaphoreCreateBinary();
0037:   assert(hsem);
0038:
0039:   rc = xTaskCreatePinnedToCore(
0040:     led_task,   // Function
0041:     "led1task", // Task name
0042:     3000,       // Stack size
0043:     (void*)LED1_GPIO, // arg
0044:     1,          // Priority
0045:     nullptr,    // No handle returned
0046:     app_cpu);   // CPU
0047:   assert(rc == pdPASS);
0048:
0049:   // Allow led1task to start first
0050:   rc = xSemaphoreGive(hsem);
0051:   assert(rc == pdPASS);
0052:
0053:   rc = xTaskCreatePinnedToCore(
0054:     led_task,   // Function
0055:     "led2task", // Task name
0056:     3000,       // Stack size
0057:     (void*)LED2_GPIO, // argument
0058:     1,          // Priority
0059:     nullptr,    // No handle returned
0060:     app_cpu);   // CPU
0061:   assert(rc == pdPASS);
0062: }
0063:
0064: // Not used
0065: void loop() {
0066:   vTaskDelete(nullptr);
0067: }
```

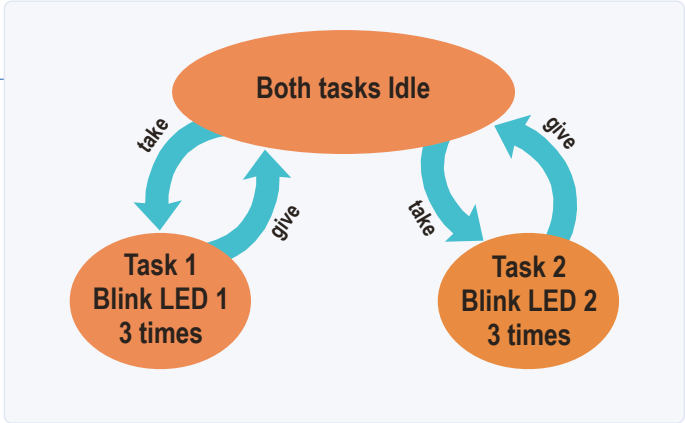

Figure 2. Les états des tâches d'exécution dans le programme semaphores.ino.

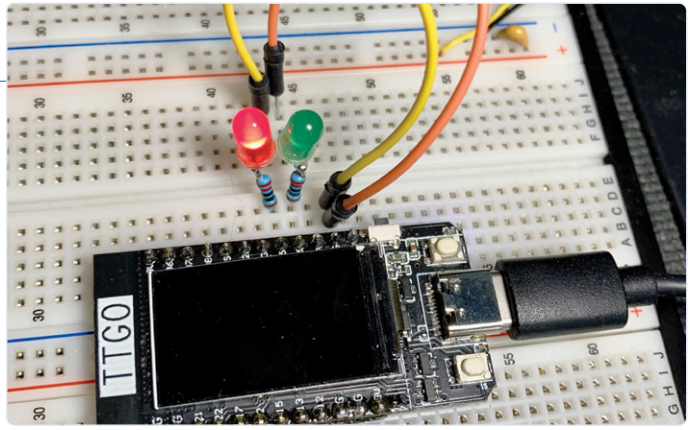

Figure 3. L'ESP32 TTGO pilotant deux LED à l'aide du programme semaphores.ino.

```
...
rc = xSemaphoreTake(hMySemaphore, wait);
// retourne pdPASS quand il est pris,
// sinon pdFAIL en cas de dépassement
```

L'opération de *prise* peut échouer si l'identifiant est invalide ou si l'appel a expiré (période définie par l'argument `wait`). Dans le cas contraire, l'appel bloquera l'exécution de la tâche appelante jusqu'à ce qu'elle ait *pris* le sémaphore.
Le paramètre `wait` peut avoir une des trois valeurs différentes selon les exigences de l'appelant :

- valeur macro `portMAX_DELAY` - attendre une éternité jusqu'à *pris*.
- valeur positive non nulle – attendre un nombre de tics correspondant à cette valeur.
- zéro - échec immédiat si le sémaphore ne peut pas être *pris*.

Démonstration

En guise de démonstration, deux tâches feront chacune clignoter leur propre LED (**listage 1**). En partageant un sémaphore binaire, une seule des tâches sera autorisée à faire clignoter sa LED à la fois. La tâche qui n'est pas en cours doit attendre que le sémaphore ait été *donné* par la tâche opposée en cours, ce qui permet de *prendre* le sémaphore une fois de plus. La **figure 1** illustre le schéma de branchement des LED pour tout ESP32 que vous choisirez d'utiliser.
La **figure 2** illustre les états de la paire de tâches s'exécutant dans le programme de démonstration. Lorsqu'une tâche prend le sémaphore, la tâche propriétaire est capable de s'exécuter et de faire clignoter sa LED, tandis que l'autre est bloquée en attente. Ce n'est que lorsque la tâche propriétaire rend le sémaphore que l'autre tâche peut le prendre et l'exécuter. La **figure 3** illustre un exemple de configuration de la carte d'expérimentation avec la carte de développement TTGO ESP32.
Dans la routine `setup()`, la ligne 36 crée le sémaphore et assigne la poignée à la variable statique globale `hsem`. Les lignes 39 à 46 créent la première tâche pour faire clignoter la LED1 (notez l'argument de la ligne 43 de l'appel de création de tâche). La broche GPIO à utiliser est passée comme un pointeur `void`. Elle est ensuite renvoyée à une valeur GPIO `int` dans la fonction de tâche `led_task()` (ligne 11), de la fonction de tâche. C'est abuser du pointeur que de l'utiliser pour passer une valeur, mais cela fonctionne tant que la valeur tient dans le même nombre de bits qu'une adresse de pointeur.
Après la création de la première tâche, nous *donnons* le sémaphore à la ligne 50. En faisant cela avant la création de la deuxième tâche, il est probable que la première tâche acquière le sémaphore en premier. Les lignes 53 à 60 créent ensuite la deuxième tâche. Les deux tâches exécutent le même code de fonction `led_task()`, mais avec des valeurs d'argument différentes spécifiant la broche GPIO de la LED à utiliser. Les lignes 14 et 15 configurent la LED utilisée dans le cadre de la tâche. La tâche entre alors dans une boucle sans fin à partir de la ligne 17. La première étape effectuée dans cette boucle est de *prendre* le sémaphore (ligne 19). Une seule tâche à la fois y parviendra.
La tâche qui réussit à prendre le sémaphore se poursuivra par la boucle des lignes 22 à 25. Cette boucle fait clignoter trois fois la LED de la tâche. Ensuite le sémaphore est *donné* dans la ligne 27, permettant à l'autre tâche de *prendre* le sémaphore. De cette façon, les tâches prennent à tour de rôle chacune le contrôle de l'autre.

Résumé

Le sémaphore binaire utilisé dans cette démonstration a fonctionné de deux manières différentes. Avant la première opération *give* de la fonction `setup()`, aucune des deux tâches ne peut être exécutée. Le sémaphore se comportait donc comme une barrière, car aucune tâche ne pouvait être exécutée. Après cette première opération *give* (= donne), l'une des deux tâches *prend* ce sémaphore, fait clignoter sa LED, puis *rend* le sémaphore. En opérant de cette manière, le sémaphore semble se comporter comme un *mutex*. Les deux tâches tentent continuellement d'exécuter leur code, mais, par l'exclusion mutuelle du sémaphore, une seule tâche à la fois est autorisée à faire clignoter sa LED.
Il est important de retenir que les sémaphores binaires sont créés à l'état *pris* (contrairement au mutex). Si le sémaphore de cette démonstration n'avait pas été *donné* initialement dans la routine `setup()` (ligne 50), les deux tâches seraient restées bloquées en attente sans fin. Dans FreeRTOS il existe d'autres primitives de synchronisation (y compris le *mutex*), mais le simple sémaphore binaire est parfois suffisant.

200274-03

LIEN

[1] https://github.com/ve3wwg/esp32_freertos/blob/master/semaphores/semaphores.ino