Ordonnancement de tâches indépendantes avec suspension

F. Ridouard
{frederic.ridouard@ensma.fr}
Tel. (+33/0)5 49 49 83 36

P. Richard
{pascal.richard@ensma.fr}
Tel. (+33/0)5 49 49 80 69

F. Cottet
{francis.cottet@ensma.fr}
Tel. (+33/0)5 49 49 80 52

Laboratoire d'Informatique Scientifique et Industrielle
École Nationale de Mécanique et d'Aérotechnique
Téléport 2 – BP 40109 F-86961 Chasseneuil Futuroscope Cedex, France
fax : (+33/0)5 49 49 80 64

Résumé : Les systèmes temps-réel utilisent souvent un modèle de tâche simplifié. Dans la majorité des applications, les tâches sont représentées par un unique bloc d'exécution. Mais certaines tâches ont besoin au cours de leur exécution d'effectuer des opérations externes (comme par exemple des opérations d'entrée/sortie) et ainsi doivent se suspendre. Ce sont des tâches dites à suspension. Dans cet article, nous montrons que le problème d'ordonnancement de tâche à suspension est un problème $\mathcal{NP}$-Difficile au sens fort. Nous démontrons également qu'il n'existe aucun algorithme optimal qui soit polynomial ou même pseudo-polynomial en temps de calculs pour choisir la prochaine tâche à s'exécuter. Et, nous démontrons la faiblesse de politiques classiques d'ordonnancement bien connues comme *EDF* (Earliest Deadline First), *RM* (Rate Monotonic), *DM* (Deadline Monotonic) et *LLF* (Least Laxity First) même sur des configurations de tâches avec une charge processeur faible, proche de zéro.

Plan

1- Introduction
2- Complexité
3- Anomalies d'ordonnancement
4- Analyse de compétitivité
5- Analyse de la charge processeur
5- Conclusion

Mots clés : Temps-réel, Ordonnancement, Tâches à suspension, Analyse de compétitivité.

1 Introduction

Le fonctionnement correct d'un système temps-réel ne dépend pas seulement de l'exactitude du résultat mais aussi du temps auquel il est survenu. Un système temps-réel peut se voir comme un système concurrentiel de tâches. Les tâches composant de tels systèmes ont des contraintes temporelles qu'elles doivent respecter. Les systèmes temps-réel utilisent généralement des processeurs annexes sur lesquels certaines tâches effectuent des opérations externes comme par exemple des opérations d'entrée-sortie ou des lancements de procédure. Ces tâches sont plus connues sous l'appellation de *tâche à suspension*. Le principe de ces tâches est qu'au cours de leur exécution, la tâche lance une opération externe engendrant sa suspension jusqu'à ce que l'opération externe soit terminée. Alors, la tâche se réveille et peut finir son exécution.

Soit I une configuration à n tâches. Les tâches seront considérées indépendantes. Chacune des tâches τ_i ($1 \leq i \leq n$) arrive dans le système à la date r_i, et doit terminer son exécution avant la date limite de fin d'exécution notée $d_i = r_i + D_i$ où D_i est l'échéance relative à sa date d'arrivée. Enfin, chacune des activations de τ_i est séparée par une période T_i. Une tâche τ_i est dite à échéance sur requête si sa période est égale à sa date limite de fin d'exécution ou $T_i = D_i$. Le facteur d'utilisation U_i d'une tâche τ_i est égal à $\frac{C_i}{T_i}$. Le facteur d'utilisation U_I d'une configuration de tâches I est égal à la somme des facteurs d'utilisation des tâches τ_i de cette configuration. Ainsi, $U_I = \sum_{i \in I} \frac{C_i}{T_i}$

La plupart des études existantes délaissent le problème des tâches à suspension ou au mieux le simplifient. Ainsi, le modèle de représentation des tâches utilisé dans l'ensemble de ces études est bien souvent réducteur car limité à un seul bloc d'exécution noté C_i comme le montre la figure 1. Ce qui implique que pour les tâches qui effectuent des opérations externes, le temps nécessaire à la tâche pour effectuer cette opération est compris à l'intérieur même du bloc d'exécution. Mais pendant qu'elle effectue une opération externe, une tâche n'utilise pas le processeur et par conséquent, elle peut laisser une autre tâche s'exécuter. Le modèle de tâches à suspension est donc plus réaliste.

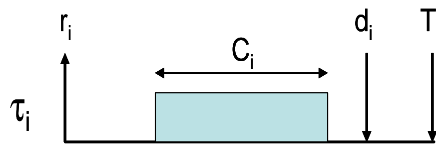

FIG. 1 – Caractéristiques des tâches dans les études simplifiées

Pour simplifier notre étude nous considérons que les tâches ne peuvent se suspendre qu'une seule fois au plus. Ce modèle est représenté par la figure 2. Chaque tâche τ_i ($1 \leq i \leq n$) est composée de deux blocs d'exécution, appelés aussi sous-tâches

(de longueur $C_{i,k}$, $1 \leq k \leq 2$), séparés par une suspension de durée maximale X_i pendant laquelle la tâche ne s'exécute pas. La somme des longueurs des deux blocs d'exécution $C_{i,k}$ ($1 \leq k \leq 2$) est notée C_i. A l'intérieur de chaque bloc d'exécution, par définition il n'y a aucune suspension. La durée de suspension d'une tâche τ_i varie d'une exécution à une autre sans jamais dépasser sa pire durée X_i.

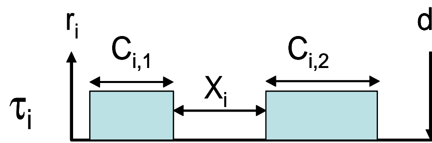

FIG. 2 – Le modèle des tâches

La théorie de l'ordonnancement temps-réel se focalise sur les tests d'ordonnançabilité pour s'assurer que toutes les échéances sont respectées. Plusieurs tests de faisabilité sont connus pour l'analyse de tâche à suspension. Des tests pour les algorithmes classiques existent déjà. Dans [3] est présenté un test basé sur le facteur d'utilisation du processeur. Ce test, pour les algorithmes à priorité fixe est basé sur le calcul du pire temps de réponse des tâches [6, 8]. Une telle approche peut aussi être utilisée pour l'algorithme d'ordonnancement *EDF* [8]. Mais nous montrons que ces ordonnanceurs sont globalement inefficaces pour les tâches à suspension.

Les systèmes temps réel ont souvent outre celle de respecter les contraintes temporelles de chaque tâche, une autre finalité : celle d'optimiser un critère de performance. Il y a deux types de critères à optimiser, les critères à minimiser comme par exemple, le temps moyen de réponse et des critères à maximiser comme le nombre de tâches respectant leurs échéances lorsque des dépassements d'échéance sont tolérés. Pour comparer plus finement les algorithmes, nous avons considéré les deux critères suivants : la minimisation du plus grand temps de réponse et la maximisation du nombre de tâches respectant leurs échéances.

Dans le paragraphe 2, nous présenterons la complexité de notre problème et plus particulièrement, nous démontrerons qu'il n'existe aucun algorithme optimal qui soit polynomial ou même pseudo-polynomial en temps de calculs. Ensuite, nous montrerons les faiblesses de certaines politiques d'ordonnancement comme *EDF* (Earliest Deadline First), *RM* (Rate Monotonic), *DM* (Deadline Monotonic) et *LLF* (Least Laxity First) même sur des configurations de tâches avec une charge processeur arbitrairement faible, proche de zéro. Le paragraphe 3 montrera la présence d'anomalies d'ordonnancement sous *EDF*. Dans le paragraphe 4, nous calculerons la compétitivité de ces algorithmes sur l'optimisation de deux critères : la minimisation du nombre de tâches en retard (nous démontrerons en plus que la technique d'augmentation des ressources est inefficace dans le cas de l'ordonnancement sous *EDF*) et la minimisation du temps de réponse maximum des tâches.

2 Complexité du problème

Dans cette partie, nous établissons que le problème d'ordonnancement de tâches qui peuvent se suspendre au plus une fois est un problème $\mathcal{NP}$-Difficile au sens fort. Nous avons déjà démontré (cf [10]) que ce problème est $\mathcal{NP}$-Difficile au sens fort pour l'ordonnancement de système de tâches périodiques, à départ simultané. Nous étudions le problème ouvert de complexité où les tâches sont à échéance sur requête et quand elles ne peuvent se suspendre qu'au plus une fois.

Théorème 1 : *L'ordonnancement de tâches périodiques, à échéances sur requête et se suspendant au cours de leur exécution au plus une fois est un problème $\mathcal{NP}$-Difficile au sens fort.*

Démonstration : cf Annexe 6.1

Un algorithme d'ordonnancement est dit *universel* si cet algorithme effectue le choix de la prochaine tâche à ordonnancer en temps polynomial [5]. Nous démontrons maintenant qu'il n'existe pas d'algorithme *universel* pour ordonnancer des tâches avec suspension, à moins que $\mathcal{P} = \mathcal{NP}$.

Théorème 2 : *S'il existe un algorithme d'ordonnancement universel pour les configurations de tâches où chaque tâche se suspend au plus une fois alors $\mathcal{P} = \mathcal{NP}$.*

Démonstration : cf Annexe 6.2

3 Les anomalies d'ordonnancement sous *EDF*

Dans cette sous-partie, nous allons étudier les possibilités d'apparition d'anomalies sous la politique d'ordonnancement *E*arliest-*D*eadline-*F*irst (*EDF*). Nous allons tout d'abord expliciter la notion d'*anomalie d'ordonnancement*. Soit A un algorithme, nous savons que le temps processeur requis pour A peut varier d'une exécution à une autre. A possède des anomalies d'ordonnancement, s'il existe une configuration I de tâches telle que diminuer la durée d'exécution ou de suspension d'une des tâches rend la configuration I non ordonnançable par A alors qu'elle l'était en considérant les pires durées d'exécution et de suspension de l'ensemble des tâches de la configuration. Un algorithme supportant les anomalies d'ordonnancement est dit *robuste*. En pratique, la robustesse simplifie les démonstrations puisque si la preuve qu'une configuration de tâches I en utilisant pour chaque tâche son pire temps processeur est ordonnançable avec l'algorithme A est établie, alors c'est une condition nécessaire et suffisante d'ordonnançabilité de la configuration par A.

Il a été prouvé dans [7] qu'*EDF* est robuste pour l'ordonnancement de tâches indépendantes sans suspension. Ainsi, dans un ordonnancement, toutes les dates limites de fin d'exécution sont respectées avec les pires durées d'exécution, diminuer la durée d'exécution d'une des tâches ne peut pas amener *EDF* à ne plus respecter l'échéance d'une des tâches du système. Nous allons démontrer que cette assertion est fausse dès lors que des tâches sont autorisées à se suspendre.

Théorème 3 : *Des anomalies d'ordonnancement peuvent apparaître en exécutant des tâches à suspension par l'algorithme d'ordonnancement EDF.*

Démonstration :
Afin de démontrer ce théorème, une configuration de tâches I va être définie. Puis sur cette configuration, il va être démontré que si la durée d'exécution ou la durée de suspension d'une des tâches est réduite d'un unité de temps, une échéance ne sera plus respectée. Soit I la configuration étudiée à trois tâches suivante :

$$\tau_1 : r_1 = 0, C_{1,1} = 2, X_1 = 2, C_{1,2} = 2, D_1 = 6, T_1 = 10$$
$$\tau_2 : r_2 = 5, C_{2,1} = 1, X_2 = 1, C_{2,2} = 1, D_2 = 4, T_2 = 10$$
$$\tau_3 : r_3 = 7, C_{3,1} = 1, X_3 = 1, C_{3,2} = 1, D_3 = 3, T_3 = 10$$

Lorsque toutes les tâches sont exécutées avec leurs pires durées d'exécution et de suspension, *EDF* définit l'ordonnancement suivant : à l'instant 0, il ordonnance la tâche τ_1 jusqu'à l'instant 2. Elle se suspend de l'instant 2 à 4. A l'instant 4, elle redémarre son exécution et la termine à l'instant 6. A ce moment là, la tâche τ_2 se réveille et s'exécute jusqu'à l'instant 7. Moment auquel elle se suspend et auquel la tâche τ_3 se réveille et alors commence son exécution. A l'instant 8, la tâche τ_3 se suspend et alors la tâche τ_2 qui ayant fini sa suspension peut finir son exécution à l'instant 9. La tâche τ_3 se réveillant à ce même instant peut finir elle-même son exécution à l'instant 10. Cet ordonnancement est montré sur la figure 3.a. L'ordonnancement se déroule correctement et toutes les échéances sont respectées. Par conséquent, en utilisant les pires durées de suspension et d'exécution, l'ordonnancement de I par *EDF* est faisable.

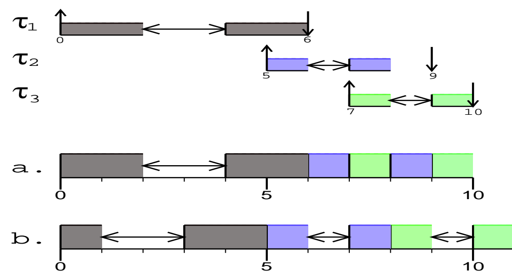

FIG. 3 – Ordonnancement de la configuration I avec ses pires durées d'exécution des tâches

Maintenant, en diminuant la durée d'exécution de la tâche τ_1 ou sa durée de suspension d'une unité de temps, il va être démontré que la tâche τ_3 ne respecte plus son échéance. Par exemple, diminuons la durée d'exécution $C_{1,1}$ d'une unité de temps : $C_{1,1} = 1$ et laissons toutes les autres durées d'exécution et de suspension inchangées. Comme τ_1 a une unité de moins d'exécution, elle termine par conséquent son exécution une unité plus tôt donc à l'instant 5. Mais à cet instant, τ_2 se réveille et peut par conséquent commencer son exécution une unité plus tôt. Par conséquent à l'instant 7 quand elle se réveille après sa suspension, elle rentre en concurrence avec la tâche τ_3 qui s'active. Cette concurrence n'existait pas au préalable puisque quand toutes les tâches sont exécutées avec leurs pires durées d'exécution et de suspension, la tâche τ_2 se suspend à la date 7 et ne demande donc pas à s'exécuter. La tâche τ_2 disposant d'une échéance plus petite que celle de la tâche τ_3, elle peut s'exécuter et finir son exécution à l'instant 8. Mais de ce fait, la tâche τ_3 se retrouve retardée d'une unité de temps et par conséquent elle ne respecte plus son échéance (cf. Figure 3.b), ce qui provoque une anomalie d'ordonnancement. Ainsi, en diminuant la durée d'exécution de la première sous-tâche de τ_1, l'ordonnancement est devenu infaisable. La même anomalie apparaît avec le même scénario en diminuant d'une unité de temps la durée de suspension X_1 ou la durée d'exécution $C_{1,2}$ (i.e. $X_1 = 1$ ou $C_{1,2} = 1$).

Enfin, il est facile de démontrer que dans tous les cas présentés où l'algorithme *EDF* échoue, il existe des ordonnancements (hors-lignes) faisables.

□

4 Analyse de compétitivité

4.1 Introduction

Dans cette partie, l'étude des comportements d'algorithmes bien connus tels que *EDF* ou *LLF* en ordonnançant des configurations de tâches à suspension va montrer que ces algorithmes ne sont pas ou peu compétitifs pour la minimisation du nombre de tâches en retard et pour la minimisation du temps de réponse maximum. Pour déterminer si un algorithme en-ligne est compétitif, une technique couramment utilisée est l'analyse de compétitivité. Pour simplifier les résultats, nous supposons que les périodes des tâches sont prises suffisamment grandes pour qu'une unique occurrence de chaque tâche appartienne à l'hyperpériode.

4.2 Analyse de compétitivité

L'analyse de compétitivité compare l'algorithme en-ligne à évaluer à un algorithme clairvoyant optimal appelé aussi *l'adversaire*. Un bon adversaire définit des instances de problème pour que l'algorithme en-ligne atteigne sa pire performance. Un algorithme qui minimise un critère de performance est c-compétitif si la performance obtenue par l'algorithme en-ligne est c fois la valeur obtenue par l'adversaire. Plus formellement, étant donné un algorithme en-ligne A à évaluer, et une configuration

de tâches I. La performance obtenue par A en ordonnançant I est notée $\sigma_A(I)$ et celle obtenue par l'adversaire sur cette même configuration est notée $\sigma^*(I)$. A est dit c-compétitif s'il existe une constante c telle que $\sigma_A(I) \leq c\sigma^*(I)$. Le ratio de compétitivité c_A de l'algorithme en-ligne A est égal à la pire valeur en considérant toutes les instances I du problème :

$$c_A = \sup_{toutI} \frac{\sigma_A(I)}{\sigma^*(I)}$$

Selon le critère à optimiser, il appartient à l'une des deux catégories suivantes :
- Minimisation : La valeur du ratio est supérieure ou égale à 1. Si la valeur du ratio de compétitivité d'un algorithme en-ligne est 1 alors il est optimal pour le critère étudié. Sinon, si ce ratio est différent de toute constante, le ratio est une fonction en n, où n est le nombre de tâches. Dans ce cas, l'algorithme est non-compétitif.
- Maximisation : La valeur du ratio est comprise entre 0 et 1. Si la valeur du ratio de compétitivité d'un algorithme en-ligne est 1 alors il est optimal pour le critère étudié. Sinon, si son ratio vaut 0, il est non-compétitif.

4.3 Minimisation du nombre de tâches en retard

Le critère de minimisation du nombre de tâches en retard est étudié dans cette partie. Ce critère est bien sûr équivalent à celui de la maximisation du nombre de tâches respectant leurs échéances. Nous allons rappeler tout d'abord les principaux résultats connus puis nous étudierons les performances d'*EDF* ou *LLF* pour l'ordonnancement de tâches à suspension.
Nous savons qu'il n'existe pas d'algorithme en-ligne compétitif pour la maximisation du nombre de tâches respectant leurs échéances en ordonnançant des systèmes de tâche classique, sans suspension. Mais il en existe dans des cas particuliers [1, 2]. Dans ce contexte particulier, nous allons prouver qu'en ordonnançant des tâches à suspension, les algorithmes classiques comme *EDF* ne sont pas compétitifs. Il est à noter que nos résultats sont valides d'un point de vue de la faisabilité puisque les résultats que nous allons présenter, le sont avec pour les configurations étudiées, des facteurs d'utilisation faibles, proches de zéro.

4.3.1 Résultats connus

Baruah *et al.* [1, 2] ont prouvé qu'il n'existe pas d'algorithme d'ordonnancement en-ligne préemptif pour maximiser le nombre de tâches respectant leurs échéances dans les systèmes monoprocesseurs. Pour obtenir son résultat, l'adversaire définit une configuration de tâches avec des surcharges du processeur. Mais ces auteurs montrent qu'il existe des résultats positifs dans des cas particuliers. Nous allons maintenant présenter un de ces cas. La définition 1 présente les propriétés d'un de ces cas particuliers.

Définition 1 *Monotonic Absolute Deadlines (MAD) :*
Un système de tâche est qualifié de MAD si l'échéance absolue de chaque nouvelle

tâche qui arrive dans le système est supérieure ou égale à celle de n'importe quelle tâche déjà arrivée dans le système.

La définition 2 présente l'algorithme *SRPTF*.

Définition 2 *Shortest Remaining Processing Time First (SRPTF) :*
SRPTF est un algorithme d'ordonnancement en-ligne préemptif. Il alloue à chaque instant, le processeur à la tâche ayant le plus petit temps processeur restant à exécuter.

Dans [1, 2], Baruah *et al.* ont prouvé que si un système de tâches a la propriété *MAD* alors l'algorithme d'ordonnancement en-ligne *SRPTF* est 2-compétitif pour minimiser le nombre de tâches en retard. *SRPTF* est alors un des meilleurs algorithmes en-ligne possibles.

4.3.2 Mauvais résultats pour l'ordonnancement de tâches à suspension

La démonstration qu'en ordonnançant des systèmes de tâches à suspension, l'algorithme d'ordonnancement *SRPTF* n'est plus compétitif pour minimiser le nombre de tâches en retard et ce même si le système est *MAD*, est présentée ici :

Théorème 4 : *Pour les systèmes de tâches à suspension MAD même avec un facteur d'utilisation arbitrairement petit, l'algorithme en-ligne SRPTF n'est pas compétitif pour maximiser le nombre de tâches respectant leurs échéances.*

Démonstration : L'algorithme clairvoyant génère la configuration I à $n+1$ tâches suivante :

$$\tau_0 : r_0 = 0, C_{01} = 1, X_1 = 0, C_{02} = 0, D_0 = K$$
$$\tau_i : r_i = i-1, C_{i1} = 1, X_i = K-2, C_{i2} = 1, D_i = K \quad avec \quad i \in \{1,..,n\}$$

Où K est un entier positif strictement supérieur à 1.
La configuration I respecte bien la propriété *MAD*. La figure 4 montre les résultats de l'ordonnancement de I par l'algorithme d'ordonnancement *SRPTF* et par son adversaire clairvoyant. A la date 0, τ_0 et τ_1 sont disponibles, *SRPTF* ordonnance τ_0 parce que τ_0 a le plus petit temps processeur restant. Ensuite, pour chaque tâche τ_i, ($1 \leq i \leq n$), la première sous-tâche est ordonnancée à la date i et la seconde à la date $K + (i-1)$. L'adversaire, lui choisit d'ordonnancer la tâche τ_1 à l'instant 0. Puis, il ordonnance chaque tâche τ_i ($i \in \{2, \ldots, n\}$) à l'instant $i-1$. Pour finir, il ordonnance τ_0.

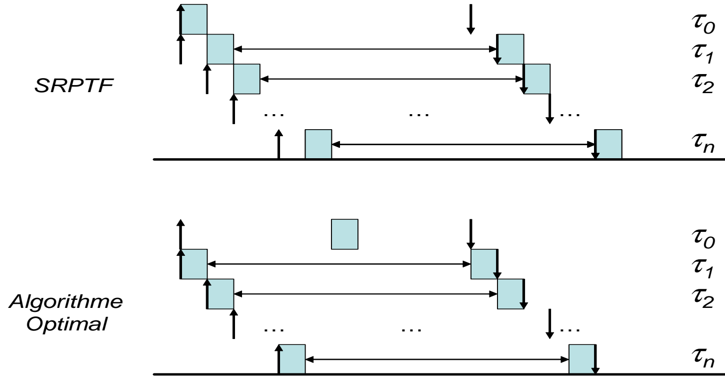

FIG. 4 – Limites de *SRPTF* dans un système de tâches *MAD*

Calcul du ratio de compétitivité de *SRPTF* :

$$c_{SRPTF} = \frac{\sigma_{SRPTF}}{\sigma_{Opt}} = \lim_{n \to \infty} \frac{1}{n+1} = 0$$

Et pour finir, calculons le facteur d'utilisation de I :

$$\begin{aligned} U_I &= \sum_{i=0}^{n} \frac{C_{i,1}+C_{i,2}}{T_i} = \frac{1}{K-1} + \sum_{i=1}^{n} \frac{2}{K} \\ &= \lim_{K \to \infty} \frac{2n+1}{K} = 0 \end{aligned}$$

Pour conclure, une configuration de tâches I a été générée par l'adversaire. Elle respecte la propriété du *MAD* et a un faible facteur d'utilisation alors que *SRPTF* ne soit pas compétitif pour maximiser le nombre de tâches respectant leurs échéances.

□

Avec la même configuration I que celle présentée dans la démonstration du théorème 4, les résultats obtenus pour *SRPTF* peuvent s'étendre à *EDF*, *DM* and *RM*.

Corollaire 1 : *Pour des systèmes de tâches à suspension et ayant la propriété du MAD, EDF, DM et RM ne sont pas compétitifs pour maximiser le nombre de tâches respectant leurs échéances.*

Démonstration : Soit I la configuration générée par l'adversaire lors de la démonstration du théorème 4. Sur cette configuration, *EDF*, *DM* et *RM* assignent exactement les mêmes priorités aux tâches que *SRPTF* le fait. Par conséquent, nous obtenons les mêmes ratios de compétitivité et facteurs d'utilisation pour ces algorithmes que pour *SRPTF*. Finalement, la même conclusion s'impose à savoir que ni *EDF*, ni *DM* et ni *RM* ne sont compétitifs pour maximiser le nombre de tâches respectant leurs échéances.

□

Étude de l'algorithme *LLF*.

Théorème 5 : *Pour des systèmes de tâches à suspension et ayant la propriété MAD, LLF n'est pas compétitif pour maximiser le nombre de tâches respectant leurs échéances.*

Démonstration :
Pour l'algorithme *L*east *L*axity *F*irst (*LLF*), soit I la configuration à n tâches suivante :

$$\tau_i : r_i = 0, C_{i1} = 3, X_i = K - 3(n+1), C_{i2} = 3, D_i = K \quad avec \quad \forall i = 1..n$$

où K est un nombre très grand tendant vers l'infini.

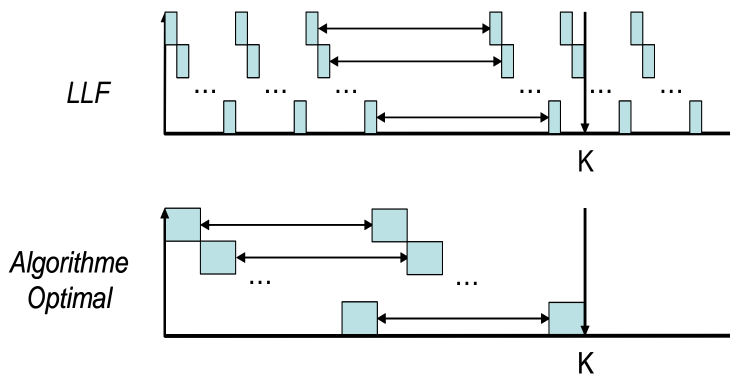

FIG. 5 – Etude de *LLF* sur la minimisation du nombre de tâche en retard

Les résultats de l'ordonnancement de I par *LLF* sont donnés par la figure 5. A l'instant 0, *LLF* commence par ordonnancer la tâche τ_1. Mais après avoir exécuté τ_1 pendant une unité de temps, les autres tâches τ_i ($i \in 2, .., n$) ont une priorité plus importante que celle de τ_1. Par conséquent, *LLF* préempte τ_1 et exécute τ_2. Mais, après une unité de temps d'exécution de τ_2, les autres tâches ont une priorité plus importante que celle de τ_2. Les tâches τ_i ($i \in 1, .., n$) ont la même laxité obligeant *LLF* à préempter à chaque fois la tâche active après seulement une unité de temps de son exécution. En conséquence, l'exécution des tâches ne peut se faire qu'unité après unité. Ce qui oblige *LLF* à ne respecter aucune des échéances de ces tâches.
L'algorithme optimal ordonnance la première sous-tâche de τ_1 puis la première sous-tâche de τ_2 pour finir avec la première sous-tâche de τ_n et dans le même ordre, la seconde sous-tâche de ces tâches. Par conséquent, l'algorithme optimal ne manque aucune échéance.
Le calcul du ratio de compétitivité de *LLF* donne :

$$c_{LLF} = \frac{\sigma_{LLF}}{\sigma_{Opt}} = \frac{0}{n} = 0$$

Ce qui implique que *LLF* n'est pas compétitif pour la maximisation du nombre de tâches respectant leurs échéances. Et le calcul du facteur d'utilisation de cette configuration I (cf. 1), montre bien qu'il est quasi nul. Ce qui nous permet de conclure quant à la faiblesse de *LLF*.

$$U_I = \sum_{i=1}^{n} \frac{C_{i1} + C_{i2}}{T_i} = \lim_{K \to \infty} \frac{6n}{K} = 0 \quad (1)$$

□

4.3.3 Technique de l'augmentation de ressources

En analyse de compétitivité, l'algorithme en-ligne et l'adversaire clairvoyant s'exécutent sur la même machine. Mais les résultats obtenus par cette technique d'analyse, sont quelques fois pessimistes et afin de diminuer ce pessimisme des alternatives existent. Afin d'analyser les performances relatives des algorithmes sous un autre angle, une extension de l'analyse de compétitivité a été créée : l'augmentation de ressources [9]. Dans cette extension, l'algorithme en-ligne s'exécute sur une machine plus rapide que celle de l'adversaire. Il a été prouvé dans [9] que si vous disposez d'une configuration de tâches qui est, avec une même machine, non-ordonnançable par *EDF* mais ordonnançable par un algorithme optimal alors, elle sera ordonnançable par *EDF* avec une machine deux fois plus rapide. Cette dernière assertion est fausse dès que le système est composé de tâches à suspension et nous le démontrons.

Le résultat connu d'*EDF* ne tient plus si les tâches sont à suspension. Le théorème qui suit, va démontrer que même s'il existe une configuration I ordonnançable par un algorithme optimal alors, elle ne sera pas toujours ordonnançable par *EDF* même sur une machine plus rapide.

Théorème 6 : *Augmenter la vitesse s du processeur n'améliore pas les performances d'EDF quand les tâches sont autorisées à se suspendre par rapport à un algorithme d'ordonnancement optimal utilisant un processeur de vitesse unitaire.*

Démonstration :
Raisonnement par l'absurde :
Hypothèse : Il existe un entier s, $s > 1$ tel que si *EDF* dispose d'une machine s fois plus rapide que celle de son adversaire clairvoyant, alors *EDF* est optimal.
Soit la configuration I à deux tâches suivante :

$$\tau_1 : r_1 = 0, C_{1,1} = 2s, X_1 = 0, C_{1,2} = 0, D_1 = 4s - 1$$

$$\tau_i : r_i = 0, C_{i,1} = 1/n, X_i = 4s - 2 + \frac{n-1}{n}, C_{i,2} = 1/n, D_i = 4s \qquad 2 \leq i \leq n$$

– La figure 6.a donne l'ordonnancement de I par un algorithme optimal. Cette figure montre que l'adversaire a respecté toutes les échéances. Les tâches τ_i $(1 \leq i \leq n)$ sont ordonnancées avant τ_1 dès leur arrivée, à l'instant 0.

– La figure 6.b montre l'ordonnancement de I par l'algorithme *EDF* sur une machine s fois plus rapide. A l'instant 0, *EDF* ordonnance τ_1 car son échéance est plus proche que celle des autres tâches. Ainsi, il retarde l'exécution des tâches τ_i $(2 \leq i \leq n)$ qui manquent leurs échéances. Par conséquent, *EDF* même avec une machine s fois plus rapide ne parvient toujours pas à respecter toutes les échéances et par conséquent à construire un ordonnancement faisable.

Calculons maintenant le ratio de compétitivité de l'algorithme d'ordonnancement *EDF* avec une machine s fois plus rapide. L'algorithme optimal respecte quant à lui toutes les échéances comme le montre la figure 6.a à la différence d'*EDF* qui ne respecte qu'une seule échéance, celle de τ_1 (cf. figure 6.b) :
Le ratio de compétitivité d'*EDF* en faisant tendre le nombre d'instances (n) vers l'infini est égal à :

$$c_{EDF} = \frac{\sigma_{EDF}}{\sigma_{Opt}} = \lim_{n \to \infty} \frac{1}{n} = 0$$

EDF n'est par conséquent pas compétitif pour la minimisation du nombre de tâches ne respectant pas leur échéance, même en augmentant la vitesse de la machine allouée à l'algorithme d'ordonnancement *EDF*.
Ce qui contredit l'hypothèse de départ et conclut cette démonstration.

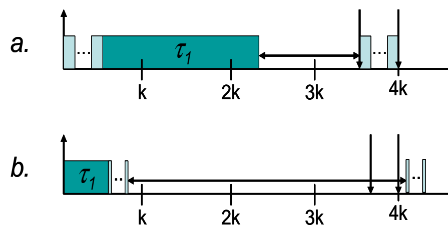

FIG. 6 – L'ordonnancement de I par *EDF*

□

La technique d'augmentation des ressources utilisée ainsi, est inefficace pour *EDF* car lorsqu'*EDF* dispose d'une machine plus rapide, seule la machine sur laquelle s'exécute les tâches a été améliorée. Ainsi lorsque les tâches ont besoin de se suspendre pour effectuer des opérations sur des machines auxiliaires, elles le font sur des machines qui n'ont pas été accélérées. Effectuant ces opérations sur la même machine

que le fait l'algorithme optimal, *EDF* ne va pas plus vite et par conséquent, les durées de suspension ne sont pas améliorées et donc pas diminuées.

4.4 Minimisation du temps de réponse maximum

Lorsque les tâches ne se suspendent pas, pour tout algorithme d'ordonnancement conservatif (i.e. n'insérant pas de temps creux dans l'ordonnancement s'il existe au moins une tâche prête à s'exécuter), alors le plus grand temps de réponse d'une tâche ne peut excéder la durée de la période d'activité synchrone (*Synchronous Busy Period*). Mais cette assertion devient inexacte quand les tâches sont à suspension comme l'illustre les résultats suivants pour *RM, DM, EDF, LLF*.

4.4.1 Compétitivité d'*EDF*, *RM* et *DM*

La politique d'ordonnancement *EDF* est au mieux 2-compétitive pour minimiser le temps de réponse maximum.

Théorème 7 : *L'algorithme d'ordonnancement EDF est au mieux* 2*-compétitif pour la minimisation du temps de réponse maximum en ordonnançant des tâches à suspension, se suspendant au plus une fois.*

Démonstration :
Soit la configuration de tâches I suivante :

$$\tau_1 : r_1 = 0, C_{1,1} = \epsilon, X_1 = K, C_{1,2} = \epsilon, D_1 = 4K$$
$$\tau_2 : r_2 = 0, C_{2,1} = K, X_2 = 0, C_{2,2} = 0, D_2 = 4K - 1$$

Où K est un entier positif supérieur à 1.
Où ϵ est un nombre positif strictement inférieur à 1 et tendant vers zéro.
Sur la figure 7 sont présentés les résultats de l'ordonnancement de la configuration I par l'algorithme d'ordonnancement *EDF* et par un algorithme clairvoyant optimal. L'algorithme *EDF* ordonnance en premier la tâche τ_2 car son échéance est la plus proche. Mais l'algorithme clairvoyant, ordonnance en premier la tâche τ_1 pour permettre d'ordonnancer τ_2 pendant la suspension de τ_1.

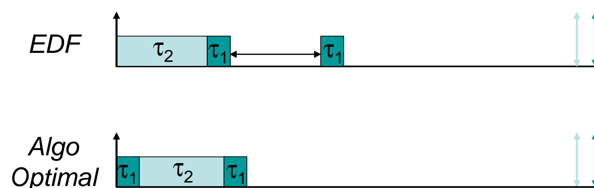

FIG. 7 – Compétitivité d'*EDF* pour la minimisation du temps de réponse maximum

Le calcul du ratio de compétitivité d'*EDF* finit cette démonstration. Le temps de réponse obtenu par *EDF* pour l'ordonnancement de I est égal à $2K + 2\epsilon$ alors que celui de l'algorithme optimal est de $K + 2\epsilon$. Ce qui nous permet d'obtenir un ratio de compétitivité d'*EDF* de 2 en faisant tendre ϵ vers 0 (cf. 2).

$$c_{EDF} = \frac{\sigma_{EDF}}{\sigma_{OPT}} = \lim_{\epsilon \to 0} \frac{2K + 2\epsilon}{K + 2\epsilon} = \frac{2K}{K} = 2 \quad (2)$$

Ce qui conclut notre démonstration puisqu'une configuration I a été trouvée sur laquelle le ratio de compétitivité de l'algorithme d'ordonnancement *EDF* est de 2. Par conséquent, *EDF* est un algorithme au mieux 2-compétitif.

□

Nous allons maintenant étendre ce résultat aux algorithmes *RM* et *DM*.

Corollaire 2 : *Les algorithmes d'ordonnancement RM et DM sont au mieux* 2-*compétitif pour l'ordonnancement de tâches à suspension où chaque tâche se suspend au plus une seule fois, et en minimisant le temps de réponse maximum.*

Démonstration :
La configuration I utilisée dans la démonstration du théorème 7 est conservée. Si nous ordonnançons cette configuration I avec les algorithmes *RM* et *DM*, les priorités affectées par ces algorithmes seront les mêmes que celles affectées par *EDF*. Par conséquent comme *EDF* dans la démonstration du théorème 7, *RM* et *DM* ordonnancent τ_2 avant τ_1, ce qui implique que les mêmes résultats que pour l'algorithme *EDF* sont obtenus. Ainsi, *RM* et *DM* ont un ratio de compétitivité d'au moins deux. Ce qui permet de conclure que *RM* et *DM* sont au mieux 2-compétitifs.

□

4.4.2 Compétitivité de *LLF*

La politique d'ordonnancement *LLF* est au mieux 2-compétitive pour minimiser le temps de réponse.

Théorème 8 : *L'algorithme d'ordonnancement LLF est au mieux* 2-*compétitif pour la minimisation du temps de réponse maximum pour l'ordonnancement de tâches à suspension et se suspendant au plus une fois.*

Démonstration :
Pour démontrer ce théorème, le ratio de compétitivité de *LLF* en ordonnançant une certaine configuration I doit être égal à 2. Soit I la configuration suivante :

$$\tau_1 : r_1 = 0, C_{1,1} = \epsilon, X_1 = K, C_{1,2} = \epsilon, D_1 = 4K$$
$$\tau_2 : r_2 = 0, C_{2,1} = K, X_2 = 0, C_{2,2} = 0, D_2 = 2K + 2$$

Où K est un entier positif supérieur à 1.
Où ϵ est un nombre positif strictement inférieur à 1 et tendant vers zéro.
La figure 8 montre les résultats de l'ordonnancement de la configuration de tâches I par l'algorithme d'ordonnancement *LLF* et par un algorithme clairvoyant optimal. L'algorithme *LLF* (cf figure 8) ordonnance en premier, la tâche τ_2 car sa laxité dynamique est plus faible. Quant à l'algorithme optimal, il commence à ordonnancer τ_1 pour permettre d'exécuter τ_2 pendant la suspension de τ_1.

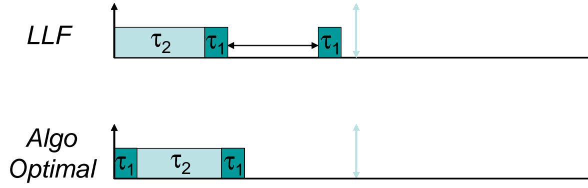

FIG. 8 – Compétitivité de *LLF* pour la minimisation du plus grand temps de réponse

Pour finir, calculons le ratio de compétitivité de *LLF* (3), en sachant que la longueur d'ordonnancement obtenue par *LLF* est égal à $2K - 2\epsilon$ et que celle obtenue par un algorithme optimal n'est que de $K + 2\epsilon$.

$$c_{LLF} = \frac{\sigma_{LLF}}{\sigma_{OPT}} = \lim_{\epsilon \to 0} \frac{2K - 2\epsilon}{K + \epsilon} = \frac{2K}{K} = 2 \tag{3}$$

Ce qui permet de conclure la démonstration puisqu'une configuration I sur laquelle le ratio de compétitivité de l'algorithme d'ordonnancement *LLF* est de 2 a été déterminée. Et donc, *LLF* est un algorithme au mieux 2-compétitif.

□

5 Conclusion

Nous avons présenté plusieurs résultats négatifs sur l'ordonnancement de tâches qui peuvent se suspendre pour effectuer des opérations sur des machines auxiliaires externes au système. Nous avons premièrement prouvé que l'ordonnancement de tâches qui peuvent se suspendre au plus une fois est un problème $\mathcal{NP}$-Difficile au sens fort et il n'y a pas d'algorithme universel à moins que $\mathcal{P} = \mathcal{NP}$. Nous avons également établi qu'il existe des anomalies d'ordonnancement en exécutant des tâches à suspension, sous la politique d'ordonnancement *EDF*. En utilisant l'analyse de compétitivité, nous avons démontré que les algorithmes classiques peuvent dépasser des échéances

et ce même si la charge processeur des configurations est presque nulle alors que des algorithmes hors-ligne simples qui respectent l'ensemble des échéances peuvent être facilement établis. Enfin, nous avons démontré que la technique d'augmentation des ressources n'amène aucune amélioration pour l'ordonnancement de tâches à suspension sous *EDF*.
L'une des perspectives de ce travail serait de chercher des solutions pour l'ordonnancement de tâches à suspension. Une autre serait de considérer des configurations de tâches non indépendantes.

Références

[1] S. Baruah, J. Haritsa, and N. Sharma. On-line scheduling to maximize task completions. *The Journal of Combinatorial Mathematics and Combinatorial Computing*, 39 :65–78, 2001.

[2] S. Baruah, J. Haritsa, and N. Sharma. On-line scheduling to maximize task completions. *In Proceedings of the 15th IEEE Real-TIme Systems Symposium, San Juan, Puerto Rico*, Dec 1994.

[3] U. C. Devi. An improved schedulability tast for uniprocessor periodic task systems. *proc. Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 23–30, 2003.

[4] R. Garey and D. S. Johnson. Computers and intractability : A guide to the theory of np-completeness. *W. H. Freemann*, 1979.

[5] K. Jeffay, D.F. Stanat, and C.U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. *proc. Real-Time Systems Symposium*, pages 129–139, 1991.

[6] I-G. Kim, K-H. Choi, S-K. Park, D-Y. Kim, and M-P. Hong. Real-time scheduling of tasks that contain the external blocking intervals. *Real-Time and Embedded Computing Systems and Applications(RTCSA'95)*, 1995.

[7] Jane W. S. Liu. *Real-Time Systems*, chapter Priority-Driven Scheduling of Periodics Tasks, pages 164–165. Prentice Hall, 2000.

[8] J.C. Palencia and M. Gonzales-Harbour. Offset-based response time analysis of distributed systems scheduled under edf. *Proceedings of the IEEE Real-Time Systems Symposium*, 2003.

[9] C.A Philips, C. Stein, E Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *prc.* 29th *Ann. ACM Symp. on Theory of Computing*, pages 110–149, 1997.

[10] P. Richard. On the complexity of scheduling tasks with self-suspensions on one processor. *proc. Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 187–194, 2003.

6 Annexes

6.1 Démonstration du théorème 1

Théorème 1 : *L'ordonnancement de tâches périodiques, à échéances sur requête et se suspendant au cours de leur exécution au plus une fois est un problème $\mathcal{NP}$-Difficile au sens fort.*

Démonstration :
Pour démontrer ce théorème, nous allons transformer une instance du problème de 3-partition connu pour être $\mathcal{NP}$-Complet au sens fort en une instance de notre problème [4]. Enfin, nous essayerons d'ordonnancer cette configuration générée.

Instance du problème de 3*-partition :* Soit A un ensemble de $3m$ éléments de $\mathbb{N}$, B une limite appartenant à $\mathbb{N}$ et pour tout $j \in \{1 \dots 3m\}$, $s_j \in \mathbb{N}$ tels que $B/4 < s_j < B/2$ et tel que $\sum_{i=1}^{3m} s_j = mB$.

Problème : Peut on partitionner A en m ensembles disjoints (A_1, A_2, ..., A_m) tels que, pour $1 \leq i \leq m$, $\sum_{j \in A_i} s_j = B$. Ce qui implique que chaque ensemble A_i, $i \in \{1, \dots, n\}$ contient exactement trois éléments. En effet, il ne peut ni en contenir moins car $s_j < B/2$ et ni en contenir plus car $B/4 < s_j$.

Nous générons à partir de l'instance du problème 3-partition, une instance d'ordonnancement à $3m + 1$ tâches :

– Les tâches $\tau_1, \dots, \tau_{3m}$ sont générées avec le même profil :

$$i \leq 3m \qquad \begin{cases} C_{i,1} = C_{i,2} & = & s_i \\ X_i & = & (2m-1)B \\ D_i = T_i & = & 4mB \end{cases}$$

– La tâche τ_{3m+1} :

$$\begin{aligned} C_{3m+1,1} &= \left\lceil \frac{B}{2} \right\rceil \qquad C_{3m+1,2} = \left\lfloor \frac{B}{2} \right\rfloor \\ X_{3m+1} &= B \\ D_{3m+1} &= T_{3m+1} = 2B \end{aligned}$$

Nous allons maintenant prouver qu'une solution pour le problème de 3-partition existe si et seulement si la configuration de tâches générée à partir de ce problème est ordonnançable.

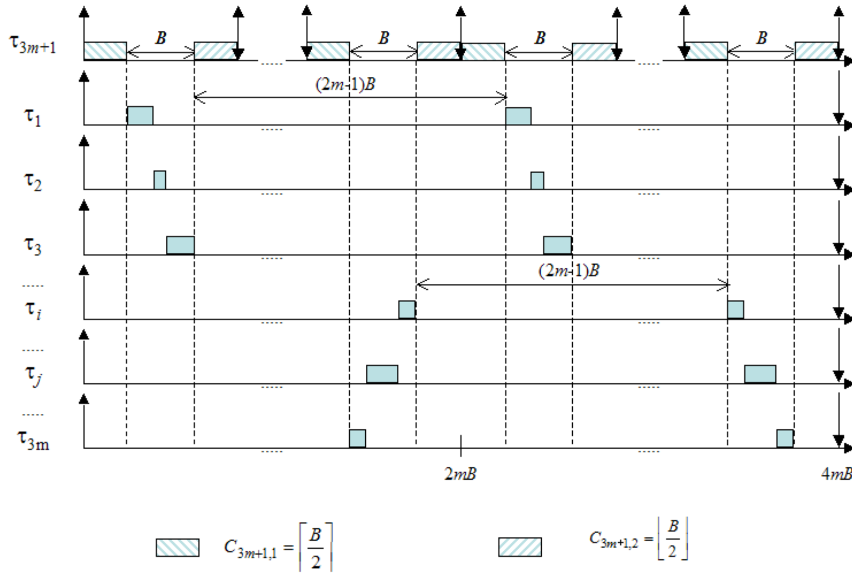

FIG. 9 – Ordonnancement faisable de la configuration générée

Les tâches générées, sont toutes à départ simultané. Par conséquent, nous pouvons limiter notre étude d'ordonnançabilité à l'hyperpériode dont la longueur est égale à $H = ppcm(T_1, \ldots, T_{3m+1}) = 4mB$. Nous limiterons notre étude qu'à l'intervalle $[0, 4mB]$.
Nous pouvons également montrer que le facteur d'utilisation de cette configuration est égal à 1

– Le facteur d'utilisation des tâches $\tau_1, \ldots, \tau_{3m}$ est égal à :

$$\sum_{i=1}^{3} m + \frac{C_{i,1}+C_{i,2}}{T_i} = \sum_{i=1}^{3} m2\frac{s_i}{4mB} = \frac{2mB}{4mB} = \frac{1}{2}$$

– Le facteur d'utilisation de τ_{3m+1} vaut :

$$\frac{C_{3m+1,1}+C_{3m+1,2}}{T_{3m+1}} = \frac{\lceil \frac{B}{2} \rceil + \lfloor \frac{B}{2} \rfloor}{2B} = \frac{B}{2B} = \frac{1}{2}$$

Le facteur d'utilisation de cette configuration est par conséquent bien égal à $\frac{1}{2}+\frac{1}{2} = 1$. Ce résultat implique que le processeur est constamment occupé et que par conséquent, il n'y a aucun temps libre dans l'ordonnancement. La tâche τ_{3m+1} possède une laxité nulle par conséquent, chacune de ses instances doit être exécutée dès son arrivée dans le système. La tâche τ_{3m+1} laisse libre le processeur pendant seulement pour cha-

cune de ses instances, un unique bloc de durée B (la longueur de la suspension) pour l'exécution des autres tâches. La figure 9 présente un ordonnancement réussit de la configuration générée par le problème de 3-partition. Les bornes du $k^{\text{ième}}$ bloc de temps-creux sont calculables et sont les suivantes :

$$\left[2(k-1)B + \left\lceil\frac{B}{2}\right\rceil ; 2kB - \left\lfloor\frac{B}{2}\right\rfloor\right) \qquad \forall k \geq 1 \tag{4}$$

Nous allons aborder notre démonstration en deux parties.

1. **Si une solution existe pour le problème de 3-partition alors la configuration de tâches générée admet un ordonnancement faisable :**
   D'après l'énoncé du problème et si nous nous intéressons à A_1, il possède exactement trois éléments (notés s_j, s_k et s_l) dont la somme $(s_j + s_k + s_l)$ donne B. Pendant le temps creux correspondant à la suspension de la première instance de τ_{3m+1}, nous ordonnançons les premières sous-tâches ($C_{i,1}$) des tâches (τ_j, τ_k et τ_l) correspondant aux éléments de A_1. Les secondes sous-tâches sont ordonnancées pendant le temps creux créés par la $(m+1)^{\text{ième}}$ instance de τ_{3m+1}. Ainsi le temps écoulé entre les deux sous-tâches de chaque tâche générée à partir des éléments de A_1 est égal à $(2m-1)B$, soit la durée de suspension requise d'après la formule 4. Nous recommençons le même processus en prenant les éléments de A_2 et en réitérant l'opération en ordonnançant les premières sous-tâches des tâches générées par les éléments de A_2 pendant le temps creux généré par la seconde instance de τ_{3m+1} et les secondes sous-tâches pendant le temps creux obtenu par l'exécution de la $(m+2)^{\text{ième}}$ instance de τ_{3m+1}. Puis, il suffit de continuer les mêmes opérations en ordonnancant les tâches correspondant aux éléments A_i pendant les $i^{\text{ième}}$ et $(m+i)^{\text{ième}}$ temps creux générés par les instances de τ_{3m+1}. Ainsi, nous obtenons un ordonnancement faisable pour notre configuration de tâches.
2. **Si la configuration de tâches générées admet un ordonnancement faisable alors il existe une solution pour le problème de 3-partition :** Nous allons d'abord supposer qu'il existe un ordonnancement non-préemptif pour les tâches générées puis nous traiterons le cas général.
   – Il existe un ordonnancement non-préemptif pour les tâches générées à partir du problème de 3-partition :
     Comme le facteur d'utilisation est de 1 et que l'ordonnancement est faisable, dans chaque temps creux dû à l'exécution d'un job de τ_{3m+1} trois tâches exécutent soit leur première sous-tâche, soit leur seconde (elles sont trois car $B/4 < C_{i,j} < B/2, i = 1..3m, j = 1,2$). Les trois tâches dont la première sous-tâche s'est exécutée pendant le temps creux dû à l'exécution du premier job de la tâche τ_{3m+1}, ne peuvent pas exécuter leur seconde sous-tâche pendant les $m-1$ blocs de temps creux suivants générés par τ_{3m+1} sinon la durée de suspension n'est pas respectée. Par conséquent, la seconde sous-tâche ne peut être ordonnancée qu'à partir du $(m+1)^{\text{ième}}$ bloc de temps creux. Pendant ce temps creux, toutes les premières sous-tâches de toutes les tâches τ_i (pour $i \in \{1 \ldots 3m\}$) ont été exécutées et toutes ces tâches sont en

période de suspension sauf celles qui ont exécutées leur première sous-tâche durant le premier bloc de temps creux. Par conséquent, comme le facteur d'utilisation est de 1, pendant le $(m+1)^{\text{ième}}$ bloc de temps creux, les seules tâches ordonnancées sont celles dont la première sous-tâche a été exécutée durant le premier bloc de temps creux. En répétant cette logique pour les autres tâches, les tâches dont la première sous-tâche a été exécutée pendant le $i^{\text{ième}}$ $(i \leq m)$ bloc de temps creux a exécuté sa seconde sous-tâche pendant le $(m+i)^{\text{ième}}$ bloc de temps creux. Ainsi, si nous nommons A_i, l'ensemble des s_k ayant servi à la conception des tâches dont la première sous-tâche a été exécutée pendant le $i^{\text{ième}}$ temps creux généré par τ_{3m+1}. Alors nous obtenons une partition de A en m sous-ensembles respectant les contraintes données par le problème de 3-partition.

– Cas général :
  Nous allons en fait démontrer qu'une sous-tâche ne peut s'exécuter que dans un unique bloc de temps creux généré par τ_{3m+1}. En effet, utilisons un raisonnement par l'absurde et supposons qu'une sous-tâche ait commencé son exécution dans un bloc k $(k < m)$ et qu'elle finisse son exécution dans le bloc $k+1$. Les autres sous-tâches sont supposées s'être exécutées dans un unique bloc. Comme il ne peut y avoir plus de trois sous-tâches par bloc, il n'y a que deux sous-tâches dont l'exécution s'est terminée dans le bloc k. Par respect des suspensions des tâches, il n'y a seulement que deux sous-tâches complétées dans le bloc $m+k$. Comme une conséquence, il y a un temps creux dans le bloc $k+m$ ce qui contredit le fait que le facteur d'utilisation est égal à 1. Ce qui finit cette démonstration.

□

6.2 Démonstration du théorème 2

Théorème 2 : *S'il existe un algorithme d'ordonnancement universel pour les configurations de tâches où chaque tâche se suspend au plus une fois alors $\mathcal{P} = \mathcal{NP}$.*

Démonstration :
Nous allons utiliser une technique de preuve classique, telle que celle présentée dans [5]. Plus précisément, nous allons supposer l'existence d'un tel algorithme. Et nous allons démontrer que si cet algorithme choisit en un temps polynomial (polynomial en la longueur de la configuration) la prochaine tâche exécutée, alors $\mathcal{P} = \mathcal{NP}$. Parce que dans ce cas là, nous aurons trouvé un algorithme pseudo-polynomial qui résout le problème de 3-partition.
Nous supposons qu'il existe un algorithme d'ordonnancement pour les configurations de tâches périodiques, où chaque tâche peut se suspendre au plus une fois, et sur des systèmes monoprocesseurs. Cet algorithme est noté A. En utilisant la même technique que celle détaillée dans la preuve du théorème 1, nous définissons à partir d'une instance du problème de 3-partition un ensemble de $3m+1$ tâches, noté I. L'ensemble des tâches composant la configuration nouvellement générée I sont à départ simultané. En utilisant le même raisonnement que dans la démonstration du théorème 1,

pour vérifier que tous les instances respectent leurs échéances, nous pouvons limiter notre période d'étude à l'intervalle $[0, 4Bm]$. Par conséquent nous ordonnançons la configuration I avec l'algorithme A, puis nous vérifions que chaque tâche respecte son échéance. L'hyperpériode de l'ordonnancement est, comme nous l'avons calculé précédemment, est égale à $4Bm$. De plus, d'après nos hypothèses, A est un algorithme polynomial. Par conséquent, vérifier que toutes les échéances sont respectées se fait en un temps au plus pseudo-polynomial (i.e, c'est clairement réalisé en temps proportionnel à Bm). En utilisant le même raisonnement que lors de la démonstration du théorème 1, la configuration I n'est ordonnançable par l'algorithme A que si et seulement s'il existe une partition des tâches $\tau_1, \ldots, \tau_{3m}$ en m ensembles disjoints $A_1, A_2, \ldots, A_m$ tels que pour chaque ensemble A_i ($i \in \{1, \ldots, m\}$), nous avons $\sum_{\tau_j \in A_i} C_{j,1} = B$. Ainsi, la solution obtenue par l'algorithme A donne une solution pour résoudre le problème de 3-partition. Pour déterminer cette solution, nous transformons l'instance du problème de 3-partition en simplement construisant l'ensemble de tâches avec au plus une suspension par tâche comme dans la preuve du théorème 1 et alors en présentant cette configuration de tâches à la procédure de décision basée sur l'algorithme A.

Par conséquent, nous avons trouvé un algorithme pseudo-polynomial pour résoudre le problème de 3-partition. Mais le problème de 3-partition $\mathcal{NP}$-complet au sens fort. Donc, et pour conclure, si l'algorithme A existe alors $\mathcal{P} = \mathcal{NP}$. Un tel algorithme ne peut donc par conséquent pas exister.

□