Techniques avancées de systèmes d'exploitation

Cours de M.Raymond Devillers

Première licence et maîtrise en informatique

5 janvier 2000

A propos de ce "syllabus"

Les Greetings

Il a rédigé ces notes : Vincent Gérard (vgerard@ulb.ac.be)
Ils l'ont aidé dans la rédaction et l'ont corrigé : Simon Rosen (srosen@ulb.ac.be), Daniel Robinson (drobinso@ulb.ac.be), Eric Swalens (eswalens@ulb.ac.be)
Il l'a relu et corrigé : Mr. Devillers (qui d'autre ?)
Ils ont fourni de bien judicieuses idées : Gloria De Arriba Rodero (gdearrib@ulb.ac.be), Joël Heymbeeck (jheymbee@ulb.ac.be)
Ils ont fourni des notes plus que complètes : Eythan Levy (elevy@ulb.ac.be) et Charlotte Welle (cwelle@ulb.ac.be)
Un merci tout particulier à Catherine Gérard (pour la correction des fôtes) et à Valentina Martinotti (pour son soutien)
Et tous les autres qui, de près ou de loin, ont participé a l'élaboration de ce syllabus.

Les remarques générales :

- Informatique et langue française font rarement bon ménage, alors même si l'on a apporté beaucoup d'attention à la rédaction de ce document, il reste probablement beaucoup de fautes, désolé.
- Ces notes ne constituent pas un syllabus officiel. Elles ont été rédigées sur base des notes des personnes citées ci-dessus et inspirées du précédent syllabus d'OS 1, de Didier Blanjean et Olivier Verloove, ainsi que du mémoire de Vincent Van Lier (que toutes ces personnes en soient remerciées). Si vous voulez les enrichir, faites m'en part (vgerard@ulb.ac.be) afin que tout le monde en profite.
- Même si l'on a cherché à être le plus complet possible et même si ces notes ont été relues, nous vous conseillons vivement de lire attentivement tout ce qui suit. N'hésitez pas non plus à aller au cours. La matière n'est pas simple et le cours change d'année en année !

Voilaaaaaaa, bonne lecture et bonne chance ... pour l'examen !

Table des matières

Chapitre 1

Les Systèmes Temps Réel

1.1 Introduction

Un système "temps réel" (real time system, RT) est un système dont la cohérence ne dépend pas seulement de la correction des résultats des calculs mais également des instants où ils sont fournis. Ce sont, par exemple, des systèmes de contrôle de trafics aériens, de centrales nucléaires, ou même des systèmes de réservation de places (avions, trains, ...). Ce sont des systèmes où le temps de réponse est primordial.

Il existe un certain nombre de problèmes intéressants concernant ce type de système comme, par exemple, le calcul des temps de réponses et la recherche de moyens pour les rendre compatibles avec les "deadlines" (limite de temps à ne pas dépasser pour l'exécution d'une requête) mais ce problème relève plus de l'algorithmique, de l'évaluation des performances et du hardware. Nous allons plutôt considérer les problèmes liés au scheduling des tâches (comment gérer les ressources pour s'assurer qu'on respecte les contraintes temporelles), et considérer les autres comme résolus.

Pour nous simplifier la tâche, nous allons considérer le temps comme étant une variable entière et, de plus, nous n'allons examiner que des systèmes monoprocesseurs et non des systèmes multiprocesseurs. Dans quel contexte allons-nous nous placer ? Plusieurs cas peuvent être envisagés :
Les ressources (et en particulier le processeur) peuvent être

- Préemptives avec
  1. priorités statiques (quand une tâche plus prioritaire arrive, ou redevient schedulable, on interrompt la tâche courante).
  2. priorités dynamiques (attention à la perte de temps pour le calcul des priorités).
- Non préemptives

Les tâches peuvent être

1. Indépendantes (la seule interaction entre les tâches est la compétition pour le processeur) ou dépendantes.
2. Périodiques (les tâches reviennent périodiquement ; on a une infinité de requêtes) ou apériodiques (simples ou sporadiques).

Les deadlines peuvent être

- Hard (la deadline doit absolument être satisfaite).

- Soft (ce n'est pas catastrophique si la deadline n'est pas respectée, mais il y a un coût à subir qu'on préférerait éviter ou limiter. On peut choisir de continuer la tâche ou bien de l'arrêter).

1.2 Quelques notions

- Une tâche τ_i périodique sera définie par les 4 paramètres suivants :

  T_i représente la période (intervalle de temps au bout duquel la tâche τ_i relancera une nouvelle requête).

  D_i est la deadline de la tâche τ_i (plus exactement le délai de deadline imposé à ses requêtes) : si une requête est lancée en t, la réponse devra arriver avant $t + D_i$. La deadline peut être de trois types :
    - Late deadline : $D_i = T_i$; dans ce cas, on impose simplement que la j^{eme} requête de la i^{eme} tâche soit terminée avant la $j+1^{eme}$ requête ; il n'y aura donc jamais (si les deadlines sont respectées) deux requêtes simultanément actives (ready to run) pour cette tâche.
    - General deadline : $D_i \leq T_i$; c'est donc en général plus restrictif que dans le cas late deadline.
    - Arbitrary deadline : D_i est quelconque par rapport à T_i ; dans ce cas, on pourra donc avoir plusieurs requêtes d'une même tâche actives en même temps.

  C_i donne le temps de calcul (si on est seul à travailler) de chaque requête de la tâche τ_i. Nous supposerons C_i connu (C_i sera en général une borne supérieure). Il faudra évidemment veiller à ce que $C_i \leq D_i$.

  O_i est l'instant (Offset) de première apparition de la tâche τ_i, et donc de sa première requête ; la j^{eme} requête de la tâche τ_i arrivera donc en $t = O_i + (j-1) \cdot T_i$. Les O_i peuvent être
    - Synchrones (toutes les tâches démarrent en même temps ; les O_i seront alors égaux à 0).
    - Asynchrones : ils sont alors quelconques (≥ 0), mais fixés par les contraintes de l'application.
    - Libres : ils peuvent alors être choisis par l'os, plus exactement par l'algorithme de scheduling dans une passe préliminaire.
- Le processeur peut être géré de façon préemptive ou non.
- On va généralement associer une notion de priorité aux requêtes et donner le processeur à la requête (active) de priorité maximum. Les priorités peuvent être
  - Statiques (attachées aux tâches et fixées a priori) ; dans le cas où plusieurs requêtes actives d'une même tâche coexistent (cas arbitrary ou soft deadline), il faut encore préciser laquelle sera prioritaire : en général, c'est la plus ancienne.
  - Dynamiques :
    - dans le cas simple, les priorités sont attachées aux requêtes ; chaque requête a une priorité par rapport aux autres, mais les requêtes successives d'une même tâche n'ont pas la même priorité ; lorsqu'il n'y a jamais deux requêtes actives pour une même tâche, on traduit généralement cela en disant que la tâche change de priorité au cours du temps.

– dans le cas général, la priorité de chaque requête peut elle-même évoluer au cours du temps.

Rmq : Une tâche apériodique peut être caractérisée par les mêmes paramètres que pour les tâches périodiques, sauf que T_i indique alors le temps minimal entre deux requêtes. Souvent cependant on considérera des requêtes apériodiques (sans paramètre T_i) dont les caractéristiques ne sont connues que lors de leur apparition, avec des contraintes soft (souvent même on omet le paramètre D_i : on demande simplement que le temps de réponse soit le plus court possible, tout en continuant à respecter les deadlines des requêtes des tâches périodiques).

1.3 Le facteur d'utilisation U

Le facteur d'utilisation U du processeur est donné par la formule.

$$U = \sum_i \frac{C_i}{T_i}$$

Son nom provient de la propriété très générale qui dit que, pour un système de tâches périodiques schédulables (il y a moyen de schéduler toutes les requêtes en respectant leur deadline), U est le taux limite d'utilisation du processeur, c'est-à-dire la limite lorsque $T \to \infty$ de la fraction de temps de processeur utilisé jusqu'à l'instant T (que les tâches soient indépendantes ou non, avec un processeur préemptif ou non, avec des priorités statiques ou dynamiques, que les deadlines soient late, générales ou arbitraires, que les offsets soient synchrones, asynchrones ou libres).

Démonstration :

Quelle est la limite de la fraction de temps CPU exploitée ? Observons le système jusqu'à un instant T et intéressons-nous d'abord aux requêtes dont on est sûr qu'elles sont terminées en T. On sait que $O_i + (k-1) \cdot T_i$ donne l'instant d'apparition de la k^{eme} requête de la tâche τ_i (on suppose que la numérotation des requêtes des tâches commence à 1 ; on suppose aussi généralement que le plus petit des O_i est nul, mais cela n'est pas nécessaire ici : il suffit qu'il soit non négatif).

La condition pour que l'on soit sûr que la k^{eme} requête soit complètement exécutée est

$$O_i + (k-1) \cdot T_i + D_i \leq T$$

De la formule ci-dessus, on peut tirer k_i, le nombre de requêtes de la tâche i sûrement exécutées complètement dans l'intervalle [0, T] : c'est la plus grande valeur k telle que

$$k \leq \frac{T - D_i - O_i + T_i}{T_i}$$

Comme k_i doit être entier, on voit immédiatement que la plus grande valeur possible de k_i est donnée par

$$k_i = \left\lfloor \frac{T - D_i - O_i + T_i}{T_i} \right\rfloor$$

Cependant, si O_i est très grand, k_i peut être négatif. Or cela veut dire que l'on ne dispose alors d'aucune requête : on considérera donc que $k_i = 0$. Pour éviter ce genre de désagrément, on introduit la notation

$$(x)^+ = \begin{cases} x & \text{si } x \geq 0 \\ 0 & \text{sinon} \end{cases}$$

De ce qui précède, nous pouvons conclure que le temps total d'exécution de ces k_i requêtes pour chaque tâche est donné par

$$\sum_i \left\lfloor \frac{(T - D_i - O_i + T_i)^+}{T_i} \right\rfloor \cdot C_i$$

(si C_i donne le pire temps d'exécution de la tâche τ_i, on suppose donc ici que le pire cas se produit à chaque fois).

Essayons à présent de calculer t, le temps d'utilisation du CPU dans l'intervalle [0, T].

Clairement, t est égal à la somme des temps d'utilisation CPU des requêtes dont on est sûr qu'elles sont complètement exécutées, d'une part, et des temps d'utilisation CPU des requêtes dont l'exécution a démarré mais trop près de T pour que l'on soit sûr qu'elles sont complètement exécutées en T (ce qui ne veut pas dire qu'elles sont nécessairement non-terminées), d'autre part.

$$\Longrightarrow\ t = \sum_i \left\lfloor \frac{(T - D_i - O_i + T_i)^+}{T_i} \right\rfloor \cdot C_i\ + \text{Reste}$$

Essayons de préciser ce Reste.

Reprenons la formule qui caractérise les requêtes qui ont démarré dans l'intervalle [0,T) : $O_i + (k-1) \cdot T_i < T$; de là, on peut tirer que : $k < \frac{T - O_i}{T_i} + 1$.

La plus grande valeur de k satisfaisant cette inéquation est aussi la plus petite valeur de k telle que $k + 1 \geq \frac{T - O_i}{T_i} + 1$.

Comme k est un entier naturel, cette valeur est :

$$\left\lceil \frac{(T - O_i)^+}{T_i} \right\rceil$$

qui donne le nombre de requêtes de τ_i ayant démarré avant T.

Le nombre de requêtes de la tâche τ_i qui ont démarré mais dont on n'est pas sûr qu'elles se soient terminées sera donc donné par :

$$\left\lceil \frac{(T - O_i)^+}{T_i} \right\rceil - \left\lfloor \frac{(T - D_i - O_i + T_i)^+}{T_i} \right\rfloor$$

c'est-à-dire la différence entre le nombre de requêtes démarrées et le nombre de requêtes sûrement terminées.

Le Reste peut donc s'exprimer comme :

$$\left[\sum_i \left(\left\lceil \frac{(T-O_i)^+}{T_i} \right\rceil - \left\lfloor \frac{(T-D_i-O_i+T_i)^+}{T_i} \right\rfloor\right) \cdot C_i\right] \cdot \varepsilon$$

On a multiplié par C_i pour avoir le temps d'exécution des requêtes ; on a ensuite sommé sur i pour tenir compte de toutes les tâches ; et finalement on a multiplié par ε, une constante réelle comprise entre 0 et 1, qui exprime le degré de complétude de ces requêtes entamées (égal à 0 lorsqu'aucune des requêtes lancées n'a eu le CPU avant T, 1 si elles sont toutes terminées).

La quantité qui nous intéresse ici est le pourcentage d'utilisation du processeur par les différentes tâches durant les T premières unités de temps, pris à la limite, c'est-à-dire la quantité $\lim_{T\to\infty} \frac{t}{T}$.

Que se passe-t-il lors du passage à la limite ?

- On peut éliminer les fonctions $()^+$ introduites précédemment
- la parenthèse

  $$\left\lceil \frac{(T-O_i)^+}{T_i} \right\rceil - \left\lfloor \frac{(T-D_i-O_i+T_i)^+}{T_i} \right\rfloor \text{ est bornée par } \frac{D_i}{T_i}+1$$

  En effet, jusqu'en $T - D_i$, on est certain que les requêtes pourront terminer leurs exécutions. Par contre de $T - D_i$ à T, il se pourrait qu'une exécution ait commencé mais ne puisse pas se terminer. La parenthèse calcule l'avancement de l'exécution et est donc bien bornée par $\frac{D_i}{T_i}+1$ (le +1 n'intervenant que pour s'assurer que l'on a bien une borne supérieure ; la fonction plafond conviendrait également).

  Le Reste peut donc s'exprimer de la manière suivante :

  $$\left(\sum_i \left(\frac{D_i}{T_i}+1\right) \cdot C_i\right) \cdot \varepsilon$$

  qui est borné puisque D_i, T_i, C_i et ε sont des nombres finis ; lors du passage à la limite, le Reste divisé par T (qui tend vers l'infini), donnera donc une contribution nulle.
- On peut exprimer $\lfloor \frac{T-D_i-O_i+T_i}{T_i} \rfloor$ sous la forme $\frac{T-D_i-O_i+T_i}{T_i} - \varepsilon_i$, où $0 \leq \varepsilon_i < 1$.

L'équation peut alors s'exprimer de la manière suivante :

$$\lim_{T\to\infty} \frac{t}{T} = \lim_{T\to\infty} \sum_i \left(\frac{T-D_i-O_i+T_i}{T_i} - \varepsilon_i\right) \cdot \frac{C_i}{T} = \sum_i \frac{C_i}{T_i} = U$$

puisque $\frac{-D_i-O_i+T_i}{T_i} - \varepsilon_i$ est borné pour la même raison qu'auparavant, et sa division par T donne à la limite une contribution nulle.

CQFD

On a donc réussi à obtenir une propriété qui demande très peu d'hypothèses de départ : seulement que le système soit schédulable et les tâches périodiques.

De ce qui précède, nous pouvons tirer une propriété importante : si $U > 1$ dans un système monoprocesseur (ou $U > n$ pour un système à n processeurs), cela signifie que le système n'est pas schédulable, quelles que soient les circonstances et les techniques de scheduling ; il ne sert donc à rien de se poser des questions sur le scheduling, sauf si les deadlines sont softs et que l'on désire connaître nombre moyen de deadlines ratées, par exemple.

Rmq : $\frac{C_i}{T_i}$ représente le pourcentage limite d'utilisation du CPU pour chaque tâche τ_i.

1.4 Gestion non préemptible

En se basant sur des techniques non préemptibles et en faisant l'hypothèse que toutes les tâches sont en mémoire, on accroît clairement la rapidité du système car de cette manière on évite tout swap de processus. De plus, en prenant soin de choisir une granularité au niveau de la minute, on pourra négliger le temps de switch entre deux requêtes (quand on sait que les switchs peuvent prendre plusieurs micro-secondes, on évitera de prendre une granularité trop faible).

Plaçons-nous dans un environnement non préemptif, où les tâches sont périodiques et indépendantes et où les deadlines sont de type hard. A quelle requête doit-on attribuer le processeur en premier ? Naturellement, on pourrait se dire qu'on va donner le CPU au processus dont la deadline est la plus courte (c'est-à-dire le processus le plus urgent). Cependant il n'en est rien.

Vérification :

Soit une tâche τ_1 avec $C_1 = 2$, $D_1 = 4 = T_1$ et $O_1 = 0$
Soit une tâche τ_2 avec $C_2 = 1 = D_2$, $T_2 = 3$ et $O_2 = 0$

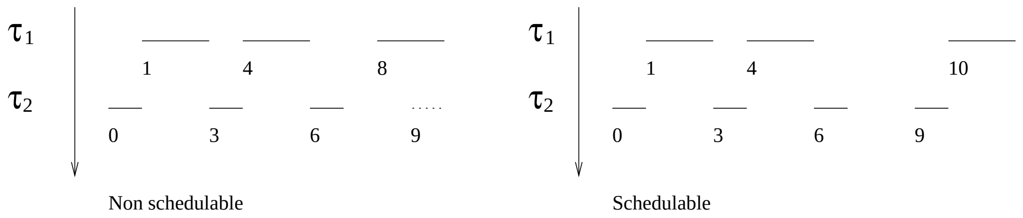

FIG. 1.1: Exemple d'attribution du CPU

Si on attribue, à chaque fois, le processeur à la requête qui est la plus urgente, on constate que le système n'est pas schédulable (voir le schedule de gauche de la figure 1.1 : la requête de τ_2 démarrant à l'instant 9 rate sa deadline) ; alors qu'en utilisant une autre technique (voir le schedule de droite de la figure 1.1, répété toutes les 12 unités de temps), ce système le devient.

Le problème qui se pose ici vient du fait que le système accorde le processeur dès que celui-ci est libre et qu'une requête (ou plus) le demande. Cette technique est connue sous le nom d'expédiante (un système est expédiant s'il ne retarde jamais un processus qui demande une ressource disponible, sauf pour accorder la ressource à un autre processus demandeur). Or, nous avons vu que le moyen de résoudre le problème à la figure 1.1 nécessitait une stratégie non expédiante. Cela signifie que, avant d'accorder le processeur, et de se demander à qui va-t-on l'attribuer, il faut d'abord se demander s'il faut l'attribuer tout de suite. En effet, imaginons que l'on donne le processeur à une tâche τ_i et que survienne, pendant l'exécution de τ_i, une requête de la tâche τ_j plus urgente, qu'il faille traiter immédiatement. Comme l'on se situe dans des techniques non préemptibles, on ne pourra pas arrêter l'exécution de τ_i pour donner le CPU à la tâche τ_j, qui va alors rater sa deadline.

Cela conduit à une recherche systématique, de lourdeur exponentielle en le nombre de tâches, sur le scheduling des requêtes. C'est-à-dire qu'il faudra examiner tous les cas avant de savoir exactement à qui accorder le CPU et quand.

Cependant, on peut remarquer que le système est cyclique ; et donc qu'après un certain laps de temps, on se retrouvera dans la même situation qu'au départ. Pour illustrer cela, on peut reprendre l'exemple précédent : à l'instant 12, on se retrouvera exactement dans la même situation qu'à l'instant 0. De par ce fait, si on arrive à schéduler le système jusqu'à l'instant 12, on y arrivera forcément après.

Comme l'étude de ces systèmes est très complexe, on en restera là dans ce cours quant à l'exploitation de techniques non-préemptives.

1.5 Gestion préemptible avec priorités statiques et late deadline

Apportons d'emblée des précisions sur le système :

1. On considère les switching times comme très petits, et donc négligeables, grâce à une granularité bien choisie.
2. On ne considérera que des stratégies expédiantes.
3. On donne le CPU à la requête la plus prioritaire qui en a besoin (si une requête plus prioritaire arrive pendant son exécution, il suffira de lui reprendre le CPU, vu que nous sommes dans des techniques préemptibles).

Nous considérerons d'abord le cas où les priorités sont statiques (elles sont associées aux tâches et ne changent pas au cours du temps) et avec late deadline ; c'est-à-dire que l'on attribue le CPU à la tâche la plus prioritaire ayant une requête active, et cette dernière doit se terminer au plus tard au moment de l'arrivée de la requête suivante. On peut toujours supposer que les tâches sont rangées par ordre de priorité décroissante :

$$\text{classement des priorités} : \tau_1 < \tau_2 < \tau_3 < \dots$$

Le scheduling d'une tâche τ_k ne dépend pas des tâches suivantes dans l'ordre de priorité, puisqu'elles sont toutes moins prioritaires et donc uniquement traitées dans les trous d'usage du processeur laissés par les tâches plus prioritaires. Par contre, le scheduling de τ_k dépendra des tâches τ_h avec $h < k$ (car celles-là sont plus prioritaires).

Nous allons d'abord montrer que, si le système est schedulable, le schedule est périodique.

FIG. 1.2: Cas avec une tâche

- Prenons un cas simple : τ_1 ;
  comme aucune tâche ne la précède (dans l'ordre de priorité), on peut la considérer comme étant seule et périodique à partir de O_1.
  Une amélioration possible consiste à considérer la tâche un peu avant O_1, car elle est alors inactive tout comme elle l'est pendant un intervalle de longueur $T_1 - C_1$ juste avant T_1, $2 \cdot T_1$, ... Donc, on peut considérer que le schedule de τ_1 est périodique de période $P_1 = T_1$ à partir de l'instant : $O'_1 = O_1 + C_1 - T_1$.
- Regardons maintenant le cas de la deuxième tâche :

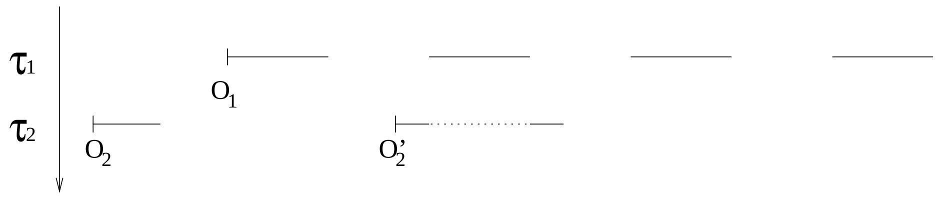

FIG. 1.3: Cas avec deux tâches

  Comme τ_1 est périodique et qu'elle possède la plus haute priorité, tous les trous qu'elle laisse disponibles pour τ_2 sont eux-mêmes périodiques à partir de O'_1. Comme les requêtes de τ_2 se reproduisent tous les T_2 unités de temps à partir de O_2, examinons-les à partir de la première requête arrivant après O'_1 (inclus) : on a donc $O_2 + (k-1) \cdot T_2 \geq O'_1$, c-à-d $k = \left\lceil \frac{(O'_1 - O_2)^+}{T_2} \right\rceil + 1$; leur scheduling va se reproduire tous les $P_2 = ppcm(P_1, T_2)$ unités de temps à partir de (peut-être un petit peu avant en fait, comme pour τ_1 et pour la même raison, mais cela nous suffira ici car le gain est difficile à préciser pour les tâches suivantes)

$$O'_2 = O_2 + \left\lceil \frac{(O'_1 - O_2)^+}{T_2} \right\rceil \cdot T_2$$

  Tout ce qui vient d'être dit est valable dès que le système est schédulable et qu'il n'y a jamais 2 requêtes actives simultanément pour les tâches considérées, donc pas uniquement dans le cas late deadline.
- le même type de raisonnement peut être appliqué aux tâches suivantes : le schedule de τ_i sera périodique de période $P_i = ppcm(P_{i-1}, T_i) = ppcm(T_1, T_2, \ldots, T_i)$, à partir de l'instant

$$O'_i = O_i + \left\lceil \frac{(O'_{i-1} - O_i)^+}{T_i} \right\rceil \cdot T_i$$

Conclusion : un système composé de n tâches sera schédulable à partir de O'_n avec une période $P = P_n = ppcm(T_1, T_2, \ldots, T_n)$, aussi appelée l'hyperpériode du système. Elle représente la période du système tout entier.

On peut remarquer que

- la période peut être très longue
- le comportement du système n'est en général pas directement périodique (sauf dans le cas synchrone). On dit qu'il passe par une phase de mise en régime (et ça peut être long !).

1.5.1 Pires temps de réponses

Essayons de déterminer le temps de réponse (c'est à dire la différence entre l'instant d'arrivée d'une requête et son instant de terminaison) le plus long (c'est à dire le pire cas) pour chaque tâche, lorsqu'on laisse les offsets libres, en supposant qu'il n'y ait jamais de dépassement de deadline.

On sait déjà qu'une tâche ne dépend pas de ses suivantes. A partir de là, introduisons le concept de *busy period* de niveau i. C'est une période pendant laquelle le processeur est continuellement utilisé par les requêtes des tâches $\tau_1, \ldots \tau_i$ et qu'on ne peut pas allonger, ni à gauche ni à droite, sans perdre cette propriété ; elle sera donc délimitée de chaque côté par un trou dans l'usage du processeur par ces tâches, sauf si elle s'étend à l'infini vers la droite (ce ne sera jamais le cas vers la gauche) mais alors les tâches de priorité inférieure (s'il y en a) ne pourront plus jamais recevoir le processeur et respecter leurs deadlines ; on pourra considérer que, à l'intérieur (strictement, en les instants entiers) d'un trou, on a des busy periods de taille nulle.

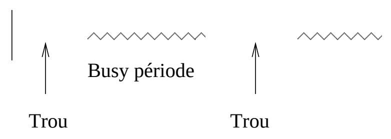

FIG. 1.4: Le concept de busy period

Considérons une requête quelconque de τ_i pour $i > 1$ (pour τ_1, toutes les requêtes ont le même temps de répone C_1) et la busy period de niveau $i - 1$ (éventuellement de longueur nulle) à laquelle appartient son instant d'arrivée. On peut faire les remarques suivantes :

- les requêtes précédentes de τ_i (s'il y en a) sont terminées par définition (systeme schedulable et late deadlines) : on peut donc les oublier et considérer qu'il s'agit de la première requête de τ_i. On peut de même oublier les requêtes de τ_j ($j < i$) qui précèdent la busy period, puisque cette dernière est précédée d'un trou dans l'usage du processeur par les tâches plus prioritaires, et donc considérer que la première requête de τ_j arrive en ou après le début de la busy period.
- si la requête de τ_i n'arrive pas au début de la busy period et si on la décale vers cet instant, cela ne change rien à son instant de terminaison (de toute façon elle devra attendre la fin de la busy period avant de commencer à s'exécuter), et donc ça ne peut qu'augmenter le temps de réponse.

- Si une requête de τ_j $(j < i)$ arrive après celle considéré pour τ_i mais avant son instant de terminaison, elle a un impact sur le temps de réponse d'exactement C_j unités de temps (l'exécution de τ_i est repoussée de C_j). L'impact total de la tâche τ_j sur le temps de réponse considéré est donc de $n_j \cdot C_j$, où n_j désigne le nombre de requêtes de τ_j arrivant pendant la réponse de τ_i.
- Si la première requête de τ_j n'arrive pas au début de la busy period et si on la décale vers cet instant, cela ne peut qu'augmenter le facteur n_j (il restera en fait le même, sauf si le décalage amène de nouvelles requêtes dans l'ancien intervalle de réponse, postposant d'autant sa terminaison).

En conséquence, le pire moment d'arrivée pour une requête de la tâche τ_i sera donc celui où τ_i arrive en même temps que toutes les tâches τ_j $(j < i)$: ce cas est tout simplement le **cas synchrone**. Plus précisément, on vient de prouver que le pire temps de réponse pour chaque tâche est celui de la première requête dans le cas synchrone.

Le même type de raisonnement permet de montrer que si un système n'est pas schédulable, il ne l'est pas non plus dans le cas synchrone (qui peut donc être considéré comme le pire cas) et l'échec se produit dans la réponse pour la première requête de l'une des tâches.

On peut donc aussi affirmer que, pour savoir si un système synchrone est schédulable, il suffit de simuler son évolution jusqu'à la fin de la plus longue première requête. Cependant, simuler est une opération assez lourde. Il sera donc plus indiqué de recourir aux mathématiques afin de simuler la simulation (ouf).

Soit ρ_i , le temps de réponse de la première requête de la tâche τ_i dans le cas synchrone. On peut écrire la relation suivante :

$$\rho_i = C_i + \sum_{j<i} C_j \cdot \left\lceil \frac{\rho_i}{T_j} \right\rceil$$

En effet, le temps de réponse d'une requête (le temps qu'il faut attendre entre le moment où la requête est apparue dans le système et le moment où l'on a la réponse) comprend évidemment le temps de calcul de la requête (donné ici par C_i), auquel il faut ajouter le temps pendant lequel la requête a été retardée par des requêtes plus prioritaires pendant son exécution. Comme vu plus haut, ce dernier temps est fourni par la somme des temps d'exécution des requêtes des tâches plus prioritaires arrivées strictement avant la fin de la première requête de τ_i ; on a vu plus haut que le nombre de requêtes de τ_j arrivées strictement avant un instant x est fourni par la formule $\left\lceil \frac{(x-O_j)^+}{T_j} \right\rceil$; appliqué à notre cas, cela donne $n_j = \left\lceil \frac{\rho_i}{T_j} \right\rceil$, puisque x correspond ici à la période d'attente et que le système est synchrone, et donc le décalage total induit par les tâches plus prioritaires est donné par $\sum_{j<i} C_j \cdot \left\lceil \frac{\rho_i}{T_j} \right\rceil$.

Rmq : cette relation peut avoir plusieurs solutions, par exemple $\rho_i + C_1$ si τ_1 lance une nouvelle requête en ρ_i et qu'aucune autre tâche prioritaire ne lance une nouvelle requête dans l'intervalle $[\rho_i, \rho_i + C_1)$; cependant on s'intéressera uniquement à la plus petite solution (positive), car c'est elle qui fournit le temps de réponse recherché.

<u>Méthode de calcul :</u>

$$\begin{cases} x_0 = \sum_{j \leq i} C_j & \text{valeur de départ} \\ x_k = C_i + \sum_{j<i} C_j \cdot \left\lceil \frac{x_{k-1}}{T_j} \right\rceil & \text{pour } k > 0 \end{cases}$$

On itérera ce processus jusqu'au moment où cela se stabilisera (dans ce cas, on aura la plus petite solution recherchée) ou bien que l'on aura dépassé la deadline (dans ce cas, on saura que le système synchrone considéré est non schedulable).

Dans le cas présent, on n'a effectué les calculs que pour la première requête de la tâche τ_i, en supposant que toutes les autres requêtes n'ont pas connu d'ennuis auparavant. On pourrait améliorer la situation en calculant tous les ρ_i (c'est-à-dire pour toutes les tâches τ_i) et ce dans l'ordre. Dans ce cas, on peut prendre comme valeur de départ $x_0 = \rho_{i-1} + C_i$, puisque τ_i ne peut commencer à travailler que lorsque la première requête de τ_{i-1} est terminée.

Rmq : On ne doit pas simuler le système pendant toute une hyper-période car on ne prend que les premières requêtes.

Il nous reste encore à déterminer la stratégie optimale, c'est à dire le meilleur choix des priorités ! Plaçons-nous dans le cas synchrone et supposons que l'on a :

$$\tau_1 < \tau_2 < \ldots < \tau_i < \tau_{i+1} < \ldots < \tau_n$$

Supposons aussi que le système est schédulable et que $T_i > T_{i+1}$. Que se passerait-il si τ_{i+1} devenait plus prioritaire que τ_i ?

$$\tau_1 < \tau_2 < \ldots < \tau_{i+1} < \tau_i < \ldots < \tau_n$$

D'après ce que l'on a vu plus haut, pour tester la schédulabilité, il suffit d'examiner les premières requêtes de chaque tâche. Si on permute τ_i et τ_{i+1}, il n'y aura aucun impact sur les tâches plus prioritaires. Il n'y aura non plus aucun impact sur les tâches moins prioritaires (voir par exemple la formule pour ρ_j donnée ci-dessus). Comme D_{i+1} était respectée avant la permutation, la deadline de τ_{i+1} est certainement respectée si τ_{i+1} devient plus prioritaire que τ_i. De plus, puisque $T_i > T_{i+1}$, comme le système était schedulable avant la permutation, la seconde requête de τ_i ne pouvait pas perturber la première requête de τ_{i+1} (elle s'était terminée avant T_{i+1} et donc avant T_i). Dès lors, on peut voir que, après la permutation, la première requête de τ_i va s'achever au même point que la première requête de τ_{i+1} avant la permutation, donc avant T_{i+1}, donc avant T_i, et τ_i respecte bien aussi sa (première) deadline après la permutation. La permutation des deux tâches n'a donc rien changé à la schédulabilité : Si c'était schédulable avant, ça l'est encore maintenant. Par contre, si ce n'était pas schédulable avant, ça pourrait l'être maintenant. Le meilleur choix des priorités consiste donc à ordonner les tâches suivant leur période, de la plus petite à la plus grande. Cette technique est appellée RMS (pour Rate Monotonic Scheduling).

De tout ce qui précède, on peut tirer une conclusion importantissime : la stratégie optimale de scheduling temps réel dans le cas statique, late deadline et synchrone est le RMS.

Rmq : "Entre borne et un, ca ressemble à Borneo."

R. Devillers, le 23/09/1999

1.5.2 Least Upper Bound de U (lub U)

Le paragraphe 1.3 nous a fourni un critère nécessaire de schédulabilité, à savoir $U \leq 1$. Serait-il possible d'en tirer aussi un critère suffisant ?

Introduisons le $Lub_n\ U$. Le $Lub_n\ U$ est l'infimum des valeurs de U pour les systèmes à n tâches utilisant pleinement le processeur. On dit qu'un système "occupe pleinement le processeur" s'il est schédulable et qu'on ne sait augmenter aucun des C_i sans qu'il ne devienne non schédulable. *Lub* signifie "least upper bound",least bound puisqu'on prend l'infimum, upper bound puisqu'on ne considère que des systèmes qui utilisent pleinement le processeur :

$$Lub_n\ U = inf\{U|\text{plein usage avec } n \text{ tâches}\}$$

Nous allons voir que si nous avons n tâches synchrones avec late deadlines sous RMS,

$$lub_n\ U = n(\sqrt[n]{2} - 1).$$

Pour avoir un système utilisant pleinement le processeur, il faut et il suffit de choisir les C_i de manière à ce que les $n-1$ premières tâches soient schédulables et, pour τ_n, on détermine le C_n qui permet de récupérer tous les trous entre 0 et T_n.

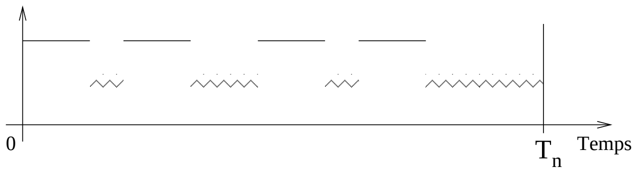

FIG. 1.5: Récupération des trous pour la tâche τ_n

Ce système est schédulable car $T_n = D_n$ et ρ_n coincide soit avec T_n, soit avec le début de la dernière busy period de niveau $n-1$ englobant T_n. De plus, on utilise pleinement le processeur car si l'on augmente un C_i ou C_n, le système ne sera plus schédulable.

Nous allons calculer $lub_2\ U$, dans le cas de deux processus donc, et schématiser ensuite le raisonnement dans le cas plus général avec n tâches.

Rmq : Le cas avec une tâche n'est pas intéressant car $U = \frac{C_1}{T_1}$, et comme nous sommes dans un système avec plein usage du CPU, on doit prendre $C_1 = T_1$, de sorte que $Lub_1\ U = 1$.

Deux tâches ($n = 2$)

On ne perd pas de vue que nous sommes dans le cas RMS donc que $T_1 \leq T_2$. On va d'abord fixer T_1 et T_2, et déterminer comment choisir C_1 et C_2 de façon à minimiser U tout en utilisant pleinement le processeur. Plusieurs cas se présentent :

- T_2 multiple de T_1 ($T_2 = kT_1$) :
  Dans ce cas-ci, on peut prendre $C_2 = k\ (T_1 - C_1)$. On obtient alors

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2}$$

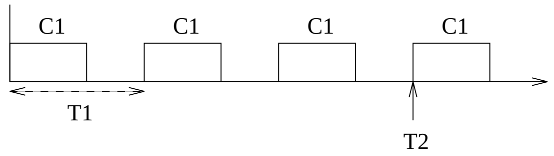

FIG. 1.6: Deux tâches avec $T_2 = k\ T_1$

$$
\begin{aligned}
&= \frac{C_1}{T_1} + \frac{1}{T_2}\, k\ (T_1 - C_1) \\
&= \frac{C_1}{T_1} + \frac{k\ T_1}{T_2} - \frac{k\ C_1}{T_2} \\
&= \frac{C_1}{T_1} + 1 - \frac{C_1}{T_1} \\
&= 1
\end{aligned}
$$

U est donc maximal (et le système occupe continuellement le processeur : le processeur n'est jamais idle). Comme on cherche à minimiser U, ce cas n'est pas très intéressant.

- T_2 n'est pas un multiple de T_1 ($T_2 = k\ T_1 + \delta$ avec $\delta = T_2\ mod\ T_1$)

  1. $C_1 \leq \delta$ (τ_1 a eu le temps de se terminer pendant l'intervalle δ) :

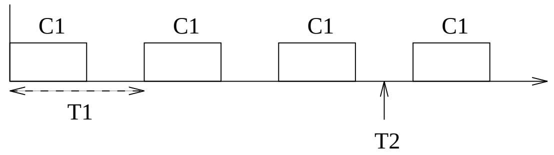

FIG. 1.7: Deux tâches avec $T_2 = k\ T_1 + \delta$ et $C_1 \leq \delta$

Nous pouvons prendre $C_2 = T_2 - (k+1)\ C_1$. Nous avons alors

$$
\begin{aligned}
U &= \frac{C_1}{T_1} + \frac{C_2}{T_2} \\
&= \frac{C_1}{T_1} + \frac{1}{T_2}\left[T_2 - (k+1)C_1\right] \\
&= \frac{C_1}{T_1} + \frac{T_2}{T_2} - \frac{C_1}{T_2}\ (k+1) \\
&= \frac{C_1}{T_1} + 1 - \frac{C_1}{T_2}\,\frac{T_1}{T_1}\ (k+1) \\
&= 1 + \frac{C_1}{T_1}\left(1 - \frac{T_1}{T_2}\ (k+1)\right) \\
&= 1 + \frac{C_1}{T_1}\left[1 - \frac{T_1}{T_2}\left(\frac{T_2 - \delta}{T_1} + 1\right)\right] \\
&= 1 + \frac{C_1}{T_1}\left[1 - \left(\frac{T_1\ T_2}{T_2\ T_1} - \frac{T_1\ \delta}{T_2\ T_1} + \frac{T_1}{T_2}\right)\right] \\
&= 1 + C_1 \underbrace{\left(\frac{\delta - T_1}{T_1 T_2}\right)}_{<0}
\end{aligned}
$$

Le processeur est donc parfois inutilisé. En effet, il n'est totalement utilisé qu'au début (car les deux tâches commencent en même temps), puis en $ppcm(T_1, T_2), 2 \cdot ppcm(T_1, T_2), \ldots$ On voit même que, plus C_1 augmente, plus le facteur d'utilisation diminue.

2. $C_1 > \delta$:

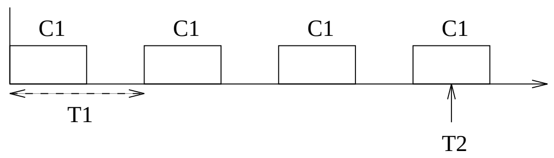

FIG. 1.8: Deux tâches avec $T_2 = k\,T_1 + \delta$ et $C_1 > \delta$

Nous pouvons prendre $C_2 = k \cdot (T_1 - C_1)$. Nous trouvons alors que

$$
\begin{aligned}
U &= \frac{C_1}{T_1} + \frac{C_2}{T_2} \\
&= \frac{C_1}{T_1} + \frac{k}{T_2}\,(T_1 - C_1) \\
&= \frac{C_1}{T_1} + k\,\frac{T_1}{T_2} - k\,\frac{C_1}{T_2} \\
&= \frac{k\,T_1}{T_2} + C_1\left(\frac{1}{T_1} - \frac{k}{T_2}\right) \\
&= \frac{(T_2 - \delta)}{T_1}\,\frac{T_1}{T_2} + C_1\left(\frac{1}{T_1} - \frac{(T_2 - \delta)}{T_1}\,\frac{1}{T_2}\right) \\
&= \left(1 - \frac{\delta}{T_2}\right) + C_1\left[\frac{1}{T_1}\left(1 - \frac{(T_2 - \delta)}{T_2}\right)\right] \\
&= \left(1 - \frac{\delta}{T_2}\right) + C_1 \underbrace{\left(\frac{\delta}{T_1 T_2}\right)}_{>0}
\end{aligned}
$$

Nous pouvons maintenant dessiner le graphe de U en fonction de C_1 (dans le cas où les deux périodes ne sont pas multiples l'une de l'autre). Cette fonction atteint un minimum en $\delta = T_2 \; mod \; T_1$. Pour minimiser U, C_1 doit donc être égal à δ. Il faut évidemment vérifier que la fonction est continue en ce point (ce qui est le cas !).

Essayons maintenant de minimiser U en jouant sur T_1 et T_2 :
on a

$$T_2 = k\,T_1 + \delta\,;\; C_1 = \delta\,;\; C_2 = T_2 - (k+1)\,\delta = k(T_1 - \delta)$$

et donc, en posant $f = \frac{\delta}{T_1}$ de sorte que $\frac{T_2}{T_1} = k + f$,

$$
\begin{aligned}
U &= \frac{C_1}{T_1} + \frac{C_2}{T_2} \\
&= \frac{\delta}{T_1} + \frac{T_2 - (k+1)\delta}{T_2}
\end{aligned}
$$

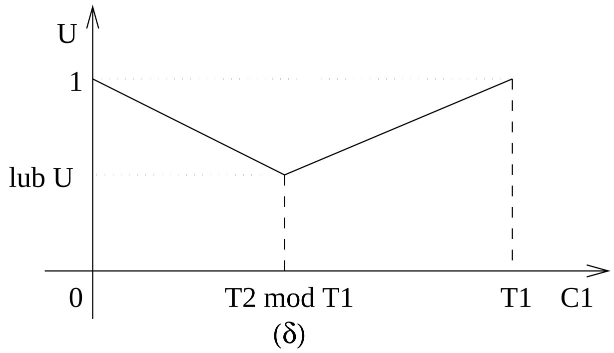

FIG. 1.9: Graphe de U en fonction de C_1

$$
\begin{aligned}
&= 1 + \frac{\delta}{T_1} - \frac{\delta}{T_2}\,(k+1)\,\frac{T_1}{T_1} \\
&= 1 + \frac{\delta}{T_1}\left(1 - \frac{T_1}{T_2}\,(k+1)\right) \\
&= 1 + f\left(1 - \frac{k+1}{k+f}\right) \\
&= 1 + f\left(\frac{k+f-k-1}{k+f}\right) \\
&= 1 - \frac{f(1-f)}{k+f}
\end{aligned}
$$

Fixons provisoirement f. Puisque le numérateur de la fraction est positif (car $0 \leq f < 1$: c'est la partie fractionnaire du quotient $\frac{T_2}{T_1}$, alors que k en est la partie entière) et que nous cherchons à minimiser U, nous devons avoir $k+f$, et donc k, le plus petit possible. Or on sait depuis le début que $k \geq 1$ puisque $T_2 > T_1$; la valeur minimum de k est donc 1.

Essayons maintenant de minimiser U en jouant sur le paramètre f :

$$U = 1 - \frac{f(1-f)}{1+f} = \frac{1+f-f+f^2}{1+f} = \frac{1+f^2}{1+f}$$

Pour chercher le minimum, on va dériver la fonction :

$$U' = \frac{(1+f)2f - 1(1+f^2)}{(1+f)^2} = \frac{2f+2f^2-1-f^2}{(1+f)^2} = \frac{f^2+2f-1}{(1+f)^2} = 0$$

Il faut donc calculer les racines de l'équation $f^2+2f-1=0$. On obtient facilement que $f = -1 \pm \sqrt{2}$. La racine négative est supprimée car elle correspond à un maximum de la fonction (de toute façon, f ne peut pas être négatif) et nous avons donc $f = \sqrt{2}-1$. En fait, f ne sera jamais égal à $\sqrt{2}-1$ car il s'agit d'un nombre irrationnel et f est une fraction, c'est à dire un rationnel. On pourra néanmoins s'en approcher autant qu'on veut.

Si maintenant nous revenons à U :

$$lub_2\,U = \frac{1+f^2}{1+f} = \frac{1+(\sqrt{2}-1)^2}{1+\sqrt{2}-1} = \frac{4-2\sqrt{2}}{\sqrt{2}} = 2\sqrt{2}-2 = 2(\sqrt{2}-1) \simeq 0,83$$

Ce qui correspond bien à $n(\sqrt[n]{2}-1)$ dans le cas de 2 tâches. Ici encore, nous pouvons remarquer que ce nombre est irrationnel, alors que U doit toujours être un nombre rationnel. Cette valeur de $lub_2\,U$ ne sera donc jamais atteinte.

Cas général

Nous disposons d'une liste de tâches, soit $\tau_1, \tau_2, \ldots \tau_n$, classées par périodes croissantes.

Dans un premier temps, nous allons prouver qu'il est inutile de regarder des systèmes avec $T_n \geq 2\ T_1$.

Considérons un système qui ne satisfait pas cette condition, c'est à dire qu'il utilise pleinement le CPU mais que $T_n \geq 2\ T_1$; on peut donc poser $T_n = k\ T_1 + \delta$ avec $k \geq 2$

Prenons $T'_1 = k\ T_1$; en augmentant T_1, on diminue la fréquence d'exécution de τ_1 : appelons τ'_1 cette version allégée de τ_1. τ'_1 ayant vu sa période augmenter, elle n'est peut-être plus la plus prioritaire mais elle reste de toute façon plus prioritaire que τ_n. De plus, dans l'intervalle $[0, T_n)$, il y aura plus de temps disponible pour τ_n, mais aussi pour les autres tâches. Comme nous devons utiliser pleinement le processeur, nous aurons donc $C'_n = C_n + \varepsilon\ (k-1)\ C_1$.
ε est là pour tenir compte de ce que l'on n'est pas sûr de pouvoir récupérer tous les trous supplémentaires laissés par τ'_1 car d'autres processus ont pu en récupérer une partie (sans augmenter leur C_i, mais du fait qu'ils ont une plus grande priorité). Comme d'habitude, $0 \leq \varepsilon \leq 1$.

En calculant U', on arrive à

$$\begin{aligned}
U' &= U - \frac{C_1}{T_1} - \frac{C_n}{T_n} + \frac{C_1}{kT_1} + \frac{C'_n}{T_n} \\
&\leq U - \frac{(k-1)C_1}{kT_1} - \frac{C_n}{T_n} + \frac{C_n + (k-1)C_1}{T_n} \\
&= U - (k-1)C_1(\frac{1}{kT_1} - \frac{1}{T_n}) \\
&\leq U
\end{aligned}$$

puisque $kT_1 \leq T_n$. On en conclut qu'en donnant à τ_1 une période plus proche de τ_n, on peut faire au moins aussi bien en ce qui concerne le facteur d'utilisation.

Nous allons maintenant montrer que l'on peut supposer que $C_1 \leq T_2 - T_1$.

A nouveau, considérons un système qui utilise pleinement le processeur mais pour lequel $C_1 = T_2 - T_1 + \Delta > T_2 - T_1$

Comme $T_1 \leq T_n < 2T_1$, nous savons que jusqu'à T_n (inclus) les tâches font exactement 2 requêtes. Si maintenant nous prenons $C'_1 = C_1 - \Delta$, $C'_2 = C_2 + \Delta$ et $C'_i = C_i$ (pour $i > 2$), on peut voir (cf. figure 1.5.2) que jusqu'en T_n rien ne sera changé pour les tâches moins prioritaires, que le système reste schedulable et utilise pleinement le processeur. Mais si nous calculons U', il ressort que :

$$\begin{aligned}
U' &= \frac{C'_1}{T1} + \frac{C'_2}{T_2} + \sum_{i>2} \frac{C_i}{T_i} \\
&= \frac{C_1 - \Delta}{T_1} + \frac{C_2 + \Delta}{T_2} + \sum_{i>2} \frac{C_i}{T_i} \\
&= \Delta \left(\frac{1}{T_2} - \frac{1}{T_1} \right) + \left(\frac{C_1}{T_1} + \frac{C_2}{T_2} \right) + \sum_{i>2} \frac{C_i}{T_i}
\end{aligned}$$

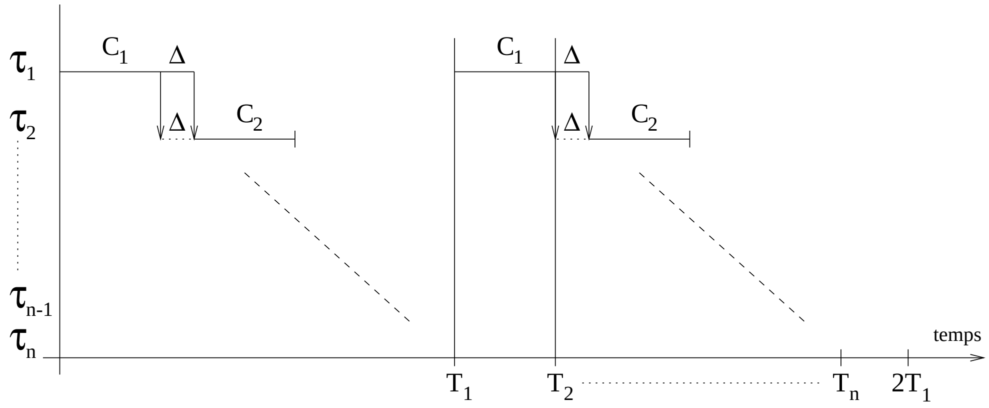

FIG. 1.10: cas où $C_1 > T_2 - T_1$

$$
\begin{aligned}
&= \Delta \underbrace{\left(\frac{1}{T_1} - \frac{1}{T_2}\right)}_{<0 \; car \; T_2 > T_1} + U \\
&< U
\end{aligned}
$$

Et donc, nous obtenons un meilleur facteur d'utilisation... de sorte que le choix $C_1 > T_2 - T_1$ n'est pas satisfaisant !

Le même raisonnement conduit à l'observation que l'on peut se contenter de considérer des systèmes où $C_2 \leq T_3 - T_2$, $C_3 \leq T_4 - T_3$, ... et donc que l'on peut supposer que, pour tout $i = 1, 2, \ldots n-1$, $C_i \leq T_{i+1} - T_i$.

Nous allons maintenant monter que l'on peut supposer que $C_n = 2T_1 - T_n$ et que, pour tout $i = 1, 2, \ldots n-1$, $C_i = T_{i+1} - T_i$.

En effet, comme pour tout $i = 1, 2, \ldots n-1$, $C_i \leq T_{i+1} - T_i$, on peut voir (cf. figure 1.5.2) que pour que le système utilise pleinement le processeur, on doit avoir $C_n = T_n - 2\sum_{i=1}^{n-1} C_i$ (il récupère tout ce qui traîne !).

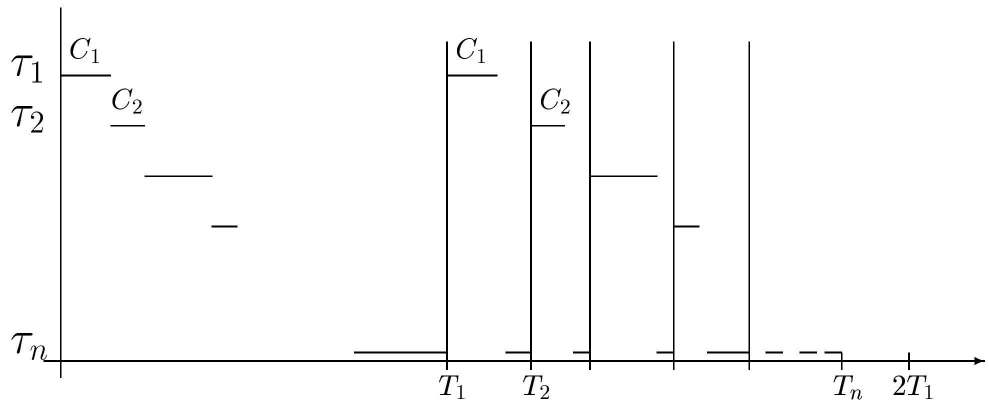

FIG. 1.11: cas où $C_i \leq T_{i+1} - T_i \; \forall i = 1, ..., n-1$

Dès lors, si l'on augmente les C_i jusqu'à leur maximum $C_i' = T_{i+1} - T_i$, on aura

$$C_n' = T_n - 2\sum_{i=1}^{n-1} C_i' = T_n - 2\sum_{i=1}^{n-1}(T_{i+1} - T_i) = T_n - 2(T_n - T_1) = 2T_1 - T_n.$$

Et

$$\begin{aligned} U' - U &= \sum_{i=1}^{n-1}\left(\frac{C_i' - C_i}{T_i} + \frac{2(C_i - C_i')}{T_n}\right) \\ &= \sum_{i=1}^{n-1}(C_i' - C_i)\left(\frac{1}{T_i} - \frac{2}{T_n}\right) \\ &\leq 0 \end{aligned}$$

car, $\forall i < n$, $C_i' \geq C_i$ et $T_n < 2T_1 \leq 2T_i$.
Donc, comme annoncé, on n'a aucun intérêt à choisir des C_i inférieurs à leur borne. On peut donc supposer que

$$\begin{aligned} C_i &= T_{i+1} - T_i \quad \forall i : 1 \leq i \leq n-1 \\ C_n &= 2T_1 - T_n \end{aligned}$$

Posons à présent, dans un but de simplification des formules,

$$g_i = \frac{T_n - T_i}{T_i} = \frac{T_n}{T_i} - 1$$

g_i est la partie fractionnaire de $\frac{T_n}{T_i}$ car on a $1 \leq \frac{T_n}{T_i} < 2$. Il vient

$$\begin{aligned} C_i &= g_i T_i - g_{i+1} T_{i+1} \quad \forall i : 1 \leq i \leq n-1 \\ C_n &= T_n - 2g_1 T_1 \end{aligned}$$

Nous avons alors

$$U = 1 + \sum_{i=1}^{n-1} g_i \frac{g_i - g_{i+1}}{g_i + 1}$$

Pour trouver $lub_n\ U$, il faut minimiser par rapport aux g_i, et donc annuler $n-1$ dérivées partielles, ce qui conduit à $n-1$ équations ; on obtient finalement que

$$g_i = 2^{\frac{n-i}{n}} - 1 \quad \forall i : 1 \leq i \leq n-1$$

De là, on trouve que

$$lub_n\ U = n(\sqrt[n]{2} - 1)$$

Vers quelle valeur tend $lub_n\ U$ quand le nombre de tâches augmente (il décroît à partir de 1) ?

$$\lim_{n\to+\infty} lub_n\ U = \lim_{n\to+\infty} n(\sqrt[n]{2} - 1) = \lim_{n\to+\infty} \frac{2^{\frac{1}{n}} - 1}{\frac{1}{n}} = \lim_{x\to 0} \frac{2^x - 1}{x} \qquad \text{avec } x = \frac{1}{n}.$$

Nous avons une forme indéterminée $\frac{0}{0}$. Nous utilisons alors la règle de *l'Hospital*.

$$\lim_{n\to+\infty} lub_n\ U = \lim_{x\to 0} \frac{2^x \ln 2}{1} = \ln 2 \simeq 0,683.$$

1.5.3 Test de schédulabilité

Le $lub_n U$ que nous venons de déterminer nous fournit un critère simple de schédulabilité : si $U \leq lub_n U$ alors le système correspondant (supposé synchrone avec late deadlines sous RMS) est toujours schédulable. Il s'agit d'une condition suffisante de schédulabilité (mais en général pas nécessaire).

Pour le montrer, procédons par induction :

Si $n = 1$ (nous avons une seule tâche) :

$\Longrightarrow \frac{C_1}{T_1} = U \leq 1 \Longrightarrow C_1 \leq T_1$

et comme nous sommes dans le cas late deadline, le système est d'office schédulable (ici la condition est nécessaire et suffisante)

.
.
.

Si c'est vrai pour $n = k$, est-ce vrai pour $n = k + 1$?
Supposons que $U_{k+1} < lub_{k+1}\ U$ et considérons le système obtenu en laissant tomber la dernière tâche ; il vient

$$U_k = U_{k+1} - \frac{C_{k+1}}{T_{k+1}} < U_{k+1} < lub_{k+1}U < lub_k U$$

car la fonction $lub_k U$ est strictement décroissante en k. Donc, $U' < lub_k\ U$ et par hypothèse d'induction, le système à k tâches obtenu est schédulable. Mais est-ce que le système original est aussi schédulable ?

Procédons par l'absurde et supposons qu'il n'est pas schédulable : ce ne peut être que par la faute de la dernière tâche car on vient de prouver que les k premières sont schédulables.
L'idée pour rendre le système schédulable est de diminuer C_{k+1} de manière à récupérer exactement les trous laissés par les k premières tâches jusqu'à T_{k+1}. On aura alors

1. un système qui utilise pleinement le processeur.
2. et comme C_{k+1} diminue, $U" < U$

Nous obtenons donc un système tel que $U" < U < lub_{k+1}\ U$. Ce qui n'est pas possible car $U"$ désigne le facteur d'utilisation de ce système qui utilise pleinement le CPU, qui serait donc plus petit que l'infimum des valeurs U des systèmes schédulable utilisant pleinement le CPU. Fort de cette contradiction, nous pouvons donc affirmer que le système original **est** schédulable.

CQFD

Nous terminerons cette section en résumant le moyen de vérifier qu'un système RMS, synchrone, late deadline, avec priorité statique est schédulable :

1. Si $U > 1?$ $\longrightarrow$ ce n'est pas schédulable.
2. Sinon si $U < ln2?$ $\longrightarrow$ le système est d'office schédulable.
3. Sinon si $U < lub_n U$ $\longrightarrow$ c'est schédulable.

4. Sinon, il nous reste un dernier moyen de vérifier : on calcule les ρ_i pour la première requête de chaque tâche ; et si, pour chaque ρ_i, on peut vérifier qu'il est bien plus petit que (ou égal à) T_i, on pourra affirmer que le système est schédulable (ce critère est nécessaire et suffisant mais plus lourd à mettre en oeuvre).

1.5.4 Distribution des temps de réponses

Si, au delà de la simple schédulabilité, nous nous intéressons à la distribution des temps de réponses sur un intervalle, comment devrions-nous nous y prendre ? On pourrait simuler. Cependant, dans la mesure où une simulation est toujours très lourde, on cherchera à transformer cette simulation en formule calculable. On va donc s'intéresser, non plus aux temps de réponses des premières requêtes des tâches, mais plutôt à ceux de leur k^e requête.

$$\text{Soit } \delta_i^k : \rho_i^k \text{ avec } k > 1$$

où δ_i^k : désigne la k^e requête d'une tâche i.
où ρ_i^k : désigne le temps de réponse pour cette requête.

Comment procédions-nous pour $k = 1$? Nous étions arrivés à une formule

$$\rho_i = C_i + \sum_{j<i} \left\lceil \frac{\rho_i}{T_j} \right\rceil C_j$$

C'est-à-dire temps de réponse = temps de calcul + les temps de calcul des requêtes plus prioritaires démarées avant la fin des C_i unités de temps calcul à attribuer.

Pour la k^e requête, c'est plus compliqué car elle pourrait être repoussée par des fractions de certains C_j.

Explication :

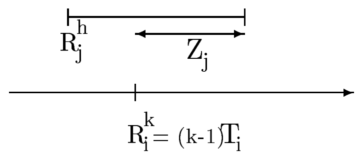

FIG. 1.12:

On va supposer que l'on a déjà calculé les temps de réponses des requêtes plus prioritaires.

Ce n'est pas par n'importe quelle requête de τ_j que l'exécution de δ_i^k peut être repoussée partiellement : c'est la requête $\delta_j^{h_j}$ qui arrive juste avant δ_i^k (strictement). D'après ce que l'on a déjà vu, $h_j = \left\lceil \frac{R_i^k}{T_j} \right\rceil$, où R_i^k est le moment d'arrivée de la k^e requête de la tâche τ_i.

Rmq : On est certain d'avoir une requête avant δ_i^k , car $k > 1$ et on est dans le cas synchrone (on a au moins eu la requête de départ avant).

On peut dès lors déterminer de combien $\delta_j^{h_j}$ va éventuellement repousser l'exécution de δ_i^k :

$$\left(R_j^{h_j} + \rho_j^{h_j} - R_i^k\right)^+ = Z_j$$

On utilise la fonction $()^{+}$ car, si Z_j est négatif, ça signifie que la requête précédente de la tâche j s'est terminée avant R_i^k. Cela n'a donc pas d'impact direct sur le temps de réponse qui nous intéresse.

Rmq : ça pourrait avoir un impact indirect car elle pourrait avoir repoussé une autre requête plus prioritaire que δ_i^k ; on le verra en calculant les autres Z_j.

Recherchons la tâche τ_y tel que le Z_y sera maximum :

$$Z_y = max_{j<i} \left\{ (R_j^{h_j} + \rho_j^{h_j} - R_i^k)^+ \right\}$$

$$y \begin{cases} = 0 & \text{si tous les } Z_j = 0 \text{, et on posera alors } Z_y = 0 \\ > 0 & \text{sinon, } y \text{ est alors unique et donne un numéro de tâche} \end{cases}$$

On va donc distinguer deux cas :

- si $y = 0$ $(= Z_y)$

  Les impacts des tâches plus prioritaires sont des nombres entiers de fois leur temps d'exécution : il n'y a pas de fraction de temps d'exécution d'une autre requête qui repousse l'exécutionde la requête δ_i^k. Dès lors,

  $$\rho_i^k = C_i + \sum_{j<i} \left\lceil \frac{R_i^k + \rho_i^k - R_j^{h_j+1}}{T_j} \right\rceil C_j$$

  <u>Explication de la formule</u> :
  $R_i^k + \rho_i^k$ est l'instant de terminaison de la requête δ_i^k. On voudrait connaître le nombre de requêtes de τ_j arrivées avant cet instant de terminaison : c'est donné par $\left\lceil \frac{R_i^k + \rho_i^k}{T_j} \right\rceil$. Cependant, on ne veut que les requêtes qui ont un impact sur la requête δ_i^k. En fait, les requêtes arrivant strictement avant R_i^k n'en ont pas puisque tous les Z_j sont nuls. On va donc les retirer de la fraction plus haut et on obtient : $\left\lceil \frac{R_i^k + \rho_i^k - R_j^{h_j+1}}{T_j} \right\rceil$. A partir de là, pour obtenir le temps de réponse de δ_i^k, il suffit d'additionner le temps de calcul de la requête avec les temps de calcul des requêtes qui ont un impact sur son temps de réponse.

  Pour trouver le minimum, on utilisera une méthode déjà bien connue : on pratiquera par itérations (c'est une formule croissante) jusqu'à stabilisation ou dépassement de T_i (T_i est bien la deadline car nous sommes dans un cas late deadline).

- si $y > 0$ $(< Z_y)$

  La formule devient

  $$\rho_i^k = C_i + Z_y + \sum_{j<i} \left\lceil \frac{R_i^k + \rho_i^k - R_j^{k_j}}{T_j} \right\rceil C_j$$

  On distingue deux cas pour k_j

  - si $j > y \Longrightarrow k_j = h_j + 1$, c'est à dire que l'on considère les requêtes arrivées à partir de l'instant R_i^k

– si $j \leq y \Longrightarrow k_j = \left\lceil \frac{R_i^k + Z_y}{T_j} \right\rceil + 1$, c'est à dire que l'on considère les requêtes arrivées à partir de l'instant $R_i^k + Z_y$ car celles arrivées auparavant, soit étaient déjà terminées en R_i^k, soit ont repoussé la terminaison de $\rho_y^{h_y}$ (elles sont plus prioritaires) et sont déjà incorporées dans le terme Z_y (on remarque que plus j est petit, plus Z_j, s'il est non nul, diminue : voir figure 1.5.4). Si on prenait $h_j + 1$ comme précédemment, des requêtes seraient comptées deux fois.

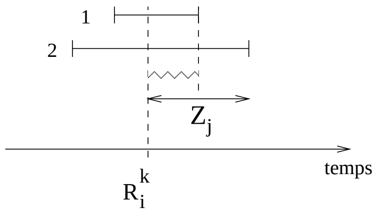

FIG. 1.13:

Pour calculer ρ_i^k, on pratique de nouveau par itérations, en commençant avec $C_i + Z_y$.

1.5.5 Autres cas intéressants

Jusqu'ici, nous avons essentiellement étudié des cas statiques, late deadline, synchrones, et parfois RMS.

Rmq : Un calcul comme celui du $lub_n\ U$ dépend du fait que l'on a une stratégie RMS ; alors que d'autres résultats, comme le calcul des temps de réponses, en sont indépendants.

Le cas synchrone est le pire cas : il fournit les temps de réponses les plus longs. Voyons d'autres cas plus "sympathiques" (du moins en apparence) :

– cas asynchrone : les instant de démarrage sont fixés mais les tâches ne démarrent pas toutes en même temps.

  Il se pourrait qu'elles se resynchronisent à un instant $t = O_i + k_i\ T_i\ \ \forall i$.
  On peut montrer que, si $\forall i < j : (O_i - O_j)\ \underline{\text{mod}}\ (pgcd\{T_i, T_j\}) = 0$, c'est-à-dire si l'offset relatif est exactement divisible par le plus grand commun diviseur des périodes, alors on aura une resynchronisation du système.

– cas offset free : l'instant de démarrage est non fixé et ils ne démarrent pas forcément tous en même temps.

  On a un phénomène semblable au cas asynchrone :
  Supposons que $\forall i < j : pgcd\{T_i, T_j\} = 1$: dans ce cas, offset free n'offre rien de plus que le cas synchrone car, quel que soit le choix des offsets, il y a resynchronisation du système.

MAIS, si ces conditions ne sont pas satisfaites, il n'y aura pas de resynchronisation globale du système (on pourrait avoir des resynchronisations partielles). On pourra alors espérer rendre schédulables des systèmes qui ne l'étaient pas dans le cas synchrone.

Exemple : considérons le système suivant, composé de 3 tâches :

$$\tau_1 \quad = \quad \{C_1 = 1, T_1 = D_1 = 12, O_1 = 10\}$$

$$\begin{aligned}\tau_2 &= \{C_2 = 6, T_2 = D_2 = 12, O_2 = 0\}\\ \tau_3 &= \{C_3 = 3, T_3 = D_3 = 8, O_3 = 0\}\end{aligned}$$

Son scheduling, avec les priorités $\tau_3 > \tau_2 > \tau_1$, est fourni par la figure 1.14 et respecte toutes ses deadlines (le schedule est périodique avec une période de 24 unités), alors qu'il est facile de voir que le cas synchrone, même avec RMS, n'est pas schedulable.

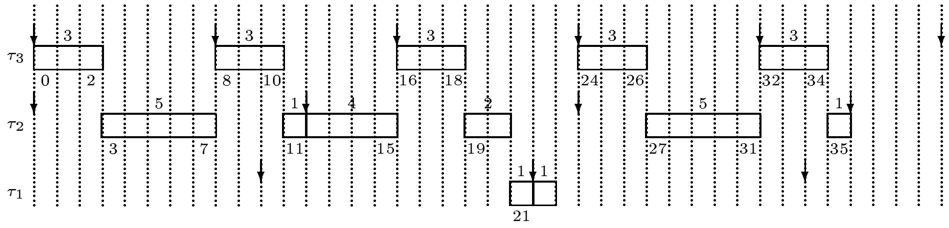

FIG. 1.14: Un scheduling asynchrone fructueux.

Ces cas plus "sympathiques" posent cependant pas mal de problèmes. Ainsi, comment pourrait-on vérifier que ces systèmes sont schédulables ? On ne peut plus se baser sur les résultats précédents, pour au moins trois raisons :

1. on n'est plus certain que RMS soit encore optimale. Il existe des systèmes asynchrones non schédulables qui le deviennent en ne choisissant plus RMS pour le choix des priorités. Par exemple, le système asynchrone considéré ci-dessus n'est pas schedulable avec RMS, alors qu'il l'est avec les priorités utilisées dans la figure 1.14.
2. les résultats obtenus pour lub_nU ne sont plus valides.
3. le pire cas n'est plus forcément donné par la première requête.

Par contre, le fait que le système est périodique d'hyperpériode P à partir de O'_n reste vrai. Dès lors, le seul moyen de vérifier qu'un système est schédulable est de calculer tous les temps de réponses de l'hyperpériode, en comptant aussi la mise en régime.

Comment calculer ces temps de réponses ? Le raisonnement est semblable à ce que l'on a déjà fait précédemment (cfr. les calculs avec le Z_y) à une exception près : il faudra tenir compte des offsets lors des calculs.

$$\Longrightarrow R_i^k = (k-1)T_i + O_i$$

$$\Longrightarrow h_j = \left\lceil \frac{(R_i^k - O_j)^+}{T_j} \right\rceil$$

Idem pour les k_j.

Mis à part cela, la démarche reste la même.

Nous avons donc un moyen de calculer les temps de réponses mais c'est très lourd car la méthode est exponentielle (l'hyperpériode P est exponentielle en le nombre de tâches). Cependant, l'utilisateur peut souvent jouer sur les durées des périodes afin de diminuer l'hyperpériode.
Exemple : si on a une période de 10 et une période de 11 dans le système, les calculs nous donneront un facteur de 110. Par contre, si nous parvenons à ramener la période de 11 à 10, le ppcm des deux ne vaudra plus que 10. Nous aurons gagné un facteur 11 !

Dans le cas offset free, ça se gâte encore plus. RMS n'est plus optimale et il n'existe pas de moyen optimal pour choisir les offsets. On pourra juste éviter la resynchronisation globale.

Rmq : si RMS ne fonctionne pas, on pourra essayer d'autres assignations de priorités, combinées avec tous les essais d'offsets (non-équivalents) possibles : on se retrouve de nouveau dans un cas (encore plus) exponentiel.

1.6 Cas synchrone, statique, avec general deadlines

Le pire temps de réponse reste celui où toutes les requêtes commencent en même temps (le raisonnement que nous avons mené dans le cas synchrone avec late deadlines ne dépendait en fait pas de la deadline, pourvu qu'il n'y ait jamais 2 requêtes actives simultanément pour une même tâche). De plus, les calculs des temps de réponse restent les mêmes que dans le cas late deadline, synchrone ou non, à la différence que pour chaque tâche on ne doit pas dépasser la deadline ; cela modifie les tests d'arrêts dans les itérations : on s'arrête soit parce qu'il y a stabilisation, soit parce qu'il y a dépassement de la deadline (et pas de la période : le système est alors non schédulable).

Existe-il une autre assignation de priorités que RMS qui soit optimale ? OUI ! on prend DMS (Deadline Monotonic Scheduling) c'est-à-dire que la priorité est en fonction de la deadline (la plus haute priorité est donnée à la tâche qui a la plus petite deadline) : le raisonnement utilisé pour justifier RMS peut être réutilisé tel quel pour justifier DMS.

Rmq : dans le cas late deadline ($T_i = D_i$), DMS est équivalent à RMS.

Tous les raisonnements sur le lub U tombent, car ils dépendent du fait que pour toutes les tâches $T_i = D_i$ or ici ce n'est plus vrai.

1.7 Cas asynchrone ou offset free, statique, avec general deadlines

Ici aussi, on peut assez facilement vérifier la validité ou non des raisonnements et résultats précédents ; on obtient :

- Il n'y a pas de stratégies optimales connues pour fixer les origines (dans le cas offset free) et les priorités.
- Les critères de resynchronisation, $\forall i < j : (O_i - O_j) \underline{\text{mod}} (pgcd\{T_i, T_j\}) = 0$ (cas asynchrone) ou $\forall i < j : pgcd\{T_i, T_j\} = 1$ (cas offset free), restent valables.
- Les temps de réponse se calculent toujours de la même manière, en tenant compte des O_i. On veillera à introduire la deadline dans le test d'arrêt à tous les coups, pas seulement pour la première requête.
- Il est facile de trouver des exemples où le cas synchrone n'est pas schédulable alors qu'en asynchrone, ça l'est.

  Exemple :
  Considérons 2 tâches, avec $T_1 = T_2 = 2; C_1 = C_2 = D_1 = D_2 = 1$.
  Le cas synchrone n'est pas schédulable. Par contre, si on prend $O_1 = 0$ et $O_2 = 1$, cela fonctionne.

1.8 Priorités statiques avec arbitrary deadlines

Pour illustrer l'intérêt (et les problèmes comportementaux) des systèmes avec arbitrary deadlines, considérons le système suivant, composé de 2 tâches :

$$\begin{aligned}\tau_1 &= \{C_1 = 52, T_1 = 100, D_1 = 110, O_1 = 0\} \\ \tau_2 &= \{C_2 = 52, T_2 = 140, D_2 = 154, O_2 = 0\}\end{aligned}$$

Son scheduling, avec les priorités $\tau_2 > \tau_1$, est illustré par la figure 1.15 : toutes les deadlines sont respectées, alors que ce ne serait pas vrai dans le cas late deadline (en fait, dès que $U \leq 1$, avec des deadlines suffisamment grandes le système est toujours schedulable).

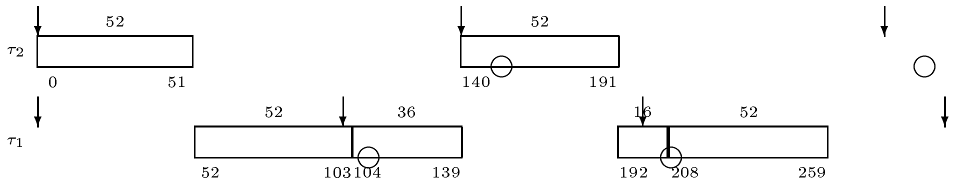

FIG. 1.15: Un schedule avec des deadlines arbitraires.

Ces systèmes sont beaucoup plus difficiles à analyser, car on peut montrer que :

- même dans le cas synchrone, DMS n'est pas toujours une affectation de priorités optimale, comme en témoigne l'exemple ci-dessus : on peut vérifier que, avec les priorités $\tau_1 > \tau_2$, telles que suggérées par DMS, le système n'est pas schedulable (la première requête de τ_2 rate sa deadline).
- Le cas synchrone est toujours le pire cas, mais c'est plus difficile à prouver (le raisonnement est du même type que dans le cas late deadline cependant, mais en tenant compte de ce qu'une requête d'une tâche peut être repoussée par les requêtes précédentes de la même tâche).
- Les pires temps de réponse apparaissent bien dans le cas synchrone, mais ne correspondent plus nécessairement à la première requête de chaque tâche, car on pourrait avoir une requête qui dépasse sa deadline et repousse d'autant la requête suivante de la même tâche. Par exemple, dans le cas (synchrone) illustré par la figure 1.15, la deuxième requête de τ_1 dure 108 unités alors que la première ne durait que 104 unités !
- On peut cependant montrer que le pire temps de réponse pour une tâche τ_i apparaît dans le cas synchrone avant le premier *idle point* de niveau i. On appelle idle point de niveau i un instant t ($t > 0$) tel que toutes les requêtes de niveau inférieur ou égal à τ_i qui ont démarré strictement avant t sont terminées au plus tard en t (idle car tout ce qui est avant est terminé). On notera par λ_i le premier idle point de niveau i. Dans le cas synchrone considéré ici, on peut voir que λ_i est la plus petite solution strictement positive de l'équation :

  $$\lambda_i = \sum_{j \leq i} C_j \left\lceil \frac{\lambda_i}{T_j} \right\rceil$$

  et donc on peut l'obtenir à nouveau par un procédé itératif :

  $$\begin{cases} \lambda_i \longleftarrow \sum_{j \leq i} C_j & \text{(point de départ)} \\ \lambda_i \longleftarrow \sum_{j \leq i} C_j \left\lceil \frac{\lambda_i}{T_j} \right\rceil & \text{(itération)} \end{cases}$$

On itère jusqu'à stabilisation (on est certain que cela se stabilisera si $U \leq 1$, ce qu'on supposera puisque sinon on sait que le système n'est pas schedulable).

- La détermination des temps de réponses se fait comme précédemment, sauf que le calcul de Z_y doit prendre en compte tous les Z_j, jusqu'à i compris :

$$Z_y = max_{j \leq i} \left\{ (R_j^{h_j} + \rho_j^{h_j} - R_i^k)^+ \right\}.$$

On peut remarquer que les calculs de temps de réponse dans le cas de deadlines arbitraires peuvent s'appliquer aussi au calcul des temps de réponse avec des deadlines soft (que les deadlines soient late, générales ou arbitraires), si on laisse les processus se prolonger au-delà de leur deadline (et donc éventuellement au-delà de leur période) : il faut simplement supprimer le test d'arrêt sur deadline. Cela peut même s'appliquer au cas où $U > 1$, mais alors il se peut que le calcul diverge : soit i le premier indice tel que $\sum_{j=1}^{i-1} \frac{C_j}{T_j} \geq 1$, à partir de τ_i on obtiendra finalement (parfois tout de suite) des temps de réponses infinis.

Exemple : considérons 2 tâches avec $C_1 = 1 = T_1 = D_1$, $O_1 = O_2 = 0$, $C_2 = 1$ et $T_2 = 2 = D_2$. La première requête de la deuxième tâche ne sera jamais prise en charge (car la première tâche occupe constamment le processeur). Le temps de réponse de la première requête de la seconde tâche sera donc infini.

- Jusqu'où faut-il simuler/calculer les temps de réponse pour savoir si un système est schédulable ou non ?

Dans le cas synchrone, nous savons déjà qu'il suffit d'aller jusqu'en $\lambda_n = max_{i=1,\ldots n}\{\lambda_i\}$. Dans le cas asynchrone (et donc aussi dans chacun des cas d'offsets à considérer dans le cas offset free), on peut montrer qu'il suffit d'aller jusqu'en $O_{max} + 2P$, où O_{max} désigne le plus grand des offsets. Ceci peut s'interpréter comme suit :

  - Avant O_{max}, toutes les tâches ne sont pas encore apparues dans le système.
  - A partir de O_{max}, on entre dans la mise en régime, c'est-à-dire que l'impact des requêtes précédentes ne reflète pas encore toujours tout ce qu'il faut.
  - Enfin, en $O_{max} + P$ tout est mis en place. Le système commence à être périodique de période P (si $U \leq 1$). D'où, en simulant/calculant les temps de réponse jusqu'en $O_{max} + 2P$, on saura, avec certitude, si le système est schédulable ou non.

L'intervalle $[0, O_{max} + 2P]$ sur lequel on vérifie si un système est schédulable est également appelé intervalle de feasibilité (feasibility en anglais).

1.9 Systèmes à Priorités Dynamiques

Outre le fait que des priorités dynamiques sont généralement plus lourdes à gérer et à calculer, et donc allongent les switching times (qu'on continuera pourtant à négliger, mais cette approximation peut devenir criticable si les formules donnant les priorités sont trop compliquées ; il se peut aussi que le dynamisme des priorités augmente la fréquence des switches), leur introduction conduit à des situations assez "sympathiques" à certains égards, mais beaucoup moins à d'autres égards.

1.9.1 Les avantages

- On connaît des stratégies dynamiques optimales pour l'attribution des priorités, qui marchent de façon tout à fait générale, que le système soit synchrone ou non (même offset free), que les deadlines soient late, générales ou arbitraires. On en connaît même de deux types, avec des comportements assez différents bien qu'elles soient équivalentes du point de vue de la schédulabilité. Rappelons que dans le cas statique, une seule stratégie optimale est connue, et uniquement dans le cas synchrone non arbitraire.
- Dans le cas late deadline (synchrone ou non), on a une condition nécessaire et suffisante de schédulabilité extrèmement simple. Rappelons que dans le cas statique synchrone avec late deadline, on connaît une condition suffisante assez simple, une condition nécessaire assez simple, mais les tests nécessaires et suffisants sont plus longs.
- Le cas synchrone reste le pire cas.
- On retrouve le même genre d'hyperpériode que dans le cas statique (l'hyperpériode est toujours donnée par $P = ppcm_{i=1,\ldots n}\{T_1\}$, ainsi que les mêmes intervalles de feasibilité : $[0, \lambda]$ dans le cas synchrone et $[0, O_{max} + 2P]$ dans le cas asynchrone.

1.9.2 Les inconvénients

- Le cas synchrone reste le pire cas, mais il n'est plus vrai que la première requête donne le plus grand temps de réponse, même dans le cas general deadline ; elle n'est même pas toujours dans la première busy period.
- Mieux que ça ! Ce n'est pas parce que le cas synchrone est le pire cas, que celui-ci fournira les temps de réponse les plus longs.
- On peut donc déjà se dire que, sauf dans le cas late deadline, il faudra beaucoup de calculs pour savoir si un système est schédulable ou pas.

1.9.3 DDS

La première stratégie que l'on va examiner porte le nom de DDS pour Deadline Driven Scheduling (elle est aussi connue dans la littérature sous le nom Earliest Deadline Scheduling, EDS, ou Earliest Deadline First, EDF). Cette stratégie donne le processeur à la requête active (donc démarrée mais pas encore terminée) qui a la deadline la plus proche (si on considère des deadline softs, elles peuvent se situer dans le passé, et il vaut mieux alors parler de la plus petite deadline). Comme on le verra, DDS est une stratégie optimale.

Mais avant cela, illustrons les comportements parfois bizarres de cette technique, par ailleurs assez naturelle.

Considérons le système synchrone suivant, avec 2 tâches :

$$\begin{aligned}\tau_1 &= \{C_1 = 2, D_1 = 3, T_1 = 4\}\\ \tau_2 &= \{C_2 = 3, D_2 = 6, T_2 = 7\}.\end{aligned}$$

Son scheduling par DDS est illustré à la figure 1.16. On peut y remarquer que la première busy period se termine à l'instant 7 (toutes les requêtes démarrées auparavant sont alors terminées), le premier temps de réponse pour τ_2 est de 5, tandis que le quatrième est de 6 et il démarre en 21. Donc, non seulement le premier temps de réponse n'est pas le plus long, mais le plus long peut arriver au delà de la première busy period.

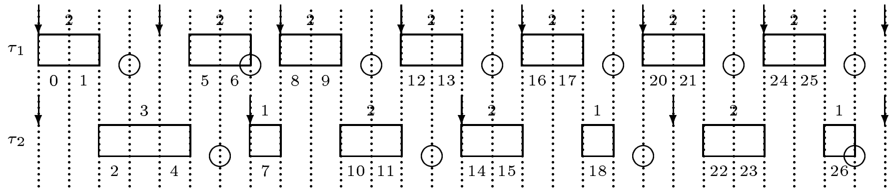

FIG. 1.16: Un scheduling bizarre.

Un autre exemple intéressant est illustré par les figures 1.17 et 1.18, et correspond au système offset free à 3 tâches :

$$\begin{aligned}
\tau_1 &= \{C_1 = 1, D_1 = 3, T_1 = 4\} \\
\tau_2 &= \{C_2 = 2, D_2 = 7, T_2 = 8\} \\
\tau_3 &= \{C_3 = 1, D_3 = 6, T_3 = 12\}.
\end{aligned}$$

dans le cas synchrone, le plus long temps de réponse pour τ_3 est de 2 unités, alors que dans le cas asynchrone, avec $O_1 = 0$, $O_2 = 2$ et $O_3 = 3$, il atteint 3 unités, montrant bien que le pire temps de réponse n'est pas toujours obtenu dans le cas synchrone, même si celui-ci reste bien le pire cas du point de vue de la feasibilité du scheduling.

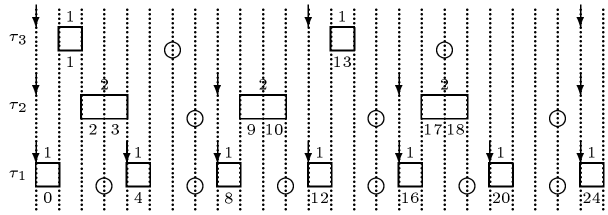

FIG. 1.17: Un schedule synchrone (à partir de $t = 24$ il se répète).

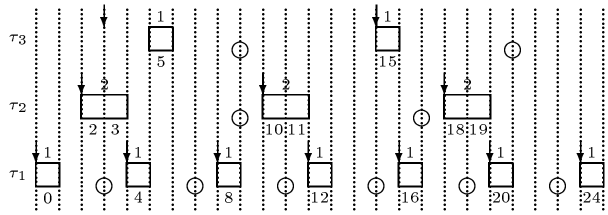

FIG. 1.18: Un schedule asynchrone (à partir de $t = 24$ il se répète).

Afin d'étudier et de mieux comprendre les propriétés des schedules DDS, il est intéressant d'introduire la fonction $\eta_i(t, t')$, où $0 \leq t < t'$ désignent 2 instants croissants et i désigne un numéro de tâche. Cette fonction donne le nombre de requêtes de la tâche τ_i, apparues après l'instant t et qui ont une deadline avant ou en t' :

$$\eta_i(t, t') = \text{ nombre de } k \text{ tels que } \left\{ \begin{array}{l} O_i + (k-1)T_i \geq t \\ O_i + (k-1)T + D_i \leq t' \end{array} \right.$$

C'est donc le nombre de valeurs (entières) de k telles que la k^e requête de τ_i n'arrive pas plus tôt que t **et** telles que la deadline est en t' ou avant.

On peut remarquer que, si le système est schédulable, $\sum_i \eta_i(t,t')C_i \leq t' - t$, sinon cela signifie que l'on n'a certainement pas assez de temps processeur pour satisfaire les requêtes concernées.
Cela nous fournit donc une condition nécessaire de schédulabilité, indépendamment d'ailleurs de la stratégie utilisée pour l'attribution des priorités (ce n'est pas spécifique à DDS). Par contre, nous allons voir que, dans le cas DDS, la condition ci-dessus est nécessaire ET suffisante.

Démonstration :
Que la condition est nécessaire a déjà été prouvé ; il reste donc à prouver qu'elle est aussi suffisante. Pour cela on va montrer la contraposée, c'est-à-dire que, si ce n'est pas schédulable, alors il existe une paire (t, t') telle que $\sum_i \eta_i(t,t')C_i > t' - t$.

Supposons un système non schédulable,

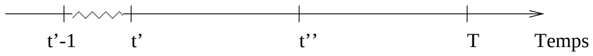

FIG. 1.19: Illustration d'un système non schédulable

t'' est le premier instant où l'on rate une deadline. t' est le dernier instant avant t'' tel que

1. soit il ne se passe rien entre $t' - 1$ et t' : le système est idle au moins depuis $t' - 1$. (nous sommes dans des stratégies expédiantes donc si l'on avait eu une requête, on lui aurait donné le processeur).
2. soit toutes les requêtes actives en $t' - 1$ ont une deadline strictement supérieure à t'', et en particulier celle qui a reçu le processeur.

On peut voir que $t'' > t' \geq O_{min} \geq 0$, où O_{min} désigne le plus petit offset. Que se passe-t-il entre t' et t'' ?

- Le processeur est constamment utilisé, car t' est soit la fin du dernier idle slot, soit plus tard.
- toutes les requêtes qui reçoivent le processeur entre t' et t'' n'ont pas de deadline supérieure à t".

En plus de cela, $t'' - t' > 0$ car en t'' on rate une deadline $\Longrightarrow$ en $t'' - 1$, ou bien la requête qui a raté sa deadline était active et elle utilisait le processeur, ou alors le processeur était utilisé par une autre requête qui avait également une deadline en t'' mais malheureusement elle n'était pas la seule. $\Longrightarrow$ t' est au plus tard en $t'' - 1$ (de par sa définition), nous avons donc bien $t'' - t' > 0$.

$\Longrightarrow \sum_i \eta_i(t',t'')\ C_i > t'' - t'$ car on a raté une deadline : il n'y avait pas assez de temps pour exécuter toutes les requêtes répertoriées par $\eta_i(t',t'')$.

CQFD

Première propriété : On peut remarquer que DDS possède des ambiguïtés : si deux requêtes ont la même deadline, le système ne sait pas laquelle il doit scheduler. Cependant, cela ne change rien car η_i ne dépend pas de la manière dont on a résolu les ambiguïtés. Ce n'est pas important pour la schédulabilité mais cela pourrait avoir un impact sur le type de scheduling. Cela pourrait même conduire à un scheduling non périodique.

Exemple :

$$C_1 = 1, T_1 = D_1 = 2, O_1 = 0$$
$$C_2 = 1, T_2 = D_2 = 2, O_2 = 0$$

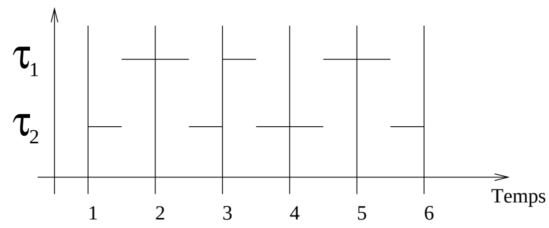

FIG. 1.20: Exemple d'ambiguïtés

Il y a ambiguïté toutes les 2 unités de temps; supposons que l'on attribue le CPU à la tâche τ_1 ssi le numéro de sa requête est un nombre premier : le schedule obtenu est non périodique (car la suite des nombres premiers est non périodique).

Pour que ce soit périodique de période P, il faut donner la priorité à la même requête en T et en $T + i\ P$ pour tout $i \geq 1$ et tout instant T où se produit une ambiguïté (elle se reproduira en $T + i\ P$).

Seconde propriété : DDS est une technique optimale car la condition est nécessaire et suffisante (on ne le répètera jamais assez).

Troisième propriété : Le cas synchrone est toujours le pire cas. Cela résulte de l'observation que si on désigne par η'_i la fonction η_i pour le système synchrone correspondant, on a que pour tout $t < t'$:

$$\eta_i(t, t') \leq \eta'_i(0, t' - t).$$

En effet, pour avoir un $\eta_i()$ le plus grand possible pour un intervalle donné, il suffit d'imposer que toutes les requêtes commencent au début de l'intervalle.

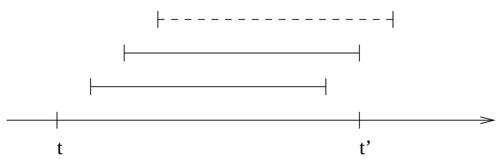

FIG. 1.21: Décalage d'une requête vers la droite

Ainsi, la requête en pointillés sur la figure 1.21 n'apparaît pas dans $\eta_i()$, car sa deadline est supérieure à t', mais en décalant l'instant d'arrivée des requêtes vers la gauche, on la prend.

Le raisonnement n'est pas lié aux temps de réponse. Ceci explique le fait que le cas synchrone, bien qu'étant le pire cas, ne fournit pas forcément les pires temps de réponses.

Quatrième propriété : Si une requête rate sa deadline dans un système synchrone sous DDS, il n'y a pas d'idle point auparavant.

A nouveau, on appelle *idle point* un instant (> 0) tel que toutes les requêtes apparues strictement auparavant sont terminées au plus tard en cet instant (simplement, ici, contrairement à ce que nous avions fait dans le cas de priorités statiques, il ne faut plus préciser de niveau).

Pour le prouver, nous allons procéder par l'absurde :
Supposons que la deadline non respectée (dans le cas de deadlines soft, il faudrait préciser que l'on ne regarde que le premier dépassement de deadline) est située en D, après le premier idle point ! Désignons par ι le dernier idle point situé avant le dépassement de

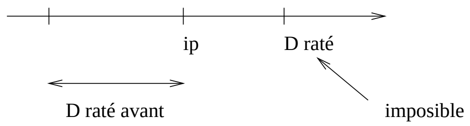

FIG. 1.22: Cas impossible de ratage de deadline

deadline ($0 < \iota < D$), et par t le dernier instant situé entre ι et D ($\iota \leq t < D$) tel que en t ou bien toutes les requêtes précédentes sont terminées ou bien on vient de servir une requête dont la deadline est au delà de D. Comme précédemment, entre t et D on ne sert que des requêtes arrivées après t avec une deadline au plus tard en D, et on rate la deadline de sorte que $\sum_{i=1}^{n} \eta_i(t,D) > D - t$. Or on vient de voir que $\eta_i(t,D) \leq \eta'_i(0, D-t)$. Donc on doit avoir aussi $\sum_{i=1}^{n} \eta'_i(0, D-t) > D - t$ et on doit avoir un dépassement de deadline dans le cas synchrone au plus tard en $D - t < D$, de sorte que D ne pouvait pas être le (premier) dépassement de deadline dans le cas synchrone.

1.9.4 Une autre technique optimale : Least Laxity First

On appelle *laxité*[1] (ou *latitude*, ou *temps de mou*) d'une requête à un instant donné, la différence entre le temps qui nous sépare de la deadline et le temps de calcul qu'il reste à effectuer pour la requête (si la requête est apparue en R_i^k, qu'elle a déjà reçu x unités de CPU et qu'on est en t, sa laxité est $R_i^k + D_i - t - C_i + x$). Elle représente le "mou", le battement qui nous reste pour exécuter la requête. Dans cette technique LLF, on attribue le processeur à la requête qui a la plus petite laxité.

Rmq : "Least Laxity signifie le moins de mou et ce n'est pas le mou qu'on donnait au chat avant Whiskas".

M R.Devillers, le 14/10/1999

On peut montrer que cette technique est également optimale. Ces deux stratégies (DDS et LLF) sont équivalentes d'un point de vue feasibilité mais elles ont un comportement différent.

[1]En anglais, laxity signifie "relâchement des moeurs".

Exemple :

$$C_1 = C_2 = 10 \text{ et } O_1 = 0$$
$$T_1 = D_1 = T_2 = D_2 = 20 \text{ et } O_2 = 0$$

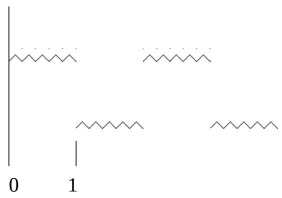

FIG. 1.23: Un exemple avec une stratégie least laxity

En 0 **:** la laxité de $\tau_1 = 10$ et celle de $\tau_2 = 10$

En 1 **:** la laxité de $\tau_1 = 10$ et celle de $\tau_2 = 9$

En 2 **:** la laxité de $\tau_1 = 9$ et celle de $\tau_2 = 9$

Dans l'exemple ci-dessus, si une ambiguïté apparaît, on accorde le CPU à la requête de la tâche qui a le plus petit numéro. Quand une tâche ne travaille pas, sa laxité diminue, alors que quand elle s'exécute sa laxité reste inchangée. Pour chaque time slot, on change de requête.

On peut remarquer que, à ce point de vue, cette technique est moins intéressante que DDS car on n'arrête pas de switcher. De plus, les priorités d'une même requête évoluent au cours du temps (cfr. l'exemple ci-dessus), ce qui n'est pas le cas de DDS. Par contre, comme en DDS, par rapport au cas de priorités statiques avec arbitrary deadlines, nous avons un problème en moins. En effet, deux requêtes d'une même tâche n'auront jamais la même priorité. Mieux que ça : celle qui aura la plus haute priorité sera bien la plus ancienne (petit rappel : dans le cas statique avec arbitrary deadline, on avait ajouté une règle disant qu'on donne le contrôle à la requête la plus ancienne en cas d'ambiguïté).

1.9.5 Schedulabilité de systèmes late deadline sous DDS.

Nous avions déjà une condition nécessaire de schédulabilité, $U \leq 1$, dans les cas synchrones ou asynchrones. Si on a late deadline et DDS, ce critère devient nécessaire et suffisant (et d'un point de vue schédulabilité, cela fonctionne aussi bien en synchrone qu'en asynchrone).

Nous allons observer le système dans le cas synchrone, car c'est le pire cas et donc si la condition est nécessaire et suffisante dans le cas synchrone, elle le sera aussi dans le cas asynchrone (où on sait déja qu'elle est nécessaire).

Procédons par l'absurde :

On va supposer que $U \leq 1$ et que malgré tout le système est non schédulable. Cela signifie que nous avons un overflow quelque part, en T par exemple ($T > 0$). On doit avoir que $\sum_{i=1}^{n} \eta_i(0,T) > T$ car on sait déja que, un peu avant le dépassement de deadline, il y a un instant t tel que $\sum_{i=1}^{n} \eta_i(t,T) > T - t$; or on a aussi vu que dans le cas synchrone $\eta_i(t',t'') \leq \eta_i(0,t''-t')$ pour tous i, t' et t'' ; dès lors, $t = 0$ car sinon on doit avoir un dépassement de deadline au plus tard en $T - t$. Or, dans le cas synchrone avec late deadline, on peut voir que $\eta_i(0,T) = \left\lfloor \frac{T}{T_i} \right\rfloor$.

$$\Longrightarrow \sum_i C_i \lfloor \frac{T}{T_i} \rfloor > T \Longleftrightarrow \sum_i \frac{C_i}{T} \lfloor \frac{T}{T_i} \rfloor > 1 \Longrightarrow U = \sum_i \frac{C_i}{T} \frac{T}{T_i} \geq \sum_i \frac{C_i}{T} \left\lfloor \frac{T}{T_i} \right\rfloor > 1 \Longrightarrow U > 1$$

Ce qui est bien une contradiction.

CQFD

Rmq : le même critère fonctionne aussi pour LLF, puisque DDS et LLF sont équivalents au point de vue de la schedulabilité.

1.9.6 Schedulabilité de systèmes avec general/arbitrary deadlines sous DDS.

Si toutes les deadlines ne sont pas late, on peut (après avoir vérifié que $U \leq 1$) calculer les temps de réponses sur un intervalle de feasibilité, c'est à dire un intervalle tel que si aucune deadline n'y est ratée, aucune deadline ne sera jamais ratée.

Dans le cas synchrone : On a déjà vu que, si il y a un dépassement de deadline, c'est nécessairement avant le premier idle point L. Pour déterminer ce dernier, il suffit d'exprimer qu'entre 0 et L il n'y a pas de slot libre et que c'est exactement le temps qu'il faut pour exécuter toutes les requêtes précédentes.

$$L = \sum_{i=1}^{n} C_i \left\lceil \frac{L}{T_i} \right\rceil$$

car $\lceil \frac{L}{T_i} \rceil$ donne le nombre de requêtes de τ_i qui, dans le cas synchrone, ont démarré strictement avant L.

Comme d'habitude, on recherche la plus petite solution strictement positive et on peut procéder par itération : on initialise L à $\sum_{i=1}^{n} C_i$ car toutes les premières requêtes doivent être terminées, puis on itère jusqu'à stabilisation (qui se produira au plus tard en P) :

$$L \leftarrow \sum_{i=1}^{n} C_i \left\lceil \frac{L}{T_i} \right\rceil$$

Dans le cas asynchrone : On peut montrer que $[0, O_{max} + 2\ P]$ est à nouveau un intervalle de feasibilité. On pourra à nouveau souvent agir sur les périodes T_i pour que l'hyperpériode P ne soit pas trop grande. Cependant, il reste O_{max} qui peut être gigantesque.

On peut améliorer cette situation en remarquant que les premières requêtes de chaque tâche peuvent être supprimées sans que cela ne modifie le fonctionnement en régime du système ; on peut donc ne garder pour chaque tâche que les requêtes à partir de la dernière qui précède O_{max}, ou même n'importe quel $O' \geq O_{max}$, l'intérêt de ce dernier choix étant que l'on peut parfois trouver un tel O' de façon à ce que la dernière requête de chaque tâche avant O' soit assez proche de O', plus en tout cas que pour O_{max}. Cela revient à modifier les offsets en posant, pour tout i,

$$O'_i = max_j \left\{ R_i^j \leq O' \right\} = R_i^{1+\left\lfloor \frac{O'-O_i}{T_i} \right\rfloor}.$$

Dans les cas de resynchronisation, cela nous ramènera d'ailleurs au cas synchrone. Sinon, on observera le nouveau système sur l'intervalle de feasibilité $[O'_{min}, O'_{max} + 2P]$.

Rmq : "Le croupion doit être le plus petit possible".

M R.Devillers, le 21/10/99

Calcul du temps de réponse :

Il nous reste à montrer comment on peut calculer les temps de réponse dans un tel intervalle de feasibilité. On va travailler dans le cadre le plus général possible, à savoir le cas asynchrone avec arbitrary deadlines ; les formules seront en fait valables, que l'on suppose qu'il n'y a pas de dépassement de deadlines avant la fin de la réponse ou que l'on travaille avec des deadlines softs et qu'on laisse les requêtes se poursuivre jusqu'à leur terminaison ; par contre cela ne marchera pas si on tue les requêtes dès qu'elles dépassent leur deadline, ou dans le cas late/general deadline si on ne la tue que lorsqu'une nouvelle requête du même type arrive ; cela ne marche pas non plus avec une stratégie LLF, même si elle est équivalente en terme de scheduling (mais pas de temps de réponse), car on va à nouveau se baser sur le fait que si une requête prioritaire apparaît pendant le réponse, on est repoussé de tout un temps de calcul.

Soit ρ_i^k le temps de réponse de la k^e requête de la tâche τ_i. Comme dans le cas statique, on va obtenir une formule du type

$$\rho_i^k = C_i + \pi_i^k + \sum_{j \neq i} n_{i,j}^k(\rho_i^k)\ C_j$$

où $n_{i,j}^k()$ donne le nombre de requêtes plus prioritaires de τ_j qui se sont exécutées certainement entièrement pendant l'intervalle ρ_i^k, alors que π_i^k donne le décalage induit par les requêtes qui peuvent avoir un impact partiel seulement. Ce dernier terme dépendra de certains temps de réponse précédents, que l'on supposera donc avoir calculé auparavant pour éviter des calculs récursifs. A nouveau, il faudra déterminer soigneusement pour chaque tâche quelle est la première requête qui peut avoir un impact partiel, quelles requêtes ont certainement un impact complet, et veiller à ne pas compter deux fois un même impact. Imaginons :

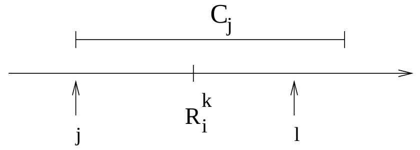

FIG. 1.24:

Si l est plus prioritaire que j, cela va augmenter le temps de réponse de j, et donc apparaître dans π_i^k. Il ne faudra donc pas le compter dans $n_{i,j}^k$.

Détaillons à présent les calculs de π_i^k et $n_{i,j}^k(\rho_i^k)$:

pour π_i^k : Aucune requête de τ_j n'aura un impact partiel sur le calcul du temps de réponse (cfr. figure 1.24) si

$$\begin{cases} O_j \geq R_i^k \text{ (pas de requête arrivée avant)} & \\ \text{ou } O_j < R_i^k \text{ et } O_j + D_j > R_i^k + D_i & \text{(la première requête a déjà} \\ \text{ou } O_j < R_i^k \text{ et } O_j + D_j = R_i^k + D_i \text{ et } j > i & \text{une trop faible priorité)} \end{cases}$$

en supposant que, si deux requêtes ont la même deadline, c'est celle de la tâche qui a le plus petit indice qui sera considérée prioritaire.

Sinon, $\left\lceil \frac{R_i^k - O_j}{T_j} \right\rceil$ fournit le numéro de la dernière requête de τ_j apparaissant strictement avant R_i^k, mais ce n'est peut-être pas celle qui nous intéresse car elle peut avoir une trop faible priorité. Le numéro p de la dernière requête de τ_j avec une priorité suffisante est caractérisé par $O_j + (p-1)T_j + D_j < R_i^k + D_i$ si $j \geq i$, et par $O_j + (p-1)T_j + D_j \leq R_i^k + D_i$ si $j < i$. Dès lors :

$$p = \begin{cases} \left\lfloor \frac{R_i^k + D_i - O_j - D_j}{T_j} \right\rfloor + 1 & \text{si } j < i \\ \left\lceil \frac{R_i^k + D_i - O_j - D_j}{T_j} \right\rceil & \text{si } i \leq j \end{cases}$$

et l'on prendra donc pour le numéro m_j de la dernière requête de τ_j qui peut avoir un impact partiel (il peut y en avoir d'autres avant, mais leur impact sera alors inclus dans celui de la ${m_j}^e$) :

$$m_j = \min\left\{ \left\lceil \frac{R_i^k - O_j}{T_j} \right\rceil, p \right\}$$

Son impact sera $\pi_j = (R_j^{m_j} + \rho_j^{m_j} - R_i^k)^+$, c'est à dire le temps utilisé au delà de R_i^k pour terminer la requête m_j. On posera $\pi_j = 0$ si on n'était pas dans les conditions pour avoir un impact partiel.

- Si tous les π_j sont égaux à zéro, les requêtes arrivées avant R_i^k n'ont pas d'impact sur $\rho_i^k \Longrightarrow \pi_i^k = 0$.
- Si on a des π_j non nuls, on prendra le plus grand (car de cette manière, on est sûr que les plus petits seront comptés dans le plus grand).

$$\pi_i^k = max_{j=1}^n\{\pi_j\}$$

On désignera par z l'indice du plus grand π_j (on ne peut pas avoir deux $\pi_j \neq 0$ identiques car sinon on a deux requêtes qui se terminent en même temps).

Il reste à déterminer les $n_{i,j}^k$. Une fois encore, on fera attention à ne pas compter dans les $n_{i,j}^k$ les requêtes déjà présentes dans π_i^k.

- Si $\pi_i^k = 0$: $x = \left\lceil \frac{(R_i^k - O_j)^+}{T_j} \right\rceil + 1$ est le numéro de la première requête de τ_j qui peut avoir un impact entier sur ρ_i^k, et elle n'est certainement pas comptabilisée dans π_i^k ; il ne faut cependant considérer cette requête, et les suivantes, que si elles sont plus prioritaires que celle dont on calcule le temps de réponse et qu'elles sont arrivées strictement avant $R_i^k + \rho_i^k$; en conséquence, en rassemblant les formules déjà obtenues précédemment, on obtient :

$$n_j(\rho_i^k) = \begin{cases} \min\left\{ \left\lceil \frac{(R_i^k+\rho_i^k-R_j^x)^+}{T_j} \right\rceil, \left\lfloor \frac{(R_i^k+D_i-R_j^x+T_j-D_j)^+}{T_j} \right\rfloor \right\} & \text{si } j < i \\ \min\left\{ \left\lceil \frac{(R_i^k+\rho_i^k-R_j^x)^+}{T_j} \right\rceil, \left\lceil \frac{(R_i^k+D_i-R_j^x-D_j)^+}{T_j} \right\rceil \right\} & \text{si } j > i \end{cases}$$

– Si $\pi_i^k > 0$: Quelles sont les requêtes qui interviennent dans le calcul de $n_{i,j}^k()$? Les requêtes qui arrivent après ou en $R_i^k + \pi_i^k$ ne sont pas comptées dans le calcul du π_i^k, mais certaines de celles arrivées dans l'intervalle $[R_i^k, R_i^k + \pi)$ peuvent ne pas l'être non plus.

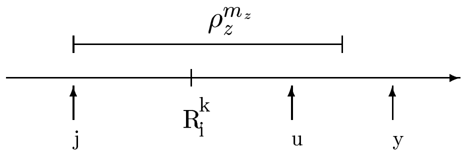

FIG. 1.25:

Pour la requête arrivant en u, il faut faire attention : si elle est plus prioritaire que celle en j, ce sera compté dans π_i^k ; par contre, si elle est moins prioritaire que celle en j mais plus que celle en R_i^k, alors ce sera compté dans $n_{i,j}^k$.

"C'est très agréable de ne pas se mordre la queue."

M R.Devillers, le 21/10/99

$$x = \left\lceil \frac{(R_i^k - O_j)^+}{T_j} \right\rceil + 1$$

désigne comme précédemment le numéro de la première requête de τ_j qui peut avoir un impact entier sur ρ_i^k, tandis que

$$y = \left\lceil \frac{(R_i^k + \pi_i^k - O_j)^+}{T_j} \right\rceil + 1$$

désigne le numéro de la première requête de τ_j qui arrive après ou en $R_i^k + \pi_i^k = R_z^{m_z} + \rho_z^{m_z}$: elle n'est certainement pas incluse dans le terme π_i^k.

Pour déterminer s'il y a des requêtes antérieures qui ne sont pas non plus incluses dans le terme π_i^k, calculons le numéro de la dernière requête plus prioritaire que celle qui a fourni π_i^k ; suivant le même schéma que précédemment, il vient

$$u = \begin{cases} \left\lfloor \frac{(R_z^{m_z}+D_z-O_j-D_j)^+}{T_j} \right\rfloor + 1 & \text{si } j < z \\ \left\lceil \frac{(R_z^{m_z}+D_z-O_j-D_j)^+}{T_j} \right\rceil & \text{si } z \leq j \end{cases}$$

Dès lors,

$$\widehat{m}_j = \max\{x, \min\{u, y\}\}$$

donne le numéro de la première requête dont l'impact éventuel n'est pas inclus dans π_i^k, et comme il ne faut considérer que celles qui arrivent strictement avant $R_i^k + \rho_i^k$ et qui sont plus prioritaires que celle dont on calcule le temps de réponse, il vient

$$n_j(\rho_i^k) = \begin{cases} \min\left\{ \left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\widehat{m}_j})^+}{T_j} \right\rceil, \left\lfloor \frac{(R_i^k + D_i - R_j^{\widehat{m}_j} + T_j - D_j)^+}{T_j} \right\rfloor \right\} & \text{si } j < i \\ \min\left\{ \left\lceil \frac{(R_i^k + \rho_i^k - R_j^{\widehat{m}_j})^+}{T_j} \right\rceil, \left\lceil \frac{(R_i^k + D_i - R_j^{\widehat{m}_j} - D_j)^+}{T_j} \right\rceil \right\} & \text{si } j > i \end{cases}$$

A nouveau, le temps de réponse peut se calculer par itération à partir de l'initialisation $\rho_i^k \leftarrow C_i + \pi_i^k$.
Il est intéressant de remarquer que, pour les techniques dynamiques du genre de DDS, les temps de réponse sont toujours finis, même s'il y a des dépassemnts de deadline, et en particulier si $U > 1$ (π_i^k augmentera alors indéfiniment, mais le nombre d'itérations restera borné).

sont plus prioritaires que celle dont on calcule le temps de réponse et qu'elles sont arrivées strictement avant $R_i^k + \rho_i^k$; en conséquence, en rassemblant les formules déjà obtenues précédemment, on obtient :

1.9.7 Les exécutifs cycliques

Afin de réduire les switching times, aussi bien dans le cas de stratégies dynamiques que statiques, on peut utiliser des implantation particulières appelées exécutifs cycliques!

Principe :
Pendant que l'on simule (off line, lors d'une phase préparatoire) l'arrivée des requêtes sur une hyperperiode, plus éventuellement lors de la montée en régime, on enregistre l'ordre de succession des exécutions de (bouts de) requêtes et des trous dans une "map".

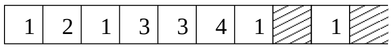

FIG. 1.26: Simulation des arrivées de requêtes dans un système

L'exécutif cyclique va alors donner le contrôle aux diverses requêtes d'après cette map, sans devoir calculer 'on the fly' les priorités ni maintenir la liste des requêtes actives.

Cette technique présente quelques inconvénients :

1. elle ne fonctionne correctement que dans le cas synchrone, car alors le scheduling est cyclique dès le départ.
2. c'est une technique qui est très lourde à implanter d'un point de vue mémoire (notamment à cause du stockage du scheduling sur toute une hyperpériode).

3. elle ne fonctionne que si les temps de calculs sont vrais (si les C_i sont vraiment les temps d'exécution) or la plupart des temps de calculs ne sont que des pires cas. On risque donc d'avoir des pertes dues à une non-expédiance. Or il est intéressant d'être expédiant car il se pourrait que les tâches prennent plus de temps que prévu : soit on les tue, soit on les laisse tourner avec une faible priorité en se disant que d'autres requêtes peuvent prendre moins de temps que prévu.
4. il peut y avoir des requêtes apériodiques or celles-ci ne rentrent pas dans la map. De plus, pour les traiter, il est aussi intéressant d'être expédiant

1.10 Les tâches apériodiques

En général, dans un système temps réel, les tâches périodiques sont majoritaires. Ce sont elles qui permettent le déroulement habituel du système en contrôlant les différents périphériques, effectuant la gestion du système et assurant les principales fonctions. En général, ces tâches sont primordiales pour la fiabilité et l'intégrité globale du système. Elles doivent donc respecter des contraintes temporelles strictes (hard deadline) ; leurs caractéristiques sont fixes et bien déterminées et ces tâches sont généralement de type critique.

Lorsqu'il se produit un événement imprévisible ou exceptionnel, les traitements nécessaires ne peuvent être effectués par les tâches périodiques de base. Ils sont alors pris en charge par une ou plusieurs tâches ou requêtes apériodiques qui ont pour rôle spécifique de gérer un événement de ce type particulier et les conséquences de celui-ci. Les requêtes apériodiques apparaissent donc de manière imprévisible dans le système. Les caractéristiques de ces requêtes ne sont connues qu'au moment de leur activation mais elles ne sont généralement pas considérées comme critiques. L'objectif du scheduler est alors de respecter les contraintes strictes des tâches périodiques tout en minimisant le temps de réponse des requêtes apériodiques. Dans le cas où les requêtes apériodiques sont de type critique, elles sont alors appelées *sporadiques*, et le scheduler doit alors garantir le respect des contraintes strictes des tâches périodiques et apériodiques simultanément, si cela est possible (le scheduler peut en général refuser une requête sporadique s'il n'est pas possible de respecter ses contraintes temporelles).

En ce qui nous concerne, nous allons porter notre attention sur le scheduling des requêtes apériodiques non critiques (avec soft deadline) en présence de tâches périodiques critiques (avec hard deadline). Le but de cette partie sera d'introduire différentes techniques qui permettent d'exécuter le plus rapidement possible (ou en tout cas pas trop mal) des requêtes apériodiques survenues dans le système, tout en satisfaisant les contraintes de scheduling pour les tâches périodiques. Comme la gestion de ces requêtes apériodiques doit se faire 'on the fly', il n'est pas question de perdre trop de temps à gérer des priorités élaborées ni à se demander si, avec elles, le système des tâches périodiques reste schédulable. C'est pourquoi on privilégiera des environnements late deadline (statiques ou dynamiques), pour lesquels on dispose de critères simples (suffisants, ou nécessaires et suffisants) pour garantir la schédulabilité du système.

1.10.1 Background Processing

Dans cette stratégie, on rajoute à l'ensemble des tâches périodiques, un processus serveur en background qui récupère les trous laissés par les tâches périodiques dans l'usage du processeur, pour satisfaire les demandes apériodiques en cours.

Si, plus précisément, nous nous plaçons dans un environnement statique, late deadline, synchrone et RMS (en fait le cas asynchrone marchera aussi, puisqu'il est plus favorable ; mais alors, RMS n'est pas optimal et le système n'est pas immédiatement périodique), cette technique consiste à rajouter une tâche serveur, périodique, de caractéristiques $T = D = P$; $C = P(1-U_p)$ (où U_p est le facteur d'utilisation pour les tâches périodiques) ; de par sa période, elle aura automatiquement la plus faible valeur de priorité parmi les tâches du système. Cette tâche serveur, chaque fois qu'elle aura le contrôle, exécutera les tâches apériodiques en cours (s'il y en a), dans leur ordre d'arrivée.

Rmq 1 : La fraction du temps processeur utilisée par les tâches périodiques au cours d'une hyperpériode est donnée par U. Il reste donc un pourcentage $1 - U$ de temps libre. Si nous multiplions cette quantité par P, on obtiendra bien le temps total que l'on aura à sa disposition à chaque hyperpériode pour l'exécution des tâches apériodiques.

Rmq 2 : Dans un environnement avec priorités dynamiques DDS, on choisira plutôt $T = D = C = \infty$ pour garantir la plus faible priorité.

Cette technique est simple à mettre en oeuvre mais pas très efficace. En effet, une tâche apériodique, comme elle reçoit la plus basse priorité, devra attendre au minimum le premier time slot libre avant de pouvoir commencer à travailler. Si, par malheur, elle arrive en début d'hyperpériode, son temps d'attente risque d'être très long ! Or il n'est pas nécessaire de donner la plus faible priorité au serveur apériodique. Au contraire, la seule chose qui importe au requêtes périodiques, c'est de respecter leur deadline, et non de se terminer le plus tôt possible avant cette deadline. On peut donc essayer de retarder les requêtes périodiques le plus possible, pourvu que l'on soit (rapidement) certain de continuer à respecter leurs deadlines.

On peut donc s'arranger pour améliorer la distribution des "miettes" du CPU laissées par les tâches périodiques. On peut introduire une ou plusieurs tâches serveurs avec une priorité plus élevée (voire même avec la plus haute priorité) afin de créer des trous libres mieux répartis.

1.10.2 Polling server

Comment peut-on introduire un (ou plusieurs) serveur périodique à haute priorité pour servir les requêtes apériodiques sans perturber la schédulabilité des tâches périodiques ?

Dans un environnement RMS, on dispose d'un critère suffisant de schédulabilité extrêmement simple (on peut l'améliorer un peu avec les $lub_n\ U$) :

$$U < \ln(2)$$

U peut être vu comme la somme entre le facteur d'utilisation $U_{periodiques}$ pour les tâches périodiques et le facteur d'utilisation $U_{serveurs}$ pour les tâches serveurs :

$$U = U_{periodiques} + U_{serveurs}$$

Dès lors, si on s'arrange pour rester en dessous de $\ln(2)$, cela garantit de conserver un système schédulable. On peut faire quelques remarques sur $U_{serveurs}$:

1. On se rappelle que c'est égal à $\frac{C_s}{T_s}$ (dans le cas où nous avons une seule tâche serveur). Si T_s est petit, nécessairement C_s doit l'être aussi. Il vient en fait :

$$C_s = \lfloor(\ln(2) - U_{periodiques})T_s\rfloor\ .$$

2. On n'atteindra jamais la valeur de $\ln(2)$, car c'est un rationnel. Pire, on risque de ne pas s'en rapprocher autant qu'on le souhaiterait, surtout si on a un seul serveur. En effet, si T_s diminue (pour avoir une plus grande priorité), C_s diminue aussi. On constate alors que l'on n'a pas beaucoup de jeu et que l'on ne saura pas s'approcher du log de 2 de plus de $\frac{1}{2T_s}$ en moyenne.
3. Imaginons qu'on ait trouvé une paire (C_s, T_s) qui soit satisfaisante, on va obtenir un système avec un ensemble de trous plus ou moins bien répartis, d'autant mieux répartis d'ailleurs que la priorité du serveur est élevée (idéalement, si on a pu se placer en tête des tâches, les trous sont exactement équidistants et assez fréquents).

FIG. 1.27: Système avec trous bien répartis.

Cependant, à cause même de sa haute priorité, et du fait qu'il convient de rester en-dessous d'un facteur d'utilisation de $\ln(2)$ ou $lub_{n+1}\ U$, le serveur ne récupérera pas tous les intervalles de temps libres du système : il en reste $(1 - U_{periodiques} - U_{serveurs})P$, c'est à dire au moins $(1 - \ln(2))P$, par hyperpériode. L'idée serait alors d'introduire d'autres processus serveur avec des priorités plus faibles, y compris un serveur background, qui essayeront de forcer et de récupérer d'autres trous laissés par les tâches périodiques.
4. Il se peut que les tâches périodiques n'utilisent pas tout leur temps de calcul car celui-ci correspond au pire cas. La présence d'un serveur background pourra alors se révéler très précieuse, mais il est encore préférable de laisser la tâche périodique inachevée servir elle-même, à son niveau de priorité, donc dans des trous plus précoces, les requêtes apériodiques en cours, s'il y en a.

On a dû jouer des coudes pour avoir des trous en début d'hyperpériode, mais que se passe-t-il s'il n'y a aucune tâche apériodique à traiter, ou s'il n'y a pas de quoi exploiter les C_s unités de CPU réservées au serveur apériodique à chacun de ses cycles ? Il y a en fait plusieurs façons d'implémenter le (ou chacun des) serveur apériodique.

1. On peut considérer que, comme dans le cas des tâches périodiques, C_s est un pire cas et que, si on ne l'exploite pas entièrement, ou pas du tout, la requête correspondante du serveur est terminée et par expédiance le processeur est laissé aux requêtes périodiques (à plus faible priorité) en cours ou qui arriveront avant le prochain cycle de serveur : la capacité de calcul résiduelle du serveur est purement et simplement perdue.

   Rmq : A chaque nouvelle période, on réarme la capacité de la tâche serveur.
   La **capacité de service** d'une tâche serveur correspond à un temps de calcul disponible à une certaine priorité pour le traitement des requêtes apériodiques.

   Et donc, si une requête apériodique arrive un peu après, elle ne sera prise en charge que lors du cycle suivant du serveur (à moins qu'il y ait d'autres serveurs prêts à s'en occuper entre-temps), et cela même si il y avait encore des (bouts de) slots normalement prévus pour ce serveur avant le cycle suivant (voir Fig. 1.28).
2. On peut améliorer cette situation (mais ça peut être un peu plus lourd à implanter) en considérant que le (chaque) serveur apériodique sert prioritairement les requêtes apériodiques puis, s'il n'y en a plus, les requêtes périodiques actives, dans leur ordre de priorité, puis l'idle loop s'il n'y a vraiment rien à faire. Mais si une nouvelle requête

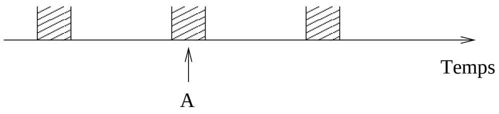

FIG. 1.28: Une requête apériodique arrive au milieu d'un trou perdu.

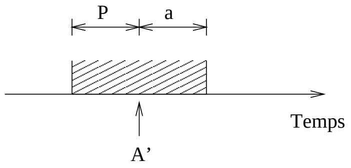

FIG. 1.29: Une requête apériodique arrive au milieu d'un trou non perdu.

apériodique arrive pendant un cycle, elle sera immédiatement considérée comme prioritaire pour le serveur (voir Fig. 1.29).
Cette amélioration permet à la tâche serveur de rester active tant que sa capacité de calcul n'est pas nulle, même si elle n'a pas de requêtes apériodiques à traiter pour le moment, car de nouvelles peuvent apparaître sous peu.

1.10.3 Deferrable Server

Le *deferrable server* utilise le même principe que le polling server sans perte des trou mais en plus sur une base *"un prêté pour un rendu"*. C'est à dire que lorsque le serveur apériodique sert une requête périodique, il conserve sa capacité de service pour usage futur (dans le même cycle) : si plus tard une requête apériodique arrive, même en dehors d'un slot normalement attribué au serveur, ce dernier reprend la main au proprata de sa capacité de service résiduelle. A chaque nouvelle période, on lui redonne sa capacité nominale. Ceci est illustré dans les graphiques ci-joints.

Un problème assez gênant peut survenir avec cette technique de deferrable server : on se base sur le fait que, tant que l'on a pas de requêtes apériodiques, on "prête" du temps CPU aux requêtes périodiques. Cependant, lorsqu'on voudra récupérer ce temps, on pourrait très bien récupérer quelque chose que l'on n'a pas prêté dans les mêmes conditions, ou de quelqu'un qui n'est plus dans les conditions de nous le rendre ! En effet,

- on récupérera à un haut niveau de priorité du temps qu'on a fourni à une requête de basse priorité ;
- on peut récupérer du temps, que l'on ôte à une requête, alors que c'est à une requête précédente (mais maintenant disparue) de la même tâche qu'on l'a emprunté.

Tout cela modifie les critères de schédulabilité et peut conduire à rater une deadline d'une tâche périodique alors que le facteur d'utilisation global est bien inférieur à $\ln(2)$. On peut cependant déterminer un $lub_{n,s}\, U$ qui joue le même rôle pour un système périodique avec un deferrable serveur, et on arrive (apès bien des calculs) à un critère

$$U_{periodique} \leq \ln\left(\frac{2 + U_{serveur}}{1 + 2U_{serveur}}\right)$$

et il faut donc choisir une paire (C_s, T_s) qui respecte cette condition. Cela conduira généralement à un C_s plus petit ou à un T_s plus grand (ou les deux) que précédemment.

1.10.4 Priority Exchange Server

Cette technique constitue une amélioration du deferrable server, basée sur les remarques précédentes. Elle se différencie quant à la manière de préserver la capacité du serveur. Elle se souvient en effet de la quantité de temps que le serveur a prêté aux tâches périodiques, séparément pour chaque niveau de priorité. De cette manière, on perturbe moins le scheduling des tâches périodiques, mais c'est beaucoup plus difficile à mettre en oeuvre puisqu'il faut maintenir plusieurs priorités de services avec les capacités de service correspondantes. Ceci est à nouveau illustré dans les graphiques ci-joints.

Ce n'est cependant toujours pas équivalent à un système de $n+1$ tâches périodiques, mais on peut encore réévaluer le critère suffisant de schedulabilité, et on obtient :

$$U_{periodique} \leq \ln\left(\frac{2}{1+U_{serveur}}\right)$$

qui en fait permet des charges supérieures à $\ln 2$.

1.10.5 Sporadic Server

Une variante de la technique précédente, appelée le *serveur sporadique*, repose sur la manière de rétablir la capacité de service ; les techniques précédentes réinstallent la totalité de la capacité du serveur à chacune de ses périodes. Dans le cas du serveur sporadique, le rétablissement n'est plus périodique mais correspond à l'utilisation réelle du processeur par les tâches périodiques. Nous ne détaillerons pas ici la façon de fixer les instants et montants de ces restaurations ; nous nous contenterons de les illustrer dans le schéma ci-joint. On peut signaler que, pour ce serveur, le critère suffisant de schédulabilité est exactement celui des systèmes périodiques classiques.

1.10.6 Slack Stealing

Cette technique de "vol de mou" est la technique optimale (du moins quand il n'y a qu'une requête apériodique à satisfaire ; quand il y en a plusieurs simultanément, la notion même d'optimalité est plus délicate à définir) et consiste à déterminer de combien au maximum on peut retarder le service de chaque requête périodique tout en respectant sa deadline. Elle est hélas très complexe et très lourde, à la fois en temps de calcul et en stockage d'informations, aussi bien en précalcul sur la partie périodique du système que "on the fly" lors de l'apparition des requêtes apériodiques. Pour cette dernière raison elle est pratiquement inapplicable.

1.10.7 Total Bandwidth

Les techniques précédentes ont essentiellement été développées pour des environnements RMS. On peut aussi les adapter pour fonctionner dans des environnements dynamiques avec late deadline et DDS. Cependant, dans ce cas, on a une technique beaucoup plus simple et assez efficace.

L'idée est simplement d'associer à chaque requête apériodique une deadline artificielle et de la gérer comme n'importe quelle requête périodique en fonction de cette deadline (c'est bien le principe de DDS). Pour être certain que la schedulabilité est respectée, il suffira de choisir cette deadline de telle façon que l'on respecte le critère (nécessaire et suffisant, mais ici utilisé uniquement comme suffisant puisque les tâches ajoutées ne sont pas vraiment périodiques) $U \leq 1$.

U se décompose comme suit : $U = U_{periodiques} + \frac{C_s}{D_s}$ où D_s est la simili période de la tâche apériodique ajoutée (s'il y en a plusieurs, on peut procéder de façon semblable ; mais une autre technique très simple consiste à attendre d'abord la fin de la tâche apériodique en cours). Dès lors, il vient

$$\frac{C_s}{D_s} \leq 1 - U_{periodiques}$$
$$\rightarrow D_s \leq \frac{C_s}{1 - U_{periodiques}}$$
$$\rightarrow D_s = \left\lceil \frac{C_s}{1 - U_{periodiques}} \right\rceil$$

En agissant de cette manière, les requêtes périodiques restent schédulables, c'est très simple à mettre en oeuvre, et on a une bonne borne sur le temps d'exécution de la requête apériodique.

FIG. 1.30: Illustration d'un serveur apériodique.

FIG. 1.31: Illustration des autres serveurs apériodiques.

1.11 Tâches dépendantes

Jusqu'à présent, nous avons toujours supposé que les tâches étaient indépendantes, c'est-à-dire que leurs seules interactions résultaient de la compétition pour le processeur. Nous allons maintenant supposer que les tâches peuvent entrer en compétition pour des ressources non-préemptives. On peut alors assister à un phénomène d'*inversion de priorité*.

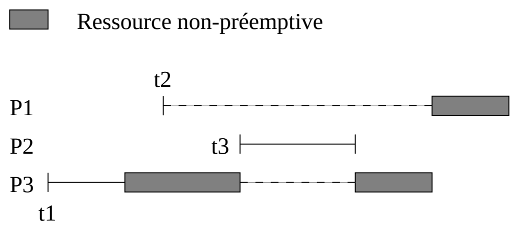

FIG. 1.32: Exemple avec ressources non-préemptives.

La figure 1.32 est un bon exemple de ce problème. Soit les processus P_1, P_2 et P_3. P_1 est le plus prioritaire et P_3 le moins prioritaire. P_3 commence son exécution puis demande la ressource non-préemptive. Comme c'est la seule tâche présente, le schéduler des ressources l'accepte. P_1 arrive ensuite et demande directement la ressource, mais ne peut pas l'avoir puisque P_3 l'a déjà. P_1 est donc obligé d'attendre. P_2 arrive ensuite. Le schéduler lui donne alors la main car cette requête est plus prioritaire que P_3, et elle peut se terminer car elle n'a pas besoin de la ressource. P_3 reçoit de nouveau la main, se termine et relâche la ressource, et P_1 peut enfin recevoir le contrôle de la ressource. La tâche la plus prioritaire (P_1) a donc, d'une certaine façon, été suspendue par une autre tâche moins prioritaire (P_2). Ces deux tâches n'avaient pourtant rien (en terme de ressource non-préemptible) en commun.

1.11.1 Héritage de priorité

Pour éviter ce problème, on va momentanément changer certaines priorités. Une technique très simple, appelée *héritage de priorité*, consiste à procéder de la manière suivante : si une tâche possède une ressource demandée par une tâche plus prioritaire, la tâche (moins prioritaire) qui a la ressource hérite de la priorité de l'autre. La tâche de faible priorité nominale va donc bloquer la ressource le moins longtemps possible. Le schéduler utilise pour cela des priorités dynamiques, données par le maximum entre la priorité statique de la tâche et les priorités dynamiques des tâches qui demandent une ressource qu'on utilise. Le délai maximum d'une tâche demandeuse de ressource sera la somme des temps d'utilisation de la ressource par des tâches utilisatrices de priorité inférieure.

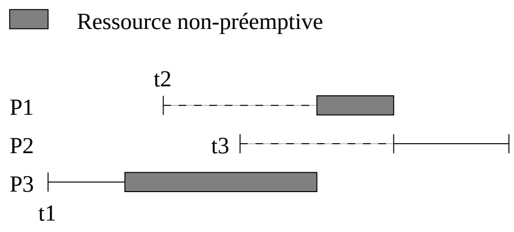

FIG. 1.33: Exemple d'héritage de priorité.

On peut voir l'intérêt de cette méthode avec le même exemple que précédemment (cf figures 1.32 et 1.33). Grâce à l'héritage de priorité, P_3 est devenue plus prioritaire que P_2 et se termine le plus rapidement possible.

1.11.2 Priority Ceiling

On va essayer de supprimer les *deadlocks* pouvant survenir à cause de l'héritage des priorités en appliquant un "plafonnage" des priorités. On va tenir compte de la priorité de tous les processus *pouvant* avoir accès à la ressource. Nous allons ainsi associer une priorité à chaque ressource, et la priorité d'un processus est sa priorité statique s'il n'utilise pas de ressource, et le maximum des priorité de toutes ses ressources dans le cas contraire. Le problème, qui reste, c'est que l'on va retarder (peut-être inutilement) des processus pour éviter des deadlocks. De plus, il faudra faire une simulation pour voir comment ça se déroule car il n'existe pas de techniques d'évaluation générales pour déterminer les temps de réponse.

1.12 Quelques ouvrages de référence

J. Goossens : *Scheduling of Hard Real-Time Periodic Systems with Various Kinds of Deadline and Offset Constraints*, thèse ULB, 1999.
V. Van Lier : *Scheduling de tâches apériodiques en présence de tâches périodiques indépendantes dans les systèmes temps réel monoprocesseur*, mémoire ULB, 1997.
M. Macq : *Etude pratique des principaux algorithmes d'ordonnancement de tâches apériodiques en présence de tâches périodiques dans un système temps réel*, mémoire ULB, 1998.
A. van Tilborg et G. Koob : *Foundations of Real-Time Computing : Scheduling and Resource Management*, Kluwer Academic Publishers, 1991.
A. Burns et A. Wellings : *Real-Time Systems and Programming Languages*, Addison-Wesley, 1997.