Un premier pas vers l'étude de la cyclicité en environnement multiprocesseurs

Annie Choquet-Geniet
LISI - ENSMA Téléport 2 - 1 Avenue Clément Ader
BP 40109 86960 FUTUROSCOPE Cedex
ageniet@ensma.fr

Résumé

Nous nous plaçons dans un contexte bi processeurs pour des applications temps réel périodiques à contraintes strictes, constituées de tâches indépendantes. Nous considérons le problème de la cyclicité, c'est à dire de l'existence et des caractéristiques du fonctionnement en régime permanent. Nous montrons que les résultats obtenus dans le contexte monoprocesseur ne peuvent pas s'étendre dans le cas général au cas multiprocesseurs. Puis nous considérons les seuls algorithmes à priorités fixes, et nous caractérisons pour ces algorithmes l'entrée en régime permanent.

Plan

Mots clés

Ordonnancement multiprocesseurs - Régime permanent - Cycle - Temps creux acyclique - Priorités fixes

1 Introduction

1.1 Problématique

La généralisation de l'utilisation d'architectures multiprocesseurs a donné lieu, ces dernières années, à de nombreuses recherches en ordonnancement multiprocesseurs. Bien maitrisée dans le contexte monoprocesseur, la problématique de l'ordonnancement devient complexe en environnement multiprocesseurs, et de nombreuses questions subsistent encore. L'un des problèmes non encore résolu est celui de la cyclicité des ordonnancements. Il s'agit d'étudier temporellement le comportement cyclique des séquences, et plus particulièrement de caractériser l'instant de démarrage du régime permanent, après la phase de montée en charge. Nous avons déjà résolu ce problème en environnement monoprocesseur [4]. L'étude reposait sur une analyse de l'activité du processeur, caractérisée par l'existence de charge en attente d'exécution. Dans le cas multiprocesseur, l'existence d'une telle charge ne suffit plus à caractériser l'activité des processeurs. Ceci est dû à la non parallélisation des tâches. La démarche monoprocesseur ne peut donc pas être étendue au cas multiprocesseurs. L'objectif de ce travail est de réaliser l'étude dans le contexte des applications constituées de tâches périodiques indépendantes, pour les ordonnancements à priorités fixes. Ce premier travail sur le thème de la cyclicité, effectué dans un contexte relativement simple, a pour vocation de nous permettre de mieux mettre en évidence les problèmes et les mécanismes mis en jeu dans le cas multiprocesseurs.

1.2 Ordonnancement multiprocesseurs

L'ordonnancement des applications temps réel sur une machine multiprocesseurs pose de nombreux problèmes : en premier lieu, [5] ont montré qu'il ne peut pas exister d'algorithmes d'ordonnancement en ligne **optimal** [1] dans le cas général. Notons toutefois qu'il existe un algorithme optimal polynomial, ainsi qu'une CNS d'ordonnançabilité dans le cas de tâches indépendantes **à départs simultanés** (toutes les dates de première activation sont égales) et à **échéance sur requête** (le délai critique est égal à la période) [2, 1]. [10] ont montré que le problème de l'ordonnancement en environnement multiprocesseurs est NP- complet (voir [15] pour un panorama complet de la complexité de ce problème).
Par ailleurs, l'ordonnancement en ligne sur des architectures multiprocesseurs se heurte au problème de la **non stabilité** des algorithmes, même lorsque l'on ne considère que des tâches indépendantes : une durée d'exécution plus courte que prévue peut provoquer une faute temporelle [12, 15]. Une solution alternative consiste à se tourner vers des stratégies d'ordonnancement hors ligne : ces stratégies sont clairvoyantes [2] ce qui les rend plus performantes que les stratégies d'ordonnancement en ligne qui fondent leurs décisions sur la seule connaissance instantanée du sytème. Mais en contrepartie, la mise en œuvre des stratégies hors ligne est nettement plus coûteuse, notamment parce qu'il s'agit souvent de stratégies d'exploration exhaustive de l'espace des solutions. La majeure partie des stratégies présentes dans la littérature traitent d'applications non périodiques. Elles ordonnancent un ensemble fini de tâches, en utilisant éventuellement des heuristiques permettant de limiter le coût de la méthode.

[1] un ordonnancement est optimal s'il est capable d'ordonnancer correctement n'importe quelle application pour laquelle il existe au moins un ordonnancement respectant toutes les contraintes temporelles.

[2] les dates d'arrivée des instances de toutes les tâches sont connues a priori

Certaines d'entre elles ont également pris en compte des tâches non indépendantes. [14] ont considéré le partage de ressources, et proposé des algorithmes basés sur une exploration arborescente de l'espace des solutions, couplée avec l'utilisation d'heuristiques permettant de limiter le nombre de branches explorées. [6] ont pris en compte le problème de la précédence dans le cas d'une architecture biprocesseurs, ils opèrent par réajustements successifs de certains des paramètres temporels. [13] ont étudié le problème de l'ordonnancement en présence de contraintes de précédence, pour un nombre quelconque de processeurs, mais en l'absence d'échéances. Enfin, [16] présente une stratégie permettant de prendre en compte à la fois le partage de ressources et les contraintes de précédence. Le principe consiste à utiliser une exploration arborescente, couplée avec des étapes de découpages des tâches, de modifications des paramètres et d'adjonction de contraintes complémentaires (permettant la gestion des sections critiques).

Il existe beaucoup moins de résultats en ce qui concerne les applications périodiques : [2, 1] ont étudié les systèmes à départs simultanés et à échéance sur requête, et [3] ont étudié des sytèmes de tâches périodiques, à échéance sur requête, **à départs différés** (les dates de première activation ne sont pas toutes égales), lorsque les processeurs possèdent chacun leur propre vitesse (vérifiant des conditions arithmétiques).

1.3 Contribution

L'étude systématique du problème de la cyclicité dans le cadre multiprocesseurs permettrait de pallier le manque de stratégies d'ordonnancement pour les applications périodiques à départs différés. En effet, si l'existence d'un cycle est établie, et si on sait caractériser la date t d'entrée dans le cycle, l'application des stratégies développées pour les tâches non périodiques à l'ensemble des instances démarrant avant la date t + P permettra d'obtenir un ordonnancement fonctionnant en régime permanent. Sans ce résultat, par contre, il sera impossible de limiter a priori la taille de la séquence à construire. Par suite, les stratégies hors ligne ne pourront pas fonctionner elles non plus.

Nous nous plaçons ici dans le contexte des ordonnancements à priorités fixes (un intérêt majeur de ce type d'ordonnancement est sa disponibilité sur tous les noyaux). Nous considérons des tâches indépendantes, s'exécutant sur deux processeurs. Notre objectif est de caractériser le début du cycle d'un ordonnancement. Dans la partie 2, nous présentons nos hypothèses et nos notations. Dans la partie 3, nous introduisons la notion de temps creux acycliques partiels et complets, et nous montrons que les résultats clés de l'étude monoprocesseur ne sont pas valides dans le contexte multiprocesseurs. Enfin, dans la partie 4, nous montrons que, dans notre contexte, le cycle correspond à la première séquence de taille égale au PPCM des périodes et ne comportant pas de temps creux acyclique. Ce résultat n'est pas vrai dans le cas général.

2 Contexte de l'étude

2.1 Architecture

Nous considérons le modèle PRAM [7] : tous les processeurs sont identiques, chacun possède une mémoire qui lui est propre, mais une mémoire commune est accessible par tous les processeurs en temps constant. Nous supposons que le système

comporte 2 processeurs. Nous nous plaçons sous l'hypothèse de **migration totale** : une tâche n'est jamais définitivement affectée à un processeur, elle peut à tout instant migrer d'un processeur à un autre. Nous retenons cette hypothèse, car c'est celle qui offre la plus grande puissance d'ordonnançabilité [9].

2.2 Applications temps réel

Nous considérons des applications constituées de n **tâches périodiques indépendantes et non parallélisables**.
Chaque tâche τ_i est caractérisée par les 4 paramètres temporels classiques [11] : sa **date de réveil** r_i, sa **pire duré d'exécution** C_i, son **délai critique** R_i et sa **période** P_i. Nous supposons de plus que le **taux d'utilisation**[3] du système est égal à 2. Nous notons P la **métapériode** du système définie par P = PPCM($P_1, \ldots, P_n$).
Dans la suite, pour tout instant donné t, et pour toute tâche τ_i, nous notons (voir figure 1) :

1. $IC_i(t)$ l'instance courante de la tâche τ_i à l'instant t.
2. $CR_i(t)$ la charge restant à exécuter à l'instant t pour l'instance $IC_i(t)$ de la tâche τ_i.
3. $\overline{CR}_i(t)$ la charge déjà effectuée à l'instant t par l'instance $IC_i(t)$ de la tâche τ_i.

Il résulte de ces définitions que $CR_i(t) + \overline{CR}_i(t) = C_i$
Nous notons de plus W(t, t') le cumul de la charge exécutée entre les instants t et

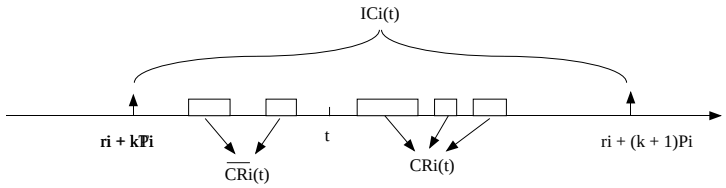

FIG. 1 – Instance courante, charge restante et charge déjà effectuée.

t'. S'il n'y a aucun temps d'inactivité des processeurs entre deux instants t et t', nous avons W(t, t') = 2 * (t' - t). Enfin, nous notons W_k(t, t') la charge exécutée par la tâche τ_k entre les instants t et t'.

3 Temps creux

Comme nous l'avons fait dans le cas monoprocesseur, nous examinons tout d'abord l'activité des processeurs.

[3] Le taux d'utilisation est défini par $U = \sum_{i=1}^{n} \frac{C_i}{P_i}$

3.1 Temps creux acycliques

Lorsque la charge du système est inférieure à 2, les processeurs sont inactifs pendant P(2 - U) unités de temps toutes les métapériodes. Ces temps d'inactivité sont appelés temps creux **cycliques**. Ils ont lieu toutes les métapériodes, et n'apparaissent pas si le taux d'utilisation est égal à 2. L'exemple suivant met en évidence l'existence d'un autre type de temps creux.
Considérons un système constitué de 5 tâches :
S1 : {τ_1(0,1, 3, 3), τ_2(0, 1, 3, 3), τ_3(0, 4, 9,9),τ_4(0, 2, 3, 3), τ_5(8, 2, 9, 9)}. Supposons de plus que Prio(τ_1) $\geq$Prio(τ_2) $\geq$ Prio(τ_3)$\geq$ Prio(τ_4)$\geq$ Prio(τ_5). Nous obtenons l'exécution suivante (le placement à chaque instant est arbitraire, car sans conséquence sur le problème traité) : (figure 2)

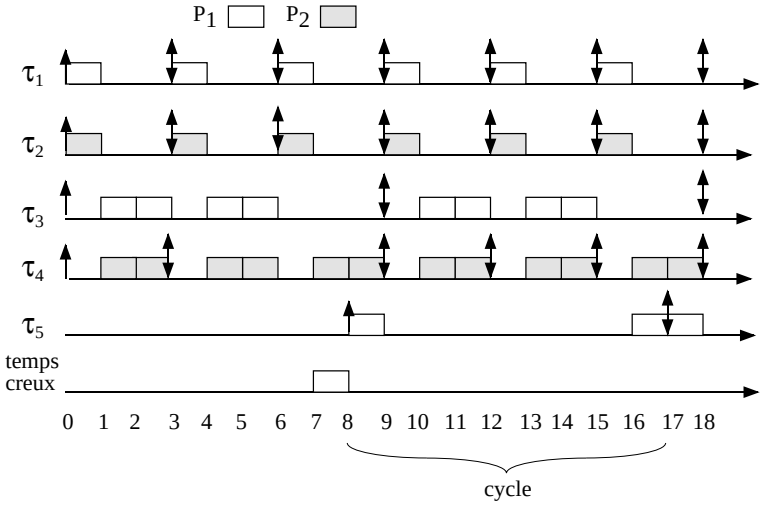

FIG. 2 – Ordonnancement du système S1. Il y a un temps creux à l'instant 7 sur le processeur 1.

La charge processeur pour le système S1 est égale à 2. Cela signifie qu'il n'y a pas de temps creux cyclique, mais on constate cependant la présence d'un temps creux à l'instant 7. On constate ensuite que le système est dans le même état[4] aux instant 8 et 17 = 8 + P. La séquence définie sur la fenêtre temporelle [7, 17] sera donc itérée, elle définit la partie cyclique de l'ordonnancement. Le temps creux détecté à l'instant 7 ne se reproduira pas. C'est un temps creux dit **acyclique**.

Dans le contexte monoprocesseur, nous avions montré que le nombre et la position des temps creux acycliques étaient caractéristiques de l'application, indépendemment de la stratégie d'ordonnancement choisie, dès lors que celle-ci était déterministe[5]

[4] le système est dans le même état à deux instants différents ssi chacune des tâches est dans le même état d'avancement et à même distance de son échéance. Si la distance entre les deux instants est égale à P, le second point est automatiquement acquis.

[5] une stratégie d'ordonnancement est dite déterministe ssi dans un même contexte : mêmes tâches prêtes, au même état d'avancement, et à même distance de leur échéance, les décisions d'ordonnancement sont les mêmes.

et conservative[6] Cette constatation avait grandement facilité l'étude de la cyclicité. Il n'en est malheureusement pas de même dans le cas multiprocesseurs, comme le montre l'ordonnancement valide suivant (qui n'est pas à priorités fixes), toujours du système S1 (figure 3).

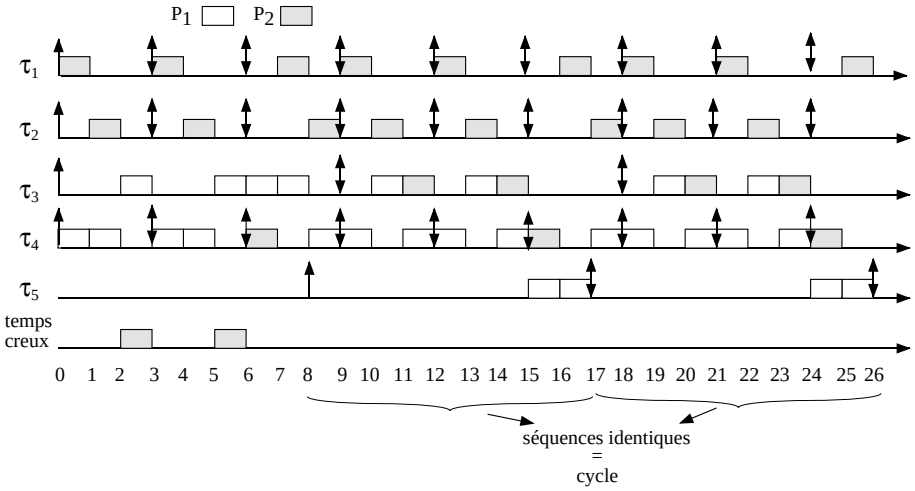

FIG. 3 – Un autre ordonnancement valide du système S1. Il y a deux temps creux, le dernier temps creux est à l'instant 5, et le cycle démarre à l'instant 8.

Un autre point avait été mis en évidence : le cycle démarrait toujours directement après le dernier temps creux acyclique. Cette propriété n'est plus systématique dans le cas multiprocesseurs, comme l'illustre également la figure 3. Le dernier temps creux a lieu à l'instant 5, mais le cycle ne démarre qu'à l'instant 8. On peut remarquer pour preuve qu'entre les instants 6 et 15, la tâche τ_5 n'est jamais ordonnancée. Notons enfin que malgré tout la stratégie d'ordonnancement présentée figure 3 est bien déterministe et conservative. Cela montre donc qu'on ne peut pas étendre les résultats obtenus en monoprocesseur.

Ces exemples négatifs, nous ont incités à restreindre le champ de notre étude. Nous ne considérons plus désormais que des stratégies d'ordonnancement à priorités fixes. Nous supposons en outre que deux tâches distinctes ont des priorités distinctes.

3.2 Temps creux partiels et complets

La charge étant égale à 2, les seuls temps creux à considérer sont acycliques. La différence majeure avec le cas monoprocesseur est qu'il peut y avoir deux sortes de temps creux (voir figure 4) :

1. **les temps creux complets** : les deux processeurs sont simultanément inactifs.
2. **les temps creux partiels** : un seul des processeurs est inactif.

Les temps creux complets sont caractérisés par l'absence de charge en attente d'exécution, mais ce n'est pas le cas pour les temps creux partiels.

[6] une stratégie est conservative si le processeur n'est jamais oisif lorsqu'il y a des tâches prêtes

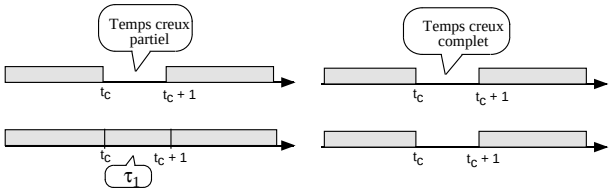

FIG. 4 – Temps creux partiels et complets

4 Étude des propriétés de cyclicité dans le cas d'ordonnancement à priorités fixes

Notre objectif est de prouver que l'on peut, dans ce contexte, caractériser l'entrée en régime permanent à l'aide du dernier temps creux acyclique. Le cycle correspond alors à la première séquence de taille P sans temps creux trouvée.

Theorème 1. *Dans un ordonnancement à priorités fixes, si on trouve un temps creux à une date t_c suivi de P unités de temps sans temps creux, alors la séquence produite sur la fenêtre temporelle [t_c, t_c + P] définit le fonctionnement en régime permanent de l'application.*

4.1 Cas d'un temps creux complet

On partitionne l'ensemble des tâches en trois sous-ensembles :
• RA = ensemble des tâches (ré)activées à l'instant t_c + 1.
• RT = ensemble des tâches dont la première activation a lieu après t_c + 1.
• DR = ensemble des tâches déjà réveillées (i.e. dont la première activation est avant t_c), mais non réactivées à t_c + 1.

4.1.1 Calcul de la charge exécutée entre t_c + 1 et t_c + P + 1

La preuve que nous proposons repose sur l'évaluation de la charge exécutée entre t_c + 1 et t_c + P + 1. Cette charge est égale à 2P, puisque nous avons supposé qu'il n'y avait aucun temps creux entre t_c + 1 et t_c + P + 1. Nous calculons alors la contribution de chacune des tâches à la charge totale.

1. *Charge issue de RA* : chaque tâche τ_i de RA s'exécute exactement $\frac{P}{P_i}$ fois, donc ces tâches engendrent une charge égale à $\sum_{\tau_i \in RA} \frac{P}{P_i} * C_i$.
2. *Charge issue de RT* : chaque tâche τ_i de RT exécute $\lfloor \frac{t_c+P+1-r_i}{P_i} \rfloor$ instances complètes, et en démarre une dernière. Donc la charge cumulée est égale à : $\sum_{\tau_i \in RT} \lfloor \frac{t_c+P+1-r_i}{P_i} \rfloor * C_i + \overline{CR}_i(t_c + P + 1)$.
3. *Charge issue de DR* : chaque tâche τ_i de DR exécute $\frac{P}{P_i} - 1$ instances complètes, et en démarre une dernière. De plus, à l'instant t_c + 1, il ne reste aucune charge à exécuter à la tâche τ_i, puisqu'il y a un temps creux complet à l'instant t_c, et t_c

+ 1 n'est pas un instant de réactivation de la tâche. La charge cumulée est donc : $\sum_{\tau_i \in DR}(\frac{P}{P_i} - 1) * C_i + \overline{CR}_i(t_c + P + 1)$.

Nous avons donc

$$\begin{aligned} W(t_c+1, t_c+P+1) &= \sum_{\tau_i \in RA} \frac{P}{P_i} * C_i \\ &+ \sum_{\tau_i \in RT} \lfloor \frac{t_c + P + 1 - r_i}{P_i} \rfloor * C_i + \overline{CR}_i(t_c + P + 1) \\ &+ \sum_{\tau_i \in DR} (\frac{P}{P_i} - 1) * C_i + \overline{CR}_i(t_c + P + 1) \\ &= 2P \end{aligned}$$

4.1.2 Réveils tardifs

Afin d'évaluer précisément la charge exécutée par les tâches à réveil tardifs (i.e. postérieurs à t_c + 1), nous devons étudier les dates de première activation des tâches de RT :

Lemme 2. *$\forall \tau_i \in RT$, nous avons $t_c + 1 < r_i < t_c + P_i + 1$*

Preuve
a - Supposons qu'il existe i_0 tel que $r_{i_0} > t_c + P_{i_0} + 1$. Alors on a $\lfloor \frac{t_c+P+1-r_{i_0}}{P_{i_0}} \rfloor < \frac{P}{P_{i_0}} - 1$. Par ailleurs, nous avons $\overline{CR}_i(t_c+P+1) \leq C_i$ pour toute tâche, et $\lfloor \frac{t_c+P+1-r_i}{P_i} \rfloor \leq \frac{P}{P_i} - 1$ pour toute tâche de RT autre que τ_{i_0}. Donc :

$$\begin{aligned} 2P &= W(t_c+1, t_c+P+1) \\ &\leq \sum_{\tau_i \in RA \cup DR \cup (RT \setminus \{\tau_{i_0}\})} \frac{P}{P_i} * C_i + \lfloor \frac{t_c + P + 1 - r_{i_0}}{P_{i_0}} \rfloor * C_{i_0} + \overline{CR}_{i_0}(t_c + P + 1) \\ &< \sum_{i=1}^{n} \frac{P}{P_i} * C_i - C_i \\ &= 2P - C_i \end{aligned}$$

D'où la contradiction.
b - Supposons qu'il existe i_0 tel que $r_{i_0} = t_c + P_{i_0} + 1$. Cette fois, nous avons $\lfloor \frac{t_c+P+1-r_{i_0}}{P_i} \rfloor = \frac{P}{P_{i_0}} - 1$ et $\overline{CR}_{i_0}(t_c + P + 1) = 0$. On aboutit donc à la même incohérence que précédemment. □

4.1.3 Preuve du théorème

Nous pouvons maintenant déterminer la contribution de chaque tâche à la charge exécutée, et donc achever la preuve du théorème.
D'après le lemme 2, nous avons pour toute tâche τ_i de RT $\lfloor \frac{t_c+P+1-r_i}{P_i} \rfloor = \frac{P}{P_i} - 1$.

Par suite, nous avons

$$\begin{aligned} 2P &= W(t_c+1, t_c+P+1) \\ &= \sum_{i=1}^{n} \frac{P}{P_i} * C_i + \sum_{\tau_i \in RT \cup DR} (\overline{CR}_i(t_c+P+1) - C_i) \\ &= 2P + \sum_{\tau_i \in RT \cup DR} (\overline{CR}_i(t_c+P+1) - C_i) \end{aligned}$$

Or $\overline{CR}_i(t_c+P+1) - C_i \leq 0$. On en déduit donc que $\overline{CR}_i(t_c+P+1) - C_i = 0$ pour toute tâche τ_i de RT $\cup$ DR. Donc il n'y a plus de tâches actives à l'instant t_c + P + 1, et chaque tâche sera réactivée à même distance de t_c + P + 1 que de t_c + 1. Le système est donc dans le même état en t_c + 1 et en t_c + P + 1. Donc la séquence obtenue entre t_c + 1 et t_c + P + 1 se répétera à l'infini. □

4.2 Cas d'un temps creux partiel

Sans perte de généralité, nous pouvons supposer que c'est la tâche τ_1 qui s'exécute à l'instant t_c. L'ensemble des tâches est décomposé en τ_1, RA (le cas échéant privé de τ_1), RT et DR (le cas échéant privé de τ_1). La contribution de τ_1 à la charge est : $[\frac{P}{P_1} - 1] * C_1 + [CR_1(t_c+1) + \overline{CR}_1(t_c+P+1)]$: les instances complètes, plus ce qui reste à traiter l'instance courante à l'instant $t_c + 1$ plus ce qui a déjà été exécuté à l'instant $t_c + P + 1$. Par suite, le cumul de charge effectuée entre t_c + 1 et t_c + P + 1 est égal à :

$$\begin{aligned} W(t_c+1, t_c+P+1) &= \sum_{\tau_i \in RA} \frac{P}{P_i} * C_i \\ &+ \frac{P}{P_1} * C_1 + [CR_1(t_c+1) + \overline{CR}(T_c+P+1) - C_1] \\ &+ \sum_{\tau_i \in RT} \lfloor \frac{t_c+P+1-r_i}{P_i} \rfloor * C_i + \overline{CR}_i(t_c+P+1) \\ &+ \sum_{\tau_i \in DR} [(\frac{P}{P_i} - 1) * C_i + \overline{CR}_i(t_c+P+1)] \\ &= 2P \end{aligned}$$

La preuve du théorème s'appuie fortement sur le comportement de τ_1. Pour adapter au cas du temps creux partiel les raisonnements mis en œuvre dans le cas du temps creux complet, il nous faut tout d'abord établir que τ_1 ne peut pas avoir effectué plus de charge en t_c + P + 1 qu'en t_c + 1 :

Lemme 3. *Nous avons* $CR_1(t_c+1) + \overline{CR}_i(t_c+P+1) - C_1 \leq 0$.

4.2.1 Preuve du théorème

Supposons le lemme 3 acquis. La preuve du thórème est alors très proche de celle effectuée dans le cas du temps creux complet. La première étape consiste à montrer

que le lemme 2 s'applique aussi dans ce contexte. Le principe de la preuve est exactement le même que dans le cas précédent. Le calcul fait intervenir explicitement la contribution de τ_1 : $\frac{P}{P_1} * C_1$ qui sera intégré dans U, et $CR_1(t_c+1) + \overline{CR}_i(t_c+P+1) - C_1$ qui est négatif ou nul.

Puis nous avons :

$$\begin{aligned}
2P &= W(t_c+1, t_c+P+1) \\
&= \sum_{\tau_i \in RA} \frac{P}{P_i} * C_i \\
&+ \frac{P}{P_1} * C_1 + [CR_1(t_c+1) + \overline{CR}_1(t_c+P+1) - C_1] \\
&+ \sum_{\tau_i \in RT} [(\frac{P}{P_i} - 1) * C_i + \overline{CR}_i(t_c+P+1)] \\
&+ \sum_{\tau_i \in DR} [(\frac{P}{P_i} - 1) * C_i + \overline{CR}_i(t_c+P+1)] \\
&= 2P + [CR_1(t_c+1) + \overline{CR}_1(t_c+P+1) - C_1] \\
&+ \sum_{\tau_i \in RT} [\overline{CR}_i(t_c+P+1) - C_i] \\
&+ \sum_{\tau_i \in DR} [\overline{CR}_i(t_c+P+1) - C_i].
\end{aligned}$$

Or on a :
• $CR_1(t_c+1) + \overline{CR}_i(t_c+P+1) - C_1 \leq 0$ (lemme 3)
• $\overline{CR}_i(t_c+P+1) - C_i \leq 0$ pour toute tâche τ_i de RT $\cup$ DR (une instance ne peut exécuter plus que sa charge totale).
On en déduit donc que :
• $CR_1(t_c+1) + \overline{CR}_i(t_c+P+1) - C_1 = 0$
• $\overline{CR}_i(t_c+P+1) - C_i = 0$ pour toute tâche τ_i de RT $\cup$ DR.
Et donc on conclut comme dans le cas précédent. □

4.2.2 Preuve du lemme 3

Il nous faut maintenant prouver le lemme 3. Pour cela, nous devons différentier les cas où τ_1 s'exécute sans interruption depuis son activation jusqu'à t_c + 1 et où τ_1 est interrompue par des tâches plus prioritaires.

1^{er} *cas* - $IC_1(t_c)$ s'est exécutée sans discontinuer entre son activation et $t_c + 1$. Dans ce cas,la charge exécutée est maximale pour une plage de temps de cette durée, i.e. $\overline{CR}_1(t_c+1)$ est maximale, et donc $\overline{CR}_1(t_c+P+1) \leq \overline{CR}_1(t_c+1)$.
$2^{ème}$ *cas* - L'exécution de $IC_1(t_c)$ est discontinue entre son activation et t_c. Afin d'étudier plus finement le comportement de τ_1, nous introduisons les notions de points de préemption et de leur contexte.

Définition 4. *On appelle* **point de préemption** *tout instant t tel qu'il existe au moins une tâche prête à l'instant t qui ne s'exécute pas.*

Définition 5. *Soit t un point de prémption. On appelle* **contexte de prémption de t** *le quadruplet CtxP(t) = (t, τ_i, τ_j, liste) tel que :*
• *τ_i et τ_j s'exécutent à l'instant t.*
• *liste est la liste des tâches prêtes, mais qui ne s'exécutent pas à l'instant t.*

Nous pouvons noter que :

1. t point de préemption ⇒ liste $\neq \emptyset$
2. $\tau_k \in$ liste ⇒ Prio(τ_k) < Prio(τ_i) et Prio(τ_k) < Prio(τ_j)

Nous notons $(tp_1, \ldots, tp_f)$ la suite des points de préemption antérieurs à t_c (par définition, t_c n'est pas un point de préemption).
Nous montrons alors qu'une tâche prête qui ne s'exécute pas à un instant t ne s'exécute pas non plus une métapériode plus tard. Nous en déduirons donc qu'une tâche ne peut pas avoir exécuté à un instant t plus de charge qu'elle n'en avait exécuté une métapériode avant. Ce résultat, appliqué à la tâche τ_1 prouvera le lemme 3.

Lemme 6. *Soit tp_m un point de préemption. $\forall \tau_k \in CtxP(tp_m).liste, \tau_k$ ne s'exécute pas à l'instant tp_m + P.*

Avant de prouver ce lemme, nous présentons son corollaire et achevons alors la preuve du lemme 3.

Corollaire 7. *Si le lemme 6 est vérifié pour la suite $(tp_1, \ldots, tp_s)$, alors :*

I - $\forall tp_r$ *(r ≤ s),* $\forall \tau_k, \overline{CR}_k(tp_r + 1) \geq \overline{CR}_k(tp_r + P + 1)$

II - $\forall t < tp_{r+1}, \overline{CR}_k(t + 1) \geq \overline{CR}_k(t + P + 1)$

Preuve
I - Chaque fois que τ_k est prête mais non active, à un instant tp_u (u ≤ s), il en est de même à l'instant $tp_u + P$, i.e. τ_r ne s'exécute pas. Donc entre l'activation de $IC_k(tp_r$ + P + 1) et tp_r + P + 1, τ_r a attendu au moins autant qu'entre l'activation de $IC_k(tp_r + 1)$ et $tp_r + 1$. D'où le résultat.
II - Soit τ_k une tâche et soit t_a la date d'activation de IC_k(t).

1. τ_k s'est exécutée sans discontinuer de t_a à t. La charge exécutée $\overline{CR}_k$(t) est alors maximale, d'où le résultat.
2. τ_k a été interrompue. Soit tp_u le dernier point de préemption avant t ($tp_u \leq t$).
   a - t = tp_u. Le point I s'applique.
   b - $IC_k(t + 1)$ termine avant t + 1. Par suite, $\overline{CR}_k(t + 1) = C_k \geq \overline{CR}_k(t + P + 1)$.
   c - $tp_u <$ t. Ceci implique que t n'est pas un point de préemption.
      i. $tp_u < t_a$. Le raisonnement du point II.1 s'applique.
      ii. $t_a \leq tp_u$ On sait déjà, d'après le point I que $\overline{CR}_k(tp_u + 1) \geq \overline{CR}_k(tp_u + P + 1)$. De plus, τ_k s'exécute sans discontinuer de tp_u + 1 jusqu'à t (on a déjà traité le cas où elle termine avant t). On a donc $W_k(tp_u + 1, t + 1) = t - tp_u \geq W_k(tp_u + P + 1, t + P + 1)$. On en déduit alors le résultat.

□
Le point 2 du corollaire appliqué à s = f, k = 1 et t = t_c prouve le lemme 3.□

Pour achever notre preuve, il nous faut prouver le lemme 6. Nous le prouvons à l'aide d'une récurrence sur la séquence des points de préemption.
I - Considérons le premier point de préemption tp_1 de contexte CtxP(tp_1) = (tp_1, τ_{i_1}, τ_{j_1}, $liste_1$). Soit $\tau_k \in liste_1$.

1. τ_{i_1} et τ_{j_1} activées avant tp_1.
   Elles se sont donc exécutées continuellement depuis leurs activations, elles sont plus prioritaires que τ_k et comme tp_1 est le premier point de préemption, τ_k est activée à tp_1. On en déduit que $\overline{CR}_{i_1}(tp_1) \geq \overline{CR}_{i_1}(tp_1 + P)$ et $\overline{CR}_{j_1}(tp_1) \geq \overline{CR}_{j_1}(tp_1 + P)$ (à cause des exécutions continues), et on sait que $CR_{i_1}(tp_1)$ > 0 et $CR_{j_1}(tp_1)$ > 0 (puisque les deux tâches s'exécutent à l'instant tp_1). Par suite, à tp_1 + P, τ_{i_1} et τ_{j_1} n'ont pas non plus terminé leur exécution, elles sont plus prioritaires que τ_k, donc τ_k ne peut pas s'exécuter à tp_1 + P.
2. L'une des deux, soit τ_{i_1} est activée avant tp_1 et l'autre à tp_1. τ_k est activée au plus tard à tp_1, si elle est activée avant tp_1, elle s'exécute continuellement jusqu'à tp_1. On a cette fois $\overline{CR}_{i_1}(tp_1) \geq \overline{CR}_{i_1}(tp_1+P)$, $\overline{CR}_{j_1}(tp_1) = \overline{CR}_{j_1}(tp_1$ + P) = 0 (puisque τ_{j_1} est activée à tp_1). On conclut alors par un raisonnement similaire au précédent.
3. τ_{i_1} et τ_{j_1} activées à tp_1. Elles sont alors réactivées à tp_1 + P, elles sont plus prioritaires que τ_k, donc τ_k ne peut pas s'exécuter à tp_1 + P.

II - Supposons le résultat acquis au rang m - 1, et considérons le point de préemption tp_m. Grace à l'hypothèse de récurrence, nous pouvons utiliser le point 2 du corollaire 7 avec t = tp_m et k = i_m ou j_m. Nous avons donc :
• $\overline{CR}_{i_m}(tp_m) \geq \overline{CR}_{i_m}(tp_m + P)$
• $\overline{CR}_{j_m}(tp_m) \geq \overline{CR}_{j_m}(tp_m + P)$
• $CR_{i_m}(tp_m)$ > 0
• $CR_{j_m}(tp_m)$ > 0
On conclut alors comme dans les cas précédents. □

5 Conclusion

Nous avons mis en évidence au travers de quelques exemples les différences majeures entre le cas monoprocesseur et le cas multiprocesseurs, pour ce qui est du problème de la cyclicité. Nous avons ensuite caractérisé l'entrée en régime permanent pour les stratégies à priorités fixes. Comme dans le cas monoprocesseur, le cycle démarre après le dernier temps creux. Nous conjecturons que cette propriété s'étendra à toutes les stratégies localement déterministes, i.e. telles que, à tout instant, si on considère deux tâches quelconques, le choix de la plus prioritaires des deux dépend uniquement de l'état des deux tâches, et pas des autres tâches. Cette propriété est par exemple valide pour l'algorithme EDF. Par contre, la stratégie d'ordonnancement utilisée dans l'exemple de la figure 3 n'est pas localement déterministe : les tâches τ_3 et τ_4 sont dans le même état aux instants 1 et 10, mais à l'instant 1, on a ordonnancé τ_4 avant τ_3 alors que l'inverse se produit à l'instant 10. Par ailleurs, les raisonnements présentés devraient s'étendre sans difficulté à un nombre quelconque de processeurs :

il suffira de remplacer dans le cas de temps creux partiels, la tâche τ_1 par une liste de tâches. Pour achever cette première étude, il reste à déterminer une borne pour t_c. Pour les priorités fixes, nous conjecturons que t_c est bornée par Max(r_i) + P, comme dans le cas monoprocesseur. Notons que cette propriété n'est pas valide par exemple pour l'algorithme EDF.

Références

[1] J. Anderson, P. Holman, and A. Srinivasan. Fair scheduling of real time tasks on multiprocessors. *Handbook of scheduling : Algorithms, Models and Performance analysis*, pages 31.1–31.21, 2004.

[2] S.K. Baruah, N.K. Cohen, C.G. Plaxton, and D.A. Varvel. Proportionate progresse : a notion of fairness in resource allocation. *Algorithmica*, 15 :600–625, 1996.

[3] A.A. Bertossi and M.A Bonucelli. Preemptive scheduling of periodics jobs in uniform multiprocessor systems. *Information Processing Letters*, pages 3–6, 1983.

[4] A. Choquet-Geniet and E. Grolleau. Minimal schedulability interval for real time systems of periodic tasks with offsets. *Theoretical of Computer Science*, pages 117–134, 2004.

[5] M.L. Dertouzos and A.K.L. Mok. Multiprocessor scheduling in hard real-time environment. *IEEE transactions on sofware Engineering*, 15(12) :1497–1506, 1989.

[6] M.R. Garey and D.S. Johnson. Schedulling tasks with nonuniform deadlines on two processors. *journal of the Association for Computing Machinery*, 23(3) :461–467, 1976.

[7] R.M. Karp and V. Ramchandani. Parallel algorithms for shared-memory machines. In J.V. Leuuwen, editor, *Algorithms and complexity*, pages 869–935. MIT press, 1990.

[8] A. Khemka and R.K. Shyamasundar. An optimal multiprocessor real-time scheduling algorithm. *Journal of parallel and distributed computing*, 43(1) :37–45, 1997.

[9] G. Koren, E. Dar, and A. Amir. The power of migration in multiprocessor scheduling of real-time systems. *SIAM Journal of Computing*, 30(2) :511–527, 2000.

[10] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, pages 237–250, 1982.

[11] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1) :46–61, 1973.

[12] J. Liu and R. Ha. Efficient methods of validating timing constraints. *Advances in Real-Time Systems*, pages 199–223, 1995.

[13] R.R Muntz and E.G. Coffman jr. Preemptive scheduling of real-time tasks on multiprocessor systems. *Journal of the association for computng Machinery*, 17(2) :324–338, 1970.

[14] K. Ramamritham, J.A. Stankovic, and P. Shiah. O(n) scheduling algorithms for real-time multiprocessor systems. *International Conference on Parallel Processing*, 3 :143–153, 1989.

[15] J.A. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real time systems. *IEEE Computer*, 28(9) :16–25, 1995.

[16] J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 19(2) :139–153, 1993.