



HAL
open science

Ordonnancement temps réel : des politiques monoprocresseurs aux politiques multiprocresseurs

Maxime Chéramy, Anne-Marie Déplanche, Pierre-Emmanuel Hladik

► To cite this version:

Maxime Chéramy, Anne-Marie Déplanche, Pierre-Emmanuel Hladik. Ordonnancement temps réel : des politiques monoprocresseurs aux politiques multiprocresseurs. 2012. hal-00662741

HAL Id: hal-00662741

<https://hal.science/hal-00662741>

Submitted on 25 Jan 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Ordonnancement temps réel, des politiques monoprocesseurs aux politiques multiprocesseurs

Maxime Chéramy^{1,*}, Anne-Marie Déplanche², and Pierre-Emmanuel Hladik¹

¹CNRS; LAAS; 7 avenue du colonel Roche, F-31077 Toulouse Cedex 4, France

¹Université de Toulouse, UPS, INSA, INP, ISAE ; UT1, UTM, LAAS ; F-31077
Toulouse Cedex 4, France

²Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN);
UMR CNRS 6597

²Ecole Centrale de Nantes, Université de Nantes, Ecole des Mines de Nantes

January 10, 2012

Résumé

Ce rapport présente une vue d'ensemble des politiques d'ordonnancement en-ligne temps réel et plus particulièrement celles dédiées au multiprocesseur. Les politiques présentées reposent sur le modèle de tâches indépendantes de Liu et Layland et couvrent un grand nombre d'algorithmes aussi bien en partitionné, qu'en global ou encore semi-partitionné.

Abstract

This report presents an overview of real-time scheduling policies, in particular those dedicated to multiprocessor. The presented policies are based on the independent tasks model introduced by Liu and Layland and cover a large number of algorithms both partitioned, global or semi-partitioned.

Mots clefs: Ordonnancement; Global; Partitionné; Semi-Partitionné; RM; EDF; EDZL; PFair; DPFair; BF; EKG; EDF-WM; NPF-S.

*Adresse e-mail : maxime.cheramy@laas.fr

Contents

Introduction	3
1 Modélisation / Vocabulaire	4
1.1 Modélisation des applications	4
1.1.1 Modélisation des travaux	4
1.1.2 Modélisation des tâches	5
1.2 Architectures matérielles	6
1.3 Ordonnançabilité	7
1.4 Généralités sur l'ordonnancement	8
2 Ordonnancement monoprocesseur	11
2.1 Priorité fixe au niveau des tâches (<i>Fixed-Task-Priority</i>)	11
2.1.1 Rate Monotonic	11
2.1.2 Deadline Monotonic	12
2.1.3 Non comparabilité des ordonnancements préemptifs et non-préemptifs	12
2.2 Priorité fixe au niveau des travaux (<i>Fixed-Job-Priority</i>)	12
2.2.1 Earliest Deadline First	13
2.3 Priorité dynamique au niveau des travaux (<i>Dynamic-Job-Priority</i>)	14
2.3.1 Least Laxity First	14
2.4 Exemple de mise en œuvre	15
2.4.1 Ordonnancement non-préemptif	15
2.4.2 Ordonnancement Préemptif	16
3 Ordonnancement multiprocesseur	17
3.1 Ordonnancement par partitionnement	17
3.1.1 Modélisation du problème	18
3.1.2 Méthodes exactes	19
3.1.3 Heuristiques	19
3.1.4 Pire cas	20
3.2 Ordonnancement global	22
3.3 Généralisation des algorithmes monoprocesseurs	23
3.3.1 Critères d'ordonnançabilité RM / EDF	23
3.3.2 Algorithmes d'ordonnancement RM-US[ξ] et EDF-US[ξ]	24
3.3.3 Algorithmes d'ordonnancement à laxité nulle (<i>Zero Laxity</i>)	24
3.4 Ordonnancement global dit équitable (<i>PFair</i>)	25
3.4.1 Modélisation de l'ordonnancement PFair	25
3.4.2 Optimalité de l'ordonnancement PFair	27
3.4.3 Algorithme d'ordonnancement EPDF	28
3.4.4 Algorithme d'ordonnancement PD ²	28
3.5 Ordonnancement global DP-Fair	30
3.5.1 Principe	30
3.5.2 Abstraction du <i>T-L Plane</i>	31
3.5.3 Algorithme d'ordonnancement DP-WRAP	32
3.5.4 Algorithme d'ordonnancement LLREF	33
3.5.5 Algorithme d'ordonnancement BF	34
3.6 Ordonnancement semi-partitionné	35
3.6.1 Algorithme d'ordonnancement EKG	36
3.6.2 Algorithme d'ordonnancement EDHS	39
3.6.3 Algorithme d'ordonnancement EDF-WM	42
3.6.4 Algorithme d'ordonnancement NPS-F	44
Conclusion	47
Références	48

Introduction

Les politiques d'ordonnancement ont fait l'objet de nombreux travaux depuis les débuts de l'informatique. En 1973, LIU et LAYLAND publient un papier fondateur pour l'ordonnancement temps réel et permettent dès lors d'ordonnancer des tâches indépendantes tout en utilisant 100% de la capacité du processeur, sur un seul processeur. Depuis, les travaux sur le sujet ont été nombreux avec récemment un nouvel engouement pour les architectures multiprocesseurs.

Un système temps réel dur a comme particularité d'imposer l'exécution des tâches dans des limites temporelles précises. Le problème d'ordonnancement monoprocesseur consiste alors à déterminer pour chaque instant quelle tâche doit bénéficier de la ressource d'exécution. Le cas multiprocesseur est plus compliqué dans le sens où en plus d'un problème d'allocation temporelle, s'ajoute un problème d'allocation spatiale, c'est-à-dire quel processeur choisir.

Les algorithmes optimaux dans le cas monoprocesseur tels que EDF, perdent cette propriété dans le cas multiprocesseur, ce qui a encouragé le développement de nouvelles politiques. Plusieurs approches ont été étudiées, la plus simple étant de transformer le problème d'ordonnancement sur m processeurs en m problèmes d'ordonnancement monoprocesseur (partitionnement). Une seconde approche, dite globale, consiste à n'avoir qu'un seul ordonnanceur pour l'ensemble des processeurs. Les premières politiques globales proposées étaient des généralisations d'algorithmes monoprocesseurs, puis en 1996, BARUAH et al. introduisent la notion de *fairness* qui permet d'atteindre l'optimalité. Cependant, cette optimalité se fait au détriment d'un grand nombre de préemptions et de migrations. Ce constat est à la base des algorithmes semi-partitionnés. Afin de limiter le nombre de migrations, l'idée est alors de partir d'un algorithme partitionné et de permettre à un nombre limité de tâches de migrer.

Ce rapport est organisé comme suit : tout d'abord (partie 1), le modèle utilisé et le vocabulaire de base sont présentés; dans un second temps (partie 2), une présentation des principaux algorithmes monoprocesseur est faite; enfin, la partie 3 est consacrée aux algorithmes multiprocesseurs en les distinguant suivant trois catégories "partitionnés", "globaux" et "semi-partitionnés".

1 Modélisation / Vocabulaire

Afin de présenter et d'étudier des politiques d'ordonnancement, et en particulier traiter des problématiques d'ordonnancement, il convient de modéliser le problème. Dans cette partie, est présenté le modèle le plus fréquemment employé dans le domaine, à savoir le modèle de LIU et LAYLAND [LIU et LAYLAND 1973], qui permet de traiter le cas des tâches périodiques. Ce modèle a ensuite été généralisé aux tâches sporadiques [MOK 1983 ; LEUNG et WHITEHEAD 1982].

Les notations varient d'un auteur à l'autre mais le choix a été fait ici de reprendre les notations de [GOOSSENS 2006]. Les définitions proviennent principalement des articles [BARUAH et GOOSSENS 2003b ; GOOSSENS 2006] et complétés par [BUTTAZZO 2006 ; GROLLEAU 2011].

Cependant, d'autres modèles existent, tels que le modèle multiframe [MOK et CHEN 1997], sa généralisation [BARUAH et al. 1999], les transactions [TINDELL 1994], le modèle de tâche récurrente [BARUAH 2003] ou encore le modèle digraph [STIGGE et al. 2011], etc.

1.1 Modélisation des applications

Une application temps réel est constituée d'un ensemble de tâches (*tasks*). Une tâche contrôle le flot d'exécution d'un programme pour différentes données. Les instructions exécutées forment ce que l'on appelle un travail (*job*). Ainsi, une tâche est constituée d'un ensemble infini de travaux. On appelle aussi ces travaux, les instances de la tâche.

1.1.1 Modélisation des travaux

La définition 1.1 permet de caractériser un travail à l'aide de trois valeurs.

Définition 1.1 *Un travail est caractérisé par le tuple (a, e, d) :*

- *a l'instant d'arrivée (release time),*
- *e le temps d'exécution (computation time),*
- *d l'échéance absolue (absolute deadline).*

Autrement dit, un travail qui arrive à l'instant a nécessite e unités de temps d'exécution qui doivent lui être attribuées dans l'intervalle $[a, d[$ pour respecter sa contrainte d'échéance. La fonction *Schedule* définie par 1.2 permet de modéliser mathématiquement l'ordonnancement d'un travail.

Définition 1.2 *La fonction *Schedule* est une fonction qui prend en argument le temps t et un travail j ($\in J$, l'ensemble des travaux) et retourne 1 si le travail est ordonné à l'instant t sur un processeur et 0 sinon : $\mathcal{S} : \mathbb{R} \times J \rightarrow \{0, 1\}$*

Une politique d'ordonnancement s'occupe de placer des travaux sur un ou plusieurs processeurs. Mais seuls les travaux actifs (Définition 1.3) peuvent être ordonnés.

Définition 1.3 *Un travail est actif à l'instant t lorsque :*

- *Le travail est arrivé ($a \leq t$),*
- *l'échéance n'est pas encore arrivée ($t < d$) et*
- *le travail n'a pas fini d'être exécuté ($(\int_a^t \mathcal{S}(t', j) dt') < e$).*

La figure 1 est un exemple de représentation de la fonction *Schedule* qui permet de résumer les définitions qui viennent d'être présentées.

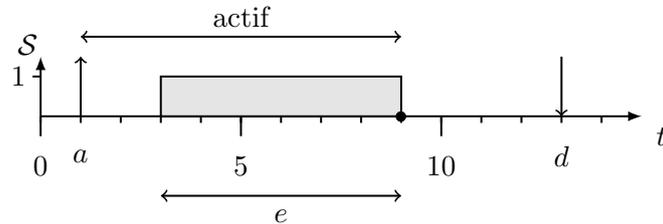


Figure 1 – Schéma représentatif d'un travail.

1.1.2 Modélisation des tâches

Comme dit précédemment, l'exécution d'une tâche donne lieu à des travaux. Cependant, on distingue principalement trois natures de tâches, selon la manière dont les travaux sont activés :

- Les tâches périodiques sont activées régulièrement à période fixe;
- Les tâches sporadiques sont activées de manière irrégulière mais avec toutefois au moins une propriété sur la durée entre l'arrivée de deux travaux consécutifs;
- Les tâches aperiodiques sont activées de manière irrégulière.

Le cas des tâches aperiodiques ne sera pas traité car il peut exister des solutions mieux adaptées à leur spécificité. Un système temps réel τ , qu'il soit périodique ou sporadique, est constitué d'une collection finie de tâches : $\tau = \{\tau_1, \dots, \tau_n\}$. La définition 1.4 permet de formaliser une tâche à l'aide de trois valeurs. À ces trois valeurs peuvent se rajouter, pour les tâches périodiques, la date de première activation (O_i) et une gigue temporelle (Définition 1.5). Chaque tâche τ_i étant constituée d'une collection infinie de travaux, on note : $\tau_i = \{\tau_{i,1}, \tau_{i,2}, \dots\}$ où $\tau_{i,j}$ est le $j^{\text{ème}}$ travail de la tâche τ_i .

Définition 1.4 Une tâche τ_i périodique ou sporadique est caractérisée par le tuple (C_i, T_i, D_i) :

- la durée d'exécution C_i dans le pire cas (WCET) de chaque travail de la tâche τ_i ;
- sa période d'activation T_i . Dans le cas de tâches sporadiques, c'est la durée minimale entre ses activations successives;
- son échéance relative ou délai critique D_i . Durée entre l'arrivée d'un travail et son échéance (un travail qui arrive à l'instant t doit se terminer avant l'instant $t + D_i$).

Définition 1.5 La gigue temporelle J_i représente l'incertitude quant à la date réelle à laquelle les travaux sont activés. Activation du $k^{\text{ème}}$ travail de τ_i dans l'intervalle : $[O_i + (k-1)T_i, O_i + (k-1)T_i + J_i]$.

De nombreux travaux sur les ordonnanceurs, et en particulier sur l'étude d'ordonnabilité, ont besoin de distinguer plus précisément les différents types de tâches. Nous définissons alors ce qu'est une tâche concrète (Définition 1.6) et ce qu'est un système à

départ simultané (Définition 1.7). Enfin, une distinction importante est faite sur le type d'échéance des tâches (Définition 1.8).

Définition 1.6 *On dit que la tâche est concrète, lorsqu'on connaît les dates d'activation. C'est le cas par exemple des tâches périodiques lorsqu'on connaît la date de la première activation.*

Définition 1.7 *Si les tâches démarrent au même instant ($\forall i, O_i = 0$), on parle de système à départ simultané (synchronous task system).*

Définition 1.8

- Si $D_i = T_i$, on parle de tâche à échéance implicite ou échéance sur requête (Implicit-deadline),
- Si $D_i < T_i$, on parle de tâche à échéance contrainte (Constrained-deadline),
- Sinon, on parle de tâche à échéance arbitraire (Arbitrary-deadline).

Enfin, nous pourrons avoir besoin de ces grandeurs :

- Date de début d'exécution du travail $\tau_{i,j} : s_{i,j}$
- Date de fin d'exécution du travail $\tau_{i,j} : f_{i,j}$
- Temps de réponse du travail $\tau_{i,j} : R_{i,j} \stackrel{\text{def}}{=} f_{i,j} - a_{i,j}$
- Temps de réponse maximum de la tâche $\tau_i : R_i \stackrel{\text{def}}{=} \max_j R_{i,j}$
- Laxité : Temps maximum pendant lequel le travail peut ne pas disposer du processeur sans manquer son échéance.

Les notions les plus importantes sont représentées par la figure 2 à travers un exemple d'exécution d'une tâche périodique à échéance contrainte τ_i avec ses travaux.

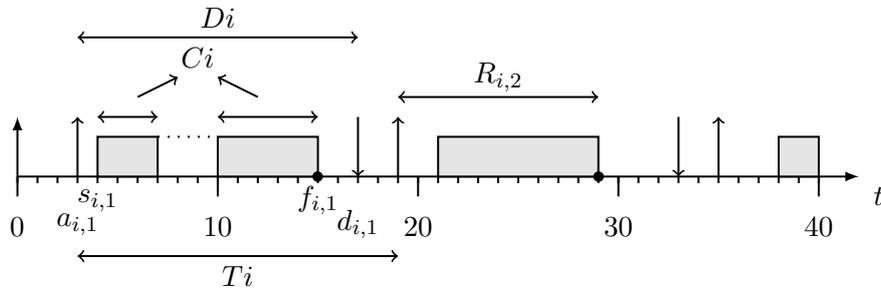


Figure 2 – Schéma représentatif d'une tâche τ_i .

1.2 Architectures matérielles

Dans le cadre d'architectures multiprocesseurs, on distingue trois types de plates-formes :

- Processeurs identiques (ou homogènes). Plate-forme multiprocesseur constituée de plusieurs processeurs identiques (même caractéristiques, interchangeables).
- Processeurs uniformes. Chaque processeur est caractérisé par sa capacité de calcul : lorsqu'un travail s'exécute sur un processeur de capacité de calcul s pendant t unités de temps, il réalise $s \times t$ unités de travail.

- Processeurs indépendants (ou hétérogènes). Une capacité de calcul $c_{i,j}$ est associée à chaque couple travail-processeur (J_i, P_j) . L'interprétation est la suivante : le travail J_i réalise $c_{i,j} \times t$ unités de travail lorsqu'il s'exécute sur le processeur P_j pendant t unités de temps. Ce modèle permet de gérer les processeurs spécialisés qui ne peuvent traiter que certains travaux : il suffit de fixer $c_{i,j}$ à 0 pour exprimer que le processeur P_j ne peut pas prendre en charge le travail J_i .

1.3 Ordonnabilité

Les études d'ordonnabilité visent à fournir des conditions suffisantes et/ou nécessaires pour savoir si un système composé de tâches sera ordonnable par une politique d'ordonnement sur une architecture matérielle donnée. La définition 1.9 formalise la notion de système fiablement ordonnable par un algorithme d'ordonnement, et la définition 1.10 permet de conclure sur l'ordonnabilité d'une tâche.

Définition 1.9 *Un système S est fiablement ordonné par un algorithme d'ordonnement A , si pour toute instance de système correspondant à la spécification de S , l'ordonnement de S par A respecte toutes les contraintes temporelles.*

Définition 1.10 *Un système S est ordonnable s'il existe un algorithme qui l'ordonne fiablement.*

De nombreux travaux sur l'ordonnement visent à fournir des algorithmes optimaux (Définition 1.11), mais cette optimalité n'est atteinte que dans un cadre précis (restriction sur le type de tâches et sur l'architecture matérielle) et généralement en négligeant les surcoûts liés à l'ordonneur (temps de calcul, préemption, etc.) qui ne sont pas présents dans le modèle.

Définition 1.11 *Un algorithme A est dit optimal pour une classe de tâches, une architecture matérielle et parmi une classe de politiques d'ordonnement, si et seulement si il peut ordonner fiablement tout système ordonnable.*

Lorsqu'on a le choix entre plusieurs politiques d'ordonnement, il peut être utile de pouvoir les comparer. Un premier élément de comparaison est la notion de dominance (Définition 1.12) qui permet de montrer la supériorité forte d'un algorithme sur un autre. Nous verrons par exemple que certaines politiques de type semi-partitionné dominent toute politique de type partitionné. Au contraire, il existe des algorithmes qui sont généralement meilleurs que d'autres (montré par simulation) mais pour lesquels il existe des cas qui font exception. Dans ce cas, on parle d'algorithmes non comparables (Définition 1.13). Cette notion d'algorithmes non comparables n'est pas strictement équivalente à dire que A ne domine pas B et B ne domine pas A . Un contre-exemple simple : s'il n'existe aucun système de tâches qui soit ordonnable par A ou par B alors ils ne sont pas non comparables d'après la définition.

Définition 1.12 *Un algorithme A domine un algorithme B si tout système de tâches ordonnable par B l'est par A et s'il existe au moins un système de tâche ordonnable par A et qui ne le soit pas par B .*

Définition 1.13 *Deux algorithmes A et B sont dit non comparables si, et seulement si :*

- *il existe un système de tâches qui soit ordonnable par A et non ordonnable par B*

- il existe un système de tâches qui soit ordonnançable par B et non ordonnançable par A

Les études d'ordonnançabilité reposent essentiellement sur les critères qui suivent, et permettent généralement de ressortir des conditions nécessaires (Définition 1.15) et/ou suffisantes (Définition 1.14) d'ordonnançabilité.

- Facteur d'utilisation d'une tâche :

$$U(\tau_i) \stackrel{\text{def}}{=} C_i/T_i \tag{1.1}$$

- Facteur d'utilisation maximum de l'ensemble des tâches :

$$U_{max}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} U(\tau_i) \tag{1.2}$$

- Utilisation du système¹ :

$$U(\tau) \stackrel{\text{def}}{=} \frac{1}{m} \sum_{\tau_i \in \tau} U(\tau_i), \text{ avec } m \text{ le nombre de processeurs.} \tag{1.3}$$

Définition 1.14 *Lorsqu'un système de tâches respecte une condition suffisante de l'algorithme d'ordonnancement utilisé, alors il est ordonnançable par cet algorithme. À l'inverse, si un système de tâches ne respecte pas une condition suffisante, alors cela ne veut pas dire que le système n'est pas ordonnançable.*

Définition 1.15 *Lorsqu'un système de tâches ne respecte pas une condition nécessaire de l'algorithme d'ordonnancement utilisé, alors il n'est pas ordonnançable. Dans le cas contraire, on ne peut pas en déduire que le système est ordonnançable.*

1.4 Généralités sur l'ordonnancement

L'ordonnancement consiste à placer dans le temps les travaux sur le ou les processeurs. Ce travail peut être effectué avant l'exécution du système, et revient donc à un problème particulier de planification. Ou il peut être fait pendant l'exécution du système, apportant ainsi une plus grande flexibilité. On parle alors d'ordonnancement hors-ligne ou en-ligne (Définition 1.16 et 1.17). Un avantage de l'ordonnancement hors-ligne est qu'il est possible d'effectuer des calculs lourds, en revanche, il n'est plus possible de tenir compte des paramètres variables au cours de l'exécution.

Définition 1.16 *Un algorithme d'ordonnancement s'exécute hors-ligne lorsque le calcul de la séquence d'ordonnancement est effectué avant l'exécution du système.*

Définition 1.17 *Un algorithme d'ordonnancement s'exécute en-ligne lorsque le calcul de la séquence d'ordonnancement est fait pendant l'exécution du système.*

Dans la plupart des systèmes, il est possible d'interrompre une tâche à tout moment pour libérer le processeur. Pour cela, le système mémorise l'état actuel de la tâche de façon à pouvoir plus tard reprendre l'exécution de celle-ci. Cette interruption s'appelle une préemption (Définition 1.18) et un ordonnanceur capable de préempter une tâche est

1. Certains auteurs préfèrent garder la même formule que dans le cas monoprocesseur, à savoir sans diviser par le nombre de processeurs. Dans ce rapport, le terme d'utilisation cumulée sera employé en ce sens.

dit préemptif. Les ordonnanceurs étudiés dans ce document seront majoritairement de type préemptif car cela apporte une flexibilité très importante.

Définition 1.18 *Un ordonnanceur préemptif est un ordonnanceur qui peut interrompre l'exécution d'une tâche non terminée pour en exécuter une autre.*

Dans le cadre des systèmes multiprocesseurs, on introduit aussi la notion de migration. Certains ordonnanceurs permettent aux travaux successifs d'une tâche d'être exécutés sur des processeurs différents, on parle alors de migration limitée aux travaux. Et enfin, certains ordonnanceurs sont même capables de préempter un travail et le faire reprendre sur un processeur différent.

Dans [ANDERSON, BUD et DEVI 2005], les auteurs distinguent trois degrés de migration autorisée :

1. Aucune migration : C'est l'ordonnancement par partitionnement.
2. Migration restreinte aux frontières des travaux.
3. Migration libre : Les travaux peuvent aussi migrer pendant leur exécution.

Les préemptions et les migrations induisent naturellement un surcout à l'exécution. Ces coûts sont ignorés dans le modèle présenté mais certains travaux essayent d'en tenir compte, en particulier en effectuant des simulations [CALANDRINO et al. 2006].

La majorité des algorithmes choisissent les travaux à exécuter en fonction de leur priorité. On distingue trois manières différentes de gérer ces priorités [GOOSSENS 2006] :

- Les algorithmes à priorité fixe au niveau des tâches (*Fixed-Task-Priority*), pour lesquels les tâches et les travaux partagent la même priorité. Exemple : Rate Monotonic
- Les algorithmes à priorité fixe au niveau des travaux (*Fixed-Job-Priority*) fixent pendant l'exécution des priorités fixes aux travaux. Ainsi, les travaux d'une même tâche peuvent avoir des priorités différents, mais cette priorité ne varie pas au cours du temps. Exemple : Earliest Deadline First
- Les algorithmes à priorité dynamique au niveau des travaux (*Dynamic-Job-Priority*) définissent à chaque instant une priorité pour chaque travail. Exemple : Least Laxity First

ANDERSON, BUD et DEVI font les mêmes distinctions dans [ANDERSON, BUD et DEVI 2005] mais en utilisant les termes respectifs : *static*, *dynamic but fixed within a job*, *fully-dynamic*.

Attention, un nombre non négligeable d'auteurs ne font apparaître que deux catégories, les algorithmes à priorité fixe et les algorithmes dynamiques. Les algorithmes classés FJP et DJP font alors partie de la catégorie des algorithmes dynamiques.

On distinguera aussi les ordonnanceurs conservatifs (Définition 1.19) de ceux qui ne le sont pas. Un ordonnanceur non-conservatif peut être intéressant dans le cas non-préemptif car il peut être utile d'attendre l'arrivée d'un travail court avant d'exécuter un travail plus long.

Définition 1.19 *Un ordonnanceur conservatif a comme particularité de ne jamais laisser le processeur dans un état oisif (idle) s'il reste des travaux à exécuter. Au contraire, un ordonnanceur non-conservatif peut décider d'introduire des laps de temps pendant lesquels aucune tâche n'est ordonnancée.*

Une possible généralisation de la définition d'ordonnanceur conservatif, appliquée aux systèmes multiprocesseurs uniformes, est donnée par la définition 1.20. Cette définition est particulièrement stricte dans le cas de processeurs différents dans le sens où cela implique des migrations en cascade lorsqu'un travail très prioritaire libère le processeur le plus rapide (décalage de l'ensemble des travaux en cours d'exécution).

Définition 1.20 *Un algorithme d'ordonnement pour une plate-forme uniforme est dit conservatif s'il satisfait les conditions suivantes [GOOSSENS 2006] :*

- *Il ne laisse jamais un processeur oisif lorsqu'il y a des travaux en attente.*
- *S'il doit mettre des processeurs dans l'état oisif, ils devront être les plus lents.*
- *Il exécute toujours les travaux les plus prioritaires sur les processeurs les plus rapides.*

Enfin, il existe des algorithmes qui sont basés sur des quanta de temps (discrétisation du temps et prise de décision possible à chaque quantum), d'autres utilisent aussi des timers mais ne forment pas d'intervalles réguliers, et enfin, certains algorithmes se limitent aux événements simples tels que l'arrivée d'un nouveau travail et la fin d'un travail. Le lecteur devra rester attentif à ces aspects car certains travaux théoriques ne tiennent pas compte de la résolution des timers.

2 Ordonnancement monoprocesseur

Dans cette section, seront présentés les algorithmes classiques d'ordonnancement monoprocesseur en-ligne. Tous les algorithmes présentés partagent au moins la condition nécessaire évidente de $U \leq 1$, c'est-à-dire que l'utilisation totale du système n'excède pas la capacité de calcul du processeur.

2.1 Priorité fixe au niveau des tâches (*Fixed-Task-Priority*)

Cette sous-partie présente les algorithmes qui appartiennent à la famille des algorithmes d'ordonnancement à priorité fixe. Chaque tâche a une certaine priorité, dont héritent ses travaux, et qui est définie avant l'exécution du système et qui ne varie pas au cours de la vie du système. Généralement, l'ordonnancement est préemptif, mais ce n'est pas une obligation. En l'absence de précision, l'ordonnanceur présenté sera considéré comme préemptif.

2.1.1 Rate Monotonic

L'ordonnanceur Rate Monotonic (RM) est un ordonnanceur à priorité fixe. La priorité des tâches est fonction de leur période. Plus précisément, plus la période est petite et plus la tâche est prioritaire.

Dans le cas d'un système de tâches sporadiques à échéance implicite et départ simultané, LIU et LAYLAND ont montré dans le théorème 2 de [LIU et LAYLAND 1973] que le choix de priorités effectué par RM est optimal dans le sens où il n'existe pas d'autre affectation de priorité fixe qui permettrait d'ordonner un ensemble de tâches qui ne peut pas être ordonné par RM.

L'affectation des priorités par RM n'est pas toujours optimale dans les autres cas. [GOOSSENS et DEVILLERS 1997] traite en particulier le cas où les tâches ne sont plus à départ simultané.

Dans [LIU et LAYLAND 1973], les auteurs déterminent une condition suffisante d'ordonnabilité pour des systèmes de n tâches à échéance implicite, sporadiques et à départ simultané. Cette condition suffisante est donnée par l'équation 2.1. La preuve donnée par LIU et LAYLAND est contestée par DEVILLERS et GOOSSENS et est corrigée et complétée par ces derniers [DEVILLERS et GOOSSENS 2000].

$$U(\tau) = \sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \quad (2.1)$$

Pour avoir un meilleur ordre de grandeur, il est intéressant de calculer la limite de cette fonction pour n qui devient grand (Équation 2.2). On trouve que pour $n > 5$, la valeur obtenue est déjà très proche de la limite, à savoir environ 0.7.

$$\lim_{n \rightarrow \infty} n(2^{1/n} - 1) = \ln 2 \approx 0.69 \quad (2.2)$$

Une autre condition suffisante, appelée condition hyperbolique, a été introduite par [BINI et BUTTAZZO 2001] et est donnée par l'équation 2.3. Celle-ci offre une meilleure

limite d'utilisation pour un temps de calcul similaire. Par simulation et étude analytique, les auteurs trouvent que lorsque le nombre de tâches à traiter tend vers l'infini, la limite d'utilisation fournie par la condition hyperbolique est $\sqrt{2}$ fois plus grande que celle fournie par la condition de LIU et LAYLAND.

$$\prod_{i=1}^n \left(\frac{C_i}{T_i} + 1 \right) \leq 2 \quad (2.3)$$

Il est aussi possible de calculer l'ordonnabilité exacte d'un système par RM, les travaux de [LEHOCZKY, SHA et DING 1989; AUDSLEY et al. 1993] vont dans ce sens. La complexité de ces tests étant pseudo-polynomiale, il est impossible de faire ces tests en ligne. Pour cela, des approximations sont fournies par d'autres tests [BINI et BUTTAZZO 2002].

Enfin, si les périodes des tâches sont harmoniques, alors la condition suffisante d'ordonnabilité de RM devient :

$$U(\tau) = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1 \quad (2.4)$$

2.1.2 Deadline Monotonic

L'ordonnanceur Deadline Monotonic est un ordonnanceur à priorité fixe avec pour priorité l'échéance relative de la tâche [LEUNG et WHITEHEAD 1982]. Plus l'échéance est petite et plus la tâche est prioritaire.

L'affectation des priorités est optimale pour des systèmes de tâches synchrones à échéance contrainte. Dans le cas des systèmes à échéances implicites, Deadline Monotonic et Rate Monotonic se confondent.

La condition suffisante pour DM pour des systèmes de tâches à échéance contrainte, donnée par l'équation 2.5, est similaire à celle de RM.

$$\sum_{i=1}^n \frac{C_i}{D_i} \leq n(2^{1/n} - 1) \quad (2.5)$$

2.1.3 Non comparabilité des ordonnancements préemptifs et non-préemptifs

Enfin, remarquons qu'un algorithme à priorité fixe tel que Rate Monotonic préemptif et son équivalent non-préemptif ne sont pas comparables pour des tâches à échéance contrainte :

- Il existe un système de tâche τ ordonnable avec préemption et non ordonnable sans (voir Figure 3).
- Il existe un système de tâche τ ordonnable sans préemption et non ordonnable avec (voir Figure 4).

2.2 Priorité fixe au niveau des travaux (*Fixed-Job-Priority*)

Dans cette sous-partie, nous nous intéressons aux algorithmes à priorité fixe au niveau des travaux. C'est-à-dire que la priorité d'une tâche peut varier d'un travail à un autre

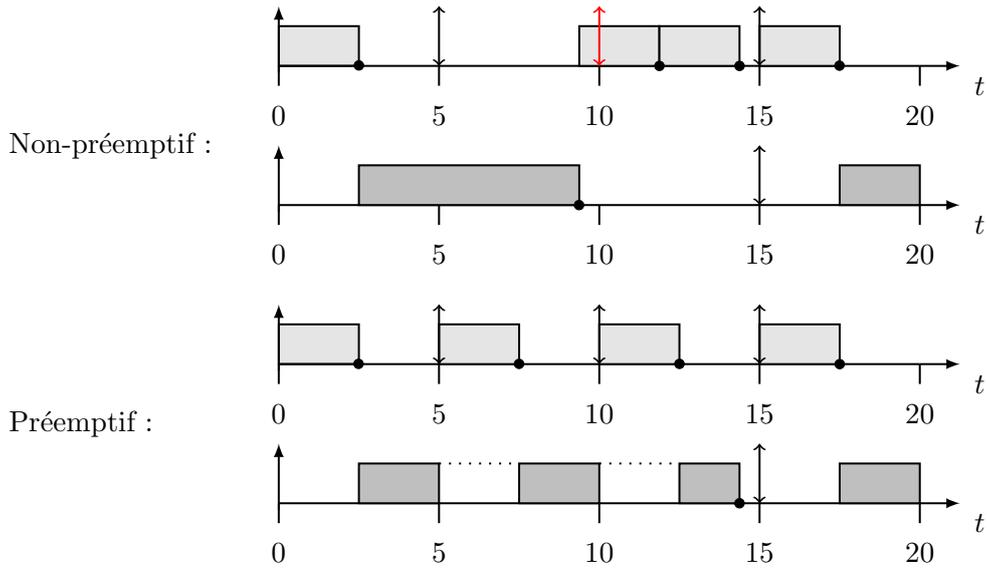


Figure 3 – Exemple simple où un ordonnanceur non préemptif sera mis en échec (τ_1 est prioritaire sur τ_2). $\tau_1 : (C_1 = 2.5ms, T_1 = 5ms, D_1 = 5ms)$ et $\tau_2 : (C_2 = 6.8ms, T_2 = 15ms, D_2 = 15ms)$

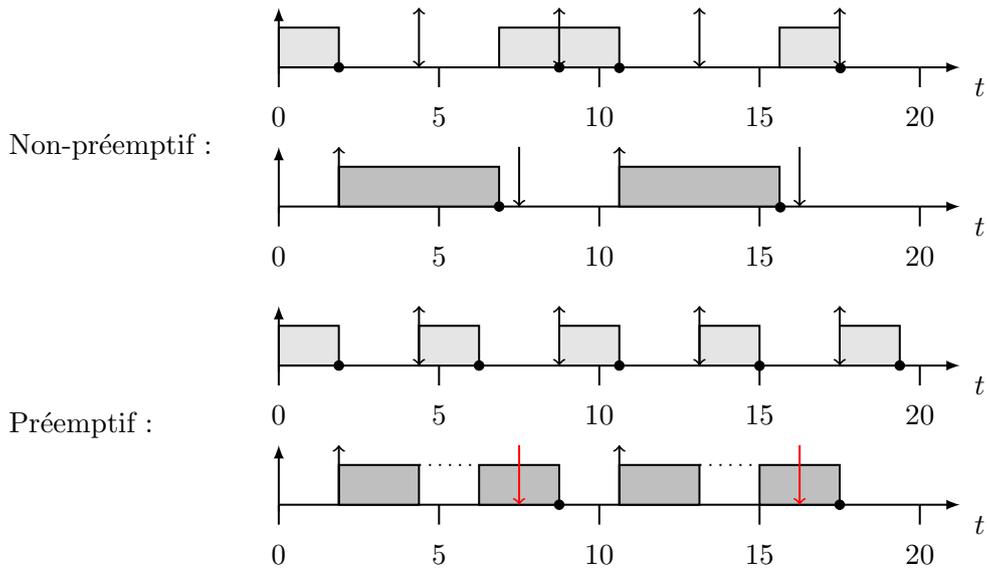


Figure 4 – Exemple simple où un ordonnanceur préemptif sera mis en échec (τ_1 est prioritaire sur τ_2). $\tau_1 : (C_1 = 1.8ms, T_1 = 4.3, D_1 = 4.3)$ et $\tau_2 : (C_2 = 5ms, T_2 = 8.6ms, D_2 = 5.6ms)$

mais que la priorité d'un travail ne changera pas. Ces algorithmes sont souvent appelés à priorité dynamique dans le sens où la priorité d'une tâche peut varier.

2.2.1 Earliest Deadline First

Earliest Deadline First (EDF) est un ordonnanceur à priorité fixe au niveau des travaux dont la priorité est plus forte lorsque l'échéance absolue est plus proche. Généralement,

EDF est utilisé avec préemption : si une nouvelle tâche arrive et que sa date d'échéance absolue est plus rapprochée, alors la tâche en cours d'exécution est préemptée et la nouvelle tâche est exécutée. Cependant, EDF peut aussi être utilisé sans préemption, mais ce cas ne sera pas traité ici et les résultats qui sont fournis ne concernent que EDF préemptif.

La condition nécessaire et suffisante de EDF, pour des tâches périodiques ou sporadiques à échéance implicite, est donnée par l'équation 2.6 [LIU et LAYLAND 1973 ; DERTOUZOS 1974].

$$U(\tau) = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1 \quad (2.6)$$

Pour des tâches à échéances contraintes, on retiendra que EDF est un algorithme d'ordonnancement optimal dans le sens où si un système de tâches est ordonnançable, alors EDF sera capable de l'ordonnancer [BARUAH et GOOSSENS 2003b].

EDF offre de meilleures performances que RM en terme d'ordonnançabilité dans bien des cas, Buttazzo en fait une étude comparative dans [BUTTAZZO 2005].

2.3 Priorité dynamique au niveau des travaux (*Dynamic-Job-Priority*)

2.3.1 Least Laxity First

La laxité dynamique est le temps maximum pendant lequel l'exécution de la tâche peut être retardée sans manquer son échéance. Cela s'écrit : $d - (t_{cur} + c_{ret})$ avec d la date d'échéance absolue, t_{cur} la date actuelle et c_{ret} le temps d'exécution restant de la tâche.

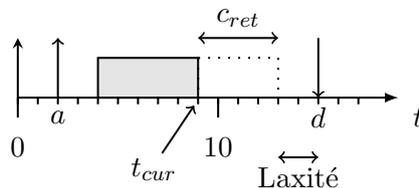


Figure 5 – Représentation visuelle de la notion de laxité dynamique.

L'algorithme Least Laxity First (LLF) rend prioritaire les travaux dont la laxité dynamique est la plus faible. C'est donc un algorithme avec priorité dynamique au niveau des travaux car la priorité d'un travail évolue avec le temps (la priorité de chaque travail non exécuté augmente).

Contrairement à un algorithme à priorité fixe pour les travaux tel que EDF, il est nécessaire pour LLF de mettre à jour fréquemment les priorités des travaux alors que pour EDF ce n'est fait qu'à l'activation de la tâche.

LLF est optimal pour ordonnancer des systèmes de tâches indépendantes [MOK 1983].

Cet algorithme peut engendrer de nombreux changements de contexte, en effet, pour s'en persuader, il suffit de prendre 2 travaux (notons a et b) avec la même laxité (*laxity-tie*) : si on choisit d'exécuter a alors la priorité de b augmente et dépasse donc la priorité de a , donc b préempte a et la situation inverse se produit alors. La figure 6 montre ce phénomène avec deux tâches quelconques et une mise à jour des priorités toutes les millisecondes. Bien

évidemment, plus le nombre de tâches est important, plus la charge système est grande et plus ce problème risque de survenir.

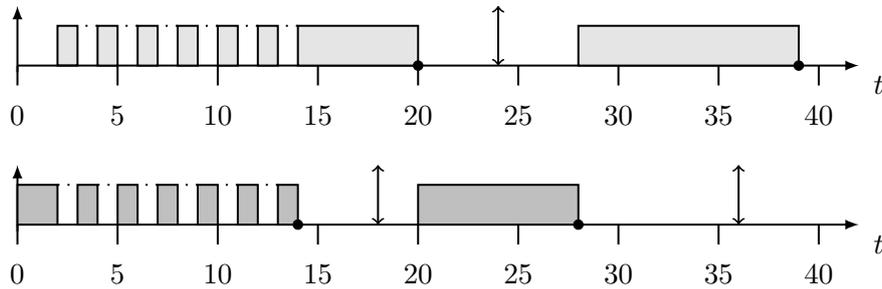


Figure 6 – Exemple du phénomène d’oscillation entre deux tâches engendré par LLF.

L’algorithme MLLF présenté dans [OH et YANG 1998] tente de résoudre ce problème tout en gardant son optimalité. Dans ce papier, les résultats par simulation indiquent une réduction par deux du nombre de préemptions lorsque la charge devient maximale.

2.4 Exemple de mise en œuvre

Dans cette section, sont proposés deux algorithmes écrits en pseudo-code qui permettent de mettre un œuvre des ordonnancements de type priorité fixe sur les tâches (ex : RM) ou sur les travaux (ex : EDF), préemptif ou non préemptif. Il existe d’autres façons de faire la même chose, mais l’objectif ici est simplement de fixer les idées et non de donner une solution optimisée.

Nous aurons besoin d’un ensemble de tâches actives que l’on va nommer A . Le choix de la structure de donnée est laissée libre (tas, liste, tableau...) mais n’est sûrement pas à négliger lors de la phase d’implémentation.

2.4.1 Ordonnement non-préemptif

Le cas non-préemptif est le plus simple dans la mesure où l’ordonneur ne peut rien faire tant que le processeur est déjà occupé. Les décisions se font donc à la fin de chaque travail et à l’arrivée d’un nouveau travail. Lors de l’activation d’une tâche, c’est-à-dire à l’instant d’arrivée d’un travail de la tâche, il faut ajouter cette tâche dans la liste des tâches actives et donc en attente du processeur (Algorithme 1). Attention, dans le cas d’un algorithme à priorité fixe sur les travaux, il faut aussi recalculer la priorité de la tâche.

Entrées : τ_i : la tâche qui est activée.

- 1 $A \leftarrow A \cup \{\tau_i\};$
- 2 `ordonnance();`

Algorithme 1 : `activation_tache(τ_i)`

Lors de la fin de l’exécution du travail d’une tâche, il suffit de provoquer un réordonnement

puisque le processeur est maintenant libéré (Algorithme 2).

```
Entrées :  $\tau_i$  : la tâche qui est terminée.  
1 ordonnance();
```

Algorithme 2 : terminaison_tache(τ_i)

Et enfin, l'algorithme principal d'ordonnancement va simplement lancer l'exécution de la tâche la plus prioritaire, si elle existe, sur le processeur (Algorithme 3).

```
1  $\tau_i \leftarrow$  la tâche de plus grande priorité de  $A$ ;  
2 Si processeur est libre et  $\tau_i$  existe Alors  
3   | Exécute  $\tau_i$  sur le processeur;  
4   | Retire  $\tau_i$  de  $A$  ;  
5 Fin Si
```

Algorithme 3 : ordonnance() sans préemption

2.4.2 Ordonnancement Préemptif

La procédure d'ordonnancement traite maintenant le cas où le processeur n'est pas libre. Le choix a été fait ici de ne préempter les tâches que dans la procédure **ordonnance** (Algorithme 4).

```
1  $\tau_i \leftarrow$  la tâche de plus grande priorité de  $A$ ;  
2 Si  $\tau_i$  existe Alors  
3   | Si processeur est libre Alors  
4   |   | Exécute  $\tau_i$  sur le processeur;  
5   |   | Retire  $\tau_i$  de  $A$ ;  
6   | Sinon  
7   |   | Soit  $\tau_j$  la tâche actuellement ordonnancée sur le processeur;  
8   |   | Si  $priorite(\tau_i) > priorite(\tau_j)$  Alors  
9   |   |   | Préempte  $\tau_j$ ;  
10  |   |   |  $A \leftarrow A \cup \{\tau_j\}$ ;  
11  |   |   | Exécute  $\tau_i$  sur le processeur;  
12  |   |   | Retire  $\tau_i$  de  $A$ ;  
13  |   | Fin Si  
14  | Fin Si  
15 Fin Si
```

Algorithme 4 : ordonnance() avec préemption

3 Ordonnancement multiprocesseur

Cette partie s'intéresse aux algorithmes pour ordonnancer les tâches sur de multiples processeurs. Les politiques présentées et les résultats associés se limitent au cas des processeurs identiques pour des raisons de simplicité, cependant, certains algorithmes sont capables de tirer profit d'architectures avec des processeurs différents.

L'ordonnancement monoprocesseur vise à résoudre le problème d'allocation temporelle du processeur aux tâches. L'ordonnancement multiprocesseur rajoute en plus un problème d'allocation spatiale, c'est-à-dire quel processeur utiliser.

Les premiers travaux sur l'ordonnancement multiprocesseur classaient les algorithmes en deux catégories :

- Ordonnancement par partitionnement : répartition a priori des n tâches sur les m processeurs. L'ordonnancement est réalisé localement sur chaque processeur et les tâches ne peuvent pas migrer d'un processeur à l'autre.
- Ordonnancement global : l'ordonnanceur s'applique sur l'ensemble des tâches et processeurs. Les tâches peuvent donc migrer d'un processeur à l'autre.

L'intérêt des chercheurs s'est porté au début sur l'ordonnancement par partitionnement car il est simple et il permet de réutiliser les résultats des travaux sur l'ordonnancement monoprocesseur. En effet, l'ordonnancement par partitionnement revient à réduire le problème d'ordonnancement sur m processeurs à m problèmes d'ordonnancement monoprocesseur.

L'ordonnancement global a longtemps été mis de côté en raison des faibles taux d'utilisation autorisés, et en particulier DHALL 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 appliquant globalement les politiques RM ou EDF [DHALL et LIU 1978]. Depuis, de nombreuses politiques d'ordonnancement globales dédiées au cas multiprocesseur ont vu le jour et certaines permettent d'atteindre l'optimalité vis-à-vis de la limite d'utilisation ($\sum_{\tau_i \in \tau} U(\tau_i) \leq m$).

Il est aussi important de noter que dans la plupart des cas, il n'est pas possible de comparer directement ces politiques [LEUNG et WHITEHEAD 1982]. LEUNG et WHITEHEAD ont montré par exemple que pour des priorités fixes au niveau des tâches, les approches par partitionnement et globales sont non comparables (Définition 1.13).

Les politiques d'ordonnancement globales souffrant de certains défauts, des politiques d'ordonnancement intermédiaires où la migration des tâches est mieux contrôlée ont été imaginées. Ces algorithmes forment alors une troisième catégorie appelée : ordonnancement semi-partitionné.

3.1 Ordonnancement par partitionnement

Le principe de l'ordonnancement par partitionnement est relativement simple dans le cadre de processeurs identiques. Nous nous limitons aussi au cas des tâches périodiques ou sporadiques, à échéance sur requête. Il est aussi nécessaire de connaître, dès la conception du système, l'ensemble des tâches. L'ordonnancement par partitionnement consiste à partitionner les tâches pour affecter chaque tâche à un unique processeur. Ensuite, les tâches associées à chaque processeur sont ordonnancées selon un ordonnanceur monoprocesseur

(Figure 7). Généralement, le partitionnement est effectué hors-ligne, en se basant sur les conditions d'ordonnançabilité et les caractéristiques des tâches, en particulier leur facteur d'utilisation. Ce problème de partitionnement est un exemple de problème de bin-packing, qui pour rappel est NP-Complet.

Deux variantes au problème existent : dans certains cas, on souhaite déterminer le nombre minimum de processeurs nécessaires pour pouvoir ordonnancer correctement le système. Dans d'autres cas, le nombre de processeurs est fixé et l'on souhaite juste trouver un partitionnement tel que le système soit ordonnançable. Le premier problème est donc un problème de minimisation tandis que le second est un problème de satisfaction de contraintes.

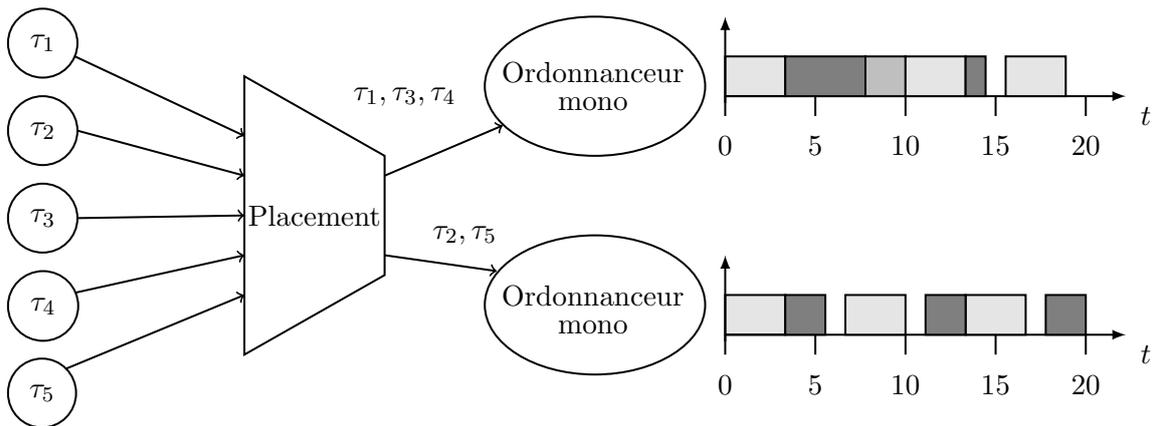


Figure 7 – Représentation graphique de la stratégie d'ordonnancement par partitionnement

3.1.1 Modélisation du problème

Dans cette sous-partie, est présentée une modélisation possible du problème qui se base sur les facteurs d'utilisation.

Soit x_{ij} , une variable binaire qui vaut 1 lorsque la tâche τ_i est affectée au processeur P_j et 0 sinon. Il va de soi qu'une tâche ne doit être affectée qu'à un seul processeur, ce qui se modélise par la contrainte (3.1). La seconde contrainte est que l'utilisation totale de l'ensemble des tâches affecté à un processeur ne dépasse pas l'utilisation que le processeur est capable d'accepter avec l'ordonnanceur monoprocasseur choisi. Ceci se formalise par (3.2), avec $CapMax_j$ l'utilisation totale maximale telle que le sous-ensemble de tâches soit ordonnançable sur le processeur P_j (constante qui vaut 1 par exemple avec un ordonnancement local par EDF).

$$\forall i \in \{1, \dots, n\}, \sum_{j=1}^m x_{ij} = 1 \quad (3.1)$$

$$\forall j \in \{1, \dots, m\}, \sum_{i=1}^n U(\tau_i) \times x_{ij} \leq CapMax_j \quad (3.2)$$

Si l'objectif est de déterminer le nombre minimum de processeurs nécessaires, il est alors nécessaire de rajouter une variable binaire y_j qui vaut 1 lorsque le processeur P_j est utilisé et 0 sinon (m devient alors le nombre maximum de processeurs autorisés). La contrainte (3.2) est alors modifiée pour intégrer y_j (3.3) de façon à ce qu'un processeur non utilisé n'accepte aucune tâche. Enfin, on cherche à minimiser le nombre de processeurs nécessaires et cela s'exprime par la relation (3.4).

$$\forall j \in \{1, \dots, m\}, \sum_{i=1}^n U(\tau_i) \times x_{ij} \leq CapMax_j \cdot y_j \quad (3.3)$$

$$\text{minimize} \sum_{j=1}^m y_j \quad (3.4)$$

3.1.2 Méthodes exactes

Le nombre de partitions de n éléments en m ensembles est tel qu'énumérer toutes les possibilités devient rapidement impossible. Cependant, il est possible de profiter des nombreux travaux réalisés pour résoudre le problème de bin-packing. KORF présente un algorithme de type séparation et évaluation capable de déterminer le nombre de containers nécessaires pour ranger N éléments [KORF 2002]. En 2003, KORF présente dans une nouvelle publication des améliorations pour améliorer l'algorithme initial [KORF 2003]. D'après les expériences réalisées, il serait capable de déterminer le nombre minimum de containers nécessaires pour ranger 95 éléments en seulement 1.3 secondes en moyenne sur une machine peu puissante (440 Mhz). En revanche, l'algorithme présenté pose des problèmes de variation de temps d'exécution pour des problèmes de plus de 100 éléments.

De nombreuses autres approches, complètement différentes, existent pour résoudre ce problème. En 2004, SHAW a introduit une nouvelle contrainte pour réduire significativement le nombre de solutions à tester [SHAW 2004]. Dans cette publication, il compare ses résultats avec ceux de KORF et s'aperçoit que les instances difficiles pour son algorithme sont résolues instantanément par l'algorithme de KORF et inversement. On peut donc en conclure qu'une approche hybride, par changement de stratégie, pourrait permettre de résoudre rapidement une très grande quantité de problèmes.

Pour un nombre faible de tâches, il peut donc être pertinent de réaliser un partitionnement exact. Mais dès que le nombre de tâches devient trop grand, il n'est plus possible d'utiliser une méthode exacte.

3.1.3 Heuristiques

Lorsque le nombre de tâches devient très grand, ou si on n'a pas besoin d'un partitionnement exact et que l'on connaît le nombre de processeurs à notre disposition, il peut être intéressant d'essayer de résoudre le problème avec une heuristique.

Les heuristiques les plus classiques sont le First-Fit, le Best-Fit, le Next-Fit, etc. Ainsi que leurs variantes où les tâches sont d'abord triées par ordre décroissant selon leur taille, on les nomme alors Decreasing First-Fit, Decreasing Best-Fit, Decreasing Next-Fit, etc.

Ils ont tous une complexité de l'ordre de $\mathcal{O}(n \times m)$, le tri des tâches est de l'ordre de $\mathcal{O}(n \cdot \log(n))$ et ne change donc pas vraiment la complexité de l'heuristique.

La qualité de ces heuristiques est généralement plutôt correcte [LEE et LEE 1985 ; OH et BAKKER 1998]. COFFMAN, GAREY et JOHNSON ont étudié en détail (et de manière quasi-exhaustive) les performances des heuristiques pour le bin packing [COFFMAN, GAREY et JOHNSON 1997]. La suite de cette partie se concentre sur la description de ces algorithmes uniquement. Nous nous limitons à un nombre fixé de processeurs.

First-Fit L'idée principale de l'algorithme First-Fit est, comme son nom l'indique, d'affecter chaque tâche au premier processeur trouvé tel que l'ordonnanceur local peut l'ordonnancer avec les tâches déjà affectées ($U[j] + U(\tau_i) > CapMax_j$, avec $U[j]$ l'utilisation du processeur P_j). Pour chaque tâche, on reprend la recherche depuis le début de la liste des processeurs (Algorithme 5).

1	Pour tout $\tau_i \in \tau$ Faire
2	$j \leftarrow 1$;
3	Tant Que $U[j] + U(\tau_i) > CapMax_j$ Faire
4	$j \leftarrow j + 1$;
5	Si $j > m$ Alors
6	Pas de solution !;
7	Fin Si
8	Fin Tant Que
9	$P_j \leftarrow P_j \cup \{\tau_i\}$;
10	$U[j] \leftarrow U[j] + U(\tau_i)$;
11	Fin Pour

Algorithme 5 : Algorithme First-Fit

Next-Fit L'algorithme Next-Fit est similaire au First-Fit à la différence près que l'on ne reprend pas la recherche depuis le début à chaque tâche (Algorithme 6).

Best-Fit L'algorithme Best-Fit choisit le processeur disposant de la plus petite capacité disponible et capable d'accepter la tâche tout en restant ordonnançable (Algorithme 7). En pratique, c'est équivalent au First-Fit mais en triant les processeurs par capacité. Si la liste des processeurs est maintenue dans un tas, la complexité supplémentaire pour maintenir la liste triée est de l'ordre de $\mathcal{O}(m \cdot \log(m))$ ce qui ne change pas significativement la complexité totale de l'heuristique.

Worst-Fit Exactement identique au Best-Fit mais en triant par ordre croissant des capacités disponibles les processeurs (Algorithme 8).

3.1.4 Pire cas

Les algorithmes de type partitionnés souffrent d'un très mauvais taux d'utilisation du système dans le pire des cas. Cette limite d'utilisation du système est de seulement $\frac{m+1}{2 \cdot m}$. En effet, soit un système de m processeurs avec $m + 1$ tâches à ordonnancer. Si chaque

```

1  $j \leftarrow 1$ ;
2 Pour tout  $\tau_i \in \tau$  Faire
3    $k \leftarrow 1$ ;
4   Tant Que  $U[j] + U(\tau_i) > CapMax_j$  Faire
5      $j \leftarrow j + 1$ ;
6      $k \leftarrow k + 1$ ;
7     Si  $j > m$  Alors
8        $j \leftarrow 1$ ;
9     Fin Si
10    Si  $k > m$  Alors
11      Pas de solution !;
12    Fin Si
13  Fin Tant Que
14   $P_j \leftarrow P_j \cup \{\tau_i\}$ ;
15   $U[j] \leftarrow U[j] + U(\tau_i)$ ;
16 Fin Pour

```

Algorithme 6 : Algorithme Next-Fit

```

1 Pour tout  $\tau_i \in \tau$  Faire
2    $j \leftarrow 1$ ;
3   Tant Que  $U[j] + U(\tau_i) > CapMax_j$  Faire
4      $j \leftarrow j + 1$ ;
5     Si  $j > m$  Alors
6       Pas de solution !;
7     Fin Si
8   Fin Tant Que
9    $P_j \leftarrow P_j \cup \{\tau_i\}$ ;
10   $U[j] \leftarrow U[j] + U(\tau_i)$ ;
11  Remise en ordre (décroissant) de la liste des processeurs;
12 Fin Pour

```

Algorithme 7 : Algorithme Best-Fit

```

1 Pour tout  $\tau_i \in \tau$  Faire
2    $j \leftarrow 1$ ;
3   Tant Que  $U[j] + U(\tau_i) > CapMax_j$  Faire
4      $j \leftarrow j + 1$ ;
5     Si  $j > m$  Alors
6       Pas de solution !;
7     Fin Si
8   Fin Tant Que
9    $P_j \leftarrow P_j \cup \{\tau_i\}$ ;
10   $U[j] \leftarrow U[j] + U(\tau_i)$ ;
11  Remise en ordre (croissant) de la liste des processeurs;
12 Fin Pour

```

Algorithme 8 : Algorithme Worst-Fit

tâche a un taux d'utilisation de $0.5 + \epsilon$ ($\epsilon > 0$), alors une tâche ne pourra pas être affectée à un processeur. Le taux d'utilisation de ce système est de $(m + 1) \cdot (0.5 + \epsilon)/m$, ce qui nous donne bien $\lim_{\epsilon \rightarrow 0} (m + 1) \cdot (0.5 + \epsilon)/m = \frac{m+1}{2 \cdot m}$.

Pour $m > 4$ processeurs, le taux d'utilisation est en dessous de 60% et tend vers 50%.

3.2 Ordonnancement global

La stratégie visant à partitionner les tâches pour les affecter statiquement à des processeurs montre rapidement certaines limites dues à la non-migration possible des tâches. Par exemple, un système composé de trois tâches identiques avec $C = 2$, $T = 3$, $D = T$ et deux processeurs, n'est pas ordonnançable par partitionnement alors qu'en autorisant l'une des tâches à s'exécuter sur un processeur puis l'autre, on voit que le système est ordonnançable (voir Figure 8).

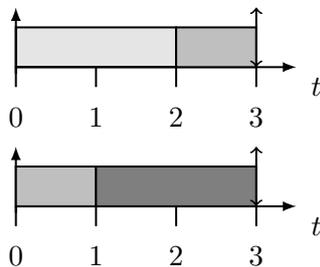


Figure 8 – Exemple de système de tâches non ordonnançable par une politique par partitionnement mais ordonnançable avec migration (ici, politique EDZL).

La stratégie d'ordonnancement global n'emploie qu'un seul ordonnanceur et une seule file de tâches (voir Figure 9). Les tâches ne sont pas destinées à un processeur en particulier, contrairement à la stratégie par partitionnement. C'est cette différence majeure qui permet d'autoriser les migrations. Puisque l'ordonnancement global permet de supprimer la contrainte de non-migration, il semble logique que la classe de problèmes ordonnançables soit plus grande. Malheureusement, il a été montré qu'appliquer RM ou EDF de manière globale, ne permet pas d'obtenir des conditions suffisantes plus avantageuses qu'un ordonnancement par partitionnement [DHALL et LIU 1978]. C'est pour cette raison que de nouveaux algorithmes dédiés à l'ordonnancement multiprocesseur ont vu le jour.

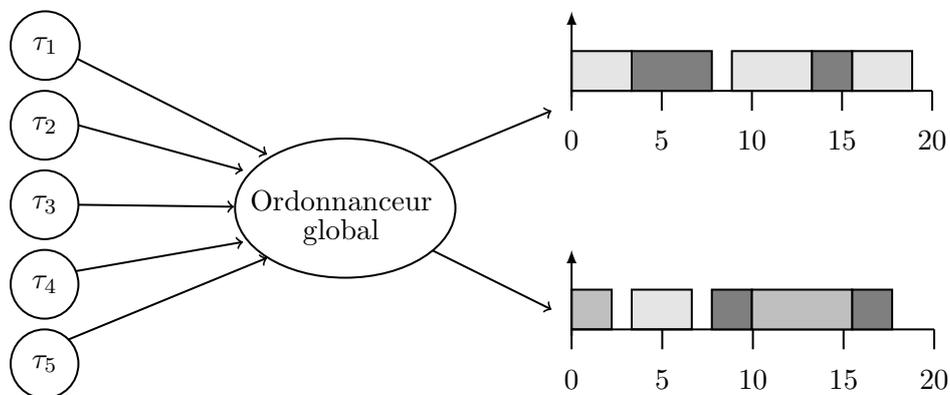


Figure 9 – Représentation graphique de la stratégie d'ordonnancement global

3.3 Généralisation des algorithmes monoprocesseurs

Il est possible de généraliser les techniques d'ordonnancement monoprocesseurs telles que RM et EDF. Si on reprend l'algorithme 4, il suffit d'ajouter une boucle pour parcourir l'ensemble des processeurs et affecter la tâche au premier processeur libre ou exécutant une tâche moins prioritaire (dans ce dernier cas, la tâche préemptée rejoint l'ensemble des tâches prêtes et pourrait alors être placée sur un autre processeur). Dans le cas de processeurs uniformes, afin d'avoir un algorithme conservatif, il convient de trier les processeurs par capacité de calcul.

La résolution des compétitions entre travaux de même priorité peut aussi poser problème dans le cas multiprocesseur. Il a été montré que la technique de résolution a de l'influence sur l'ordonnançabilité [GOOSSENS, FUNK et BARUAH 2002].

Afin de déterminer des conditions suffisantes d'ordonnançabilité, PHILLIPS et al. ont étudié une technique par augmentation de ressources, c'est-à-dire comment peut-on rendre un système ordonnançable par une politique en-ligne en fournissant des processeurs plus puissants.

Appliqué à EDF, ils montrent que si un ensemble de travaux est faisable sur m processeurs identiques, alors le système est ordonnançable avec EDF sur m processeurs identiques dont la puissance de calcul est $(2 - \frac{1}{m})$ fois plus grande que le système d'origine [PHILLIPS et al. 1997].

3.3.1 Critères d'ordonnançabilité RM / EDF

Nous nous intéressons maintenant, aux critères suffisants (mais non nécessaires) pour montrer qu'un système est ordonnançable. Nous nous limitons aux systèmes de tâches périodiques à échéances implicites dans un premier temps.

Rate Monotonic : Tout ensemble de tâches périodiques à échéances implicites vérifiant l'un des critères suivants est ordonnançable sur m processeurs identiques par la politique RM [ANDERSSON, BARUAH et JONSSON 2001 ; BARUAH et GOOSSENS 2003a ; BERTOGNA, CIRINEI et LIPARI 2005].

$$U_{max}(\tau) \leq \frac{m}{3m-2} \text{ et } \sum_{\tau_i \in \tau} U(\tau_i) \leq \frac{m^2}{3m-2} \quad (3.5)$$

$$U_{max}(\tau) \leq \frac{1}{3} \text{ et } \sum_{\tau_i \in \tau} U(\tau_i) \leq \frac{m}{3} \quad (3.6)$$

$$U(\tau) \leq \frac{m}{2}(1 - U_{max}(\tau)) + U_{max}(\tau) \quad (3.7)$$

Earliest Deadline First : Tout ensemble vérifiant l'un des critères suivant est ordonnançable sur m processeurs identiques par la politique EDF [SRINIVASAN et BARUAH 2002 ; BARUAH 2004].

$$U_{max}(\tau) \leq \frac{m}{2m-1} \text{ et } \sum_{\tau_i \in \tau} U(\tau_i) \leq \frac{m^2}{2m-1} \quad (3.8)$$

$$U_{max}(\tau) \leq \frac{1}{2} \text{ et } \sum_{\tau_i \in \tau} U(\tau_i) \leq \frac{m+1}{2} \quad (3.9)$$

3.3.2 Algorithmes d'ordonnancement RM-US[ξ] et EDF-US[ξ]

Les politiques RM-US[ξ] [ANDERSSON, BARUAH et JONSSON 2001] et EDF-US[ξ] [SRINIVASAN et BARUAH 2002] dérivent respectivement des algorithmes RM et EDF (généralisés aux architectures multiprocesseurs). Partant de l'analyse des critères d'ordonnabilité des politiques RM et EDF, ces nouvelles politiques proposent d'ajouter un seuil sur le taux d'utilisation au delà duquel les tâches sont considérées à priorité maximale (avec résolution arbitraire des conflits).

Autrement dit, ces politiques classent les tâches en deux catégories :

- Si $U(\tau_i) > \xi$, alors τ_i est une tâche lourde et sa priorité ainsi que celle de ses travaux est maximale.
- Sinon, τ_i est une tâche légère et le calcul de la priorité reste inchangé par rapport à RM ou EDF.

Ceci permet d'obtenir de meilleures conditions suffisantes d'ordonnabilité, en supprimant en particulier la contrainte sur $U_{max}(\tau)$.

3.3.3 Algorithmes d'ordonnancement à laxité nulle (*Zero Laxity*)

Les politiques RMZL, FPZL et EDZL sont des algorithmes à priorité dynamique, basés sur les algorithmes respectifs RM, FP et EDF, et intégrant la notion de laxité. Pour rappel, la laxité est la marge temporelle d'un travail pendant lequel il peut ne pas être exécuté sans manquer son échéance. Lorsqu'un travail devient à laxité nulle, cela signifie que s'il n'est pas immédiatement exécuté et ce sans interruption, alors il ne pourra pas respecter son échéance.

Le principe de ces algorithmes est d'attribuer une priorité maximale aux travaux de laxité nulle afin d'éviter un non respect d'échéance et d'ordonner selon RM, FP ou EDF sinon.

Des études montrent que EDZL domine EDF [PARK et al. 2005]. En effet, si un système de tâche est ordonnançable par EDF, alors EDF et EDZL se comportent de manière identique. De plus, il existe des exemples de cas où un système de tâches est ordonnançable par EDZL et ne l'est pas par EDF (voir figure 10).

PARK et al. montrent par simulation que EDZL permet aussi d'ordonner un nombre plus important de systèmes que les variantes d'EDF, EDF-US [SRINIVASAN et BARUAH 2002] et fpEDF [BARUAH 2004]. De plus, le nombre de préemptions reste similaire à EDF

contrairement à LLF qui intègre lui aussi la laxité mais qui provoque de trop nombreuses situations de préemption.

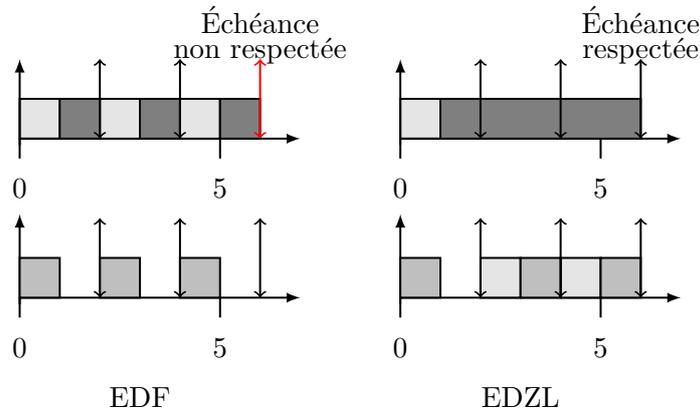


Figure 10 – Exemple où EDZL permet un ordonnancement alors que EDF est en échec. $\tau_1 = (1, 2), \tau_2 = (1, 2), \tau_3 = (5, 6)$

3.4 Ordonnancement global dit équitable (*PFair*)

Les algorithmes de la famille *PFair* (Proportionate Fair) ont une approche différente de l’ordonnancement et sont destinés aux architectures multiprocesseurs [BARUAH et al. 1996]. La différence avec les algorithmes classiques d’ordonnancement est que les algorithmes *PFair* imposent explicitement l’exécution des tâches à un taux régulier.

LEVIN et al. explique dans son paragraphe ”Why Greedy Schedulers Fail” dans [LEVIN et al. 2010] pourquoi les approches qui généralisent les ordonnanceurs monoprocresseur ne permettent pas d’atteindre l’optimalité dans le cas multiprocresseur. En effet, contrairement au cas monoprocresseur, il peut être nécessaire de prendre des décisions sur l’ordonnancement alors qu’il n’y a pas d’événement “habituel” donnant lieu à un réordonnancement (c’est-à-dire, autre qu’une fin de tâche, une activation d’une nouvelle tâche, une laxité nulle, etc.). La contrainte d’équité de l’algorithme *PFair*, expliquée ci-après, va permettre de résoudre cette difficulté.

L’objectif d’un algorithme *PFair* est de se rapprocher d’un ordonnancement idéal. Un ordonnancement est dit idéal, ou équitable, lorsque chaque tâche reçoit exactement $U(\tau_i) \cdot t$ unités de temps processeur dans l’intervalle $[0, t[$. Bien sûr, un tel ordonnancement est impossible dans le cas discret, et un algorithme *PFair* va simplement essayer de s’en rapprocher.

3.4.1 Modélisation de l’ordonnancement *PFair*

Discrétisation du temps Le temps est discrétisé en en intervalles uniformes appelés des slots de temps. Les durées fournies pour illustrer les explications, dans la suite de cette partie consacrée à *PFair*, sont exprimées en nombre de slots. L’intervalle de temps $[t, t + 1[$, avec $t \in \mathbb{N}$, correspond au slot t .

Contrainte PFair De façon à modéliser une séquence d'ordonnancement, nous définissons la fonction binaire $S : \tau \times \mathbb{N} \rightarrow \{0, 1\}$ qui renvoie 1 lorsqu'un travail τ_i est ordonné sur le slot t et 0 sinon.

Grâce à cette fonction, il est possible d'introduire la notion de décalage en temps (*lag*), qui mesure l'écart entre l'exécution idéale et la séquence construite (Définition 3.1).

Définition 3.1 Le décalage d'une tâche τ_i à l'instant t se note :

$$\text{lag}(\tau_i, t) = U(\tau_i) \cdot t - \sum_{l=0}^{t-1} S(\tau_i, l) \quad (3.10)$$

Grâce à cette notion de décalage, il nous est possible de définir la contrainte PFair (Définition 3.2). La figure 11 montre l'ordonnancement d'une tâche qui respecte cette contrainte.

Définition 3.2 Un algorithme est dit PFair lorsque, $\forall t \in \mathbb{N}, \forall \tau_i \in \tau, |\text{lag}(\tau_i, t)| < 1$. Autrement dit, à chaque instant, l'erreur d'exécution est strictement inférieure à un quantum de temps, ou encore que $\sum_{l=0}^{t-1} S(\tau_i, l) = \lfloor U(\tau_i) \cdot t \rfloor$ ou $\lceil U(\tau_i) \cdot t \rceil$.

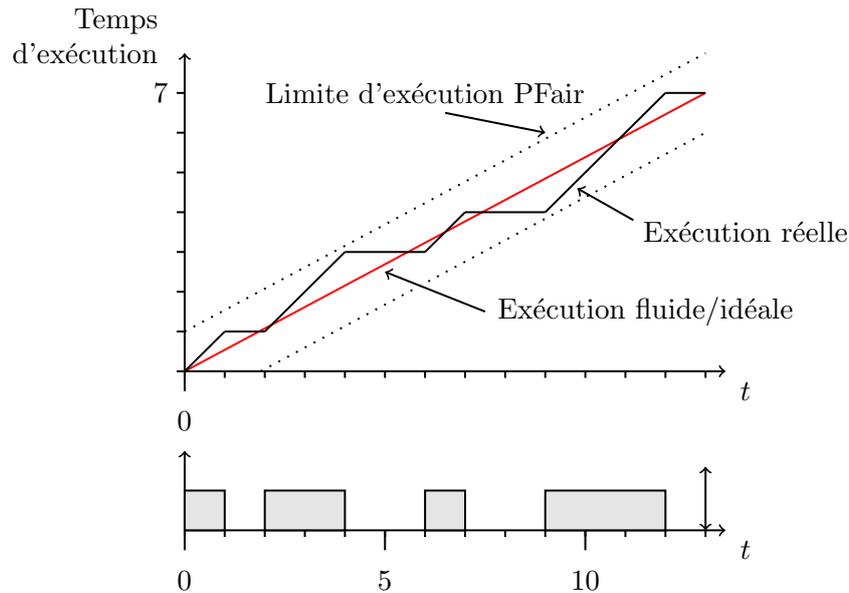


Figure 11 – Ordonnancement PFair d'une tâche τ_i ($C_i = 7, T_i = 13$).

Pseudo-réveil et pseudo-échéance : La construction d'un ordonnancement PFair passe par la division de chaque tâche en sous-tâches de durée d'exécution un quantum. Une tâche τ_i est ainsi divisée en une séquence infinie de sous-tâches τ_i^j avec τ_i^1 la première sous-tâche. Chaque sous-tâche possède une fenêtre d'exécution possible : $[r(\tau_i^j), d(\tau_i^j)]$. On appelle, $r(\tau_i^j)$ la date de pseudo-réveil et $d(\tau_i^j)$ la date de pseudo-échéance. La fenêtre d'exécution de τ_i^j se note $w(\tau_i^j)$ et sa longueur est définie par : $|w(\tau_i^j)| = d(\tau_i^j) - r(\tau_i^j) + 1$.

La contrainte PFair implique [BARUAH et al. 1996] :

$$r(\tau_i^j) = \left\lfloor \frac{j-1}{U(\tau_i)} \right\rfloor \text{ et } d(\tau_i^j) = \left\lceil \frac{j}{U(\tau_i)} \right\rceil - 1 \quad (3.11)$$

La figure 12 montre les fenêtres d'exécution pour l'exemple précédent ($C_i = 7, T_i = 13$). On peut constater qu'en effet, chaque sous-tâche s'est bien exécutée à l'intérieur de sa fenêtre d'exécution.

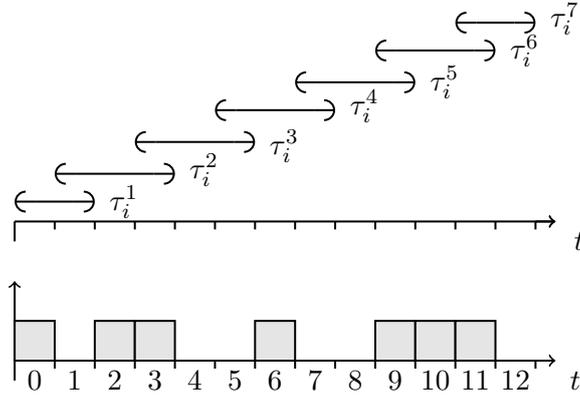


Figure 12 – Fenêtres d'exécution des sous-tâches de τ_i .

Dans le cadre de l'ordonnement PFair, une tâche est qualifiée de lourde si et seulement si son facteur d'utilisation est supérieur ou égal à $1/2$, sinon il s'agit d'une tâche légère. Il est intéressant de remarquer alors que seule une tâche lourde possède des fenêtres d'exécution de taille 2 [ANDERSON et SRINIVASAN 1999].

Remarque : Un ordonnanceur PFair n'est pas conservatif dans la mesure où il peut être amené à ne pas exécuter de tâches pour perdre de son avance. Une variante conservative a été proposée [ANDERSON et SRINIVASAN 2000a] et propose d'alléger la contrainte sur le décalage par : $lag(\tau_i, t) < 1$. Les avantages sont des temps de réponse plus courts et une mise en œuvre simplifiée.

3.4.2 Optimalité de l'ordonnement PFair

L'ordonnabilité d'un algorithme respectant la contrainte PFair doit être vérifiée à chaque échéance. L'échéance d'une tâche à échéance implicite τ_i se produit aux instants $k \cdot T_i$ ($k \in \mathbb{N}$).

Si l'échéance a été respectée, alors à ces instants, la tâche doit avoir reçu $k \cdot C_i$ unités de temps processeur. Or, d'après la formule donnée par la définition 3.2, nous savons que la tâche a reçu $\lfloor U(\tau_i) \cdot k \cdot T_i \rfloor$ ou $\lceil U(\tau_i) \cdot k \cdot T_i \rceil$ unités de temps processeur. En explicitant $U(\tau_i) = C_i/T_i$, on retrouve $k \cdot C_i$ qui est une valeur entière.

Un ordonnancement PFair de n tâches périodiques, synchrones et à échéances implicites, sur m processeurs identiques existe si et seulement si la condition 3.12 est respectée [BARUAH et al. 1996].

$$\sum_{i=1}^n U(\tau_i) \leq m \quad (3.12)$$

Bien qu'en théorie un tel ordonnancement est optimal, il convient de nuancer ce résultat en pratique. En effet, les surcoûts d'exécution tels que les prises de décision à chaque quantum, ou encore les nombreuses préemptions et possibles migrations, ne sont absolument pas prises en compte. Et enfin, les processeurs doivent être parfaitement synchronisés entre eux pour respecter les précédences des morceaux de tâches (des travaux visent à adapter PFair pour tenir compte de ce problème de synchronisation [HOLMAN et ANDERSON 2005]).

3.4.3 Algorithme d'ordonnancement EPDF

Un certain nombre d'ordonnanceurs PFair (ordonnanceurs qui respectent la condition PFair) ont été définis. Nous nous intéressons dans un premier temps à l'algorithme EPDF (*Earliest Pseudo-Deadline First*) pour sa simplicité [ANDERSON et SRINIVASAN 2000b]. Cependant, il n'est optimal que pour deux processeurs au maximum.

L'idée est d'appliquer l'algorithme EDF sur les sous-tâches en utilisant les pseudo-échéances. Les cas d'égalités sont traités de manière arbitraire. La figure 13 montre un exemple d'ordonnancement EPDF avec deux processeurs sur les tâches suivantes :

- $\tau_1 : C_1 = 5, T_1 = 10$
- $\tau_2 : C_2 = 3, T_2 = 4$
- $\tau_3 : C_3 = 4, T_3 = 6$

L'utilisation cumulée du système est environ de 1.92, la condition nécessaire et suffisante est donc respectée.

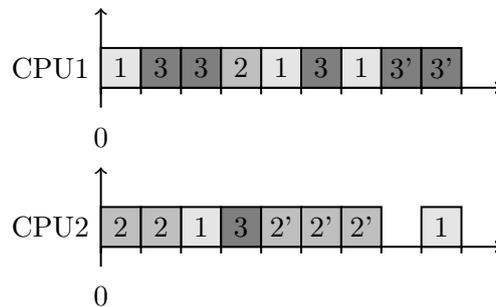


Figure 13 – Ordonnancement EPDF pour les tâches : $\tau_1 : (C_1 = 5, T_1 = 10)$, $\tau_2 : (C_2 = 3, T_2 = 4)$, $\tau_3 : (C_3 = 4, T_3 = 6)$.

Sur la figure 14, nous pouvons constater le respect de la contrainte PFair.

3.4.4 Algorithme d'ordonnancement PD²

Les algorithmes PF, PD [BARUAH et al. 1996] et PD² [ANDERSON et SRINIVASAN 1999] complètent EPDF par l'évaluation et la comparaison d'autres critères. Nous ne détaillerons que PD² qui est reconnu comme étant le plus efficace (plus faible complexité pour les prises de décision).

PD² utilise deux critères supplémentaires pour déterminer quelle tâche est prioritaire en cas d'égalité de leurs pseudo-échéances.

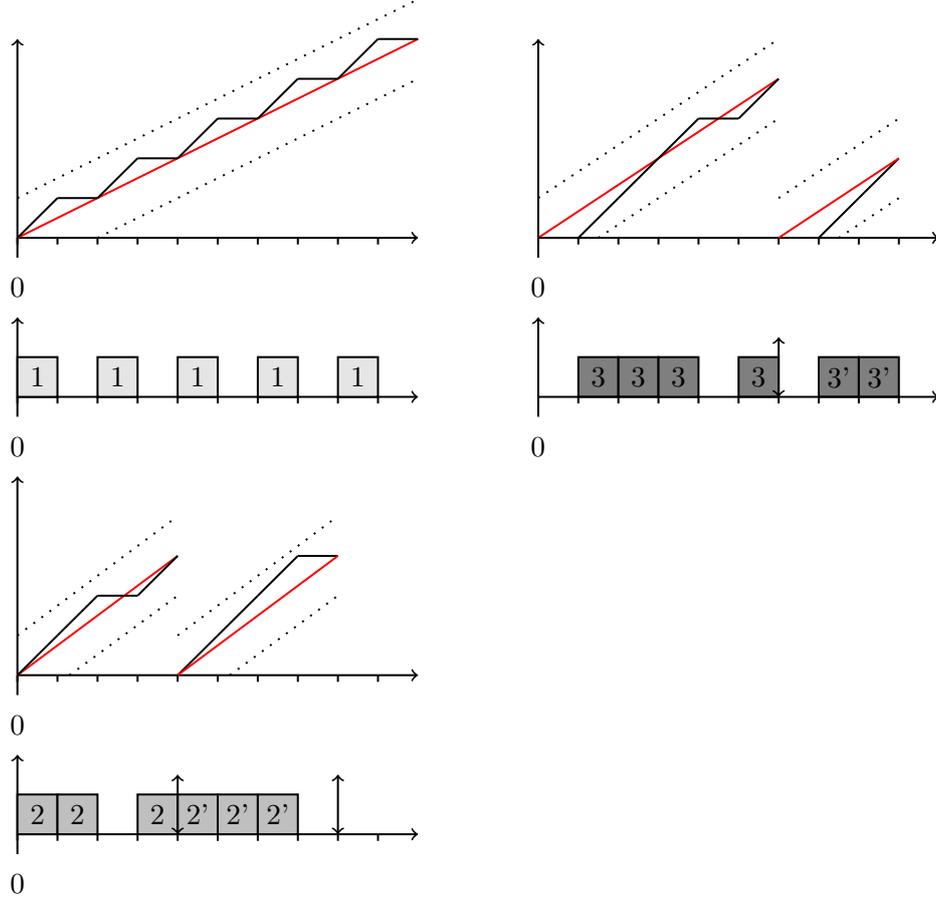


Figure 14 – Ordonnancement EPDF avec comparaison par rapport à l’ordonnancement fluide.

Premier critère : *successor bit*. D’après l’équation 3.11, $r(\tau_i^{j+1})$ est égal à $d(\tau_i^j)$ ou $d(\tau_i^j) + 1$. Autrement dit, les fenêtres successives se chevauchent de précisément un slot ou sont disjointes. On définit la fonction binaire $b(\tau_i^j)$ qui vaut 1 en cas de chevauchement et 0 sinon.

$$b(\tau_i^j) = \begin{cases} 1, & \text{si } r(\tau_i^{j+1}) = d(\tau_i^j) \\ 0, & \text{sinon} \end{cases} \quad (3.13)$$

La fonction définie par 3.13 est le premier critère, appelé *successor bit*.

Deuxième critère : Échéance de groupe (*Group deadline*) Soit une séquence $\tau^i, \tau^{i+1}, \dots, \tau^j$ de sous-tâches d’une tâche lourde et telle que, $\forall k \in [i, j], |w(\tau^k)| = 2 \wedge b(\tau^{k-1}) = 1$ (c’est-à-dire que la fenêtre est de taille 2 et qu’il y a chevauchement d’une sous-tâche à la suivante), et soit $|w(\tau^{j+1})| = 3$, soit $b(\tau^j) = 0$ (c’est-à-dire que la sous-tâche successeur de la dernière sous-tâche est de taille 3 ou que la dernière sous-tâche du groupe ne chevauche pas la suivante).

Une telle séquence a pour particularité que si une sous-tâche est exécutée pendant le dernier slot de sa fenêtre, alors toutes les sous-tâches consécutives n’auront pas d’autre

choix que de s'exécuter dans le dernier slot de leur fenêtre respective. Les sous-tâches d'une telle séquence partagent donc en quelque sorte une échéance de groupe : $d(\tau^j) + 1$.

On définit $D(\tau^i)$, l'échéance de groupe de la sous-tâche τ^i . Attention, ceci ne s'applique qu'au cas des tâches lourdes, pour le cas des tâches légères, $D(\tau^i) = 0$.

Priorité des tâches La relation d'ordre entre les priorités de deux tâches à l'instant τ s'exprime à partir des deux critères précédents. Soit deux sous-tâches prêtes τ_i^j et τ_q^k , τ_i^j est considérée prioritaire sur τ_q^k si l'une des conditions suivante est respectée :

1. $d(\tau_i^j) < d(\tau_q^k)$ (échéance plus rapprochée pour τ_i^j)
2. $d(\tau_i^j) = d(\tau_q^k) \wedge b(\tau_i^j) > b(\tau_q^k)$ (échéances identiques mais τ_i^j chevauche son successeur)
3. $d(\tau_i^j) = d(\tau_q^k) \wedge b(\tau_i^j) = b(\tau_q^k) \wedge D(\tau_i^j) > D(\tau_q^k)$ (sinon priorité à celui dont l'échéance de groupe est la plus éloignée)

3.5 Ordonnement global DP-Fair

HONG et LEUNG ont prouvé qu'il n'existe pas d'ordonneur en-ligne optimal pour un ensemble de travaux avec deux ou plus échéances distinctes sur n'importe quel système multiprocesseur de deux ou plus processeurs [HONG et LEUNG 1988]. Les algorithmes de la famille de PFair atteignent l'optimalité grâce à de nombreux découpages et l'ajout d'une contrainte forte sur la fluidité de l'exécution des tâches. Ces découpages permettent de traiter des sous-tâches partageant les mêmes échéances.

Cependant, ces découpages, ou évènements d'ordonnement, sont nombreux et coûteux en calcul. Cela engendre de nombreuses préemptions et donc un surcoût important à l'exécution. ZHU, MOSSE et MELHEM montrent qu'il n'est pas nécessaire de se rapprocher autant à l'exécution fluide des travaux afin d'améliorer les performances en réduisant le nombre d'évènements d'ordonnement [ZHU, MOSSE et MELHEM 2003]. Plus précisément, il est nécessaire de découper le temps en intervalles tels que sur chaque intervalle, les tâches partagent la même date d'échéance. DP-Fair, pour "Deadline Partitionning Fair", diffère de PFair dans le sens où la contrainte de fluidité est assouplie pour ne porter que sur ces échéances [LEVIN et al. 2010].

3.5.1 Principe

DP-Fair est la théorie qui unifie les techniques basées sur le Deadline Partitionning et celles sur PFair. Le Deadline Partitionning consiste à découper le temps en intervalles définis par l'ensemble des échéances des tâches (pour un système de tâches à échéances implicites). Dans chaque intervalle, on affecte une durée locale d'exécution à chaque travail et les sous-travaux partagent la même date d'échéance, à savoir la fin de l'intervalle. DP-Fair ajoute comme contrainte qu'à la fin de chaque intervalle, les tâches doivent avoir rejoint leur exécution fluide.

La première étape de l'algorithme est de partitionner le temps en intervalles définis par les dates d'échéance de l'ensemble des tâches. Plus formellement, et en s'inspirant de la notation utilisée dans [LEVIN et al. 2010], on note $t_0 = 0$ et on pose t_1, t_2 , etc, les dates des différentes échéances par ordre chronologique ($t_j < t_{j+1}$). Le $j^{\text{ème}}$ intervalle, noté σ_j ,

correspond donc à l'intervalle $[t_{j-1}, t_j[$, et on note sa longueur $L_j = t_j - t_{j-1}$. Par la suite, i correspond à l'indice de la tâche τ_i et j à l'indice de l'intervalle σ_j .

Pour chaque intervalle σ_j , l'ensemble des m processeurs met à disposition $m \cdot L_j$ unités temporelles d'exécution. Il convient alors de définir pour chaque tâche, le nombre d'unités temporelles d'exécution qui seront à exécuter dans cet intervalle et de définir l'ordonnancement au sein de l'intervalle. Enfin, en fonction de l'ordonnancement utilisé au sein de l'intervalle, il peut être nécessaire d'ajouter des tâches fictives pour modéliser les temps d'inactivité et ainsi obtenir une charge totale du système égale à m .

Deux étapes sont nécessaires pour ce type d'algorithme :

- La dotation temporelle est l'étape où on définit pour chaque tâche τ_i sa durée d'exécution locale pour l'intervalle σ_j . On note l_i^j cette durée. Dans le cas d'un ordonnancement DP-Fair, $l_i^j = U(\tau_i) \cdot L_j$.
- La distribution temporelle est la façon de répartir, pour l'ensemble des tâches, les durées d'exécution locales sur les m processeurs sur l'intervalle considéré. On dit d'un ordonnanceur qu'il est DP-Correct s'il ordonnance correctement les tâches dans l'intervalle.

La dotation temporelle ainsi que la distribution temporelle peuvent être décidées au début de l'intervalle ou construit dynamiquement sur l'intervalle, en fonction de l'algorithme utilisé.

3.5.2 Abstraction du *T-L Plane*

Afin de mieux comprendre l'idée sous-jacente, il est intéressant de présenter l'abstraction du "T-L Plane" qui permet une visualisation de la consommation de la dotation temporelle [CHO et RAVINDRAN 2006]. Cette représentation est particulièrement utile pour comprendre l'algorithme LLREF qui est présenté par la suite. La figure 15 montre un exemple de T-L Plane. Sur l'axe des abscisses, figure le temps et sur l'axe des ordonnées la durée restante d'exécution. À chaque tâche, on associe un "jeton", dont la position est fixée initialement aux coordonnées (t_{j-1}, l_i^j) . À chaque instant, l'ordonnée d'un jeton représente sa durée restante d'exécution, qui ne peut donc que diminuer ou rester constante.

Le rôle de la distribution temporelle est de déplacer ces jetons de telle sorte qu'ils atteignent l'axe des abscisses avant la fin de l'intervalle. Si une tâche n'est pas exécutée, alors son jeton progresse horizontalement, et lorsqu'une tâche est exécutée, son jeton descend proportionnellement au temps (son temps restant d'exécution diminue). Sur un système à m processeurs, il n'est bien sûr pas possible d'exécuter plus de m tâches à la fois.

Le triangle rectangle isocèle représenté sur la figure 15 montre le domaine de déplacement limite d'un jeton. En effet, tout jeton qui sort de ce domaine ne peut pas représenter une exécution conforme : s'il franchit l'axe des abscisses, il aura consommé plus de temps d'exécution local que prévu et s'il franchit la diagonale, alors il ne pourra pas respecter l'échéance locale.

Dans ce qui suit seront présentés quelques algorithmes DP-Fair assez simples. De nombreux travaux visent à les étendre [LEVIN et al. 2010].

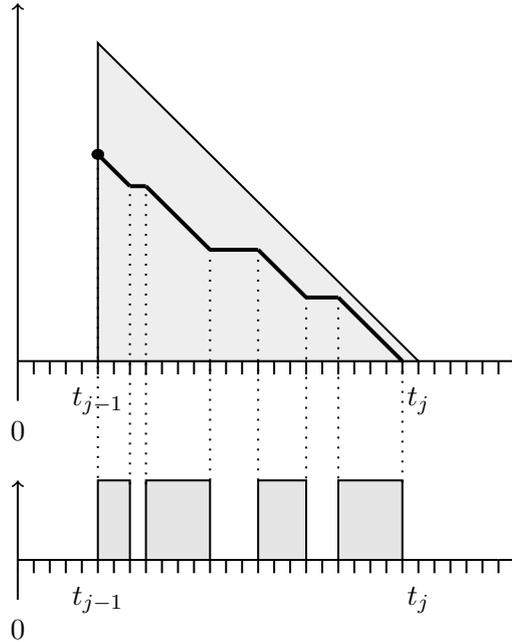


Figure 15 – Exemple de T-L Plane avec une seule tâche.

3.5.3 Algorithme d'ordonnement DP-WRAP

L'algorithme DP-WRAP a été conçu dans l'optique d'être un algorithme DP-Fair simple à comprendre [LEVIN et al. 2010]. De plus celui-ci peut être modifié pour pouvoir traiter des tâches sporadiques.

Dotation temporelle DP-WRAP est un algorithme DP-Fair, la dotation temporelle est donc proportionnelle aux taux d'utilisation des tâches, c'est-à-dire que $l_i^j = U(\tau_i) \cdot L_j$. À la fin de l'intervalle, les différentes tâches reçoivent donc un temps d'exécution fluide.

Distribution temporelle La distribution temporelle est réalisée selon l'algorithme de McNaughton [MCNAUGHTON 1959]. L'algorithme de McNaughton permet d'ordonner de manière optimale et très simplement un ensemble de tâches partageant la même échéance sur une architecture comportant plusieurs processeurs identiques. La figure 16 résume graphiquement la manière dont l'ordonnement est calculé. La première étape consiste à aligner, de manière arbitraire, des blocs le long d'une droite. Ces blocs représentent chacun une tâche et la largeur du bloc correspond au taux d'utilisation de la tâche correspondante. Dans un second temps, on découpe la succession de blocs en morceaux de longueur 1 et on associe chaque morceau à un processeur. Dans l'exemple, nous avons 2 morceaux, le premier contient les tâches τ_1 , τ_2 et une partie de τ_3 , et le second contient le reste de τ_3 et τ_4 . Enfin, nous multiplions toutes les tailles par L_j pour manipuler les durées locales d'exécution et donc déterminer à quel moment il convient d'ordonner une tâche ou une autre².

En ce qui concerne les tâches affectées à deux processeurs différents, l'exécution sera

2. Le lecteur attentif aura peut être remarqué qu'il est possible de manipuler directement les durées locales d'exécution et de découper en morceaux de taille L_j mais cela compliquerait inutilement le schéma.

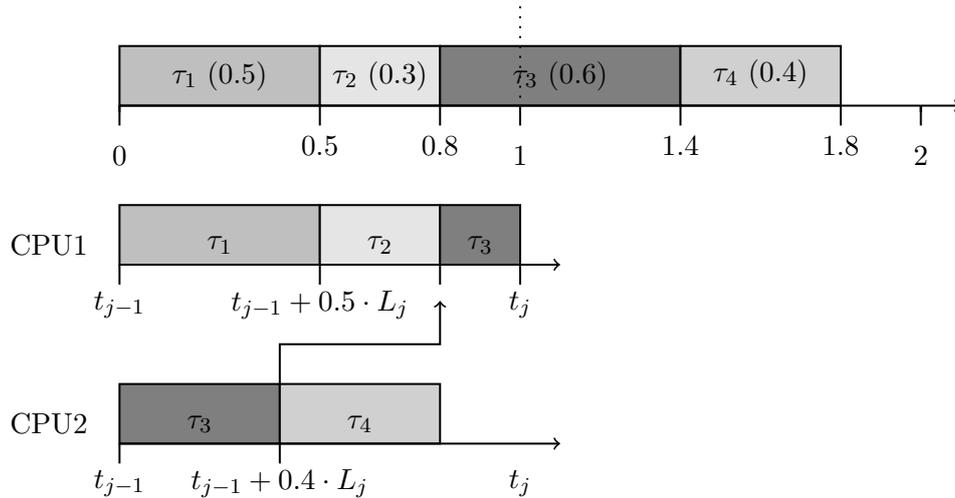


Figure 16 – Représentation de la distribution temporelle à l’aide de l’algorithme de McNaughton.

interrompue puis reprise plus tard sur un autre processeur. Cela ne pose pas de problème dans la mesure où le taux d’utilisation maximal des tâches est inférieur à 1.

3.5.4 Algorithme d’ordonnancement LLREF

LLREF (Largest Local Remaining Execution Time First) est aussi un ordonnanceur DP-Fair relativement simple à comprendre [CHO et RAVINDRAN 2006].

Dotation temporelle La dotation temporelle est la même que pour DP-WRAP, c’est-à-dire proportionnelle aux taux d’utilisation des tâches ($l_i^j = U(\tau_i) \cdot L_j$).

Distribution temporelle La distribution temporelle est cependant différente et privilégie les tâches dont la durée locale d’exécution restante est la plus grande. La figure 17 montre l’ordonnancement de trois tâches sur deux processeurs et permet de mieux comprendre les explications qui suivent. Au début de l’intervalle, l’ordonnanceur choisit les m tâches dont la durée locale d’exécution est la plus grande (ici, τ_2 et τ_3). Lorsque le jeton qui représente une tâche rencontre la diagonale (*Ceiling hitting*) ou lorsqu’un jeton rencontre l’axe des abscisses (*Bottom hitting*), on reconsidère les tâches à exécuter. À ce moment là, l’ordonnanceur choisit d’exécuter les tâches dont la durée locale d’exécution restante est la plus grande.

Levin et al. [LEVIN et al. 2010], les auteurs de DP-WRAP, considèrent que LLREF fait des calculs inutiles : à chaque évènement, il faut rechercher les m tâches dont la durée d’exécution locale restante est la plus grande. Néanmoins, LLREF sert de base à de nombreux travaux qui visent à l’améliorer ou à étendre son domaine d’application (processeurs uniformes, ou tâches sporadiques). Selon des simulations, DP-WRAP pourrait engendrer seulement 1/3 des changements de contexte et des migrations d’un ordonnancement LLREF.

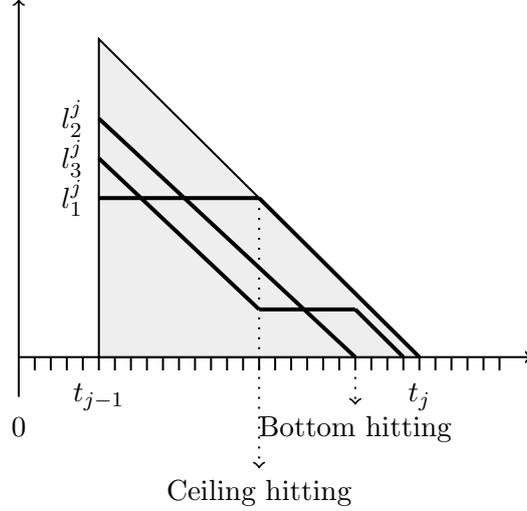


Figure 17 – Exemple d’ordonnement LLREF de 3 tâches sur 2 processeurs, représenté sous la forme de TL-Plane.

3.5.5 Algorithme d’ordonnement BF

Dans le cas des algorithmes DP-Fair, les durées telles que l_i^j sont des valeurs de \mathbb{R} . On dit de ces algorithmes, qu’ils ne sont pas basés sur des quanta (*quantum-based*).

La politique BF (*Boundary Fair*) [ZHU, MOSSE et MELHEM 2003], introduite peu avant les politiques DP-Fair, part du même constat, à savoir que c’est au moment de l’échéance que la fluidité compte. BF est une évolution de PFair et reste basé sur des quanta. BF est présenté avec les autres algorithmes DP-Fair de par sa similitude avec ces derniers, avec pour seule différence qu’il fait la dotation temporelle en nombres entiers.

L’erreur d’allocation pour la tâche τ_i à l’instant t_j , notée RW_i^j , correspond à la différence entre $t_j \cdot U(\tau_i)$ et le temps effectif d’exécution de la tâche à l’instant t_j (soit $lag(\tau_i, t_j)$ dans la notation PFair).

Définition 3.3 *Un ordonnancement est Boundary-Fair si et seulement si la valeur absolue de l’erreur d’allocation pour chaque tâche à n’importe quelle date correspondant à une échéance est strictement inférieure à 1 ($\forall i \in [1, n], \forall j \in \mathbb{N}^*, |RW_i^j| < 1$).*

Dans le cas de DP-WRAP et LLREF, l’erreur d’allocation pour chaque tâche, aux frontières des intervalles, est toujours nulle.

Dotation temporelle La dotation temporelle sur l’intervalle σ_j , sur m processeurs, se fait comme ceci (en reprenant en partie les notations introduites pour DP-Fair) :

- Chaque tâche reçoit un nombre entier d’unités temporelles d’exécution : $m_i^j = \max(0, \lfloor RW_i^{j-1} + L_j \cdot U(\tau_i) \rfloor)$. Si cette quantité est allouée à la tâche τ_i alors celle-ci aura une erreur comprise dans $[0, 1[$ à la fin de l’intervalle σ_j . Les tâches telles que $RW_i^j > 0$ et $m_i^j < L_j$ sont dites éligibles.
- Du fait de la troncature, il peut rester des unités d’exécution disponibles sur l’intervalle. On note RU, les unités restantes : $RU = m \cdot L_j - \sum_{i=1}^n m_i^j$.

- Enfin, les RU unités sont réparties sur les tâches éligibles les plus prioritaires, à raison d’une unité par tâche. La notion de priorité ne sera pas détaillée ici, mais est disponible dans [BARUAH et al. 1996]. On note $\sigma_i^j = 1$ si la tâche τ_i a obtenu une unité supplémentaire et 0 sinon.
- $l_i^j = m_i^j + \sigma_i^j$

Distribution temporelle La distribution temporelle se fait selon l’algorithme de McNaughton qui a été présenté avec l’algorithme DP-WRAP.

3.6 Ordonnancement semi-partitionné

Dans la partie consacrée à l’ordonnancement partitionné, nous avons montré les lacunes de ce type d’ordonnancement. Le problème est clairement dû à l’absence de migration des tâches d’un processeur à un autre. Reprenons l’exemple du pire cas, à savoir un système de tâches de $m + 1$ tâches avec un taux d’utilisation de chaque tâche égale à $0.5 + \epsilon$. Le système n’est pas ordonnançable par une politique partitionnée car une tâche n’est affectée à aucun processeur. Pourtant, on voit clairement que si on donnait la possibilité à cette tâche de s’exécuter sur plusieurs processeurs, le problème serait résolu.

À l’opposé, l’ordonnancement global offre une totale liberté dans la migration des tâches, et c’est cela qui lui permet d’atteindre l’optimalité. Cependant, cette flexibilité peut avoir un coût à l’exécution et il est donc souhaitable de limiter le nombre de migrations pour les mêmes raisons qui ont poussé les chercheurs à limiter le nombre de préemptions [DEVI et ANDERSON 2005].

Les algorithmes semi-partitionnés se situent entre ces deux extrêmes et proposent d’autoriser une migration contrôlée de certaines tâches. Ils peuvent donc être vus comme une amélioration des algorithmes partitionnés en ajoutant plus de flexibilité mais ils s’inspirent aussi des politiques globales.

Les principaux algorithmes semi-partitionnés ont été mis en œuvre dans LITMUS^{RT} [CALANDRINO et al. 2006] et les résultats indiquent que les politiques semi-partitionnées offrent souvent de meilleurs choix (par exemple, le compromis entre le nombre de préemptions et l’ordonnançabilité) [BASTONI, BRANDENBURG et ANDERSON 2011].

L’algorithme EDF-fm, présenté dans l’article [ANDERSON, BUD et DEVI 2005], fait partie de la catégorie des algorithmes avec migration restreinte aux frontières des travaux. Cependant, cet algorithme ne permet pas de garantir le respect des échéances.

Ce document se limitera aux algorithmes à migration libre, qui sont aussi appelés, algorithmes à migration par portion de travail (*job portion migration*). L’idée générale est de remplir chaque processeur par un ensemble de tâches et lorsqu’une tâche n’est plus en mesure d’être affectée à un processeur, alors on s’autorise à la découper en deux ou plusieurs sous-tâches, affectées à des processeurs différents. L’algorithme devra alors exécuter ces sous-tâches sans chevauchement.

L’algorithme EKG [ANDERSSON et TOVAR 2006] découpe les processeurs en groupes de K processeurs et limite la migration au sein d’un même groupe. De plus, une tâche ne peut migrer qu’entre deux processeurs. Cet algorithme est expliqué plus en détail dans la suite de ce document. EKG est conçu pour ordonnancer des tâches périodiques, il est étendu par EKG-sporadic pour gérer les tâches sporadiques [ANDERSSON et BLETSAS 2008]. EDF-SS

[ANDERSSON, BLETSAS et BARUAH 2008] est une généralisation de EKG-sporadic pour gérer les tâches à échéances arbitraires. En moyenne, EDF-SS est plus performant que EKG-sporadic pour des tâches à échéances contraintes mais n'a pas de limite d'utilisation prouvée [BLETSAS et ANDERSSON 2011].

Les algorithmes Ehd2-SIP (aussi appelé EDDHP) [KATO et YAMASAKI 2007] et EDDP [KATO et YAMASAKI 2008a] sont similaires à EKG mais diffèrent légèrement dans leurs algorithmes d'affectation et d'ordonnancement. Selon la valeur du paramètre K de EKG, ces algorithmes permettent de réduire le nombre de préemptions comparé à EKG, mais ne sont pas optimaux. RMDP est une variante de EDDHP mais basée sur l'ordonnancement RM en remplacement de EDF [KATO et YAMASAKI 2008b].

L'algorithme EDHS [KATO et YAMASAKI 2008c] change vraiment le procédé d'affectation. Pour les algorithmes précédents, une tâche ne pouvait migrer que sur deux processeurs, alors que EDHS propose de faire un partitionnement classique puis de répartir les tâches non affectées sur plusieurs processeurs. Cependant, chaque processeur ne peut accueillir qu'une tâche migrante. EDHS est détaillé à la suite. DM-PM [KATO et YAMASAKI 2009] est une variante de EDHS qui utilise un ordonnancement à priorité fixe ce qui a pour avantage de simplifier les calculs. En effet, en découpant les tâches en sous-tâches, on obtient naturellement des tâches plus prioritaires selon DM.

SPA2 [GUAN et al. 2010] est une politique qui se base sur des priorités fixes et qui a pour particularité de partager la même condition suffisante d'ordonnançabilité que RM en monoprocesseur. Comparé à RMDP, DM-PM et PDMS_HPTS_DS [LAKSHMANAN, RAJKUMAR et LEHOCZKY 2009], la limite d'utilisation dans le pire cas est meilleure.

EDF-WM [KATO, YAMASAKI et ISHIKAWA 2009] s'inspire des avantages apportés dans DM-PM par l'utilisation de l'algorithme DM mais en se basant à nouveau sur EDF. Cet algorithme d'ordonnancement sera décrit plus en détail par la suite.

Enfin, NPS-F [BLETSAS et ANDERSSON 2009b], permet d'atteindre une utilisation qui varie entre 75% et 100% (selon le compromis entre utilisation et préemptions). Cet algorithme sera également présenté plus en détail dans la suite de ce document.

3.6.1 Algorithme d'ordonnancement EKG

L'algorithme EKG [ANDERSSON et TOVAR 2006] peut être vu comme une amélioration de l'algorithme partitionné EDF next-fit ou comme une variante de DP-Fair avec moins de migrations de tâche et des intervalles plus grands. EKG signifie **E**DF avec découpage des tâches et **K** processeurs par **G**roupe. EKG permet d'ordonnancer de manière optimale un ensemble de tâches périodiques à échéances implicites sur m processeurs, lorsqu'on fixe le paramètre K égal au nombre de processeurs. Lorsque K est inférieur à m , alors la propriété d'optimalité est perdue mais au profit d'un nombre moindre de préemptions.

Principe de base Contrairement aux algorithmes partitionnés, EKG permet à des tâches de s'exécuter sur deux processeurs différents (à des moments différents, sans parallélisme). L'algorithme se divise en deux grandes étapes : l'affectation des tâches à un ou deux processeurs (hors-ligne) et l'ordonnancement des tâches sur les processeurs (en ligne).

Affectation des tâches EKG distingue deux catégories de tâches, les tâches lourdes et les tâches légères. Les tâches lourdes se verront attribuer un processeur dédié chacune, tandis que plusieurs tâches légères pourront se voir affecter le même processeur. De plus, les tâches légères pourront être découpées en deux tâches τ_i' et τ_i'' pour mieux utiliser les processeurs.

Pour déterminer si une tâche est lourde ou légère, on compare son taux d'utilisation à la valeur SEP. Si le taux d'utilisation de la tâche est supérieur ou égal à SEP, alors la tâche est dite lourde et sinon elle sera dite légère.

$$SEP = \begin{cases} \frac{k}{1+k} & k < m \\ 1 & k = m \end{cases} \quad (3.14)$$

L'affectation des tâches aux processeurs est inspirée de l'algorithme First-Fit. Dans un premier temps, on réserve les processeurs pour les tâches lourdes (un par tâche). Puis les tâches légères sont affectées aux processeurs suivants, et certaines tâches sont découpées de manière à obtenir une charge du processeur de 1. Une tâche peut être découpée pour s'exécuter sur les processeurs p et $p + 1$ à condition que p et $p + 1$ appartiennent au même groupe. Ainsi, si une tâche ne "rentre" pas sur un processeur p , alors elle est découpée pour s'exécuter sur p et $p + 1$ si les processeurs sont du même groupe, sinon la tâche sera exécutée sur $p + 1$.

Soient les variables suivantes, nécessaires à l'algorithme 9 :

- $U[p]$, la charge du processeur p . Initialement, $\forall p, U[p] = 0$.
- $\tau[p]$, l'ensemble des tâches affectées au processeur p . Initialement, $\forall p, \tau[p] = \{\}$.
- τ^{heavy} , l'ensemble des tâches lourdes.
- τ^{light} , l'ensemble des tâches légères.
- L , le nombre de tâches lourdes ($|\tau^{heavy}|$).
- τ_i , la $i^{\text{ème}}$ tâche. Mais tel que $\tau_i \in \tau^{heavy}$ ssi $i \in [1, L]$.

```

1 Pour  $i \leftarrow 1$  à  $L$  Faire
2   |  $p \leftarrow i$ ;
3   |  $U[p] \leftarrow U(\tau_i)$ ;
4   |  $\tau[p] \leftarrow \{\tau_i\}$ ;
5 Fin Pour
6  $p \leftarrow L + 1$ 
7 Pour  $i \leftarrow L + 1$  à  $n$  Faire
8   | Si  $U[p] + U(\tau_i) \leq 1$  Alors
9     |  $U[p] \leftarrow U[p] + U(\tau_i)$  ;  $\tau[p] \leftarrow \tau[p] \cup \{\tau_i\}$ ;
10  | Sinon
11    | Si  $p + 1 \leq m$  Alors
12      | Si  $p - L \bmod K = 0$  Alors
13        |  $U[p + 1] \leftarrow U(\tau_i)$  ;  $\tau[p + 1] \leftarrow \{\tau_i\}$ ;
14        | Sinon
15          | Découper  $\tau_i$  en  $\tau'_i$  et  $\tau''_i$  telle que  $T'_i = T''_i = T_i$ ,  $U(\tau'_i) = 1 - U[p]$ ,
16            |  $U(\tau''_i) = U(\tau_i) - U(\tau'_i)$ ;
17            |  $U[p] \leftarrow U[p] + U(\tau'_i)$  ;  $\tau[p] \leftarrow \tau[p] \cup \{\tau'_i\}$ ;
18            |  $U[p + 1] \leftarrow U[p] + U(\tau''_i)$  ;  $\tau[p + 1] \leftarrow \{\tau''_i\}$ ;
19          | Fin Si
20          |  $p \leftarrow p + 1$ ;
21        | Fin Si
22      | Fin Si
23    | Fin Si
24  | Fin Si
25 Fin Pour

```

Algorithme 9 : Algorithme (simplifié) d'affectation des tâches selon EKG. Attention, l'affectation peut échouer.

La figure 18 représente l'affectation de sept tâches sur une architecture à 5 processeurs et avec deux processeurs par groupe. Autrement dit, K vaut 2 et SEP vaut donc $2/3$. Dans cet exemple, seule la tâche τ_1 est classée comme lourde ($U(\tau_1) = 0.7$).

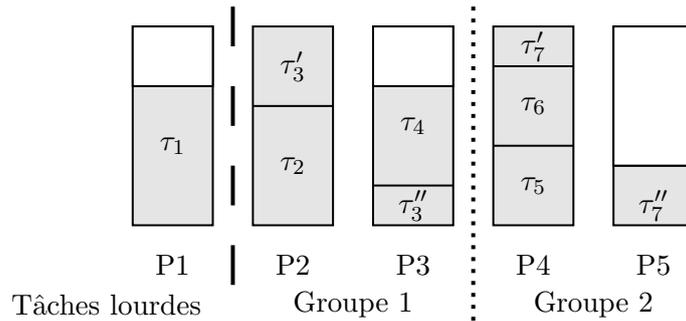


Figure 18 – Schéma représentatif de l'affectation de 7 tâches sur une architecture à 5 processeurs, et pour $K = 2$. $U(\tau_1) = 0.7, U(\tau_2) = 0.6, U(\tau_3) = 0.6, U(\tau_4) = 0.5, U(\tau_5) = 0.4, U(\tau_6) = 0.4, U(\tau_7) = 0.5$

Sur cet exemple, nous remarquons la présence de trous qui auraient pu servir à l'ordonancement de certaines tâches. Un trou se crée lorsque le taux d'utilisation d'une tâche lourde est comprise dans l'intervalle $[SEP, 1[$, ou lorsqu'une tâche légère ne peut pas être découpée. Nous pouvons remarquer que ces cas se produisent uniquement lorsque $K < m$. En effet, si $K = m$, alors $SEP = 1$ et donc l'intervalle $[SEP, 1[$ devient vide. Et si $K = m$,

alors il n'y a pas de trou et donc le second cas ne peut pas se produire non plus.

Ordonnancement des tâches Pour chaque groupe, on découpe le temps en intervalles EKG. Un intervalle EKG est défini par deux dates de réveil successives de tâche dans un même groupe. C'est donc similaire aux slots dans la terminologie DP-Fair, mais limité aux tâches d'un même groupe. Pour la suite des explications, l'intervalle EKG considéré débute en t_0 et se finit en t_1 .

De manière similaire à DP-Fair, les travaux des tâches migrantes doivent s'exécuter pendant un temps proportionnel à leur taux d'utilisation et à la durée de l'intervalle. Soit τ_i une tâche migrante dont une partie (τ_i') s'exécute sur un processeur et une autre (τ_i'') continue sur le processeur suivant. À t_0 , la tâche τ_i s'exécute pendant $U(\tau_i') \times (t_1 - t_0)$ unités de temps et se termine en $timea$. Vers la fin de l'intervalle, en $timeb$, l'exécution de la tâche τ_i reprend pour une durée de $U(\tau_i'') \times (t_1 - t_0)$ unités de temps et se termine en t_1 . Les autres tâches actives sont ordonnancées selon EDF sur l'intervalle $]timea, timeb[$. La tâche τ_i s'est exécutée en deux parties (τ_i' et τ_i'') et sur des processeurs différents mais sans chevauchement dans le temps car $U(\tau_i) \leq 1$ et donc $timea \leq timeb$. Un exemple est présenté par la figure 19.

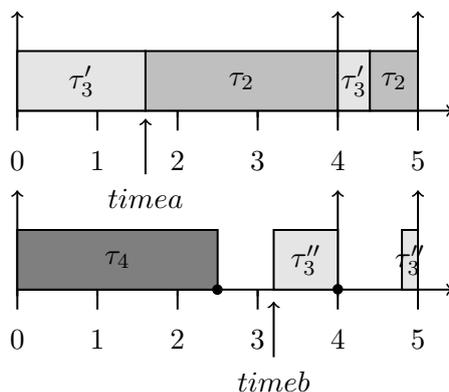


Figure 19 – Ordonnancement des tâches du groupe 1 avec EKG.

Réduction du nombre de préemptions par *mirroring* La technique dite de miroir (*mirroring*) peut être facilement mise en place en inversant simplement τ_i' et τ_i'' . Ceci divise par deux le nombre de préemptions. La figure 20 permet de comprendre visuellement l'intérêt de cette astuce. On peut constater au final, un nombre relativement faible de préemptions. À noter que cette technique est réutilisable pour d'autres politiques d'ordonnancement, c'est le cas par exemple de DP-WRAP [LEVIN et al. 2010].

3.6.2 Algorithme d'ordonnancement EDHS

Contrairement à l'algorithme EKG qui assigne les tâches migrantes et non-migrantes en même temps, l'algorithme EDHS [KATO et YAMASAKI 2008c] procède en deux étapes bien séparées : dans un premier temps, les tâches sont affectées selon un algorithme de partitionnement, puis dans un second temps, les tâches qui n'ont pas été affectées sont réparties sur plusieurs processeurs, suivant un second algorithme. Il est intéressant de noter que tout système de tâches ordonnancable par partitionnement, l'est aussi par EDHS.

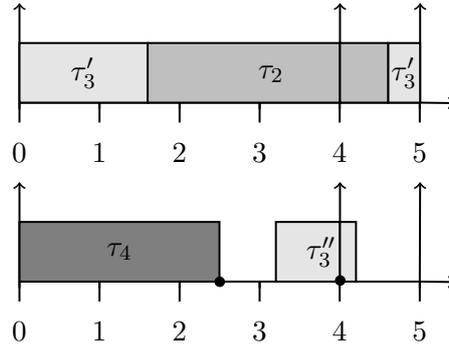


Figure 20 – Ordonnancement des tâches du groupe 1 avec EKG et la technique de miroir.

Pour simplifier, on suppose qu'il ne peut y avoir qu'une seule tâche migrante par processeur. Les tâches migrantes s'exécuteront sur plusieurs processeurs pendant une durée inférieure ou égale à la durée maximale qui conserve l'ordonnançabilité sur chaque processeur. L'ordonnancement utilisé sur les processeurs est un EDF modifié tel que les tâches migrantes sont prioritaires.

Calcul de la durée d'exécution des tâches migrantes. Après la répartition par partitionnement, nous obtenons un ensemble de tâches qui n'ont pas pu être affectées. Cet ensemble de tâche sera noté \mathbb{M} et correspond aux tâches migrantes. La figure 21 permet de comprendre visuellement comment est réparti le temps de calcul de chaque tâche sur différents processeurs. L'algorithme 10 permet de formaliser un peu plus comment peut être réalisé ce découpage. Pour aider à la compréhension, les notations utilisées pour EKG sont reprises et complétées :

- $U[p]$, la charge du processeur p .
- $\tau[p]$, l'ensemble des tâches affectées au processeur p .
- \mathbb{M} , l'ensemble des tâches migrantes.

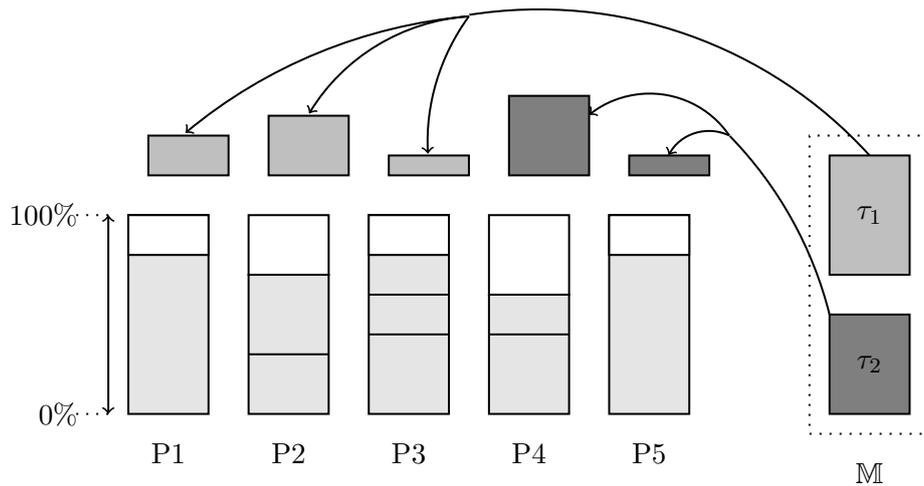


Figure 21 – Semi-partitionnement selon EDHS.

```

1 Pour tout  $\tau_i \in \mathbb{M}$  Faire
2    $p \leftarrow 1$ ;
3    $C_i^r \leftarrow C_i$ ;
4   Tant Que  $C_i^r > 0$  Faire
5      $r \leftarrow \min(\text{temps disponible conservant l'ordonnançabilité pour le processeur}$ 
6        $p, C_i^r)$ ;
7      $C_i^r \leftarrow C_i^r - r$ ;
8      $\tau[p] \leftarrow \tau[p] \cup \{\tau_i\}$ ;
9      $U[p] \leftarrow U[p] + r/T_i$ ;
10     $p \leftarrow p + 1$ ;
11    Si  $p > m$  Alors
12      | échec de l'algorithme;
13    Fin Si
14  Fin Tant Que
15 Fin Pour

```

Algorithme 10 : Algorithme (simplifié) de calcul du temps durant lequel une tâche doit s'exécuter sur différents processeurs.

Ordonnancement À l'exécution, les tâches non-migrantes s'exécutent selon EDF mais les tâches migrantes s'exécutent avec une priorité maximale et sans chevauchement dans le temps. Lorsqu'une tâche migrante a épuisé son temps d'exécution sur un processeur, elle continue son exécution immédiatement sur le processeur suivant en préemptant si nécessaire un travail en cours d'exécution. La figure 22 permet de mieux visualiser ce qui se passe.

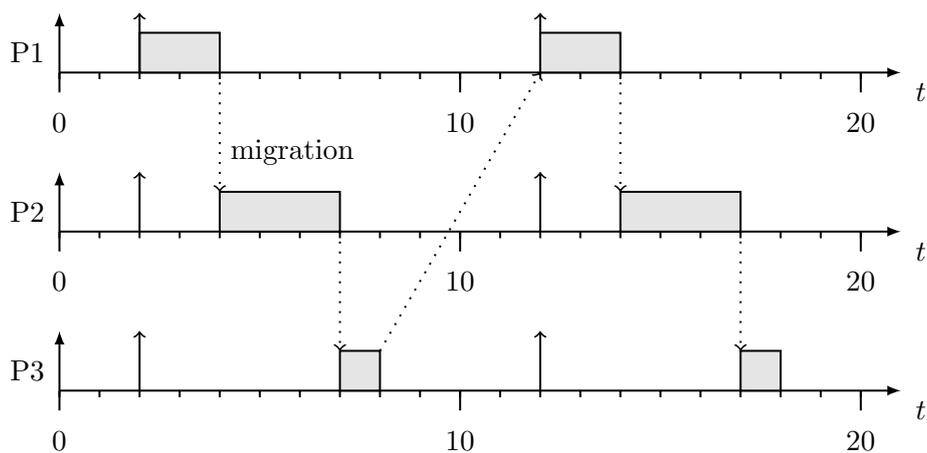


Figure 22 – Migration de la tâche $\tau_1(C_1 = 6, T_1 = 10)$.

La modification apportée à EDF engendre cependant des problèmes d'ordonnançabilité. En effet, comme on peut le voir sur le schéma 23, le fait de donner une forte priorité à la tâche migrante peut entraîner un dépassement d'échéance. L'algorithme EDF-WM tente

d'apporter un peu plus de flexibilité pour limiter ce problème mais sans pour autant le lever. Par simulation, les auteurs montrent que EDHS améliore la limite d'utilisation de 10 à 30% comparé à EDF en partitionné. Ce problème n'apparaît pas avec EKG qui utilise pourtant un algorithme EDF pour les tâches non-migrantes. En effet, EKG utilise des portions de travaux pour les tâches migrantes, c'est ceci qui permet de respecter les échéances.

3.6.3 Algorithme d'ordonnancement EDF-WM

EDF-WM (*EDF with Window-constraint Migration*) [KATO, YAMASAKI et ISHIKAWA 2009] est une amélioration de EDHS qui s'inspire des optimisations qui ont été possibles avec DP-PM. Si nous reprenons l'exemple de l'ordonnancement selon EDHS (Figure 22), nous pouvons constater que peu de flexibilité est laissée à la tâche migrante quant à ses instants de migration. Pourtant, comme le montrent les figures 23 et 24, il existe des cas où ce manque de flexibilité engendre non seulement des préemptions inutiles, mais empêche aussi l'ordonnancabilité du système.

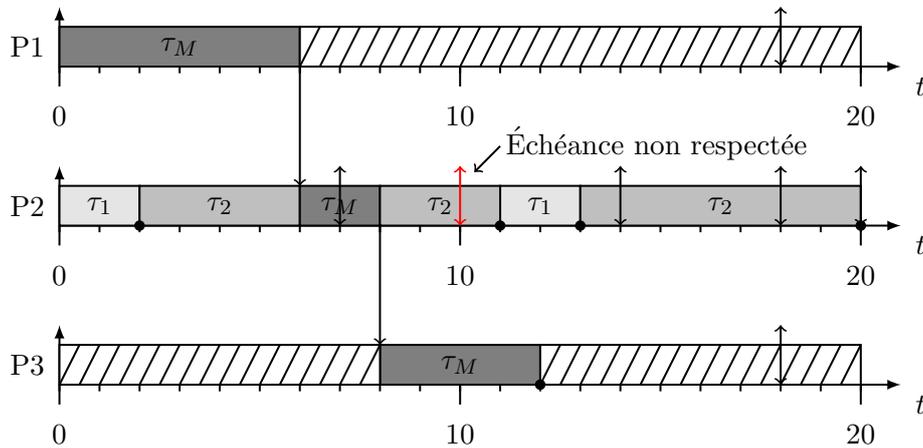


Figure 23 – Exemple où EDHS est en échec à cause d'un manque de flexibilité sur la date de préemption de la tâche τ_2 par τ_M . $\tau_M(C = 12, T = 18)$, $\tau_1(C = 2, T = 7)$, $\tau_2(C = 7, T = 10)$

L'idée principale de EDF-WM est de diviser chaque tâche migrante en sous-tâches avec une certaine fenêtre d'exécution possible. À la manière de PD², nous définissons pour chaque sous-tâche sa durée d'exécution, sa date de pseudo-réveil et sa date de pseudo-échéance. Une telle sous-tâche pourra alors être ordonnancée selon EDF, sans avoir à lui appliquer artificiellement une priorité maximale (Figure 24).

L'algorithme de semi-partitionnement vise à déterminer le nombre de sous-tâches, la durée d'exécution de chaque sous-tâche et leur fenêtre d'exécution. La détermination de la taille des fenêtres est la principale difficulté comparé à EDHS. On note :

- $\tau_{k,x}$, la sous-tâche de τ_k qui s'exécute sur le processeur P_x ,
- $w_{k,x}$, la longueur de la fenêtre d'exécution de la sous-tâche $\tau_{k,x}$,
- $C_{k,x}$, le temps d'exécution de la sous-tâche $\tau_{k,x}$,
- $T_{k,x} = T_k$, la période des sous-tâches,
- $D_{k,x} = w_{k,x}$, l'échéance relative de la sous-tâche $\tau_{k,x}$.

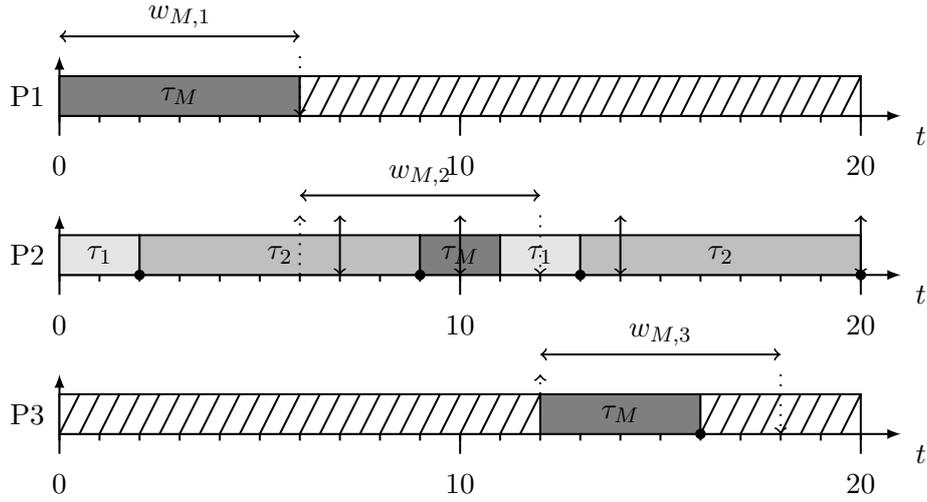


Figure 24 – Exemple qui montre comment en retardant la date de préemption, on obtient un système ordonnançable. $\tau_M(C = 12, T = 18)$, $\tau_1(C = 2, T = 7)$, $\tau_2(C = 7, T = 10)$

Demand Bound Function L’algorithme qui calcule la taille des fenêtres d’exécution repose sur ce qu’on appelle la *Demand Bound Function* et qui a été introduite dans [BARUAH, MOK et ROSIER 1990]. Cette fonction, que l’on note $dbf(\tau_i, L)$ et dont la formule est donnée par l’équation 3.15, calcule le nombre maximum de requêtes réveillées et dont la date d’échéance est dans l’intervalle $[0, L]$.

$$dbf(\tau_i, L) = \max\left(0, \left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1\right) \cdot C_i \quad (3.15)$$

Il a été montré dans [BARUAH, MOK et ROSIER 1990] que l’on peut tester la faisabilité d’un système de tâches sporadiques à échéances arbitraires en se basant sur cette fonction. Toutes les tâches respecteront leurs échéances en utilisant EDF sur un seul processeur, si et seulement si l’inéquation 3.16 est vérifiée pour toute valeur de L strictement positive. En fait, ceci peut se limiter aux valeurs de L qui correspondent aux échéances des travaux.

$$\sum_{\tau_i \in \tau} dbf(\tau_i, L) \leq L \quad (3.16)$$

Calcul de $C_{k,x}$ et $w_{k,x}$ La demande processeur d’une tâche τ_k sur le processeur P_x est donnée par l’équation 3.17.

$$dbf(\tau_{k,x}, L) = \max\left(0, \left\lfloor \frac{L - D_{k,x}}{T_k} \right\rfloor + 1\right) \cdot C_{k,x} \quad (3.17)$$

Nous nous intéressons maintenant à la façon dont nous pouvons déterminer les valeurs de $D_{k,x}$ et $C_{k,x}$. Si on note τ_x , l’ensemble des tâches déjà affectées au processeur P_x , alors $D_{k,x}$ et $C_{k,x}$ doivent respecter l’inéquation 3.18.

$$\sum_{\tau_i \in \tau_x} dbf(\tau_i, L) + dbf(\tau_{k,x}, L) \leq L \quad (3.18)$$

Le problème est que nous avons deux variables pour une seule équation. Nous devons donc fixer l'une ou l'autre de ces variables afin de pouvoir déterminer la valeur de l'autre. EDF-WM fixe $D_{k,x}$ à D_k/s , avec s , le nombre de processeurs sur lesquelles la tâche τ_k s'étend. Puis $C_{k,x}$ est déduit de l'inéquation 3.18 en cherchant sa valeur maximale.

Afin de déterminer la valeur de s , l'algorithme propose de prendre $s = 2$ puis d'incrémenter s jusqu'à ce que la somme des temps d'exécution des sous-tâches soit supérieure ou égale à la durée d'exécution de la tâche τ_k . Ainsi, le nombre de migrations est minimisé.

L'algorithme de semi partitionnement ne sera pas présenté ici. Un lecteur intéressé trouver le détail dans la publication originale [KATO, YAMASAKI et ISHIKAWA 2009].

Ordonnancement L'ordonnancement se fait selon l'algorithme EDF. L'avantage de la technique de EDF-WM est qu'il n'est plus nécessaire de fixer la priorité de la tâche migrante, chaque sous-tâche est ordonnancée comme les tâches classiques, en se basant sur ses dates d'activation, d'échéance et de périodicité.

3.6.4 Algorithme d'ordonnancement NPS-F

L'algorithme NPS-F, pour *Notional Processor Scheduling - Fractional capacity*, permet d'atteindre des taux d'utilisation entre 75% et 100% pour des systèmes de tâches sporadiques à échéances implicites [BLETSAS et ANDERSSON 2009b]. Il se base sur le concept de *Notional Processor Scheduling* introduit dans [BLETSAS et ANDERSSON 2009a]. Une variante de NPS-F destinée aux processeurs organisés sous la forme de clusters est présentée dans [BLETSAS et ANDERSSON 2009b]. Cette variante permet de limiter les migrations de tâches à des ensembles de processeurs partageant les mêmes caches. En pratique, cela pourrait être un avantage intéressant car un défaut de cache impacte fortement les performances du système [FEDOROVA et al. 2005 ; ANDERSON, CALANDRINO et DEVI 2006].

Avant de décrire l'algorithme NPS-F, il convient de présenter la notion de réserve périodique puis de *notional processor*.

Réserves périodiques (*Periodic reserves*) Une réserve périodique est une sorte de serveur permettant d'ordonnancer une ou plusieurs tâches sporadiques. Une telle réserve est une fenêtre de temps de largeur fixe et qui est disponible périodiquement toutes les S unités de temps. Le temps disponible à l'intérieur d'une réserve est destiné uniquement à l'exécution des tâches et à l'inverse, les tâches affectées à une réserve périodique ne peuvent pas être exécutées en dehors des limites de la réserve. L'algorithme d'ordonnancement des tâches à l'intérieur des réserves périodiques dans le cas de NPS-F, est EDF.

Soit T_{min} la plus petite durée entre deux arrivées consécutives d'une même tâche, parmi l'ensemble des tâches affectées à la réserve périodique considérée. Supposons alors que $S \leq \frac{T_{min}}{\delta}$, avec δ un entier strictement positif. L'ensemble des tâches, d'utilisation cumulée U , est ordonnançable par EDF si la réserve a pour longueur $in\!fl\!ate(U) \cdot S$ unités de temps. La fonction $in\!fl\!ate$ est définie par l'équation 3.19.

$$in\!fl\!ate(U) = \frac{(\delta + 1) \cdot U}{U + \delta} \tag{3.19}$$

La différence entre la capacité de processeur nécessaire à un ordonnancement correct des tâches et l’utilisation cumulée de ces dernières est appelée inflation. Cette valeur est donnée par l’équation 3.20.

$$\alpha(U) = \text{inflate}(U) - U = \frac{U \cdot (1 - U)}{U + \delta} \quad (3.20)$$

Plus la valeur de δ est grande et moins l’inflation sera grande.

Notional processor Normalement, une réserve est implantée sur un processeur en particulier. Cependant, si on prend plusieurs réserves temporellement adjacentes (sur différents processeurs) et traitant des mêmes tâches, alors elles se comportent comme une réserve “distribuée”. De telles réserves distribuées sont similaires en terme de capacité d’exécution à une unique réserve de longueur égale à la somme des longueurs des réserves constituant une réserve distribuée.

Tel que défini dans [BLETSAS et ANDERSSON 2009a], lorsqu’une réserve distribuée atteint 100% de la capacité d’un processeur, alors on parle de *notional processor*, sinon on parle de *fractional-capacity notional processor*.

Présentation de l’algorithme L’algorithme NPS-F procède en plusieurs étapes. Dans un premier temps, les n tâches à ordonnancer sont affectées à un nombre infini de “containers” de capacité 1 en utilisant l’heuristique de bin-packing *First-Fit*. À la fin de l’heuristique, nous obtenons un ensemble de m' containers utilisés ($1 \leq m' \leq n$). Soit U_p , l’utilisation cumulée des tâches assignées au $p^{\text{ème}}$ container.

Les tâches de chaque container seront ordonnancées par des *notional processors*. Ainsi, dans un second temps, l’utilisation cumulée des tâches de chaque container est augmentée par inflation (Équation 3.19) de manière à ce que les tâches soient ordonnancables selon EDF.

Enfin, les m' *notional processors* sont répartis sur les m processeurs physiques. Pour cela, deux approches sont proposées :

- La première solution consiste à placer un *notional processor* sur le processeur courant tant que le processeur n’est pas saturé et lorsqu’il n’y a plus de place, le *notional processor* est réparti sur le processeur courant et le processeur suivant. L’algorithme continue à partir du processeur suivant et ainsi de suite. Cette approche est donc similaire à EKG lors de la répartition des tâches légères mais appliquée au niveau des *notional processors*.
- La seconde solution consiste à affecter les m premiers *notional processors* sur les processeurs physiques et à faire migrer les autres sur plusieurs processeurs. Cette fois, cette approche peut être apparentée à EDHS.

La grande différence est que dans la première approche, un *notional processor* ne migre qu’entre deux processeurs au maximum alors que dans la seconde, un *notional processor* peut être réparti sur plus de deux processeurs.

Que la répartition soit faite avec l’une ou l’autre des approches, cela ne change en rien les résultats d’ordonnancabilité. La répartition est possible si et seulement si :

$$\sum_{p=1}^{m'} \text{inflate}(U_p) \leq m \quad (3.21)$$

La valeur de S est choisie telle que $S \leq \frac{T_{min}}{\delta}$ avec T_{min} la plus petite valeur pour l'ensemble des tâches du système.

Limite d'utilisation La limite d'utilisation de NPS-F est donnée par l'équation 3.22 et a été démontré dans [BLETSAS et ANDERSSON 2009b].

$$\frac{2 \cdot \delta + 1}{2 \cdot \delta + 2} \tag{3.22}$$

Ainsi, pour $\delta = 1$, NPS-F est capable d'utiliser 75% de la capacité totale des processeurs et lorsque $\delta \rightarrow \infty$, NPS-F tends vers une utilisation à 100% des ressources processeurs.

Pour rappel, le choix de S est inversement proportionnel à δ . En conséquence, lorsque δ est grand, la taille des intervalles de temps diminue et augmente donc le nombre de migrations.

Limite supérieure sur le nombre de préemptions BLETSAS et ANDERSSON prouvent aussi que le nombre de préemptions sur un intervalle de temps donné est borné (Équation 3.23).

$$N_{pr}(\Delta t) < N_{arr}(\Delta t) + \left\lceil \frac{\Delta t}{T_{min}} \right\rceil \cdot 3 \cdot m \cdot \delta \tag{3.23}$$

Avec N_{pr} , le nombre de préemptions et N_{arr} , le nombre de réveils de travail.

Conclusion

Ce rapport présente les principaux algorithmes d'ordonnancement temps réel monoprocesseur et multiprocesseur, en se basant sur le modèle de tâches indépendantes de LIU et LAYLAND. Les techniques monoprocesseurs sont connues depuis longtemps, ont fait l'objet de nombreux travaux et sont en partie utilisées par les industriels. Elles sont simples à mettre en œuvre et peuvent offrir une bonne utilisation du processeur.

Compte tenu des difficultés pour augmenter la capacité de calcul d'un processeur, les fabricants proposent des architectures dotées de plusieurs processeurs ce qui nécessite l'utilisation d'ordonnanceurs dédiés. L'utilisation des algorithmes d'ordonnancement monoprocesseurs temps réel ne donnant pas de bons résultats dans le pire cas lorsqu'ils sont appliqués sur plusieurs processeurs, de nombreux algorithmes dédiés aux architectures multiprocesseurs ont alors été proposés pour améliorer les performances.

Certains algorithmes globaux, en particulier ceux basés sur la notion de *fairness*, permettent d'atteindre l'optimalité, mais au prix d'un nombre important de préemptions et de migrations. Les algorithmes semi-partitionnés partent de ce constat et proposent un compromis entre le taux d'utilisation maximum du système et le nombre de préemptions. Mais ces algorithmes, qu'ils soient globaux ou semi-partitionnés, restent théoriques et ne tiennent généralement pas compte des surcoûts à l'exécution dûs aux migrations, aux préemptions et aux temps de calcul de l'algorithme. La suite de nos travaux viseront à déterminer et quantifier les sources de ces surcoûts afin de pouvoir les intégrer à nos outils de simulation, ceci afin d'avoir des résultats plus conformes à la réalité.

References

- ANDERSON, J., V. BUD et U. DEVI (juil. 2005). “An EDF-based scheduling algorithm for multiprocessor soft real-time systems”. Dans : *17th Euromicro Conference on Real-Time Systems, 2005. (ECRTS 2005)*. P. 199–208.
- ANDERSON, J., J. CALANDRINO et U. DEVI (avr. 2006). “Real-Time Scheduling on Multicore Platforms”. Dans : *12th IEEE Real-Time and Embedded Technology and Applications Symposium, 2006*. P. 179–190.
- ANDERSON, J. et A. SRINIVASAN (sept. 1999). *A New Look at Pfair Priorities*. Rap. tech. TR00-023, University of North Carolina at Chapel Hil.
- (2000a). “Early-Release Fair Scheduling”. Dans : *Euromicro Conference on Real-Time Systems*, p. 35.
- (2000b). “Pfair scheduling: beyond periodic task systems”. Dans : *Seventh International Conference on Real-Time Computing Systems and Applications, 2000*. P. 297–306.
- ANDERSSON, B., S. BARUAH et J. JONSSON (déc. 2001). “Static-priority scheduling on multiprocessors”. Dans : *22nd IEEE Real-Time Systems Symposium, 2001. (RTSS 2001)*, p. 193–202.
- ANDERSSON, B. et K. BLETSAS (juil. 2008). “Sporadic Multiprocessor Scheduling with Few Preemptions”. Dans : *Euromicro Conference on Real-Time Systems, 2008. ECRTS '08*. P. 243–252.
- ANDERSSON, B., K. BLETSAS et S. BARUAH (2008). “Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors”. Dans : *Real-Time Systems Symposium, 2008*, p. 385–394.
- ANDERSSON, B. et E. TOVAR (2006). “Multiprocessor Scheduling with Few Preemptions”. Dans : *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2006*. P. 322–334.
- AUDSLEY, N. et al. (1993). “Applying New Scheduling Theory to Static Priority Preemptive Scheduling”. Dans : *Software Engineering Journal* 8, p. 284–292.
- BARUAH, S. (2003). “Dynamic- and Static-priority Scheduling of Recurring Real-time Tasks”. Dans : *Real-Time Systems* 24 (1), p. 93–128.
- (juin 2004). “Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors”. Dans : *IEEE Transactions on Computers* 53.6, p. 781–784.
- BARUAH, S. et J. GOOSSENS (juil. 2003a). “Rate-monotonic scheduling on uniform multiprocessors”. Dans : *IEEE Transactions on Computers* 52.7, p. 966–970.
- (2003b). *Scheduling Real-time Tasks: Algorithms and Complexity*.
- BARUAH, S., A. MOK et L. ROSIER (1990). “Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor”. Dans : *11th Real-Time Systems Symposium*. IEEE Computer Society Press, p. 182–190.
- BARUAH, S. et al. (1996). “Proportionate progress: A notion of fairness in resource allocation”. Dans : *Algorithmica* 15 (6), p. 600–625.
- BARUAH, S. et al. (1999). “Generalized Multiframe Tasks”. Dans : *Real-Time Systems* 17, p. 5–22.
- BASTONI, A., B. BRANDENBURG et J. ANDERSON (2011). “Is Semi-Partitioned Scheduling Practical?” Dans : *23rd Euromicro Conference on Real-Time Systems (ECRTS), 2011*, p. 125–135.

- BERTOIGNA, M., M. CIRINEI et G. LIPARI (2005). “New Schedulability Tests for Real-Time Task Sets Scheduled by Deadline Monotonic on Multiprocessors.” Dans : *OPODIS'05*, p. 306–321.
- BINI, E. et G. BUTTAZZO (2001). “A hyperbolic bound for the rate monotonic algorithm”. Dans : *13th Euromicro Conference on Real-Time Systems, 2001*. P. 59–66.
- (2002). “The space of rate monotonic schedulability”. Dans : *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, p. 169–178.
- BLETAS, K. et B. ANDERSSON (2009a). “Notional Processors: An Approach for Multiprocessor Scheduling”. Dans : *Real-Time and Embedded Technology and Applications Symposium, IEEE 0*, p. 3–12.
- (déc. 2009b). “Preemption-Light Multiprocessor Scheduling of Sporadic Tasks with High Utilisation Bound”. Dans : *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, p. 447–456.
- (2011). “Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound”. Dans : *Real-Time Systems* 47 (4), p. 319–355.
- BUTTAZZO, G. (2005). “Rate Monotonic vs. EDF: Judgment Day”. Dans : *Real-Time Systems* 29 (1), p. 5–26.
- (2006). *Execution Time Analysis for Embedded Real-Time Systems*. Chapman & Hall/CRC - Taylor et Francis Group, p. 2.1–2.14. ISBN : 1-58488-678-1.
- CALANDRINO, J. et al. (déc. 2006). “LITMUS^{RT} : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers”. Dans : *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, p. 111–126.
- CHO, H. et B. RAVINDRAN (2006). “An optimal realtime scheduling algorithm for multiprocessors”. Dans : *In Proc. 27th IEEE International Real-Time Systems Symposium, Rio de Janeiro*, p. 101–110.
- COFFMAN Jr., E., M. GAREY et D. JOHNSON (1997). “Approximation algorithms for bin packing: a survey”. Dans : Boston, MA, USA : PWS Publishing Co., p. 46–93. ISBN : 0-534-94968-1.
- DETOUZOS, M. (1974). “Control Robotics: The Procedural Control of Physical Processes.” Dans : *IFIP Congress'74*, p. 807–813.
- DEVI, U. et J. ANDERSON (déc. 2005). “Tardiness bounds under global EDF scheduling on a multiprocessor”. Dans : *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, p. 12.
- DEVILLERS, R. et J. GOOSSENS (mar. 2000). “Liu and Layland’s schedulability test revisited”. Dans : *Inf. Process. Lett.* 73 (5–6), p. 157–161.
- DHALL, S. et C. LIU (1978). “On a Real-Time Scheduling Problem”. Dans : *Operations Research* 26.1, pp. 127–140.
- FEDOROVA, A. et al. (2005). “Performance of multithreaded chip multiprocessors and implications for operating system design”. Dans : *Annual conference on USENIX Annual Technical Conference. ATEC '05*. Anaheim, CA : USENIX Association, p. 26–26.
- GOOSSENS, J. (2006). *Systèmes temps réel — Ordonnancement réseaux et qualité de service*. Sous la dir. de N. NAVET. T. 2. Chapitre 2: Ordonnancement temps réel multiprocesseur (24 pages). ISBN 2-7462-1304-4. Hermès.
- GOOSSENS, J. et R. DEVILLERS (1997). “The Non-Optimality of the Monotonic Priority Assignments for Hard Real-Time Offset Free Systems”. Dans :
- GOOSSENS, J., S. FUNK et S. BARUAH (2002). “EDF scheduling on multiprocessor platforms: some (perhaps) counterintuitive observations”. Dans : *In RealTime Computing Systems and Applications Symposium*.

- GROLLEAU, E. (2011). “Ordonnancement et ordonnancabilité monoprocésseur”. Dans : *Ecole d’été temps réel, ETR’11*.
- GUAN, N. et al. (avr. 2010). “Fixed-Priority Multiprocessor Scheduling with Liu and Layland’s Utilization Bound”. Dans : *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, p. 165–174.
- HOLMAN, P. et J. ANDERSON (2005). “Adapting Pfair scheduling for symmetric multiprocessors”. Dans : *J. Embedded Comput.* 1 (4), p. 543–564. ISSN : 1740-4460.
- HONG, K. et J. LEUNG (déc. 1988). “On-line scheduling of real-time tasks”. Dans : *Real-Time Systems Symposium, 1988*, p. 244–250.
- KATO, S. et N. YAMASAKI (août 2007). “Real-Time Scheduling with Task Splitting on Multiprocessors”. Dans : *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007*. P. 441–450.
- (2008a). “Portioned EDF-based scheduling on multiprocessors”. Dans : *8th ACM international conference on Embedded software. EMSOFT ’08*. Atlanta, GA, USA : ACM, p. 139–148. ISBN : 978-1-60558-468-3.
- (avr. 2008b). “Portioned static-priority scheduling on multiprocessors”. Dans : *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008*. P. 1–12.
- (2008c). *Semi-Partitioning Technique for Multiprocessor Real-Time Scheduling*.
- (avr. 2009). “Semi-partitioned Fixed-Priority Scheduling on Multiprocessors”. Dans : *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, p. 23–32.
- KATO, S., N. YAMASAKI et Y. ISHIKAWA (2009). “Semi-partitioned Scheduling of Sporadic Task Systems on Multiprocessors”. Dans : *21st Euromicro Conference on Real-Time Systems*. IEEE Computer Society, p. 249–258. ISBN : 978-0-7695-3724-5.
- KORF, R. (2002). “A new algorithm for optimal bin packing”. Dans : *Eighteenth national conference on Artificial intelligence*. American Association for Artificial Intelligence, p. 731–736. ISBN : 0-262-51129-0.
- (2003). “An improved algorithm for optimal bin packing”. Dans : *18th international joint conference on Artificial intelligence*. Acapulco, Mexico : Morgan Kaufmann Publishers Inc., p. 1252–1258.
- LAKSHMANAN, K., R. RAJKUMAR et J. LEHOCZKY (2009). “Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors”. Dans : *21st Euromicro Conference on Real-Time Systems*. Washington, DC, USA : IEEE Computer Society, p. 239–248. ISBN : 978-0-7695-3724-5.
- LEE, C. C. et D. T. LEE (1985). “A simple on-line bin-packing algorithm”. Dans : *J. ACM* 32 (3), p. 562–572.
- LEHOCZKY, J., L. SHA et Y. DING (déc. 1989). “The rate monotonic scheduling algorithm: exact characterization and average case behavior”. Dans : *Real Time Systems Symposium, 1989*. P. 166–171.
- LEUNG, J. et J. WHITEHEAD (1982). “On the complexity of fixed-priority scheduling of periodic, real-time tasks”. Dans : *Performance Evaluation* 2.4, p. 237–250.
- LEVIN, G. et al. (juil. 2010). “DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling”. Dans : *22nd Euromicro Conference on Real-Time Systems (ECRTS), 2010*, p. 3–13.
- LIU, C. L. et J. LAYLAND (jan. 1973). “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. Dans : *J. ACM* 20 (1), p. 46–61.
- MCNAUGHTON, R. (1959). “Scheduling with Deadlines and Loss Functions”. Dans : *Management Science* 6.1, p. 1–12.

- MOK, A. (1983). *Fundamental design problems of distributed systems for the hard-real-time environment*. Rap. tech. Cambridge, MA, USA.
- MOK, A. et D. CHEN (déc. 1997). “A multiframe model for real-time tasks”. Dans : *Real-Time Systems Symposium, 1996., 17th IEEE*, p. 22–29.
- OH, D.-I. et T. BAKKER (1998). “Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment”. Dans : *Real-Time Systems* 15 (2), p. 183–192.
- OH, S.-H. et S.-M. YANG (1998). “A Modified Least-Laxity-First scheduling algorithm for real-time tasks”. Dans : *Fifth International Conference on Real-Time Computing Systems and Applications, 1998*. P. 31–36.
- PARK, M. et al. (mar. 2005). “Comparison of Deadline-Based Scheduling Algorithms for Periodic Real-Time Tasks on Multiprocessor”. Dans : *IEICE transactions on information and systems* 88.3, p. 658–661.
- PHILLIPS, C. et al. (1997). “Optimal time-critical scheduling via resource augmentation (extended abstract)”. Dans : *Twenty-ninth annual ACM symposium on Theory of computing*. STOC '97. El Paso, Texas, United States : ACM, p. 140–149. ISBN : 0-89791-888-6.
- SHAW, P. (2004). “A Constraint for Bin Packing”. Dans : *Principles and Practice of Constraint Programming – CP 2004*. Sous la dir. de Mark WALLACE. T. 3258. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, p. 648–662. ISBN : 978-3-540-23241-4.
- SRINIVASAN, A. et S. BARUAH (oct. 2002). “Deadline-based scheduling of periodic task systems on multiprocessors”. Dans : *Inf. Process. Lett.* 84 (2), p. 93–98.
- STIGGE, M. et al. (avr. 2011). “The Digraph Real-Time Task Model”. Dans : *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, p. 71–80.
- TINDELL, K. (1994). *Adding Time-Offsets to Schedulability Analysis*. Rap. tech.
- ZHU, D., D. MOSSE et R. MELHEM (2003). “Multiple-resource periodic scheduling problem: how much fairness is necessary?”. Dans : *24th IEEE Real-Time Systems Symposium, 2003. RTSS 2003*. P. 142–151.