FreeRTOS

FreeRTOS est un système d'exploitation embarqué multitâches temps réel préemptif supporte actuellement 35 architectures. Il est aujourd'hui parmi les plus utilisés dans le marché des systèmes d'exploitation temps réel.

Gestion des tâches

Tâche FreeRTOS

Pour développer une application basée sur un OS, on décompose l'application en un ensemble de tâches. Dans FreeRTOS une tâche est fonction C contenant une boucle infinie et ne renvoie pas un résultat.

```
void vATaskFunction( void *pvParameters )
{
        ⋮
        for( ; ; )
        {

        }
}
```

Etat d'une tâche

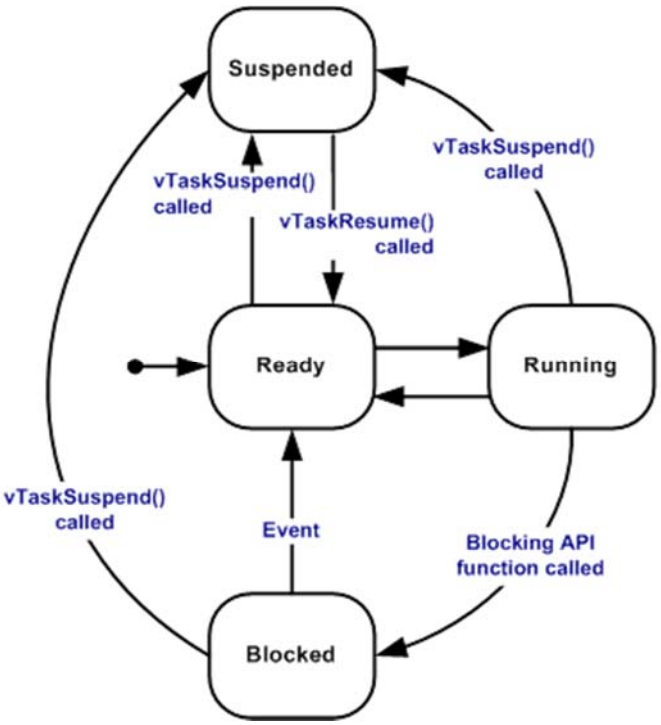
Une tâche FreeRTOS peut se trouver dans l'un des états suivants :

Prête (Ready) : une tâche qui possède toutes les ressources nécessaires à son exécution. Elle lui manque seulement le processeur.

Active (Running) : Tâche en cours d'exécution, elle est actuellement en possession du processeur.

Attente (Blocked) : Tâche en attente d'un événement (queue de messages, sémaphores, timeout ...). Une fois l'événement arrivé, la tâche concernée repasse alors à l'état prêt.

Suspendu (Suspended) : tâche à l'état dormant ; elle ne fait pas partie de l'ensemble des tâches ordonnançables.



Task Control Block (TCB)

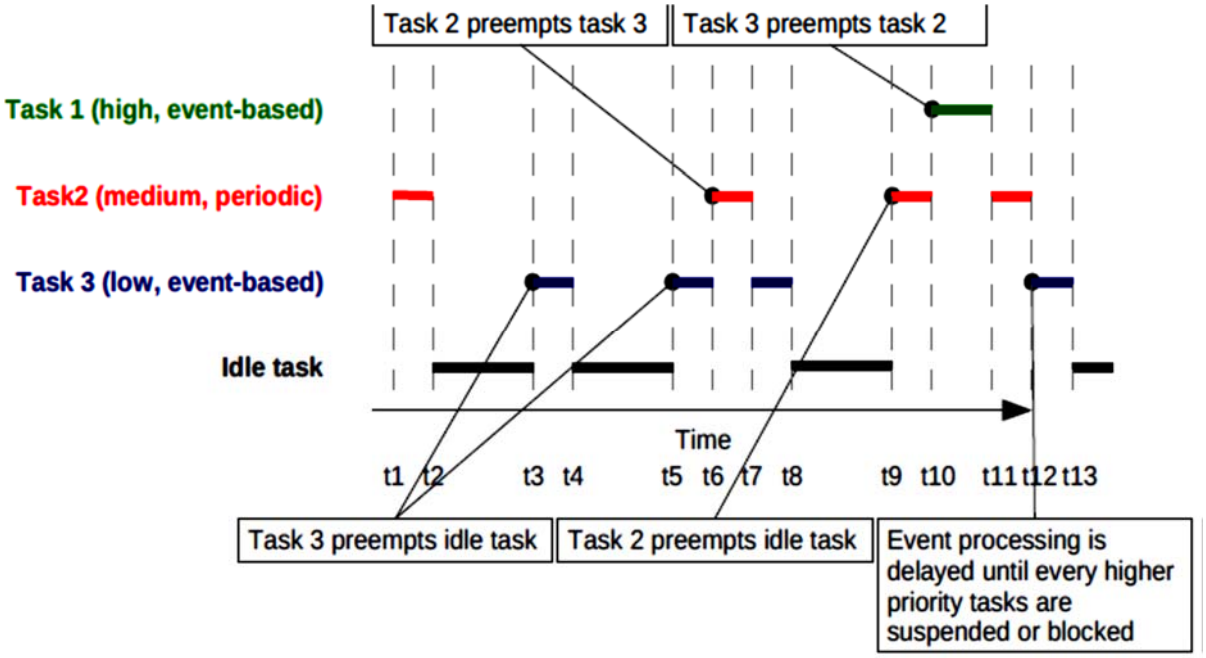
Le TCB est une structure de données contenant les informations nécessaires à la gestion des tâches. Une fois la tâche est créée, FreeRTOS lui assigne un TCB.

Ordonnancement

L'ordonnanceur de FreeRTOS est basé sur la priorité des tâches. En cas où plusieurs tâches partagent le même niveau de priorité, c'est le round-robin qui est utilisé.

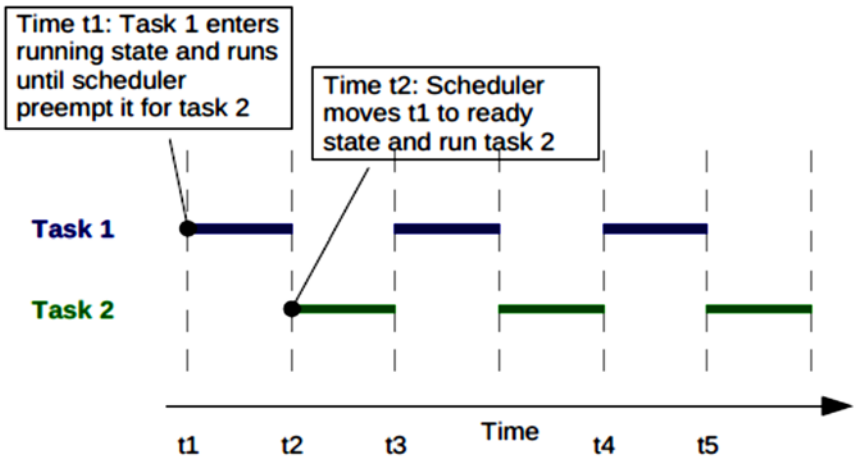
Priorité

La priorité est un nombre donné à une tâche lorsqu'elle est créée ou modifiée manuellement à l'aide de primitive vTaskPrioritySet (). La valeur priorité de 0 est la priorité minimale qu'une tâche peut avoir et ce niveau doit être strictement réservée à la tâche de fond. La priorité la plus élevée correspond à la plus grande valeur.
Le nombre maximum de priorités est défini dans la constante MAX_PRIORITIES dans FreeRTOSConfig.h. La plus grande valeur correspond à MAX_PRIORITIES - 1.



Round robin

Les tâches d'égale priorité sont traitées de manière égale par l'ordonnanceur ; il choisit à chaque quantum de temps une tâche selon l'algorithme round robin.



Gestion des tâches

Déclaration d'une tâche

Pour développer une application basée sur un OS, on décompose l'application en un ensemble de tâches. Dans FreeRTOS une tâche est fonction C contenant une boucle infinie et ne renvoie pas un résultat.

```
void ATaskFunction (void * pvParametres) ;
```

Une tâche est une fonction dont le traitement se trouve dans une boucle infinie.

```
void ATaskFunction( void *pvParameters )
{/* déclaration des variables locales */
    int iVariableExample = 0;
      for( ;; )
      {
            /* code de la fonction */
      }
}
```

Création d'une tâche

Une tâche est créée par l'intermédiaire de la fonction **xTaskCreate**().

```
BaseType_t xTaskCreate(     TaskFunction_t pvTaskCode,
                            const char * const pcName,
                            unsigned short usStackDepth,
                            void *pvParameters,
                            UBaseType_t uxPriority,
                            TaskHandle_t *pxCreatedTask
                           );
```

pvTaskCode	Les tâches sont juste des fonctions C qui ne sortiront jamais, et en tant que tels sont normalement mises en œuvre dans une boucle infinie. Le paramètre pvTaskCode est simplement un pointeur vers la fonction qui implémente la tâche (en fait seulement le nom de la fonction) qui met en œuvre la tâche.
pcName	Un nom descriptif pour la tâche. Il n'est utilisé par FreeRTOS en aucune façon. Il est inclus pour faciliter le débogage. l'identification d'une tâche par un nom lisible est beaucoup plus simple qu'un identificateur.
usStackDepth	Le noyau attribue à chaque tâche lors de sa création sa propre pile. La valeur usStackDepth indique au noyau la taille de la pile.
pvParameters	Les tâches acceptent un paramètre pointeur de type void. La valeur attribuée à pvParameters sera la valeur passée à la tâche. Il prend la valeur NULL s'il n'y a pas de paramètres.
uxPriority	Définit la priorité à laquelle la tâche sera exécutée. Les priorités peuvent être attribuées à partir de 0, qui est la priorité la plus basse, à (configMAX_PRIORITIES - 1), qui est la plus haute priorité. configMAX_PRIORITIES = 3 dans le cas de PIC18.
pxCreatedTask	pxCreatedTask peut être utilisé pour transmettre un indicateur d'indentification à la tâche en cours de création. Cet indicateur peut alors être utilisé pour faire référence à la tâche dans les appels d'API que, par exemple, changer la priorité de la tâche ou supprimer la tâche. Si votre application n'utilise pas l'indicateur de la tâche. pxCreatedTask peut être fixé NULL.

Cette fonction retourne :
pdTRUE : si la tâche est créée avec succès
errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY : s'il n'y a pas suffisamment d'espace mémoire.

Exemple de création de tâche :

```
xTaskCreate(      vBlink,
                  "Bliny",
                  100,
                  NULL,
                  1,
                  NULL );
void vBlink(void *pvParameters)
{
      while(1)
      {
            LED1=!LED1;
            vTaskDelay(300 / portTICK_RATE_MS );   //300ms
      }
}
```

Suppression d'une tâche

Une tâche créée peut être supprimé par la directive :

```
void vTaskDelete( xTaskHandle pxTask );
```

Exemple :

```
void vOtherFunction( void )
{
static unsigned char ucParameterToPass;
TaskHandle_t xHandle;
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass,
tskIDLE_PRIORITY, &xHandle );

// Utiliser xHandle pour supprimer la tâche.
vTaskDelete( xHandle );
}
```

Contrôle des tâches

vTaskDelay

Spécifie le nombre de top d'horloge (Ticks) de l'ordonnanceur pendant lequel la tâche reste bloquer, ce nombre est passé comme paramètre. Si on veut comptabiliser le temps de blocage en ms, il suffit de diviser le nombre de Ticks par la constance portTICK_RATE_MS.

Syntaxe : *void vTaskDelay(const TickType_t xTicksToDelay);*

Exemple :

```
void vTaskFunction( void * pvParameters )
{
/* bloquer pendant 500ms. */
 const TickType_t xDelay = 500 / portTICK_PERIOD_MS;
      for( ;; )
      {
            /* changer l'état de la LED chaque 500ms, revient à bloquer
               la tâche chaque changement */
            vToggleLED();
            vTaskDelay( xDelay );
      }
}
```

vTaskDelayUntil

La fonction **vTaskDelay** n'est pas précise. Si on veut bloquer une tâche à des intervalles de temps réguliers, il vaut mieux utiliser la **vTaskDelayUntil**. Cette fonction spécifie le temps absolu pendant lequel la tâche va se débloquer.

Syntaxe :

```
void vTaskDelayUntil(TickType_t *pxPreviousWakeTime, const
 xTimeIncrement );
```

pxPreviousWakeTime : valeur du compteur juste avant le blocage de la tâche

xTimeIncrement : le nombre de Ticks pendant lequel la tâche reste bloquer.

Exemple :

```
void vTaskFunction( void * pvParameters )
{
      TickType_t xLastWakeTime;
      const TickType_t xFrequency = 10;
      // Initialise  xLastWakeTime à la valeur courante du timer.
      xLastWakeTime = xTaskGetTickCount();
      for( ;; )
      {
      // Mise en attente pendant xFreqeuncy Ticks.
      vTaskDelayUntil( &xLastWakeTime, xFrequency );
      /* ou bien en ms vTaskDelayUntil( &xLastWakeTime, xFrequency
      / portTICK_PERIOD_MS ); */
      }
}
```

vTaskSuspend

La fonction **vTaskSuspend** : permet de suspendre une tâche. Cette tâche peut redémarrer en invoquant la fonction **vTaskResume**.

Syntaxe : *void vTaskSuspend(TaskHandle_t pxTaskToSuspend);*

Exemple :

```
void vAFunction( void )
{
  TaskHandle_t xHandle;
// Créer une tâche.
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

// Suspendre la tâche
vTaskSuspend( xHandle );
}
```

la tâche peut démarrer avec la fonction **vTaskResume(xHandle)**.

vTaskPrioritySet

Change la priorité d'une tâche

Syntaxe : *void vTaskPrioritySet(TaskHandle_t pxTask, UBaseType_t uxNewPriority);*

Exemple :

```
vTaskPrioritySet( xHandle, tskIDLE_PRIORITY + 1 );
```

xTaskGetTickCount

Renvoie le nombre de ticks du compteur de l'ordonnanceur.

Syntaxe : *volatile TickType_t xTaskGetTickCount(void);*

Contrôle du noyau

vTaskStartScheduler

void vTaskStartScheduler(void) : Démarre le processus du noyau temps réel.
Exemple :

```
void main( void )
{
// Créer les tâches puis démarrer le noyau
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL );
// démarrage du noyau.
 vTaskStartScheduler();
}
```

Le noyau peut être suspendu en invoquant la fonction « void vTaskSuspendAll(void) ». Cette fonction suspend toute activité du noyau sauf les interruptions. Le noyau peut reprendre son activité par la fonction « BaseType_t xTaskResumeAll(void) ».

Gestion de la queue

Les files d'attente sont la principale forme de communication Inter-tâches. Ils peuvent être utilisés pour envoyer des messages entre les tâches ou entre les interruptions et les tâches. Dans la plupart des cas, ils sont utilisés comme des files FIFO.

xQueueCreate

Crée une nouvelle instance de la file d'attente. Elle retourne en résultat un indicateur « handle » pour la file créée.
Syntaxe :

QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);

uxQueueLength : nombre maximum d'élément contenu dans la queue
uxItemSize : nombre d'octets pour chaque élément
Exemple :

```
struct AMessage
 {
    char ucMessageID;
    char ucData[ 20 ];
 };
unsigned long ulVar = 10UL;
 void vATask( void *pvParameters )
 {
      QueueHandle_t  xQueue1, xQueue2;
    // Création d'une queue contenant 10 éléments de type  unsigned long.
    xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );
    if( xQueue1 != NULL )
    {
        // La queue est créée et elle peut être maintenant utilisée.
    }
    // Créer une queue contenant 10 pointers sur des  structures AMessage.
    xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
    if( xQueue2 != NULL )
    {
        // ……..
    }
}
```

xQueueSend

Dépose un élément dans la queue

*BaseType_t xQueueSend(QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait);*

xQueue : l'indicateur retourné par la fonction xQueueCreate
pvItemToQueue : pointeur sur l'élément placé dans la file d'attente

xTicksToWait : nombre de Ticks maximal pendant lequel la tâche reste bloquer en attendant un espace disponible dans la file d'attente.
Exemple :

```
struct AMessage *pxMessage;
xQueueSend( xQueue1, ( void * ) &ulVar, ( TickType_t ) 10 ) ;
pxMessage = & xMessage;
xQueueSend( xQueue2, ( void * ) &pxMessage, (TickType_t) 0 );
```

xQueueReceive

prendre l'élément déjà déposé dans la file d'attente

BaseType_t ***xQueueReceive****(QueueHandle_t* ***xQueue****, void *pvBuffer,TickType_t* ***xTicksToWait*** *);*

xQueue : l'indicateur retourné par la fonction xQueueCreate
pvBuffer : pointeur sur le buffer dans lequel sera copié l'élément reçu.
xTicksToWait : nombre de Ticks maximal pendant lequel la tâche reste bloquer en attendant la présence d'un élément dans la file d'attente. Si xTicksToWait = 0, la fonction retourne immédiatement si la file est vide. La valeur portMAX_DELAY permet une attente infinie.
UBaseType_t uxQueueMessagesWaiting(QueueHandle_t xQueue) : retourne le nombre d'éléments stockés dans la file d'attente.
UBaseType_t uxQueueSpacesAvailable(QueueHandle_t xQueue) : retourne le nombre de place lire dans la file d'attente.
Exemple :

```
struct AMessage
{
  char ucMessageID;
  char ucData[ 20 ];
} xMessage;

QueueHandle_t xQueue;
//-------------------------------------
void vATask( void *pvParameters )
{
      struct AMessage *pxMessage;
      xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
      if( xQueue == NULL )
      {
            // la queue n'est pas créée.
      }

      pxMessage = & xMessage;
      xQueueSend( xQueue, ( void * ) &pxMessage, (TickType_t ) 0 );
}
// -------------------------------------
void vADifferentTask( void *pvParameters )
{
      struct AMessage *pxRxedMessage;
      if( xQueue != 0 )
      {

            if( xQueueReceive( xQueue, &( pxRxedMessage ), (TickType_t ) 10 ) )
            {
            // pcRxedMessage pointe sur la structure AMessage postée par vATask.
            }
      }
}
```

Gestion des ressourses

Sémaphore binaire

Les sémaphores binaires sont le moyen le plus simple de synchroniser les tâches

vSemaphoreCreateBinary

Crée un sémaphore binaire

Syntaxe : *SemaphoreHandle_t* ***xSemaphoreCreateBinary****(void);*

Exemple :

```
SemaphoreHandle_t xSemaphore;
void vATask( void * pvParameters )
{
      xSemaphore = xSemaphoreCreateBinary();
      if( xSemaphore != NULL )
      {
            // Le sémaphore est créé.
      }
}
```

xSemaphoreTake

Primitive de demande de ressource. Cette opération est équivalente à une opération P (). Si la ressource n'est pas disponible, la tâche sera bloqué jusqu'à la disponibilité de la ressource ou l'écoulement du délai.

Syntaxe *: xSemaphoreTake(SemaphoreHandle_t xSemaphore, xTicksToWait);*

xSemaphore : l'identificateur de la sémaphore (handle)

xTicksToWait : temps d'attente de la disponibilité de la ressource. La valeur portMAX_DELAY permet d'avoir une attente infinie.

Exemple :

```
xSemaphoreTake( xSemaphore, (TickType_t ) 0 ) ;
```

La tâche ne sera pas bloquée si la ressource n'est pas disponible.

```
xSemaphoreTake( xSemaphore, (TickType_t) portMAX_DELAY ) ;
```

La tâche reste bloquée jusqu'à la disponibilité de la ressource.

xSemaphoreGive

Primitive de libération de ressource. Cette opération est équivalente à une opération V ()

Syntaxe : *xSemaphoreGive(SemaphoreHandle_t xSemaphore);*

Exemple :

```
xSemaphoreGive( xSemaphore ) ;
```

Sémaphore à compte

Un sémaphore de comptage est un sémaphore qui peut être pris plusieurs fois avant qu'il ne devienne indisponible. Il maintient une valeur qui augmente au fur et à mesure que la ressource est libérée, et diminue quand elle est pris.

xSemaphoreCreateCounting

Crée un sémaphore à compte

Syntaxe *: SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount, UBaseType_t uxInitialCount)*

uxMaxCount : Valeur maximale à atteindre.

uxInitialCount : Valeur initiale de la sémaphore

Exemple :

```
void vATask( void * pvParameters )
{
	SemaphoreHandle_t xSemaphore;

	xSemaphore = xSemaphoreCreateCounting( 10, 0 );
	if( xSemaphore != NULL )
	{
		// le sémaphore est créé.
	}
}
```

Mutex

Les sémaphores binaires et les mutex sont très similaires ; les Mutex incluent un mécanisme d'héritage de prioritaire, les sémaphores binaires ne le font pas. Cela fait des sémaphores binaires le meilleur choix pour implémenter la synchronisation entre les tâches, et les mutex sont le meilleur choix pour implémenter une simple exclusion mutuelle.

xSemaphoreCreateMutex

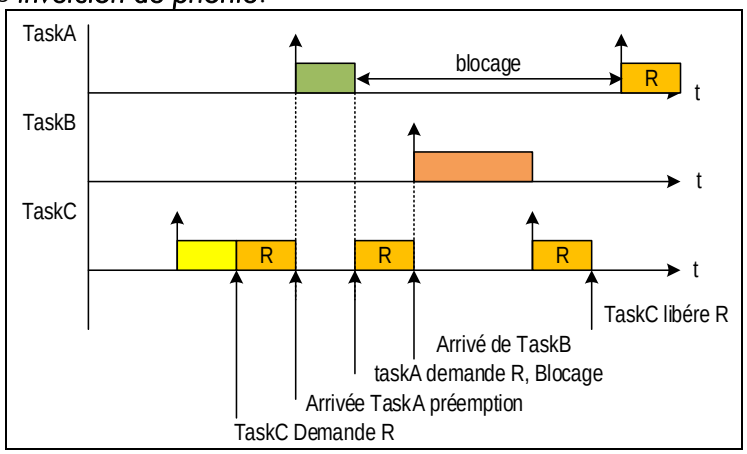
Crée un mutex
Syntaxe : *SemaphoreHandle_t xSemaphoreCreateMutex(void)*

Exemple :

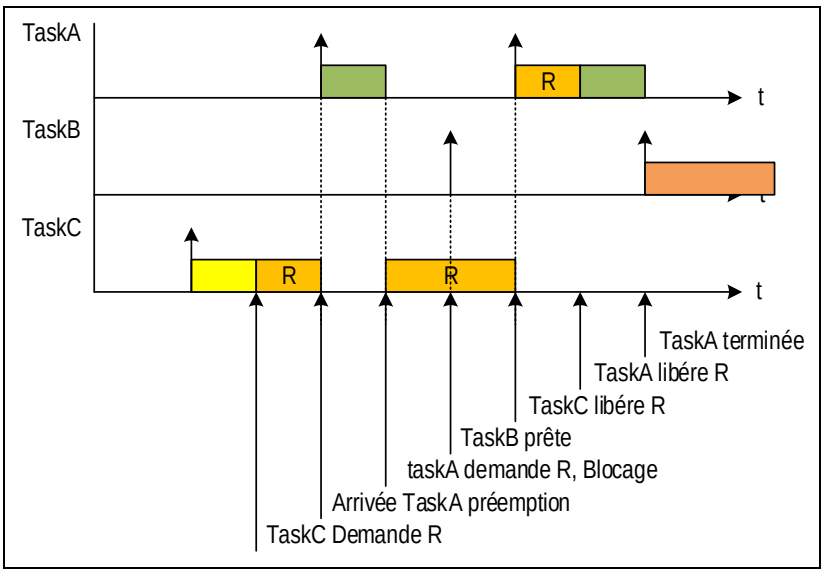
```
SemaphoreHandle_t xSemaphore;
void vATask( void * pvParameters )
{
	xSemaphore = xSemaphoreCreateMutex();
	if( xSemaphore != NULL )
	{
		// Le sémaphore est créé.
	}
}
```

Problème d'inversion de priorité

Soient trois tâches TaskA (haute priorité), TaskB et taskC (Basse priorité) et R une ressource partagée par les trois tâches. Supposons que TaskC est en possession de la ressource R ; TaskC est préemptée par TaskA plus prioritaire, lorsque TaskA souhaite prendre la ressource R elle sera bloquée à cause de la non disponibilité de R (déjà pris par TaskC), dans ce cas TaskB peut être exécutée si elle est prête. Ce phénomène est appelé *inversion de priorité*.



Pour pallier ce défaut, on attribue à la tâche en possession de ressource (TaskC), *la priorité de la tâche la plus prioritaire des tâches demandant la ressource* (héritage de priorité). Dans ce cas, lorsque TaskA est bloquée, c'est la TaskC qui reprend son exécution et non pas TaskB (TaskC à la même priorité que TaskA). TaskC revient à sa priorité ordinaire quand elle sort de la section critique (libère la ressource).


TaskA
TaskB
TaskC
R
t
R
t
t
TaskA terminée
TaskA libére R
TaskC libére R
TaskB prête
taskA demande R, Blocage
Arrivée TaskA préemption
TaskC Demande R