Ecole Centrale de Nantes *Université de Nantes* *Ecole des Mines de Nantes*

MASTER AUTOMATIQUE ROBOTIQUE ET INFORMATIQUE APPLIQUEE

SPECIALITE : TEMPS REEL CONDUITE ET SUPERVISION

Année 2015 / 2016

Thèse de Master

Présenté et soutenu par :

M^{lle} Khaoula BOUKIR

Le

15/09/2016

à

L'Ecole Centrale de Nantes

TITRE

Mise en œuvre d'un ordonnanceur global dans l'OS temps réel Trampoline.

JURY

Président :	**Olivier-Henri ROUX**	**Professeur de l'Université**
Examinateurs :	**Sébastien FAUCOU**	**Maitre de conférences**
Directeur de thèse :	**Jean-Luc BECHENNEC**	**Chargé de recherche**
	Anne-Marie DEPLANCHE	**Maitre de conférences**

Laboratoire : Institut de Recherche en Communication et Cybernétique de Nantes
- **UMR CNRS 6597**

Remerciements

Je profite de l'occasion qui m'est donnée pour remercier les personnes qui ont eu un rôle durant l'élaboration de cette thèse.

Mes premiers remerciements vont pour mes deux encadrants Jean-Luc Béchennec et Anne-Marie Déplanche qui m'ont encadré sérieusement, su me guider et m'ont consacré le temps nécessaire durant cette période de stage, tout en travaillant dans la bonne humeur et dans un cadre amicale. Ce fut donc pour moi une période très agréable et j'en garderai un bon souvenir.

Un grand merci aux membres de jury, Mr Olivier-Henri ROUX et Mr Sébastien FAUCOU, pour avoir accepté d'évaluer mes travaux. Je suis consciente que la lecture approfondie de ce document nécessite un investissement en temps important.

Pour terminer, je remercie les membres de l'équipe « Systèmes Temps Réel » de l'IRCCyN pour m'avoir accueilli. Enfin, je remercie toute personne ayant aidé, de près ou de loin, à l'achèvement de ce travail.

Merci a Tous

Liste des Figures

Liste des Tableaux

Table des Matières

INTRODUCTION GÉNÉRALE

Présentation du thème de projet

Orientation du projet et motivations

Un système temps réel est un ensemble de programmes applicatifs qui se distingue par son aptitude à contrôler un environnement par nature dynamique, qu'on appelle procédé. La dénomination temps réel provient du fait que le système doit s'adapter et réagir à la vitesse de l'évolution du procédé contrôlé, et donc fournir des résultats exacts dans un intervalle de temps fixé. De ce fait, les systèmes temps réel sont confrontés à des contraintes temporelles dont le respect est considérablement important.

Gillies[1] en 1995 a défini un système temps réel comme étant « un système dans lequel l'exactitude des applications ne dépend pas seulement de l'exactitude du résultat, mais aussi du temps auquel ce résultat est produit ». L'objectif reste de suivre exactement la vitesse d'évolution des environnements contrôlés et le respect des échéances temporelles.

A cet effet, le système temps réel utilise certains programmes applicatifs qu'il faut exécuter de manière récurrente, sur une plateforme contenant un ensemble limité d'unités de traitement[2] partagées entres eux. Chacun de ses programmes possède une date d'échéance.

Les exigences temporelles sur les applications temps réel mènent à une différenciation entre des systèmes temps réel durs et d'autres dits mous. Un système temps réel est dit dur quand la non-satisfaction d'une contrainte temporelle peut être néfaste et engendre des désastres économiques, écologiques ou humaines. Inversement, un système temps réel mou se distingue par la tolérance de dépassement des échéances temporelles sous certaines conditions. Le non-respect des contraintes temporelles ne cause pas de dégâts. Néanmoins, le temps de réponse ne doit pas être infini.

Le pilotage d'un système temps réel se fait, le plus souvent, via des systèmes d'exploitation dits systèmes d'exploitation temps réel (RTOS). L'architecture de ces systèmes d'exploitation doit être choisie de telle manière à ce qu'elle réponde aux exigences et complexité croissantes des applications temps réel, en termes de de rapidité de traitement. Afin de garantir les performances des systèmes temps réel, les RTOS possède une entité qui se charge d'ordonnancer l'exécution des programmes du système et l'accès aux ressources de calcul tout en respectant les contraintes temporelles. Il s'agit de l'ordonnanceur. **[1]**

Accroitre la performance et la rapidité de traitement d'un système temps réel dépend également de l'architecture matérielle sur laquelle il est implémenté. Avec le progrès technologique repoussant, il y a eu une tendance croissante vers l'utilisation des plateformes constituées de plusieurs cœurs. Une solution qui a apporté des architectures matérielles incontournables et toujours plus puissantes en termes de capacité de calcul. Cette solution a engendré un nombre accru d'études

[1] Donald Gillies : Ancien professeur à l'Université de la Colombie Britannique au Canada, actuellement il occupe le poste d'Ingénieur Logiciel Sénior à Google.

[2] Unités de traitement : sont les ressources que les applications requièrent pour leur exécution, e.g. ressources CPU, ressources de communication, accès aux données …

scientifiques en matière d'ordonnancement, et plusieurs politiques d'ordonnancement temps réel multiprocesseurs ont été proposées.

Afin d'obtenir un ordonnancement correct des tâches d'un système temps réel, et qui garantisse le respect de toutes ses contraintes temporelles, le choix de la politique d'ordonnancement ne doit pas être arbitraire. En effet, ce choix joue un facteur déterminant sur le coût et la performance globale du système, et doit se baser à la fois sur l'optimalité et sur l'adaptabilité d'implémentation de la politique sur la plateforme choisie (RTOS).

C'est dans cette optique que notre travail de recherche s'inscrit. Il s'agit de la suite d'une première étude bibliographique faite. Une étude qui s'est focalisée sur une politique d'ordonnancement multiprocesseur, récemment publiée par Geoffrey Nelissen dans le cadre de sa thèse **[2]**, et dont l'optimalité théorique a été prouvée. Il s'agit de la politique U-EDF (Unfair Earliest Deadline First).

Ainsi, dans le cadre de ce présent travail, nous allons étudier l'*implémentabilité* de cette politique au sein d'un système d'exploitation temps réel. Il s'agit de Trampoline, un RTOS développé dans l'équipe *Systèmes Temps Réel* de l'IRCCyN. Considéré comme une plateforme de mise en œuvre de prototypes pour un usage principalement dédié à l'enseignement et à la recherche.

Contexte réel

Le choix de la politique d'ordonnancement temps réel, pour l'implémenter au sein d'un RTOS, doit prendre en considération plusieurs facteurs. L'optimalité reste aussi un facteur important. L'étude de ses politiques est un domaine très actif dans la recherche. Ainsi, plusieurs études théoriques ont démontré que la mise en œuvre d'un algorithme d'ordonnancement dynamique (tel que U-EDF), permet une exploitation plus efficace des ressources processeur en plus d'une meilleur gestion en cas de surcharge.

Dans le marché de l'informatique embarquée, de nombreux standards de contrôle logiciel ont vu le jour pour apporter des solutions aux exigences et complexité croissantes des architectures électroniques embarquées. Les standards introduits tels que AUTOSAR[3] pour l'automobile ou la DO178[4] pour l'avionique, ont permis d'augmenter la portabilité du logiciel et maitriser les défaillances.

Toutefois, ces standards ne prévoient pas l'utilisation des politiques d'ordonnancement dynamiques, pour des raisons de complexité de calcul et d'implémentation, et des fois par ignorance de leur existence. Ainsi, avec le grand décalage entre les avancées des recherches et solutions académiques dans le domaine d'une part, et les solutions anciennes utilisées toujours dans le marché d'autre part, il s'avère très intéressant de faire une étude sur la possibilité de diminuer ce décalage et voir s'il sera bénéfique d'implémenter ce que des études poussées en matière d'ordonnancement ont trouvé.

[3] AUTOSAR (AUTomotive Open System Architecture) : c'est un partenariat de développement mondial de l'industrie automobile. Il a pour but de développer et d'établir une architecture logicielle standardisée et ouverte pour les unités de contrôle électronique (ECU) des véhicules.

[4] DO178 : c'est une norme qui fixe les conditions de sécurité applicables aux logiciels critiques de l'avionique. Elle précise les contraintes de développement liées à l'obtention de la certification d'un logiciel d'avionique.

Objectifs

La contribution visée par la présente étude peut être scindée en deux objectifs principaux :

- **Une implémentation pour U-EDF :** l'objectif premier de ce travail de recherche est l'étude et l'évaluation de la possibilité de mise en œuvre de la politique d'ordonnancement global multiprocesseur U-EDF, au sein du système d'exploitation temps réel Trampoline. Sa finalité consiste à relever les principales problématiques relatives à l'implémentation, tout en faisant apparaitre l'ensemble des modifications qu'il faut apporter pour garantir le fonctionnement prévu de l'algorithme sur une plateforme réelle.

- **Validation de l'implémentation :** afin de répondre au premier objectif, il est indispensable de proposer un moyen pour mettre en œuvre la politique U-EDF dans le cadre de validation de l'implémentation. C'est pour cela que nous allons également mettre en place un outil de simulation qui servira pour le test et la validation de l'implémentation proposée pour U-EDF.

Organisation du travail

La suite de cette étude est organisée comme suit : dans un premier chapitre nous dressons les concepts de base relatifs à la problématique de l'ordonnancement temps réel. Etant donné que le présent travail a pour objectif la mise en œuvre d'une politique d'ordonnancement globale, nous allons faire un panorama sur l'ensemble des politiques de l'approche globale, puis nous nous focalisons sur la politique U-EDF, qui fait l'objet de ce présent travail.

Le 2ème chapitre présente une étude détaillée de la plateforme sur laquelle notre implémentation sera basée. Ainsi, nous allons identifier l'architecture de cette plateforme avec les différents services qui la constitue. Nous allons également présenter la solution d'ordonnancement temps réel qui y est implémentée actuellement.

Le dernier chapitre est consacré à la mise en œuvre de la politique U-EDF dans Trampoline. Pour cela nous allons d'abord présenter les étapes que nous avons suivies pour atteindre notre objectif, puis nous allons présenter en détail notre implémentation.

Finalement, nous allons dresser les conclusions et perspectives de notre travail.

CHAPITRE.1. THEORIE DE L'ORDONNANCEMENT DES SYSTEMES TEMPS REEL

Sommaire

Ce chapitre propose un panorama des techniques d'ordonnancement temps réel globales, en présentant les caractéristiques essentielles des politiques qui nous semblent majeures. Afin de pouvoir présenter dans un cadre unifié les différentes approches pour l'ordonnancement global, ce chapitre débute par une présentation des modèles de tâches et de quelques définitions importantes.

Etant donné que la présente étude se focalise sur l'implémentation de la politique U-EDF dans un RTOS, il nous est apparu nécessaire de fournir une vue d'ensemble sur le principe de fonctionnement de cet algorithme, ce qui sera présenté à la fin de chapitre.

1.1. MODELISATION ET VOCABULAIRE

TO-REDO : Dans cette section seront fournies la terminologie et les notations utilisées pour la modélisation des systèmes temps réel. Des définitions employées dans la théorie de l'ordonnancement temps réel seront également présentées. **[3]**

1.3.1. MODELISATION DES APPLICATIONS TEMPS REEL

En matière d'ordonnancement temps réel, une application temps réel est modélisée par une série de tâches (*tasks*) dont chacune contrôle un flot d'exécution d'un programme. Ce programme s'exécute sur un ou plusieurs processeurs, selon l'architecture matérielle de la plateforme d'implémentation du système temps réel.

Ainsi, une application temps réel τ se compose de N tâches qu'on note : $\boldsymbol{\tau} = \{\boldsymbol{\tau_1}, \dots, \boldsymbol{\tau_N}\}$. Chaque tâche $\boldsymbol{\tau_i}$ $\boldsymbol{i \in \{1, \dots N\}}$, si elle est récurrente, donne lieu à un ensemble infini de travaux (*jobs*). Un travail est une instance de l'exécution de la tâche, les travaux sont notés : $\{\boldsymbol{\tau_{i,1}}, \boldsymbol{\tau_{i,2}}, \dots\}$, tel que $\boldsymbol{\tau_{i,j}}$ correspond au j-ème travail de la tâche $\boldsymbol{\tau_i}$. Chaque tache a une date d'activation (ou de réveil) $\boldsymbol{A_i}$ et $\boldsymbol{a_{i,j}}$ est la date de réveil de sa j-ème instance.

L'activation d'une tâche engendre la création d'un travail dans l'état actif. La manière dont les travaux sont générés par une tâche permet de faire la distinction entre trois natures de tâches :

- ***Tâches périodiques*** : lorsque la durée séparant les dates de réveil des travaux successifs de la tâche est constante et égale exactement à sa période, alors on dit que c'est une tâche périodique.
- ***Tâches sporadiques*** : elles sont similaires aux tâches périodiques à l'exception faite que T_i correspond à la durée minimale qui sépare deux réveils de travaux successifs.
- ***Tâches apériodiques :*** elles sont utilisées en général pour les alarmes et les états d'exception, ainsi l'instant de réveil n'est pas connu au départ et il n'y a donc pas de période.

On peut aussi distinguer entre les tâches selon la relation entre leurs périodes et leurs dates d'échéance :

- Tâche à échéance implicite ou sur requête si $D_i = T_i$
- Tâche à échéance contrainte si $D_i < T_i$
- Tâche à échéance arbitraire si $D_i > T_i$

Selon le modèle canonique de Liu[5] et Layland[6] **[4]**, chaque tâche périodique τi est modélisée par le quadruplet $\{\boldsymbol{r_i}, \boldsymbol{C_i}, \boldsymbol{D_i}, \boldsymbol{T_i}\}$ (voir figure 1-1) où :

- $\boldsymbol{r_i}$ correspond à la date réveil de la tâche, il s'agit de la date d'activation du premier travail de la tâche;
- $\boldsymbol{C_i}$ correspond à la pire durée d'exécution de la tâche (*Worst-Case Execution Time, WCET*) ;
- $\boldsymbol{D_i}$ correspond à l'échéance relative ou délai critique de la tâche et $\boldsymbol{d_{i,j}}$ correspond à l'échéance absolue du j-ème travail.
- $\boldsymbol{T_i}$ correspond à la période d'activation de la tâche.

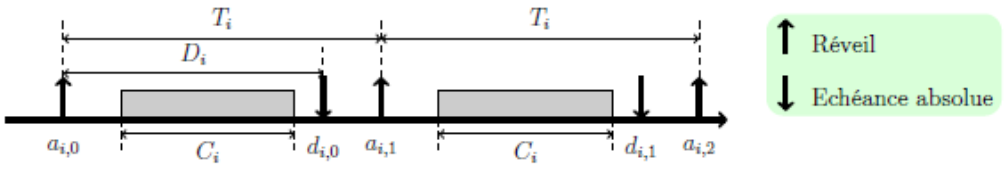


Figure 1-1 : modèle canonique d'une tâche périodique

On peut aussi définir la laxité d'un travail de la tâche $\boldsymbol{\tau_{i,j}}$ par la marge temporelle de liberté qu'il lui reste avant d'atteindre l'échéance. Si à l'instant t il reste $\boldsymbol{C_{i,j}(t)}$ à $\boldsymbol{\tau_{i,j}}$ pour s'exécuter, alors la laxité sera égale à $\boldsymbol{L_{i,j}(t) = d_{i,j} - t - C_{i,j}(t)}$. Une fois que la laxité d'un job s'annule, ce dernier doit s'exécuter immédiatement par crainte de dépasser son échéance.

La tâche se caractérise également par son *taux d'utilisation* $\boldsymbol{U_i = C_i/T_i}$, qui n'est rien que la proportion de la capacité de traitement qu'elle consomme sur un processeur pour son exécution. On définit aussi *le taux d'utilisation d'un ensemble de tâches* : $\boldsymbol{U_{sum} = \sum_{i=1}^{N} U_i}$, et *le taux d'utilisation maximum* du système $\boldsymbol{U_{max} = max\{U_1, \dots, U_N\}}$. Une tâche dont le taux d'utilisation est grand sera qualifiée de tâche *lourde*, et au contraire une tâche dont le taux d'utilisation est faible sera qualifiée de tâche *légère*. Le seuil à partir duquel une tâche est dite lourde ou légère dépend du contexte (généralement 0.5).

Etats d'une tâche :

Avant sa création, une tâche est considérée ***inexistante***, une fois créée elle se met dans l'état ***non opérationnel***. Quand elle se réveille, elle passe en état ***prêt***. Dans cet état l'instance de la tâche est activée mais en attente d'être sélectionnée par l'ordonnanceur pour l'exécution. Lorsque l'ordonnanceur le décide, suivant la politique d'ordonnancement choisie, la tâche se verra allouer le processeur afin d'être exécutée, ce qui correspond à l'état ***en cours d'exécution***. En cet état, la tâche peut : (i) être interrompue par une autre tâche plus prioritaire et passe en état ***prêt*** ; (ii) se mettre ***en attente*** d'un message, d'une date, d'un évènement, ou bien de l'accès à une ressource ; (iii) être suspendue et passer en état ***non opérationnel***. Une tâche en état d'attente d'une condition devient prête pour l'exécution une fois que cette condition soit vérifiée (voir figure 1-2). **[5]**

[5] C. L. Liu : Enseignant chercheur à l'Université d'Illinois aux Etats Unis.
[6] James W. Layland : Enseignant chercheur à l'Institut de Technologie de Californie.

Mode d'exécution des tâches :

On distingue 3 modes d'exécution des tâches en ordonnancement temps réel :

- Préemptif : une tâche peut être préemptée par des tâches plus prioritaires, et son exécution sera reprise ultérieurement ;
- Non-préemptif : une fois que la tâche a commencé son exécution, l'ordonnanceur ne peut pas l'interrompre en faveur d'une autre tâche plus prioritaire ;

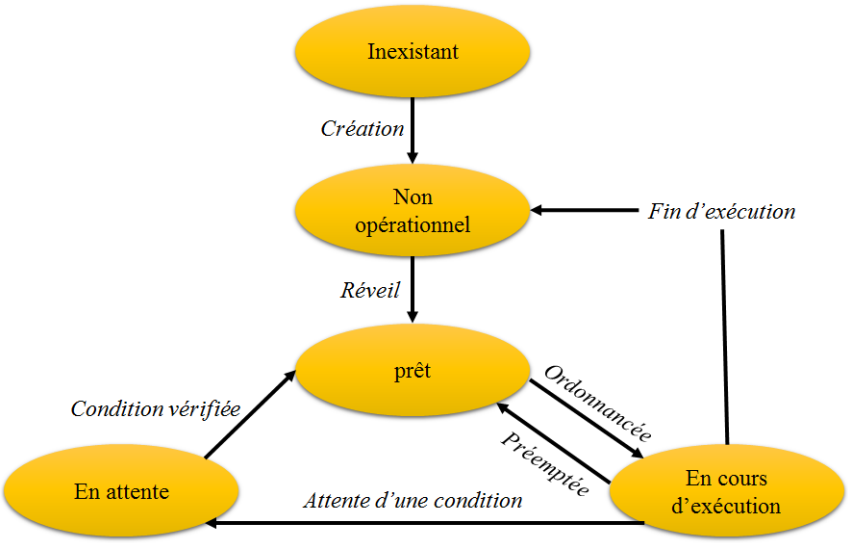


Figure 1-2 : les états de vie d'une tâche

- Coopératif : l'exécution de la tâche se compose de sections préemptible, dans lesquelles on peut la préempter, et d'autres sections non préemtible, où on ne peut pas la préempter.

1.3.2. MODELISATION DE L'ARCHITECTURE MATERIELLE

Un système temps réel est composé d'une application (ensemble de tâches) et d'une architecture matérielle. Cette architecture matérielle désigne les ressources physiques nécessaires pour l'exécution : en fonction du nombre de processeurs et de l'utilisation éventuelle d'un réseau, on distingue trois grandes catégories d'architecture :

- *Architecture monoprocesseur :* dans ce type de systèmes, toutes les fonctions (tâches) de l'application sont exécutées par un unique processeur partagé.
- *Architecture multiprocesseur :* les fonctions de l'application sont réparties sur *m* processeurs partageant une mémoire centrale commune.
- *Architecture distribuée :* les fonctions de l'application sont réparties sur *m* processeurs, mais contrairement aux architectures multiprocesseurs, il n'y a pas de mémoire centrale partagée. Les processeurs sont reliés les uns aux autres par l'intermédiaire d'un réseau.

1.3.3. VUE D'ENSEMBLE SUR L'ORDONNANCEMENT TEMPS REEL

Principe d'ordonnancement temps réel:

Le principe de l'ordonnancement d'un système temps réel revient à définir dans quel ordre et pour quelle durée il faut exécuter les tâches du système sur chaque processeur de la plateforme. Ainsi, deux catégories d'ordonnancement peuvent se définir, selon l'architecture de plateforme sur laquelle les tâches sont exécutées :

- *Ordonnancement monoprocesseur :* dans ce cas l'exécution des tâches se fait sur un seul processeur, le problème d'ordonnancement est d'une seule dimension et revient à déterminer une organisation temporelle d'exécution des tâches sur le processeur (choisir quelle tâche il faut exécuter à chaque instant).
- *Ordonnancement multiprocesseur :* la plateforme d'exécution est une architecture multiprocesseur. Ainsi, l'ordonnanceur doit assurer une organisation spatiale (sur quel processeur on exécute chaque tâche) et une organisation temporelle sur chaque processeur. De ce fait, le problème d'ordonnancement à deux dimensions.

  Dans ce type d'ordonnancement, les tâches sont peuvent être amenées à migrer entre processeurs, c'est-à-dire commencer à s'exécuter sur un processeur puis être interrompues à un certain moment pour terminer l'exécution sur un autre processeur.

La politique de choix appliquée par l'ordonnanceur utilise des algorithmes d'ordonnancement. On distingue des algorithmes ***hors ligne*** qui construisent une allocation des travaux avant le démarrage du système, cette allocation sera répétée indéfiniment, et des algorithmes ***en ligne***, pour lesquels l'ordonnancement des tâches se décide au fur et à mesure que ces dernières s'exécutent.

Caractéristiques des algorithmes d'ordonnancement :

Faisabilité : on dit qu'une tâche est faisable, s'il existe un algorithme A qui permet d'ordonnancer toutes les possibilités de séquencement des travaux de cette tâche sans qu'ils dépassent leurs échéances.

Une condition nécessaire et suffisante pour qu'un système de tâches périodiques à échéances implicites soit faisable est : $\boldsymbol{U_{sum} \leq m}$ et $\boldsymbol{U_{max} \leq 1}$, où m est le nombre de processeurs qui constituent la plateforme du système.

Ordonnançabilité : un système de tâches est dit ordonnançable par un algorithme A, si A construit une séquence telle que toutes les tâches du système respectent leurs contraintes temporelles.

Prévisibilité : on dit qu'un algorithme d'ordonnancement A est prévisible, si le temps de réponse des travaux ne peut pas augmenter avec la diminution de leur temps d'exécution.

Durabilité/Viabilité : on dit qu'un algorithme d'ordonnancement A est durable (viable) pour un modèle de tâches, si et seulement si, un ensemble de tâches compatible avec le modèle et ordonnançable avec A, garde son ordonnançabilité même avec la diminution des temps d'exécution, l'augmentation des périodes ou le rapprochement des dates d'échéance.

Optimalité : Un algorithme d'ordonnancement A est optimal pour une classe de systèmes S et parmi une classe de politiques d'ordonnancement C si et seulement si tout système de S ordonnançable par une politique appartement à C, l'est aussi avec l'algorithme A.

Comparaison de deux algorithmes d'ordonnancement A et B :

- A est dominant sur B : si tous les systèmes de tâches ordonnançables par B le sont aussi par A, et il existe au moins un système de tâches qui est ordonnançable par A et non ordonnançable par B.
- A et B sont équivalents : si les systèmes de tâches ordonnançables par B le sont aussi par A et vice versa.
- Les deux algorithmes sont incomparables : s'il existe des systèmes de tâches ordonnançables par A et non ordonnançables par B et vice versa.

Classification des algorithmes d'ordonnancement par priorité

Les algorithmes d'ordonnancement relèvent du principe d'exécuter les tâches selon la priorité dont elles font état. Ainsi, l'ordonnanceur doit affecter des priorités d'exécution aux tâches. Ces priorités peuvent être :

- **Fixes pour les tâches (FTP : Fixed Task Priority) :** c'est-à-dire que tous les travaux d'une même tâche ont la même priorité.
- **Fixes pour les travaux (FJP : Fixed Job Priority)** : ce qui signifie que la priorité d'un travail est fixé au début de sa création, mais deux travaux d'une même tâche peuvent avoir des priorités différentes.
- **Dynamiques : (DJP : Dynamique Job Priority)** : c'est-à-dire que la priorité d'un travail peut évoluer pendant son cycle de vie.

1.2. POLITIQUES D'ORDONNANCEMENT TEMPS REEL MULTIPROCESSEUR

Cette section présentera les différentes politiques d'ordonnancement multiprocesseurs notamment les politiques globales. A la fin de la section nous allons mettre l'accent sur des approches en temps continu et en temps discret, dont l'optimalité a été démontrée.

1.2.1. CLASSEMENT DES POLITIQUES D'ORDONNANCEMENT MULTIPROCESSEUR

L'ordonnancement multiprocesseur s'attache à résoudre le problème de *l'allocation spatiale*, c'est-à-dire de choisir, pour chaque tâche, le processeur sur lequel elle sera exécutée. Puis déterminer l'organisation temporelle des tâches au sein de chaque processeur. Les solutions de ces problèmes s'articulent autour de 3 catégories **[6]**:

a) Ordonnancement par partitionnement

Dans ce type d'ordonnancement, on partitionne un ensemble de tâches à exécuter en m sous-ensembles, le nombre m correspond au nombre de processeurs, et donc chaque sous-ensemble sera affecté à un processeur. A cet effet, aucune migration de tâche n'est autorisée.

A l'intérieur de chaque processeur, on procède à un ordonnancement monoprocesseur entre les tâches.

Le principe d'affectation des tâches aux processeurs est similaire au principe du *BinPacking* qui consiste à ranger un nombre d'éléments (tâches) caractérisés par leur taille dans un nombre déterminé de sacs ou boites (processeurs). La capacité des processeurs est déterminée par des conditions suffisantes d'ordonnançabilité monoprocesseur. L'affectation des tâches est alors ramenée à une solution qui utilise un nombre de sacs inférieur ou égal au nombre de processeurs. Pour ce faire, des heuristiques permettent de trier les tâches selon un certain critère et puis de les affecter au premier processeur qui satisfait les conditions d'ordonnançabilité de la tâche. Les heuristiques les plus utilisées sont :

- **First Fit (FF) :** la tâche est affectée au premier processeur libre.
- **Next Fit (NF) :** la tâche s'exécute sur le processeur qui suit celui dans lequel la tâche précédente a été placée.
- **Best Fit (BF) :** la tâche s'affecte au processeur le plus rempli pouvant la contenir.
- **Worst Fit (WF) :** la tâche s'affecte au processeur le moins rempli pouvant la contenir.

b) Ordonnancement global

Le principe pour ce type d'ordonnancement est de n'appliquer qu'un seul ordonnancement pour tout le système. Ainsi, une liste des tâches prêtes est organisée en file d'attente. L'ordonnanceur décide, selon une règle de priorité, l'attribution des m travaux qui ont la plus forte priorité aux m processeurs libres. Ces travaux peuvent migrer d'un processeur à un autre au cours de leur exécution.

c) Ordonnancement hybride

Pour ce type d'ordonnancement, deux types d'approches sont possibles :

- **Semi partitionné :** le principe est de partitionner au maximum les tâches à exécuter jusqu'à ce que le système ne soit plus partitionnable. A ce moment-là on autorise le reste à migrer entre processeurs. Deux types de migration sont possibles ; *restricted migration* : dans le cas où la migration n'est possible qu'entre les travaux, et *portioned scheduling* : dans lequel on autorise qu'un travail migre vers un autre processeur pendant son exécution.
- **Clustering :** dans cette catégorie, on ne limite plus les tâches migrantes, mais plutôt les processeurs sur lesquels les tâches peuvent migrer. Le principe est de regrouper les processeurs en clusters, diviser les tâches entre les clusters, et finalement appliquer l'ordonnancement global au sein de chaque cluster.

  Deux types de clustering peuvent être appliqués ; le clustering physique où on associe les processeurs au cluster de façon définitive selon leur architecture matérielle, et le clustering virtuel permet de changer dynamiquement l'association des processeurs aux clusters.

1.2.2. VUE D'ENSEMBLE SUR LES POLITIQUES D'ORDONNANCEMENT GLOBALES

Comme nous l'avons présenté dans la sous-section précédente, le principe d'ordonnancement global consiste à choisir d'exécuter, parmi une liste unique de tâches prêtes, les *m* tâches les plus prioritaires. Afin de déterminer les règles d'affectation des priorités pour guider l'ordonnancement multiprocesseur, une simple généralisation des algorithmes d'ordonnancement monoprocesseur peut

être appliquée. Cependant, cette généralisation n'optimise pas toujours l'utilisation des processeurs. C'est pourquoi d'autres solutions ont été proposées et dans cette partie nous allons en survoler quelques-unes. **[7]**

a) Algorithmes RM-US[ξ] et EDF-US[ξ]

RM-US[$\boldsymbol{\xi}$] et EDF-US[$\boldsymbol{\xi}$] sont respectivement deux extensions pour les algorithmes RM[7] (Rate Monotonic) et EDF[8] (Earliest Deadline First). Le choix d'attribution des priorités aux tâches est décidé en fonction de leurs taux d'utilisation et d'un seuil $\boldsymbol{\xi}$:

- Si $\boldsymbol{u_i > \xi}$, alors $\boldsymbol{\tau_i}$ est considérée comme une *tâche lourde* et donc sa priorité devient maximale. Les conflits entre les tâches ayant la priorité maximale sont résolus d'une manière arbitraire.

- Si $\boldsymbol{u_i \leq \xi}$, alors la tâche $\boldsymbol{\tau_i}$ est dite légère et dans ce cas sa priorité est évaluée selon le principe de RM pour RM-US[$\boldsymbol{\xi}$] ou selon EDF pour EDF-US[$\boldsymbol{\xi}$].

b) Algorithme EDF$^{(k)}$

Il s'agit d'une politique à priorité fixe pour les travaux, elle est similaire à EDF-US[$\boldsymbol{\xi}$]. L'affectation des priorités aux tâches est toujours en fonction de leurs taux d'utilisation de la manière présentée ci-dessous :

- Les k-1 tâches qui ont les plus grands $\boldsymbol{u_i}$ reçoivent la plus grande priorité, avec une résolution arbitraire des conflits.

- Les priorités des tâches restantes sont affectées selon le principe d'EDF.

c) Algorithmes à laxité nulle

Il s'agit des politiques EDZL (Earliest Deadline Zero Laxity), RMZL (Rate Monotonic Zero Laxity) et FPZL (Fixed Priority Zero Laxity), à priorité dynamique, qui prennent en compte la laxité (*cf.* Section 1.2.1) pour déterminer la priorité d'exécution des tâches. Ils reposent sur la généralisation des algorithmes RM, EDF et FP[9]. Ce sont des algorithmes appelés à laxité nulle (ZL : Zero Laxity) du fait qu'ils attribuent une priorité maximale aux travaux dont la laxité atteint zéro, c'est-à-dire une fois que la durée d'exécution restante du travail devient égale au délai qu'il lui reste avant son échéance. De ce fait un tel travail s'exécute continûment. Ceci évite que les contraintes temporelles du travail ne soient pas respectées. Les autres travaux dont la laxité est différente de zéro, sont traités respectivement selon leurs règles habituelles : EDF, RM ou FP pour EDZL, RMZL ou FPZL.

[7] Rate Monotonic : c'est un algorithme d'ordonnancement temps réel à priorité fixe pour les tâches (FTP). Il attribue la priorité la plus forte à la tâche qui possède la plus petite période

[8] Earliest Deadline First : c'est un algorithme d'ordonnancement monoprocesseur à priorité dynamique. Il attribue une priorité à chaque job en fonction de sa date d'échéance, plus son échéance approche plus sa priorité augmente.

[9] Fixed Priority : un algorithme qui associe à chaque tâche une priorité fixe qui ne change pas pendant son cycle de vie.

1.2.3. APPROCHES OPTIMALES POUR L'ORDONNANCEMENT TEMPS REEL MULTIPROCESSEUR

Quoique certains algorithmes soient optimaux en monoprocesseur pour les ensembles de tâches qui respectent les conditions d'ordonnançabilité, ils ne permettent pas d'optimiser l'utilisation des processeurs totalement. En effet, Dhall[10] et Liu ont montré qu'un ensemble de tâches au facteur d'utilisation cumulé proche de 1 ne peut pas être correctement ordonnancé sur *m* processeurs en généralisant les politiques optimaux monoprocesseurs RM ou EDF **[8].**

De ce fait, d'autres politiques ont vu le jour pour proposer des améliorations aux techniques décrites précédemment, et qui ont fourni des résultats encourageants en termes d'optimalité en introduisant une notion d'équité entre les tâches. Dans cette sous-section nous allons présenter les principales politiques prouvées optimales en deux types d'environnements :

- **Systèmes en temps continu :** ces systèmes se distinguent par une résolution temporelle très grande, tous les paramètres des tâches d'un système en temps continu, tel que le temps d'exécution, la date d'échéance et la période, s'expriment en nombre réel.
- **Systèmes en temps discret** : tous les paramètres des tâches du système en temps discret s'expriment en nombres entiers multiples de l'unité de temps du système.

A) Politiques optimales en environnement de temps discret

Algorithme PFair

L'approche PFair (Proportionate Fair) a été démontrée optimale en ordonnancement multiprocesseur **[9]** dans un contexte de temps discret. Comme son nom l'indique, c'est une politique qui repose sur le principe de l'équité d'exécution entre les tâches. L'allocation du temps processeur pour chaque tâche est faite de manière dite « fluide ». En d'autres termes, chaque tâche $\boldsymbol{\tau_i}$ reçoit exactement $\boldsymbol{U_i \times t}$ unités de temps pour son exécution sur l'intervalle $[\mathbf{0}, \boldsymbol{t}[$. Dans la terminologie Pfair, le taux d'exécution de la tâche $\boldsymbol{\tau_i}$ est appelé poids et noté $\boldsymbol{\omega_i}$.

Le principe de l'équité consiste à discrétiser le temps en intervalles $[\boldsymbol{t}, \boldsymbol{t+1}[$ uniformes appelés des *slots*, une seule tâche est autorisée pour l'exécution sur un processeur pendant le slot. La séquence d'ordonnancement est modélisée par une fonction binaire $\boldsymbol{S(\tau_i, t)}$ telle que :

$$\boldsymbol{S(\tau_i, t)} = \begin{cases} \mathbf{1} & \boldsymbol{si\ \tau_i\ est\ exécutée\ sur\ le\ slot\ t} \\ \mathbf{0} & \boldsymbol{sinon} \end{cases}$$

Une fonction d'erreur d'exécution d'une tâche est définie par l'écart entre l'exécution idéale de la tâche et son exécution réelle résultante de l'ordonnancement $\boldsymbol{lag(\tau_i, t) = u_i.t - \sum_{l=0}^{l=t-1} S(\tau_i, l)}$. Le premier terme correspond à l'exécution idéale ou fluide, alors que le deuxième désigne l'exécution effective résultante de l'ordonnancement.

La construction d'un ordonnancement PFair conduit à diviser chaque tâche $\boldsymbol{\tau_i}$ en sous-tâches $\boldsymbol{\tau_i^j}$ de durée d'exécution unitaire, chacune d'elles devant être exécutée dans une fenêtre temporelle $\boldsymbol{\omega\left(\tau_i^j\right) = \left[r\left(\tau_i^j\right), d\left(\tau_i^j\right)\right]}$, avec $\boldsymbol{r\left(\tau_i^j\right) = \lfloor j - 1/U_i \rfloor}$ est la date de pseudo réveil de la sous tâche, et $\boldsymbol{\left(\tau_i^j\right) = \lceil j/U_i \rceil}$ est sa date de pseudo échéance.

[10] Sudarshan K. Dhall : Enseignant chercheur à l'université d'État de Murray aux états unis.

Un ordonnancement est PFair si à chaque instant t, l'erreur d'exécution est strictement inférieure à un quantum de temps : $\forall \boldsymbol{\tau_i} \in \boldsymbol{\tau}, |\boldsymbol{lag}(\boldsymbol{\tau_i}, \boldsymbol{t})| < \mathbf{1}$. Ceci implique que $\boldsymbol{u_i}.\boldsymbol{t} = \sum_{\boldsymbol{l=0}}^{\boldsymbol{l=t-1}} \boldsymbol{S}(\boldsymbol{\tau_i}, \boldsymbol{l})$.

Pour décider l'attribution des slots aux tâches prêtes, des règles de priorité sont utilisées, tel que l'algorithme EPDF « Earliest Pseudo-Deadline First » qui applique simplement EDF sur les pseudo échéances.

La figure 1-3 montre, pour une tâche $\boldsymbol{\tau_i}(\boldsymbol{C_i} = \mathbf{7}, \boldsymbol{T_i} = \mathbf{13})$ les fenêtres d'exécution des différentes sous-tâches. On constate que l'ordonnancement de la tâche respecte bien l'exécution successive de chaque sous-tâche au sein de sa fenêtre. **[6]**

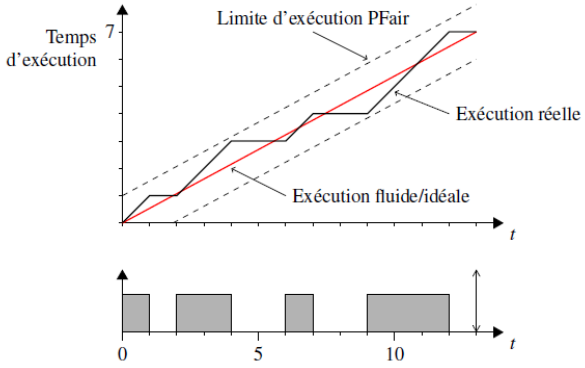


Figure 1-3 : Exécution fluide et PFair d'une tâche

Algorithme BFair

Bien que l'approche PFair mène vers l'optimalité, elle présente quelques défauts. En effet, la nécessité de décider de la sous-tâche à exécuter au bout de chaque slot introduit des surcoûts d'exécution et de migration des jobs.

Les auteurs Dakai Zhu[11] et *al.* **[10]** ont démontré que toutes les échéances peuvent être respectées en n'assurant la contrainte d'équité qu'au niveau des dates d'échéance, au lieu de la vérifier au début de chaque slot. Ils ont proposé un algorithme optimal appelé BF (Boundary Fairness). Le principe consiste à diviser le temps en tranches délimitées par deux dates d'échéance successives. Ces tranches sont notées $\boldsymbol{I_k} = [\boldsymbol{b_k}, \boldsymbol{b_{k+1}})$, tel que $\boldsymbol{b_k} = \mathbf{min}\{\boldsymbol{t}/\exists\ (\boldsymbol{\alpha}, \boldsymbol{i})\ \ \boldsymbol{avec}\ \boldsymbol{t} = \boldsymbol{\alpha}.\boldsymbol{T_i}, \boldsymbol{\alpha} \in \mathbb{N}\}$ sont les dates d'échéance successives. Ainsi, l'ordonnancement BF est appelé au début de chaque $\boldsymbol{I_k}$ pour définir un partage des $\boldsymbol{m}.(\boldsymbol{b_{k+1}} - \boldsymbol{b_k})$ unités de temps pour l'exécution de *N* tâches sur les *m* processeurs.

De manière schématique, la politique BFair réalise une dotation et une distribution temporelle dans l'intervalle $\boldsymbol{I_k}$:

- *La dotation temporelle* en nombres entiers qui consiste à décider pour chaque tâche une durée planifiée pour son exécution locale sur $\boldsymbol{I_k}$ qui va être notée $\boldsymbol{l_i^k}$.

  Toutes les *N* tâches commencent par recevoir un nombre entier d'unités temporelles d'exécution (Remaining Work) qui vaut : $\boldsymbol{m_i^k} = \mathbf{max}\{\mathbf{0}, \lfloor \boldsymbol{RW_i^k} + (\boldsymbol{b_{k+1}} - \boldsymbol{b_k}).\boldsymbol{u_i} \rfloor\}$, tel que $\boldsymbol{RW_i^k}$ est temps d'exécution restant pour la tâche $\boldsymbol{\tau_i}$ sur l'intervalle $\boldsymbol{I_k}$. Cette quantité, si elle

[11] Professeur à l'Université de Texas et member de l'IEEE Computer Society.

est uniquement allouée à la tâche $\boldsymbol{\tau_i}$, assure qu'en $\boldsymbol{b_{k+1}}$, $\boldsymbol{lag(\tau i, b_{k+1}) \epsilon [0,1]}$ c'est-à-dire que son exécution effective approchera par défaut son exécution fluide. Les tâches pour lesquelles cette erreur est non nulle et telles que $\boldsymbol{m_i^k < (b_{k+1} - b_k)}$ sont considérées éligibles pour être exécutées ;

Une fois que les $\boldsymbol{m_i^k}$ sont déterminés pour chacune des tâches, on évalue les unités d'exécution encore disponibles sur $\boldsymbol{I_k}$(Remaining Units). A cet effet, on définit $\boldsymbol{RU = m.(b_{k+1} - b_k) - \sum_{i=1}^{N} m_i^k}$. Ces RU sont partagées entre les *m* tâches éligibles les plus prioritaires de telle sorte que chaque tâche prioritaire ait une unité de temps, soit $\boldsymbol{o_i^k = 1}$, alors que les autres auront $\boldsymbol{o_i^k = 0}$.

Finalement la dotation temporelle de chaque tâche est définit par $\boldsymbol{l_i^k = m_i^k + o_i^k}$

- *La distribution temporelle* vise à répartir pour chaque tâche les $\boldsymbol{l_i^k}$ qui lui sont allouées sur les *m* processeurs dans l'intervalle $\boldsymbol{I_k}$. Elle est déterminée entièrement à l'instant $\boldsymbol{b_k}$, selon la règle suivante :

  Arbitrairement, les $\boldsymbol{l_i^k}$ des *m* tâches éligibles sont pris puis alloués sur le premier processeur à hauteur de $\boldsymbol{\max(\sum_{i=1}^{n} l_i^k/m, \max_i\{l_i^k\})}$, et de la même manière les autres $\boldsymbol{l_i^k}$ sont affectés aux processeurs restants selon la règle de remplissage de McNaughton que nous allons présenter par la suite **[11]**.

B) Politiques optimales en environnement de temps continu

Algorithme LLREF

LLREF (Largest Local Remaining Execution time First) a été à l'origine introduite par Cho et *al.* **[12]** Il s'agit d'une variante de BFair qui a été introduite pour les mêmes raisons mais pour les environnements en temps continu. Le principe de dotation temporelle reste le même sauf que sa valeur qui vaut : $\boldsymbol{l_i^k = u_i.(b_{k+1} - b_k)}$ n'est plus une valeur entière.

En ce qui concerne la distribution temporelle, elle repose sur une abstraction dite « T-L Plane » qui a été établie afin de présenter visuellement son principe, La figure 1-4 l'illustre sous la forme d'un jeton qui est à l'instant $\boldsymbol{b_k}$ en position $\boldsymbol{l_i^k}$, l'exécution de la tâche est modélisée par le déplacement du jeton suivant la bissectrice inverse, et sa préemption un déplacement horizontal. La finalité de la distribution temporelle est alors de déplacer les jetons de façon à ce qu'ils atteignent tous l'axe des abscisses au plus tard à $\boldsymbol{b_{k+1}}$. Le triangle isocèle de hauteur $\boldsymbol{I_i^k}$ représenté dans la figure 1-4 correspond au domaine de déplacement des jetons (T-L Plane).

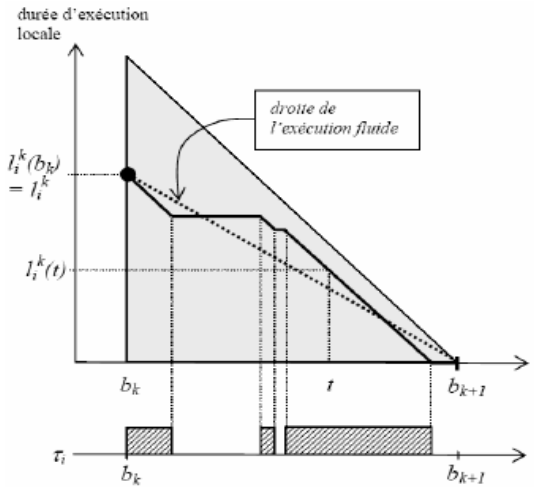


Figure 1-4 : Une trajectoire de τi au sein d'un T-L Plane

Au sein des trajectoires, des événements secondaires donnant lieu à des décisions d'ordonnancement sont introduits :

- L'événement B (Bottom hitting) qui définit la rencontre de la base du T-L Plane par un jeton, soit quand la tâche a consommé sa durée d'exécution totale.
- L'événement C (Ceiling hitting) qui correspond à la rencontre de la diagonale, et donc à l'annulation de la laxité locale de la tâche.

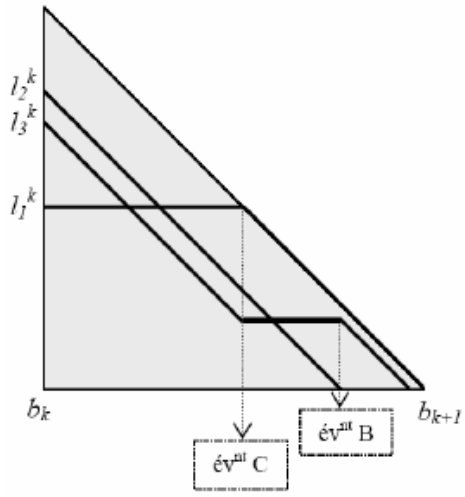


Figure 1-5 : Trajectoires et événements secondaires au sein d'un T-L Plane

Algorithme DP-WRAP

DP-WRAP (Deadline Partitioning-Wrap) est une technique qui a été présentée par Levin et *al.* **[13]** en 2010, et dont le principe de base repose sur la règle de McNaughton : pour ordonnancer les tâches du système, on construit des blocs chacun ayant une taille égale au taux d'utilisation de chaque tâche, ces blocs sont alignés arbitrairement et puis coupés en nouveaux blocs de taille unitaire.

Chaque bloc j obtenu correspond à l'assignation des tâches sur le j-ème processeur de la plateforme (figure 1-6).

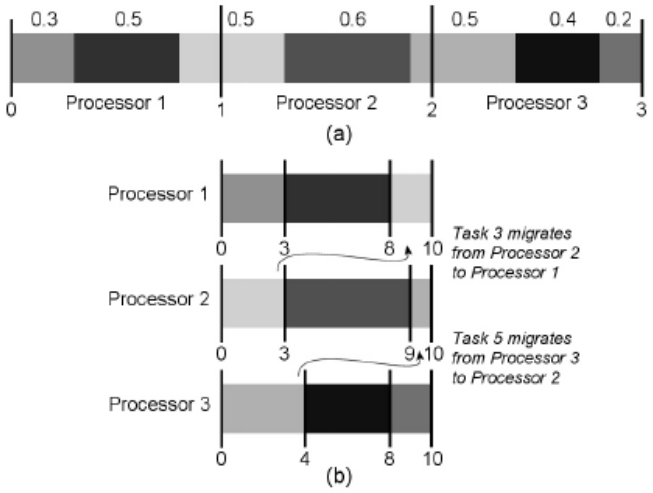


Figure 1-6 : Ordonnancement selon l'algorithme DP-WRAP

La figure 1-6 illustre le problème d'ordonnancement de 7 tâches sur 3 processeurs, dont les taux d'utilisation sont : $\boldsymbol{U_1 = 0.3, U_2 = 0.5, U_3 = 0.5, U_4 = 0.6, U_5 = 0.5, U_6 = 0.4, U_7 = 0.2}$. sur la figure (1-6.a) on a présenté la construction des blocs unitaires qui fournissent l'ordonnancement des tâches sur chaque processeur (1-7.b).

1.3. U-EDF : APPROCHE OPTIMALE POUR L'ORDONNANCEMENT TEMPS REEL DE TACHES SPORADIQUES

U-EDF (Unfair Earliest Deadline First) est une politique d'ordonnancement multiprocesseur, introduit par Geoffrey Nelissen en 2012 **[2]**. Il est destiné spécialement aux tâches sporadiques à échéance implicite, sans équité explicite.

Le principe d'U-EDF se base sur la généralisation horizontale d'EDF, politique optimale en monoprocesseur sous la condition : $\boldsymbol{U_{sum} = \sum_{i=1}^{N} U_i \leq 1}$. Le principe de cette généralisation consiste à exécuter autant que faire se peut, les tâches de plus fortes priorités, sur le premier processeur, puis le deuxième, et ainsi de suite.

Un tel principe ne peut être appliqué que si l'ordonnancement se calcule hors ligne. Toutefois, les auteurs proposent une mise en œuvre en ligne équivalente qui s'appuie sur deux étapes :

- A chaque fois qu'un travail est réveillé, on calcule pour chacun des travaux prêts et sur chaque processeur, un budget en fonction de leurs taux d'utilisation. Ce calcul tient compte aussid'une réservation pour les futurs travaux. Le détail de cette procédure est fourni dans la section 1.3.2.
- Une fois ces budgets sont calculés, on procède à un ordonnancement EDF-D (EDF with Delay) pour les travaux. EDF-D est une variante d'EDF, nous détaillerons par la suite le principe d'ordonnancement (*cf.* section 1.3.3).

1.3.1. NOTATIONS ET DEFINITIONS

Afin de présenter le principe de fonctionnement de la politique U-EDF, nous allons étudier le problème d'ordonnancement d'un système temps réel de N tâches $\boldsymbol{\tau = \{\tau_1, \tau_2, \dots, \tau_N\}}$ à échéances

implicites sur *m* processeurs identiques $P_1, P_2, \dots, P_m$. On considère que chaque tâche $\tau_i \stackrel{def}{=} < C_i, T_i >$ est soit périodique, soit sporadique, possède un pire temps d'exécution C_i et un temps inter-arrivée minimum de T_i. La tâche se définit aussi par un pire temps d'exécution restant $ret_i(t)$ à l'instant t, qui désigne les unités de temps qu'il reste à exécuter par τ_i à l'instant t. **[14]**

Dans cette partie nous allons exposer les principales notations que nous allons utiliser dans le reste du chapitre.

On suppose que le taux d'utilisation ne dépasse pas 1 pour chacune des tâches, et que le taux d'utilisation total $U \stackrel{def}{=} \sum_{i=1}^{N} Ui$ ne dépasse pas le nombre des processeurs m.

Définition 1.3.1 : (Travail Actif)

On définit un travail (job) actif de τ_i à un instant t comme étant le travail de $\tau_{i,j}$ qui est arrivé avant la date t et dont l'échéance est strictement supérieure à t. Autrement dit, c'est un job qui n'est pas encore arrivé à sa date d'échéance à l'instant t. De cette définition on peut dire qu'une tâche est active à l'instant t, si elle a un job actif à cet instant.

On peut également définir l'ensemble des tâches actives à t par :

$$\mathcal{A}(t) \stackrel{def}{=} \{ \tau_i \epsilon \tau \mid d_{i,j} > t \}.$$

$d_{i,j}$ est l'échéance du $j^{ème}$ travail de la tâche τ_i à l'instant t.

On note que même si un job a fini son exécution avant ou à l'instant *t*, il est toujours considéré actif si sa date d'échéance est supérieure à t.

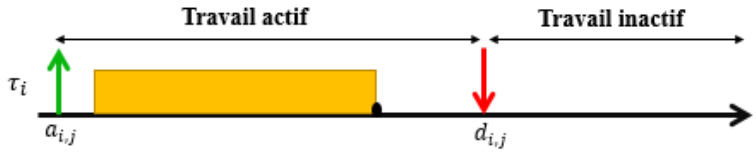


Figure 1-7 : travail actif et inactif

Définition 1.3.2 : (Echéance à l'instant t)

On définit la date d'échéance d'un travail de la tâche τ_i à l'instant t par $d_i(t)$, tel que :

- $d_i(t) = d_{i,j}$ *la première date d'échéance de la tâche τ_i qui survient après l'instant t, en d'autres termes la date d'échéance du job j courant de la tâche, si le travail est actif à cet instant (cf. Définition 1.3.1) ;*
- $d_i(t) = t$ *sinon.*

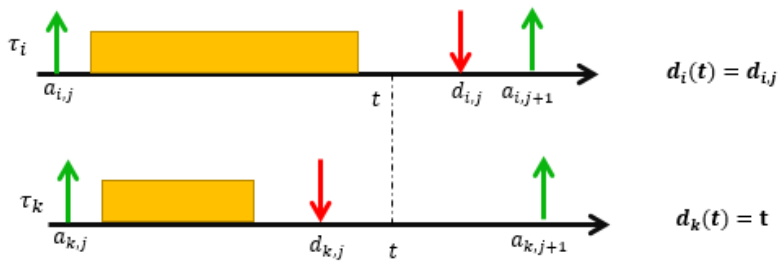


Figure 1-8 : exemple de détermination de la date d'échéance des travaux à l'instant t

Définition 1.3.3 : Ensemble des tâches plus (respectivement moins) prioritaires

L'ensemble des tâches plus (respectivement moins) prioritaires qu'une tâche $\boldsymbol{\tau_i}$ *à l'instant t* $\boldsymbol{hp_i(t)}$ *(respectivement* $\boldsymbol{lp_i(t)}$*), désigne l'ensemble des tâches actives dont la priorité est supérieure ou égale (respectivement inférieure) à celle de* $\boldsymbol{\tau_i}$ *à l'instant t, et on a :*

$$\boldsymbol{hp_i(t)} \stackrel{\text{def}}{=} \{\, \boldsymbol{\tau_j} : \left(\boldsymbol{d_j(t) < d_i(t)}\right) \vee \left(\left(\boldsymbol{d_j(t) = d_i(t)}\right) \wedge \boldsymbol{j < i}\right)$$

$$\boldsymbol{lp_i(t)} \stackrel{\text{def}}{=} \{\, \boldsymbol{\tau_j} : \left(\boldsymbol{d_j(t) > d_i(t)}\right) \vee \left(\left(\boldsymbol{d_j(t) = d_i(t)}\right) \wedge \boldsymbol{j > i}\right)$$

Définition 1.3.4 : (budget d'une tâche)

Le budget d'une tâche $\boldsymbol{\tau_i}$ *sur un processeur* $\boldsymbol{P_j}$ *entre deux instants* $\boldsymbol{t_1}$ *et* $\boldsymbol{t_2}$*, désigne la quantité de temps dédiée à la tâche sur le processeur* $\boldsymbol{P_j}$ *sur l'intervalle s'étendant de* $\boldsymbol{t_1}$ *jusqu'à* $\boldsymbol{t_2}$ *avec* $(\boldsymbol{t_2 > d_i(t_1)})$.

Ce budget comprend une dotation temporelle $\boldsymbol{Al_{i,j}(t_1)}$*, qu'on appelle également Allotment. Cette dotation est destinée à l'exécution de la tâche* $\boldsymbol{\tau_i}$ *sur le processeur* $\boldsymbol{P_j}$ *à l'instant* $\boldsymbol{t_1}$*, et une réservation* $\boldsymbol{res_{i,j}(d_i(t_1), t_2)}$ *pour l'exécution des jobs futurs de la tâche qui peuvent être réveillées entre la date d'échéance du job courant* $\boldsymbol{d_i(t_1)}$ *et* $\boldsymbol{t_2}$.

$$\boldsymbol{bdg_{i,j}(t_1, t_2)} \stackrel{\text{def}}{=} \boldsymbol{Al_{i,j}(t_1) + res_{i,j}(d_i(t_1), t_2)}.$$

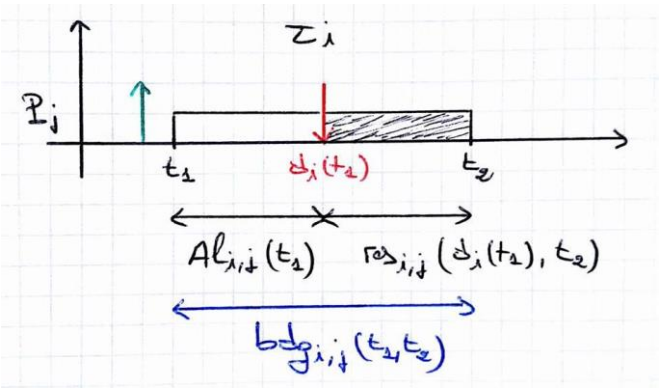


Figure 1-9 : budget d'une tâche τi

Définition 1.3.5 : (réservation d'un travail futur)

La réservation pour d'une tâche $\boldsymbol{res_{i,j}(t_1, t_2)}$ *entre deux instants* $\boldsymbol{t_1}$ *et* $\boldsymbol{t_2}$*, est le temps réservé sur le processeur* $\boldsymbol{P_j}$ *pour l'exécution de ses travaux futurs qui peuvent se réveiller entre* $\boldsymbol{t_1}$ *et* $\boldsymbol{t_2}$*, cette réservation est calculée comme suit :*

$$\boldsymbol{res_{i,j}(t_1, t_2)} \stackrel{\text{def}}{=} \boldsymbol{u_{i,j}(t) \times (t_2 - t_1)}$$

Où $\quad \boldsymbol{u_{i,j}(t)} \stackrel{\text{def}}{=} \left[\sum_{\tau_x \epsilon hp_i(t) \cup \tau_i} \boldsymbol{U_x}\right]_{j-1}^{j} - \left[\sum_{\tau_x \epsilon hp_i(t)} \boldsymbol{U_x}\right]_{j-1}^{j}$

Avec $\quad \boldsymbol{[x]_a^b} \stackrel{\text{def}}{=} \boldsymbol{max\{a, min\{b, x\}\}}$

$\boldsymbol{u_{i,j}(t)}$ désigne la proportion du facteur d'utilisation de la tâche $\boldsymbol{\tau_i}$ qui sera réservé sur le processeur $\boldsymbol{P_j}$ pour l'exécution de ses jobs futurs. Cette valeur peut être obtenue en rangeant des blocs

de taille égale à U_i pour chaque tâche τ_i, par ordre croissant selon leurs dates d'échéance actuelles $d_i(t)$. Après, on coupe cet alignement en nouveaux blocs de taille de 1. La portion de U_i contenue dans le j-ème bloc correspond à $u_{i,j}(t)$ (voir figure 1-10).

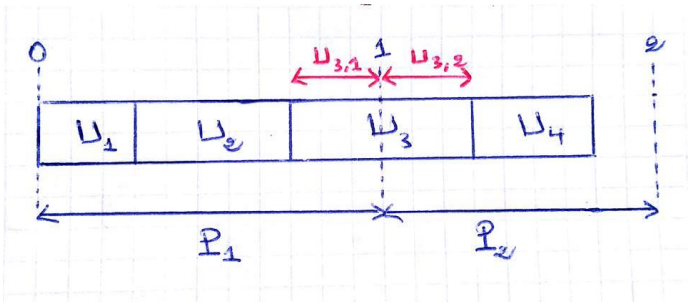


Figure 1-10 : calcul de Ui,j(t)

Définition 1.3.6: (dotation du travail courant)

Pour chaque instant t, la dotation temporelle, notée $Al_{i,j}(t)$ *ou* $allot_{i,j}(t)$*, d'une tâche* τ_i*, est la quantité du temps allouée sur le processeur* P_j *pour l'exécution de son travail actif. Elle se définit comme suit :*

$$Al_{i,j}(t) \stackrel{\text{def}}{=} min\{Al_{i,j}^{max}(t), ret_i(t) - \sum_{y<j} Al_{i,y}(t)\}.$$

Le premier terme correspond à la dotation temporelle maximale (*cf* Définition 1.3.7) et le deuxième sert à déterminer les unités de temps que τ_i doit toujours exécuter après avoir enlevé ce qui lui était attribué sur les processeurs précédents pour son exécution.

Définition 1.3.7 (dotation temporelle maximale)

A chaque instant t, la dotation temporelle maximale d'une tâche τ_i *sur un processeur* P_j *est définie par la quantité de temps maximale qu'on peut allouer au job actif de la tâche* τ_i *dans l'intervalle de temps* $[t, d_i(t))$*. Cette borne maximale est donnée par la formule :*

$$Al_{i,j}^{max}(t) \stackrel{\text{def}}{=} (d_i(t) - t) - \sum_{\tau_x \epsilon hp_i(t)} bdg_{x,j}(t, d_i(t)) - \sum_{y=1}^{j-1} Al_{i,y}(t)$$

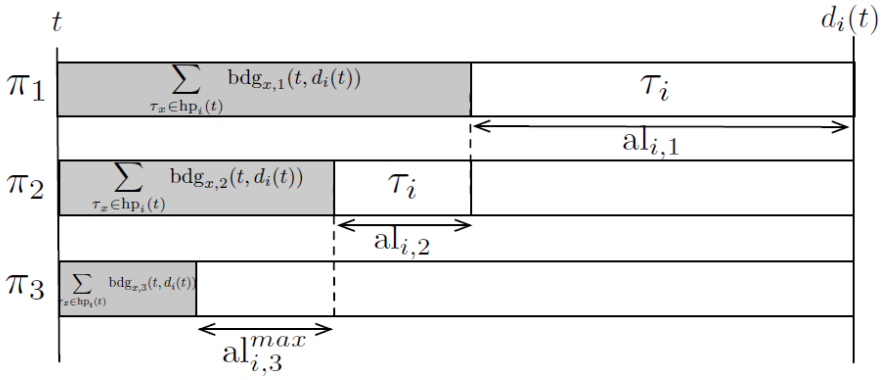


Figure 1-11 : calcul de la dotation maximale de τi *sur* $\pi 3$

Propriété 1.3. 1

La dotation temporelle d'une tâche τ_i *sur un processeur* P_j *diminue au fur et à mesure qu'elle s'exécute sur* P_j*. Ainsi, si on suppose qu'aucun nouveau job de la tâche n'apparaisse dans*

l'intervalle $[t, t')$*, et si on désigne par* $exec_{i,j}(t, t')$ *le temps durant lequel* τ_i *s'est exécutée sur* P_j *dans l'intervalle* $[t, t')$*, alors :*

$$Al_{i,j}(t') \stackrel{def}{=} Al_{i,j}(t) - exec_{i,j}(t, t').$$

Propriété 1.3. 2

Afin qu'un job courant d'une tâche τ_i *puisse respecter son échéance, il faut que la somme des unités de temps allouées à son exécution sur la plateforme soit, à tout instant t, au moins égale à son pire temps d'exécution restants* $ret_i(t)$ *:*

$$\sum_{j=1}^{m} Al_{i,j}(t) \geq ret_i(t).$$

Définition 1.3.8 : (Tâche éligible)

On dit qu'une tâche τ_i *est éligible à l'instant t sur le processeur* P_j*, si et seulement si :*

- τ_i *ne s'exécute pas à l'instant t sur un autre processeur d'indice plus petit que j ;*
- *Sa dotation temporelle actuelle sur* P_j *est non nulle (i.e.,* $Al_{i,j}(t) > 0$*).*

Remarque 1.3.1 :

Pendant l'exécution, si aucun nouveau travail ne s'est réveillé, chacune des tâches doit poursuivre son exécution pour la durée d'allotment qu'on lui a attribué. Ceci implique, que les événements d'ordonnancement, autres qu'un réveil de tâche, ne conduisent pas à une préemption des tâches qui sont en cours d'exécution.

1.3.2. PREMIERE PHASE : PRE-ALLOCATION

L'étape de pré-allocation consiste à déterminer la durée allouée à chaque tâche τ_i pour l'exécution de son job actuel sur l'ensemble des processeurs de la plateforme, et la durée réservée pour l'exécution de ses jobs futurs (ce qui constitue ce qu'on appelle le budget temporel d'une tâche (cf. définition 2.2.4)). Ainsi, en des instants précis que nous allons spécifier par la suite, l'ordonnanceur temps réel est appelé pour calculer ces durées, en se basant sur les principes suivants :

- Pour chaque tâche τ_i, on calcule sa dotation temporelle $Al_{i,j}(t)$ (*cf.* définition 1.3.6) sur chaque processeur P_j, de telle sorte qu'on alloue suffisamment de temps pour qu'elle termine l'exécution de son job actuel avant sa date d'échéance (*cf.* propriété 1.3.2).
- Pour chaque tâche τ_i, on réserve pour ses jobs futurs, une durée $res_{i,j}(d_i(t), t_2)$ proportionnelle à son taux d'utilisation U_i, sur chaque tranche de temps après son échéance $d_i(t)$ c'est-à-dire qu'on réserve $U_i \times l$ unités de temps pour chaque intervalle de durée *l.*

L'algorithme 1.3.1 présente le principe de la phase de pré-allocation de la politique U-EDF. Ainsi, pour chaque tâche on alloue une valeur égale au minimum entre $Al_{i,j}^{max}(t)$ et son pire temps d'exécution restant et qui n'est pas encore assigné à un autre processeur.

Il faut noter que ces calculs se font à chaque fois qu'un nouveau travail d'une tâche arrive. De ce fait, il n'y a aucune garantie qu'une tâche τ_i sera effectivement exécutée pendant la durée qui lui a été allouée sur P_j. **[15]**

Algorithme 1.3. 1

Input:
TaskList := list of the n tasks sorted by increasing absolute deadlines;
t := current time;

```
1 forall the τ_i ∈ TaskList do
2 |   for j := 1 to m do
3 |   |   al_{i,j}(t) := min{al^max_{i,j}(t) , ret_i(t) − Σ_{y<j} al_{i,y}(t)};
4 |   end
5 end
```

1.3.3. DEUXIEME PHASE : ORDONNANCEMENT

Une fois que la pré-allocation des tâches sur les processeurs de la plateforme est faite, on procède à l'ordonnancement du système en utilisant une variation d'EDF appelé EDF-D, jusqu'à la prochaine activation d'un travail (où il faut faire une réallocation des tâches). L'idée générale d'EDF-D est d'ordonnancer les tâches sur chaque processeur en utilisant le principe d'EDF, sauf que si une tâche s'exécute sur un autre processeur, EDF-D retarde son exécution sur le processeur actuel, afin d'éviter le parallélisme d'exécution des jobs, d'où l'appellation « with Delays ».

L'algorithme EDF-D fonctionne comme suit :

- Premièrement, on construit un ensemble $\boldsymbol{\varepsilon_j(t)}$ de tâches dites éligibles à l'instant *t* sur le processeur$\boldsymbol{P_j}$ (*cf.* Définition 1.3.8) ;
- à chaque instant, EDF-D ordonnance les tâches éligibles dans l'ensemble $\boldsymbol{\varepsilon_j(t)}$ selon la date d'échéance la plus proche.

Algorithme 1.3. 2

Data: t := current time

```
1 if t is the arrival time of a job then
  |   // Recompute the task allotment
2 |   Call Algorithm 1.3.1
3 end
4 for j := 1 to m do
  |   // Use EDF-D
5 |   Update ℰ_j(t) :
6 |   Execute on P_j the task with the earliest deadline in ℰ_j(t) ;
7 end
```

L'algorithme 1.3.2 résume le processus de fonctionnement de l'algorithme U-EDF. A chaque fois qu'un nouveau travail rejoint le système, l'ordonnanceur recalcule les dotations des tâches en appelant l'algorithme 1.3.1. Ensuite, sur chaque processeur la liste des tâches éligibles s'actualise et ces tâches sont exécutées selon l'algorithme EDF.

En effet, les articles qui expliquent le principe d'U-EDF n'étaient pas clairs sur l'instant où l'on doit actualiser la liste des tâches éligibles. Les auteurs expliquent que $\boldsymbol{\varepsilon_j(t)}$ s'actualise si un nouveau travail arrive, si une tâche arrive à son échéance ou finit son exécution. Cependant, rien n'a été spécifié sur le cas où la tâche $\boldsymbol{\tau_i}$ finit l'exécution du temps qu'il lui a été alloué sur le processeur

$\boldsymbol{P_j}$. A ce moment-là il parait indispensable d'actualiser la liste des tâches éligibles, puisqu'une dotation devient nulle $\boldsymbol{Al_{i,j}(t) = 0}$ (i.e. $\boldsymbol{\tau_i}$ n'est plus éligible sur $\boldsymbol{P_j}$).

Finalement, après avoir contacté les auteurs, nous avons pu établir les précisions suivantes :

- L'ordonnancement est appelé pour effectuer la pré-allocation à chaque fois qu'un nouveau job se réveille (ce qui confirme ce que nous avons présenté précédemment) ;
- La mise à jour des tâches éligibles est faite lorsqu'un nouveau travail se réveille, lorsqu'une tâche finit son exécution, finit l'exécution du temps qu'on lui a attribué sur un processeur ou arrive à son échéance.

1.3.4. EXEMPLE

On considère le système de 3 tâches sporadiques à échéances implicites $\boldsymbol{\tau_i = (C_i, T_i)}$. Les 3 tâches sont synchrones à l'instant t=0 : $\boldsymbol{\tau_1 = (3, 6)}$; $\boldsymbol{\tau_2 = (6, 8)}$; $\boldsymbol{\tau_3 = (5, 10)}$, avec $\boldsymbol{U_1 = 1/2}$; $\boldsymbol{U_2 = 3/4}$; $\boldsymbol{U_3 = 1/2}$. On dispose de 2 processeurs $\boldsymbol{P_1}$ *et* $\boldsymbol{P_2}$ pour l'ordonnancement de ce système.

On rappelle que pour chaque tâche sporadique $\boldsymbol{\tau_i}$, la valeur de $\boldsymbol{T_i}$ correspond à la durée minimale qui sépare deux réveils de travaux successifs.

Tout d'abord on commence par la détermination des $\boldsymbol{u_{i,j}(0)}$ en utilisant la méthode présentée dans la sous-section 1.3.1 (figure 1-12). $\boldsymbol{d_1(0) < d_2(0) < d_3(0)}$, ce qui fait que à t=0 $\boldsymbol{\tau_1 \gg \tau_2 \gg \tau_3}$[12].

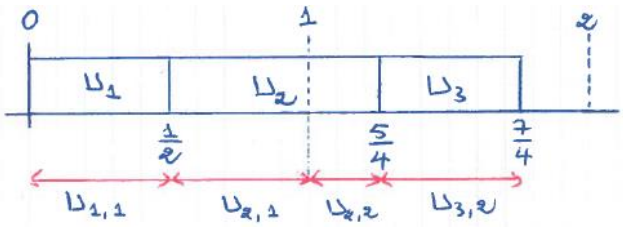


Figure 1-12 : Calcul des Ui,j des 3 tâches

En utilisant la définition 1.3.6 pour le calcul de la dotation temporelle des travaux courants (arrivés à l'instant 0), et la définition 1.3.5 pour le calcul de la réservation pour les travaux futurs (qui peuvent arriver après $\boldsymbol{d_i(0)}$) de chaque tâche $\boldsymbol{\tau_i}$ sur les processeurs $\boldsymbol{P_1}$ *et* $\boldsymbol{P_2}$, on trouve les résultats regroupés dans le tableau 1.1. On prend comme intervalle d'étude [0 , 12] afin de pouvoir illustrer le principe de réservation après l'échéances des 3 tâches :

[12] $\tau_i \gg \tau_j$: Cette notation signifie que τ_i *est plus prioritaire que* τ_j

Tableau 1-1 : résultats de calcul selon U-EDF

Tâche τ_i		$u_{i,j}(0)$	Dotation temporelle $Al_{i,j}(0)$	Réservation $res_{i,j}(d_i(0), 12)$
τ_1	P_1	½	**3**	**3**
	P_2	**0**	**0**	**0**
τ_2	P_1	½	**4**	**2**
	P_2	¼	**2**	**1**
τ_3	P_1	**0**	**0**	**0**
	P_2	½	**5**	**1**

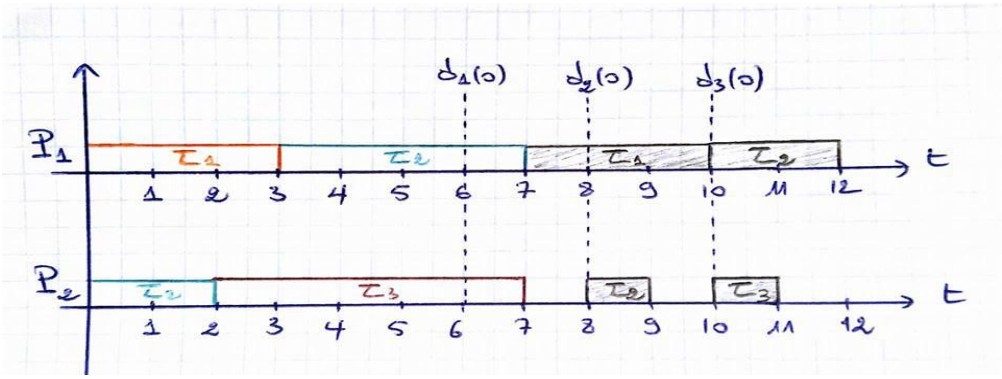


Figure 1-13 : exécution selon U-EDF

La figure 1-13 présente la séquence d'ordonnancement des tâches selon U-EDF. En effet, si aucun job n'arrive avant la date t = 12, les tâches seront exécutées pour leurs durées d'exécution attribuées sur chaque processeur, et une réservation pour leurs jobs futurs sera faite après leurs échéances relatives, en fonction de leurs taux d'utilisation (voir tableau 1-1).

On note qu'aux dates t = 2, t = 3 et t = 7, on aura une mise à jour de la liste des tâches éligibles sur les processeurs de la plateforme (cf. sous-section 1.3.1).

1.3.5. CONDITION D'OPTIMALITE

Les auteurs ont démontré qu'U-EDF est une politique optimale pour l'ordonnancement des tâches sporadiques avec échéance implicite dans un système dynamique en multiprocesseur, ceci sous la condition que le taux d'utilisation total du système ne dépasse pas le nombre m de processeurs constituant la plateforme, et que le taux d'utilisation de chaque tâche ne dépasse pas 1. C'est-à-dire :

$$\forall t \quad U_{sum} \leq m \qquad et \qquad \forall \tau_i \epsilon \tau \; U_i \leq 1$$

Un lecteur intéressé pour découvrir la démonstration de la condition d'optimalité d'U-EDF pour les tâches sporadiques, est invité à consulter la thèse de Geoffrey Nelissen « Efficient Optimal Multiprocessor Scheduling Algorithms for Real-Time Systems » **[2]**.

1.4. CONCLUSION

L'objectif de ce chapitre est de donner au lecteur les clés de compréhension de l'univers de l'ordonnancement des systèmes temps réel, en présentant une vue d'ensemble sur les principales politiques d'ordonnancement existantes. A la fin du chapitre, nous avons présenté quelques politiques

dont l'optimalité a été démontrée, telles que BFair, LLREF et DP-WRAP. Cependant, vu qu'ils reposent sur le principe de l'équité, le nombre de migrations et de préemptions introduites par ces algorithmes est toujours grand. Ils restent d'autres algorithmes qui ont aussi prouvé leur performance, à savoir U-EDF qui a été introduit récemment par Geoffrey Nelissen et qui supprime l'aspect de l'équité d'exécution entre tâches. Le principe de cet algorithme a été fourni à la fin de ce chapitre ainsi qu'un aperçu sur la condition de son optimalité. Une autre variante d'U-EDF, pour un environnement en temps discret, a été proposée par les auteurs en plus d'une version améliorée pour une implémentation efficace de l'algorithme. Un lecteur souhaitant avoir plus de détails est invité à consulter une étude que nous avons faite dans le cadre d'un séminaire bibliographique. **[16]**

CHAPITRE.2. TRAMPOLINE : UNE IMPLEMENTATION DU STANDARD OSEK/VDX

Sommaire

Au cours de ce chapitre nous allons présenter le standard OSEK sur lequel se base le RTOS Trampoline. La première section donne un panorama sur les services qu'OSEK propose. Une deuxième section se focalisera sur la présentation de l'architecture de Trampoline et un aperçu sur la méthode de son utilisation.

2.1. LE STANDARD OSEK/VDX

OSEK est le sigle pour « Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug », ce qui veut dire en français : Systèmes ouverts et interfaces correspondantes pour l'électronique des véhicules automobiles. C'est un standard qui a été créé en 1993 par un consortium de constructeurs et équipementiers automobiles allemands (BMW, Bosh, DaimlerChrysler, Opel, Siemens, et VW) ainsi qu'un département de l'université de Karlsruhe. Leur but était de développer un standard pour une architecture ouverte reliant les divers contrôleurs électroniques d'un véhicule afin de réduire les coûts récurrents relatifs au développement et redéveloppement des logiciels embarqués sur ces contrôleurs. En 1994, les constructeurs français Renault et PSA qui développaient un projet similaire, VDX (Vehicle Distributed eXecutive), rejoignirent le consortium. Ainsi, les deux entités de standardisation ont présenté, en Octobre 1995, leur standard harmonisé OSEK/VDX. Et depuis, OSEK/VDX a acquis une reconnaissance plus large dans l'industrie, et un comité de pilotage qui comprend plusieurs constructeurs mondiaux. **[17]**

OSEK/VDX définit un ensemble de spécifications pour le développement de système temps réel pour l'automobile. On trouve entre autres :

- **OSEK/VDX OS :** définit une interface standardisée pour le système d'exploitation monoprocesseur et offre les fonctionnalités pour gérer les systèmes de contrôle par évènements.
- **OSEK/VDX COM :** définit les protocoles de communication inter-tâches et inter-modules.
- **OSEK/VDX NM :** définit les protocoles de gestion des réseaux en runtime.
- **OSEK/VDX OIL :** le langage de configuration qui permet aux concepteurs de décrire la configuration du système OSEK pour la configuration et la génération.

La portabilité et la fiabilité sont, sans doute, les plus grands avantages qu'offre le standard OSEK. Les concepteurs développent des applications pour des APIs standardisés et non pour un RTOS particulier, ainsi, il est plus facile de commuter d'un processeur à un autre. En outre, OSEK exige l'utilisation d'un RTOS statique, ce qui veut dire que l'application contient uniquement les éléments dont elle a besoin, et qui sont désignés avant le démarrage. Cette configuration permet d'assurer un fonctionnement optimal de l'application.

2.1.1. SYSTEME D'EXPLOITATION OSEK

Le système d'exploitation est la première composante du standard OSEK/VDX, il définit les APIs du RTOS et un ensemble de services. Cette définition se fait statiquement, ce qui est important pour les industries où la sûreté de fonctionnement et la fiabilité sont critiques. Cela réduit également la complexité et les facteurs inconnus et facilite les tests et le débogage du système. En contrepartie, une spécification dynamique est plus ouverte et flexible.

Le système d'exploitation OSEK permet une exécution temps réel contrôlée de plusieurs processus[13] qui fonctionnent parallèlement. Ainsi, il fournit un certain nombre d'interfaces pour l'utilisateur. Ces interfaces sont utilisées comme entités concurrentes pour l'utilisation du CPU. On distingue deux types d'entités :

- ISR (Interrupt Service Routines) : ce sont des routines que le hardware déclenche une fois qu'une interruption est reçue.
- Les tâches : les tâches fournissent une structure logicielle (*Framwork*) pour l'exécution des fonctions d'une application temps réel.

Le RTOS d'OSEK définit trois niveaux de traitement : (i) traitement des interruptions ; (ii) traitement des de l'ordonnanceur ; (iii) traitement des tâches. Cependant ce traitement se fait selon une priorité : les tâches possèdent une priorité de traitement inférieure à celle l'ordonnanceur, et celui-ci possède une priorité inférieure à la priorité des interruptions. Concernant le contexte[14] d'exécution, il est occupé au début de l'exécution et devient libéré à nouveau une fois que la tâche est terminée (*cf.* figure 2-1). **[18]**

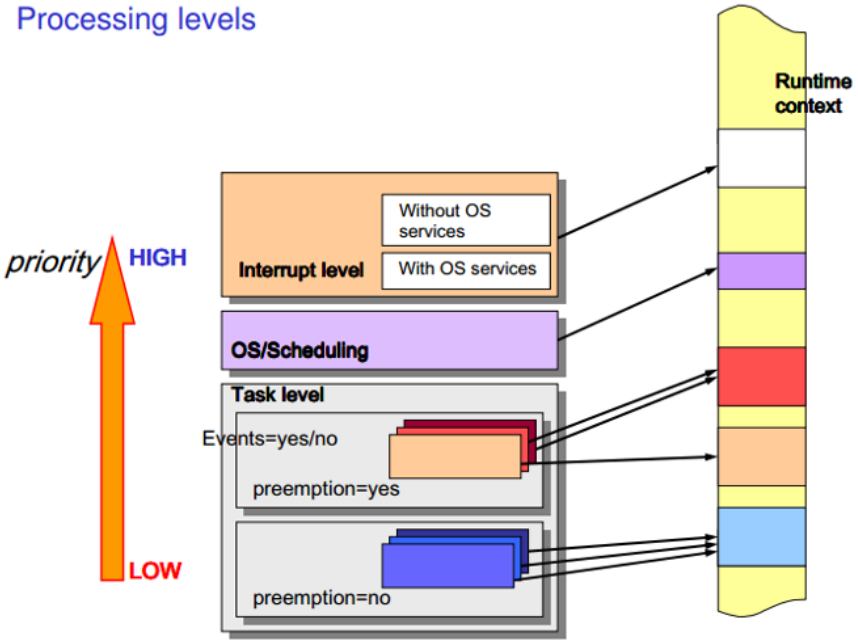


Figure 2-1 : niveaux de traitement dans un système d'exploitation OSEK

2.1.2. LA GESTION DES TACHES

Etats des tâches OSEK :

Au sein d'OSEK, il existe deux catégories de tâches : basique et étendue. La principale différence entre les deux types de tâches est le nombre d'états qu'ils supportent et leur capacité à supporter les services d'évènements. Dans la spécification du RTOS, la machine à états finis est composée de quatre états :

[13] Dans ce document, nous allons considérer qu'un processus est soit une tâche ou bien une ISR2 (*cf.* 2.1.3)
[14] Le contexte d'exécution (run time context) représente les ressources de mémoire attribuée à une tâche pour son exécution.

- Suspendue (Suspended) : la tâche est passive et peut être activée.
- Prête (Ready) : la tâche est prête à être exécutée, et attend donc l'allocation du CPU.
- Exécutée (Running) : le CPU est alloué à la tâche en question, et elle est donc en train d'être exécutée.
- En attente (Waiting) : la tâche est arrêtée car elle attend un événement (sémaphore, libération d'une ressource …)

Les tâches basiques ne supportent pas l'état Waiting. Aussi, elles ne supportent pas les services d'évènements susceptibles de la basculer dans cet état.

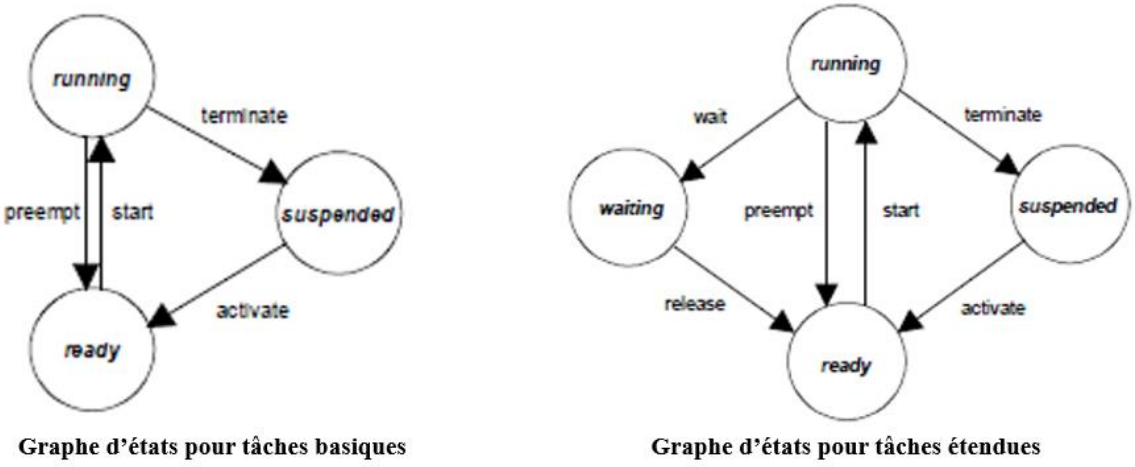


Figure 2-2 : les états d'une tâche temps réel selon le standard OSEK/VDX

Quelques services de gestion des tâches OSEK :

Ci-dessous, nous allons présenter les principaux services associés aux états des tâches OSEK :

- ActivateTask : c'est le service qui permet de lancer une nouvelle instance (Job) de la tâche. Ainsi, cette dernière passe de l'état *Suspended* à l'état *Ready*.
- TerminateTask : c'est un service qui termine l'instance en cours de la tâche : elle passe de l'état *Running* à *Suspended*. Une tâche ne peut terminer une autre tâche, elle ne peut que s'auto-terminer.
- Schedule : c'est un appel à l'ordonnanceur. Si une tâche plus prioritaire est dans l'état *Ready*, les ressources de la tâche en cours sont libérées, la tâche est placée dans l'état *Ready*, et celle de plus haute priorité se voit allouer le CPU.
- GetTaskState : renvoie l'état actuel de la tâche.
- GetTaskID : renvoie l'identifiant de la tâche en cours d'exécution.
- DeclareTask : permet de déclarer une tâche avant l'utilisation de son identifiant dans le programme d'application.

2.1.3. LES INTERRUPTIONS

Les fonctions servant à traiter les interruptions (ISR : Interrupt Service Routine) existent sous deux formes :

- ISR catégorie 1 : l'ISR ne fait pas appel aux services de l'OS. Ces interruptions sont transparentes pour l'OS, rapides et ne nécessitent pas de synchronisation avec lui.
- ISR catégorie 2 : l'ISR fait appel aux services de l'OS, c'est-à-dire que le code de l'interruption contient des appels à l'API.

Les interruptions sont ordonnancées par le matériel et peuvent interrompre les tâches préemptives et non-préemptives. Si une ISR active une tâche, alors le RTOS ordonnance la tâche après la terminaison de toutes les interruptions.

2.1.4. LES EVENEMENTS

Les événements sont la propriété d'une tâche étendue. Ils sont utilisés pour synchroniser les différentes tâches et la signalisation entre elles. Toute tâche, même les basiques, peuvent émettre un événement. Toutefois, seule la tâche propriétaire de l'évènement peut le consommer ou l'attendre.

2.1.5. ORDONNANCEMENT DES TACHES

OSEK implémente une politique d'ordonnancement à priorités fixes. La priorité la plus basse est 0. L'ordonnanceur va donc choisir quelle tâche doit être exécutée selon sa priorité. Le système peut être vu comme un ensemble de tampons, un tampon représentant une priorité. Ils sont gérés selon une politique FIFO. Il existe trois types d'ordonnancement possibles sous OSEK :

- *Full preemptive :* si une tâche de plus haute priorité que celle s'exécutant est dans l'état Ready, alors c'est elle qui prend la main sur le CPU. La tâche ainsi préemptée est placée en tête du buffer de sa priorité.
- *Non-preemptive* : les tâches ne peuvent être préemptées.
- *Mixed preemptive :* dans ce cas, la politique suivie dépend de la tâche qui s'exécute. Si elle est définie comme ne pouvant pas être préemptée, alors elle ne le sera pas. Si elle est définie comme pouvant être préemptée, alors on adoptera une politique préemptive.

2.1.6. GESTION DE TEMPS

Le système d'exploitation OSEK offre deux outils pour gérer le temps : les alarmes et les compteurs. Leur rôle est de synchroniser les activations de tâches avec des événements récurrents.

Au niveau de l'implémentation interne du système d'exploitation, on a des compteurs qui sont dérivés de timers implémentés en hardware ou en software et ils sont représentés par une valeur en ticks. Chaque implémentation fournit au moins un compteur qui peut être utilisé pour ordonnancer des activations ou des événements périodiques.

Au niveau de l'application on a des Alarmes. A chaque alarme on associe un compteur et une valeur d'expiration. Quand le compteur atteint cette valeur, l'alarme associée est déclenchée. On peut associer plusieurs alarmes à un seul compteur. A chaque alarme on associe soit une tâche soit une alarm-calback routine. Ces alarmes peuvent être des alarmes cycliques (récurrentes) ou uniques. Le système OSEK offre aussi la possibilité d'annuler les alarmes et de consulter l'état actuel d'une alarme.

Les alarmes et les compteurs sont définis statiquement, ainsi que l'association alarme-compteur et l'action à exécuter au moment de sa date d'expiration. Les dates d'expiration et les valeurs des compteurs par contre peuvent être modifiés dynamiquement. **[19]**

2.1.7. LES CLASSES DE CONFORMITES

Le système d'exploitation doit pouvoir s'adapter à différents niveaux de complexité du système logiciel qui peut être limité par les ressources disponibles (type de processeur, taille de mémoire, …). C'est pourquoi OSEK introduit le concept de classes de conformité (conformance classes) dont l'objectif est de proposer un niveau de fonctionnalités croissant pour les systèmes des plus simples aux plus complexes.

Quatre classes de conformités sont définies, elles permettent une meilleure adaptation du RTOS au contexte de l'application tout en permettant la réutilisation du code. Ces classes de conformités sont basiques ; BCC1 et BCC2, ou étendues ; ECC1 et ECC2. La principale différence entre les classes 1 et 2 est le nombre de tâches activées par niveau de priorité. BCC1 permet uniquement l'utilisation des tâches basiques et limite le nombre de tâches activées par niveau de priorité à un. Alors que la BCC2 permet de multiples activations de tâches basiques et plusieurs tâches par niveau de priorité. De même pour ECC1 et ECC2, mais en incluant le type de tâches étendues. **[20]**

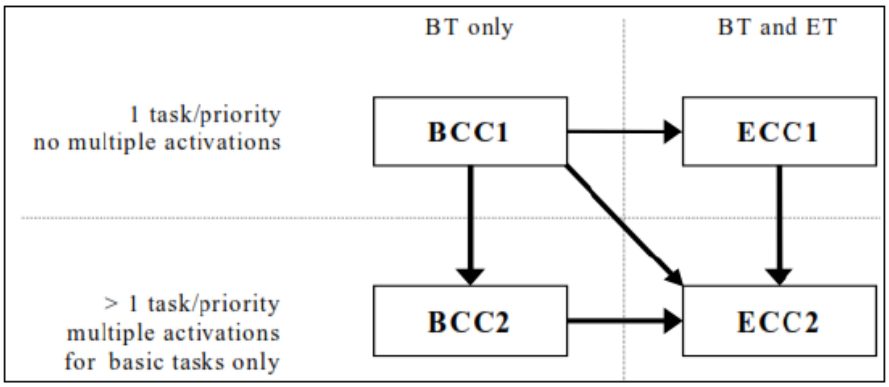


Figure 2-3 : les classes de conformité OSEK

2.2. L'EXECUTIF TEMPS REEL TRAMPOLINE

Trampoline est un système d'exploitation temps réel qui se base sur le standard OSEK/VDX 2.2.3 et son successeur AUTOSAR[15]. Il est développé dans l'équipe Systèmes Temps Réel de l'IRCCyN depuis 2005, dans le but d'un usage interne et par la suite pour la recherche scientifique, l'enseignement et pour l'expérimentation de nouveaux algorithmes.

Etant donné que notre travail se focalise sur l'implémentation d'une politique d'ordonnancement multiprocesseur, nous allons nous concentrer, dans ce qui suit, sur l'architecture de Trampoline en multicœur.

[15] AUTOSAR (AUTomotive Open System Architecture) : c'est un partenariat de développement mondial de l'industrie automobile. Il a pour but de développer et d'établir une architecture logicielle standardisée et ouverte pour les unités de contrôle électronique (ECU) des véhicules. Il est à noter qu'AUTOSAR implémente les architectures multi-cœur, et utilise l'ordonnancement partitionné.

2.2.1 ARCHITECTURE DE TRAMPOLINE

L'architecture de Trampoline comprend trois parties majeures (voir figure 2-4) :

Le BSP (Board Support Package) :

Il regroupe les types de données et les fonctions qui sont dépendantes du jeu d'instructions et de l'architecture du micro-contrôleur. Il est constitué de 4 modules :

- ***l'External Interrupt Handler :*** c'est le module qui gère les interruptions externes qui peuvent venir d'un *timer* ou d'un autre périphérique. Lorsque l'interruption conduit à un ré-ordonnancement, l'Externel Interrupt Handler interagit avec l'*Interrupt Dispatcher du Kernel* et avec le *Context switch manager*.
- ***Le System Call Handler :*** il gère les appels de service et le passage du micro-contrôleur du mode utilisateur au mode superviseur. Il interagit avec le *Context Switch Manager* lorsque le service a engendré un ré-ordonnancement et avec le *Memory Protection Manager* pour changer les droits d'accès mémoire.
- ***Le Context Switch Manager :*** il effectue la sauvegarde du contexte du processus qui perd le CPU et la restauration du contexte du processus qui gagne le CPU. Il interagit avec *l'External Interrupt Handler* et le *System Call Handler*.
- ***Le Memory Protection Manager :*** il gère la protection mémoire en fonction du contexte d'exécution. Il programme la *Memory Protection Unit* ou la *Memory Management Unit* selon que l'exécution a lieu en mode superviseur ou bien en mode utilisateur.

Le Noyau :

Le noyau (*kernel*) regroupe les fonctions de base de gestion de l'ordonnancement, de démarrage du système d'exploitaion, de démarrage et d'arrêt des processus et synchronisation des processus. Il se compose de 3 modules :

- ***L'Interrupt Dispatcher :*** il effectue une ou plusieurs des opérations suivantes : (i) l'incrémentation d'un ou plusieurs compteurs ; (ii) l'exécution d'une ISR1 ; (iii) l'activation d'une ISR2.
- ***Le Counter Manager :*** il est sollicité par *l'Interrupt Dispatcher* si la source d'interruption correspond à l'incrémentation d'un ou plusieurs compteurs.
- ***Le Scheduler :*** il occupe une position centrale. Il est sollicité par *l'Interrupt Dispatcher*, le *Counter Manager* et par tous les services qui nécessitent un ré-ordonnancement.

L'API :

C'est une interface qui comprend les services exposés à l'application. Les services peuvent être activés et désactivés selon les besoins de manière à n'inclure que les services nécessaires à l'application.

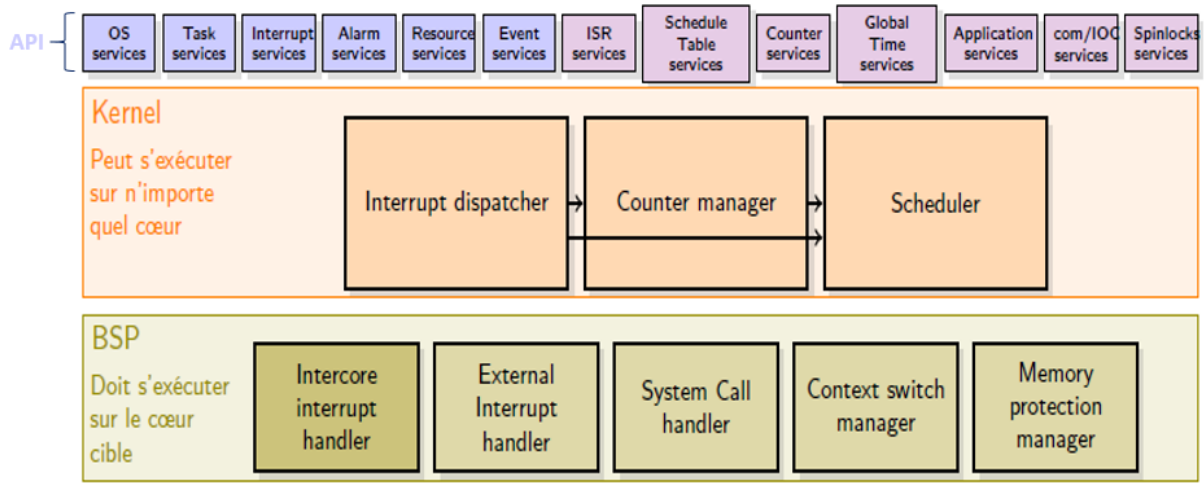


Figure 2-4 : Architecture de Trampoline multicoeur. Les services OSEK sont en bleu et les services AUTOSAR en violet

2.2.2 LA SOLUTION D'ORDONNANCEMENT ACTUEL

Comme nous l'avons précisé, Trampoline se base sur le standard OSEK, et donc l'implémentation actuelle de la fonctionnalité d'ordonnancement des tâches se base sur une politique à priorités fixes. En multicœur, la plateforme supporte des ordonnanceurs partitionnés.

Le *Scheduler*[16] s'occupe de la gestion d'une liste des processus prêts. Cette liste n'est rien d'autre que la structure de données où les descripteurs des processus[17] prêts sont rangés et classés par ordre de priorité. Si deux processus ont la même priorité, on les classe par ordre d'activation. L'implémentation actuelle de Trampoline traite une structure de données sous forme de tableau de files d'attente en FIFO. Ce tableau est indexé par la priorité du processus. Ainsi, les descripteurs des processus sont rangés dans cette file d'attente, dans l'ordre de leur activation pour chaque niveau de priorité (*cf.* figure 2-5).

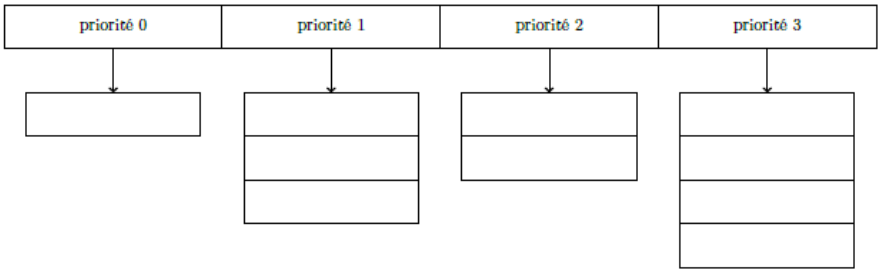


Figure 2-5 : Structure de la liste des processus prêts.

Le *Scheduler* s'occupe de faire passer les processus d'un état à l'autre par le biais de fonctions associées à chacune des transitions montrées sur la figure 2-2. Ces fonctions sont présentées comme suit :

[16] Il est nécessaire de noter la différence entre le composant « scheduler » de Trampoline et le module « ordonnanceur ». Le premier est une entité qui s'occupe de la gestion des listes de tâches, des calculs d'ordonnancement … Le deuxième est un module interne au scheduler qui, lui, se charge seulement de passer les tâches d'un état à un autre.

[17] Les descripteurs de processus sont des fichiers où sont stockées toutes les informations à propos des attributs des processus. Ils existent des descripteurs statiques où on définit les attributs statiques, et des descripteurs dynamiques pour les attributs dynamiques des processus.

- **tpl_activate_task** : correspond à la transition *activate* et passe une tâche de l'état *Suspended* à l'état *Ready* si le compteur d'activation[18] est nul. Cette fonction incrémente également le compteur d'activations. Elle est appelée soit à partir des services *ChainTask* ou *ActivateTask*, qui eux appellent le *scheduler*, soit à partir d'une action associée à une alarme.

- **tpl_terminate** : correspond à la transition *terminate* et passe une tâche de l'état *Running* à l'état *Suspended* dans le cas d'un compteur d'activation égal à 1. Si ce compteur est supérieur à 1, la tâche passe en *Ready*. Cette fonction est appelée à partir des services *ChainTask* et *TerminateTask* (qui appellent le *scheduler*).

- **tpl_start** : correspond à la transition *start* et passe un processus de l'état *Ready* à l'état *Running*. Cette fonction est appelée par l'ordonnanceur.

- **tpl_preempt** : correspond à la transition *preempt* et passe un processus de l'état *Running* à l'état *Ready*. Cette fonction est aussi appelée par l'ordonnanceur.

- **tpl_block** : correspond à la transition block et passe une tâche étendue de l'état *Running* à *Waiting* si l'événement attendu n'est pas positionné.

- **tpl_set_event** : correspond à la transition release et passe une tâche étendue de l'état *Waiting* à l'état *Ready* si l'événement positionné correspond à un événement attendu.

Le *Scheduler* gère également une structure de données par cœur appelée *tpl_kern*. Cette structure permet de mémoriser les informations relatives au processus en cours d'exécution sur chaque cœur. Les informations suivantes sont mémorisées :

- **running_id :** l'identifiant du processus en cours d'exécution ;

- **running** : un pointeur vers le descripteur dynamique du processus en cours d'exécution ;

- **s_running** : un pointeur vers le descripteur statique du processus en cours d'exécution ;

- **elected_id** : l'identifiant du processus qui réclame le CPU ;

- **elected** : un pointeur vers le descripteur dynamique du processus qui réclame le CPU ;

- **need_switch** : indique qu'un changement de contexte entre le processus en cours d'exécution et le processus qui le réclame, doit être effectué ;

- **need_schedule** : indique qu'un ré-ordonnancement doit être effectué.

Ainsi, lorsqu'une tâche est en cours d'exécution, on a *running_id == elected_id*. Si un ré-ordonnancement s'avère nécessaire, il se peut que la tâche *running* soit préemptée pour que la plus prioritaire de liste prête passe en *elected*. A la fin de la décision de l'ordonnanceur, un changement de contexte doit être effectué ainsi qu'une copie du contenu d'*elected* en *running* (voir figure 2-6).

Cette procédure assure qu'aux instants de ré-ordonnancement un seul changement de contexte est effectué à la fin. C'est parce qu'on peut avoir des ré-ordonnancements multiples au même instant, sans que le pointeur *running* soit affecté. **[21]**

[18] Le compteur d'activations sert à mémoriser un job si le job courant de la tâche n'est pas terminé.

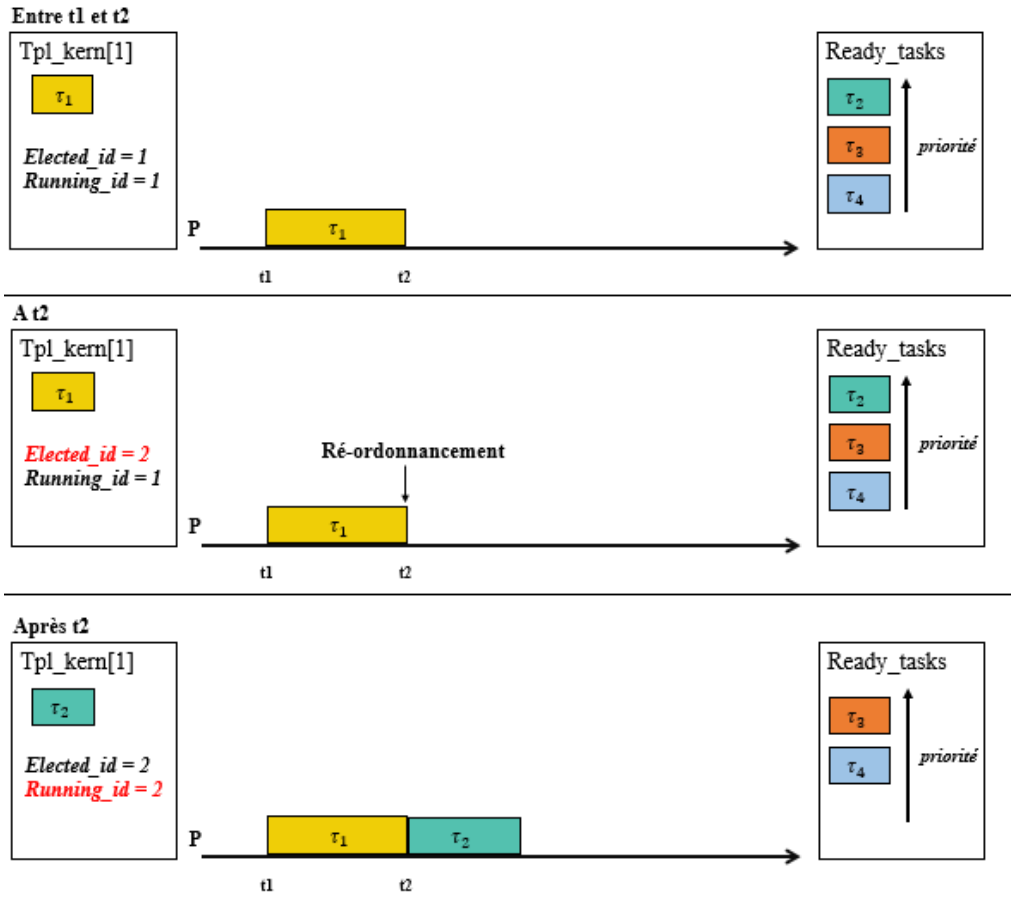


Figure 2-6 : mise à jour de tpl_kern dans le cas de ré-ordonnancement ($\tau_1 \gg \tau_2 \gg \tau_3 \gg \tau_4$)

2.2.3 DEMARCHE D'UTILISATION DE TRAMPOLINE

Trampoline étant un système temps réel statique, il n'offre pas la possibilité de créer des objets en ligne. Ainsi, une définition des objets du système doit être entièrement configurée avant la compilation. Pour cela, un langage spécifique au standard OSEK permet de définir tous les paramètres des objets de l'application (configuration de tâches, ISRs, alarmes, compteurs, événements et ressources), il s'agit du langage OIL (Osek Implementation Language). **[22]**

Une description sous OIL peut être écrite à la main ou générée par un outil de configuration système. Elle contient deux parties : (i) une définition d'implémentation : dans laquelle on déclare tous les attributs et leurs propriétés ; (ii) une définition d'application : elle contient les valeurs attribuées (initiales ou par défaut) aux attributs déclarés dans la définition d'implémentation.

Le compilateur OIL (GOIL) vérifie la syntaxe du fichier de description et fait appel à un interpréteur de Templates pour la génération des structures de données qui, elle, est dépendante du RTOS. Ces structures permettent ensuite de compiler l'application. GOIL peut également générer d'autres fichiers tels que le mapping mémoire ou le script d'édition de liens selon les attribues renseignés dans le fichier de description OIL.

Pour mettre en œuvre une application en Trampoline, il faut d'abord commencer par définir un diagramme fonctionnel de l'application. Ce diagramme doit détailler les différents objets OSEK qui seront utilisés, ainsi que les dépendances éventuelles entre les tâches et les ressources partagées. **[23]**

Une fois que cela soit fait, il devient facile de traduire le diagramme en langage OIL, car tous les objets OIL y apparaissent déjà. Ensuite les structures de données propres à Trampoline sont générées grâce à GOIL (*cf.* figure 2-7).

La dernière étape consiste à définir le corps de l'application en C, en définissant ce que font exactement les objets OSEK.

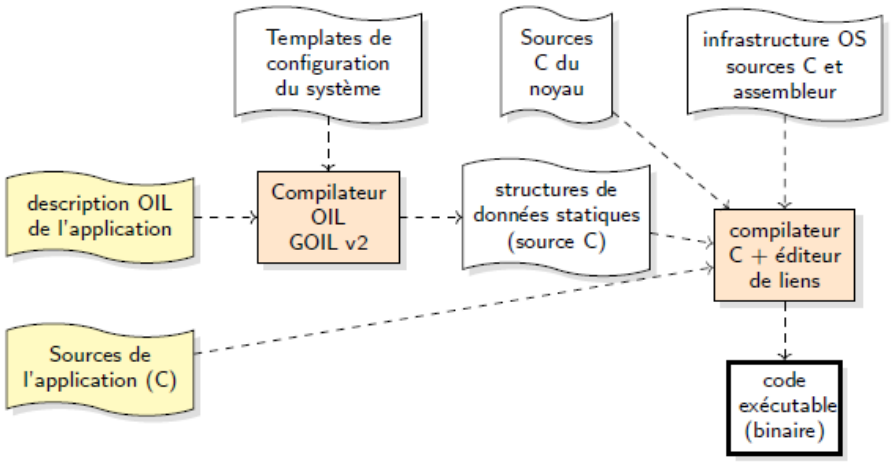


Figure 2-7 : procédure de développement d'une application en Trampoline

2.2.4 DESCRIPTION DES OBJETS OSEK A L'AIDE D'OIL

Le principe de la description des objets du système avec l'OIL est présenté comme suit **[21]** :

Description du CPU : le CPU est utilisé comme un conteneur de tous les objets

Description de l'OS : un seul OS doit être défini dans un CPU, sa description consiste à définir un *Statut* (*Standard* ou *Extended*) et des *Hook routines*[19].

Description de l'*APPMODE* : c'est un objet qui définit l'une des propriétés de l'OS qui spécifie les objets qui seront démarré au moment de démarrage de l'OS.

Description d'une tâche : la tâche est créée lors de la phase d'initialisation, grâce à un type particulier disponible en OIL (*Task*). Ce type se caractérise par les attributs suivants : *PRIORITY* : priorité de la tâche ; *SCHEDULE* : définit si la tâche est préemptive ou non ; *ACTIVATION* : nombre d'activations simultanées possibles ; *AUTOSTART* : un booléen qui définit si la tâche se lance dès le démarrage de l'application ou non ; *RESOURCE* : liste des ressources utilisées par la tâche ; *EVENT* : les événements auxquels la tâche est censée réagir.

Description d'un compteur ou d'une alarme : l'alarme est un objet OSEK utilisé pour notifier des événements récurrents. Deux états d'alarme peuvent être définis : SLEEP ou ACTIVE. Chaque alarme est reliée à un compteur qui sert à un Timer comptant les ticks d'horloge. Une fois que l'alarme s'active, Trampoline calcule sa date d'expiration et

[19] OSEK fournit des mécanismes de « crochets » qui permettent à l'utilisateur de dérouter le déroulement normal de l'OS de façon à prendre temporairement le contrôle du système. Les routines crochets (*Hook routines*) sont appelées par l'OS et ont une priorité supérieure à toutes les tâches.

l'insert dans la queue du compteur. Quand l'alarme s'expire elle est automatiquement retirée de cette queue.

Description d'une ressource : cet objet est utilisé pour coordonner l'accès des tâches ou ISR aux ressources partagées. Il permet aussi de coordonner l'accès à des bouts de code critiques ne pouvant être interrompus.

Exemple d'une application décrite en OIL : la figure 2-8 présente une simple application en OIL, dans laquelle sont définis quelques objets OSEK (*OS, APPMODE, COUNTER, ALARM* et *TASK*).

```
CPU program {
  OS ExampleOS {
    STATUS = STANDARD;
    };

  APPMODE appmode1 {};

  COUNTER Counter1{
    MINCYCLE = 16;
    MAXALLOWEDVALUE = 125;
    TICKSPERBASE = 90;
  };

  ALARM alarm1 {
    COUNTER = Counter1;
    ACTION = ACTIVATETASK { TASK = task1; };
    AUTOSTART = TRUE { APPMODE = appmode1; ALARMTIME = 100; CYCLETIME = 100; };
  };

  ALARM alarm2 {
    COUNTER = Counter1;
    ACTION = ACTIVATETASK { TASK = task2; };
    AUTOSTART = TRUE { APPMODE = std; ALARMTIME = 1000; CYCLETIME = 0; };
  };

  TASK task1 {
    PRIORITY = 1;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
  };

  TASK task2 {
    PRIORITY = 2;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
  };
};
```

Figure 2-8 : exemple d'application simple décrite en OIL

2.3. CONCLUSION

Ce chapitre a débuté par une présentation détaillée du standard OSEK/VDX sur lequel se base l'exécutif temps réel Trampoline. Un exécutif qui a été développée au sein de l'équipe « Systèmes Temps Réel » spécialement pour la recherche scientifique. Trampoline représente la plateforme sur

laquelle nous visons d'implémenter un ordonnanceur global, ainsi nous avons exposé son architecture comme nous avons introduit les différentes étapes qui constituent la conception d'une application temps réel. Pour cela il est indispensable de présenter le langage OIL qui sert à la configuration et la déclaration des objets OSEK de l'application.

CHAPITRE.3. VERS UNE IMPLEMENTATION D'U-EDF AU SEIN DE TRAMPOLINE

Sommaire

Comme nous l'avons vu plus haut, la politique d'ordonnancement U-EDF donne priorité aux tâches ayant les échéances absolues les plus proches et les exécute pour des durées calculées précisément. Cependant, lorsqu'il s'agit de la mise en œuvre, plusieurs difficultés surgissent. Les contraintes d'espace et de puissance de calcul rencontrées dans les systèmes embarqués font de l'implémentation un défi en soit. Au cours de ce chapitre, nous allons présenter notre mise en œuvre de cette politique.

Dans un premier temps, nous allons préciser les modifications les plus importantes à apporter à l'algorithme original de la politique U-EDF. Ensuite, nous allons identifier les attributs nécessaires pour son implémentation, avant d'attaquer directement la présentation et l'analyse de la mise en œuvre d'U-EDF. Finalement nous allons identifier quelques situations de difficultés qui peuvent empêcher une implémentation efficace de la politique.

3.1. ADAPTATION D'U-EDF POUR UNE IMPLEMENTATION EN TRAMPOLINE

3.1.1. MODIFICATIONS APPORTEES A L'ALGORITHME U-EDF

Nous avons présenté dans la section 1.3 le principe de fonctionnement de la politique U-EDF. Ainsi, ce principe dévoile deux algorithmes (*cf.* figure 3-1), l'un doit être appelé aux instants de calcul de la pré-allocation (arrivée d'un nouveau job) qu'on appellera *algorithme de pré-allocation*. L'autre doit être appelé aux instants de ré-ordonnancement (arrivée d'un nouveau job, terminaison d'une tâche, consommation d'une dotation temporelle ou bien atteinte d'une échéance), et va être appelé *algorithme de scheduling*.

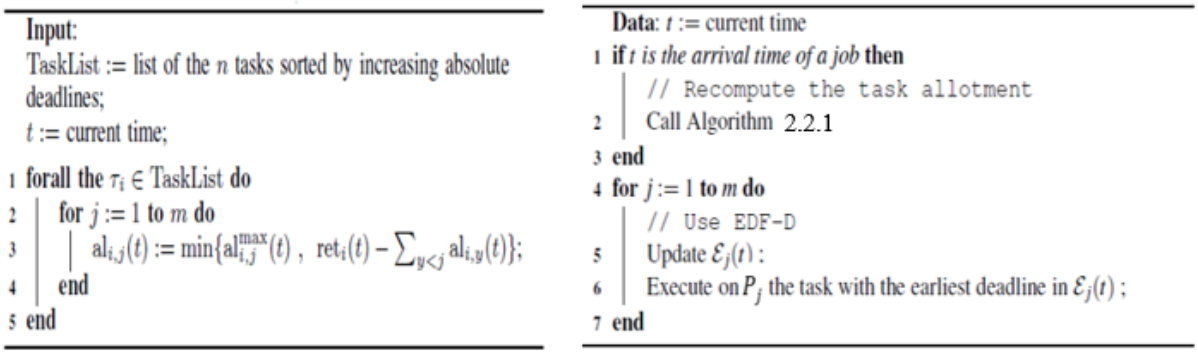


Figure 3-1 : Algorithmes de la politique U-EDF

Bien qu'U-EDF calcule efficacement les budgets d'exécution des tâches d'un système temps réel, en tenant compte aussi bien des budgets des jobs plus prioritaires (qu'ils soient actifs ou non), il n'est pas aussi facile à implémenter que le laisse paraitre la figure ci-dessus. D'ailleurs plusieurs facteurs doivent être pris en compte.

Si on considère par exemple la complexité de calcul de la dotation temporelle (allotment) $\boldsymbol{Al_{i,j}(t)}$ dans l'algorithme de pré-allocation. La détermination de la dotation d'une tâche $\boldsymbol{\tau_i}$ nécessite la détermination du budget temporel de toutes les tâches qui sont plus prioritaires qu'elle (*cf.* section 1.3). Ce calcul de budget peut avoir une complexité de l'ordre de *O(n)*. Et puisque l'algorithme de pré-allocation contient deux boucles 'for' imbriquées ($\boldsymbol{O(m \times n)}$), dans ce cas, la complexité devient de

l'ordre $\boldsymbol{O(m \times n^2)}$. Cette complexité est considérablement élevée pour un algorithme d'ordonnancement qui se calcule en ligne.

En plus de la complexité, l'implémentation doit considérer également la prise en compte des conditions pour décider l'exécution d'une tâche, la nature des attributs nécessaires pour l'implémentation, les structures de données nécessaires … Ainsi, nous avons constaté que les algorithmes proposés par les auteurs ne sont pas détaillés et restent ambigus sur quelques situations, ce qui pourrait mener vers des résultats inefficaces ou même erronés :

1er constat : A chaque fois que l'algorithme de *scheduling* est appelé, il faut choisir, pour tous les processeurs, d'exécuter la tâche de plus haute priorité dans la liste des tâches éligibles. Si on suppose que *t* est un instant où une tâche $\boldsymbol{\tau_i}$ consomme son allotment sur un processeur $\boldsymbol{\pi_j}$, alors il faut faire un ré-ordonnancement pour sélectionner une autre tâche parmi les tâches éligibles sur le processeur qui vient de se libérer. Or, le ré-ordonnancement proposé consiste à balayer tous les processeurs même ceux qui ne se sont pas libérés, ce qui ramène à avoir $\boldsymbol{m - 1}$ itérations en vain. De même pour les instants de terminaison de tâche, la sélection d'une nouvelle tâche ne concerne que les processeurs qui se libèrent.

Ce constat se base sur le fait qu'en dehors du cas d'un réveil de tâche, une tâche en cours d'exécution ne pourra être préemptée jusqu'à ce qu'elle termine la durée d'exécution qu'on lui a allouée (voir remarque 1.3.1).

Ainsi nous proposons d'ajouter une condition pour vérifier si le processeur est libre avant d'aller mettre à jour sa liste des tâches éligibles.

2ème constat : Nous avons constaté, qu'aux instants de franchissement d'échéance, aucun changement n'est réellement produit sur la décision d'exécution des tâches. C'est parce que le seul intérêt de cette date est de repérer que la tâche n'est plus considérée active (*cf.* définition 1.3.1), et donc elle doit devenir hautement prioritaire puisque sa date d'échéance est égale à l'instant courant (*cf.* définition 1.3.2). Ainsi nous proposons d'éviter un autre balayage de processeurs inutile, et de mettre à jour juste l'échéance de la tâche en question.

3ème constat : Ce constat est très important car il s'agit d'un facteur pouvant changer le résultat d'exécution selon U-EDF. Pour mieux l'expliquer il est nécessaire de passer par un exemple.

On suppose que *t* est l'instant d'arrivée d'un nouveau job et donc l'instant de calcul des allotments. On suppose également qu'après les calculs, l'ordonnanceur décide d'exécuter la tâche $\boldsymbol{\tau_i}$ sur le processeur $\boldsymbol{\pi_j}$ pour une durée d'allotment égale à $\boldsymbol{Al_{ij}(t)}$. Si aucun événement ne s'est produit pendant l'exécution de la tâche $\boldsymbol{\tau_i}$, alors sa fin d'allotment sera à $\boldsymbol{t + Al_{ij}(t)}$. Ainsi, l'algorithme de *scheduling* sera appelé pour sélectionner une autre tâche pour s'exécuter sur le processeur $\boldsymbol{\pi_j}$ (parmi ses tâches éligibles). Cependant, l'algorithme dans ce cas ne prévoit pas une actualisation ou annulation des allotment qui viennent d'être consommés, ce qui fait que $\boldsymbol{\tau_i}$ pourra être sélectionnée sur $\boldsymbol{\pi_j}$ même si elle vient de consommer sa dotation, ce qui confondra le déroulement normal de l'algorithme.

De ce fait, nous allons prévoir une mise à jour des allotments des tâches à chaque événement de type fin d'exécution ou consommation de dotation temporelle.

4ème constat : Dans sa thèse **[2]**, G.Nelissen a démontré qu'à l'instant qui suit immédiatement le moment t_a d'arrivée d'un nouveau job (on l'appelle t_a^+), et juste après le calcul de tous les allotments, nous avons :

Pour chacune des tâches τ_i, la somme des allotments qu'on lui a alloués sur tous les processeurs, doit être égale à sa durée d'exécution restante à l'instant t_a^+ (remaining execution time) : $\sum_{j=1}^{m} Al_{ij}(t_a^+) = ret_i(t_a^+)$.

Ce lemme nous a poussé à considérer d'arrêter le calcul des allotments d'une tâche une fois qu'on atteint une somme d'allotments égale à son *remaining execution time*. Ceci réduira considérablement le nombre d'itérations et des calculs faits, surtout que le calcul des allotments est amplement compliqué. Ceci conduit à remplacer, dans l'algorithme de *pré-allocation*, la boucle *for* qui balaye les processeurs par une boucle *while* qui s'arrête une fois que la somme des allocations atteint le *remaining execution time*.

5ème constat : U-EDF est une politique d'ordonnancement global, soit qu'il faut considérer une seule liste de tâches prêtes pour tous les processeurs. Cependant, les auteurs ont prévu également d'avoir pour chaque processeur une liste de tâches éligibles (*cf.* définition 1.3.8). Ainsi, il s'avère qu'il faut disposer d'une structure de données qui regroupe les tâches prêtes en plus de *m* autres structures de données pour *m* processeurs. En outre, au même instant, il est possible qu'une tâche soit éligible sur plusieurs processeurs, et donc il faut à chaque fois l'ajouter et la supprimer (au cas où elle commence ou finit son exécution) sur ces mêmes structures. Ceci, nécessite un traitement coûteux.

Pour y remédier, nous avons décidé de ne considérer qu'une seule structure de données, dans laquelle seront regroupées les tâches prêtes. Pour la notion de tâche éligible, nous allons considérer que toute tâche ayant un $Al_{ij}(t) > 0$ est éligible sur le processeur π_j.

L'algorithme va donc être modifié. Ainsi, nous allons balayer une seule liste de tâches prêtes triées par échéance croissante. Si une tâche prête est éligible sur un processeur, elle va y être assignée et supprimée de la liste des tâches prêtes, ainsi elle ne sera pas éligible sur d'autres processeurs. De même pour le processeur, une fois qu'on lui assigne une tâche, il n'est plus considéré comme ressource libre.

6ème constat : D'après les constats précédents, il s'est avéré qu'il y aura des traitements différents pour chaque événement de ré-ordonnancement. Pour cela, nous allons décider de formuler 4 algorithmes de ré-ordonnancement pour les 4 événements, en plus de l'algorithme de calcul de pré-allocation.

3.1.2. ALGORITHMES PROPOSES

Comme nous l'avons évoqué dans la section précédente, nous allons modifier l'algorithme U-EDF pour que chaque événement soit traité séparément. Ainsi, nous avons ajouté 4 algorithmes supplémentaires aux ceux qu'ont proposé les auteurs. Nous avons également modifié ces deux derniers. Ces algorithmes sont présentés ainsi : un pour la pré-allocation, un pour la décision d'ordonnancement, et 4 pour le traitement des 4 événements de ré-ordonnancement : « *réveil de tâche* », « *consommation de dotation* », « *franchissement d'échéance* » ou « *terminaison de tâche* ».

Algorithme 3.1.1 : Pré-allocation

En se basant sur le 4ème constat, nous avons décidé de remplacer la boucle *for* dans l'algorithme original par une boucle *while* qui s'arrête une fois qu'on alloue la durée nécessaire pour

l'exécution d'une tâche $\boldsymbol{\tau_i}$ à un instant *t*. Ainsi, on ne va plus faire des calculs couteux une fois que l'allocation nécessaire à une tâche est faite.

Nous avons également ajouté une variable *previousAllot* qui désigne, pour une tâche $\boldsymbol{\tau_i}$, la somme des dotations qu'on lui a allouées sur les processeurs précédents ($y < j$), lorsqu'on veut calculer la dotation sur un processeur j. Ainsi, si on veut calculer la dotation temporelle de la tâche $\boldsymbol{\tau_i}$ sur le premier processeur ($j=1$), *previousAllot* doit être nulle car il n'y a pas de processeur qui précède $\boldsymbol{P_1}$. Pour les autres processeurs $\boldsymbol{P_j}$ $(\boldsymbol{j > 1})$, on récupère la variable *previousAllot* de manière récurrente. Ceci évite de recalculer les allocations précédentes de la tâche à chaque itération.

Pre-Allocation

Input:
readyTasks := List of n ready tasks sorting by increasing absolute deadline;
t := currentTime;
Data:
$hpBdg_{i,j}$ *:= execution budget of higher priority tasks of a task* τ_i *in a processor* π_j*;*
prviousAllot := variable that stores the sum o f execution time allocated in the previous; processors f or a given task;
for all $\tau_i \in$ readyTasks **do**
previousAllot := 0;
while $(\text{ret}_i(t) > \sum_{y<j} \text{Al}_{i,y}(t)$ **and** $j < m)$ **do**
Get the budget $\text{hpBdg}_{i,j}$;
// cf. fig 1-11
$\text{Al}_{i,j}(t) = \min\{d_i(t) - t - \text{hpBdg}_{i,j}, \text{ret}_i\} - \sum_{y<j} \text{Al}_{i,y}(t)$;
previousAllot := previousAllot + $\text{Al}_{i,j}(t)$;
j := j+1;
End while
End for

Algorithme 3.1.2 : Arrivée d'un nouveau travail

Dans cet algorithme, on propose dans un premier temps de mettre à jour quelques attributs de la tâche nouvellement activée. D'abord on passe son état en *Ready*, on intialise son *remaining execution time* à la valeur de sa durée d'exécution (*WCET*) puis sa date d'échéance à l'instant *t* est mise à la valeur de la date d'échéance absolue. Ensuite, on procède à une mise à jour des *remaining execution times* des tâches, qui étaient en train de s'exécuter avant la date *t*, juste après leur préemption.

Le réveil d'une tâche est un événement qui nécessite un re-calcul des dotations temporelles et un ré-ordonnancement. Pour cela, nous nous sommes servi de deux booléens *needAllotRecompute* et *needReschedule*, qui correspondent respectivement à la nécessité de re-calcul d'allocation et de ré-ordonnancement. Pour ce type d'événement, les deux booléens sont mis à *true*.

Algorithm 2 U-EDF_activate_task

Input:
taskId := the Id of the new activated task;
t := current time
Data:
runningTasks := List of m running tasks;
readyTasks := List of n ready tasks;
needAllotRecompute := boolean to indicate wether to calculate the allotment or not;
needReschedule := boolean to indicate wether to reschedule tasks or not;

Update the state of the task taskId (suspended -> ready);
Update the remaining execution time and the absolute deadline of the task taskId;
for all $\tau_i \in$ runningTasks **do**
    Preempt τ_i;
    Update $ret_i(t)$;
end for
for all $\tau_i \in$ readyTasks **do**
    $Al_{i,j}(t) := 0$;
end for
needAllotRecompute := **true**;
needReschedule := **true**;

Algorithme 3.1.3 : Consommation d'une dotation temporelle

Comme nous l'avons précisé au constat 3, il est indispensable de mettre à jour la dotation temporelle de la tâche qui vient de la consommer, c'est ce que nous proposons dans cet algorithme. Ainsi, nous avons prévu une mise à jour d'*allotment* et du *remaining execution time* de la tâche en question et aussi des autres tâches qui étaient en train de s'exécuter. Une fois que cela est fait, on met la tâche dont l'allotment a fini en *Ready*, et le booléen *needReschedule* à *true* puisque un tel événement nécessite un ré-ordonnancement.

Il faut préciser que la mise à jour des attributs *remaining execution time* et *allotment* se fait en prenant leur dernière valeur enregistrée et en y retirant l'écart entre l'instant courant et la date de la dernière mise à jour (*last_update_date*), par exemple à un instant *t* on a : $\boldsymbol{ret_i(t) = ret_i(t) - (t - last\_update\_date)}$.

Algorithm 4 U-EDF_allot_consummed

Input:
taskId := the Id of the task that consummed its allotment;
t := current time;
Data:
runningTasks := List of m running tasks;
needReschedule := boolean to indicate wether to reschedule tasks or not;

for all $\tau_i \in$ runningTasks **do**
    Update $ret_i(t)$ and $Al_{i,j}(t)$;
end for
Update the state of the task taskId (running -> ready);
needReschedule := **true**;

Algorithme 3.1.4 : Terminaison d'une tâche

Le traitement d'une terminaison de tâche est similaire à celui d'une terminaison de dotation temporelle, la seule différence réside dans la mise à jour des attributs *remaining execution time* et *allotment* de la tâche en question. En effet, les calculs des dotations temporelles faits par U-EDF se base sur la pire durée d'exécution de la tâche (*Worst-Case Execution Time, WCET*). Cette durée d'exécution, n'est pas toujours égale à la durée d'exécution réelle de la tâche. Pour cela, le traitement de la terminaison d'une tâche en U-EDF ne doit pas vérifier si la tâche s'est exécutée pour une durée égale au *WCET* ou non. Ainsi, nous avons prévu une annulation immédiate du *remaining execution time* et d'*allotment* de la tâche qui finit son exécution et non pas une mise à jour qui élimine l'écart entre l'instant courant et la date de la dernière mise à jour.

Pour les autres tâches non concernées par cet événement, le traitement des mises à jour reste le même que dans le cas d'une fin d'*allotment*.

Une fois que tous les attributs sont à jour, on passe le booléen *needReschedule* à *true* pour informer de la nécessité d'un ré-ordonnancement.

Algorithm 3 U-EDF_terminate_task

Input:
taskId := the Id of the terminated task;
coreId := the Id of the core in which the task was running
t := current time
Data:
runningTasks := List of m running tasks;
needReschedule := boolean to indicate wether to reschedule tasks or not;

Update the state of the task taskId (running -> suspended);
$ret_{taskId}(t) := 0;$
$Al_{taskId,\,coreId}(t) := 0;$
for all $\tau_i \in$ runningTasks **do**
    Update $ret_i(t)$ and $Al_{i,j}(t)$;
end for
needReschedule := **true**;

Algorithme 3.1.5 : Franchissement d'échéance

Le franchissement d'échéance est un événement qui n'engendre aucun ré-ordonnancement. En effet, le seul intérêt à repérer un tel événement est de mettre à jour l'attribut la date d'échéance absolue de la tâche en question. Cette date d'échéance entre dans des calculs de budgets temporels selon le principe de la politique U-EDF. Ainsi, une fois qu'une tâche dépasse son échéance, sa date d'échéance absolue doit être à tout moment égale à l'instant courant *t*. Comme on ne peut pas actualiser sa valeur à chaque tick d'horloge, nous proposons de mettre l'attribut de la date d'échéance à -1, et pendant les calculs (des budgets temporels) il suffit de tester pour toutes les tâches s'il est supérieur ou égal à 0. Si oui, on le remplace par la vraie valeur de l'échéance absolue de la tâche. Sinon, on le remplace par la valeur de la date courante (*cf.* figure 3-2).

Input:
taskId := the Id of the task that reached its deadline;
t := current time;

$d_{taskId}(t) = -1;$

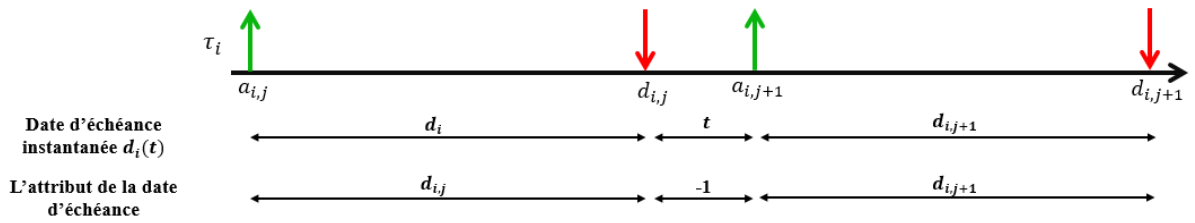


Figure 3-2 : évolution de l'attribut de la date d'échéance absolue au cours du temps

Algorithme 3.1.6 : Rescheduling

Nous avons précisé dans le 5ème constat que nous allons considérer une seule liste de tâches qui sera manipulée par l'ordonnanceur, au lieu de manipuler une liste par processeur. Cette liste regroupe les tâches prêtes. Ainsi, l'action préemption consiste à rendre une tâche à la liste des tâches prêtes, et l'action exécution consiste à la retirer de cette liste.

Il faut noter que l'algorithme de ré-ordonnancement que nous proposons, ne va être appelé que si le booléen *needReschedule* est à *true*. Ainsi, lorsqu'il est appelé, on vérifie d'abord s'il y a besoin de re-calculer les allocations de tâche via le booléen *needAllotRecompute*.

Le processus de décision d'ordonnancement qu'illustre l'algorithme ci-dessous est le suivant :

- On balaye la liste des tâches *ready* par ordre de priorité croissant. Pour chacune des tâches nous allons voir sur chaque processeur si elle éligible ($\boldsymbol{Al_{ij}(t) > 0}$) pour son exécution. Si oui, elle y sera assignée, sinon on répète le même processus jusqu'à ce qu'on trouve un processeur sur lequel elle peut s'exécuter ou jusqu'à ce que tous les processeurs soient balayés.
- Ce processus assure qu'une fois une tâche est assignée à un processeur pour son exécution, ce n'est plus la peine d'aller regarder son éligibilité sur les autres processeurs.

La séparation des événements que nous avons proposée dans les algorithmes ci-dessus permet d'alléger le traitement par l'ordonnanceur. Par exemple, si on suppose que sur une durée courte nous avons l'occurrence d'un événement de fin d'exécution suivi de deux événements successifs d'atteinte d'échéance, ça ne sera plus nécessaire de faire deux autres *rescheduling* après le traitement de la fin d'exécution, puisque de toute les manières une atteinte d'échéance ne remet pas en cause la décision d'ordonnancement prise avant.

Algorithm 6 U-EDF_scheduling

```
Input:
  t := current time
Data:
  readyTasks := List of n ready tasks sorting by increasing absolute deadline;

  if needAllotRecompute is true then
     // Recompute the tasks allotments
     Call Algorithm 1;
  end if
  for all τ_i ∈ readyTasks do
     j := 0;
     While (τ_i not assigned and j<m)
        if (processor j is free and Al_{i,j}(t) > 0) then
             Execute τ_i on processor j for Al_{i,j}(t) time units;
             j := j + 1;
        end if
     end while
  end for
```

3.2. MISE EN ŒUVRE DE LA POLITIQUE U-EDF

3.2.1. INTRODUCTION

Comme nous l'avons déjà mentionné, le RTOS OSEK (Trampoline) ne peut être que statique. C'est-à-dire que tous les paramètres et les attributs des objets OSEK (tâches, alarmes, compteurs …) doivent être configurés à l'avance, c'est-à-dire avant le démarrage du système.

Ainsi, notre objectif étant toujours d'intégrer un ordonnanceur global à Trampoline : nous allons utiliser le langage OIL pour définir la description d'une application temps réel ; renseigner les attributs des objets nécessaires ; et finalement générer les structures de données pour compiler l'application. Le programme de l'ordonnanceur va ensuite accéder à ces structures pour récupérer les données dont il a besoin pour son fonctionnement.

Afin de tester le fonctionnement de l'ordonnanceur, nous allons par la suite, développer un simulateur qui va remplir le rôle de Trampoline, et générer un scénario d'événements en se basant sur la configuration de tâches déclarée dans l'application OIL.

3.2.2. DEFINITION DES ATTRIBUTS NECESSAIRES A L'IMPLEMENTATION

Pour une implémentation d'U-EDF, il faut d'abord apporter quelques modifications au niveau des fichiers responsables de la génération des structures de données. Ainsi, nous aurons besoin d'ajouter de nouveaux attributs, qui ne figurent pas dans l'implémentation actuelle, ou de modifier le type de quelques autres déjà existants. Il est à noter que la majorité de ces modifications concerne les descripteurs de tâche, c'est parce que l'algorithme d'ordonnancement agit en fonction des attributs des tâches du système.

Le descripteur de tâche est scindé en fait en deux parties, statique et dynamique. Cela permet une exploitation plus efficace des ressources mémoires en stockant dans la ROM les attributs statiques, et en RAM les attributs dynamiques.

La démarche d'ajout d'un nouvel attribut se fait en suivant les étapes suivantes :

- Il faut commencer par déclarer les attributs des objets OSEK. Ceci se fait dans le fichier « *config.oil* » qui contient la définition d'implémentation OSEK.

- Etant donné que GOIL génère les structures de données en se basant sur des templates de configuration système (*cf.* figure 2-7), il faut ajouter les attributs des tâches dans le template responsable de la génération des descripteurs de tâches « *task_descriptor.goilTemplate* ». Les attributs statiques doivent être ajoutés dans les descripteurs statiques, et les attributs dynamiques doivent s'ajouter aux descripteurs dynamiques.

  Ces descripteurs vont être générés par la suite dans un fichier « tpl_app_config.c » que nous allons utiliser pour l'implémentation de la politique U-EDF.

- Une fois que les déclarations sont faites, on doit faire une définition d'application dans un fichier OIL, dans lequel il faut affecter des valeurs (initiales ou par défaut) aux attributs des objets OIL.

Dans notre cas, plusieurs attributs doivent être ajoutés. Nous allons dans ce qui suit citer les différentes modifications apportées à la description actuelle des attributs :

Echéance relative : c'est un attribut statique qui désigne le délai critique d'une tâche, il doit donc être déclaré dans le descripteur statique. Etant donné que nous travaillons avec des tâches à échéances implicites, cet attribut renseigne également la valeur de la période de la tâche.

Echéance absolue : il s'agit de la date d'échéance de la tâche à un instant donné, sa valeur change au cours du temps, donc c'est un attribut dynamique.

Durée d'exécution en pire cas (WCET) : c'est un attribut statique.

Durée d'exécution en meilleur cas (BCET) : c'est attribut statique qui va nous servir dans le simulateur de scénario pour pouvoir générer une durée d'exécution de tâche aléatoire, comprise entre le BCET et le WCET

Identifiant du cœur d'exécution : cet attribut existe déjà dans l'implémentation actuelle de Trampoline dans le descripteur statique. C'est parce que l'implémentation utilise un ordonnanceur partitionné et donc le cœur sur lequel la tâche s'exécute ne change pas. Cependant, dans notre implémentation il faut déplacer cet attribut dans le descripteur dynamique puisqu'on utilise une politique d'ordonnancement global.

Temps d'exécution restant : la valeur de cet attribut diminue au fur et à mesure que la tâche s'exécute. Ainsi, il sera ajouté au descripteur dynamique.

Dotation temporelle : c'est un attribut dynamique, qui va être déclaré sous forme de tableau de dimension égale au nombre de cœurs de la plateforme.

Remarque 3.2.1 :

Dans notre étude, nous avons décidé, pour des raisons de simplification, de ne pas adopter une structure de données précise pour la gestion de la liste des tâches prêtes. C'est parce qu'il est important de tester l'algorithme, dans un premier temps, avant de chercher la structures de données adéquates pour notre cas. Ainsi, nous allons récupérer les descripteurs de tâches à travers une interface qui fera l'abstraction du type de la structure de données utilisée.

3.2.3. PRESENTATION ARCHITECTURALE DE L'APPLICATION

Avant de procéder à une intégration complète au noyau de Trampoline, il est nécessaire de tester, dans un premier temps, de tester le fonctionnement de la politique d'ordonnancement en pratique. Ainsi, pour la mise en œuvre la politique U-EDF dans un contexte réel, nous allons passer par un simulateur qui permettra de tester et valider l'aspect fonctionnel de notre implémentation de la politique.

A cet effet, notre application sera constituée de deux éléments majeurs : une unité d'ordonnancement U-EDF qui se charge de faire les calculs nécessaires pour décider l'exécution des tâches ; et une unité de simulation qui modélise l'écoulement du temps, génère des occurrences d'événements, et exécute les décisions d'ordonnancement fournies par l'ordonnanceur. La relation entre les deux unités est une relation de commande/information, telle que l'ordonnanceur commande le simulateur et ce dernier l'informe de l'évolution d'exécution (*cf.* figure 3-3).

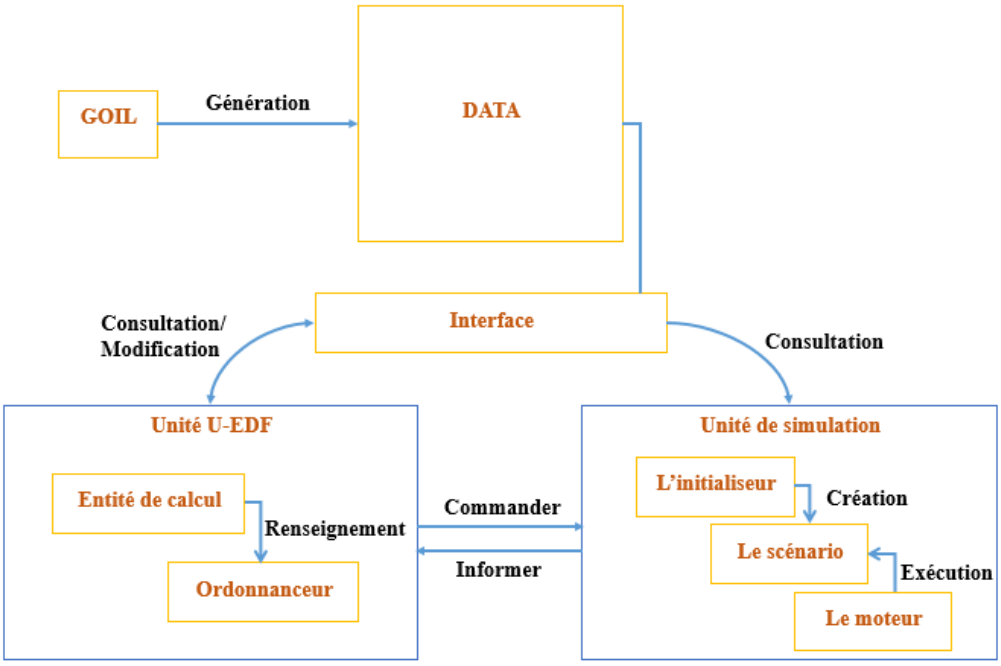


Figure 3-3 : architecture de l'application

Unité de simulation : dans notre implémentation, le simulateur va se charger d'exécuter les tâches et positionner les événements selon les commandes de l'unité d'ordonnancement, et finalement signaler les occurrences d'événements au cours du temps.

L'unité de simulation est composée de 3 éléments (*cf.* figure 3-4):

- *L'initialiseur :* cet élément s'occupe de la création d'un scénario sur une durée de simulation calculée. Cette durée est obtenue en calculant l'hyper-période de l'ensemble des tâches de l'application[20]. Ainsi, on s'assure que la simulation regroupe toutes les situations d'ordonnancement possibles, étant donné que la même séquence d'exécution sera répétée toutes les hyper-périodes. Pour cela, le simulateur doit avoir accès aux descripteurs de tâche générés par GOIL pour disposer des périodes des tâches.

[20] Nous allons supposer que toutes les tâches sont synchrones.

L'initialiseur se charge également de l'initialisation du scénario en positionnant, sur toute la fenêtre temporelle de simulation, les dates de réveil des tâches à partir de l'attribut « échéance relative ». Comme nous souhaitons également traiter le cas des tâches sporadiques, nous nous sommes servis d'une fonction qui génère aléatoirement une durée entre deux réveils successifs comprise entre $\boldsymbol{k\_min \times période\_de\_la\_tâche}$, et $\boldsymbol{k\_max \times période\_de\_la\_tâche}$, tel que $\mathbf{1 < k\_min < k\_max} \setminus \boldsymbol{k\_min, k\_max} \in \mathbb{Z}$.

Ensuite, à partir de chaque réveil de tâche positionné, on place un événement de « *franchissement d'échéance* ».

Il est à noter que si on utilise un modèle de tâches sporadiques, l'hyper-période ne sera plus la même. Dans ce cas, on choisit une durée de simulation totale égale au plus petit commun multiple entre toutes les périodes des tâches du système multiplié par $\boldsymbol{k\_max}$ (ie. $\boldsymbol{durée\_de\_simulation = k\_max \times ppcm(P_1, P_2, \ldots, \times P_N)}$.

- *Le scénario :* il s'agit d'un scénario qui regroupe 4 tableaux de taille égale à la fenêtre temporelle de simulation. Ces 4 tableaux correspondent aux 4 événements de ré-ordonnancement. Ainsi, si à un instant *t* nous avons un événement de type « *réveil de tâche* » on le trouvera dans le tableau correspondant positionné dans la case d'indice égal à *t*. Si au même instant, on a l'occurrence de plusieurs événements, ils seront stockés selon leurs types, en liste chaînée.

  Le scénario contient également des fonctions qui permettent au moteur de simulation et à l'initialiseur d'y insérer un événement, de le récupérer, de marquer qu'il est exécuté ou bien de le retirer.

- *Le moteur de simulation :* c'est l'élément qui se charge de la communication avec l'ordonnanceur. Il balaye instant par instant le scénario, récupère à chaque instant les événements positionnés dans le scénario (s'ils existent), les exécute, invoque l'ordonnanceur en le signalant les événements occurrents selon le type (s'ils existent) et insère les nouveaux événements renseignés par l'unité d'ordonnancement.

  Etant donné que les événements de type « *réveil de tâche* » et « *franchissement d'échéance* » sont positionnés au début par l'initialiseur, il ne reste pour le moteur que les événements de type « *terminaison de tâche* » et « *consommation de dotation* » à placer dans le scénario. Ainsi, une fois qu'il reçoit, à un instant *t*, l'ordre d'exécuter une tâche $\boldsymbol{\tau_i}$ pour une durée de dotation égale à $\boldsymbol{Al_{i,j}(t)}$, il positionne un événement de type « *consommation de dotation* » à l'instant $\boldsymbol{t + Al_{i,j}(t)}$ et un autre événement de type « *terminaison de tâche* » à l'instant $\boldsymbol{t + ret_i(t)}$ en positionnant des événements sur les deux dates, ce qui correspond à armer des timers. Lorsque la tâche a consommé sa dotation temporelle alors que son temps d'exécution restant n'est toujours pas nul, on retire l'événement « *terminaison de tâche* » positionné, jusqu'à une prochaine exécution.

  Le moteur de simulation assure également une gestion individuelle de la durée d'exécution de la tâche. En effet, à partir des attributs « *WCET* » et « *BCET* », il choisit aléatoirement une valeur de durée d'exécution comprise entre les deux. Ceci se fait lors du traitement d'un « *réveil de tâche* ». Ainsi, le temps d'exécution restante que le simulateur gère ($\boldsymbol{ret_i(t)}$) dépend de la valeur choisie.

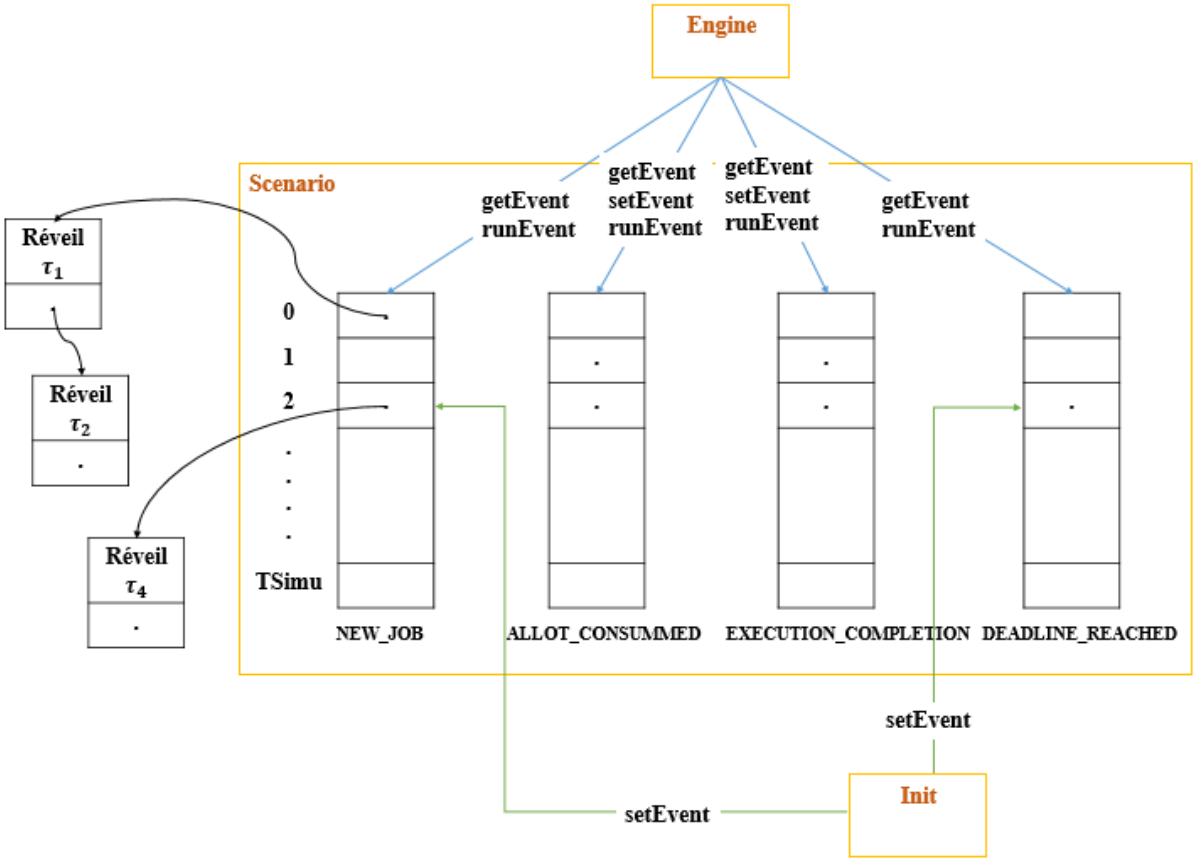


Figure 3-4 : principe général du simulateur

Interface : il s'agit de l'interface qui récupère les structures de données fournies par GOIL et dépendant de l'OS Trampoline. Elle dispose des fonctions qui permettent de récupérer les listes des tâches souhaitées, triées ou non, soit selon leurs échéances, ou bien leurs états. L'accès à cette interface peut se faire soit en lecture (récupération des listes) ou bien en écriture (modification des attributs des tâches). L'interface permet également de récupérer une liste des tâches plus prioritaires qu'une tâche donnée, ce qui servira dans les calculs faits par l'unité de calcul U-EDF.

Unité d'ordonnancement : l'unité d'ordonnancement peut être invoquée si l'un des 4 événements énumérés dans la section 3.2.1 a fait son occurrence. Cette unité décide de passer chacune des tâches de l'application d'un état à un autre, selon les informations récupérées de l'unité de simulation (Trampoline) et les calculs fournis par la politique d'ordonnancement. Dans notre implémentation, cette unité se compose de 2 éléments :

- *Entité de calcul d'U-EDF :* ce composant s'occupe de tous les calculs que doit faire l'algorithme U-EDF : les calculs des budgets temporels, des allocations de travaux actuels, des réservations des travaux futurs et des taux d'utilisation instantanés selon le principe de McNaughton. En outre, l'unité de calcul assure les mises à jour nécessaires des différents attributs, selon le type d'événement appelant l'ordonnanceur. Pour cela, elle se charge de gérer les attributs des tâches selon les occurrences d'événement (s) que signale le simulateur.

  Cette unité gère également le booléen « *needReschedule* » qui indique s'il est nécessaire de faire un ré-ordonnancement de tâche ou non.

- *Ordonnanceur :* il s'agit de l'entité centrale de l'unité d'ordonnancement. Il s'occupe des choix d'exécution des tâches sur les processeurs selon le principe d'U-EDF. Pour faire, il récupère les calculs faits par l'unité de calcul et les listes des tâches prêtes, ensuite sélectionne, sur chaque processeur libre, la tâche la plus prioritaire parmi les tâches éligibles pour l'exécuter. Il accède également aux structures de données fournies à travers l'interface, pour changer les attributs des tâches à exécuter et les attributs des processeurs qui deviennent occupés.

  Une fois que la décision d'ordonnancement est prise, l'ordonnanceur commande le simulateur pour positionner les événements de type « *dotation consommée* » et « *terminaison de tâche* » et armer des timers sur des dates données selon les allocations faites par U-EDF.

3.2.4. PRESENTATION FONCTIONNELLE DE L'APPLICATION

Dans cette section, nous allons présenter les enchainements et échanges entre les différentes unités présentées dans la section précédente.

Pour cela, nous allons modéliser l'architecture de l'application, les différents appels de fonctions et instanciations d'objets à travers trois diagrammes : diagramme des classes, diagramme d'activités et diagramme de séquence[21].

Il est à noter que nous n'avons pas fait de la programmation orientée objet

- *Diagramme des classes :* dans ce diagramme nous identifions, pour chaque module de l'application, la classe qui l'implémente avec l'ensemble des opérations et attributs qu'il utilise (*cf.* figure 3-5).

  D'un côté, nous avons le simulateur qui est modélisé par les classes « *init* » « *engine* » et « *scenario* », elles correspondent respectivement aux modules initialiseur, moteur de simulation et scénario.

  Du côté de l'unité d'ordonnancement, elle se compose des classes « uedf_*manipulation* », « *interface* » et « *scheduler* », qui modélisent respectivement les modules : unité de calcul, interface et ordonnanceur.

  La classe « *engine* » dépend de la classe « *scenario* » car elle enregistre les listes d'événements à exécuter, et de la classe « *scheduler* » puisqu'il s'agit du module qui lui commande de positionner les événements.

  L'interface est utilisée par la majorité des autres classes, étant donné qu'elle fournit les structures de données dépendantes de l'OS, comme nous l'avons déjà précisé.

- *Diagramme d'activités :* ce diagramme illustre l'ensemble des activités exécutées par les différents modules de l'application (*cf.* figure 3-6). Ces activités sont détaillées dans la section 3.2.3.

- *Diagramme de séquence :* le diagramme de séquence proposé représente les échanges entre les classes (*cf.* figure 3-7).

[21] Il est à noter que nous n'avons pas fait de la programmation orientée objet. Toutefois, nous nous sommes servis des diagrammes de modélisation afin de simplifier la compréhension de l'architecture de l'application ainsi des interactions entre ses composants.

3.3. RESULTATS DE L'EXECUTION

3.3.1. EXEMPLE DE TEST

Afin de tester l'implémentation de la politique U-EDF, nous proposons une application composée de 4 tâches périodiques à échéances implicites, pour être exécutée sur une plateforme constituée de deux processeurs. La configuration des 4 tâches est la suivante :

Tableau 3-1 : configuration de l'application

Tâche	Durée d'exécution (ms)	Période (ms)	Taux d'utilisation
$\boldsymbol{\tau_0}$	500	1000	½
$\boldsymbol{\tau_1}$	3000	5000	3/5
$\boldsymbol{\tau_2}$	1000	2000	½
$\boldsymbol{\tau_3}$	500	2500	1/5

Le taux d'utilisation total est égal à $\mathbf{1,8} < \boldsymbol{nombre\ de\ processeurs}$. La configuration est donc ordonnançable selon la politique U-EDF. En outre, la période d'étude est égale à l'hyper-période, $\boldsymbol{H = ppcm(\tau_0, \tau_1, \tau_2, \tau_3) = 10000}$.

La séquence d'ordonnancement obtenue en calculant les allocations selon le principe d'U-EDF est présentée sur la figure 3-8.

En simulant le fonctionnement d'U-EDF à l'aide de simulateur que nous avons élaboré, nous sommes arrivé au même résultat de la séquence d'ordonnancement ci-dessous. Les résultats d'exécution de l'ordonnanceur sont regroupés en Annexe A.

Nous avons mené également un jeu d'essais pour tester l'unité d'ordonnancement toute seule, avec de systèmes composés de 3 à 6 tâches sur nombre de processeur variable entre 2 à 4 processeurs. L'unité proposée est arrivée à ordonnancer correctement les configurations de tâches qu'on lui a fourni.

Nous avons également testé l'application élaborée avec deux autres configurations de tâches, mais toujours sur deux processeurs. Les résultats fournis sont toujours adéquate avec les séquences d'ordonnancement fourni en utilisant le calcul théorique que propose la politique U-EDF. Cependant, nous n'avons pas eu le temps pour aller plus loin avec les tests.

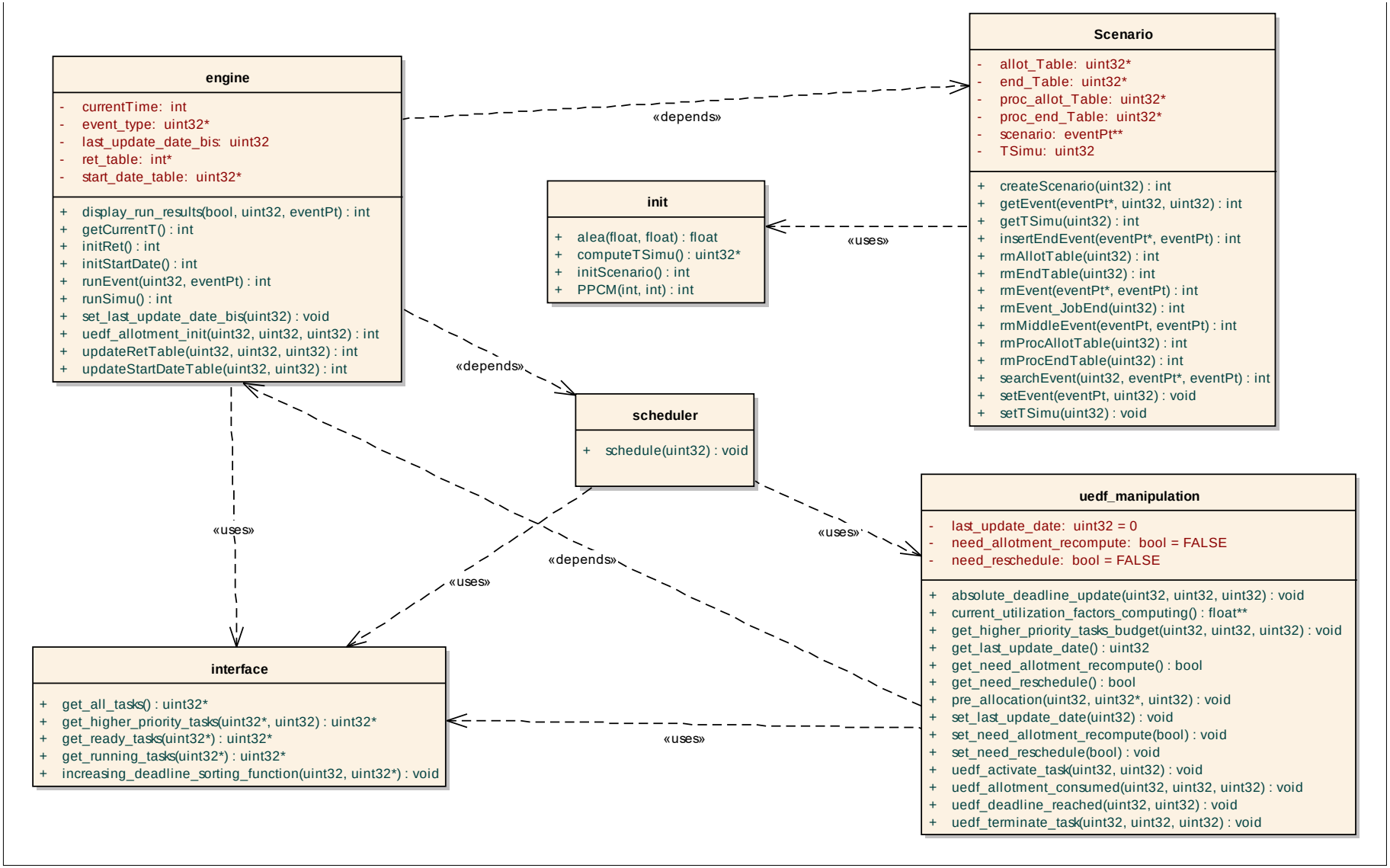


Figure 3-5 : diagramme des classes de l'application

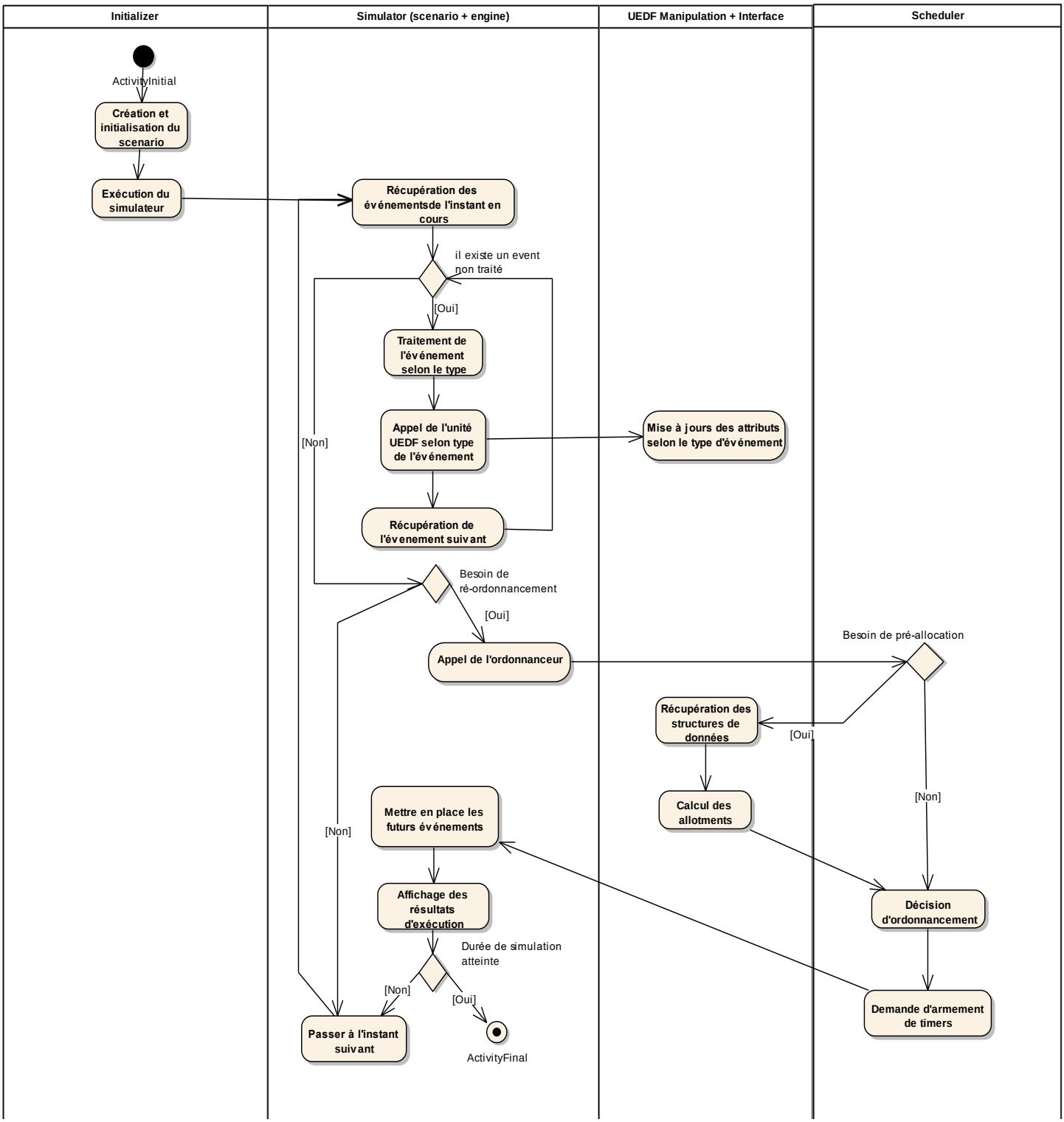


Figure 3-6 : diagramme des activités

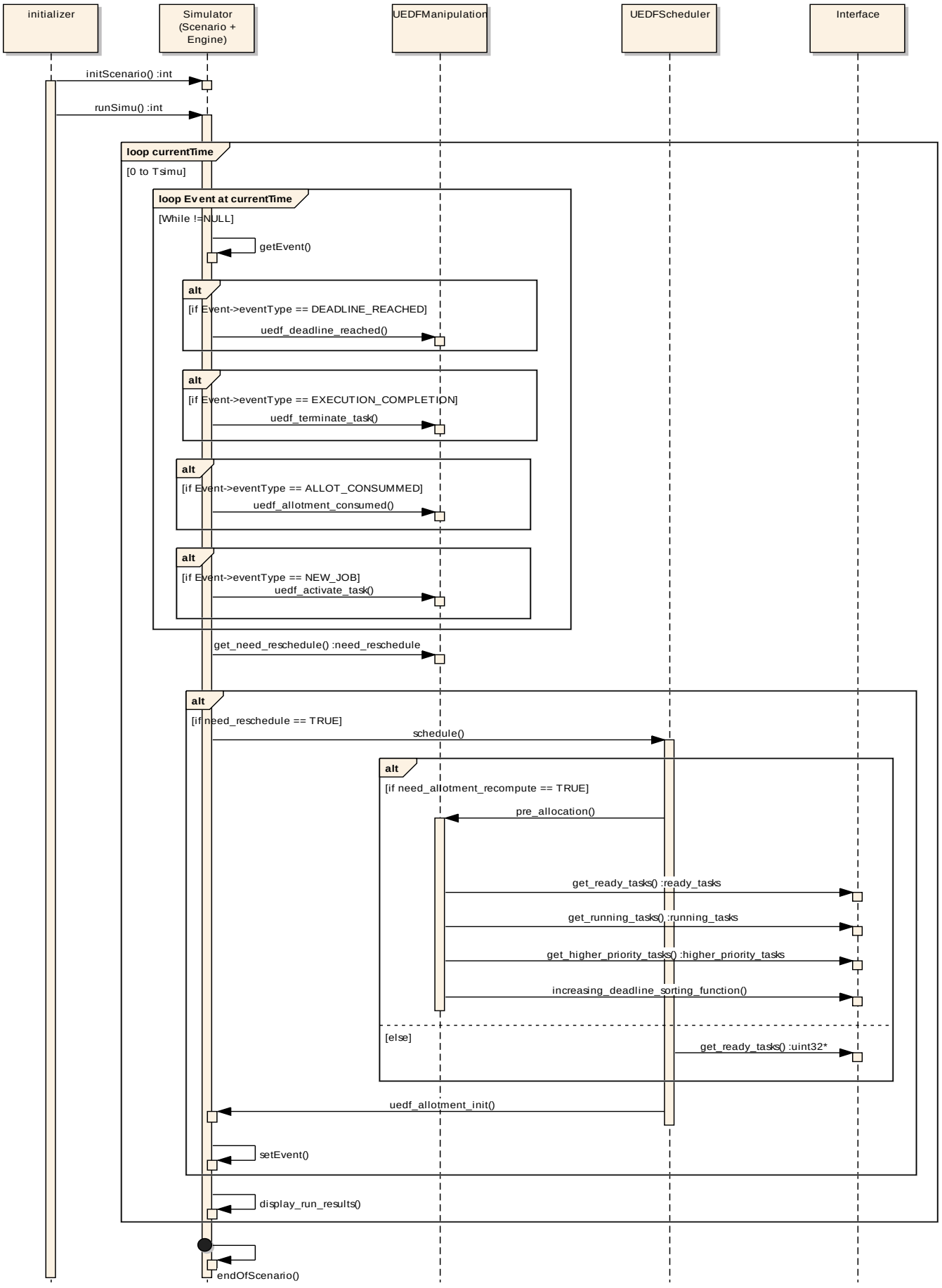


Figure 3-7 : Diagramme de séquence

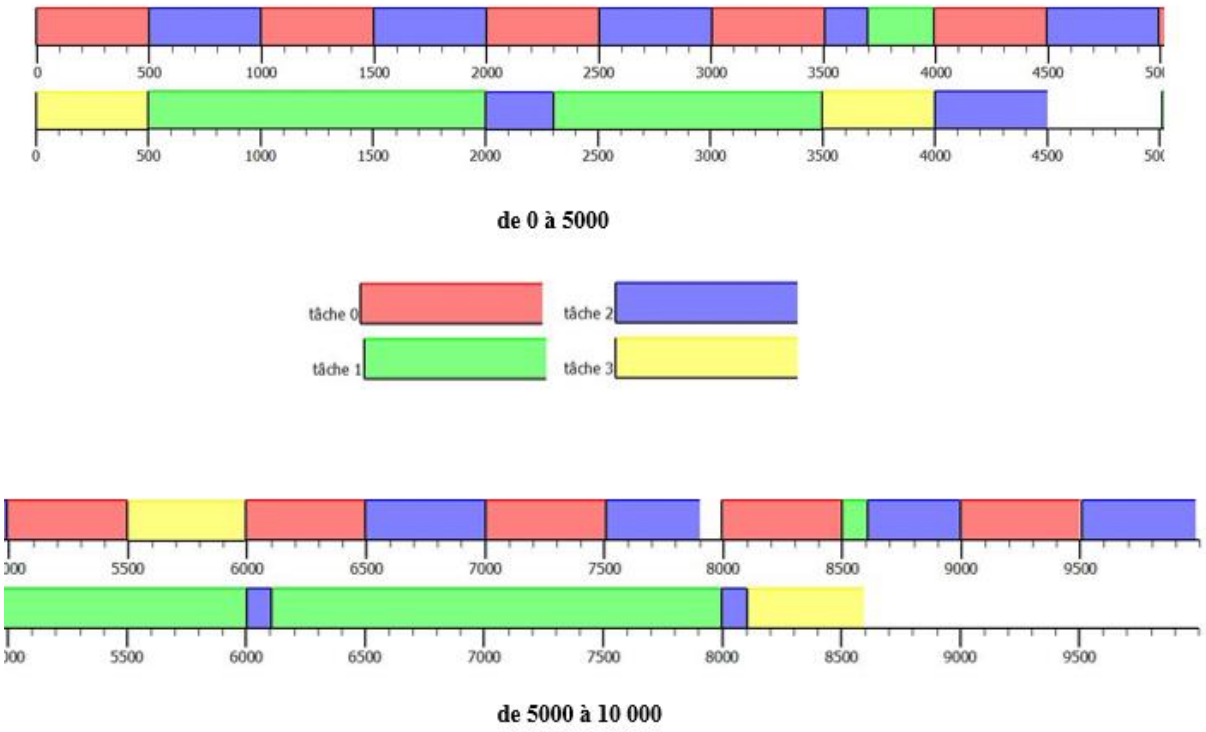


Figure 3-8 : séquence d'ordonnancement selon U-EDF

3.3.2. DIFFICULTES CONSTATEES

Le test du fonctionnement de l'algorithme avec plusieurs scénarios nous a permis d'identifier deux situations délicates, qui limite l'efficacité maximum de l'implémentation de la politique. Dans cette section nous allons résumer les situations observées.

Situation 1 : la difficulté la plus importante rencontrée réside dans la garantie de la simultanéité de traitement des événements qui arrivent en même temps. En effet, si plusieurs événements sont simultanés, la politique U-EDF nécessite de les connaître tous à l'instant où ils arrivent, avant de prendre la décision d'ordonnancement. Si cette condition n'est pas satisfaite, l'ordonnanceur peut fournir des résultats erronés.

Prenons, par exemple, deux tâches T4 et T3 qui s'exécutent respectivement sur deux processeurs P1 et P2, et on estime que leur fin de dotations temporelles survient à la même date (t = 10), qui est une date de ré-ordonnancement. Supposant, selon la politique U-EDF, qu'après t = 10, T3 doit poursuivre son exécution sur P1 et T4 sur P2 (*cf.* figure 3-8). On peut distinguer deux approches pour le traitement des événements :

- *Approche globale :* qui consiste à mettre à jour les attributs des tâches successivement, et passer leurs états à « *Ready* », et ne prendre la décision d'exécution qu'après le traitement des deux événements. Ainsi dans notre exemple, quand l'ordonnanceur va regarder la liste des tâches prêtes, pour choisir les tâches les plus prioritaires et éligibles sur chaque processeur, il pourra sélectionner T3 et T4.

- *Approche unitaire :* cette approche consiste à prendre la décision d'ordonnancement après le traitement de chaque occurrence d'événement. Si on procède ainsi pour l'exemple ci-dessus, on va d'abord traiter l'événement de fin de dotation de la tâche T4, sans savoir que T3 a aussi fini sa dotation. Ainsi, étant donné que pour l'ordonnanceur, T3 est toujours en exécution, il

ne pourra pas la sélectionner pour s'exécuter sur P1. Autrement dit, il va sélectionner une autre tâche à sa place, ce qui ne permet pas d'aboutir au résultat exact.

En outre, lorsqu'on va aller traiter la fin de dotation de la tâche T3, comme elle devait poursuivre son exécution sur P1 (qui n'est plus libre), il est possible qu'elle ne trouve pas un autre processeur pour s'y exécuter et potentiellement dépasser son échéance.

⇨ Par conséquent, il est important de garder l'approche globale dans le traitement des événements.

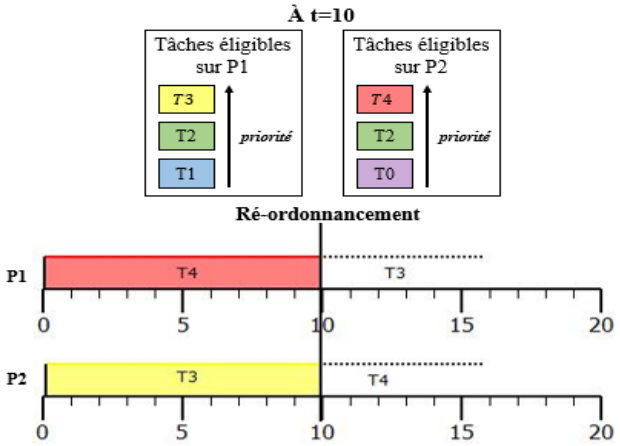


Figure 3-9 : exemple d'exécution

D'ailleurs, avec le simulateur proposé, le traitement global est assuré étant donné que les événements sont regroupés dans des tableaux (selon leurs types), et donc si on souhaite traiter les événements dont l'occurrence surgit à un instant *t*, il suffit d'aller regarder les cases d'indice *t*, les récupérer et traiter les événements un par un, puis prendre la décision au final.

Ceci n'est pas facile à réaliser si on veut intégrer les programmes de l'ordonnanceur dans un contexte réel. C'est parce que les événements sont liés à différents timers comptant jusqu'à la date de leurs occurrences et envoyant une interruption une fois qu'ils expirent. Quoique deux timers puissent avoir la même date d'expiration, leurs interruptions ne peuvent pas être traitées par le code d'ordonnanceur au même instant. D'ailleurs une fois qu'une interruption est reçue par l'ordonnanceur, on n'est pas sûr si on va recevoir une autre à l'instant près, ce qui fait l'ordonnanceur ne peut pas ralentir le traitement jusqu'à ce qu'il vérifie s'il y a d'autres événements simultanés.

Le traitement global de deux interruptions ne peut se faire que si elles sont reliées au même timer. Cependant, on ne peut pas relier toutes les interruptions à un seul timer, étant donné qu'il ne peut pas gérer les différents types d'événements.

Situation 2 : lorsqu'on a adopté le traitement global des événements simultanés par le simulateur, nous avons constaté que l'ordre de traitement de ces événements peut mettre en cause le calcul des allocations des tâches ou changer la séquence d'ordonnancement.

Par exemple on suppose qu'on a un événement de type « *franchissement d'échéance* » qui arrive simultanément avec un événement de type « *réveil de tâche* ». Si on commence par traiter ce dernier événement et donc par calculer les dotations, on ne va pas s'apercevoir qu'il y a une tâche qui est arrivée à son échéance et est devenue « *inactive* ». Or, il le faut, car cette même tâche doit être considérée hautement prioritaire dans les

calculs des budgets temporels des tâches prioritaires, et qui entrent dans le calcul des allocations temporelles des tâches « *actives* ». Ainsi, il est important de traiter l'événement « *franchissement d'échéance* » avant celui de « *réveil de tâche* ».

De même, si nous avons des événements de types « *terminaison de tâche* » ou « *consommation de dotation* » surgissant au même instant avec un « *réveil de tâche* ». L'ordonnanceur doit s'en apercevoir et avoir une valeur correcte du temps d'exécution restant, pour ne pas allouer du temps processeur à des tâches qui ont fini leur exécution.

Afin de remédier à cette situation, nous avons décidé de commencer le traitement des événements de type « *franchissement d'échéance* » en premier, suivi de ceux de type « *terminaison de tâche* », puis des « *consommation de dotation* » et finalement des « *réveil de tâche* ». Nous avons avancé le traitement d'une fin d'exécution, car si à un instant donné nous avons une fin d'exécution et une fin de dotation temporelle pour la même tâche, il sera plus simple de la passer directement à l'état « *Suspended* » au lieu de la passer par l'état « *Ready* ».

Ce tri d'événements, quoique facile à mettre en œuvre avec un simulateur, n'est toujours pas possible dans un contexte réel, vu que l'ordonnanceur ne peut pas attendre l'arrivée (éventuelle) de tous les événements simultanés, faire le tri et puis les traiter. Comme nous l'avons mentionné, cela va ralentir le fonctionnement de l'ordonnanceur et donc mettre en cause son efficacité.

3.4. CONCLUSION

Au cours de ce chapitre, nous avons présenté notre mise en œuvre de la politique U-EDF, en se basant sur des structures de données dépendantes de l'OS Trampoline. Pour cela, une précision des attributs de tâches a été indispensable. Une telle implémentation devait aussi passer par une analyse fonctionnelle, dans laquelle nous avons identifié les composants de l'application et les différentes interactions entre eux.

Enfin, afin de pouvoir tester la validité fonctionnelle de notre implémentation, nous nous développé un simulateur qui remplace Trampoline. Finalement, après avoir testé l'implémentation avec plusieurs scénarios, il s'est avéré que, bien que l'algorithme arrive à calculer parfaitement les dotations temporelles, et donc aboutir à la même séquence d'ordonnancement obtenue par des calculs théoriques, il exige certains traitements d'événements d'ordonnancement qui sont difficiles à assurer dans un contexte réel.

Conclusion générale

Dans ce travail de recherche, nous avons abordé la problématique de l'implémentation d'un ordonnanceur global au sein d'un système d'exploitation temps réel OSEK. Après avoir mené une étude approfondie sur le principe de la nouvelle politique U-EDF proposée par G.Nelissen, qui se base sur le célèbre algorithme d'ordonnancement monoprocesseur EDF, et qui est spécialement dédié à l'ordonnancement des tâches sporadiques en multiprocesseur, nous nous sommes intéressés à sa mise en œuvre au sein d'un RTOS développé au sein de l'équipe "Systèmes Temps Réel" de l'IRCCyN. L'algorithme U-EDF

Dans le premier chapitre, nous avons élaboré le cadre théorique et définit l'ensemble des modèles et notions nécessaires pour notre étude. Ainsi, nous avons présenté le classement des politiques d'ordonnancement temps réel, survolé quelques politiques en ordonnancement global. Ensuite nous avons présenté différentes politiques d'ordonnancement multiprocesseur, dont certains ont prouvé leur optimalité dans la littérature d'ordonnancement, tels que BFair et DP-WRAP qui se basent sur le principe de l'équité d'exécution entre les tâches sur les processeurs de la plateforme. Dans la deuxième partie du chapitre, nous avons dévoilé le principe de fonctionnement de la politique U-EDF. Ce principe qui consiste à attribuer, pour chaque nouvelle tâche qui vient de rejoindre le système, une durée d'exécution sur les processeurs pour son travail actif (dotation, allotment ou allocation), en plus d'une réservation pour ses jobs qui peuvent se réveiller après son échéance. Pour ordonnancer les tâches, U-EDF utilise une variante d'EDF appelée EDF-D, qui permet d'éviter le parallélisme d'exécution d'un même job.

Le deuxième chapitre était dédié à l'étude de la plateforme d'implémentation de la politique U-EDF. Pour cela, nous avons commencé par détailler les différents objets du standard OSEK, à savoir les tâches, les événements, les interruptions etc… Ensuite, nous avons présenté l'architecture de Trampoline, l'exécutif temps réel qui se base sur le standard OSEK, et sur lequel nous avons étudié l'*implémentabilité* d'U-EDF. Nous avons également donné les démarches et les outils nécessaires pour la mise en œuvre d'une application temps réel en Trampoline.

Le troisième chapitre est le cœur du présent travail. Nous avons adopté une approche progressive en commençant d'abord par préciser les modifications apportées à l'algorithme U-EDF pour une implémentation facile en Trampoline. Ainsi, nous avons proposé de séparer le traitement selon l'événement surgissant, et de faire les mises à jour nécessaires pour les attributs des tâches à chaque appel d'ordonnanceur. Ensuite, nous avons identifié les différents attributs que la politique U-EDF requiert. Dans notre étude de la faisabilité de cette implémentation, nous avons décidé de récupérer les structures de données dépendant de Trampoline et de les accompagner aux programmes de l'ordonnanceur pour le tester en situations simulées, sans avoir recours à une intégration directe dans l'OS. L'efficacité au maximum de la solution n'étant pas visée, nous avons travaillé avec des structures simples pour la gestion des données et des listes de tâches. Pour le test de l'algorithme, nous avons élaboré un simulateur, qui fournit des scénarios de test dépendant de la configuration des tâches du système.

Afin de tester les programmes d'ordonnanceur élaborés, nous avons développé quelques scénarios, à travers lesquels nous avons constaté que l'unité d'ordonnancement aboutit aux mêmes résultats obtenus en calculant théoriquement les séquences d'ordonnancement selon la politique U-EDF. Cependant, grâce aux multiples simulations menées, nous avons constaté quelques situations susceptibles de limiter l'efficacité de la politique en cas d'implémentation dans un contexte réel. Ces

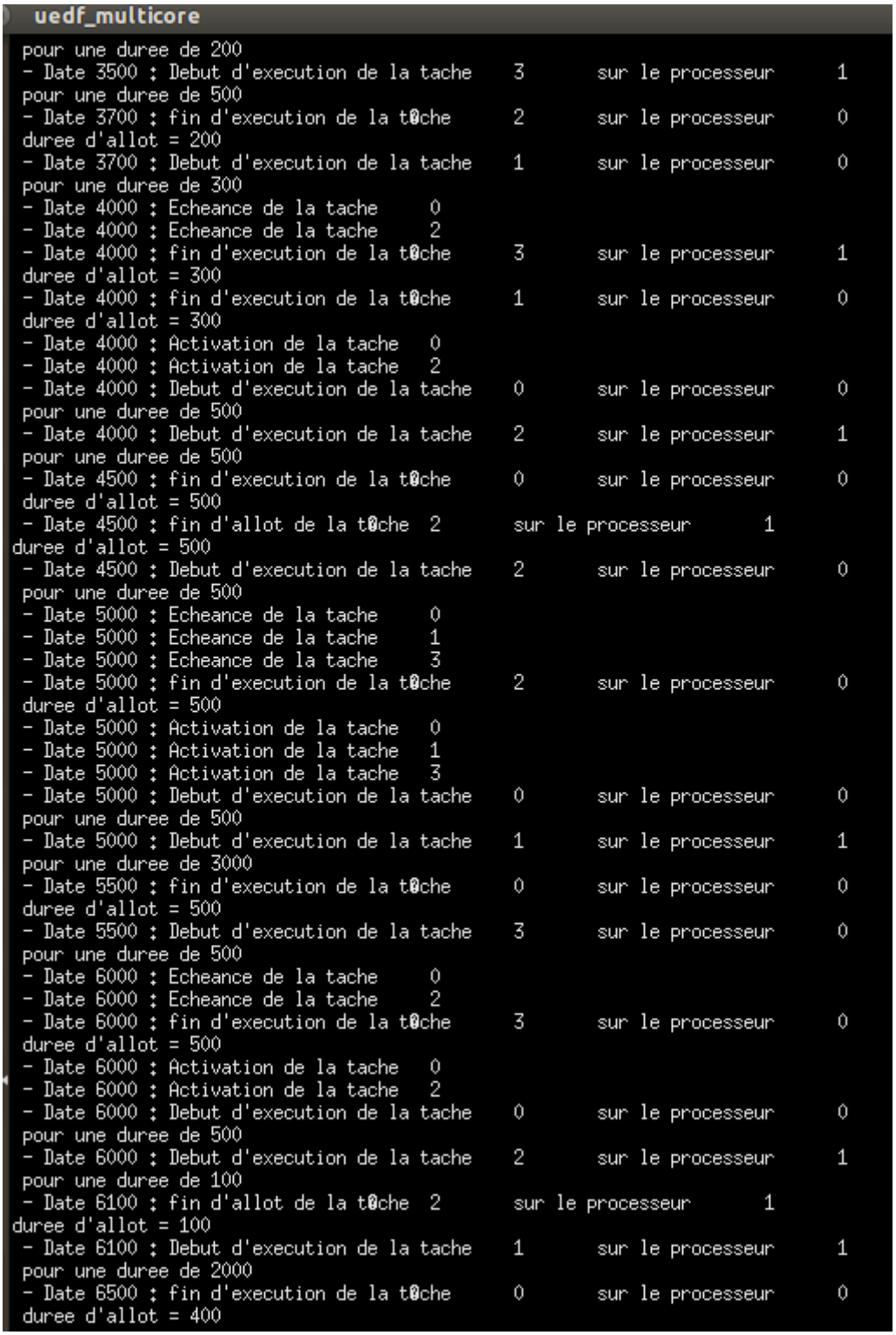
situations soulignent le cas de traitement de plusieurs événements simultanés. En effet, la politique U-EDF, étant une politique complexe, exige la connaissance de tous les événements qui arrivent en même temps avant de prendre la décision d'ordonnancement. Or, ceci n'est pas facile, étant donné que dans un contexte réel, si les événements ne sont pas reliés au même timer, on ne peut pas recevoir leurs interruptions au même instant. Ce problème peut influencer l'exactitude des calculs faits par U-EDF.

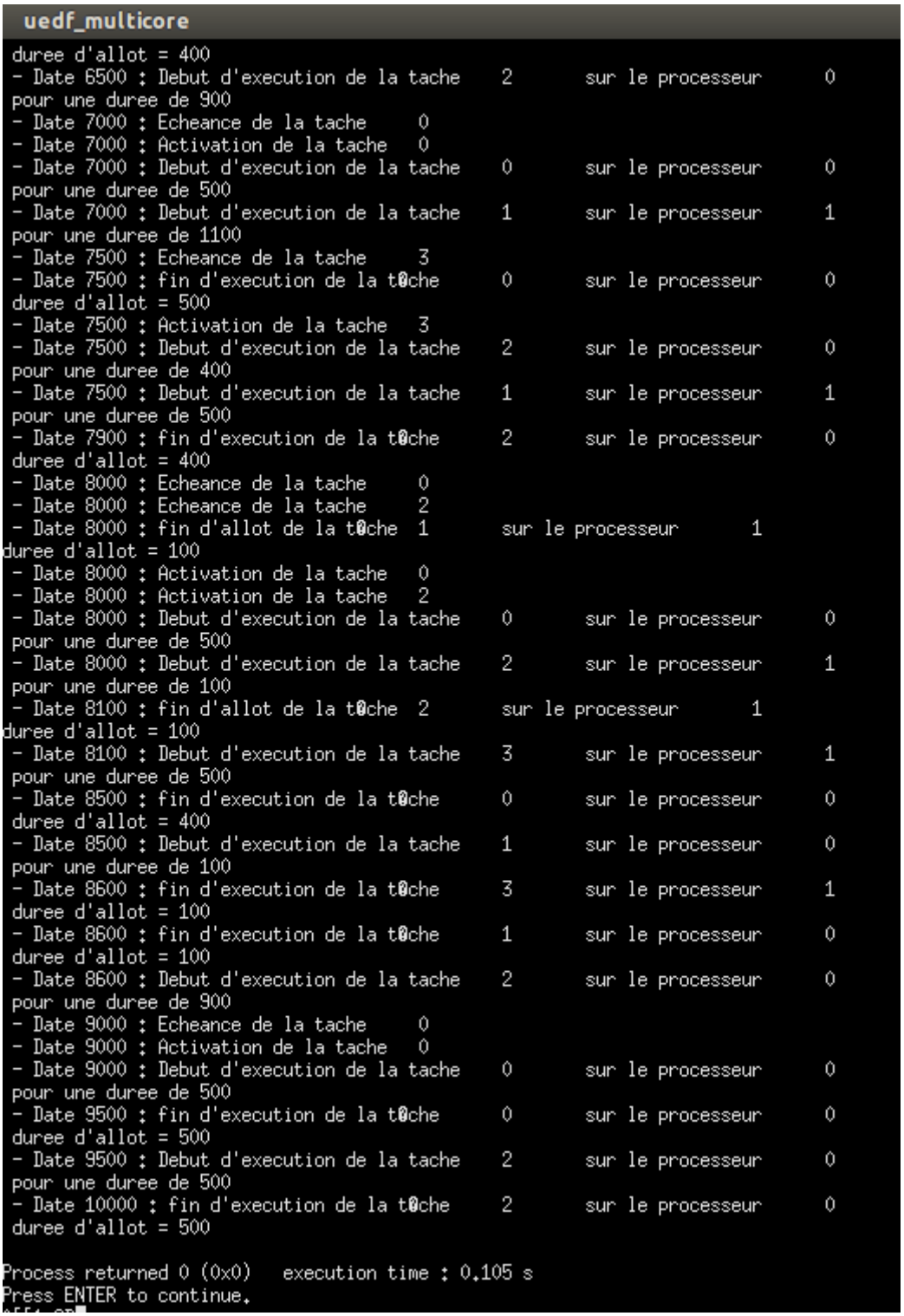
A l'issue de ce travail de recherche, nous nous fixons un objectif qui vise à poursuivre plus avant et de manière approfondie l'étude de *l'implémentatbilité* des politiques d'ordonnancement globales, de manière à ce qu'on étudie profondément la complexité induite par les activités simultanées citées ci-dessous, et à ce qu'on relève par la suite les hypothèses simplificatrices que nous avons retenus jusqu'ici.

Annexes :

Annexe A : Sequence d'execution des taches selon la politique U-EDF

```
- Date 0 : Activation de la tache      0
- Date 0 : Activation de la tache      1
- Date 0 : Activation de la tache      2
- Date 0 : Activation de la tache      3
- Date 0 : Debut d'execution de la tache       0       sur le processeur      0
pour une duree de 500
- Date 0 : Debut d'execution de la tache       3       sur le processeur      1
pour une duree de 500
- Date 500 : fin d'execution de la t0che       0       sur le processeur      0
duree d'allot = 500
- Date 500 : fin d'execution de la t0che       3       sur le processeur      1
duree d'allot = 500
- Date 500 : Debut d'execution de la tache     2       sur le processeur      0
pour une duree de 1000
- Date 500 : Debut d'execution de la tache     1       sur le processeur      1
pour une duree de 3000
- Date 1000 : Echeance de la tache     0
- Date 1000 : Activation de la tache   0
- Date 1000 : Debut d'execution de la tache    0       sur le processeur      0
pour une duree de 500
- Date 1000 : Debut d'execution de la tache    1       sur le processeur      1
pour une duree de 2500
- Date 1500 : fin d'execution de la t0che      0       sur le processeur      0
duree d'allot = 500
- Date 1500 : Debut d'execution de la tache    2       sur le processeur      0
pour une duree de 500
- Date 2000 : Echeance de la tache     0
- Date 2000 : Echeance de la tache     2
- Date 2000 : fin d'execution de la t0che      2       sur le processeur      0
duree d'allot = 500
- Date 2000 : Activation de la tache   0
- Date 2000 : Activation de la tache   2
- Date 2000 : Debut d'execution de la tache    0       sur le processeur      0
pour une duree de 500
- Date 2000 : Debut d'execution de la tache    2       sur le processeur      1
pour une duree de 300
- Date 2300 : fin d'allot de la t0che  2       sur le processeur       1
duree d'allot = 300
- Date 2300 : Debut d'execution de la tache    1       sur le processeur      1
pour une duree de 1500
- Date 2500 : Echeance de la tache     3
- Date 2500 : fin d'execution de la t0che      0       sur le processeur      0
duree d'allot = 200
- Date 2500 : Activation de la tache   3
- Date 2500 : Debut d'execution de la tache    2       sur le processeur      0
pour une duree de 700
- Date 2500 : Debut d'execution de la tache    1       sur le processeur      1
pour une duree de 1000
- Date 3000 : Echeance de la tache     0
- Date 3000 : Activation de la tache   0
- Date 3000 : Debut d'execution de la tache    0       sur le processeur      0
pour une duree de 500
- Date 3000 : Debut d'execution de la tache    1       sur le processeur      1
pour une duree de 500
- Date 3500 : fin d'execution de la t0che      0       sur le processeur      0
duree d'allot = 500
- Date 3500 : fin d'allot de la t0che  1       sur le processeur       1
```


uedf_multicore
pour une duree de 200
- Date 3500 : Debut d'execution de la tache 3 sur le processeur 1
pour une duree de 500
- Date 3700 : fin d'execution de la t0che 2 sur le processeur 0
duree d'allot = 200
- Date 3700 : Debut d'execution de la tache 1 sur le processeur 0
pour une duree de 300
- Date 4000 : Echeance de la tache 0
- Date 4000 : Echeance de la tache 2
- Date 4000 : fin d'execution de la t0che 3 sur le processeur 1
duree d'allot = 300
- Date 4000 : fin d'execution de la t0che 1 sur le processeur 0
duree d'allot = 300
- Date 4000 : Activation de la tache 0
- Date 4000 : Activation de la tache 2
- Date 4000 : Debut d'execution de la tache 0 sur le processeur 0
pour une duree de 500
- Date 4000 : Debut d'execution de la tache 2 sur le processeur 1
pour une duree de 500
- Date 4500 : fin d'execution de la t0che 0 sur le processeur 0
duree d'allot = 500
- Date 4500 : fin d'allot de la t0che 2 sur le processeur 1
duree d'allot = 500
- Date 4500 : Debut d'execution de la tache 2 sur le processeur 0
pour une duree de 500
- Date 5000 : Echeance de la tache 0
- Date 5000 : Echeance de la tache 1
- Date 5000 : Echeance de la tache 3
- Date 5000 : fin d'execution de la t0che 2 sur le processeur 0
duree d'allot = 500
- Date 5000 : Activation de la tache 0
- Date 5000 : Activation de la tache 1
- Date 5000 : Activation de la tache 3
- Date 5000 : Debut d'execution de la tache 0 sur le processeur 0
pour une duree de 500
- Date 5000 : Debut d'execution de la tache 1 sur le processeur 1
pour une duree de 3000
- Date 5500 : fin d'execution de la t0che 0 sur le processeur 0
duree d'allot = 500
- Date 5500 : Debut d'execution de la tache 3 sur le processeur 0
pour une duree de 500
- Date 6000 : Echeance de la tache 0
- Date 6000 : Echeance de la tache 2
- Date 6000 : fin d'execution de la t0che 3 sur le processeur 0
duree d'allot = 500
- Date 6000 : Activation de la tache 0
- Date 6000 : Activation de la tache 2
- Date 6000 : Debut d'execution de la tache 0 sur le processeur 0
pour une duree de 500
- Date 6000 : Debut d'execution de la tache 2 sur le processeur 1
pour une duree de 100
- Date 6100 : fin d'allot de la t0che 2 sur le processeur 1
duree d'allot = 100
- Date 6100 : Debut d'execution de la tache 1 sur le processeur 1
pour une duree de 2000
- Date 6500 : fin d'execution de la t0che 0 sur le processeur 0
duree d'allot = 400


uedf_multicore
duree d'allot = 400
- Date 6500 : Debut d'execution de la tache 2 sur le processeur 0
pour une duree de 900
- Date 7000 : Echeance de la tache 0
- Date 7000 : Activation de la tache 0
- Date 7000 : Debut d'execution de la tache 0 sur le processeur 0
pour une duree de 500
- Date 7000 : Debut d'execution de la tache 1 sur le processeur 1
pour une duree de 1100
- Date 7500 : Echeance de la tache 3
- Date 7500 : fin d'execution de la t0che 0 sur le processeur 0
duree d'allot = 500
- Date 7500 : Activation de la tache 3
- Date 7500 : Debut d'execution de la tache 2 sur le processeur 0
pour une duree de 400
- Date 7500 : Debut d'execution de la tache 1 sur le processeur 1
pour une duree de 500
- Date 7900 : fin d'execution de la t0che 2 sur le processeur 0
duree d'allot = 400
- Date 8000 : Echeance de la tache 0
- Date 8000 : Echeance de la tache 2
- Date 8000 : fin d'allot de la t0che 1 sur le processeur 1
duree d'allot = 100
- Date 8000 : Activation de la tache 0
- Date 8000 : Activation de la tache 2
- Date 8000 : Debut d'execution de la tache 0 sur le processeur 0
pour une duree de 500
- Date 8000 : Debut d'execution de la tache 2 sur le processeur 1
pour une duree de 100
- Date 8100 : fin d'allot de la t0che 2 sur le processeur 1
duree d'allot = 100
- Date 8100 : Debut d'execution de la tache 3 sur le processeur 1
pour une duree de 500
- Date 8500 : fin d'execution de la t0che 0 sur le processeur 0
duree d'allot = 400
- Date 8500 : Debut d'execution de la tache 1 sur le processeur 0
pour une duree de 100
- Date 8600 : fin d'execution de la t0che 3 sur le processeur 1
duree d'allot = 100
- Date 8600 : fin d'execution de la t0che 1 sur le processeur 0
duree d'allot = 100
- Date 8600 : Debut d'execution de la tache 2 sur le processeur 0
pour une duree de 900
- Date 9000 : Echeance de la tache 0
- Date 9000 : Activation de la tache 0
- Date 9000 : Debut d'execution de la tache 0 sur le processeur 0
pour une duree de 500
- Date 9500 : fin d'execution de la t0che 0 sur le processeur 0
duree d'allot = 500
- Date 9500 : Debut d'execution de la tache 2 sur le processeur 0
pour une duree de 500
- Date 10000 : fin d'execution de la t0che 2 sur le processeur 0
duree d'allot = 500
Process returned 0 (0x0) execution time : 0.105 s
Press ENTER to continue.

Bibliographie

[1] F. Many, "Combinaison des aspects temps réel et sûreté de fonctionnement pour la conception des plateformes avioniques," 2013.

[2] G. Nelissen, "Efficient Optimal Multiprocessor Scheduling Algorithms for Real-Time Systems," Bruxelle, 2012.

[3] R. I. Davis et A. Burns, «A survey of hard real-time scheduling for multiprocessor,» *ACM Computing Surveys,* vol. 43, n° 14, octobre 2011.

[4] J. Layland and C. Liu, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the Association for Computing Machinery,* pp. 46-61, 1973.

[5] O. H. Roux, "Cours master ARIA Temps Réel : exécutif et tâches," Ecole Centrale de Nantes, 2015.

[6] M. Cheramy, P.-E. Hladik and A. M. Déplanche, "Algorithmes pour l'ordonnancement temps," *Journal européen des systèmes automatisés,* 2015.

[7] A. M. Déplanche, "Ordonnancement temps réel multiprocesseur : panorama sur les politiques globales," *Institut de Recherche en Communications et Cybernétique de Nantes.*

[8] S. K. Dhall and C. L. Liu, "On a Real-Time Scheduling Problem," *Operations Research,* vol. 26, pp. 127 - 140, Février 1978.

[9] . S. . K. Baruah, N. K. Cohen and D. A. Varvel, "Proportionate progress: a notion of fairness in resource allocation," *Algorithmica,* vol. 15, pp. 600-625, 1996.

[10] D. Zhu, G. Nelissen, H. Su, y. Guo, V. Nélis and J. Goossens, "An optimal boundary fair scheduling," *Real-Time Systems,* vol. 71, pp. 1-53, April 2014.

[11] R. McNaugthon, "Scheduling with deadlines and loss functions," *Management Science,* vol. 6, no. 1, p. 1, 1959.

[12] H. Cho, B. Ravindran and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, Rio de Janeiro, 2006.

[13] G. Levin, S. Funk, C. Sadowski, I. Pye and S. Brandt, "DP-FAIR: A simple model for understanding optimal multiprocessor scheduling," in *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, Bruxelle, 2010.

[14] G. Nelissen, V. Berten, J. Goossens and D. Milojevic, "An Optimal Multiprocessor Scheduling Algorithm withour fairness," in *31th IEEE Real-Time Systems Symposium (Work in Progress*

session-RTSS10-WiP), 2010.

[15] G. Nelissen, V. Berten, J. Goossens and D. Milojevic, "U-EDF: An Unfair but Optimal Multiprocessor Scheduling Algorithm for Sporadic Tasks," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, Pisa, 2012.

[16] K. Boukir, "Etude d'U-EDF, une politique optimale d'ordonnancement temps réel multiprocesseur," Séminaire bibliographique, Master ARIA, Ecole Centrale de Nantes, 2016.

[17] "OSEK VDX Portal," 17 February 2005. [Online]. Available: http://www.osek-vdx.org/. [Accessed 15 March 2016].

[18] "Center for Hybrid and Embedded Software Systems," 18 October 2012. [Online]. Available: https://chess.eecs.berkeley.edu/. [Accessed 17 March 2016].

[19] T. Carrie, G. Morisot and C. Percia Mehlem, "Synthèse du standard OSEK," 2011. [Online]. Available: http://www.lattis.univ-toulouse.fr/~acco/acco_wiki/doku.php. [Accessed 27 June 2016].

[20] D. John, "OSEK/VDX history and structure," in *Open Systems in Automotive Networks*, London, 1998.

[21] J. L. Bechennec, M. Briday, S. Faucou, P. Molinaro and F. Pavin, "The trampoline handbook".

[22] J. L. Bechennec, M. Briday and S. Faucou, "Cours : Trampoline - Un système d'exploitation temps réel pour l'informatique embarquée," Ecole Centrale de Nantes, 2015.

[23] J. L. Bechennec, S. Foucou, M. Briday, P. Molinaro and Y. Trinquet, "Trampoline : un support pour le développement d'applications temps réel".

[24] J. L. Bechennec, M. Briday, S. Faucou and Y. Trinquet, "An Open Source Implementation of the OSEK/VDX RTOS Specification," *IEEE Conference on Emerging Technologies and Factory Automation,* pp. 62-69, 2006.

[25] G. Nelissen, V. Berten, J. Goossens and D. Milojevic, "Reducing Preemptions and Migrations in Real-Time Multiprocessor Scheduling Algorithms by Releasing," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, Toyama, 2011.