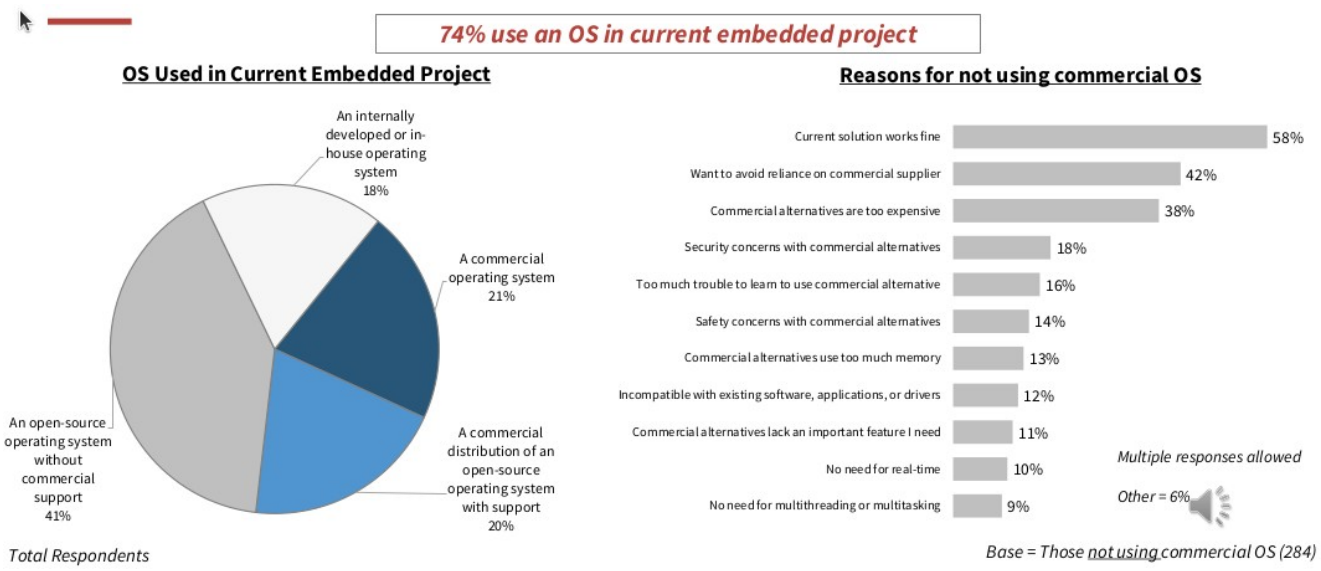

free
RTOS

SOMMAIRE

1. PREAMBULE

Durant la lecture de cette partie, nous allons nous intéresser à l'exécutif temps réel ou RTOS (Real Time Operating System) FreeRTOS. La traduction littérale en Français de RTOS est Système d'Exploitation Temps Réel. Il serait néanmoins plus rigoureux d'appeler ce type d'outil scheduler ou *ordonnanceur*, plus que système d'exploitation, à l'image par exemple de systèmes comme GNU/Linux, Android, Windows, MacOS. Néanmoins, l'acronyme OS est un abus de langage fréquemment utilisé dans le monde de l'embarqué pour parler des systèmes d'exploitation temps réel.

Un scheduler ou ordonnanceur nous propose des services logiciel (tâches, queues de messages, sémaphores ...). Bien maîtrisé, il s'agit d'une aide précieuse durant les phases de développement d'un projet (évolutivité, modélisation, gestion optimale des ressources CPU ...). Néanmoins, encore beaucoup de systèmes autour de nous fonctionnent sans OS (cf. sondage ci-dessous, UBM Tech). La question d'utiliser ou pas un ordonnanceur dépend bien sûr de l'application. Dès que les spécifications fonctionnelles d'une application mettent en avant un certain nombre (difficile à estimer) de traitements pouvant potentiellement s'exécuter en parallèle, la question d'utiliser un RTOS se pose. Pour des applications communicantes (Bluetooth, WIFI, réseaux de terrains ...), l'utilisation d'un OS peut s'avérer très utile (www.eetimes.com).

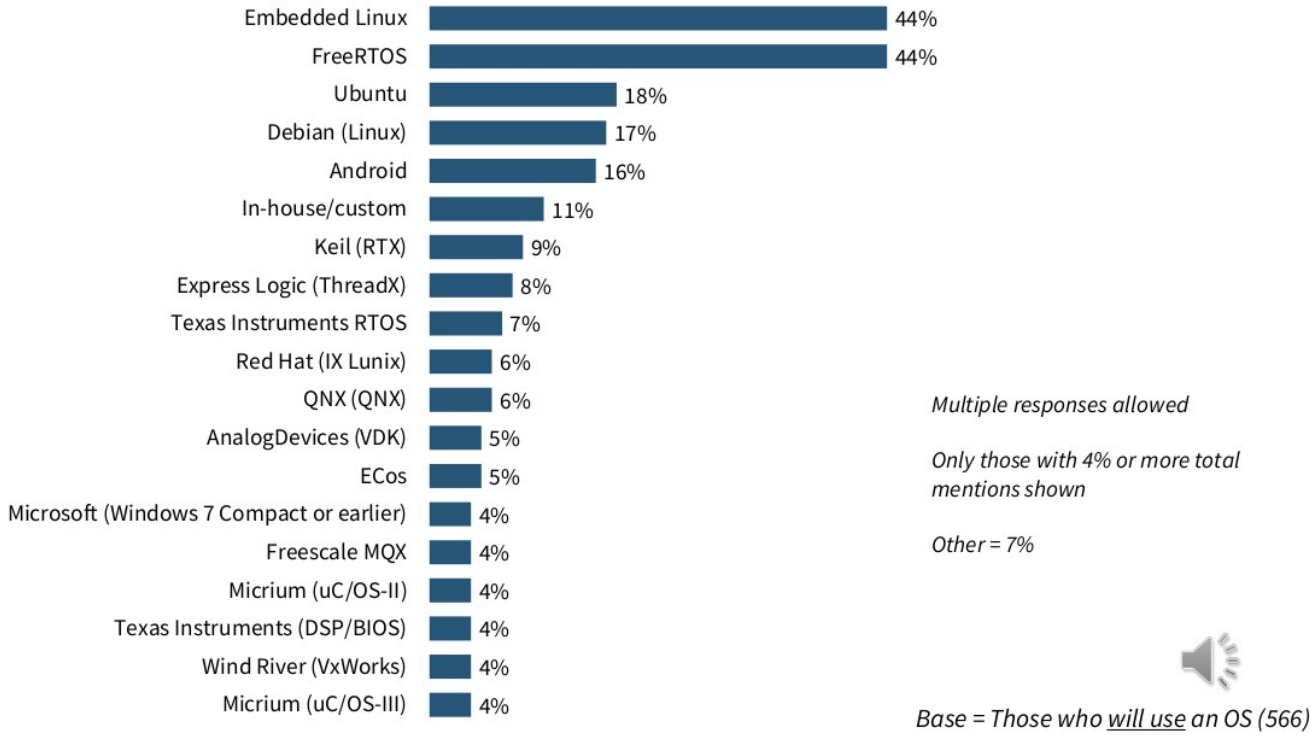


Il existe un grand nombre de RTOS, libres, open Sources et propriétaires (RTX, MicroC/OS III, FreeRTOS, VxWorks...) possédant des jeux d'avantages et d'inconvénients différents afin de s'adapter au très grand nombre de problématiques du marché. Nous avons de notre côté à l'école porté notre attention sur *FreeRTOS*, un petit scheduler temps réel offrant un certain nombre d'avantages. La documentation de FreeRTOS est directement accessible en ligne depuis le site internet (http://www.freertos.org/).

En 2024, FreeRTOS est le RTOS leader sur le marché de l'embarqué. FreeRTOS est libre de droit d'utilisation et open source (sous licence MIT). Il est officiellement supporté par 34 architectures (TI, Microchip, Atmel, NXP, Intel ...) et 18 chaînes de compilation. Il a par exemple été téléchargé plus de 100000 fois l'année passée, cela implique une communauté assez riche d'utilisateurs. Il existe de plus en tout 4 variantes de FreeRTOS :

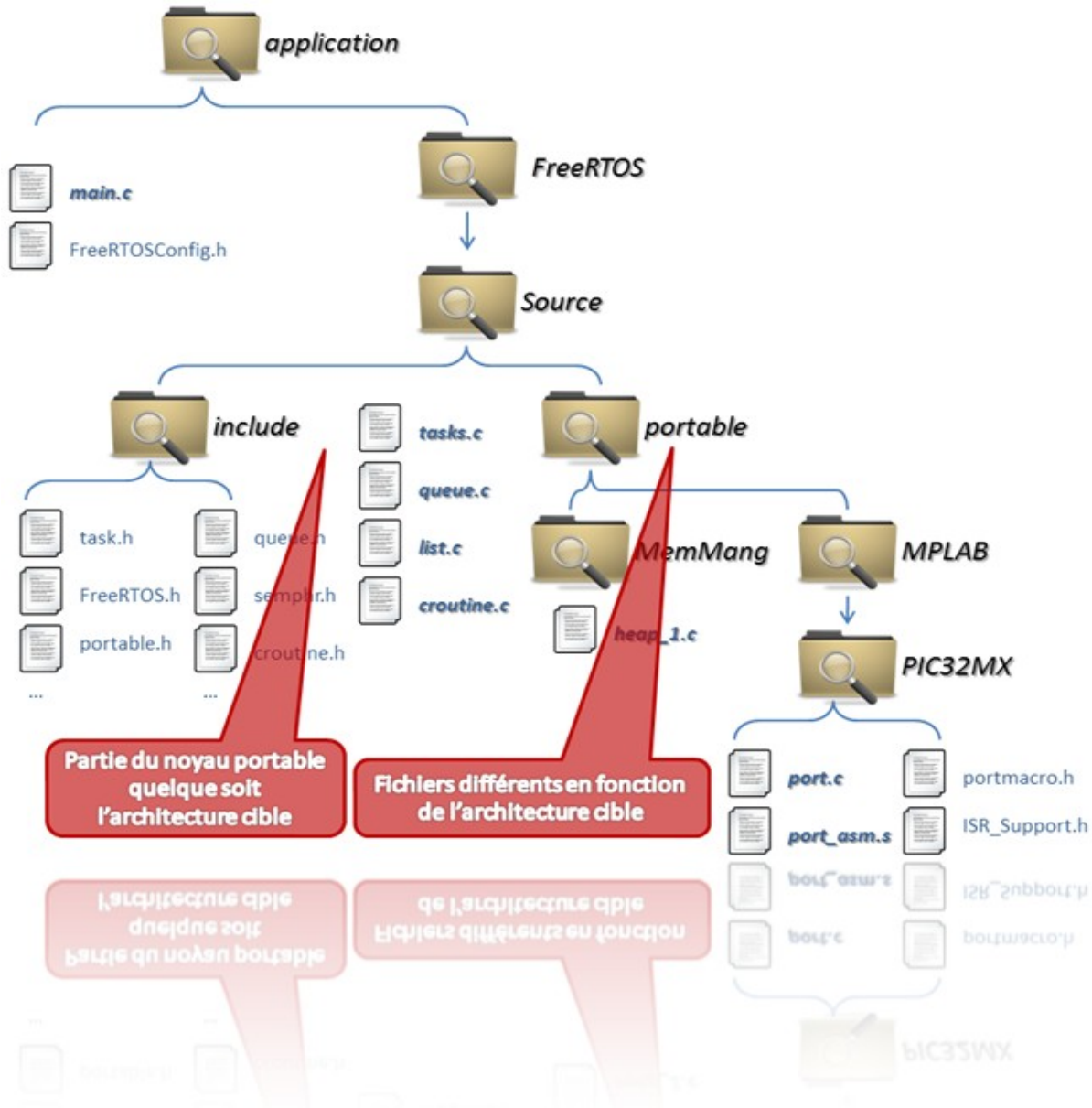
- *FreeRTOS :* cf. ci-dessus
- *FreeRTOS + Trace :* idem FreeRTOS avec des outils propriétaires permettant d'instrumenter le code. Par exemple des outils graphique de trace.
- *OpenRTOS :* version commerciale sous licence de FreeRTOS
- *SafeRTOS :* version certifié SIL3 TUV

De plus en 2024, FreeRTOS est le RTOS actuellement le plus utilisé et celui le plus regardé pour démarrer de nouveaux projets. Observons le marché en 2023 des OS et RTOS pour l'embarqué (www.eetimes.com) :



2. SYSTEME D'EXPLOITATION TEMPS REEL

FreeRTOS, comme n'importe quel autre noyau, est un outil purement logiciel, ce n'est qu'un système de fichiers. FreeRTOS nous propose une API de programmation pour gérer un environnement multitâches. La notion de tâche sera présentée par la suite. Vous trouverez ci-dessous le systèmes de fichiers du noyau :

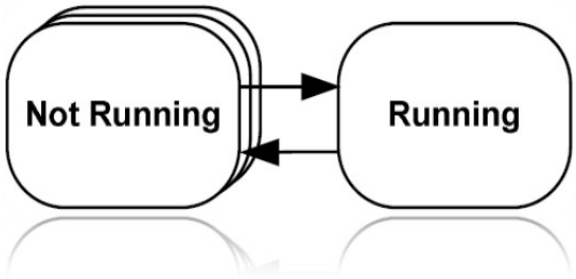


2.1. Tâche

Dans un environnement multitâches, l'application est découpée en plusieurs tâches correspondant à des actions à effectuer. Le rôle de l'ordonnanceur est alors de gérer cet environnent sachant que potentiellement, plusieurs d'entre-elles chercheront à s'exécuter en même temps. N'oublions pas qu'un PIC32 ne possède qu'un seul CPU, l'ordonnanceur ne pourra donc donner la main qu'à une seule tâche à la fois. Vous constaterez qu'à première vue, une tâche ressemble beaucoup à une simple fonction. Nous verrons par la suite qu'une tâche est bien plus que ça. Observons un exemple de tâche sous FreeRTOS :

```
void Task1( void *pvParameters ){

        // Faire toujours
        for( ;; ){
              // code utilisateur
            }
}
```

Une tâche est le plus souvent implémentée sous forme d'une boucle infinie. De plus, l'ordonnanceur a la capacité de pouvoir faire passer une tâche dans différents états :



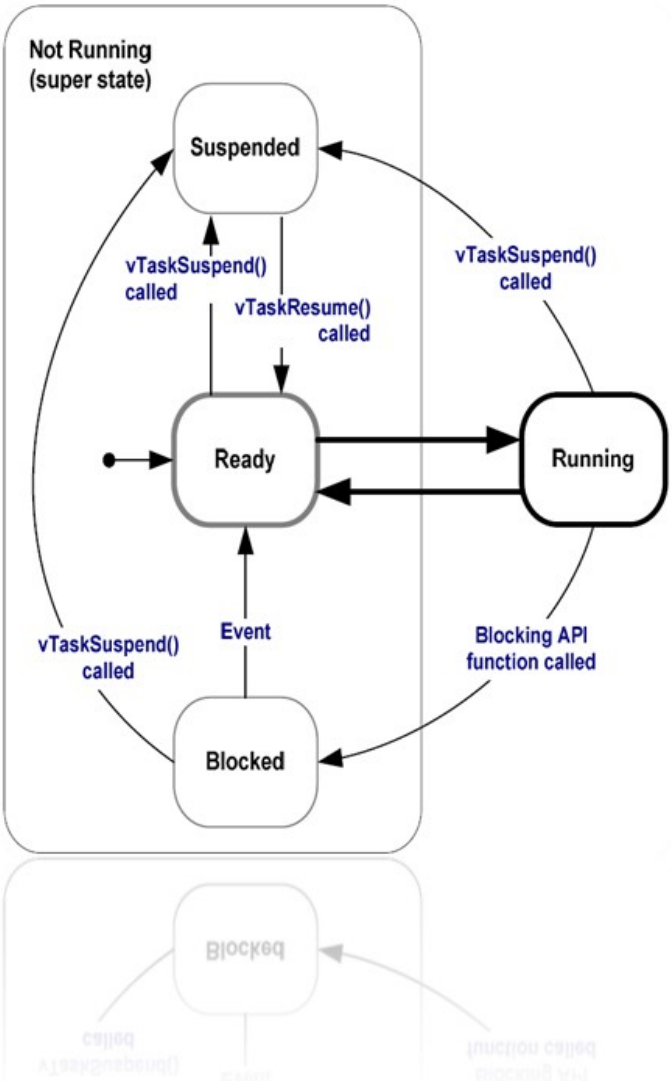
- *Running (en cours) :* il s'agit de la tâche en cours d'exécution par le CPU. Une seule tâche peut-être dans cet état.

- *Ready (prêt) :* les tâches dans cet état sont prêtes à être exécutées. Il suffit que la tâche à l'état en cours redonne la main à l'ordonnanceur ou que celui-ci la reprenne et selon le contexte d'exécution (priorité supérieure, round-robin ...) une nouvelle tâche s'exécutera.

- *Blocked (bloqué) :* les tâches dans cet état sont en attentes d'un événement pour se réveiller (queue de messages, sémaphores, timeout ...). Une fois l'événement arrivé, la tâche concernée repasse alors à l'état prêt.

- *Suspended (suspendu) :* Cet état, peu utilisé, est propre à FreeRTOS et est à manier avec précaution. Une tâche dans cet état n'est plus vue de l'ordonnanceur.

États d'une tâche sous FreeRTOS :



Une différence très importante entre une simple fonction et une tâche est qu'à chaque tâche est associée un TCB (Task Control Block). Un TCB est une structure de données décrivant une tâche (descripteur de tâche). L'ordonnanceur utilise ensuite les TCB pour le management de son environnement multitâches. Sous FreeRTOS, un TCB comporte par exemple :

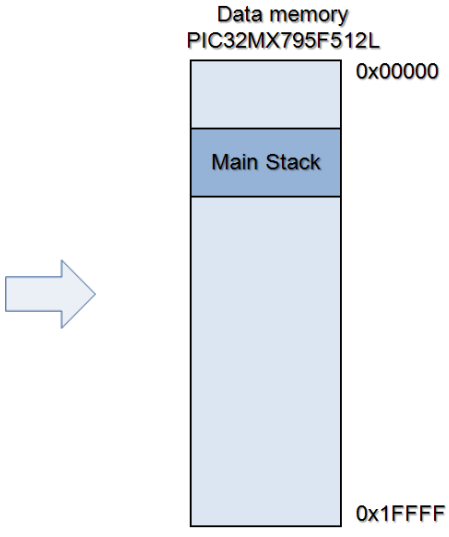
- la priorité de la tâche
- le ou les événements qu'elle attend
- le pointeur de base de la pile associée à la tâche
- le pointeur de sommet de pile
- ...

2.2. Gestion mémoire

Ce sont les TCB que l'ordonnanceur analyse afin de savoir à quelle tâche prendre ou donner du temps CPU. Depuis la première année, tous les programmes que vous avez développés sur MCU n'étaient constitués que d'un main() voir d'ISR's, aucun OS n'était embarqué. Effectuons quelques rapides rappels sur le fonctionnement du main() et notamment les mécanismes de gestion mémoire réalisés conjointement par la chaîne de compilation et le processeur.

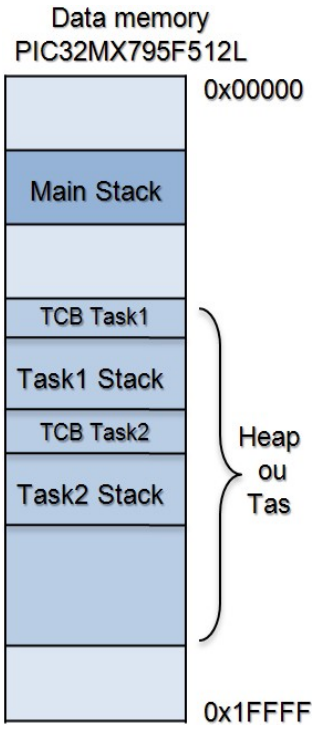
- Sans noyau, si une application n'est constituée que d'un main(), la mémoire des données peut-être découpée en deux grandes zones. La zone où se trouvent les variables statiques (variables globales, locales static ...) et celle nommée pile ou stack où sont notamment gérées les variables locales (dynamiques). Selon l'application, un tas peut également être utilisé. Toutes les variables locales du main() ou des fonctions appelées depuis le main() sont allouées dynamiquement dans la pile du main() ou pile système. Il en est de même des variables locales, paramètres de fonction et des sauvegardes de contexte des fonctions d'interruption. Prenons un exemple de mapping mémoire pour une application sans OS et sans Tas système :

```
unsigned char stringBuffer[100] = "FreeRTOS test !";

/**
* @fn main
* @brief main entry point
*/
int main(void){

    // Configurations matérielles
    UserInit();

   // envoi d'une chaîne de caractères via UART
   UARTputs(stringBuffer);

   // On reste bloqué ici ... pour le moment !
   while(1);
}
```



- Sous FreeRTOS, le noyau utilise une zone de taille configurable nommée Tas ou Heap. Attention, en fonction de la stratégie de gestion du tas choisie (fichiers heap_1.c, heap_2.c, heap_3.c ou heap_4.c) le tas peut-être le tas système ou une simple très large variable globale. *Chaque tâche possède sa propre pile dans le Tas du kernel. Chaque tâche possède donc un environnement d'exécution indépendant.*

```
void Task1( void *pvParameters );
void Task2( void *pvParameters );

/**
* @fn main
* @brief main entry point
*/
int main(void){
   // Création de 2 tâches
   xTaskCreate( Task1, "Task 1", configMINIMAL_STACK_SIZE, ...);
   xTaskCreate( Task2, "Task 2", configMINIMAL_STACK_SIZE, ...);

   // Démarrage ordonnanceur
   vTaskStartScheduler();

   while(1);   // ... Nous ne reviendrons jamais ici !
}

void Task1( void *pvParameters ){
        // Faire toujours
        for( ;; ){    ...   }
}

void Task2( void *pvParameters ){
        // Faire toujours
        for( ;; ){    ...   }
}
```



2.3. Gestion du Tas

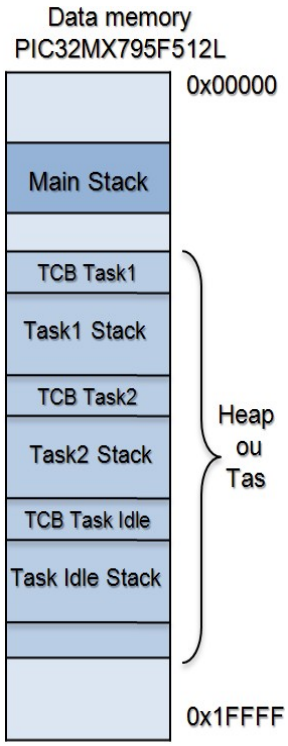
Dans le cas de FreeRTOS, le Tas ou heap n'est qu'un tableau déclaré en variable globale (stratégies heap_1.c et heap_2.c). En tant que développeur, vous n'aurez pas directement accès à ces fonctions d'allocation. Durant la création d'une tâche, le noyau génère un espace pour la TCB et la pile de celle-ci dans le Tas.

Attention, en fonction de l'application et du MCU il faut être très prudent à l'espace alloué au Tas et aux piles de chaque tâche. Par exemple, si la taille d'une pile est trop faible, en cas de débordement de pile (stack overflow) nous pouvons écraser le contenu de la TCB de la tâche suivante dans le Tas ... et donc faire tomber l'application. FreeRTOS propose 3 principales techniques pour la gestion du Tas. Il suffit pour cela d'inclure à votre projet l'un des trois fichiers sources ci-dessous :

- *heap1.c :* seules des allocations dynamiques sont possibles (exemple, créations de tâches). Nous utiliserons ce fichier durant la trame de TP.
- *heap2.c :* allocations et désallocations dynamiques sont possibles (exemple, créations et suppressions de tâches). Soyez très prudent avec la désallocation de ressources et la gestion par le kernel d'une mémoire fragmentée.
- *heap3.c :* idem heap2.c, mais utilise les API standards malloc() et free() proposées par la chaîne de compilation.

Exemple de mapping mémoire après la création de 2 tâches sous FreeRTOS :

```
void Task1( void *pvParameters );
void Task2( void *pvParameters );

/**
* @fn main
* @brief main entry point
*/
int main(void){

   // Création de 2 tâches
   xTaskCreate( Task1, "Task 1", configMINIMAL_STACK_SIZE, ...);
   xTaskCreate( Task2, "Task 2", configMINIMAL_STACK_SIZE, ...);

   // Démarrage ordonnanceur
   vTaskStartScheduler();

   // Nous ne reviendrons jamais ici !
   while(1);
}


void Task1( void *pvParameters ){
        // Faire toujours
        for( ;; ){
             // code utilisateur
           }
}


void Task2( void *pvParameters ){
        // Faire toujours
        for( ;; ){
             // code utilisateur
           }
}
```



Vous constaterez que nous n'avons créé que deux tâches alors que le noyau en a créé trois. En effet juste avant de démarrer l'ordonnanceur, le noyau crée une ultime tâche nommée tâche Idle qui est la tâche de plus basse priorité de l'application. Lorsque aucune tâche ne tourne (tâches bloquées), c'est en fait la tâche Idle qui est en cours d'exécution. Nous ne reviendrons donc jamais dans le main(). Nous verrons dans la trame de TP qu'il est d'ailleurs possible de détourner la tâche Idle. Pour information, certains noyau considèrent le main() comme une tâche, ce n'est pas le cas de FreeRTOS.

2.4. Création de tâche

Nous avons précédemment vu qu'à la création d'une tâche, le noyau alloue un espace pour la TCB et la pile dans le Tas. Découvrons maintenant le rôle des paramètres passés à l'API de création de tâche (xTaskCreate()) :

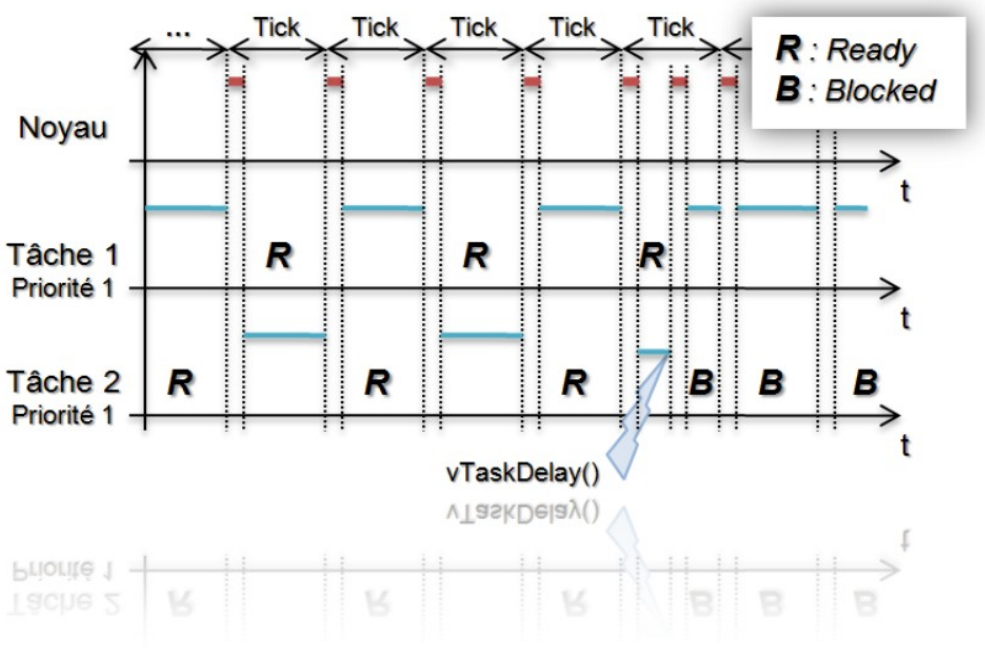
```
void Task1( void *pvParameters );
const char *pvParametersTask1 = "Passage d'un pointeur sur une chaîne de caractères !";

/**
* @fn main
* @brief main entry point
*/
int main(void){

   // Création d'une  tâche
   xTaskCreate( Task1,                  // Pointeur sur la fonction implémentant la tâche
        "Tache 1",                      // Chaîne de caractères associée à la tâche (debug).
         configMINIMAL_STACK_SIZE,      // Taille de la pile associée à la tâche
         pvParametersTask1,             // Paramètre passé à la tâche
         tskIDLE_PRIORITY + 1,          // Priorité associée à la tâche
         NULL);                         // "handle" pour la gestion de la tâche
  ...
}
```

- *Task1 :* pointeur sur la fonction implémentant le tâche
- *''Tache 1'' :* chaîne de caractères associée à la tâche (sauvée dans la TCB). Utilisé par des outils de trace ou pour de l'instrumentation de code (par exemple, stack overflow).
- *configMINIMAL_STACK_SIZE:* taille de la pile associée à la tâche. Cette macro est définie dans le fichier *FreeRTOSConfig.h*. Il s'agit par défaut de la taille de la pile pour la tâche Idle. Attention, cette taille est donnée en ''Words'' et non en octets. La taille d'un Word dépend de l'architecture matérielle utilisée. Dans notre cas un Word = 32bits (les PIC32MX sont des MCU's 32bits).
- *pvParametersTask1:* possibilité de passer un paramètre à la tâche.
- *tskIDLE_PRIORITY + 1 :* priorité de la tâche. tskIDLE_PRIORITY ou ''0'' est la priorité minimale qui correspond à la priorité de la tâche Idle. La priorité maximale vaut *configMAX_PRIORITIES* définie dans *FreeRTOSConfig.h.*
- *NULL :* il s'agit d'un outil de préemption permettant de passer la tâche en cours de création à l'état suspendu. A manipuler avec précaution.

2.5. Mode préemptif

En mode coopératif, chaque tâche doit explicitement permettre à une autre tâche de s'exécuter. Ce mode de fonctionnement était encore très rencontré notamment jusqu'à MS-DOS et Mac OS 9 qui étaient tout deux des OS coopératifs. Cependant ce mode peut amener de gros problèmes de robustesse. Par exemple, imaginons que nous restons bloqué dans une tâche (bug). Les autres tâches ne pourront donc jamais prendre la main ... nous venons de faire tomber l'application.

En mode préemptif, l'OS prend périodiquement la main et force un réordonnancement. La référence de temps utilisée est nommée tick (timer clock). Ce mode de fonctionnement est plus robuste, les principaux OS sur ordinateur sont préemptif (Windows 8, Mac OS X, Linux ...). Exemple d'exécution en mode préemptif sous FreeRTOS :



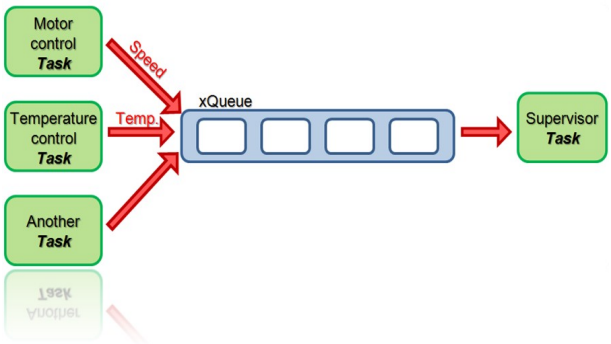
Dans notre cas, FreeRTOS configure et utilise un timer du processeur. Ce timer interrompt périodiquement le programme en cours d'exécution en envoyant une demande d'interruption au CPU. Une demande d'interruption ou IRQ peut arriver n'importe quand. Ce qui signifie que, sauf dans certains cas (par exemple sections critiques ...), n'importe quelle tâche peut être interrompue à n'importe quel moment pour donner la main à l'ordonnanceur. Quelque soit la priorité de la tâche en cours d'exécution.

3. QUEUE DE MESSAGES

Une queue de messages (messages queue ou queue) ou boîte aux lettres (mailbox) est un outil logiciel asynchrone permettant des communications voir synchronisations entre tâches. Il lui est associé deux files d'attentes, une préservant l'ordre d'arrivée des messages et une seconde préservant l'ordre d'arrivée des tâches. Les deux appellations sont très rencontrées. A titre indicatif, DSP/BIOS ou SYS/BIOS le RTOS embarqué sur le DSP TMS320C6xxx de Texas Instruments étudié en deuxième année parle de ''Mailbox'' contrairement à FreeRTOS qui parle de queue. Nous parlerons donc de préférence de queues de messages durant cet enseignement.

Une queue de messages (ou file d'attente) contient un nombre fini d'éléments dont la taille est configurable. Elle est de façon générale régit par le principe de fonctionnement d'une FIFO (First In First Out). Le premier entré dans la file sera le premier à en sortir. Nous constaterons que FreeRTOS propose si nécessaire des alternatives à ce fonctionnement. De plus, sachez que FreeRTOS créé les queues de messages et alloue les ressources mémoire nécessaires dans le Tas.

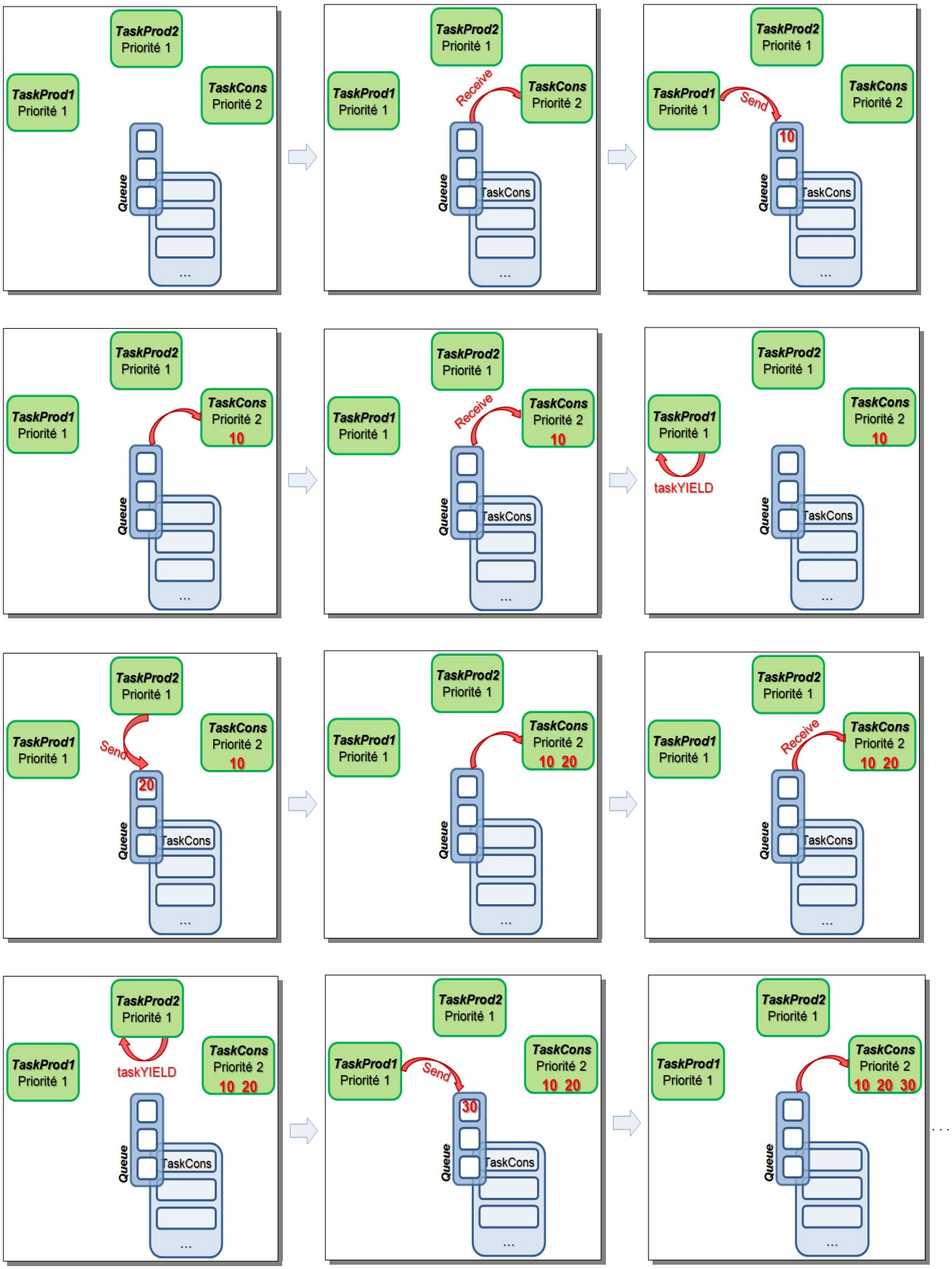
Une queue de messages peut avoir plusieurs écrivains et plusieurs lecteurs. Cependant, de façon générale, elles sont utilisées avec des écrivains multiples et un seul lecteur. Les écrivains sont les tâches pouvant écrire dans une queue de messages. Les lecteurs sont donc celles susceptibles de la lire. Un élément lu est retiré de la file d'attente. Si un lecteur cherche à lire une queue de messages vide, il se fait bloquer jusqu'à l'arrivée d'un nouveau message. Pour des tâches de même priorité, ce sera la première bloquée qui sera la première réveillée (FIFO). FreeRTOS étant un exécutif temps réel, si une tâche de plus haute priorité se fait bloquer, elle passe en tête de file. Exemple de scénario :



3.1. API de FreeRTOS

- *xQueueSend() ou xQueueSendToBack() :* envoi d'une donnée vers une queue de messages à la suite des éléments déjà présents. Cette fonction n'est bloquante que si la queue de messages est pleine.
- *XQueueReceive() :* Récupère la donnée en tête d'une queue de messages. L'élément lu est retiré de la file d'attente. Le second élément passe alors en tête de file. Cette fonction n'est bloquante que si la queue de messages est vide.

Illustrons maintenant ces concepts. Dans l'exemple ci-dessous le noyau travaille en mode coopératif. Ce scénario présente une application avec deux écrivains et un lecteur :



FreeRTOS propose également des fonctions permettant de rendre prioritaire des données envoyées à une queue de messages. Par exemple xQueueSendToFront() écrit une donnée en tête de file d'attente.

3.2. Timeout

Certaines fonctions pour la gestion de queue de messages et de sémaphores utilisent un Timeout ou temps mort. Lorsqu'une tâche est bloquée après l'appel de l'une de ces fonctions, celle-ci se réveillera (passage à l'état prêt) automatiquement après un laps de temps nommé Timeout, même si l'événement attendu n'est pas arrivé. L'utilisation du Timeout permet de garantir la robustesse d'un code en forçant par exemple le réveil d'une tâche en attente d'un événement devant être envoyé par une tâche boguée. Prenons l'exemple de l'API suivante :

```
portBASE_TYPE xQueueReceive(
        xQueueHandle xQueue,
        void *pvBuffer,
        portTickType xTicksToWait
 );
```

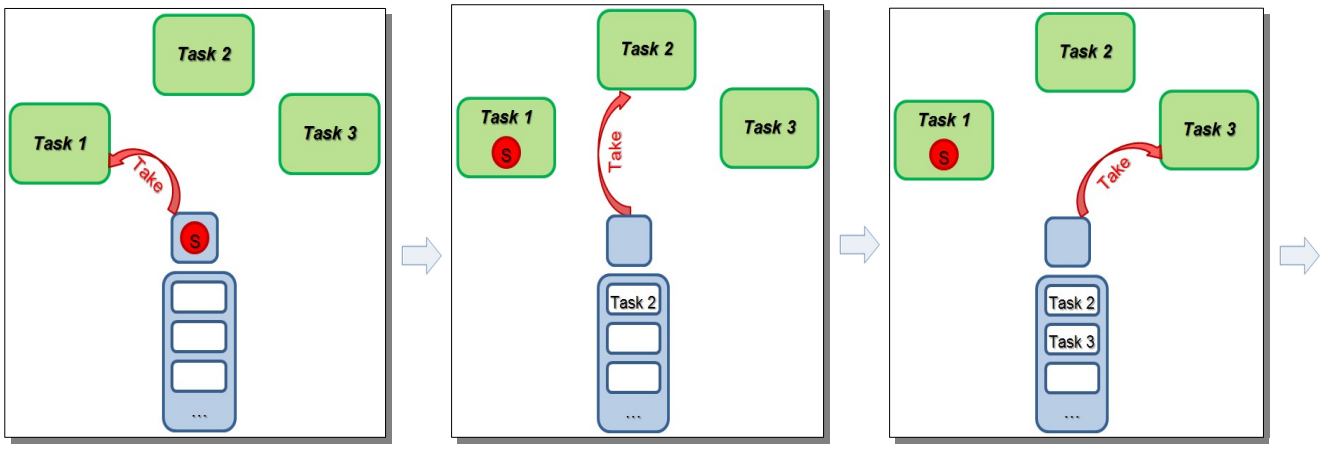
Le paramètre xTicksToWait permet de fixer le Timeout. Sous FreeRTOS, le Timeout est donné en ticks. Si nous ne souhaitons pas utiliser cette fonctionnalité (Timeout infini), il suffit de passer en paramètre la macro *portMAX_DELAY* (définie dans *portmacro.h*).

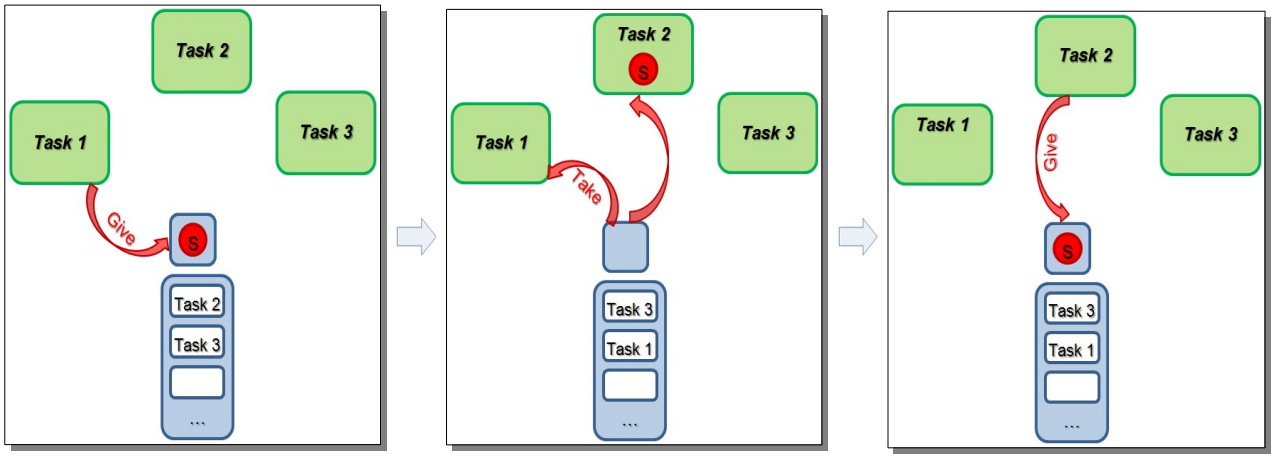
4. SEMAPHORES

Un Sémaphore est un outil logiciel couramment (mais pas seulement !) utilisé pour la protection de ressources partagées (variables, périphériques, espaces mémoires ...). Le principe de fonctionnement des sémaphores est très proche de celui des queues de messages. La preuve en est, sous FreeRTOS vous ne trouverez aucun fichier source propre aux sémaphores. Les API pour la gestion des sémaphores ne sont que des macros appelant des fonctions propres aux queues de messages (queue.c).

4.1. Sémaphore binaire

Un sémaphore binaire peut-être vu comme une variable booléenne associée à une file d'attente préservant l'ordre d'arrivée des tâches (cf. queues de messages). Un sémaphore binaire Pris (Take) par une tâche ne peut plus l'être par une autre, ni même par elle même tant qu'il n'est pas explicitement Vendu (Give). Une tâche cherchant à prendre un sémaphore binaire déjà pris se verra bloquée (blocked) jusqu'à ce que la ressource soit relâchée. Prenons un exemple de scénario :



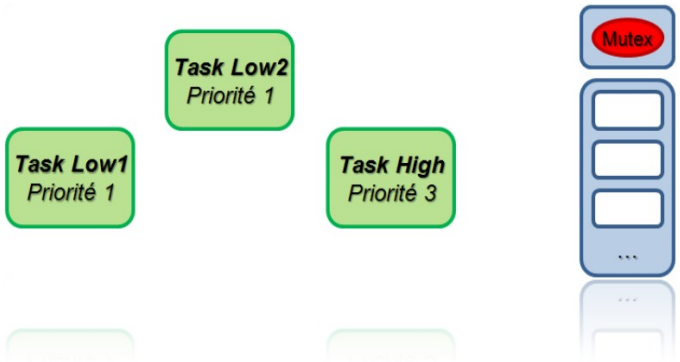


...etc

Le scénario précédent reflète par exemple une application où 3 tâches de même priorité cherchent simultanément à envoyer des données via un UART. La ressource partagée est alors l'UART qui est protégé par un sémaphore binaire. Durant ce laps de temps nous nous trouvons dans une section critique qui peut cependant être préemptée par le noyau. Les différentes tâches utiliseront donc l'UART à tour de rôle (exclusion mutuelle) .

4.2. MUTEX

Mutex signifie MUTual EXclusion (ou exclusion mutuelle), il s'agit du concept très rapidement présenté durant le scénario présenté ci-dessus. Une exclusion mutuelle traduit la notion de protection de ressources partagées. Sous FreeRTOS, elle peut notamment être réalisée à partir d'un sémaphore binaire (inversion de priorité) ou d'un mutex (héritage de priorité). Pour FreeRTOS, le principe de fonctionnement de ces deux outils est exactement le même à ceci près que le Mutex effectue un héritage de priorité. Illustrons le concept d'héritage de priorité :



- *Héritage de priorité :* La tâche *Task Low2* vient de prendre une ressource (Mutex) et elle ne la rendra qu'une fois avoir fini ce pourquoi elle l'a pris. Cependant la tâche *Task Low1* est également prête (état ready), l'ordonnanceur applique donc le round-robin et partage le temps CPU entre les deux tâches de même priorité.

  Imaginons maintenant que la tâche *Task High* (de priorité supérieure) cherche également à prendre la ressource (Mutex). Ceci est impossible, le MUTEX ayant déjà été pris et elle se fait donc bloquer, c'est ce que l'on appel l'inversion de priorité. Une tâche de haute priorité se fait bloquer par une tâche de plus basse priorité, le système de priorité choisi par le développeur est inversé. Cependant avec l'héritage de priorité, la tâche *Task Low2* hérite temporairement de la priorité de *Task High* et pourra donc potentiellement finir de s'exécuter plus rapidement que sans héritage (plus de round-robin avec la tâche de même priorité). De façon général, dans un système temps réel une tâche ne doit *jamais* (ou le moins longtemps possible !) rester bloquée par une tâche de moindre priorité.

4.3. Sémaphore à compteur

Le principe de fonctionnement d'un sémaphore à compteur est identique à celui d'un sémaphore binaire sauf que la ressource protégée peut maintenant être prise (Take) à plusieurs reprise. Elle peut être prise soit par la même tâche, soit par une nouvelle. Exemple de sémaphore à 4 jetons :