Ecole d'été Temps Réel 2003

Panorama de l'ordonnancement temps réel monoprocesseur

Annie Choquet-Geniet

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

RÉSUMÉ Nous présentons dans un premier temps les principaux algorithmes d'ordonnancement en ligne des systèmes de tâches périodiques à contraintes temporelles strictes. Nous considérons les algorithmes à priorités statiques, puis les algorithmes à priorités dynamiques. Nous en étudions le principe, la puissance d'ordonnancement et les critères éventuels d'ordonnançabilité. Nous introduisons ensuite les tâches apériodiques, et nous montrons comment elles peuvent être prises en compte par des techniques de serveurs.

MOTS-CLÉS : ordonnancement - tâches périodiques - tâches apériodiques - algorithme optimal - serveur

---

1 Introduction

Les applications temps réel ont pour vocation de contrôler des procédés physiques tels que le système d'allumage d'un véhicule, un avion en vol, un robot, une centrale nucléaire.... Afin de garantir la sûreté de fonctionnement du procédé contrôlé, ces applications doivent non seulement être algorithmiquement correctes, mais elles doivent également respecter les contraintes temporelles induites par la dynamique du procédé. Il faut garantir que les commandes envoyées s'effectueront à temps. Ceci pose les bases du problème de la validation et de l'ordonnancement temps réel. Il s'agit d'une part de répondre à la question «L'application peut-elle respecter toutes ses contraintes temporelles ?» et d'autre part de déterminer comment elle peut le faire, c'est à dire déterminer une stratégie pertinente d'allocation du processeur.

Une application temps réel classique consiste en un mélange de tâches périodiques et de tâches apériodiques. Les tâches périodiques sont les tâches de contrôle : tâches de relevé de température dans une centrale nucléaire, calcul de la trajectoire d'un robot, traitement d'informations véhiculées par une ligne de communication synchrone... Elle s'exécutent de manière régulière et sont soumises à des contraintes temporelles strictes. Les tâches apériodiques s'exécutent en réponse à des événements aléatoires tels que l'intervention d'un opérateur, la survenue d'une alarme, la détection d'une erreur lors de l'exécution d'une tâche périodique... Elles ne sont pas nécessairement soumises à des contraintes strictes. Dans ce cas, on cherche simplement à minimiser leur temps de réponse.

Notre objectif est dans un premier temps de présenter les principaux résultats concernant l'ordonnancement des tâches périodiques dans un contexte **monoprocesseur** pour des tâches **indépendantes** (elles n'interagissent pas). Puis nous présentons quelques techniques d'ordonnancement conjoint de tâches périodiques et de tâches apériodiques : technique du traitement en arrière plan et techniques de serveurs. Nous étudions le serveur à scrutation, le serveur ajournable, le serveur à échange de priorité et le serveur

sporadique.

2 Contexte général

2.1 Les applications temps réel

Une application temps réel est classiquement modélisée par un ensemble fini de **tâches** $\{\tau_1, \tau_2, \ldots, \tau_n\}$ obtenu à partir d'une étude (non formelle) du cahier des charges de l'application. Ces tâches peuvent être [27, 25] - **périodiques** : elles sont réactivées à intervalle régulier, - **apériodiques** : leurs activations ont lieu à des moments aléatoires ou - **sporadiques** : leurs activations ont lieu à des moments aléatoires, mais la durée séparant deux activations successives est bornée, et elles ont des échéances strictes.

2.2 Le modèle temporel de tâches

Chaque tâche périodique possède - une description fonctionnelle à l'instar de n'importe quelle tâche, et - une description temporelle, spécifique des tâches temps réel. Nous utilisons le modèle temporel de [19] : la tâche τ_i est caractérisée par les quatre paramètres temporels (r_i, C_i, R_i, P_i) (voir figure 1) où :

- r_i est la **date de première activation**
- C_i est la **pire durée d'exécution** (voir la contribution d'Isabelle Puaut sur ce thème)
- R_i est le **délai critique**
- P_i est la **période**

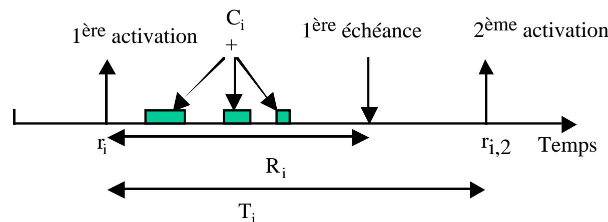

FIG. 1 – Paramètres temporels d'une tâche temps réel : $\tau_i = (r_i, C_i, R_i, P_i)$.

Lorsque les dates d'activation sont toutes égales, on parle de système **à départs simultanés**, dans le cas contraire, on parle de systèmes **à départs différés**. La $k^{\text{ème}}$ instance de la tâche τ_i est activée à l'instant $r_{i,k} = r_i + (k-1)P_i$. Les délais critiques sont supposés stricts. Ils définissent la durée maximale tolérée entre l'activation d'une instance de τ_i et sa terminaison. La $k^{\text{ème}}$ instance de τ_i a **l'échéance absolue** $d_{i,k} = r_i + (k-1)P_i + R_i$, qui est la date à laquelle elle doit avoir terminé. Nous supposons de plus que $R_i \leq P_i$: une instance doit être terminée avant que la suivante ne soit activée. Enfin, si $R_i = P_i$, la tâche est dite **à échéances sur requêtes**.
Dans cette présentation, nous ne considérons que des systèmes **déterministes** (tous les paramètres temporels sont connus a priori) car ce sont les seuls pour lesquels on puisse garantir le respect des contraintes temporelles.

Afin de caractériser l'activité du processeur, nous définissons :

- la charge processeur d'une tâche τ_i qui correspond au taux d'activité du processeur dédiée à l'exécution des instances successives de la tâche : $U_i = \frac{C_i}{P_i}$
- la **charge processeur de l'application** ou **taux d'utilisation** $U = \displaystyle\sum_{i=1}^{n} \frac{C_i}{P_i}$ qui représente la proportion de l'activité du processeur dédiée à l'exécution de l'application.

Une tâche apériodique est caractérisée par une date d'arrivée r, non connue a priori et une durée C, supposée connue à l'arrivée de la tâche. Elle peut également posséder un délai critique R. Dans ce cas, son échéance absolue est égale à r + R.

2.3 Le problème de l'ordonnancement

2.3.1 Ordonnancement de tâches périodiques

Le problème de l'**ordonnancement** consiste à définir une politique d'attribution du processeur qui assure qu'aucune **faute temporelle** ne sera commise, c'est-à-dire qu'aucune instance de tâche ne terminera après son échéance.
Le problème de la **validation** de l'application consiste à déterminer si il existe une solution au problème de l'ordonnancement.
Deux approches peuvent être envisagées :
- l'**ordonnancement en ligne** : on implémente un algorithme au niveau de l'ordonnanceur, les décisions d'ordonnancement étant prises au cours de la vie de l'application.
- l'**ordonnancement hors ligne** : une séquence valide est calculée avant l'exécution effective de l'application. Elle est ensuite chargée dans une table utilisée par le séquenceur au cours de la vie de l'application.

L'utilisation d'un ordonnancement hors ligne permet d'éviter les surcharges processeur liées à l'exécution d'un algorithme d'ordonnancement et présente davantage de garanties en cas d'utilisation de ressources critiques. En contrepartie, un ordonnancement en ligne est plus souple, en particulier en cas de reconfiguration de l'application, ou en cas de prise en compte de tâches sporadiques. Nous ne considérerons ici que les stratégies d'ordonnancement en ligne. Pour quelques exemples de stratégies hors ligne, le lecteur pourra consulter [8, 13, 29].

Une fois choisie la politique d'ordonnancement, il faut valider l'application, i.e. s'assurer qu'elle est bien correctement ordonnancée. Dans les cas favorables (soit que l'on dispose de conditions nécessaires et suffisantes d'ordonnançabilité, soit que l'on dispose de conditions suffisantes qui sont vérifiées) on dispose de critères analytiques calculables en temps polynômial, utilisant les paramètres temporels des tâches. Dans les autres cas, le problème de la validation d'une politique d'ordonnancement est NP-difficile [17] et la seule solution consiste en la simulation de l'application dans une fenêtre temporelle de taille bornée. Si les tâches sont à départs simultanés, la durée de simulation est égale au PPCM des périodes des tâches. Si les tâches sont à départs différés, une durée minimale de simulation peut être calculée [12, 7].

L'un des problèmes centraux lorsque l'on veut valider une application réside dans le choix de la stratégie d'ordonnancement à mettre en œuvre. Ce choix peut être simplifié pour certaines classes d'applications pour lesquelles il existe des algorithmes optimaux, qui sont ceux dont la puissance d'ordonnancement est la plus forte.

Soit $\mathcal{T}$ une classe d'applications. Un algorithme d'ordonnancement est dit **optimal** (éventuellement parmi une sous-famille d'ordonnancements) pour les applications de la classe $\mathcal{T}$ si et seulement si quelle que soit l'application de la classe, soit l'algorithme l'ordonnance de manière correcte, soit aucun autre algorithme (de la sous-famille) ne le pourra.

Notons cependant que dans le cas général le problème de l'ordonnancement est NP-difficile [17, 20, 3]. Notons enfin une première condition nécessaire d'ordonnançabilité [5] :

Proposition 1. *Si une application temps réel stricte est ordonnançable, alors on a* $U \leq 1$.

2.3.2 Prise en compte de tâches non périodiques

L'**ordonnancement conjoint** consiste à définir une stratégie d'ordonnancement qui :
- garantit le respect de toutes les échéances de toutes les tâches périodiques,
- assure de bons temps de réponse pour les tâches apériodiques,

- propose une routine de garantie pour les tâches sporadiques : une telle routine est un test d'acceptation, qui permet de vérifier, lors de son arrivée dans le système, qu'une requête sporadique pourra bien être exécutée dans les temps, et ce sans mettre en péril les tâches périodiques ou les tâches sporadiques déjà acceptées. Dans le cas contraire, la requête est rejetée.

Notons que dans le cas de tâches apériodiques, on cherche à obtenir de bons temps de réponse sans pour autant se fixer d'objectif de mimisation des temps de réponse. Ceci provient du fait qu'un tel objectif ne peut être atteint comme le stipulent les deux résultats suivants :

Theorème 2. *[28] Il n'existe pas d'algorithme d'ordonnancement conjoint permettant à la fois de minimiser le temps de réponse de chacune des requêtes apériodiques et d'ordonnancer fiablement les tâches périodiques selon une statégie à priorités fixes.*

Theorème 3. *[28] Il n'existe pas d'algorithme d'ordonnancement conjoint en ligne permettant à la fois de minimiser le temps moyen de réponse des requêtes apériodiques et d'ordonnancer fiablement les tâches périodiques selon une statégie à priorités fixes.*

3 Ordonnancement périodique

Les principales stratégies d'ordonnancement de tâches périodiques reposent sur la notion de priorités : chaque tâche possède une priorité qui peut être statique (constante tout au long de la vie de l'application) ou dynamique (qui évolue au cours de la vie de l'application). Ces priorités peuvent être attribuées par le concepteur de l'application en fonction de critères non formels, ou bien être dérivés des paramètres temporels. Le processeur est à tout instant attribué au processus de plus forte priorité. Ces stratégies sont le plus souvent **préemptives** : une tâche peut être interrompue en cours d'exécution par une tâche plus prioritaire et sont **conservatives** (ou bien fonctionnent au plus tôt) : le processeur ne peut rester inactif que si aucune instance n'est prête à être exécutée.

3.1 Algorithmes à priorités statiques

Les priorités statiques sont des fonctions constantes du temps. Elles sont attribuées aux tâches avant le début de l'exécution, une fois pour toutes, et demeurent inchangées au cours du temps. Il n'y aura donc pas de surcoût engendré par le recalcul des priorités, comme ce sera le cas pour les algorithmes à priorités dynamiques. De ce fait, l'implémentation des algorithmes à priorités fixes est relativement simple. Nous présentons ici deux algorithmes, utilisant respectivement les périodes et les délais critiques pour le calcul des priorités. On poura consulter [1] pour un panorama des algorithmes à priorités fixes.

3.1.1 Rate Monotonic

Le premier algorithme considéré est l'algorithme **RM : Rate Monotonic** introduit par Liu et Layland [19]. Les priorités sont inversement proportionnelles aux périodes des tâches : $\mathbf{P_i} < \mathbf{P_j} \Rightarrow \mathbf{Prio}(\tau_i) > \mathbf{Prio}(\tau_j)$. En cas de conflit, lorsque plusieurs tâches ont la même période, le choix est arbitraire (FIFO peut être utilisé par exemple). Si l'on considère uniquement des sytèmes de tâches à échéances sur requêtes, alors l'attibution des priorités selon les périodes est la meilleure attribution possible pour obtenir une séquence valide :

Proposition 4. *L'algorithme RM est optimal parmi les algorihtmes à priorités fixes pour les applications constituées de tâches indépendantes à échéances sur requêtes et à départs simultanés.*

Une propriété intéressante de l'algorithme RM est qu'il vérifie le théorème de l'**instant critique** : une tâche aura le pire temps de réponse si elle est activée en même temps que toutes les tâches plus prioritaires qu'elle. Il s'ensuit que, lorsque les tâches sont à départs simultanés, si aucune faute ne se

produit sur la fenêtre temporelle [0, Max(P_i)], alors il n'y aura aucune faute temporelle, quelles que soient les dates de réveil des tâches. Liu et Layland ont utilisé ce résultat pour dériver un critère analytique de validation d'une application ordonnancée par RM (condition suffisante).

Proposition 5. *Un système de n tâches indépendantes à échéances sur requêtes est ordonnançable par RM si* $U \leq n(2^{\frac{1}{n}} - 1)$.

Cette condition n'est qu'une condition suffisante, mais elle présente l'avantage d'être très simple à évaluer (en O(n)). On peut trouver dans [15], une condition nécessaire et suffisante mais qui est plus délicate à évaluer (de complexité pseudo-polynômiale en O($Max(P_i) * n^3$)).

Considérons le système constitué de trois tâches S1 = {τ_1 (2,2,8,8), τ_2 (1,4,12,12), τ_3 (0,4,24,24)}. Nous avons U = $\frac{2}{8} + \frac{4}{12} + \frac{4}{24}$ = 0,75. Or $3(2^{\frac{1}{3}} - 1) = 0,78$. D'après la propositon 5, le système est donc ordonnançable, et la séquence obtenue est donné par la figure 2

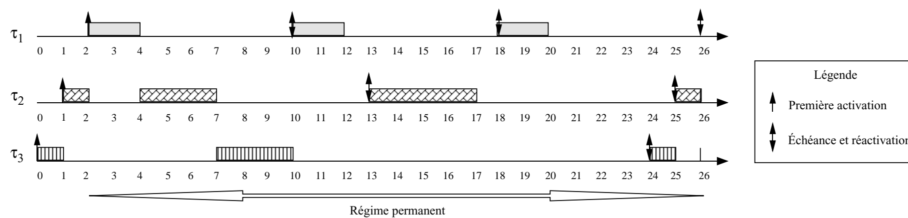

FIG. 2 – Séquence RM pour le système S1. La séquence entre les instants 2 et 26 définit le régime permanent du système.

Notons que si la durée de la tâche τ_1 est égale à 3 au lieu de 2, la charge du système est égale 0, 825. La condition de la proposition 5 n'est plus vérifiée, mais le système reste cependant ordonnançable.

3.1.2 Deadline Monotonic

Autant RM fonctionne bien si les tâches sont à échéances sur requêtes, autant ce n'est plus le cas si les délais critiques sont plus petits que les périodes. Afin de prendre en compte les délais critiques, tout en gardant le principe des piorités fixes, Leung et Whitehead ont proposé [17] l'algorithme DM : **deadline monotonic**, qui utilise des priorités fixes inversement proportionnelles aux délais critiques : $\mathbf{R_i} < \mathbf{R_j} \Rightarrow \mathbf{Prio}(\tau_\mathbf{i}) > \mathbf{Prio}(\tau_\mathbf{j})$. Les conflits sont résolus de manière arbitraire, par exemple par FIFO. Si les tâches sont à échéances sur requêtes, RM et DM coïncident. Dans le cas contraire, si l'on suppose que l'application est à départs simultanés, on obtient une stratégie optimale :

Proposition 6. *L'algorithme DM est optimal parmi les algorihtmes à priorités fixes pour les applications constituées de tâches indépendantes à départs simultanés.*

Leung et Whitehead ont également montré que DM vérifie le théorème de l'instant critique. On peut alors dériver une condition suffisante d'ordonnançabilite (proposition 7). On peut par ailleurs trouver dans [1, 5] une condition nécessaire et suffisante, mais comme dans le cas de RM, celle-ci est beaucoup plus coûteuse que la condition suffisante.

Proposition 7. *Un système de n tâches indépendantes est ordonnançable par DM si* $\sum_{i=1}^{n} \frac{C_i}{R_i} \leq n(2^{\frac{1}{n}} - 1)$.

Considérons les système de tâches S2 = $\{\tau_1(0,1,2,2), \tau_2(0,1,1,3)\}$. La figure 3 donne les assignations produites d'une part par l'algorithme RM et d'autre part par l'algorihtme DM.

Enfin, dans le cas des systèmes à départs différés, ni RM ni ED ne sont optimaux, ce qu'illustre l'exemple suivant. On considère le système constitué de deux tâches S3 = $\{\tau_1(2,3,3,4), \tau_2(0,3,4,8)\}$. La figure 4 donne les assignations produites par les ordonnancements à priorités fixes.

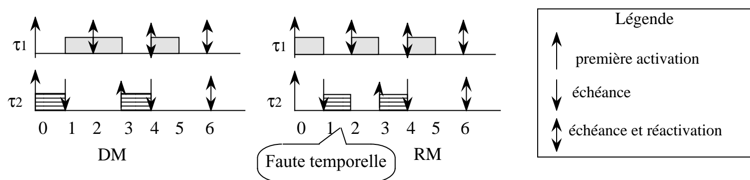

FIG. 3 – Comparaison de RM et DM. Lorsque l'on utilise RM pour ordonnancer S2, une faute temporelle est commise par la tâche τ_2 alors que toutes les échéances sont respectées dans le cas de DM

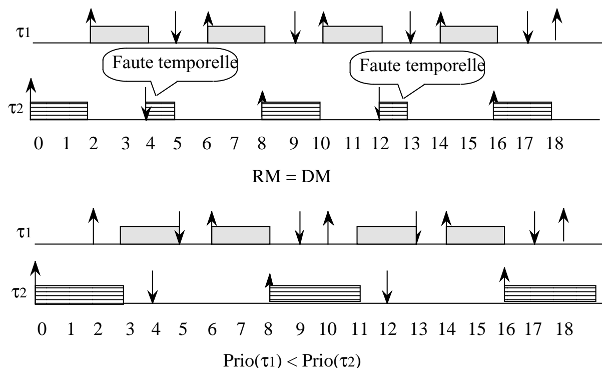

FIG. 4 – Non optimalité de RM et DM pour les systèmes des tâches à départs différés. Si on affecte les priorités selon RM ou DM, la tâche τ_2 commet des fautes temporelles, alors que si on affecte la priorité la plus forte à τ_2, on obtient une séquence valide.

3.2 Algorithmes à priorités dynamiques

Bien que plus difficiles à mettre en œuvre que les algorithmes à priorités fixes, les algorithmes à priorités dynamiques sont en général plus performants, en ce sens qu'ils ordonnancent davantage de systèmes. Les priorités sont cette fois des fonctions du temps, et donc évoluent au cours de la vie de l'application. Ceci induit une surcharge processeur due aux recalculs successifs des priorités. Nous présentons ci-après deux algorithmes, l'un fondé sur les échéances et l'autre sur la laxité.

3.2.1 Earliest Deadline

L'algorithme ED : **Earliest Deadline**, étudié initialement dans [19], est certainement l'algorithme le plus populaire. Le principe consiste à ordonnancer à tout instant la tâche dont l'échéance est la plus proche : $\mathbf{d_i} < \mathbf{d_j} \Rightarrow \mathbf{Prio}(\tau_\mathbf{i}) > \mathbf{Prio}(\tau_\mathbf{j})$. Pour une présentation approfondie de ED, on pourra se reporter à [27].

Soit le système de tâches S4 = $\{\tau_1(0,3,5,5), \tau_2(0,1,3,3)\}$. L'ordonnancement par ED de ce système est présenté sur la figure 5.

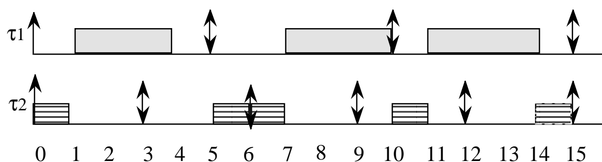

FIG. 5 – Ordonnancement par ED du système S4

C'est un algorithme plus puissant que RM ou DM, comme l'illustre le résultat suivant (proposition 8), obtenu tout d'abord par [19] pour les systèmes de tâches à départs simultanés et à échéances sur requêtes, puis étendu par [11] aux systèmes de tâches à départs différés et à délais critiques inférieurs ou égaux aux périodes.

Proposition 8. *L'algorithme ED est optimal pour les systèmes de tâches indépendantes.*

L'un des principaux atouts de ED est l'existence d'une condition nécessaire et suffisante d'ordonnançabilité, très simple à mettre en œuvre (proposition 9). Cette condition a été établie par Liu et Layland pour les systèmes de tâches à départs simultanés, puis ensuite par Coffman [9] pour les tâches à départs différés.

Proposition 9. *Un système de tâches à départs simultanés ou différés et à échéances sur requêtes est ordonnançable par ED si et seulement si sa charge processeur U est inférieure ou égale à 1.*

Et comme ED est optimal, ceci fournit même un critère d'ordonnançabilité de l'application. Ce résultat est en particulier valide pour les systèmes à départs différés, ce qui illustre bien la supériorité de ED sur RM ou DM.

Considérons le système de tâches S5 = $\{\tau_1(4,1,3,3), \tau_2(0,3,5,5)\}$. Nous avons $U = \frac{1}{3} + \frac{3}{5} = \frac{14}{15}$. Le système est donc ordonnançable par ED, et la séquence obtenue est donnée figure 6.

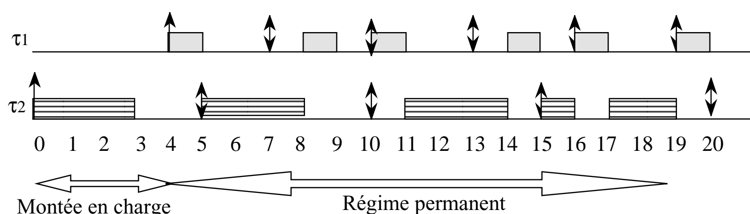

FIG. 6 – Ordonnancement par ED de tâches à départs différés

Par contre, là encore, si l'on considère des délais critiques plus petits que les périodes, le problème de l'ordonnancement est à nouveau NP-difficile [18, 3]. De par l'optimalité de ED, on peut obtenir le résultat par la simulation, mais la durée de simulation est linéaire en le PPCM des périodes [12, 7] qui n'est pas borné polynômialement, ce qui explique la complexité de la validation.

3.2.2 Least Laxity

L'algorithme LL : **least laxity**, présenté par Mok et Dertouzos [21], prend comme critère d'ordonnancement la marge de manœuvre dont dispose une tâche. Le processeur est attribué à la tâche dont la date de départ au plus tard est la plus proche. Pour cela, on définit la laxité de la tâche, qui est une fonction du temps : **Laxité(t,** τ_i**)** est égale à la différence entre l'échéance de l'instance en cours et la charge de travail restante (voir figure 7). La priorité maximale est attribuée à la tâche de plus faible laxité (et Fifo peut être utilisé en cas d'égalité).

L'algorithme LL présente les mêmes propriétés d'optimalité que ED, et le critère d'ordonnançabilité établi pour les systèmes de tâches à échéances sur requêtes fonctionne. Mais il engendre un nombre supérieur de préemptions en cours d'exécution comme l'illustre la figure 8, ce qui explique qu'il soit peu utilisé dans le cas monoprocesseur.

Considérons un système de deux tâches S6 = $\{\tau_1(0,4,8,8), \tau_2(0,3,6,6)\}$ ordonnancées d'une part par ED et d'autre part par LL. Le nombre de prémptions est le plus faible pour ED.

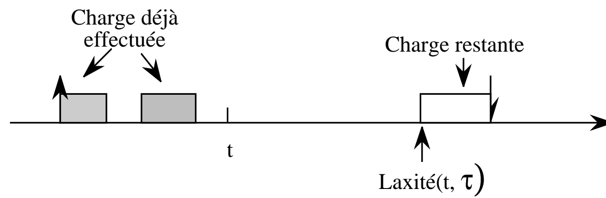

FIG. 7 – La laxité d'une tâche τ

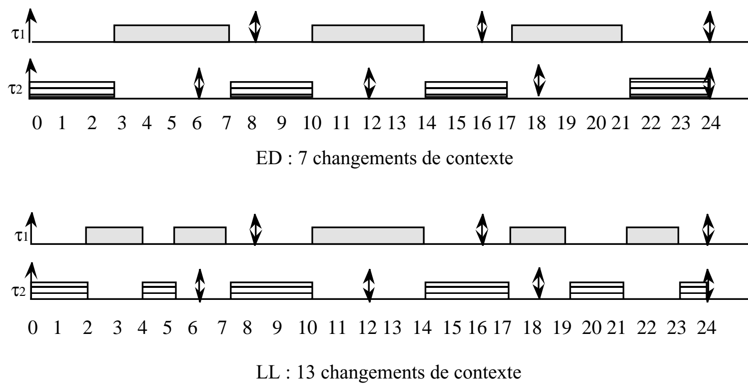

FIG. 8 – Comparaison de ED eet LL

4 Ordonnancement conjoint

Nous supposons maintenant que les applications considérées comportent :

- des tâches périodiques à départs simultanés et à échéances sur requêtes.
- des tâches apériodiques.

Nous supposerons que les tâches périodiques sont ordonnancées par RM et que les requêtes apériodiques sont servies en ordre FIFO.

Les stratégies de prise en compte des tâches périodiques se répartissent en deux catégories :

- Les techniques de vol de temps creux, qui consistent à tirer profit de l'inactivité du processeur. La technique la plus simple est celle de l'ordonnancement en arrière plan : les tâches apériodiques sont placées dans les temps creux laissés par les tâches périodiques. Elle présente le double avantage d'être très simple à mettre en œuvre et de ne pas mettre en péril l'ordonnançabilité des tâches périodiques. De plus, l'algorithme d'ordonnancement des tâches périodiques peut être quelconque. Mais en contre partie, dès que la charge périodique augmente, les performances de cette stratégie deviennent très médiocres, i.e. les temps de réponse des tâches apériodiques augmentent. On trouve aussi dans cette catégorie d'autres algorithmes beaucoup plus complexes, dont le principe consiste à modifier l'ordonnancement des tâches périodiques lorsque survient une tâche apériodique. Les taches périodiques sont ordonnancées au plus tôt tant qu'il n'y a pas de requêtes apériodiques, afin de préserver les temps creux, et au contraire, quand survient une requête, on repousse les tâches périodiques (tout en garantissant le respect des échéances) de manière à traiter au plus vite la tâche apériodique. Citons dans cette catégorie l'algorithme du «slack stealing» [14, 28] et le serveur EDL [24, 23, 26].
- Les techniques de **serveur** qui consistent à définir une tâche serveur, périodique, dédiée exclusivement au service des requêtes apériodiques. Un serveur est caractérisé par sa période T_s et par sa capacité C_s, qui définit la quantité maximale de charge apériodique qu'il peut exécuter chaque

période. Dans la suite de cette présentation, nous donnons quelques exemples de serveurs

Nous supposons qu'il y a n tâches périodiques, de charge totale U_p et une tâche serveur de charge U_s. L'ensemble de ces (n + 1) tâches est ordonnancé par RM. Par ailleurs, nous supposons que les requêtes apériodiques sont traitées dans l'ordre de leur arrivée, c'est à dire que la file d'attente des tâches apériodiques est gérée en mode FIFO.
Ce qui différencie les différents serveurs entre eux est la manière dont ils se comportent lorsqu'ils sont activés alors qu'il n'y a aucune requête apériodique en attente. Pour chacun d'eux, nous décrivons le principe de fonctionnement, puis nous donnons quelques résultats concernant l'analyse d'ordonnançabilité : il s'agit de déterminer, pour une charge de serveur donnée U_s la capacité périodique U_p maximale pour laquelle le respect des échéances des tâches périodiques sera garanti. Les critères dégagés sont les conditions suffisantes, et s'appuient sur le critère d'ordonnançabilité pour RM (proposition 5).

4.1 Le serveur à scrutation

4.1.1 Principe de fonctionnement

Le serveur à scrutation, lorsqu'il est activé, exécute les requêtes apériodiques jusqu'à épuisement de sa capacité. S'il n'y a pas de requête apériodique en attente, sa capacité restante devient nulle : le serveur se suspend jusqu'à sa prochaine activation, où sa capacité sera restaurée(voir figure 9). Si une tâche apériodique survient juste après que le serveur s'est suspendu, elle devra attendre la prochaine réactivation pour être servie, donc éventuellement attendre une période complète. Ceci explique les performances médiocres de ce serveur en terme de temps de réponse.

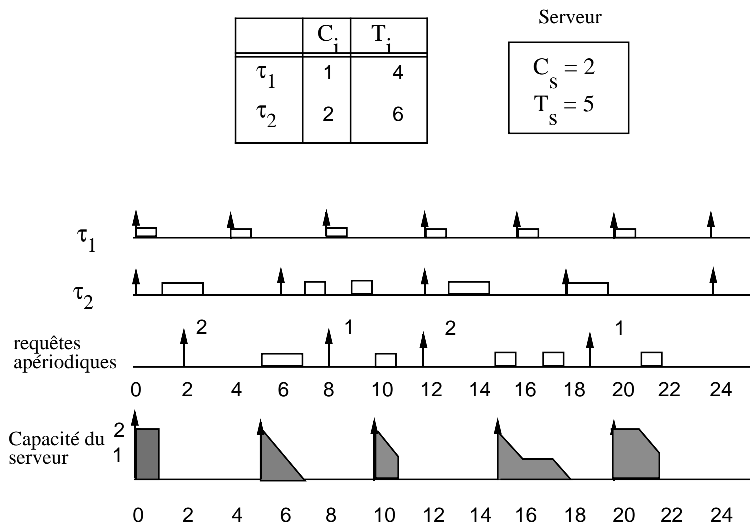

FIG. 9 – Serveur à scrutation : la tâche serveur est de priorité intermédiaire. A l'instant 1, elle est élue, et comme il n'y a pas de requête apériodique en attente, elle se suspend. A l'instant 2 survient une requête apériodique, mais comme la tâche serveur s'est supendue, elle doit attendre la prochaine instance du serveur. A l'instant 5, le serveur est réactivé et consomme sa capacité en entier. A l'instant 11, le serveur a consommé la moitié de sa capacité et il se suspend.

4.1.2 Analyse d'ordonnançabilité

Le pire cas en ce qui concerne l'ordonnancement des tâches périodiques se produit lorsqu'il y a en permanence des tâches apériodiques en attente, donc lorsque le serveur se comporte comme une tâche périodique. Par suite, le respect des échéances des tâches périodiques sera garanti (condition suffisante) si $U_p + U_s \leq (n+1)(2^{\frac{1}{n+1}} - 1)$.

4.2 Serveur ajournable

4.2.1 Principe de fonctionnement

Ce serveur, introduit par [16], permet d'obtenir de bien meilleures performances que le serveur à scrutation, car il préserve sa capacité tout au long de sa période tant qu'elle n'a pas été totalement utilisée. De plus, afin d'améliorer encore les temps de réponse, la priorité la plus forte est attribuée à la tâche serveur. Ainsi, dès qu'une tâche apériodique survient, si le serveur n'a pas encore épuisé sa capacité, il la traitera sans délai (voir figure 10).

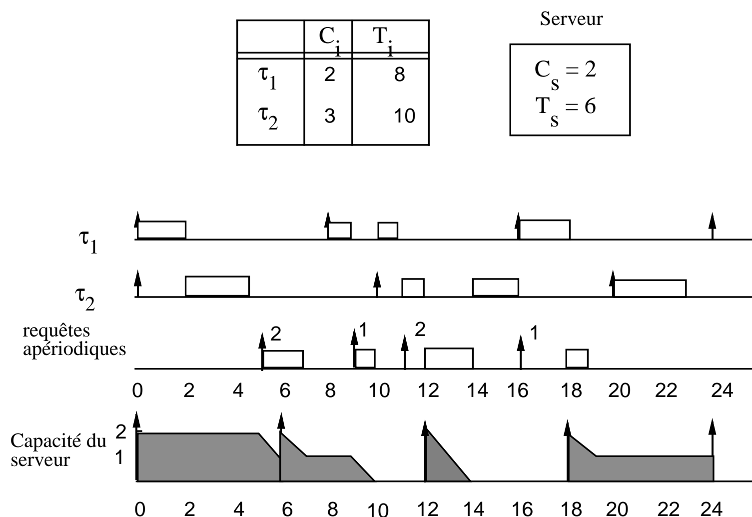

FIG. 10 – Serveur ajournable de priorité maximale. Lors de sa première activation, le serveur laisse la main aux tâches périodiques car il n'y a pas de requête apériodique en attente. Quand survient la première requête apériodique, à l'instant 5, le serveur peut alors la traiter, puisqu'il a préservé sa capacité. De même, quand survient le seconde requête à l'instant 9, le serveur n'avait pas épuisé sa capacité et il peut donc la prendre en charge.

4.2.2 Analyse d'ordonnançabilité

En ce qui concerne l'ordonnancement des tâches périodiques, ce serveur a des effets beaucoup plus négatifs que le serveur à scrutation : celui-ci pouvait être assimilé à une tâche périodique dans l'analyse d'ordonnançabilité, ce qui n'est plus le cas pour le serveur ajournable. Un système peut être déclaré ordonnançable si l'on assimile le serveur ajournable à une tâche périodique alors que des dépassement d'échéances pourront se produire si on utilise le serveur ajournable. Il s'ensuit que la charge périodique maximale pour laquelle on pourra garantir un ordonnancement correct des tâches périodiques est plus faible dans le cas du serveur ajournable que dans le cas du serveur à scrutation.

Proposition 10. *Si* $U_p \leq n[(\frac{U_S+2}{2U_s+1})^{\frac{1}{n}} - 1]$ *alors l'ordonnancement des tâches périodiques par RM sera valide.*

Cette borne supérieure admet comme limite $\mathrm{Ln}(\frac{U_s+1}{2U_S+2})$. Donc les tâches périodiques seront ordonnançables si $U_p \leq Ln(\frac{U_s+2}{2U_S+1})$.

4.3 Serveur à échange de priorités

4.3.1 Principe de fonctionnement

Ce serveur, introduit par [16] diffère du précédent en ce que, s'il conserve sa capacité si elle n'est pas utilisée, celle-ci perd de sa priorité au fur et à mesure qu'elle est ajournée. Le principe est donc le suivant :

- on utilise un serveur de priorité forte,
- à chaque activation du serveur, sa capacité intégrale lui est attribuée :
  - si une tâche apériodique est en attente, elle est immédiatement traitée,
  - sinon, le serveur échange sa priorité avec la tâche prête de priorité maximale. Celle-ci s'exécute donc avec la priorité du serveur et la capacité courante du serveur s'accumule à la priorité de la tâche périodique. Il peut y avoir plusieurs échanges successifs, et, au bout de quelques périodes, on peut avoir accumulé de la capacité à différents niveaux de priorité,
- s'il y a des temps creux, alors la capacité accumulée décroit,
- en cas de conflit, i.e. s'il y a égalité de priorité entre une tâche périodique et de la capacité accumulée, en présence de tâches apériodiques, c'est la capacité qui l'emporte, ceci afin de privilégier le temps de réponse des tâches apériodiques.

Considérons l'exemple de la figure 11 : à l'instant 0, le serveur a la priorité maximale, mais il n'y a pas de requête apériodique en attente, la capacité du serveur s'échange donc avec celle de τ_1 qui s'exécute alors. A l'instant 4, la capacité du serveur s'échange à nouveau, cette fois avec celle de τ_2, qui démarre son exécution. A l'instant 5, le serveur est réactivé et échange une nouvelle fois sa capacité avec celle de τ_2. On a donc accumulé une capacité égale à 2 au niveau de priorité de τ_2. A l'instant 10, le serveur et τ_1 sont réactivés, et τ_2 est preémpté. Le serveur échange sa capacité avec τ_1 qui s'exécute. A $t = 12$ survient une requête apériodique. τ_1 est alors en conflit avec la capacité accumulée à son niveau de priorité, et est prémptée. La tâche apériodique s'exécute pendant une unité de temps, puis τ_1 reprend la main, car la capacité restante est au niveau de priorité de τ_2. A l'instant 15, le serveur est réactivé et termine l'exécution de la tâche apériodique à la priorité maximale. A l'instant 17, le niveau de priorité de τ_2 redevient actif, et comme il y a une requête apériodique en attente, la capacité est prioritaire. La tâche apériodique est traitée, donc il ne reste plus qu'une capacité égale à 1 à ce niveau de priorité. Ensuite, τ_2 termine, puis il y a un temps creux qui achève de consommer la capacité accumulée au niveau de τ_2.

4.3.2 Analyse d'ordonnançabilité

Le serveur à échange de priorité fournit des temps de réponse un peu moins bons que le serveur ajournable (ceci étant dû à la baisse de priorité de la capacité du serveur en l'absence de tâches apériodiques), mais par contre, la borne analytique est meilleure [16, 25].

Proposition 11. *Si* $U_p \leq n[(\frac{2}{U_s+1})^{\frac{1}{n}} - 1]$ *alors l'ordonnancement des tâches périodiques par RM sera valide.*

Cette borne supérieure admet comme limite $\mathrm{Ln}(\frac{2}{U_S+1})$. Donc les tâches périodiques seront ordonnançables si $U_p \leq Ln(\frac{2}{U_S+1})$. Pour une charge serveur U_s donnée, la charge périodique garantie est plus forte dans le cas du serveur à échange de proritès que dans le cas du serveur ajournable.

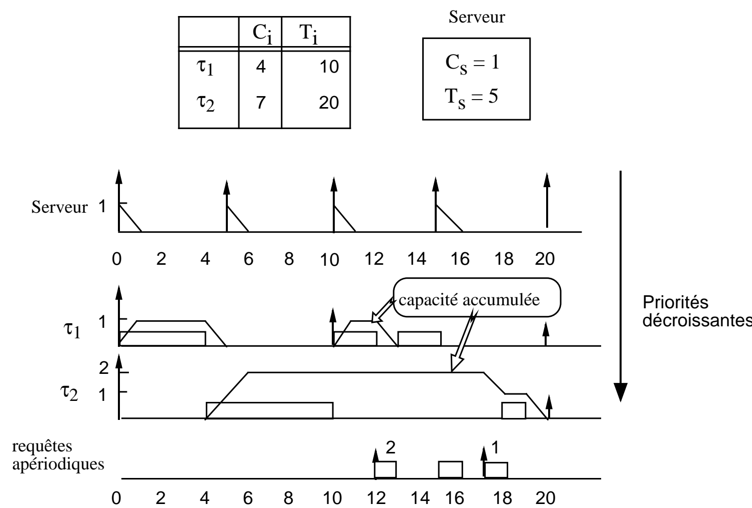

FIG. 11 – Serveur à échange de priorités.

4.4 Le serveur sporadique

4.4.1 Principe de fonctionnement

Le principe général de ce serveur, introduit dans [25] consiste à utiliser une tâche serveur avec préservation de capacité à son niveau de priorité. Par contre, les restaurations de capacités ne se produisent plus à instants fixes, mais dépendent de l'activité du serveur, et les quantités restaurées dépendent des quantités utilisées. Chaque fois que le niveau de priorité du serveur devient actif, une restauration de capacité est programmée une période plus tard. Une fois le serveur actif, on attend - soit que sa capacité restante soit entièrement consommée, - soit qu'il n'y ait plus de requêtes apériodiques à traiter. La quantité restaurée sera alors égale à la capacité utilisée dans cet intervalle de temps. La figure 12 montre un serveur sporadique de priorité maximale : le serveur devient actif à l'instant 1 : la prochaine restauration a lieu à l'instant 6, et le serveur redevient inactif à l'intant 2, donc le montant restauré est égal à 1. Puis le serveur est à nouveau actif à l'instant 4, une restauration a donc lieu à l'instant 9, elle est également de 1, et le serveur retrouve sa pleine capacité. Puis le serveur redevient actif à l'instant 10 et cette fois il utilise toute sa capacité. La capacité restaurée à l'instant 15 est donc égale à 2.

4.4.2 Analyse d'ordonnançabilité

Les critères d'ordonnançabilité sont les mêmes que pour le serveur à scrutation. En effet, [25] ont montré qu'à nouveau, le pire cas se présente lorsque le serveur se comporte comme une tâche périodique.

5 Conclusion

Nous avons présenté les principales stratégies d'ordonnancement en ligne pour des systèmes de tâches indépendantes, en environnement monoprocesseur. Si l'on désire prendre en compte des tâches inter-agissantes, il faut compléter l'étude : les contraintes de précédence peuvent être prise en compte par des réajustements des paramètres temporels des tâches (voir par exemple [4, 10]). Si l'on veut de plus permettre l'utilisation de ressources critiques, il faut adjoindre à l'algorithme d'ordonnancement (RM ou ED) un protocole de gestion de ressources, par exemple le protocole à priorité plafond [22, 6] ou le protocole à priorité de pile [2]. Nous avons ensuite considéré le cas des tâches apériodiques, et nous avons

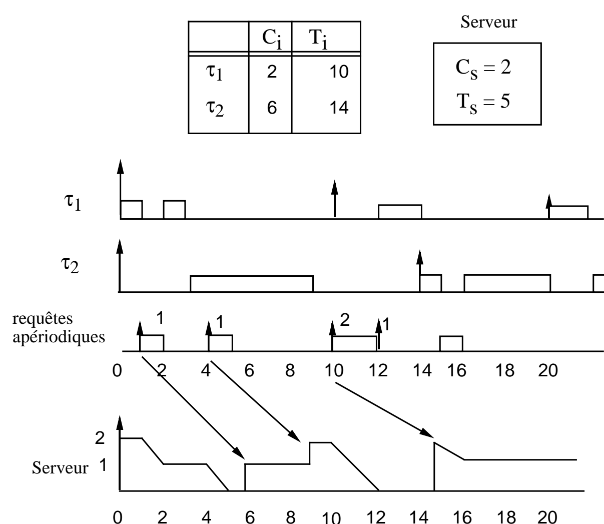

FIG. 12 – Serveur sporadique à priorité maximale.

présenté quelques serveurs à priorités fixes. Pour chacun d'eux, nous avons décrit le principe et donné quelques éléments d'analyse d'ordonnançabilité. Notons pour finir qu'il existe des serveurs à priorités dynamiques, en particulier, il existe des versions du serveur à échange de priorités et du serveur sporadique qui permettent d'utiliser EDF pour ordonnancer les tâches périodiques (on pourra consulter [27] pour un panorama de ces serveurs).

Références

[1] N.C. Audsley, A. Burns, R.I. David, K.W. Tindell, and A.J. Welling. Fixed priority preemptive scheduling : an historical perspective. *The journal of Real-Time Systems*, 8 :173–198, 1995.

[2] T.P. Baker. Stack-based scheduling of real-time processes. *the Journal of Real-Time Systems*, 3 :67–99, 1991.

[3] S.K. Baruah, L.E. Rosier, and R.R. Howell. Algorithms and complexity concernig the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, pages 301–324, 1990.

[4] J. Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In E. Gelenbe and H. Bellner, editors, *Modelling and performance evaluation o colputer systems*, pages 57–65. North-Holland, 1976.

[5] G.C. Buttazzo. *Hard Real-Time Computing Systems.* Kluwer Academic Publishers, 1997.

[6] M. Chen and K. Lin. Dynamic priority ceilings : a concurrency protocol for real-time systems. *Real-Time Systems*, 2(24) :325–346, 1990.

[7] A. Choquet-Geniet and E. Grolleau. Minimal schedulability interval for real time systems of periodic tasks with offsets. *Theoretical of Computer Science*, to appear.

[8] A. Choquet-Geniet, E. Grolleau, and F. Cottet. Etude hors ligne d'une application temps-réel à contraintes strictes. *Technique et science informatique*, 19(10) :1373–1398, 2000.

[9] E.G Coffman. Introduction to deterministic scheduling theory. *Computer and Job-shop scheduling theory*, 1976.

[10] F. Cottet and J.P. Babau. Off-line temporal analysis of hard real-time applications. In 2nd *IEEE Workshop on Real-Time Applications*, 1994.

[11] M.L. Dertouzos. Control robotics : the procedural control of physical processes. *Informatop Processing 74*, 1974.

[12] E. Grolleau and A.Choquet-Geniet. Cyclicité des ordonnancements des systèmes de tâches périodiques différés. *proceedings of RTS'2000*, pages 216–229, 2000.

[13] E. Grolleau and A. Choquet-Geniet. Off line computation of real time schedules by means of petri nets. *Journal of Discrete Event Dynamic Systems*, 12 :311–333, 2002.

[14] J. Lehoczky and S. Ramos-Thuel. Cheduling periodic and aperiodic tasks using the slack stealing algorithm. In S.H. Son, editor, *Advances in real-ime systems*, pages 175–198. Prentice-Hall, 1995.

[15] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm : exact characterisation and average case behaviour. *Proc. of the 10th IEEE Real Time Systems Symposium*, pages 166–171, 1989.

[16] J. Lehoczky, L. Sha, and J.K Strosnider. Enhanced aperiodic responsiveness in hard real time environments. In *Proc. of the 8th IEEE Real-Time Systems Symposium*, pages 261–271, 1987.

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

[18] J.Y.T. Leung and M.L. Merill. A note on preemptive scheduling of periodic real-time tasks. *Information Processing Letters*, 11(3) :115–118, 1980.

[19] 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.

[20] A.K. Mok. *Fundamental Design Problems for the Hard Real-Time Environments*. PhD thesis, MIT, 1983.

[21] A.K. Mok and M.L. Dertouzos. Multi processor scheduling in a hard real-time environment. In *Proc. of* 7th *Texas Conference on Computer Systems*, 1978.

[22] L. Sha, R. Rajkumar, and J. Lehockzy. Priority inheritance protocols : an approach to real-time synchronisation. *IEEE Transaction Computers*, 39(9), 1990.

[23] M. Silly. Un algorithme d'ordonnancement de tâches sporadiques pour les systèmes temps réel. *APII*, 28(2) :179–205, 1994.

[24] M. Silly, H. Chetto, and N. Elyounsi. An optimal algorithm for guaranteeing sporadic tasks in hard real-time systems. In *Proc. of SPDS'90*, pages 578–585, 1990.

[25] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real time systems. *The journal of Real time Systems*, 1 :27–60, 1989.

[26] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *The journal of Real Tim Systems*, 10 :179–210, 1996.

[27] A. Stankovic, M. Spuri, K. Ramamritham, and G.C. Buttazzo. *Deadline Scheduling for Real-Time Systems*. Kluwer Academic Press, 1998.

[28] T.S. Tia, J.W. Liu, and M. Shankar. Algorithms and optimality of scheduling soft aperiodic requests in fixed priority preemptive systems. *The Journal of Real Time Systems*, 10(1) :23–43, 1996.

[29] J. Xu and D.L. Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3) :360–369, 1990.