Bilel RAHMOUNI - Meriem NOUIRA

TP 2 : LES TÂCHES - 1

Matériel : Arduino UNO R4 WiFi, câble USB, environnement Arduino v2

Date : 16/10/2024

Introduction

L'objectif de ce TP est d'étudier le fonctionnement des tâches sous FreeRTOS, un système d'exploitation temps réel utilisé pour la gestion des tâches sur des systèmes embarqués. Nous allons créer et manipuler plusieurs tâches avec des priorités différentes, observer leur comportement et expérimenter avec les interruptions **Tick** pour la gestion du temps. Nous utiliserons l'Arduino UNO R4 WiFi, équipé de FreeRTOS, pour tester nos implémentations.

Configuration de FreeRTOS

Types de Données

FreeRTOS utilise deux types de données importants :

- **TickType_t** : Utilisé pour représenter le nombre d'interruptions **tick** depuis le démarrage du programme. Cela permet de mesurer la durée d'exécution des tâches.
- **BaseType_t** : Type de données le plus efficace pour l'architecture, généralement utilisé pour les fonctions qui retournent des valeurs limitées ou des booléens.

Ordonnancement Préemptif

Nous avons configuré FreeRTOS pour utiliser l'algorithme d'ordonnancement préemptif avec découpage temporel (**Prioritized Pre-emptive Scheduling with Time Slicing**). Pour cela, les paramètres suivants ont été modifiés dans le fichier **FreeRTOSConfig.h** :

```
#ifndef configUSE_PREEMPTION

#define configUSE_PREEMPTION (1)
```

```
#ifndef configUSE_TIME_SLICING

#define configUSE_TIME_SLICING (1)
```

Ces configurations garantissent que les tâches sont interrompues périodiquement pour permettre l'exécution d'autres tâches ayant la même priorité.

Expérimentations et Résultats

Exercice 2.1 : Création et Exécution de Deux Tâches Simples

Nous avons créé deux tâches simples qui affichent un message dans la console série avec un délai périodique. Le code est fourni ci-dessous :

```
#include <Arduino_FreeRTOS.h>  // Inclusion de la bibliothèque FreeRTOS pour Arduino



// Déclaration de deux tâches

void Task1( void *pvParameters );  // Prototype de la tâche 1

void Task2( void *pvParameters );  // Prototype de la tâche 2



// La fonction setup() est exécutée une fois lors du démarrage ou de la réinitialisation de la carte

void setup() {

    // Initialisation de la communication série à une vitesse de 9600 bits par seconde
```

```
    Serial.begin(9600);



    while (!Serial) {

        ; // Attendre que le port série soit connecté. Cela est nécessaire pour certaines cartes comme LEONARDO, MICRO, YUN et autres basées sur le microcontrôleur 32u4.

    }



    // Mise en place de deux tâches qui s'exécuteront indépendamment avec la même priorité

    xTaskCreate(

        Task1,    // Pointeur vers la fonction Task1

        "Task 1",  // Nom de la tâche (utilisé uniquement pour des raisons humaines, non requis par FreeRTOS)

        128,  // Taille de la pile allouée pour cette tâche (en mots, pas en octets)

        NULL, // Pas de paramètre passé à la tâche

        1,  // Priorité de la tâche (1 étant une priorité faible)

        NULL ); // Pas de handle pour la tâche (pointeur de gestion de la tâche)



    xTaskCreate(

        Task2,    // Pointeur vers la fonction Task2

        "Task 2",  // Nom de la tâche (utilisé uniquement pour des raisons humaines)

        128,  // Taille de la pile allouée pour cette tâche

        NULL, // Pas de paramètre passé à la tâche
```

```
        1,  // Priorité de la tâche (même priorité que Task1)
        NULL ); // Pas de handle pour la tâche

    vTaskStartScheduler(); // Démarrer l'ordonnanceur FreeRTOS qui gère l'exécution des tâches
}

void loop()
{
  // Cette boucle est vide, car toutes les opérations sont gérées par les tâches FreeRTOS
}

/*---------------------- Tâches ---------------------*/

void Task1( void *pvParameters )  // Fonction de la tâche 1
{
   // Affiche "Tache 1" dans le moniteur série
   volatile uint32_t cnt;  // Variable pour simuler un délai
   for (;;)  // Boucle infinie (la tâche ne doit jamais sortir ou retourner)
   {
      Serial.println("Tache 1");  // Affiche le message "Tache 1"
      for(cnt=0;cnt<=2400000;cnt++);  // Simule un délai en comptant jusqu'à 2,4 millions
   }
}
```

```
void Task2( void *pvParameters )  // Fonction de la tâche 2
{
   // Affiche "Tache 2" dans le moniteur série
   volatile uint32_t cnt;  // Variable pour simuler un délai
   for (;;)  // Boucle infinie (la tâche ne doit jamais sortir ou retourner)
   {
      Serial.println("Tache 2");  // Affiche le message "Tache 2"
      for(cnt=0;cnt<=1200000;cnt++);  // Simule un délai en comptant jusqu'à 1,2 million
   }
}
```

Sortie :

```
Tache 2
Tache 1
Tache 2
Tache 2
Tache 1
Tache 2
Tache 2
Tache 1
Tache 2
Tache 2
Tache 1
Tache 2
```

Les deux tâches semblent s'exécuter en parallèle, mais comme le processeur de l'Arduino n'a qu'un seul cœur, elles alternent rapidement leur exécution. Chaque tâche partage le temps d'exécution disponible sur le processeur, d'où l'apparente simultanéité.

Nous avons activé l'horodatage de la console série pour mesurer approximativement la durée d'exécution des boucles des tâches.

```
15:14:09.227 ->  Tache 1
15:14:09.273 ->  Tache 2
15:14:09.784 ->  Tache 2
15:14:10.249 ->  Tache 1
15:14:10.293 ->  Tache 2
15:14:10.770 ->  Tache 2
15:14:11.277 ->  Tache 1
15:14:11.322 ->  Tache 2
15:14:11.820 ->  Tache 2
15:14:12.288 ->  Tache 1
15:14:12.334 ->  Tache 2
15:14:12.848 ->  Tache 2
```

On peut voir que les deux tâches alternent leur exécution très rapidement, ce qui est un comportement normal avec FreeRTOS lorsque les tâches ont la même priorité.

Les différences de temps entre les affichages des messages "Tâche 1" et "Tâche 2" sont dues aux boucles dans le code, qui simulent des délais différents (2.4 millions pour `Tâche 1` et 1.2 million pour `Tâche 2`).

Entre `15:14:09.227` (Tâche 1) et `15:14:09.273` (Tache2), il y a environ 46 ms d'écart, ce qui montre la rapidité du commutateur d'ordonnancement.

Exercice 2.2 : Ajout d'une Troisième Tâche

Pour cet exercice, une troisième tâche a été ajoutée afin d'afficher mon prénom ("Bilel") toutes les 2 secondes.

```
void Task3( void *pvParameters );  // Prototype de la tâche 3
```

```
   xTaskCreate(
     Task3,
     "Task 3",
     128,
     NULL,
     1,
```

```
      NULL);
```

```
void Task3(void *pvParameters)

{

   volatile uint32_t cnt;

   for (;;)

   {

      Serial.println("Bilel");

      for(cnt = 0; cnt <= 4800000; cnt++);  // Simulation d'un délai de 2
secondes

   }

}
```

Sortie :

```
15:40:16.284 -> Tache 1
15:40:16.935 -> Bilel
15:40:16.981 -> Tache 2
15:40:17.707 -> Tache 2
15:40:17.786 -> Tache 1
15:40:18.475 -> Tache 2
15:40:19.246 -> Tache 2
15:40:19.339 -> Tache 1
15:40:19.936 -> Bilel
15:40:20.013 -> Tache 2
15:40:20.803 -> Tache 2
15:40:20.837 -> Tache 1
```

L'horodatage montre que la tâche **Task3** (qui affiche "Bilel") s'exécute correctement toutes les 2 secondes, comme attendu. Les autres tâches, **Task1** et **Task2**, s'exécutent également de manière alternée, partageant le temps d'exécution.

Exercice 2.3 : Création d’une Tâche à Partir d’une Autre

Dans cet exercice, nous avons testé la possibilité de créer une tâche à partir d'une autre. FreeRTOS permet de créer dynamiquement des tâches pendant l'exécution d'une autre tâche, ce qui peut être utile pour gérer des actions spécifiques sans surcharger le code initial.

```
#include <Arduino_FreeRTOS.h>


void Task1(void *pvParameters);

void Task2(void *pvParameters);


void setup() {

    Serial.begin(9600);

    while (!Serial);  // Attendre que le port série soit connecté


    // Création de Task1

    xTaskCreate(Task1, "Task 1", 128, NULL, 1, NULL);


    // Démarrer l'ordonnanceur FreeRTOS

    vTaskStartScheduler();

}


void loop() {

  // Vide car toutes les opérations sont gérées par les tâches FreeRTOS

}


void Task1(void *pvParameters) {
```

```
    // Création de Task2 depuis Task1
    xTaskCreate(Task2, "Task 2", 128, NULL, 1, NULL);

    for (;;) {
        Serial.println("Tache 1");
        vTaskDelay(pdMS_TO_TICKS(1000));  // Délai de 1 seconde
    }
}

void Task2(void *pvParameters) {
    for (;;) {
        Serial.println("Tache 2");
        vTaskDelay(pdMS_TO_TICKS(500));  // Délai de 0,5 seconde
    }
}
```

La Tache 1 est créée lors du démarrage du programme et affiche "Tache 1" toutes les secondes. Tache 1 crée ensuite Tache 2, qui s'exécute en parallèle et affiche "Tache 2" toutes les 500 millisecondes.
L'alternance entre les deux tâches se fait correctement, même avec l'ajout dynamique de la deuxième tâche comme on peut le voir ci-dessous.

```
15:47:12.357 -> Tache 2
15:47:12.866 -> Tache 2
15:47:12.998 -> Tache 1
15:47:13.364 -> Tache 2
15:47:13.876 -> Tache 2
15:47:14.008 -> Tache 1
15:47:14.404 -> Tache 2
15:47:14.870 -> Tache 2
15:47:15.044 -> Tache 1
15:47:15.417 -> Tache 2
15:47:15.902 -> Tache 2
15:47:16.027 -> Tache 1
```

Alternance correcte des tâches : Les deux tâches s'exécutent de manière alternée avec un délai de 500 ms pour **Tache 2** et environ 100 ms pour **Tache 1**, ce qui correspond aux paramètres de délai que nous avons configurés.

Exercice 2.4 : Résultats et Analyse

Dans cet exercice, nous avons ajouté une quatrième tâche (**Task4**) qui affiche "Rahmouni" toutes les 4 secondes. Cette tâche est créée par la **Task3** qui affiche "Bilel" toutes les 2 secondes.

```
#include <Arduino_FreeRTOS.h>



void Task1(void *pvParameters);

void Task2(void *pvParameters);

void Task3(void *pvParameters);

void Task4(void *pvParameters);



void setup() {

    // Initialisation de la communication série

    Serial.begin(9600);

    while (!Serial);  // Attendre que la connexion série soit prête



    // Création des tâches

    xTaskCreate(Task1, "Task 1", 128, NULL, 1, NULL);  // Tâche 1 avec
priorité 1

    xTaskCreate(Task2, "Task 2", 128, NULL, 1, NULL);  // Tâche 2 avec
priorité 1

    xTaskCreate(Task3, "Task 3", 128, NULL, 2, NULL);  // Tâche 3 avec
priorité 2
```

```
    // Démarrer l'ordonnanceur FreeRTOS
    vTaskStartScheduler();
}

void loop() {
    // Vide car tout est géré par les tâches sous FreeRTOS
}

// Tâche 1 : Affiche "Tache 1" toutes les secondes
void Task1(void *pvParameters) {
    for (;;) {
        Serial.println("Tache 1");
        vTaskDelay(pdMS_TO_TICKS(1000));  // Délai de 1 seconde
    }
}

// Tâche 2 : Affiche "Tache 2" toutes les 500 ms
void Task2(void *pvParameters) {
    for (;;) {
        Serial.println("Tache 2");
        vTaskDelay(pdMS_TO_TICKS(500));  // Délai de 0,5 seconde
    }
}
```

```
// Tâche 3 : Affiche "Bilel" toutes les 2 secondes et crée Task4
void Task3(void *pvParameters) {
    // Créer Task4 à l'intérieur de Task3
    xTaskCreate(Task4, "Task 4", 128, NULL, 1, NULL);  // Task4 avec priorité 1

    for (;;) {
        Serial.println("Bilel");
        vTaskDelay(pdMS_TO_TICKS(2000));  // Délai de 2 secondes
    }
}

// Tâche 4 : Affiche "Rahmouni" toutes les 4 secondes
void Task4(void *pvParameters) {
    for (;;) {
        Serial.println("Rahmouni");
        vTaskDelay(pdMS_TO_TICKS(4000));  // Délai de 4 secondes
    }
}
```

Sortie :

```
16:16:52.908 -> Bilel
16:16:52.908 -> Rahmouni
16:16:53.001 -> Tache 1
16:16:53.172 -> Tache 2
16:16:53.682 -> Tache 2
16:16:53.996 -> Tache 1
16:16:54.165 -> Tache 2
16:16:54.709 -> Tache 2
16:16:54.893 -> Bilel
16:16:55.032 -> Tache 1
16:16:55.220 -> Tache 2
16:16:55.730 -> Tache 2
```

Task3 s'exécute toutes les 2 secondes, affichant "Bilel" sur le moniteur série.
Task4, qui est créée dynamiquement par **Task3**, s'exécute toutes les 4 secondes, affichant "Rahmouni".
Task1 et **Task2** continuent de s'exécuter de manière indépendante, affichant leurs messages respectifs avec des délais de 1 seconde et 500 ms.
Les quatre tâches partagent bien le temps processeur grâce à l'utilisation de `vTaskDelay()` qui permet à l'ordonnanceur FreeRTOS de gérer efficacement l'exécution parallèle.

Exercice 2.5 : Utilisation de la Communication Série et de la LED

Cet exercice consiste à gérer plusieurs tâches FreeRTOS qui affichent des messages différents à des intervalles prédéfinis tout en permettant de contrôler la LED via des commandes série. Nous avons deux nouvelles tâches : **TaskMeriem** qui affiche "Meriem" toutes les 2 secondes et **TaskBilel** qui affiche "Bilel" toutes les 4 secondes.

```
#include <Arduino_FreeRTOS.h>

// Déclaration des cinq tâches
void Task1( void *pvParameters );
void Task2( void *pvParameters );
void TaskMeriem( void *pvParameters );  // Tâche pour afficher "Meriem"
void TaskBilel( void *pvParameters );   // Tâche pour afficher "Bilel"
void TaskLEDControl( void *pvParameters ); // Tâche pour le pilotage de la LED

void setup() {
  // Initialisation de la communication série
  Serial.begin(9600);
  pinMode(LED_BUILTIN, OUTPUT); // Initialisation de la LED (D13)

  while (!Serial);  // Attente de la connexion série

  // Création des cinq tâches indépendantes
  xTaskCreate(Task1, "Task 1", 128, NULL, 1, NULL);
  xTaskCreate(Task2, "Task 2", 128, NULL, 1, NULL);
  xTaskCreate(TaskMeriem, "Task Meriem", 128, NULL, 1, NULL);
  xTaskCreate(TaskBilel, "Task Bilel", 128, NULL, 1, NULL);
  xTaskCreate(TaskLEDControl, "LED Control", 128, NULL, 1, NULL);

  // Démarrer l'ordonnanceur FreeRTOS
  vTaskStartScheduler();
```

```
}

void loop() {
  // Vide, car toutes les opérations sont gérées par FreeRTOS
}

// Tâche 1 : Affiche "Tache 1" toutes les 1 seconde
void Task1( void *pvParameters ) {
  for (;;) {
    Serial.println("Tache 1");
    delay(1000);
  }
}

// Tâche 2 : Affiche "Tache 2" toutes les 500 ms
void Task2( void *pvParameters ) {
  for (;;) {
    Serial.println("Tache 2");
    delay(500);
  }
}

// Tâche Meriem : Affiche "Meriem" toutes les 2 secondes
void TaskMeriem( void *pvParameters ) {
  for (;;) {
    Serial.println("Meriem");
    delay(2000);  // Délai de 2 secondes
  }
}

// Tâche Bilel : Affiche "Bilel" toutes les 4 secondes
void TaskBilel( void *pvParameters ) {
  for (;;) {
    Serial.println("Bilel");
    delay(4000);  // Délai de 4 secondes
  }
}

// Tâche pour piloter la LED via des commandes série
void TaskLEDControl( void *pvParameters ) {
  char receivedChar;
  for (;;) {
```

```
    if (Serial.available() > 0) {
      receivedChar = Serial.read();  // Lire les commandes du terminal

      if (receivedChar == 'M') {
        digitalWrite(LED_BUILTIN, HIGH); // Allumer la LED
        Serial.println("LED ON");
      } else if (receivedChar == 'A') {
        digitalWrite(LED_BUILTIN, LOW);  // Éteindre la LED
        Serial.println("LED OFF");
      }
    }
    delay(10); // Petite pause pour éviter l'utilisation excessive du CPU
  }
}
```

TaskMeriem : Affiche "Meriem" toutes les 2 secondes.
TaskBilel : Affiche "Bilel" toutes les 4 secondes.
Task1 et **Task2** continuent d'afficher "Tache 1" toutes les 1 seconde et "Tache 2" toutes les 500 ms, respectivement.
TaskLEDControl permet de contrôler l'état de la LED avec les commandes série : 'M' allume la LED, et 'A' l'éteint.

Exercice 2.6 : Utilisation de la Fonction `vTaskDelayUntil`

Dans cet exercice, nous introduisons l'utilisation de la fonction `vTaskDelayUntil()` qui permet à une tâche de s'exécuter à des intervalles périodiques précis, en conservant une synchronisation temporelle exacte. Contrairement à `vTaskDelay()`, cette fonction ne subit pas les variations dues à la durée d'exécution de la tâche elle-même, garantissant ainsi une exécution plus précise.

```
#include <Arduino_FreeRTOS.h>

// Déclarations des messages pour chaque tâche
const char *pcTextForTask1 = "Tache 1";
const char *pcTextForTask2 = "Tache 2";

// Définition des tâches
void Task1(void *pvParameters);
void Task2(void *pvParameters);

// La fonction setup s'exécute une seule fois au démarrage
```

```
void setup() {
  // Initialisation de la communication série à 9600 bauds
  Serial.begin(9600);

  // Attendre que le port série soit prêt
  while (!Serial);

  // Créer la tâche 1 avec une priorité de 1
  xTaskCreate(
    Task1,                  // Fonction associée à la tâche
    "Task 1",               // Nom de la tâche (pour le débogage)
    128,                    // Taille de la pile
    (void*) pcTextForTask1,  // Paramètres de la tâche (message à afficher)
    1,                      // Priorité de la tâche
    NULL                    // Handle de la tâche (non utilisé ici)
  );

  // Créer la tâche 2 avec la même priorité
  xTaskCreate(
    Task2,                  // Fonction associée à la tâche
    "Task 2",               // Nom de la tâche (pour le débogage)
    128,                    // Taille de la pile
    (void*) pcTextForTask2,  // Paramètres de la tâche (message à afficher)
    1,                      // Priorité de la tâche
    NULL                    // Handle de la tâche (non utilisé ici)
  );

  // Démarrer l'ordonnanceur de FreeRTOS
  vTaskStartScheduler();
}

// La fonction loop est vide car tout est géré dans les tâches
void loop() {
  // Rien à faire ici
}

// Tâche 1
void Task1(void *pvParameters) {
  char *pcTaskName;
  pcTaskName = (char *) pvParameters;
  volatile uint32_t cnt;
```

```
  for (;;) { // La tâche ne retourne jamais
    // Affiche le message associé à la tâche 1
    Serial.println(pcTaskName);
    vTaskDelay(pdMS_TO_TICKS(1000)); // Attendre 1000 ms (1 seconde)

    // Petite pause en faisant une boucle vide
    for (cnt = 0; cnt <= 320000; cnt++);
  }
}

// Tâche 2
void Task2(void *pvParameters) {
  char *pcTaskName;
  pcTaskName = (char *) pvParameters;
  volatile uint32_t cnt;

  for (;;) { // La tâche ne retourne jamais
    // Affiche le message associé à la tâche 2
    Serial.println(pcTaskName);
    vTaskDelay(pdMS_TO_TICKS(500)); // Attendre 500 ms (0.5 seconde)

    // Petite pause en faisant une boucle vide
    for (cnt = 0; cnt <= 320000; cnt++);
  }
}
```

Ce programme utilise le système d'exploitation temps réel FreeRTOS sur une carte Arduino pour exécuter deux tâches en parallèle. Chaque tâche est responsable d'afficher un message spécifique sur le moniteur série à des intervalles réguliers. **Task1** affiche "Tache 1 en execution" toutes les secondes, tandis que **Task2** affiche "Tache 2 en execution" toutes les 500 millisecondes. Le programme tire parti de la fonction `xTaskCreate()` pour créer les deux tâches et passer des paramètres, permettant ainsi à une seule fonction de gérer plusieurs tâches avec des comportements différents. L'utilisation de `vTaskDelay()` permet de mettre en pause chaque tâche pendant une durée définie sans bloquer le fonctionnement du microcontrôleur, assurant une exécution fluide et continue des deux tâches en parallèle.

Exercice 2.7 : Utilisation de la Fonction de Tâche Générique

Dans cet exercice, vous avez choisi d'utiliser une fonction de tâche générique (**TaskFunction**) qui permet d'exécuter deux tâches distinctes avec le même code, mais avec des paramètres différents. L'objectif est d'optimiser le code en utilisant la même fonction pour plusieurs tâches en passant des paramètres personnalisés à chaque tâche, afin qu'elles affichent des messages différents.

```
#include <Arduino_FreeRTOS.h>

// Déclarations des messages pour chaque tâche
const char *pcTextForTask1 = "Tache 1 en execution";
const char *pcTextForTask2 = "Tache 2 en execution";

// Définition des tâches
void TaskFunction(void *pvParameters);

// La fonction setup s'exécute une seule fois au démarrage
void setup() {
  // Initialisation de la communication série à 9600 bauds
  Serial.begin(9600);
  while (!Serial);  // Attendre que le port série soit prêt

  // Créer la première tâche avec une priorité de 1
  xTaskCreate(
    TaskFunction,           // Fonction associée à la tâche
    "Task 1",               // Nom de la tâche (pour le débogage)
    128,                    // Taille de la pile
    (void*) pcTextForTask1,   // Paramètres de la tâche (message à afficher)
    1,                      // Priorité de la tâche
    NULL                    // Handle de la tâche (non utilisé ici)
  );

  // Créer la deuxième tâche avec la même priorité
  xTaskCreate(
    TaskFunction,           // Fonction associée à la tâche
    "Task 2",               // Nom de la tâche (pour le débogage)
    128,                    // Taille de la pile
    (void*) pcTextForTask2,   // Paramètres de la tâche (message à afficher)
    1,                      // Priorité de la tâche
    NULL                    // Handle de la tâche (non utilisé ici)
```

```
  );

  // Démarrer l'ordonnanceur FreeRTOS
  vTaskStartScheduler();
}

// La fonction loop est vide car tout est géré dans les tâches
void loop() {
  // Rien à faire ici
}

// Fonction associée aux deux tâches
void TaskFunction(void *pvParameters) {
  char *pcTaskName;
  pcTaskName = (char *) pvParameters;
  volatile uint32_t cnt;

  for (;;) { // La tâche ne retourne jamais
    // Affiche le message associé à la tâche
    Serial.println(pcTaskName);
    vTaskDelay(pdMS_TO_TICKS(1000));  // Attendre 1000 ms (1 seconde)

    // Petite pause en faisant une boucle vide
    for (cnt = 0; cnt <= 320000; cnt++);
  }
}
```

Sortie :

```
16:45:11.004 -> Tache 1 en execution
16:45:11.113 -> Tache 2 en execution
16:45:12.094 -> Tache 1 en execution
16:45:12.183 -> Tache 2 en execution
16:45:13.211 -> Tache 1 en execution
16:45:13.305 -> Tache 2 en execution
16:45:14.315 -> Tache 1 en execution
16:45:14.410 -> Tache 2 en execution
16:45:15.404 -> Tache 1 en execution
16:45:15.482 -> Tache 2 en execution
16:45:16.486 -> Tache 1 en execution
16:45:16.580 -> Tache 2 en execution
```

Le moniteur série affiche "Tache 1 en execution" suivi de "Tache 2 en execution" de manière alternée, à un intervalle d'environ 1 seconde pour chaque tâche, comme prévu.
Task1 et **Task2** partagent la même fonction mais reçoivent des paramètres différents, ce qui permet de réutiliser le même code pour afficher des messages distincts.

vTaskDelay(pdMS_TO_TICKS(1000)) assure que chaque tâche attend 1 seconde avant de s'exécuter à nouveau.

Le code fonctionne correctement et affiche les messages des deux tâches à intervalles réguliers. Il démontre comment utiliser efficacement les paramètres pour réutiliser une seule fonction de tâche dans FreeRTOS, tout en réduisant la duplication de code.

Exercice 2.8 : Changement de priorité d'une tâche

Ce programme utilise FreeRTOS pour créer deux tâches avec des priorités différentes. **Task1** a une priorité de 3, tandis que **Task2** a une priorité de 2. Les deux tâches s'exécutent en boucle infinie et affichent un message sur le moniteur série.

```
#include <Arduino_FreeRTOS.h>


// Définition des deux tâches
void Task1(void *pvParameters);
void Task2(void *pvParameters);


// La fonction setup s'exécute une seule fois au démarrage
void setup() {
  // Initialisation de la communication série à 9600 bauds
  Serial.begin(9600);


  // Créer la première tâche avec une priorité de 3
  xTaskCreate(
    Task1,          // Fonction associée à la tâche
    "Task 1",       // Nom de la tâche (pour le débogage)
    128,            // Taille de la pile
```

```
    NULL,           // Paramètres pour la tâche (NULL ici)
    3,              // Priorité de la tâche (3)
    NULL            // Handle de la tâche (non utilisé ici)
  );

  // Créer la deuxième tâche avec une priorité de 2
  xTaskCreate(
    Task2,          // Fonction associée à la tâche
    "Task 2",       // Nom de la tâche (pour le débogage)
    128,            // Taille de la pile
    NULL,           // Paramètres pour la tâche (NULL ici)
    2,              // Priorité de la tâche (2)
    NULL            // Handle de la tâche (non utilisé ici)
  );

  // Démarrer l'ordonnanceur FreeRTOS
  vTaskStartScheduler();
}

// La fonction loop est vide car tout est géré dans les tâches
void loop() {
  // Rien à faire ici
}
```

```
// Tâche 1
void Task1(void *pvParameters) {
  volatile uint32_t cnt;

  for (;;) { // La tâche ne retourne jamais
    // Affiche un message sur le moniteur série
    Serial.println("Tache 1 en execution");

    // Petite pause avec une boucle vide
    for (cnt = 0; cnt <= 240000; cnt++);
  }
}

// Tâche 2
void Task2(void *pvParameters) {
  volatile uint32_t cnt;

  for (;;) { // La tâche ne retourne jamais
    // Affiche un message sur le moniteur série
    Serial.println("Tache 2 en execution");

    // Petite pause avec une boucle vide
    for (cnt = 0; cnt <= 120000; cnt++);
  }
```

```
}
```

Sortie :

```
16:47:06.553 -> Tache 1 en execution
16:47:06.679 -> Tache 2 en execution
16:47:07.688 -> Tache 1 en execution
16:47:07.733 -> Tache 2 en execution
16:47:08.733 -> Tache 1 en execution
16:47:08.868 -> Tache 2 en execution
16:47:09.824 -> Tache 1 en execution
16:47:09.949 -> Tache 2 en execution
16:47:10.940 -> Tache 1 en execution
16:47:11.031 -> Tache 2 en execution
16:47:12.036 -> Tache 1 en execution
16:47:12.127 -> Tache 2 en execution
```

Task1 et **Task2** affichent respectivement "Tache 1 en execution" et "Tache 2 en execution" sur le moniteur série.

Les deux tâches sont créées avec des priorités différentes :

- **Task1** (priorité 3) a une priorité plus élevée, ce qui signifie qu'elle est favorisée par l'ordonnanceur.
- **Task2** (priorité 2) s'exécute lorsque **Task1** ne monopolise pas le CPU.

Les boucles vides après chaque affichage introduisent une pause qui simule un délai.

On observe que **Task1** s'exécute plus fréquemment que **Task2**, car elle a une priorité plus élevée. **Task2** s'exécute moins souvent, ne prenant le contrôle que lorsque **Task1** laisse du temps à l'ordonnanceur.