République **A**lgérienne **D**émocratique et **P**opulaire
Ministère de l'**E**nseignement **S**upérieur et de la **R**echerche **S**cientifique

UNIVERSITE **M**OULOUD **M**AMMERI DE **T**IZI-**O**UZOU

FACULTE DE **G**ENIE **E**LECTRIQUE ET D' **I**NFORMATIQUE
DEPARTEMENT D'AUTOMATIQUE

Mémoire de Fin d'Etudes de MASTER ACADEMIQUE

Domaine : **Sciences et Technologies**
Filière : **Automatique**
Spécialité : **Automatique et Informatique industrielle**

Présenté par
Nacéra ICHEDDADENE

Thème

Conception d'un algorithme d'ordonnancement par rétroaction sous FreeRTOS.

Mémoire soutenu publiquement le 11/01/ 2021 devant le jury composé de :

M. KARA Redouane
Professeur, UMMTO, Président

M. KERMIA Omar
MRA, CDTA, Encadreur

M. TOUAT Mohand Achour
MCB, UMMTO, Co-Encadreur

M. DAOUI Mhemmed
Professeur, UMMTO, Examinateur

Remerciements

Avant tout je tiens à remercier Dieu le tout puissant de m'avoir donné le courage, la volonté ainsi que la force afin d'achever ce projet.

Je voudrais ensuite adresser toute ma reconnaissance à **M.TOUAT** et **M.KERMIA** de m'avoir dirigé et pour la patience, disponibilité et orientations qu'ils m'ont prodigué tout au long de la réalisation de ce travail.

J'aimerai remercier **M.DAOUI** pour son aide et judicieux conseils.

Enfin, J'adresse mes sincères remerciements à ma famille et toute personne qui m'a encouragé et cru en moi.

Dédicaces

Je dédie ce modeste travail :

A la femme qui doit ma vie celle qui a souffert sans me laisser souffrir, qui a sacrifié sa vie pour que j'aie une et qui n'a jamais dit non à mes exigences : Mon adorable maman **OUNISSA**, quoi que je fasse ou je dise je ne saurais point te remercier comme il se doit,

JE T'AIME ma chère.

A l'homme qui a été toujours à mes côtés pour m'encourager et me soutenir, celui qui m'a doté d'une éducation digne : Mon cher papa **LOUNES** que se travail traduit ma gratitude et mon affection.

A ma sœur **DALILA** et mes frères **NORDINE** et **FATEH**, qui m'ont toujours soutenu et crus en moi, que dieu leurs donne santé et longue vie et nous garde toujours unis.

A la mémoire de mon petit frère **MAHMOUDE** et mes grands-parents.

Table des matières

Liste des figures

Liste des abréviations

Présentation du centre de recherche (CDTA)……1

Introduction générale…….3

CHAPITRE I : Généralités

1 Introduction 5

2 Présentation de FreeRTOS et la notion d'ordonnancement 5

2.1 Présentation de FreeRTOS 5

2.2 Présentation de l'ordonnancement 10

2.2.1 Ordonnancement des processus 10

2.2.2 L'ordonnanceur 11

2.2.3 Les classes d'algorithmes d'ordonnancement 12

2.2.4 Modélisation des tâches temps réel 14

2.2.4.1 Modélisation des tâches périodiques 14

2.2.4.2 Modélisation des tâches non périodiques (apériodiques et sporadiques) 15

2.2.5 Les algorithmes d'ordonnancement des tâches indépendantes 15

2.2.5.1 Algorithmes à priorités statiques 15

2.2.5.2 Algorithmes à priorité dynamique 16

2.2.6 Les limites des algorithmes d'ordonnancement classiques 15

2.2.6.1 Connaissance des contraintes temporelles 16

2.2.6.2 Procédure de répartition des priorités 16

2.2.6.3 Environnement dynamique 16

3 Conclusion 16

Chapitre II*:* L'ordonnancement par rétroaction

1 Introduction 21

2 Présentation de l'ordonnancement par commande rétroactive (feedback scheduling) 21

2.1 Loi de commande 21

2.2 Le feedback scheduling 22

2.3 Les travaux de feedback scheduling 22
2.3.1 Cervin 22
2.3.2 Eker et all 23
2.3.3 Cervin et all 24
2.3.4 Control server Model de Cervin 24
2.3.5 Le feedback scheduling de Xia et all [18][19][20] 24
3 Algorithme Feedback Control Earliest Deadline First (FC-EDF) 25
3.1 Le contrôleur PID 25
3.2 Feedback control EDF (FC-EDF) 26
3.3 Rôle et fonctionnement des différents composants de FC-EDF 28
4 Conclusion 32

CHAPITRE III: Feedback scheduling sous FreeRTOS

1 Introduction 33
2 Solution proposée 33
2.1 Architecture proposée 33
2.1.1 Présentation du TCB étendu (eTCB) 34
2.1.2 Implémentation de l'eTCB 34
2.2 La tâche FST (Feedback Control Task) 36
2.2.1 Les modules FST 38
2.2.1.1 Module AppSched 38
2.2.1.2 Module AppSchedEDF 39
2.2.1.3 Module AppSchedPID 39
2.2.1.4 Module AppSchedSLC 40
2.2.1.5 Module AppSchedAC 40
3 FC-EDF sous FreeRTOS 40
3.1 Implémentation du module AppSched 41
3.1.1 Implémentation de l'API 41
3.1.2 Implémentation des fonctions privées 41
3.2 Implémentation du module AppSchedEDF 42
3.3 Implémentation du module AppSchedPID 43
3.4 Implémentation du module AppSchedSLC 44

3.5 Implémentation du module AppSchedAC 44
4 Exemple d'application 45
5 Conclusion 50
Conclusion générale 51
Références bibliographiques

Liste des figures

Figure I 1 Schéma qui représente les différents états d'une tâche. 5
Figure I 2 exemple de création d'une tâche sous FreeRTOS. 6
Figure I 3 Caractéristiques temporelles d'une tâche 7
Figure I 4 exemple d'ordonnancement préemptif. 10
Figure I 5 Exemple avec EDF. 14

Figure II 1 Boucle d'asservissement d'un système commandé par rétroaction 18
Figure II 2 Architecture feedback scheduler d'Eker 20
Figure II 3 Architecture du feedback scheduling de XIA. 22
Figure II 4 L'architecture de l'ordonnanceur FC-EDF. 25

Figure III 1 Interaction entre FST et les tâches acceptées et le noyau FreeRTOS. 34

Liste des abréviations

CDTA : Centre de **D**éveloppement des **T**echnologies **A**vancées.

RTOS: Real **T**ime **O**perating **S**ystem.

TCB: Task **C**ontrol **B**loc.

eTCB: extended TCB.

WCTE: Worst **C**ase **E**xecution **T**ime.

RM: Rate **M**onotonic.

DM: Deadline **M**onotonic.

EDF: Earliest **D**eadline **F**irst.

CPU: Control **P**rocess **U**nit.

FC-EDF: Feedback **C**ontrol EDF.

PID: Proportional **I**ntegral **D**erive.

SLC: Service **L**evel **C**ontroller.

AC: Admission **C**ontroller.

FST: Feedback Scheduling **T**ask.

Introduction générale

Introduction générale

Les procédés avancés répondus aujourd'hui dans les différentes industries, à l'exemple de la télécommunication, la robotique et le transport , nécessitent bien évidement des systèmes de contrôle\commande de plus en plus avancés et ayant de plus en plus d'exigences et de contraintes, d'où l'introduction de systèmes à temps réel qui sont caractérisés par non seulement l'exactitude des résultats fournis mais aussi par le respect des contraintes temporelles [7].

Pour les microprocesseurs et\ou microcontrôleurs en général, on parle de systèmes d'exploitation qui sont un ensemble de programmes qui permettent de gérer les ressources matérielles, afin d'y accéder et optimiser leur utilisation.

Parmi les systèmes d'exploitation temps réel leader du marché on trouve FreeRTOS qui est conçu spécialement pour les petits systèmes embarqués, il est basé sur un algorithme d'ordonnancement statique (SPP) où les tâches ont une priorité fixe, d'autre part plusieurs algorithmes plus fiables et plus évolués ont été développés ces dernières années, à savoir RM (Rate Monotonic), DM (Deadline Monotonic), EDF (Earliest Deadline First) et LLF (Least Laxity First) [3].

Dans ce travail, nous réalisons l'ordonnancement par rétroaction (Feedback scheduling), qui est une technologie capable de gérer l'incertitude de la disponibilité des ressources dans les systèmes de contrôle en temps réel sous FreeRTOS. Pour ce faire, nous avons opté pour l'algorithme FC-EDF (Feedback Control EDF) qui permet d'utiliser les techniques de commande par rétroaction sur l'ordonnancement en intégrant le régulateur PID avec l'ordonnanceur EDF.

Nous voudrions donc développer une nouvelle version FreeRTOS, basée sur un algorithme plus fiable afin de montrer les avantages du feedback scheduling ainsi que de l'algorithme choisi.

Ce travail est subdivisé en trois chapitres, dans un premier temps nous verrons une présentation de notre système d'exploitation temps réel FreeRTOS, mais encore des généralités sur l'ordonnancement et quelques algorithmes classiques qui le réalisent. Ensuite, nous présentons les inconvénients majeurs de ces algorithmes.

Dans le deuxième chapitre, nous allons définir l'ordonnancement par rétroaction et analyser la stratégie d'ordonnancement FC-EDF.

Finalement, on finit par une description de l'implémentation de l'algorithme FC-EDF sous FreeRTOS en plus d'un exemple d'application qui met en évidence l'intérêt du travail.

Le mémoire sera clôturé par une conclusion générale.

Chapitre I:

Généralités

1 Introduction

Un système temps réel (RT), est un système qui lit les données entrantes assez vite (réagit assez vite à ces données), où les actions se réalisent à temps déterminé quel que soit les contraintes de l'environnement, ces systèmes peuvent être classifier en deux classes : les systèmes temps réel souples, ou le « soft real time » qui sont des procédés avec un degré de flexibilité par rapport au respect des contraintes temporelles, et les systèmes temps réel critiques, ou le « hard real time » sont des procédés avec une tolérance nulle au non-respect des contraintes temporelles, c'est-à-dire ils exigent un stricte management de temps et de ressources [2].

Dans ce chapitre, nous introduisons les systèmes d'exploitation temps réel, où on va s'intéresser au FreeRTOS en le définissant ainsi que ses caractéristiques. Nous allons ensuite donner un état de l'art sur l'ordonnancement temps réel, et les différentes stratégies d'ordonnancement en se focalisant sur l'algorithme d'ordonnancement EDF.

2 Présentation de FreeRTOS et la notion d'ordonnancement

2.1 Présentation de FreeRTOS

Un système d'exploitation à temps réel (RTOS : Real Time Operating System) est un système d'exploitation possédant des fonctionnalités spécifiques pour gérer les contraintes temporelles exigées.

Les RTOS possèdent beaucoup d'avantages, dont la prédictibilité, gestion des interruptions, ils autorisent l'ordonnancement multitâche , l'ordonnancement avec priorité et permettent la communication inter tâches [3].

- **Définition :** Le FreeRTOS est un système d'exploitation à temps réel gratuit, conçu pour être suffisamment petit pour fonctionner sur un microcontrôleur, il a été mis à disposition par *Richard Barry* en 2003, il est destiné principalement pour les systèmes embarqués qui ont des contraintes

d'espace pour le code, ainsi que pour des systèmes de traitement vidéo et des applications réseau qui ont des contraintes temps réel [3].

Le FreeRTOS est aujourd'hui parmi les plus utilisés dans le marché des systèmes d'exploitation temps réel, et cela grâce aux caractéristiques dont il dispose qui se résument dans ce qui suit :

- ✓ Disponible en open source (sous licence MIT) et porté sur 33 architectures différentes;
- ✓ Disponible dans un seul fichier zip qui contient tous les ports officiels de FreeRTOS et les fichiers source en C;
- ✓ Faible impact car il consomme peu de ROM, RAM et ressources processeur ;
- ✓ Permet la communication inter-tâches par la création de files d'attente ;
- ✓ Capable de gérer un nombre infini de tâches simultanément et leur priorité ce nombre n'est limité que par le matériel;
- ✓ La documentation de FreeRTOS est accessible en ligne via le site internet (www.FreeRTOS.org) [3].

- **Politique d'ordonnancement de FreeRTOS :**

L'algorithme d'ordonnancement de FreeRTOS est basé sur la priorité statique SPP (Static Priority Policy), lors de la création d'une tâche l'utilisateur attribue une priorité à chaque tâche, la tâche prête avec la plus haute priorité c'est elle qui va s'exécuter en premier (elle obtient le temps CPU à l'état prêt).

Dans le cas où plusieurs tâches prêtes ont la même priorité, elles sont exécutées à tour de rôle ou par le Round Robin. Le principe de ce dernier se repose sur le partage du temps CPU entre ces tâches de manière circulaire [3].

- **Structure du noyau FreeRTOS :**

FreeRTOS est destiné aux systèmes embarqués vu sont principale avantage qui se repose sur sa très faible taille. Le noyau FreeRTOS est composé de trois fichiers source écrits en langage C, ces fichiers fournissent les trois fonctions suivantes :

task.c : C'est là où sont définit les tâches ainsi que les mécanismes d'ordonnancement.

queue.c : C'est dans ce fichier que les structures utilisées pour la communication et la synchronisation (files d'attente) des tâches y sont décrites.

list.c : On trouve dans ce fichier, la structure des donnée des listes ainsi que leurs fonctions de maintenance, sachant que les listes sont utilisées à la fois par les fonctions des tâches et les files d'attente (queue) [4]**.**

- **Options de configuration de FreeRTOS :**

ConfigUSE_PREEMPTION = 0 :l'ordonnanceur n'est pas préemptif c'est-à-dire si une tâche T1 de faible priorité devance une tâche T2 de priorité plus haute, la tâche T1 continue quand même à s'exécuter.

ConfigUSE_TIME_SLICING = 1: Activation du round robin [2].

- **Installation :**

On va prendre l'exemple de l'installation de FreeRTOS sur une Raspberry, on va donc avoir besoin d'une carte Raspberry PI, un ordinateur avec un système d'exploitation de type linux et une carte mémoire de type microSD. On va suivre les étapes suivantes :

1. Formater la carte microSD en format FAT32.

2. Télécharger FreeRTOS port pour Raspberry PI https://github.com/Forty-Tw0/RaspberryPi-FreeRTOS.

3. Installer le « gcc-arm-none-eabi toolchain » : dans Linux ouvrir le terminal et écrire « sudo apt-get Install gcc-arm-none-eabi ».

4. Ouvrir le fichier « makefile » situé dans FreeRTOS dossier et vérifier la version et l'emplacement des deux lignes suivantes :

kernel.elf : LDFLAGS += -L''/usr/lib/gcc/arm-none-eabi/4.9.3'' -lgcc.

kernel.elf : LDFLAGS += -L''/usr/lib/ arm-none-eabi/lib'' –lc.

5. Changer la direction d'opération de terminal à la direction de dossier de FreeRTOS, le chemin doit être court pour éviter des problèmes de compilation.

Exemple : cd /home/ « nom d'utilisateur » / Raspberry pi-FreeRTOS-final/

6. Ecrire « make » pour compilation et création de kernel7.img, pour modifier le code de votre programme ouvrir « demo/main.c ».

7. Copier les fichiers :

- « kernel7.img » localisé dans Raspberry Pi-FreeRTOS-final dossier.
- « config.txt » localisé dans Raspberry Pi-FreeRTOS-final/boot_stuff dossier.
- «bootcode.bin » et « start.elf » téléchargé de:

https://github.com/raspberrypi/firmware/tree/master/boot

Dans la carte microSD.

8. Ejecter la carte et insérer dans la carte Raspberry PI et allumer.

9. Le FreeRTOS programme fonctionne si la LED verte de la carte « ACT LED » clignote [5]**.**

- **Les tâches :**

Une tâche est un programme C, avec une entrée qui est compilé dans une boucle infinie et qui ne retourne rien c'est-à-dire sans RETURN, quand la tâche n'est plus utile elle est effacée.

La fonction tâche doit contenir un pointeur de type **void** comme paramètre et retourner un **void**, cette fonction peut être utilisée pour créer plusieurs tâches [2]**.**

Une tâche généralement utilise trois zones :

- **Zone code** : instructions et constantes;
- **Zone données** : écrire ou lire;
- **Zone pile** : informations temporelles [2].

Toutes les informations d'une tâche sont contenues dans le **TCB** (Task Control Block), ce dernier est une structure de données décrivant une tâche (descripteur de tâches), sous FreeRTOS le **TCB** contient :

- ✓ Priorité de la tâche ;
- ✓ Les éléments que la tâche attend ;
- ✓ Le pointeur de la pile associée à la tâche ;
- ✓ Le pointeur de sommet de la pile [6].

Le TCB aide l'ordonnanceur à savoir quelle tâches aura le temps CPU [6].

- **Les différents états d'une tâche:**

READY : La tâche est prête pour être exécutée, elle doit attendre jusqu'à ce que la tâche de priorité plus haute finis son exécution pour passer à l'état RUNNING.

RUNNING : La tâche est en cours d'exécution (une seule tâche peut s'exécuter à la fois).

BLOCKED : La tâche dans cet état est en attente d'un évènement, elle ne peut pas être sélectionnée dans la liste prête; à l'état bloqué la tâche possède un délai après lequel elle sera débloquée même si l'évènement qu'elle attendait n'est pas arrivée, c'est ce qu'on appelle délai d'expiration.

SUSPENDED : Comme dans l'état bloqué, la tâche suspendue ne peut pas passer à l'état prêt sauf si l'API **xTaskResume ()** est appelé**,** mais ici elle ne possède pas de délai d'expiration, une tâche est suspendue via **vTaskSuspend ()** [6]**.**

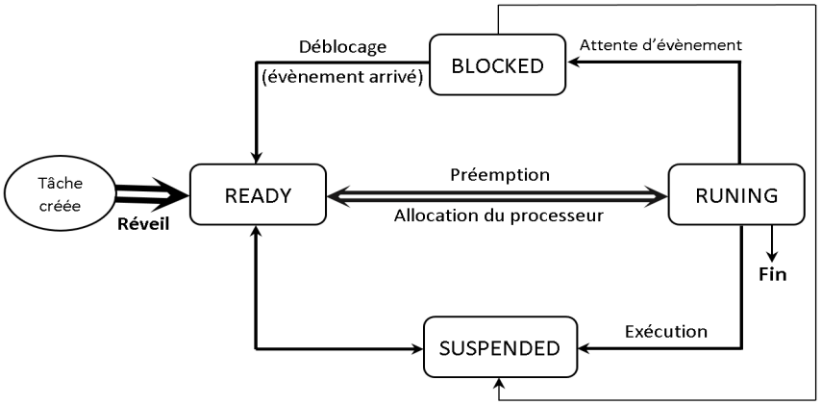


Figure I 1 Schéma qui représente les différents états d'une tâche.

Création d'une tâche sur FreeRTOS :

Sous FreeRTOS, le noyau alloue un espace TCB et la pile dans le Tas, ce dernier n'est qu'une zone de taille configurable appelée aussi Heap, chaque tâche possède sa propre pile dans le Tas, c'est-à-dire elle a un environnement d'exécution indépendant [2].

Une tâche est créée en invoquant la fonction **xTaskCreate ()** dans task.c, cette fonction créée une nouvelle tâche avec une priorité attribuée et l'ajoute à l'ensemble de tâches prêtes [3].

Exemple de création d'une tâche sous FreeRTOS :

```
BaseType_t xTaskCreate(
                  TaskFunction_t pvTaskCode,
                  const char * const pcName,
                  uint16_t usStackDepth,
                  void *pvParameters,
                  UBaseType_t uxPriority,
                  TaskHandle_t *pvCreatedTask
                );
```

Figure I 2 exemple de création d'une tâche sous FreeRTOS.

2.2 Présentation de l'ordonnancement

2.2.1 Ordonnancement des processus

Les systèmes temps réel ont des caractéristiques particulières, qui se résument par l'utilisation du temps concret et ce dernier est utilisé en différentes manières, soit en déterminant la date du début d'une action ou la date à laquelle cette action se termine qui est l'échéance. Réellement dans un système temps réel il peut y avoir plusieurs tâches qui évoluent simultanément soit en parallèle soit en concurrent, et afin de bien calquer la réalité et d'améliorer la conception, la programmation concurrente est bien recommandée et cela par utilisation de tâches et de processus concurrents, il faut que les moyens de communication et de synchronisation soient inter-tâches ou inter-process ce qui mène vers l'utilisation d'une mémoire partagée, boite aux lettres, fils de messages, moniteurs …etc [7].

Tout système temps réel doit respecter les contraintes temporelles, chaque tâche a un temps limité important qu'on appelle délai critique, c'est le temps maximal qu'il lui faut pour fonctionner à partir de sa date de réveil, la date finale résultante c'est l'échéance qu'il ne faut jamais dépasser si c'est le cas on aura une faute temporelle comme le montre le schéma ci-dessous[8] :

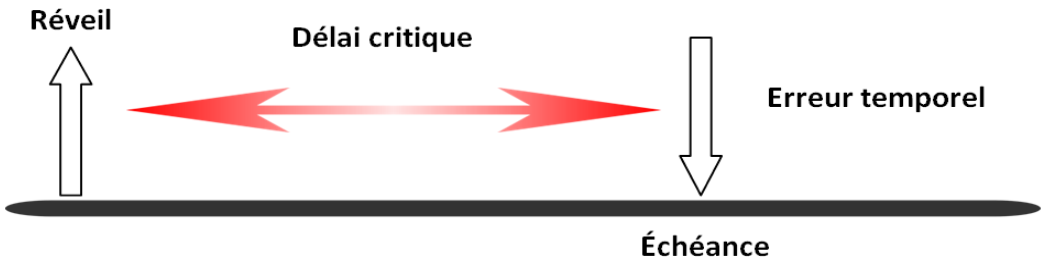


Figure I 3 Caractéristiques temporelles d'une tâche

Dans le cas de ressources limitées c'est-à-dire il y a un manque dans le processeur, les processus ne progressent pas, pour éviter cela la bonne gestion de la pénurie est nécessaire ainsi la favorisation des processus dont l'avancement ou l'exécution est le plus urgent (tâches les plus prioritaires) [7].

L'ordonnancement a pour but d'assurer le respect des contraintes temporelles liées aux processus et aux tâches, il consiste donc à créer une procédure impliquant l'utilisation des ressources du système dans le but d'atteindre les délais (échéances)[7].

2.2.2 L'ordonnanceur

On appel ordonnanceur le dispositif qui permet de gérer et d' organiser l'ordonnancement des tâches, en anglais c'est *scheduler,* il s'agit d'un programme qui contrôle l'exécution de divers processus et fonctions dans un système temps réel, il permet de choisir un élément du noyau du système qui va donc désigner l'ordre d'accomplissement des fonctions [8].

Il fait face à deux problèmes principaux qui sont le choix du processus à exécuter ainsi que le temps d'allocation du processeur au processus choisi; selon les caractéristiques du système ou processus, la stratégie d'ordonnancement ou bien l'algorithme d'ordonnancement est choisi.

Il existe deux façons pour appeler l'ordonnanceur :

- Soit à chaque unité de temps c'est-à-dire à chaque intervalle régulier ;
- Ou bien à chaque fin d'exécution d'une tâche [9].

A l'instant de l'appel de l'ordonnanceur, ce dernier choisit le travail de la tâche qui sera exécuté à cet instant, si la tâche n'a pas de travail à ce temps-là, le processeur reste inactif.

2.2.3 Les classes d'algorithmes d'ordonnancement

Un ordonnancement **fiable** c'est celui qui satisfait les contraintes temporelles et celles de ressources, si cet ordonnancement existe bien alors on dit que le système temps réel est ***ordonnançable,*** on distingue plusieurs classes de stratégies d'ordonnancement selon les caractéristique des systèmes, dans ce qui suit nous citerons quelques critères de problèmes d'ordonnancement [10]:

- **Ordonnancement statique et l'ordonnancement dynamique**

Dans l'ordonnancement statique tous les paramètres assignés aux tâches sont connus dès le départ ce qui implique la possibilité de prédictibilité, ce types d'ordonnancement est adapté aux systèmes figés ; à l'inverse de l'ordonnancement dynamique les paramètres des tâches ne sont connus qu'au moment de leur activation donc il n'y a pas de prédictibilité [11].

- **Ordonnancement hors ligne et en ligne :**

En s'appuyant sur les paramètres de tâches, un ordonnancement hors ligne indique que la séquence d'ordonnancement est calculée à l'avance et qu'elle sera exécuter en ligne par un dispatcheur (automate) ; dans le cas de l'ordonnancement en ligne qui signifie que la séquence d'ordonnancement est conçue au cours de l'exécution et que les tâches à exécuter sont choisies selon leurs priorités [8].

- **Ordonnancement oisif et non oisif :**

Un ordonnanceur est dit oisif, s'il fonctionne par insertion de temps creux c'est-à-dire avec un délai de retard, dans lequel le processeur reste inactif même si une tâche est prête ,par exemple si une tâche n'a pas de travail donc le processeur reste oisif, c'est aussi le cas des systèmes dont l'ordonnançabilité exige qu'aucune tâche ne sera choisie pendent un délai précis,

l'ordonnanceur non oisif fonctionne sans temps creux, autrement dit, si une tâche est prête à s'exécuter elle sera choisie par l'ordonnanceur.

- **Ordonnancement monoprocesseur et multiprocesseur :**

Etant de donné que toutes les tâches s'exécutent uniquement sur le même processeur (un seul), on dit alors que l'ordonnancement est monoprocesseur, si le système dispose de plusieurs processeurs alors l'ordonnancement est multiprocesseur.

- **Ordonnancement optimal est non optimal :**

Un ordonnancement est optimal, s'il minimise le retard maximal, ce qui conduit au respect des contraintes temporelles (ordonnançabilité); prenons exemple d'un travail qu'il faut réaliser le plus rapidement possible, et qui demande de nombreuses tâches , qui seront exécutées sous certains contraintes, pour accélérer le processus plusieurs processeurs sont utilisés, ici la fonction objectif c'est bien la durée totale du travail si elle est minimiser, alors l'ordonnancement réalisé est optimal si non, l'ordonnancement est non optimal.

- **Ordonnancement centralisé et distribué :**

Dans un système à multiprocesseurs, si l'algorithme d'ordonnancement se fait localement en chaque nœud, alors l'ordonnancement est distribué, or si l'algorithme de tout le système s'applique uniquement sur un processeur favorisé alors, l'ordonnancement est centralisé.

- **Ordonnancement préemptif et non préemptif :**

Un ordonnancement est préemptif lorsque les préemptions sont autorisées[2].

Exemple :

Soient deux tâches t1 et t2 avec des priorités différentes tel que t2 est prioritaire sur t1:

La tâche t1 passe du mode exécution en mode prête, et elle attend la fin de l'exécution de la tâche t2 pour qu'elle puisse continuer son exécution, on dit que t2 préempte t1 car elle n'est plus prioritaire, comme le montre la figure ci-après :

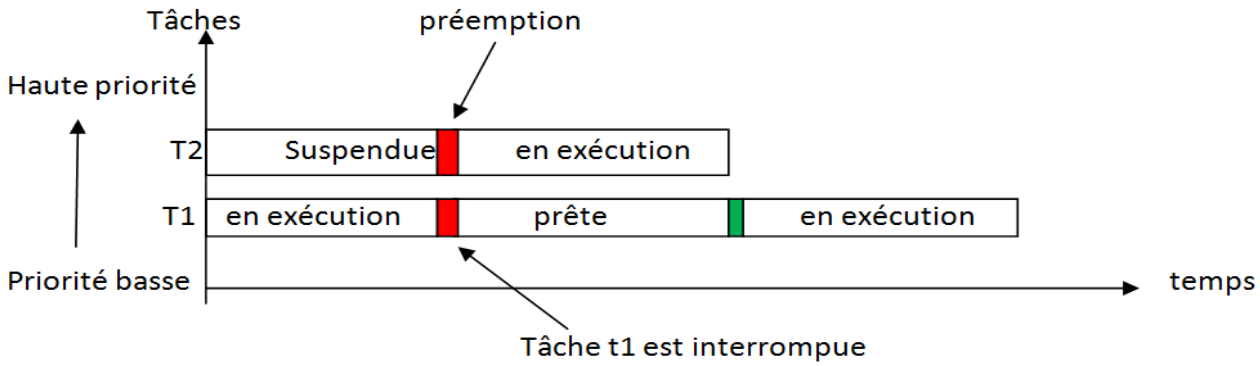


Figure I 4 exemple d'ordonnancement préemptif.

Dans le cas de l'ordonnanceur non préemptif il n'y a pas de préemption, ce qui élimine la problématique de synchronisation des ressources dans les systèmes monoprocesseurs [2].

2.2.4 Modélisation des tâches temps réel

2.2.4.1 Modélisation des tâches périodiques

On distingue quatre paramètres qui caractérisent les tâches temps réelles, soit T_i une tâche périodique (elle s'exécute en un intervalle de temps régulier), qui est caractérisé par les 4 paramètres suivants : **T_i (r_0, C_i, R_i, P_i).**

r_0 : la date de réveil de la première instance de la tâche T_i ;

C_i : la charge maximal du processeur nécessaire pour accomplir l'exécution d'une instance de la tâche T_i, ou bien la pire durée d'exécution notée (WCTE: Worst Case Execution Time) ;

R_i : l'échéance accordée à une tâche pour finir son exécution à chacune de ses activations ;

P_i : la période d'activation de la tâche T_i : après chaque P_i unité de temps à partir de r_0, une instance aura lieu [9].

Remarque : lorsque les tâches temps réel sont définies uniquement par leurs paramètres temporels, alors on dit qu'elles sont indépendantes.

2.2.4.2 Modélisation des tâches non périodiques (apériodiques et sporadiques)

- **Les tâches sporadiques :**

Une tâche est sporadique si son activation est non régulière, l'activation de ces tâches est limitée par une durée minimal et non pas par période, on distingue trois paramètres qui caractérisent une tâche sporadique T_i : (C_s, R_s, P_s) tel que :

C_s : la durée d'exécution de T_i ;

R_s : est l'échéance (délai critique) ;

P_s : le temps minimal entre deux activations successive [9].

- **Les tâches apériodiques :**

Les tâches apériodiques peuvent demandées à s'exécuter à n'importe quel instant, sans avoir un temps minimal d'invocation, étant donné que l'instant d'exécution d'une tâche n'est pas connu à priori et dépend du contexte d'évolution du processus, donc la date de réveil est aléatoire.

Le seul paramètre connu c'est bien la durée de l'exécution de la tâche, ainsi que le temps entre deux instances d'une tâche apériodique qui est soit connu en moyenne ou bien il est inconnu.

2.2.5 Les algorithmes d'ordonnancement des tâches indépendantes

Dans ce qui suit, nous décrivons les principaux algorithmes d'ordonnancement temps réel des tâches indépendantes, tout en donnant plus de détails sur l'un de ces algorithmes qui est l'algorithme EDF vu que c'est ce dernier qu'on utilisera dans notre étude.

2.2.5.1 Algorithmes à priorités statiques

Dans ce type d'algorithmes les caractéristiques temporelles des tâches sont connues à l'avance, et elles restent figées au fil du temps, on distingue deux algorithmes principaux qui sont:

- **L'algorithme RM (Rate Monotonic):**

Proposé par Liu et Layland en 1973, Il s'agit d'un algorithme à priorité fixe destiné aux tâches périodiques, cette technique consiste à affecter la priorité la plus élevée à la tâche de plus petite période, donc la priorité de la tâche dépend de sa période, ainsi que le temps d'exécution des tâches est connu à priori.

Soit un ensemble de **n** tâches périodiques à échéance sur requête (T_{pi} (r_0, C_i, R_i, P_i)) : Si la condition ci-dessous est vérifiée alors la configuration est ordonnançable :

La condition suffisante : $\sum_{i=1}^{n} \frac{Ci}{Pi} \leq n\,(2^{1/n\,-1})$[8].

- **L'algorithme DM (Deadline Monotonic) ou bien Algorithme inverse deadline :**

Cet algorithme est destiné aux tâches à échéance inferieur au période ($Ri \leq Pi$), la tâche la plus prioritaire est celle qui a le plus petit délai critique, alors la priorité des tâches dépend du délai critique.

La condition suffisante : $\sum_{i=1}^{n} \frac{Ci}{Ri} \leq n\,(2^{1/n\,-1})$.

Si cette condition est satisfaite donc la configuration de n tâches périodiques avec DM est ordonnançable [8].

2.2.5.2 Algorithmes à priorité dynamique

Les algorithmes présentés précédemment fournissent deux inconvénients majeurs :

- ils ne fournissent aucun mécanisme permettant de changer dynamiquement les priorités et de prendre en compte les tâches apériodiques.
- Ils présentent un risque d'inversion de priorités lorsque les tâches utilisent d'autres ressources que le processeur, par exemple une tâche de priorité inferieur bloque l'accès aux ressources d'une tâche de priorité supérieure.

Un autre type d'algorithmes est proposé, pour pallier aux limites citées ci-dessus ; il s'agit des algorithmes à priorité dynamique, on parle de priorité dynamique lorsque les priorités des

instances de la même tâche peuvent être différentes ; ils affectent aux tâches des propriétés variantes au cours de vie de l'application [9].

Par contre, ces algorithmes ont aussi des limites liées à leurs complexités de mise en œuvre, qui sont dues à la surcharge du processeur.

On distingue deux algorithmes à priorité dynamique principaux qui sont EDF et LLF, dans notre cas, nous allons nous intéresser plus particulièrement à l'algorithme EDF puisque c'est ce qu'on va utiliser dans notre étude.

2.2.5.2.1 L'algorithme Earliest Deadline First (EDF)

Il s'agit d'un algorithme à priorité dynamique pour tâches périodiques, qui à tout instant, alloue le processeur à l'instance de tâche dont l'échéance est la plus proche, c'est-à-dire la tâche la plus prioritaire à l'instant « t » est celle qui a le plus petit délai critique dynamique, donc la priorité de la tâche dépend de son délai critique dynamique ; à chaque arrivée ou terminaison d'une tâche, la priorité de la tâche est recalculée dynamiquement.

Cet algorithme est le meilleur du point de vu de faisabilité, et permet le taux le plus élevé d'utilisation du processeur à l'inverse des algorithmes RM et DM.

- Pour un ensemble de **n** tâches périodiques à échéance sur requête, T_{pi} (r_i, C_i, R_i, P_i), la configuration est faisable, si la condition suivante est satisfaite :

Condition nécessaire et suffisante : $\sum_{i=1}^{n} \frac{Ci}{Pi} \leq 1$;

- Pour un ensemble de **n** tâches périodiques quelconques T_{pi} (r_i, C_i, R_i, P_i),

La configuration est faisable si la condition suivante est satisfaite :

Condition suffisante : $\sum_{i=1}^{n} \frac{Ci}{Ri} \leq 1$[8]

Exemple avec EDF :

Soient trois (3) tâches périodiques avec leurs paramètres :

Tâche A (r_0=0, C=2, R=4, P=6) ;

Tâche B (r_0=0, C=3, R=8, P=8) ;

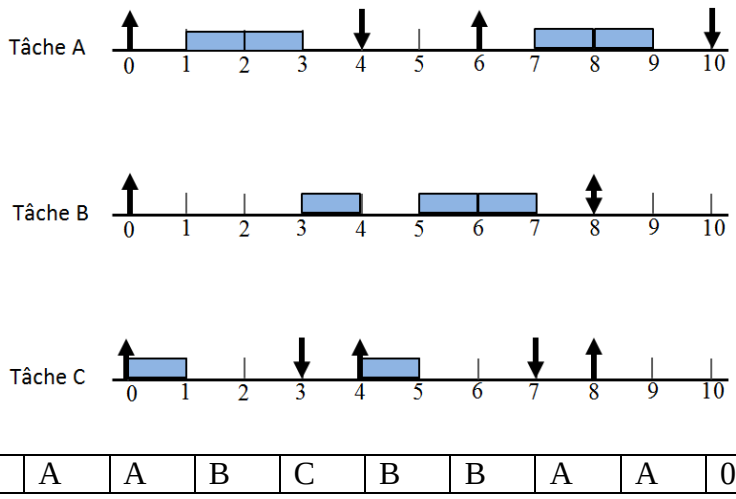
Tâche C (r_0=0, C=1, R=3, P=4).

Selon l'algorithme EDF, la tâche ‹C› est la plus prioritaire (c'est elle qui va s'exécuter en premier), car elle a l'échéance la plus proche (R=3).

Vérification de la condition d'ordonnançabilité :

U= $\sum_{i=1}^{n} \frac{Ci}{Pi}$ **=0.95,** U‹1 donc la configuration est faisable.[8]

Représentation graphique:[8]



C	A	A	B	C	B	B	A	A	0

Figure I 5 Exemple avec EDF.

Par conséquent, EDF permet d'exécuter les tâches selon l'ordre d'urgence mesuré par proximité de leur échéance, ce qui mène à dire qu'une instance ne peut utiliser la ressource que si toutes les instances avec échéances plus petites, ont terminées leur exécution ou ne sont pas encore actives.

Le plus grand avantage de EDF, il est optimal vis-à-vis l'ordonnançabilité du système dans des contextes variés, autrement dit, tout ensemble de tâches fonctionnelles sous un autre algorithme que EDF, sera fonctionnelle dans EDF.

Par contre, le plus gros inconvénient d'EDF, est son comportement instable en situation de surcharge. Néanmoins, plusieurs mécanismes ont été proposés pour améliorer le comportement d'EDF en situation de surcharge.

2.2.5.2.2 L'algorithme Least Laxity First (LLF)

LLF est un algorithme fondé sur la marge de manœuvre restante, c'est-à-dire il est basé sur la laxité, la tâche avec une laxité plus petite (celle avec l'exécution moins retardée) est la plus prioritaire. **LLF** possède la même capacité d'ordonnancement qu'EDF, par contre, le défaut que cet algorithme a est :

Un calcule de priorité ce fait à chaque changement de laxité, c'est-à-dire à chaque exécution en unité de temps (un nombre élevé de changement de contexte), donc il n'est pas si intéressant en microprocesseur qu'EDF.

$$\textbf{Laxité}(t)=P_i-t-C_{restant}(t)= r_i+R-t-C_{restant}(t).$$

Malgré les avantages que les algorithmes à priorité dynamique ont, et qui se résume sur leur flexibilité et leur optimalité, mais ils ont quand même un désagrément, qui est le fait de recalculé les priorités des tâches, qui conduit à la consommation du temps de système.

2.2.6 Les limites des algorithmes d'ordonnancement classiques

Les stratégies d'ordonnancement discutées si dessus (RM, DM, EDF et LLF), sont sensibles à de nombreuses lacunes qui sont :

- Connaissances des contraintes temporelles.
- Procédure de répartition des priorités.
- Environnement dynamique.

2.2.6.1 Connaissance des contraintes temporelles

La connaissance des contraintes temporelles des tâches à priori, est exigée par les algorithmes précédent, mais la définition du temps d'exécution de la tâche est délicate, vu qu'il dépend du langage principale ainsi que le système cible.

Des ennuis importants sont causés en conséquence de l'imprévisibilité du temps d'exécution, et dans certains cas, ce n'est pas suffisant de connaitre uniquement ce temps pour obtenir un ordonnancement fiable (le cas du processus de vision) [12].

2.2.6.2 Procédure de répartition des priorités

L'utilisation complète de la CPU conduit à des performances non satisfaisantes, et c'est l'exemple de procédures utilisées dans les algorithmes d'ordonnancement classiques ; par contre, pour avoir le comportement désiré, il est recommandé de découpé l'algorithme [12].

2.2.6.3 Environnement dynamique

Le temps d'exécution des tâches dépend aussi de l'environnement dans lequel elle s'exécute, cet environnement n'est pas forcement connu à priori, tant dit que dans d'ordonnancement classique, les algorithmes sont étudiés séparément du contexte applicatif, ce qui conduit à recommander l'ordonnancement optimal, du point de vu l'utilisation de ressources de calcul (exemple), mais qui n'est pas efficace pour le contrôle de l'application qui est le point de départ [12].

3 Conclusion

Ce chapitre a été consacré à la définition du système d'exploitation temps réel FreeRTOS. Après avoir présenté les notions d'ordonnancement, l'algorithme EDF a été définit tout en présentant ces avantages et ces limites ainsi que celles des autres algorithmes classiques.

Pour surmonter ces limites et afin d'avoir un ordonnancement plus fiable et robuste, l'ordonnancement régulé a été introduit. Ce dernier fera l'objet du chapitre suivant.

Chapitre II:

Ordonnancement par rétroaction

1 Introduction

Les systèmes temps réel, nécessitent des algorithmes spécifiques pour ordonnancer l'exécution des tâches et assurer le respect des contraintes de temps, et afin de tenir compte des changements des paramètres imprévisibles dans les systèmes dynamiques, on a recours à des ordonnanceurs par rétroaction.

Le présent chapitre est consacré à la définition du Feedback scheduling (ordonnancement régulé) et les travaux de ce dernier ; nous présentons également un algorithme en feedback scheduling très pratique qui est l'algorithme FC-EDF (Feedback Control EDF) que nous utiliserons dans notre étude.

2 Présentation de l'ordonnancement par commande rétroactive (feedback scheduling)

2.1 Loi de commande

En automatique, une boucle d'asservissement est composée du système à commander muni d'un actionneur, un capteur et un correcteur, le correcteur a pour objectif d'atteindre les performances désirées, et d'assurer la stabilité tout en rejetant toutes les perturbations internes ou externes.

La fonction primordiale d'un système commandé par rétroaction, est de mesurer en permanence l'écart entre la consigne désirée et l'état du système (la valeur réelle), d'une façon à le corriger efficacement, afin d'aboutir le plus rapidement possible à la consigne que l'on souhaite atteindre.

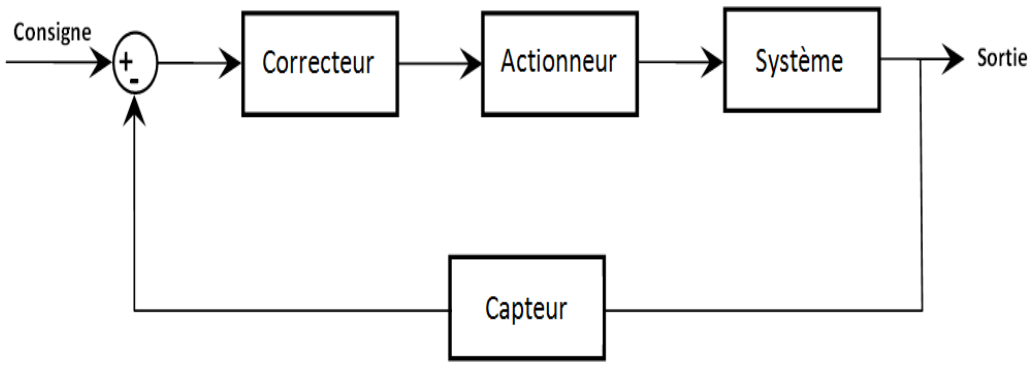


Figure II 1 Boucle d'asservissement d'un système commandé par rétroaction.

La loi de commande est conçue à partir des variables d'état, il s'agit d'une fonction ou programme qui donne la nouvelle commande à partir des mesures, on distingue plusieurs stratégies pour la synthèse du régulateur (régulateur PID, commande optimale, …etc.)

2.2 Le feedback scheduling

Les algorithmes d'ordonnancement présentés ci-dessus, comme EDF et RM sont en **boucle ouverte,** qui fait référence qu'une fois que les horaires sont créés et fixés, ils ne sont pas modifiés, alors que les algorithmes d'ordonnancement en boucle ouverte peuvent fonctionner avec des systèmes statiques et dynamiques, dans lesquels les conditions de travails peuvent être formalisés avec précision, mais ils peuvent fonctionner mal dans des systèmes dynamiques imprévisibles [13].

Afin de faire face à ce problème, et afin d'améliorer la qualité de commande du système d'exécution, le feedback scheduling consiste à adapter l'ordonnancement en temps réel avec un contrôleur en boucle fermée, en ajustant les paramètres d'ordonnancement du système en fonction des mesures faites sur lui grâce à un algorithme de commande, c'est le principe de l'ordonnancement par commande de rétroaction.

2.3 Les travaux de feedback scheduling

Il existe plusieurs études sur les commandes rétroactives, on va s'intéressé à quelques une qui sont basées sur l'optimisation en ligne :

2.3.1 Cervin

Dans le but d'améliorer la qualité de commande, tout en diminuant le délai de calcule ainsi la gigue d'échantillonnage, Cervin propose de partager la tâche de commande en deux sous-tâches tel que :

- La première sous-tâche contient les opérations que le calcul de la nouvelle commande nécessite immédiatement.
- La seconde sous-tâche contient les opérations nécessaires pour mettre à jours le correcteur.

Pour les tâches de contrôle, il propose une heuristique qui permet d'assigner les priorités au sous-tâches selon l'algorithme d'ordonnancement choisit [14].

2.3.2 Eker et all

Pour réparer les dégradations de la commande, Eker et all proposent d'utiliser un ordonnancement adaptatif ou régler, avec les résultats obtenus il est possible d'adapter la période des contrôleurs au taux d'utilisation du processeur, en utilisant les contrôleurs linéaires et quadratiques (LQ), l'optimisation de la période de ces derniers ce fait analytiquement avec des fonctions linéaire ou quadratique, pour avoir un ordonnancement favorable, il suffit de dilater les périodes nominales en fonction de l'occupation du processeur désirée [15].

Architecture proposée:

Eker et all proposent un modèle de tâches où ces dernières ont plusieurs modes de fonctionnement avec des durées d'exécution différentes, chaque contrôleur est associé à une fonction coût qui indique sa performance, l'optimisation de cette dernière est l'objectif primordiale du feedback scheduler, qui fait en sorte d'ajuster en ligne les période d'échantillonnage des correcteurs afin de garder l'ordonnançabilité du système, et d'éviter la surcharge du processeur et le dépassement d'échéance de quelques tâches.

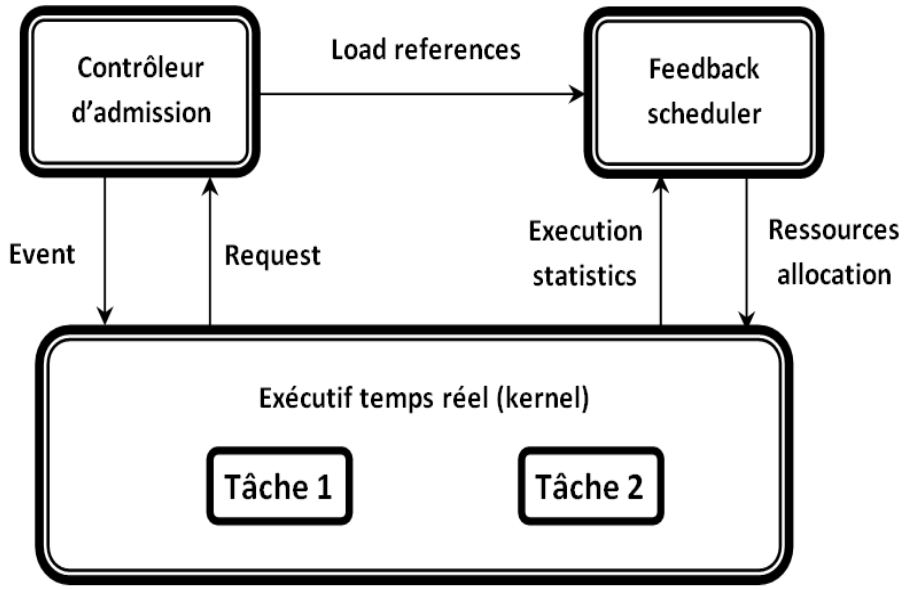


Figure II 2 Architecture feedback scheduler d'Eker

- **Feedback scheduler** : permet d'ajuster les intervalles d'échantillonnage pour assurer l'ordonnançabilité du système, pour cela il a besoin de connaitre les statistiques d'exécution.

- **Contrôleur d'admission** : permet le calcul de la charge de travail référence, et d'assurer la communication entre les tâches et le feedback scheduling en utilisant des requêtes ou des événements : quand une tâche nécessite plus de temps afin de s'exécuter, elle envoie une requête au contrôleur d'admission, qui va transmettre son accord par un événement, après avoir recalculer la charge de travail de référence et l'envoyer au feedback scheduler [15].

2.3.3 Cervin et all

Leur solution s'appuie sur la création d'ordonnancement de tâches de contrôle conjointement par l'utilisation des contrôleurs hybrides tels que, entre différents modes de fonctionnement les durées d'exécution changent brusquement. Alors ils proposent de garder le processeur occupé à un niveau acceptable ce qui permet d'éliminer le dépassement, ainsi la distribution de ressources entre les tâches convenablement par la reconfiguration linéaire des périodes nominales [16].

2.3.4 Control server Model de Cervin

Ce nouveau modèle de tâches, issues des travaux de Cervin et Eker [17]vise à faciliter la co-conception des applications de temps réel, il doit respecter 4 critères :

- Assurer l'indépendance entre les tâches ;
- Assurer des courtes latences entrées/sorties ;
- Minimiser les gigues d'échantillonnage ;
- Fournir une interface en l'implémentation du contrôleur et e l'ordonnancement temps réel.

En prenant en considération les latences et les gigues, ce modèle rapporte une souplesse dans la Co-conception ordonnancement/contrôle.

Dans le Contrôle server(CS), les tâches de contrôle ne dépassent pas les limites de ressources qui sont alloué aux CBS (Constant Bandwith Server) associés à chaque tâche.

2.3.5 Le feedback scheduling de Xia et all [18][19][20]

Dans cette étude on a considéré les processeurs embarqués aux ressources limitées, la période et la durée d'exécution des tâches de contrôle contiennent des incertitudes.

Le modèle de tâches, consiste à associé les décisions de l'ordonnancement à la dynamique des procédés contrôlés, ce modèle doit respecter 3 critères :

- Fournir les paramètres de base pour le fonctionnement des algorithmes ; d'ordonnancement classiques ;
- Prendre en considération la dynamique des processus ;
- Faciliter l'intégration contrôle/ordonnancement (les tâches possèdent 3 caractéristiques : le temps d'exécution Ci, la période Ti et l'urgence Uri) ;

Ce modèle utilise un algorithme d'ordonnancement appelé MUF (Maximum Urgency First), pour cet algorithme le paramètre prioritaire est l'urgence, donc tâche qui s'exécute en premier est la tâche avec le Ur le plus grand. Avec cette architecture les incertitudes de la période et la durée d'exécution ont un effet quasi nul sur l'ordonnancement.

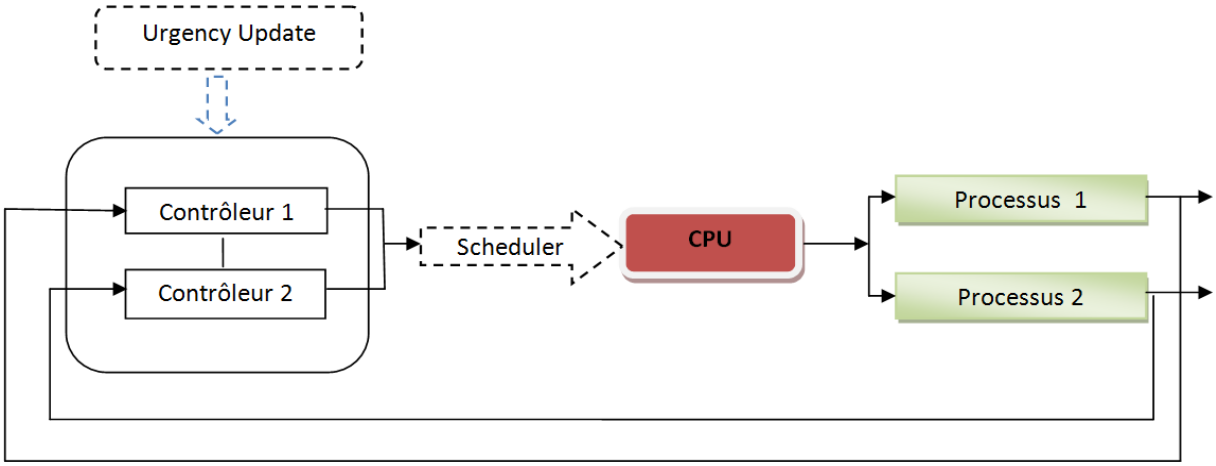


Figure II 3 Architecture du feedback scheduling de XIA.

3 Algorithme Feedback Control Earliest Deadline First (FC-EDF)

3.1 Le contrôleur PID

Le régulateur PID est très répondu dans l'industrie, car il peut contrôler un bon nombre de procèdes en boucle fermée, le paramètre d'entrée d'un PID est l'erreur (différence entre la consigne et la mesure). Il combine entre 3 actions :

- **L'action proportionnelle(P)** : qui consiste à multiplier l'erreur par un gain K ce qui garantit la rapidité du système commandé, si K est trop grand y aura un dépassement.

$$c(t) = K\varepsilon(t)$$

- **L'action intégrale (I)** : cette action consiste à intégrer l'erreur sur une période puis multiplier par un gain K_i, afin de réduire voir éliminer l'erreur statique par conséquent un système plus stable en régime permanant. Pour avoir une correction plus rapide, il faut augmenter le gain si le K_i est trop grand le régime permanant sera long à atteindre.

$$c(t) = K_i \int \varepsilon(t)dt$$

- **L'action dérivée (D)** : cette action consiste à dériver l'erreur sur une période puis multiplier par un gain K_d. En augmentant K_d le dépassement diminuera ainsi que le temps d'établissement du régime stationnaire, mais la sensibilité au bruit augmentera.

$$c(t) = K_d \frac{d\varepsilon(t)}{dt}$$

- **Action proportionnelle- intégrale-dérivée (PID):**

Le contrôleur PID est alors obtenu en combinant les trois actions, Ce qui lui permet d'agir sur la stabilité, l'erreur statique et la rapidité du système.

$$u(t) = K \times \varepsilon(t) + K_i \int \varepsilon(t)dt + K_d \frac{d\varepsilon(t)}{dt}$$

3.2 Feedback control EDF (FC-EDF)

Afin d'utiliser les techniques de la commande par rétroaction sur l'ordonnancement, il faut d'abord définir : les variables du correcteur, les variables à manipuler, la consigne, l'erreur, la loi de commande, et le mécanisme des actionneurs, donc le système commandé en feedback scheduling, commence par les hypothèses sur les caractéristiques temporelles des tâches entrantes (date de début d'exécution, la durée d'exécution, l'échéance), puis ce système fait en sorte de comparer les mesures réelles obtenues, avec les mesures exigées par le système tout en corrigeant les erreurs [12].

L'algorithme FC-EDF, intègre le régulateur PID avec l'ordonnanceur EDF ; pour appliquer le régulateur PID sur un système d'ordonnancement, il est nécessaire de choisir l'objectif en premier, c'est-à-dire, la variable à réguler ainsi que la consigne de ce système, cet algorithme doit toujours minimiser le temps d'exécution des tâches, dans le but d'éviter les fautes temporelles et le dépassement de l'échéance.

Supposons que toutes les tâches sont indépendantes, dans ce cas si le système n'est pas en surcharge, le taux de dépassement de l'échéance par EDF est de 0%, alors que les deux objectifs de FC-EDF, en premier c'est de maintenir ce taux de non-respect des délais prêt ou à 0, deuxièmement, c'est d'accepter le maximum de tâches possible (risque de surcharge) [21].

Le premier objectif est atteint grâce à la commande par rétroaction: la variable à réguler utilisée par l'ordonnanceur est **MissRatio(t)**, c'est le taux d'échec de l'échéance en temps t, la consigne et **MissRatio=0**, alors l'erreur c'est :

$$\text{Erreur} = \text{MissRatio} - \text{MissRatio}(t) = -\text{MissRatio}(t)$$

Remarque :

Un système qui atteint le taux d'échéance de 0%, mais rejette beaucoup de tâches n'est pas acceptable. Pour éviter cette condition dans FC-EDF, il est recommandé d'augmenter l'utilisation du processeur quand le système est surchargé. Le raisonnement dit, qu'EDF peut garantir le taux 0% une fois que le système n'est pas surchargé, en contrôlant l'utilisation du processeur qui est demandée prêt de 100%, ça conduit vers la minimisation du manque ratio de toutes les tâches entrantes [21].

Pour manipuler l'utilisation de la CPU demandée, il est nécessaire de concevoir un mécanisme (actionneurs) utilisé par l'ordonnanceur, ce mécanisme inclus le contrôleur d'admission ainsi que le contrôleur de niveau. Comme le montre la figure qui suit[21] :

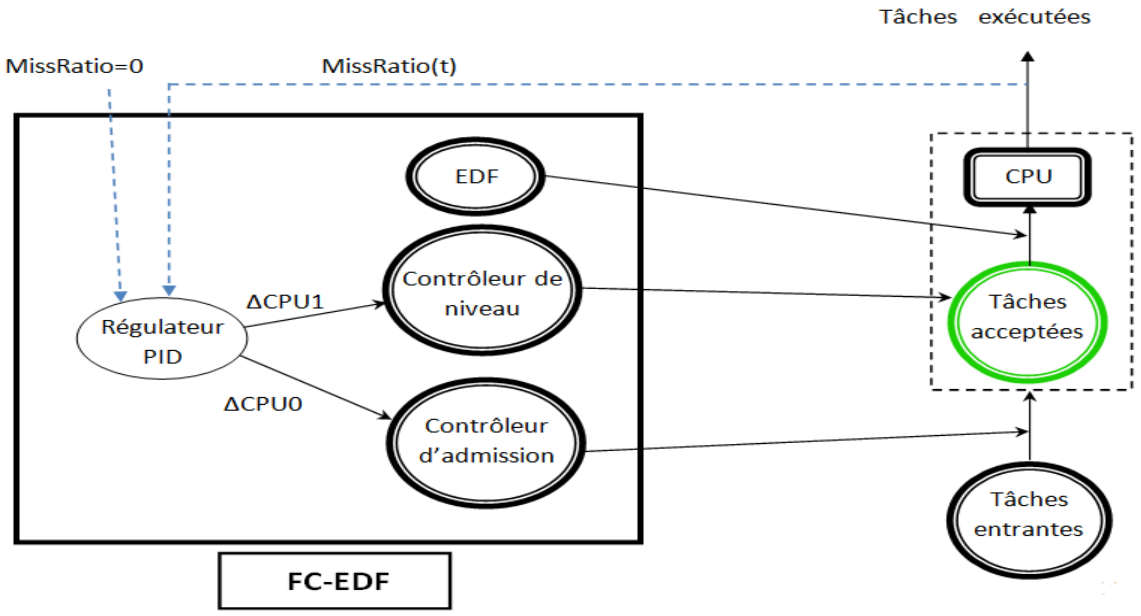


Figure II 4 L'architecture de l'ordonnanceur FC-EDF.

3.3 Rôle et fonctionnement des différents composants de FC-EDF

En résumé, l'ordonnanceur FC-EDF est composé d'un Régulateur PID, Régulateur de niveau, Contrôleur d'admission et d'un ordonnanceur EDF.

a. Régulateur PID :

Le contrôleur PID est le cœur de FC-EDF, il permet mapper le taux d'échec des tâches acceptées (c'est-à-dire les erreurs) selon le changement d'utilisation demandée (c'est-à-dire la commande) afin d'aboutir à la consigne désirée (c'est-à-dire ramener le taux d'échec égale à 0). Pendant chaque période, le PID contrôle la variable régulée MissRatio(t) et calcule ΔCPU(t) par l'équation suivante :

$$\Delta\mathrm{CPU(t)} = -\mathrm{K} \times \mathrm{MissRatio(t)} - \mathrm{Ki}\sum_{iw}\mathrm{MissRatio(t)} + Kd\,\frac{\mathrm{MissRatio(t-DW)} - \mathrm{MissRatio(t)}}{\mathrm{DW}}$$

Où :

K, K_i, K_d, Ps, IW et DW sont des paramètres réglables du régulateur PID, tel que :

Ps : est la période d'échantillonnage durant laquelle le régulateur est appelé ;

K, K_i et K_d : sont les coefficients du régulateur PID ;

IW : est l'intervalle de temps sur lequel les erreurs sont additionnées (Seules les erreurs dans les dernières unités de temps IW seront prises en compte dans le terme intégral) ;

DW : est l'intervalle de temps des erreurs dérivées (Seul le changement d'erreur au cours des dernières unités de temps DW sera pris en compte dans le terme dérivé, c'est-à-dire que l'erreur dérivée est $(\frac{\mathrm{MissRatio\ (t-DW)} - \mathrm{MissRatio\ (t)}}{\mathrm{DW}})$).

La sortie du PID est bien ΔCPU(t) qui est le signal de commande, si il est positif (ΔCPU (t)> 0), alors l'utilisation demandée du processeur doit être augmentée, si non cette dernière doit être diminuée.

L'équation précédente permet uniquement de contrôler le taux d'échec des tâches acceptées, or FC-EDF vise également d'accepter autant de tâches que possible afin de minimiser le taux d'échec de toutes les tâches entrantes. Pour ce faire, un ajustement de ΔCPU (t) est inclus dans le régulateur PID comme suit :

Si $(\Delta CPU(t) \geq 0)$: $\Delta CPU(t) = \Delta CPU(t) + \Delta CPU_A$

Avec:

Le ΔCPU_A appelé le *greedy-factor* qui est un paramètre réglable, il est rajouté quand le système est un peut charger dans le but d'augmenter la charge de la CPU, pour obtenir plus de travail c'est-à-dire accepter plus de tâches.

On veut dire par $\Delta CPU(t) \geq 0$ que le système est en surcharge légère et fonctionne bien.

Pour modifier l'utilisation CPU demandée par ΔCPU, le PID appellera les deux contrôleurs de niveau et d'admission. Si l'utilisation actuelle demandée est CPU(t), le SLC et l'AC changeront l'Utilisation par CPU (t) + ΔCPU.

Le contrôleur PID appelle d'abord le SLC en premier afin que le système puisse répondre à l'erreur plus rapidement (en supposant que les tâches précédemment acceptées ont tendance à avoir des délais plus courts que les tâches acceptées plus tard par le contrôleur d'admission). Le contrôleur d'admission est appelé uniquement si le contrôleur de niveau n'arrive pas à accueillir complètement le ΔCPU.

Ce qui suit est le pseudo code du contrôleur PID :

```
/* il est appelé  pendant chaque période ps*/
Void PID ()
{
Get MissRatio (t) during last sampling period PS;
/*la fonction du régulateur PID*/
ΔCPU(t) = -K*MissRatio(t)-KI*SIW MissRatio(t)+Kd*(MissRatio(t-DW) -MissRatio(t))/DW
/* augmentation de la charge du système lorsqu'il est légèrement chargé */
If (ΔCPU(t) ≥ 0)
ΔCPU(t) = ΔCPU(t) + ΔCPUA

/* appel du SLC,  qui renvoie la partie de ΔCPU (t) qui n' est pas terminée  */
ΔCPUo = SLC (ΔCPU(t));
/* appelle du contrôleur d'admission pour prendre en charge la partie de ΔCPU (t) qui n'est
pas complétée par SLC, s'il y en a */
If (ΔCPUo != 0)
ACadjust (ΔCPUo);
    }[21].
```

b. Contrôleur de niveau de service:

En anglais c'est Service Level Controller (SLC), le contrôleur de niveau a pour rôle d'ajuster le niveau de service des tâches acceptées afin de modifier l'utilisation du processeur demandée dans le système.

Prenons exemple d'une tâches T_i avec une utilisation CPU de $C_i/P_i(a)$ qu'on désire changer à $C_i/P_i(n)$. le SLC adapte l'utilisation système demandée à $C_i/P_i(a)-C_i/P_i(n)$, tel que C_i est la durée d'exécution de la tâche T_i et Pi(a) et Pi(n) sont respectivement la période actuelle et la nouvelle période qui va être appliquée.

Ajuster le niveau de service d'une tâche implique la modification de sa période, vu la relation complémentaire entre ces deux paramètres, cette relation est définie dans les équations suivantes :

$N_i = C_i/P_i$ et $U = \sum N_i \Rightarrow U = \sum C_i/P_i$

Avec :

N_i : Niveau de service de la tâche T_i ;

U : utilisation CPU de tout le système (toutes les tâches).

En gros, le niveau de service dépend de la période, il suffit donc d'ajuster cette dernière afin de changer le niveau de service d'une tâche. Ainsi on remarque qu'on accroissant la période la surcharge du processeur est atténuée.

La fonction SLC renvoie la partie de ΔCPU qui n'a pas pu être ajustée par elle-même, ce cas de figure arrive lorsqu'il n'existe pas plus de taches dégradées et que le système est toujours en surcharge ($\Delta CPU_1 < 0$).Cette partie du contrôle sera ajustée par le contrôleur d'admission. Ce qui suit est le pseudo code du contrôleur de niveau de service :

```
double SLC (ΔCPU)
{
ΔCPU1=ΔCPU;
if (ΔCPU1 <0)
while (ΔCPU1<0 && Exist Degradable Task) {
Ti = Select_Degraded_Task ();
ChangeSeviceLevel(task,current_level,new_level);
```

$\Delta CPU_1 = \Delta CPU_1 - C_i/P_i(n) + C_i/P_i(a);$

}

else

while($\Delta CPU_1 > 0$ && Exist Enhancable Task) {

Ti = Select_Enhanced_Task();

ChangeSeviceLevel(task,current_level,new_level);

$\Delta CPU_1 = \Delta CPU_1 - C_i/P_i(n) + C_i/P_i(a);$

}

/ retourne* ΔCPU_1, la *partie de* ΔCPU *non prise en charge.* $\Delta CPU1$ *sera ajustée par le contrôleur d'admission */*

return ΔCPU_1;

}[21]

c. Contrôleur d'admission (Admission Controller):

Le contrôleur d'admission (AC) permet de contrôler les tâches entrantes dans le système, quand une nouvelle tâche T_i est soumise au système, c'est le contrôleur d'admission qui décide si elle peut être acceptée ou pas, selon l'utilisation actuelle de processeur et selon la demande de CPU des tâches entrantes qu'on note par ET_{i1}, donc le contrôleur d'admission procède de cette manière :

Si CPU (t) + ET_{i1} <1 alors la tâche T_i est acceptée si non le AC rejette T_i.

Le paramètre CPU (t) du contrôleur d'admission est ajusté quand le PID n'a pas pu adapter ΔCPU (t) complètement avec le SLC.

Admettons que le SLC change l'utilisation demandée par ΔCPU_1 et que $\Delta CPU_1 < \Delta CPU$, dans ce cas AC va ajuster cette partie non fini par SLC par $\Delta CPU_0 = \Delta CPU - \Delta CPU_1$. Et cela en adaptant l'estimation de l'utilisation exigée CPU (t) comme suit :

Void ACadjust (ΔCPU_0)

{CPU (t)= CPU (t) - ΔCPU_0 ;}[21].

d. Ordonnanceur EDF :

L'ordonnanceur EDF, ordonne les tâches acceptées selon l'algorithme EDF.

4 Conclusion

Le chapitre présenté ce porte principalement sur l'ordonnanceur en feedback scheduling FC-EDF, qui se compose d'un régulateur PID, régulateur de niveau, contrôleur d'admission et d'un ordonnanceur EDF. Vu les avantages que cette combinaison offre, nous allons l'implémenter dans notre système d'exploitation temps réel FreeRTOS, ce qui permet à ce dernier de traiter les différents aspects de processus temps réel qui ne peuvent pas être traitées avec l'ancien ordonnanceur (SPP).

La procédure de cette implémentation sera présentée dans le chapitre qui suit.

Chapitre III: Ordonnancement par rétroaction sous FreeRTOS

1 Introduction

L'ordonnancement régulé a été fait sous plusieurs RTOS à l'exemple de Xenomai ou TrueTime. En outre, il n'existe aucun travail dans la littérature qui a réalisé le feedback scheduling sous FreeRTOS.

L'étude de notre travail se base sur l'implémentation de l'algorithme FC-EDF sous FreeRTOS. Dans ce chapitre nous présentons en premier lieu le eTCB (extended Task Control Bloc) qui réalise le FC-EDF sous FreeRTOS et l'expliquer d'une façon détaillée. En deuxième lieu, nous présentons les résultats obtenus c'est-à-dire la nouvelle version de FreeRTOS en les illustrant avec un exemple d'application.

2 Solution proposée

Dans le but d'adapter FreeRTOS à un nouveau mécanisme d'ordonnancement, il existe deux options :

a. Modifications du noyau du FreeRTOS et insertion de la nouvelle politique d'ordonnancement désirée, ce choix exige une connaissance approfondie de la structure de notre RTOS, il peut donc entrainer la perte de certification, de sécurité, de la robustesse et de la fiabilité.

b. Apporter un nouvel ordonnanceur au niveau de l'utilisateur afin de remplacer les actions du planificateur du noyau de FreeRTOS. Cet ordonnanceur est développé via une tâche au niveau utilisateur appelée **FST** (Feedback Control Task), tâche d'ordonnancement par rétroaction.

Dans notre cas, nous avons opté pour la seconde alternative afin de garantir la fiabilité, la robustesse et la sécurité pour notre système [22].

2.1 Architecture proposée

Afin de réaliser le feedback scheduling sous FreeRTOS, nous avons choisi l'algorithme FC-EDF (Feedback Control EDF), qui est une extension de l'algorithme EDF, comme nous l'avons expliqué dans le chapitre précédent, le FC-EDF utilise une rétroaction PID dans le but de prévoir la charge à venir, cette prévention conditionne le choix de la version de chaque tâche à accepter, pour des tâches qui existent en plusieurs versions.

En se référant au travail réalisé par Páez [22]nous avons conçu notre solution comme suit :

2.1.1 Présentation du TCB étendu (eTCB)

Sous FreeRTOS chaque tâche est définie par un TCB. Par contre, ce dernier n'inclut pas d'attributs pour contenir des paramètres particuliers pour des modèles de tâches génériques.

Pour faire face à ce problème, un TCB étendu a été défini, noté eTCB qui a une référence au TCB pour garder le fonctionnement des opérations de contrôle des tâches FreeRTOS [22].

L'eTCB d'une tâche dispose des paramètres suivants:

- Pointeur vers FreeRTOS TCB (gestionnaire des tâches) ;
- Importance de la tâche (Ii);
- Pire délai d'exécution du Tas (Ci) ;
- Période min de la tâche (P MINi) ;
- Période max de la tâche (PMAXi) ;
- Délai relatif (Di) ;
- Délai absolu (di) ;
- Dernière heure de sortie (xi) ;
- Nombre d'instances (ji) ;
- Temps d'exécution actuel à l'instant t (ci (t)) ;
- Pointeur vers la structure des paramètres supplémentaires.

Au moyen de ce dernier attribut, l'eTCB peut être étendu afin de l'adapter aux exigences spécifiques d'une politique de programmation [22].

Notons que chaque tâche possède plusieurs configurations autrement dit plusieurs temps d'exécution, périodes.

L'eTCB est accessible par la FST (Feedback Scheduling Task) Tâche d'ordonnancement régulé [22].

2.1.2 Implémentation de l'eTCB

Dans cette section nous présentons l'implémentation du TCB étendu dans FreeRTOS ; le code ci-dessous commence d'abord par la création d'une tâche sous FreeRTOS, cette création est assurée par la fonction "**xSchedulerTaskCreate"** qui définit tous les attributs de la tâche à créer (priorité, période min, période max, importance, échéance, durée d'exécution…).

Ensuite, la fonction **"xSchedulerTaskCreate"** fait appel à la fonction FreeRTOS conçu spécifiquement à la création des tâches appelée **" xTaskCreate" (**ligne 98).

Enfin la tâche créée est ajoutée à la liste des tâches prêtes (Ready) à partir de laquelle l'ordonnanceur choisit celle qui s'exécute selon le principe du FC-EDF.

Comme toutes les informations et les paramètres des tâches sont contenues dans le eTCB, ce dernier est accessible par un pointeur **" pxTaskFstTCB",** comme le montre le code ci-après :

```
67  /* AppSched_TaskCreate()*/
68  BaseType_t xSchedulerTaskCreate( TaskFunction_t pxTaskCode, const char * const pcName,
69  const uint16_t usStackDepth, void * const pvParameters,UBaseType_t uxPriority,
70  FstTCB_t **pxCreatedTask, TickType_t xPeriodMin,TickType_t xPeriodMax,TickType_t xImportance,
71  TickType_t xDeadline, TickType_t xWcet )
72  {
73      FstTCB_t *pxTaskFstTCB = ( FstTCB_t * ) pvPortMalloc( sizeof( FstTCB_t ) );
74      BaseType_t xResult = pdFAIL;
75
76      if( pxTaskFstTCB != NULL )
77      {
78          /* Initialize the scheduler tasks TCBe members. */
79          pxTaskFstTCB->xPriority = uxPriority;
80          pxTaskFstTCB->xPeriodMin = xPeriodMin;
81          pxTaskFstTCB->xPeriodMax = xPeriodMax;
82          pxTaskFstTCB->xImportance = xImportance;
83          pxTaskFstTCB->xDeadline = xDeadline;
84          pxTaskFstTCB->xAbsoluteDeadline = xDeadline;
85          pxTaskFstTCB->xRelease = 0;
86          pxTaskFstTCB->xWcet = xWcet;
87          pxTaskFstTCB->xWcrt = 0;
88          pxTaskFstTCB->uxReleaseCount = 0;
89          pxTaskFstTCB->xCur = 0;
90          pxTaskFstTCB->xFstTaskType = FST_PERIODIC;
91          pxTaskFstTCB->xState = FST_READY;
92
93          if ( pxTaskFstTCB->xPeriod == 0 )
94          {
95              pxTaskFstTCB->xFstTaskType = FST_APERIODIC;
96          }
97          /* Create the FreeRTOS task. */
98          xResult = xTaskCreate( pxTaskCode, pcName, usStackDepth, pxTaskFstTCB, TASK_PRIORITY, &( pxTaskFstTCB->xHandle ) );
99
100         if( xResult == pdPASS)
101         {
102             if( ( void * ) pxCreatedTask != NULL )
103             {
104                 /* Pass the TCBe out. */
105                 *pxCreatedTask = pxTaskFstTCB;
106             }
107
108             /* Initialize task absolute deadline item. */
109             if ( pxTaskFstTCB->xFstTaskType == FST_PERIODIC )
110             {
111                 vListInitialiseItem( &( pxTaskFstTCB->xAbsDeadlineListItem ) );
112                 listSET_LIST_ITEM_OWNER( &( pxTaskFstTCB->xAbsDeadlineListItem ), pxTaskFstTCB );
113                 listSET_LIST_ITEM_VALUE( &( pxTaskFstTCB->xAbsDeadlineListItem ), pxTaskFstTCB->xAbsoluteDeadline );
114                 vListInsert( &xAbsDeadlinesList, &( pxTaskFstTCB->xAbsDeadlineListItem ) );
115             }
116
117             /* Add the created task to the scheduler ready list. */
118             vSchedulerLogicAddTask( pxTaskFstTCB );
119
120             /* Associate the eTCB and TCB. */
121             vTaskSetThreadLocalStoragePointer( pxTaskFstTCB->xHandle, 0, ( void * ) pxTaskFstTCB );
122
123             /* The initial state of a FST scheduled task is suspended. */
124             vTaskSuspend( pxTaskFstTCB->xHandle );
125         }
126         else
127         {
128             vPortFree( pxTaskFstTCB );
129         }
130     }
131
132     return xResult;
133 }
```

2.2 La tâche FST (Feedback Control Task)

La FST permet de modifier la liste prête de FreeRTOS et elle est implémentée en tant que tâche normale, cette tâche est exécutée avec une priorité FST_P. D'autre part, les autres tâches FreeRTOS ont des priorités égales à FST_TP, qui sont configurables, mais ils doivent toujours satisfaire :

FST_P> FST_TP [22]

La libération FST a lieu sur les événements suivants:

1. Timer tick: cet événement a lieu à chaque interruption de l'horloge de la CPU.

2. Libération de la tâche: cet événement a lieu chaque fois qu'une tâche est libérée et insérée dans la liste prête (la tâche passe de l'état d'attente à l'état prêt).

3. Achèvement de la tâche: cet événement a lieu lorsqu'une tâche termine son exécution. Dans ce cas, la tâche est suspendue jusqu'à sa prochaine exécution.

4. Blocage de la tâche: cet événement a lieu lorsqu'une tâche nécessite une ressource partagée mais qui est accordée à une autre tâche.

5. Suspension de tâche: cet événement a lieu lorsque l'exécution d'une tâche est suspendue pendant un certain intervalle (à l'aide des fonctions de retard ou de veille) [22].

La figure qui suit, présente l'interaction entre FST et les tâches acceptées et le noyau FreeRTOS :

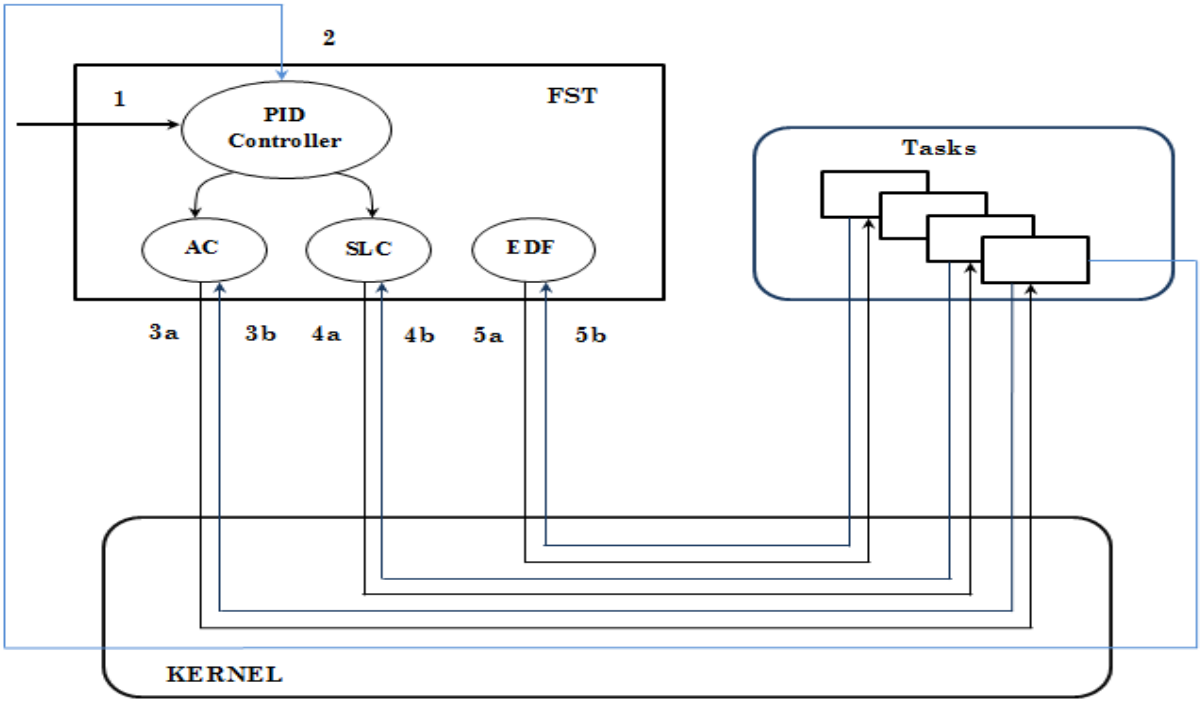


Figure III 1 Interaction entre FST et les tâches acceptées et le noyau FreeRTOS.

Tel que :

1-La consigne du régulateur;
2- Tâche qui n'ont pas respecté leurs échéances;
3a- Tâches à exécuter;
3b- Tâches acceptées et non acceptées;
4a- Caractéristiques des tâches acceptées (échéances ou périodes par exemple);
4b- Nouvelles caractéristiques à prendre en compte;
5a- Tâche terminée ou bloquée;
5b- Tâche à suspendre ou à reprendre l'exécution.

2.2.1 Les modules FST

La tâche FST est implémentée par 5 modules qui sont :

- Module AppSched;
- Module AppSchedEDF;
- Module AppSchedPID;
- Module AppSchedAC;
- Module AppSchedSLC.

2.2.1.1 Module AppSched

Ce module permet l'élaboration de la politique d'ordonnancement :

- **AppSched_FST ()** : Une fonction qui permet la mise en œuvre du comportement logique de la tâche FST. Cette fonction appelle une autre fonction dans le module AppSched_EDF qui est **AppSchedEDF_Sched (),** afin d'exécuter l'algorithme d'ordonnancement, ensuite elle se bloque.

Ce module définit aussi une API (Application Programming Interface), composée des fonctions suivantes:

- **AppSched_Init ():** crée le FST et ses ressources et exécute l'ordonnanceur RTOS.
- **AppSched_Create ():** crée une tâche ordonnancée par la FTS au lieu du planificateur RTOS.

La tâche créée est insérée dans la liste des tâches gérées par la FST, elle est initialement suspendue, pour éviter qu'elle soit exécutée par l'ordonnanceur RTOS.

- **AppSched_WaitForNextPeriod ():** bloque la tâche appelante jusqu'à sa prochaine période.

Le module définit également les fonctions privées suivantes, qui sont appelées lorsque certains événements ont lieu:

- **AppSched_Tick ():** cette fonction est appelée à chaque tick d'horloge RTOS.
- **AppSched_Ready ():** cette fonction est appelée lorsque la tâche passe de l'état bloqué ou suspendu à l'état prêt; Cette fonction doit vérifier que l'état de la tâche a été modifié par le RTOS au lieu du FST.
- **AppSched_Delay (), AppSched_Block () et AppSched_Suspend ():** sont des fonctions qui bloquent et débloquent la FST, elles sont appelées lorsqu'une tâche termine son exécution, elle est bloquée ou suspendue respectivement.

2.2.1.2 Module AppSchedEDF

Ce module implémente la nouvelle stratégie d'ordonnancement à travers les fonctions suivantes :

- **AppSchedEDF_Init ():** initialise toutes les structures de données exigées par la politique d'ordonnancement implémentée.
- **AppSchedEDF_add () :** ajoute ce qu'il faut au données précédentes.
- **AppSchedEDF_Tick ():** implémente l'action à effectuer à chaque interruption du tick d'horloge.
- **AppSchedEDF_Sched ():** implémente la politique d'ordonnancement EDF.

2.2.1.3 Module AppSchedPID

Ce module définit les fonctions qui permettent d'implémenter le régulateur PID, ces fonctions sont définies comme suit :

- **AppSchedPID_def ():** cette fonction permet de définir l'équation mathématique du régulateur PID;
- **AppSchedPID_CPU () :** cette fonction définit la sortie du régulateur (signale de commande), c'est-à-dire la charge de la CPU; la fonction appelle une autre fonction du module **AppSchedSLC** qui est **AppSchedSLC_slc ()** pour que cette dernière puisse renvoyer la partie de la CPU qui n'est pas terminée si nécessaire, dans le cas où

AppSchedSLC_slc () n'a pas pu compléter sa tâche une autre fonction du module **AppSchedAC** est appelée, la fonction est **AppSchedAC_ac ().**

2.2.1.4 Module AppSchedSLC

Ce module est en charge de la mise en œuvre du contrôleur de niveau (SLC), il est décrit par la fonction suivante :

- **AppSchedSLC_select_task () :** la fonction permet de sélectionner parmi les configurations de la tâche acceptée celle qui réalise l'utilisation demandée.
- **AppSchedSLC_slc () :** cette fonction a pour rôle d'ajuster le niveau de service des tâches acceptées afin de modifier l'utilisation du processeur demandée dans le système ; cet ajustement se fait sur les tâches selon l'utilisation CPU demandée, donc le SLC change le niveau de service de ces tâches soit en le diminuant ou bien en l'augmentant.

La fonction est appelée par le régulateur PID lorsque il y'a une surcharge légère dans le système afin modifier l'utilisation CPU demandée.

2.2.1.5 Module AppSchedAC

Ce module décrit le fonctionnement du contrôleur d'admission via les fonctions ci-dessous :

- **AppSchedAC_task () :** cette fonction définit les nouvelles tâches entrantes dans le système.
- **AppschedAC_accepted_tasks () :** c'est ici où sont définies les tâches acceptées par le AC, en tenant en considération la condition sur l'utilisation CPU.
- **AppSchedAC_ac () :** cette fonction est appelée par le régulateur PID lorsque il y'a une surcharge légère dans le système, le AC doit donc ajuster le reste d'utilisation CPU que le contrôleur de niveau n'a pas pu modifier, et cela en supprimant les tâches les moins importantes dans la liste des tâches prêtes.

3 FC-EDF sous FreeRTOS

Dans ce qui suit nous présentons l'implémentation des modules définis ci-dessus dans FreeRTOS :

3.1 Implémentation du module AppSched

3.1.1 Implémentation de l'API

La FST est créé dans la fonction **AppSched_Init ()**, en appelant la fonction FreeRTOS **vTaskCreate ()** avec une priorité égale à FST_P. La FST est implémentée comme une boucle infinie, à l'intérieur de laquelle la politique d'ordonnancement est exécutée en appelant la fonction **AppSchedEDF_Sched ()**, cette tâche se bloque à la fin de chaque boucle, ensuite elle est débloquée dans l'un des événements présentés dans (2.2.)

AppSched_Create (): cette fonction permet de créer une tâche FreeRTOS avec la fonction xTaskCreate (), avec une priorité de FST_PT, pour mettre cette tâche à l'état suspendue pour éviter son insertion dans la liste prêt de FreeRTOS, on appelle la fonction vTaskSuspend ().

AppSched_WaitForNextPeriod (): cette fonction appelle la fonction FreeRTOS vTaskDelayUntil (), pour bloquer la tâche jusqu'à la prochaine période en utilisant les paramètres xi et Ti dans l'eTCB comme arguments.

3.1.2 Implémentation des fonctions privées

AppSched_Tick (): cette fonction est exécutée à chaque tick d'horloge, elle est implémentée via une fonction de rappel de FreeRTOS qui est vApplicationTickHook ().

Afin d'appeler le reste des fonctions, FST utilise une fonctionnalité FreeRTOS appelée Trace macros, cette dernière permet de implémenter des technique de profilage et de déblocage de code.

Cette fonctionnalité propose également des macros de développeur afin d'atteindre ses fonctionnalités, Trace macros est utilisée par la FST pour implémenter les fonctions suivantes :

AppSched_Ready (): Cette fonction est appelée quand une tâche est à l'état prêt, elle permet de lancer la FST en lui envoyant un signe, cette fonction est associé à la macro de trace **traceMOVED_TASK_TO_READY_STATE ().**

AppSched_Delay (), AppSched_Block () et AppSched_Suspend (): La FST envoie une notification, en utilisant la fonction **xTaskNotifyGive (),** pour que ces fonctions puissent débloquer la FST, si cette dernière est en cours d'exécution, aucune notification n'est envoyée.

AppSched_Suspend () : cette fonction est associée à la macro de trace **traceTASK_SUSPEND ()** appelée dans la fonction FreeRTOS **vTaskSuspend ()**.

AppSched_Delay () : cette fonction est associée à la macro de trace **traceTASK_DELAY_UNTIL (),** cette dernière est appelée dans la fonction **vTaskDelayUntil ().**

Ce mécanisme assure la synchronisation des tâches, en utilisant de simples notifications qui peuvent avoir une valeur, ces notifications servent à débloquer une tâche, la FST se bloque en essayant de prendre une notification, et elle est débloquée par une notification émise par les fonctions de macros de traces correspondantes.

3.2 Implémentation du module AppSchedEDF

Ce module implémente la nouvelle politique d'ordonnancement qui dans notre cas c'est EDF. Comme présenté dans le précédent chapitre EDF est un algorithme d'ordonnancement à priorité dynamique dans lequel les priorités les plus élevées sont attribuées aux tâches avec des échéances plus proches.

Le raisonnement dit de créer une nouvelle liste prête, où les tâches sont ordonnées selon leurs échéances, alors la tâche sur la tête de la liste est celle qui a l'échéance la plus proche (donc c'est la tâche la plus prioritaire qui sera exécuté en premier).

Concernant le reste de l'architecture et des structures FreeRTOS (liste d'attente et mécanisme d'horloge), ils sont maintenus avec de légers changements, pour ce faire le module utilise les trois fonctions suivantes :

AppSchedEDF_Init (): cette fonction a pour rôle d'initialiser toutes les listes des tâches à la création de la première tâche, en appelant la fonction **prvInitialiseTaskLists ()** FreeRTOS, cette dernière est modifiée en ajoutant l'initialisation de la liste **xReadyTasksListEDF ().**

AppSchedEDF_add () : cette fonction permet d'ajouter les tâches crées à la liste prête, en utilisant la fonction FreeRTOS suivante : **prvAddTaskToReadyList ().**

AppSchedEDF_Tick (): On sait que, lorsqu'une tâche passe à la liste des tâches prêtes, il est essentiel de connaître sa prochaine échéance pour pouvoir l'insérer à la bonne place, pour ce faire, à chaque tick d'horloge CPU la nouvelle échéance est calculée comme suit :

$$\boldsymbol{DeadlineTask = tickC + PeriodTask}$$

AppSchedEDF_Sched (): Notons que même la gestion de la tâche **IDLE** sera modifiée, cette dernière n'est qu'une simple tâche FreeRTOS inactive avec une priorité très basse, créée automatiquement au démarrage de l'ordonnanceur FreeRTOS pour garantir qu'il y a toujours au moins une tâche capable de s'exécuter.[3]

Mais l'algorithme EDF, prend en considération le comportement le moins prioritaire, ce dernier peut être simulé par une tâche dont l'échéance est la plus éloignée.

Pour surmonter cette difficulté, **AppSchedEDF_Sched ()** appelle la fonction FreeRTOS **vTaskStartScheduler ()** qui initialise la tâche **IDLE** et l'insère dans la liste des tâches prêtes, ainsi cette fonction active le mécanisme d'ordonnancement de la manière suivante :

Selon le guide de FreeRTOS, une variable de configuration **configUSE_EDF_SCHEDULER**, est ajouté au fichier de configuration FreeRTOS: h, tel que : lorsque **configUSE_EDF_SCHEDULER** est mis à 1, l'ordonnanceur EDF est utilisé et activé en appelant la fonction **vTaskStartScheduler ()**.

3.3 Implémentation du module AppSchedPID

AppSchedPID_def (): cette fonction permet de définir d'abord l'entrée du contrôleur PID, comme définit précédemment l'entrée de notre régulateur n'est que l'erreur qui s'agit de la consigne moins la sortie réelle qui sont dans notre cas : le taux de dépassement d'échéance désiré (MissRatio= 0) et le nombre de tâches qui ont manqué leurs échéances (le taux d'échéance raté) MissRatio () respectivement.

Donc la fonction **AppSchedPID_def ()** appelle la fonction **xTaskGetMissRatio (),** afin d'avoir la valeur MissRatio () et de calculer l'entrée du régulateur, puis elle fait appel à une autre fonction, **VoidPID_Control ()** qui définit l'équation mathématique du PID ou bien le signale de commande (ΔCPU).

AppSchedPID_CPU () : Le signal de commande calculé par le régulateur PID est la charge CPU noté ΔCPU (), afin de modifier l'utilisation CPU demandée le régulateur fait appel à SLC via la fonction **AppSchedPID_CPU (),** qui appel la fonction **xTaskCallSLC ()** si le SLC n'a pas pu compléter sa tâche alors, le AC est appelé par la tâche **xTaskCallAC ()**.

3.4 Implémentation du module AppSchedSLC

AppSchedSLC_select_task () : selon l'utilisation CPU demandée, cette fonction choisit parmi les configurations des tâches acceptées celle qui va assurer cette utilisation ; si l'utilisation exigée est positive (ΔCPU>0) donc une tâche améliorable est sélectionnée par la fonction **xSelect_enhanced_task ()** afin d'améliorer son niveau de service, si non (si ΔCPU<0) dans ce cas une tâche dégradable est sélectionnée via la fonction **xSelect_degradable_task ()** pour dégrader son niveau de service**.**

AppSchedSLC_slc () : Cette fonction assure le changement du niveau de service des tâches sélectionnées par **AppSchedSLC_select_task (),** afin de modifier l'utilisation processeur demandée.

Si ΔCPU>0 alors SLC va augmenter le niveau de service des tâches améliorable sélectionnées, sinon il va diminuer leur niveau de service, ce changement se fait par la fonction **xChange_service_level ().**

Une fois le niveau de service des tâches est modifié, l'utilisation CPU exigée est ajustée et la partie non ajustée est renvoyée par le SLC afin que le AC puisse l'adapter et tout ça par la fonction **xTaskSLC ()**.

Cette fonction est appelé par le module du PID via la fonction **xTaskCallSLC ().**

3.5 Implémentation du module AppSchedAC

AppSchedAC_task () : la fonction a pour rôle de définir les tâches entrantes, par la fonction FreeRTOS **xTaskDefine ()** qui décrit les caractéristiques de ses tâches comme leurs échéances, leurs paramètres d'importances ainsi que la demande CPU.

AppschedAC_accepted-tasks () : cette fonction définit les tâches acceptées selon la condition sur l'utilisation CPU; les tâches acceptées sont définies par **xTaskAccepted ()**.

AppSchedAC_ac () : Cette fonction assure d'ajuster la partie ΔCPU () qui n'est pas terminée par la fonction **xTaskAC ()**, cette dernière fait en sorte de supprimer de la liste prêtes les tâches qui ont moins d'importances afin de réduire la charge CPU en utilisant le paramètre d'importance des tâches définit dans la fonction **xTaskDefine ()**.

AppSchedAC_ac () est appelée par le module du PID via la fonction **xTaskCallAC ().**

4 Exemple d'application

Tâche 1 : Nommée "T01"avec un intervalle période de [10,15], échéance variable en fonction de la période égale à 10, durée d'exécution 5, priorité et importance de 1.

Tâche 2 : Nommée "T02"avec un intervalle période de [10,16], échéance variable en fonction de la période égale à 10, durée d'exécution 4, priorité et importance de 2.

Tâche 3 : Nommée "T03"avec un intervalle période de [11,17], échéance variable en fonction de la période égale à 11, durée d'exécution 1, priorité et importance de 3.

Tâche 4 : Nommée "T04"avec un intervalle période de [15,19], échéance variable en fonction de la période égale à 12, durée d'exécution 3, priorité et importance de 4.

Les paramètres de ces tâches se trouvent dans l'eTCB présenté précédemment, tel que la période de la tâche est sous forme d'intervalle sur lequel le contrôleur SLC joue afin d'ajuster le niveau de services. Ces informations sont accessibles via la tache FST comme définit dans le prochain code.

Afin d'utiliser la stratégie d'ordonnancement FC-EDF pour ordonnancer les quatre tâches l'utilisateur doit modifier le paramètre FST_SCHED dans le makefile qui autorise l'exécution par application de l'algorithme FC-EDF. Ainsi la consigne du régulateur PID est déclarée aussi dans ce makefile.

Un code générique est présenté ci-après démontrant d'abord comment nos taches sont déclarées, puis l'activation de l'ordonnanceur qu'on a choisi à employer, ce choix se fait en remplaçant le paramètre FST_SCHED par l'une de ces valeurs:

* `EDF`: ordonnancement par la stratégie EDF ;

* `RM`: ordonnancement par la stratégie RM ;

* `FC-EDF`: ordonnancement par la politique FC- EDF.

Les quatre tâches sont créées par l'intermédiaire de la fonction **"xSchedulerTaskCreate"** (présentée dans la section **2.1.2**) qui va prendre en compte à chaque appel, les attributs de la tâche à créer. Ensuite, pour simuler l'exécution des tâches nous avons choisi de faire de l'affichage comme code de ces fonctions (voir la fonction task_body).

L'affichage se fait vers un écran PC à l'aide du protocole UART.

Nous attendons comme résultats de l'exécution de cet exemple d'application ce qui suit :

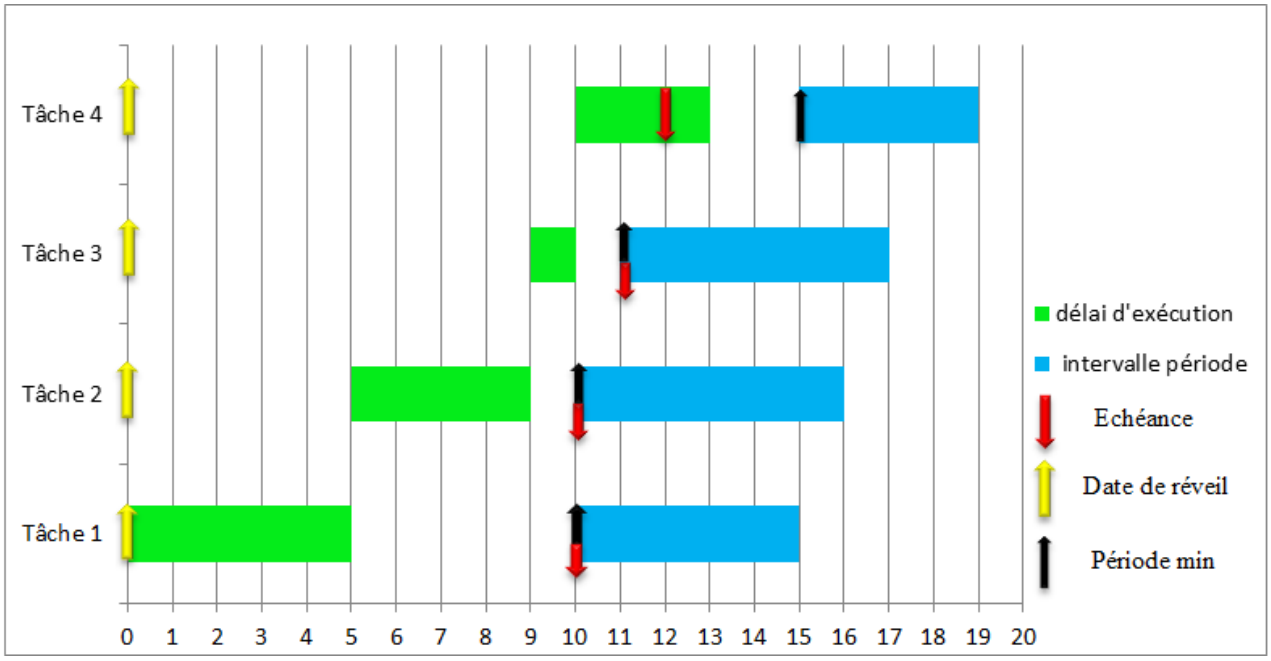
- L'utilisation CPU soit toujours proche des 100% avec quelques dépassements furtifs.

- La tache T04 est celle qui va être sacrifiée le plus souvent par l'AC vu sa moindre importance.

- La tâche T01 est celle qui va occuper la CPU le plus souvent et, par conséquence, FC-EDF va utiliser sa période Max (15) pour minimiser son impact sur la charge totale du CPU.

```
#include "FreeRTOS.h"
#include "task.h"
/* The extern "C" is required to avoid name mangling between C and C++ code. */
extern "C"
{
// FreeRTOS callback/hook functions
void vApplicationMallocFailedHook( void );
void vApplicationStackOverflowHook( TaskHandle_t pxTask, char *pcTaskName );

// FST callback/hook functions
void vSchedulerDeadlineMissHook( FstTCB_t * xTask, const TickType_t xTickCount );
void vSchedulerWcetOverrunHook( FstTCB_t * xTask, const TickType_t xTickCount );
void vSchedulerStartHook( void );
}

static void task_body( void* params );

/* Create a Serial port, connected to the USBTX/USBRX pins; they represent the
 * pins that route to the interface USB Serial port so you can communicate with
 * a host PC. */
static Serial pc( USBTX, USBRX );

int main() {
    /* Set the baud rate of the serial port. */
    pc.baud( 9600 );

    vSchedulerSetup();

    /* Create the application scheduled tasks. */
    xSchedulerTaskCreate( task_body, "T01", 256, NULL, 1, NULL,10, 15, 10, 5,1);
    xSchedulerTaskCreate( task_body, "T02", 256, NULL, 2, NULL,10, 16, 10, 4,2);
    xSchedulerTaskCreate( task_body, "T03", 256, NULL, 3, NULL,11, 17, 11, 1,3);
    xSchedulerTaskCreate( task_body, "T04", 256, NULL, 4, NULL,15, 19, 12, 3,4);

    /* Create and start the scheduler task. */
    vSchedulerInit();

    /* The execution should never reach here. */
    for (;;);

}

/* Periodic task body. */
static void task_body( void* params )
{
    // eTCB
    FstTCB_t *taskInfo = ( FstTCB_t * ) params;

    // A pointer to the task's name, standard NULL terminated C string.
    char *pcTaskName = pcTaskGetTaskName( NULL );*!


    for (;;)
    {
        vTaskSuspendAll();
        pc.printf( "%d\t%s\tS\t%d\t%d\n", xTaskGetTickCount(), pcTaskName, taskInfo->uxReleaseCount, taskInfo->xCur );
        xTaskResumeAll();

        vUtilsEatCpu( taskInfo->xWcet );

        vTaskSuspendAll();
        pc.printf( "%d\t%s\tE\t%d\t%d\n", xTaskGetTickCount(), pcTaskName, taskInfo->uxReleaseCount, taskInfo->xCur );
        xTaskResumeAll();

        vSchedulerWaitForNextPeriod();
    }

    /* If the tasks ever leaves the for loop, kill it. */
    vTaskDelete( NULL );
}
```

Le diagramme qui suit représente la première exécution des quatre tâches en utilisant uniquement l'ordonnanceur EDF (initialement).

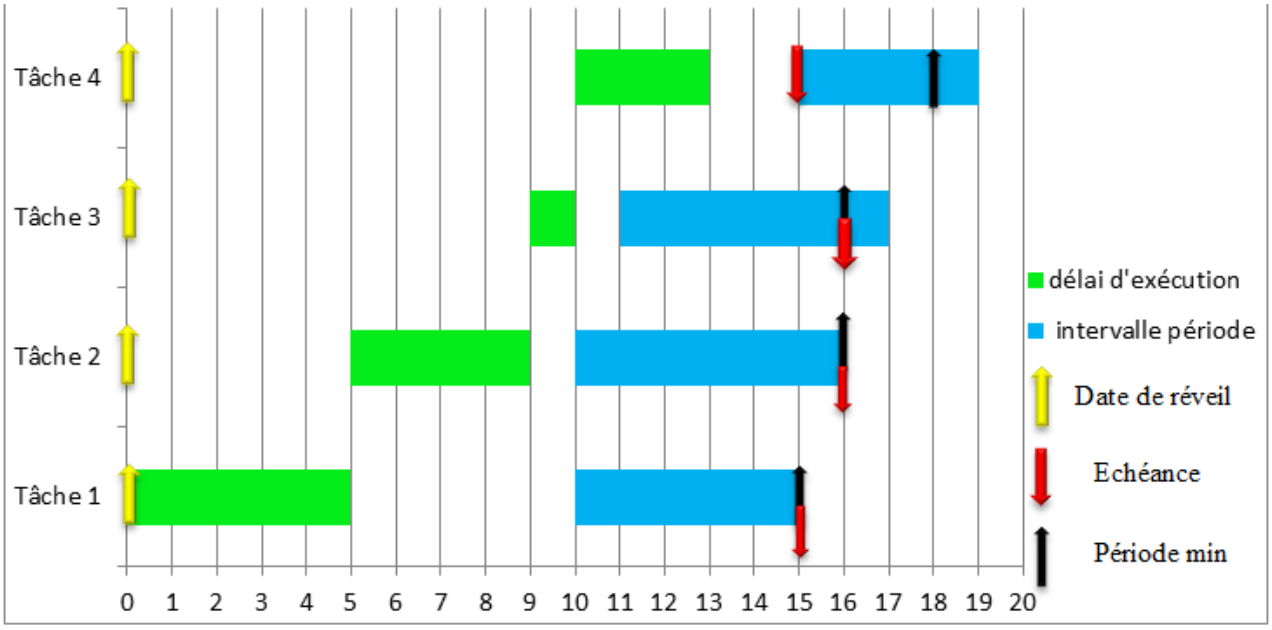


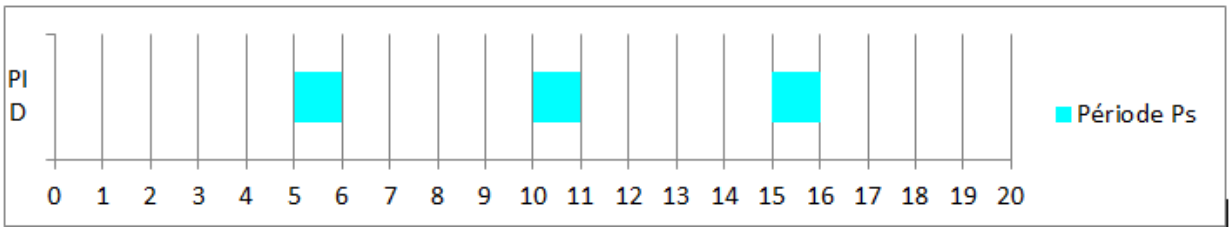
D'après le diagramme, la tache 4 a dépassé son échéance et aussi le système est en surcharge car :

$$\mathbf{U}=\sum_{i=1}^{n}\frac{Ci}{Pi}=5/10+4/10+1/11+3/15=1.19>1$$

Pendant chaque période P_s, la Miss Ratio est renvoyée au le régulateur PID à P_s=10 MissRatio=1/4 ;

Donc le régulateur PID fait appel au contrôleur SLC, qui va ajuster les périodes des tâches en les augmentant ce qui entraine l'augmentation de leurs échéances, les résultats sont présentés dans le diagramme suivant :





La période de la tâche 1 est augmentée jusqu'à 15 donc son échéance est 15, pour le reste des tâches elles auront les valeurs suivantes :

Tâche 2 : P= 16, E=16 ;

Tâche 3: P= 16, E=16 ;

Tâche 4: P= 18, E=15.

Finalement, le problème de surcharge est résolu ainsi que celui de dépassement d'échéance.

$$\mathbf{U}=\sum_{i=1}^{n}\frac{\mathbf{Ci}}{\mathbf{Pi}}\mathbf{= 5/15+4/16+1/16+3/18= 0.8125.}$$

Sachant que si on a obtenu la charge totale U trop faible c'est-à-dire le système est en sous-utilisation, dans ce cas la fonction SLC diminuera les périodes (si c'est possible) afin d'ajuster l'utilisation du système.

D’après cet exemple, les tâches exécutées selon EDF risquent de rater leurs échéances par contre en utilisant le FC-EDF les risques sont très faibles.

5 Conclusion

Dans ce chapitre, nous avons présenté en détail la méthode de conception de la stratégie FC-EDF sous FreeRTOS. Notre choix s’est porté sur un développement de cet algorithme au niveau utilisateur via la tâche FST. Ensuite, nous avons expliqué les différentes fonctions que constituent les différents modules de la FST ainsi que leur implémentation. Pour finir un exemple d’application a été proposé afin de montrer comment un utilisateur pourra utiliser FC-EDF pour construire une application temps-réel à base d'un ordonnanceur par rétroaction, efficace, obéissant à un critère de performance que l'utilisateur peut choisir selon les exigences de son application.

Conclusion générale

Conclusion générale

Ce mémoire avait pour ambition de développer une nouvelle version de FreeRTOS basée sur un algorithme d'ordonnancement régulé.

L'élaboration de ce travail a commencé par la description de FreeRTOS, ce qui nous a permis d'observer qu'il est fondé sur un algorithme d'ordonnancement basique (SPP). D'un autre coté l'étude de quelques stratégies d'ordonnancement classiques nous a dévoilés que même ces algorithmes souffrent de plusieurs inconvénients. Cependant, l'ordonnancement par rétroaction été la clé pour atténuer voir éliminer les défauts des algorithmes usuels.

L'approche du feedback scheduling est réalisable sous FreeRTOS par un algorithme spécifique qui est le FC-EDF, ce dernier est implémenté en tant que tâche pour éviter de nuire à la sécurité, la robustesse et la fiabilité de notre système.

La politique FC-EDF a prouvé son efficacité vis-à-vis du respect des contraintes temporelles, en outre, elle a permis l'amélioration de la qualité de commande du système, ce qui offre à la version FreeRTOS obtenue l'avantage de traiter divers aspects de procédés temps réel. De ce fait, la nouvelle version est beaucoup plus pertinente que l'originale.

Les résultats obtenus dans ce travail nous encouragent à envisager comme perspectives la réalisation d'une implémentation de l'ordonnanceur FC-EDF sous FreeRTOS sur une carte Raspberry et de faire un test sur une véritable application temps- réel.

Références bibliographiques

Références bibliographiques

[1] CDTA, “Center for Development of Advanced Technologies,” *historique*. http://en.rcdat.ir.

[2] M. Lardellier, “FREERTOS COURS ET EXERCICE SUR STM32 AVEC SW4STM32 ET STM32CUBE,” France, 2018.

[3] R. Barry, “WWW.FreeRTOS.org” 2003.

[4] E. Carraro, “Implementation and Test of EDF and LLREF Schedulers in FreeRTOS,” Université de Padoue, Italie, 2016.

[5] A. Abdelghani, B. Mohammed, O. Kermia, and Z. Guettafi, “introduction aux FreeRTOS et l’application sur La carte Raspberry PI,” Alger, 2019.

[6] H. Descoubes, G. Frey, and B. Dufay, “Systemes temps reel 2,” Caen, 2016.

[7] S. Bouzefrane, “Introduction aux systèmes temps réel,” Paris,France, 2011.

[8] S. Bouzefrane, “Ordonnancement centralisé tâches temps réel,” Paris,France, 2009.

[9] HAMID Rabah, “Ordonnancement Temps Réel des Tâches dans un Système Industriel par Intégration d’une loi de Régulation Automatique,” 2013.

[10] E. G. Francis Cottet, *Systèmes temps réel embarqués - 2e éd: Conception et implémentation*. 2005.

[11] Z. Mammeri, “Ordonnancement temps réel,” TOULOUSE, 2000.

[12] O. S. et D. R. D. Simon, “‘Conception conjointe commande/ordonnancement et ordonnancement régulé,’” in *Systèmes Temps réel 2, ordonnancement, réseaux et qualité de service*, Nancy, 2005.

[13] J. A. Stankovic, “Feedback Control Real-Time Scheduling : Framework , Modeling , and,” 1905.

[14] A. Cervin, “Improved scheduling of control tasks,” *Proc. - Euromicro Conf. Real-Time Syst.*, pp. 4–10, 1999, doi: 10.1109/EMRTS.1999.777444.

[15] J. Eker, P. Hagander, and K. E. Årzén, “Feedback scheduler for real-time controller tasks,” *Control Eng. Pract.*, vol. 8, no. 12, pp. 1369–1378, 2000, doi: 10.1016/S0967-0661(00)00086-1.

[16] A. Cervin and J. Eker, “Feedback scheduling of control tasks,” *Proc. IEEE Conf. Decis. Control*, vol. 5, pp. 4871–4876, 2000, doi: 10.1109/cdc.2001.914702.

[17] A. Cervin and J. Eker, “Control-scheduling codesign of real-time systems: The control server approach,” *J. Embed. Comput.*, vol. 1, no. 2, pp. 209–224, 2005.

[18] S. Xia, Shen, L.P., Liu, Z., Wang, Y.X., “Fuzzy logic based feedback scheduler for embedded control systems,” in *Fuzzy logic based feedback scheduler for embedded control systems*, 2005.

[19] F. Xia, Y. Sun, and Y. C. Tian, “Feedback scheduling of priority-driven control networks,” *Comput. Stand. Interfaces*, vol. 31, no. 3, pp. 539–547, 2006, doi:

10.1016/j.csi.2008.03.020.

[20] F. Xia and Y. Sun., “Control and Scheduling Codesign,” *Springer Zheijiang Univ. Press*, 2008.

[21] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao, “The Case for Feedback Control Real-Time Scheduling,” *Univ. VirginiaCharlottesville, VAUnited States*, p. 10, 1998.

[22] F. E. Paez, J. M. Urriza, R. Cayssials, and J. D. Orozco, “FreeRTOS user mode scheduler for mixed critical systems,” *2015 6th Argentine Conf. Embed. Syst. CASE 2015*, pp. 37–42, 2015, doi: 10.1109/SASE-CASE.2015.7295845.