Systèmes Temps Réels

4GI/4GE

2023/2024
Version 3.2

Présentation du cours :

L'objectif général de ce cours est d'étudier les concepts de base du fonctionnement des systèmes temps réel. Les caractéristiques liées à un fonctionnement « temps réel » sont introduites en s'appuyant sur l'étude des exécutifs temps réel à travers les ordonnancements et leur utilisation dans la conception de systèmes logiciels/matériels multitâches et leur mise en œuvre concrète dans un OS (ou exécutif) temps réel.

Chapitres du cours :

- Aperçu des systèmes temps réels et embarques
- Qu'est-ce qu'un système temps réel ?
- Comment vérifier a priori le comportement
- Ordonnancement
- Comment implémenter le comportement
- Les exercices proposés pendant les séances

Coordonnées et disponibilités

Abdellatif CHAFIK chargé de cours abdellatif.chafik@iga.ac.ma

Disponibilités pour les questions générales sur le cours, le contenu, les travaux drivés, etc., je vous invite à poser vos questions par mail via l'adresse du groupe. Il est à noter que si vous m'envoyez une question d'intérêt général sur ma boîte courriel, je vous inviterai de mettre en copie le groupe des étudiants.

I. La notion de temps réel :

L'expression "temps réel" est apparue aux débuts des années 1950 chez les informaticiens scientifiques et militaires. Il ne s'agissait pas de l'expression "temps réel" mais de son équivalent anglais "real-time". Aussi, il m'a paru nécessaire de chercher sa définition :

> *« **Un système fonctionne en Temps Réel s'il est capable d'absorber toutes les informations en entrée avant qu'elles soient trop vielles pour l'intérêt qu'elles présentent et de réagir à celles-ci suffisamment vite pour que cette réaction est un sens** » (ABRIAL – BOURGNE).*
>
> *Et l'adage suivant devra être garde en mémoire :*
>
> *« **Un résultat juste, mais hors délai, est un résultat faux** »*

Un retard est considéré comme une erreur qui entraine de graves conséquences

Une autre définition :

> ***"Un système en temps réel est un système logiciel qui maintient une interaction constante et ponctuelle avec son environnement"*** *(Bran Selic)*

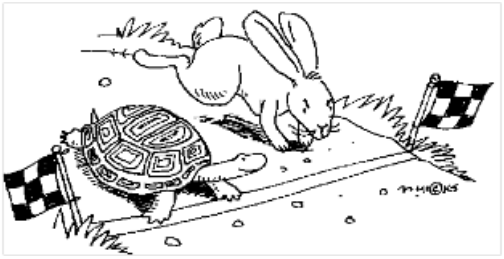
Lorsqu'on parle de "temps" dans l'expression "temps réel", parle-t-on du temps physique et donc par définition du temps local de l'observateur ? Ou bien du temps logique, rythmé par la cadence d'horloge de l'ordinateur, un temps discret par définition, instable (il a la stabilité de la référence qui le génère) et fini ? Evidemment, il s'agit de la seconde hypothèse. Et cela borne en les minorant la notion de "suffisamment rapide" et d'immédiat". Le délai de temps le plus petit mesurable est le pas temporel de l'horloge de l'ordinateur, variable entre quelques nanosecondes et quelques microsecondes selon les machines.

Et que dire du mot "réel" ? Existerait-il en informatique un temps imaginaire ? Quelle est la réalité du temps ? En physique comme en informatique, nous mesurons des durées, c'est à dire des intervalles de temps et dans un ordinateur, chaque intervalle est un multiple du pas temporel d'horloge de l'ordinateur. En physique, nous ne savons pas si notre temps est discret (il l'est probablement), ni même ce qu'est réellement le temps.

En bref, la notion de "temps" en informatique n'a pas grand-chose à voir avec le temps des physiciens. Mais l'avantage, c'est qu'on peut le quantifier rigoureusement car il possède une origine (la mise sous tension de l'ordinateur) et une unité fondamentale, le pas d'horloge de l'ordinateur. L'expression "temps réel" désigne finalement le plus souvent une image du monde réel, ou du moins d'une de ses parties, à un instant donné.

Mauvaises interprétations de la notion de système temps réel :

- « Real-time is not real-fast » ou « rien ne sert de courir, il faut partir à point »



- Aller vite n'est pas l'objectif recherché

I.1. Propriétés attendues :

Les systèmes temps réel sont des systèmes devant respecter les contraintes de temps imposées. A priori, le code de chacune des tâches composant un tel système est connu. Il peut donc être analysé en termes de temps d'exécution, ressources nécessaires, et dépendance envers d'autres tâches. Les propriétés suivantes devraient être respectées :

Gestion du temps. Les résultats doivent être corrects, non seulement en termes de valeur, mais également termes de temps d'arrivée. Le système doit garantir que les tâches se terminent dans les délais impartis, en respectant les contraintes de temps imposées.

Conception pour stress. Un système temps réel ne doit pas succomber lorsqu'il est soumis à une charge importante. Des scénarios mettant en avant des charges limites doivent être anticipés, et testés.

Prédictibilité. A chaque décision liée à l'ordonnancement des tâches, le système doit être capable de prédire les conséquences de son choix. Il doit pouvoir dire si les tâches sont ordonnançables, et si tel est le cas proposer un ordonnancement. S'il n'est pas possible de gérer l'ensemble de tâches, des actions coordonnées doivent être prises afin de ne pas compromettre le fonctionnement du système.

Si le système effectue le traitement avant le délai qui lui était imparti, le résultat de l'opération est correct, sinon il est considéré comme invalide comme le montre la figure ci-dessous où se trouvent placés sur l'axe temporel l'événement de départ (timer, interruption, signal, etc.) et la limite au-delà de laquelle la réponse du système n'est plus acceptable.

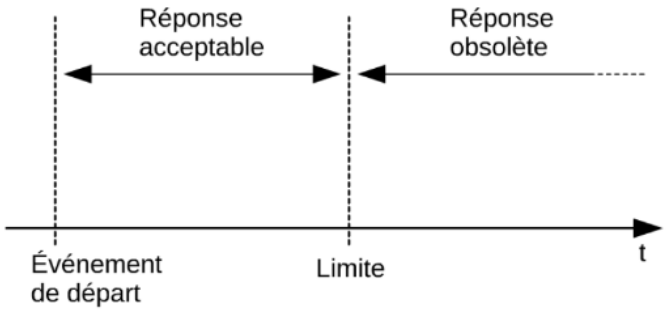


Figure 1 Contrainte temporelle

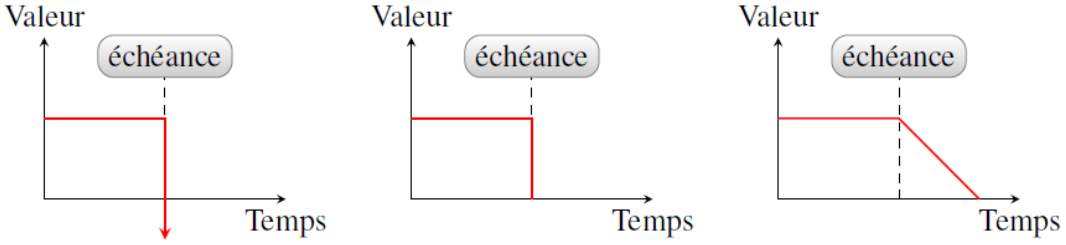
I.2. Temps réel strict vs. Souple

En fonction de la nature des contraintes temporelles appliquées à un système temps réel, celui-ci peut être de deux types :

Temps réel strict ou dur (hard). Il s'agit d'un système où l'ensemble des contraintes temporelles doit absolument être respecté. Pour ce faire il faut pouvoir définir les conditions de fonctionnement du système, c'est-à-dire connaître parfaitement l'environnement du système. Il faut également être capable de garantir la fiabilité du système avant son exécution. Tous les scénarios possibles d'exécution doivent donc être étudiés et le bon fonctionnement du système doit être garanti pour chacun de ces scénarios. L'ensemble du système doit évidemment être suffisamment connu pour pouvoir fournir de telles garanties.

A titre d'exemple, le contrôle d'un avion par un pilote automatique est un système temps réel strict.
Il existe également des systèmes appelés temps réel ferme (firm), où la pertinence du résultat est nulle passée l'échéance, mais dont le non-respect de l'échéance ne compromet pas le fonctionnement du système ou l'intégrité des personnes. Une application bancaire responsable de calculer des risques en fonction des paramètres courants du marché en est un exemple.
Temps réel souple ou mou (soft). Les systèmes temps réel souple ont des contraintes de temps moins exigeantes que les précédents. Une faute temporelle n'y est pas catastrophique pour le fonctionnement. Un tel système pourra donc accepter un certain nombre de fautes temporelles, tout en pouvant continuer son exécution.

A titre d'exemple, les applications de type multimédia telles que la téléphonie ou la vidéo sont des applications temps réel souple, car la perte de certaines informations, liée à un débit trop faible, n'est pas dangereux ou catastrophique. Les systèmes de ce type peuvent ensuite être comparés en fonction de la qualité de service qu'ils offrent, en termes de probabilité d'erreur.
La figure dessous résume la valeur d'un calcul en fonction de son instant d'arrivée. Dans le cas d'une échéance stricte ou firme, passé l'échéance, le calcul n'a plus aucune valeur. La différence entre ces deux cas vient du fait qu'une échéance stricte manquée met en péril le système, alors qu'une échéance firme manquée implique uniquement que le résultat du calcul n'est plus pertinent. Dans le cas d'une échéance molle, la pertinence du résultat perd de sa valeur après l'échéance, mais ce d'une manière plus douce.



(a) Echéance stricte (b) Echéance ferme (b) Echéance ferme

I.3. Qu'est-ce qu'une tâche ?

En programmation séquentielle, un programme est décomposé en sous-programmes (procédures ou fonctions). Chaque sous-programme correspond à une suite d'instructions, et l'exécution du programme voit ces instructions être exécutées les unes à la suite des autres. Il est en général possible de réaliser une application en une seule tâche, mais la scrutation de périphériques risque d'y être délicate et coûteuse en temps processeur.

Un système temps réel est toujours décomposé en tâches, chacune ayant des fonctionnalités, des temps d'exécution et des échéances différentes. Il serait parfois possible de les combiner en une seule tâche, mais cet exercice n'est pas souhaité, notamment à cause du contrôle plus délicat sur les parties d'exécution que ceci impliquerait.

Considérons un simple système composé de deux entrées (un port série et un clavier), d'une partie de traitement, et d'une sortie. Dans un contexte temps réel, le port série doit être scanné régulièrement afin d'éviter qu'un octet ne soit perdu. Si la tâche de traitement est conséquente, il faudrait, en milieu de traitement, aller vérifier l'état du port série, sans oublier la gestion du clavier. Il faudrait également faire de même si la transmission vers la sortie est lente. Dans un tel cas, la décomposition du système en

tâches distinctes permet d'éviter ces inconvénients. Nous pouvons imaginer une tâche responsable de lire les données venant du port série, et ce à une cadence garantissant le respect des échéances visant à ne pas perdre d'octet. Une autre tâche serait responsable de la gestion du clavier, une autre du traitement, et enfin une dernière de la gestion de la sortie. Des mécanismes de communication peuvent être mis en œuvre pour permettre aux différentes tâches de s'échanger des données, et des priorités peuvent être assignées, notamment pour garantir que le port série est bien géré.

Sur un simple cœur, les parties de tâches s'exécutent tour à tour de manière transparente, et sur un multicœur, un réel parallélisme peut être observé, chaque cœur pouvant exécuter un ensemble de tâches.

II. Ordonnancement Temps Réel

II.1. Introduction :

Tâches temps réel soumises à des contraintes de temps, plus ou moins strictes.

- Instant de démarrage
- Instant de fin
- Absolus ou relatifs à d'autres tâches

Le but de l'ordonnancement est de permettre le respect de ces contraintes, lorsque l'exécution se produit dans un mode courant, il doit permettre de borner les effets d'incidents ou de surcharges.

II.2. Caractéristiques des tâches

La définition d'une tâche est évidemment fortement liée au programme décrivant son exécution.
Toutefois, dans le cadre de l'ordonnancement de systèmes temps réel, différents paramètres sont également nécessaires à la mise au point de solutions correctes.
Les paramètres d'une tâche T sont :

- **r** : date de réveil de la tâche (ou date de demande d'activation) moment du déclenchement de la 1ère requête d'exécution
- **C** : durée d'exécution maximale (capacité). Il s'agit du pire temps d'exécution
- **D** : délai critique : délai maximum acceptable pour son exécution
- **P** : période (si tâche périodique)
- **d** : pour une tâche à contraintes strictes, son échéance, calculée comme étant
  $$d = r + D$$
- Tâche périodique : $r_k = r_0 + k*P$
  Si D = P, tâche à *échéance sur requête*

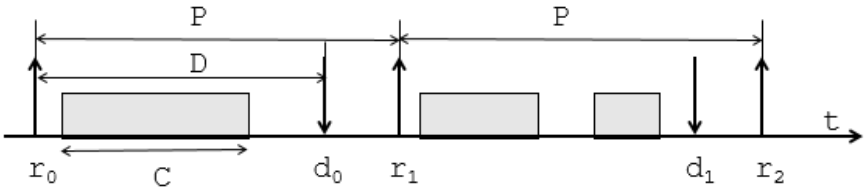


Figure 2: Paramètres typiques d'une tâche périodique T

II.3. Paramètres statiques

- U = C/P : facteur d'utilisation du processeur
- CH = C/D : facteur de charge du processeur
- Paramètres dynamiques
- s : date du début de l'exécution
- e : date de la fin de l'exécution
- D(t) = d-t : délai critique résiduel à la date t ($0 \leq D(t) \leq D$)
- C(t) : durée d'exécution résiduelle à la date t ($0 \leq C(t) \leq C$)
- L = D-C : laxité nominale de la tâche, le retard maximum pour son début d'exécution s (si elle est seule)
- L(t) = D(t) - C(t) : laxité nominale résiduelle, le retard maximum pour reprendre l'exécution
- TR = e - r : temps de réponse de la tâche
- CH(t) = C(t)/D(t) : charge résiduelle ($0 \leq CH(t) \leq C/P$)

Une tâche apériodique est donc représentée par le tuple $T(r,C,D)$, alors qu'une tâche périodique l'est par le tuple $T(r_0,C,D)$, r_0 étant la date de réveil de la première occurrence.

Une tâche à échéance sur requête est une tâche périodique pour laquelle le délai critique est égal à la période ($D = P$).

II.4. Caractéristiques des tâches

- Préemptibles[1] ou non
- Dépendance ou indépendance
- Ordre partiel prédéterminé ou induit
- Partage de ressources
- Priorité externe
- Ordonnancement hors ligne déterminé à la conception
- Gigue (*jitter*) maximale
- Variation entre la requête et le début de l'exécution
- Urgence ↔ échéance
- Importance

II.5. Tâches dans un contexte temps réel

Cycle de vie

La figure ci-dessous illustre les différentes étapes de la vie d'une tâche, dans un contexte temps réel. L'état initial d'une tâche est inexistant. Après sa création, elle passe à l'état passive lorsque toutes les ressources à son bon fonctionnement ont été réquisitionnées. Elle reste dans cet état jusqu'à être réveillée, ce qui peut n'est fait qu'une seule fois pour une tâche apériodique, ou de manière répétée pour une tâche périodique. Après le réveil, elle se trouve dans l'état prête, où elle se retrouve en compétition avec les autres tâches disposées à être exécutées. L'ordonnanceur a alors la charge de choisir les tâches à activer. De l'état prête, le système d'exploitation peut ensuite la faire passer dans l'état élue, état dans lequel la tâche s'exécute.

Ce passage n'est pas du ressort de la tâche, mais bien de l'ordonnanceur, qui s'occupe d'allouer le processeur aux différentes tâches concurrentes, et ce en suivant la politique

[1] Une **tâche préemptible** peut être suspendue (mise à l'état « ready ») au profit d'une **tâche** de priorité plus élevée ou d'une interruption.

d'ordonnancement choisie. A tout instant l'ordonnanceur peut replacer la tâche dans l'état prête, pour laisser une autre

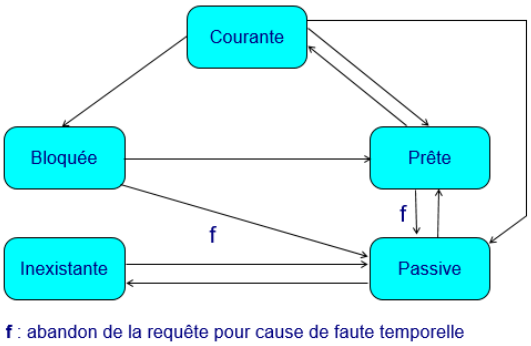


Figure 3 : Etats et transitions d'une tâche dans un contexte temps réel

Tâche s'exécuter. Il s'agit de la préemption d'une tâche, qui se fait sans que la tâche préemptée.

Définitions :

- Configuration : ensemble de n tâches mises en jeu par l'application
  - facteur d'utilisation du processeur :

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i}$$

  - Facteur de charge :

$$CH = \sum_{i=1}^{n} \frac{C_i}{D_i}$$

- Intervalle d'étude : intervalle de temps minimum pour prouver l'ordonnançabilité d'une configuration
- Le PPCM[2] des périodes dans le cas d'une configuration de tâches périodiques
- Laxité du processeur LP(t) = intervalle de temps pendant lequel le processeur peut rester inactif tout en respectant les échéances
- Laxité conditionnelle :

$$LC_i(t) = D_i - \sum C_i(t)$$

  (Somme sur les tâches déclenchées à la date t et qui sont devant i du point de vue de l'ordonnancement)
- $LP(t) = \min (LC_i(t))$

II.6. Ordonnancement

Tout système d'exploitation dispose d'un ordonnanceur, qui a pour fonction de répartir l'utilisation du processeur entre les différentes tâches demandeuses. Diverses solutions existent, et peuvent être classées en fonction de différents paramètres. La section suivante propose une taxonomie des algorithmes d'ordonnancement classiques, au sens où ils sont utilisés dans des systèmes ayant recours à des contraintes temps réel.

[2] PPCM : Plus petit multiple commun

Taxonomie

Les tâches sont divisées en deux grandes catégories, en fonction de leur périodicité.

- Les tâches périodiques (ou cycliques) sont des tâches qui sont exécutées à intervalles réguliers.

Elles sont entre autres définies par la période avec laquelle elles sont censées s'exécuter, ce qui est géré par l'ordonnanceur.

Une tâche périodique peut par exemple être responsable de l'observation d'un capteur à intervalles réguliers, de la régulation de moteurs, de monitoring, etc.

- Les tâches apériodiques sont, quant à elles, exécutées à intervalles irréguliers, et peuvent donc survenir à n'importe quel instant. Il s'agira typiquement de tâches de configuration, et de tâches activées par une alarme, ou par une entrée de l'utilisateur. Les interruptions jouent un rôle important, étant le principal vecteur d'activation.

Une tâche est dite sporadique si elle est apériodique et que l'intervalle minimal entre deux activations est connu.

Au niveau de l'interaction entre tâches, nous pouvons distinguer :

- Les **tâches indépendantes**, dont l'ordre d'exécution peut être quelconque, elle n'a pas besoin de ressources.
- Les **tâches dépendantes**, pour lesquelles il existe des contraintes de précédence. Il s'agit de cas où une tâche doit attendre qu'une autre ait terminé son traitement pour pouvoir commencer le sien.

Au niveau des politiques d'ordonnancement, il est clair que les dépendances entre tâches auront un rôle crucial et devront être prises en compte de manière judicieuse.

III. Typologie des algorithmes

III.1. En ligne/hors-ligne :

Un ordonnancement hors-ligne est effectué avant le lancement du système. Ceci implique que tous les paramètres des tâches soient connus a priori, et notamment les dates d'activation. L'implémentation du système est alors très simple : il suffit de stocker l'identifiant de la tâche à effectuer à chaque moment d'ordonnancement dans une table, l'ordonnanceur exploitant ensuite cette information pour sélectionner la tâche à exécuter. Le désavantage de cette approche est la grande dépendance au système cible, par contre l'analyse hors-ligne permet une meilleure prédiction de la satisfaction ou non des contraintes temporelles. De même, la puissance de calcul disponible hors-ligne permet de calculer un ordonnancement optimal, ce qui est un problème NP-complet, tâche impossible à réaliser dans un système embarqué.

Un ordonnancement en ligne est effectué durant le fonctionnement du système. L'ordonnanceur recalcule un nouvel ordonnancement à chaque fois qu'une nouvelle tâche est activée. L'avantage de cette approche est la flexibilité, et l'adaptabilité à l'environnement que procure le calcul en ligne.

Par contre, ce calcul devant être exécuté par le système, qui est temps réel, il doit être le plus simple et rapide possible, rendant impossible une solution dite optimale.

Différents algorithmes permettent de résoudre ceci, notamment en utilisant des heuristiques.

III.2. Statique/dynamique

Un ordonnancement est dit statique s'il est uniquement basé sur les propriétés des tâches avant le lancement du système. Un ordonnancement est dynamique s'il est capable de réagir à la modification des propriétés des tâches durant le fonctionnement du système (typiquement un changement de priorité).

III.3. Dans ce qui suit nous allons voir :

- Ordonnancement des tâches indépendantes
- Ordonnancement des tâches dépendantes
  - Partage de ressources
  - Contraintes de précédence
- Ordonnancement en cas de surcharge

IV. Ordonnancement des tâches indépendantes

IV.1. Rate Monotonic Analysis (RMA)

L'ordonnancement à taux monotone (en anglais, **rate-monotonic Analysis**) est un algorithme d'ordonnancement temps réel en ligne à priorité constante (statique). Il attribue la priorité la plus forte à la tâche qui possède la plus petite période. RMA est optimal dans le cadre d'un système de tâches **périodiques**, **synchrones**, **indépendantes** et à **échéance sur requête** avec un ordonnanceur **préemptif**. De ce fait, il n'est généralement utilisé que pour ordonnancer des tâches vérifiant ces propriétés.

Le test d'acceptabilité (condition Liu et Layland est suffisante) :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n\left(2^{1/n} - 1\right)$$

Mais pas nécessaire : si elle est satisfaite, les tâches peuvent être ordonnancées, sinon, on ne peut rien conclure.

Quand le nombre de tâches tend vers l'infini (est très grand) :

$$\lim_{n\to\infty} n\left(2^{\frac{1}{n}} - 1\right) = ln2 = 0{,}69$$

Dans la pratique, on peut rencontrer des ordonnancements valides qui vont jusqu'à 88%, on peut aussi monter que RMA est un algorithme « optimal » : une configuration qui ne peut pas être ordonnancée par RMA ne pourra pas être ordonnée par un autre algorithme à priorité fixe.

Exemple :

Soit une configuration avec trois tâches à échéance sur requête :

- T_1 (r_0=0, C=3, P=20)
- T_2 (r_0=0, C=2, P=5)
- T_3 (r_0=0, C=2, P=10)

La configuration est-elle ordonnançable avec RMA ?

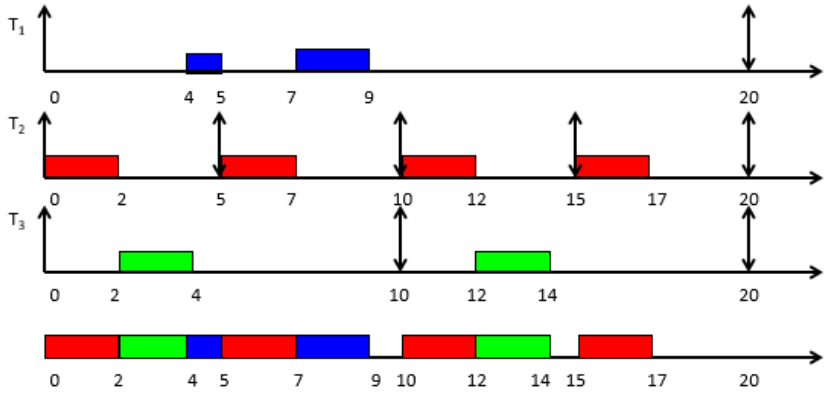
Pour répondre à cette question on commence par vérifier la condition si elle est satisfaite :

$$\sum \frac{C_i}{P_i} = \frac{3}{20} + \frac{2}{5} + \frac{2}{10} = 0{,}75 \quad \leq n\left(2^{\frac{1}{n}} - 1\right) = 0{,}77$$

Les tâches peuvent être ordonnancées.

Le PPCM (P_1, P_2, P_3) = PPCM (20,5,10) = 20

La priorité d'exécution des tâches est : $Prio_2 > Prio_3 > Prio_1$



IV.2. Deadline Monotonic Analysis (DMA)

L'ordonnancement à taux monotone (en anglais, **Deadline Monotonic Analysis**) est un algorithme d'ordonnancement temps réel en ligne à priorité constante (statique) équivalent à RMA dans le cas des tâches à échéance sur requête, meilleur dans les autres cas. Il attribue la priorité la plus forte à la tâche qui possède le plus petit délai critique. DMA est optimal dans le cadre d'un système de tâches **périodiques**, **synchrones**, **indépendantes** base sur **le délai critique** avec un ordonnanceur **préemptif**. De ce fait, il n'est généralement utilisé que pour ordonnancer des tâches vérifiant ces propriétés.

Le test d'acceptabilité (condition est suffisante) :

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

a. Exemple :

Soit une configuration avec trois tâches :

- T_1 (r_0 = 0, C=3, D=7, P=20)

- T_2 (r_0 = 0, C=2, D=4, P=5)
- T_3 (r_0 = 0, C=2, D=9, P=10)

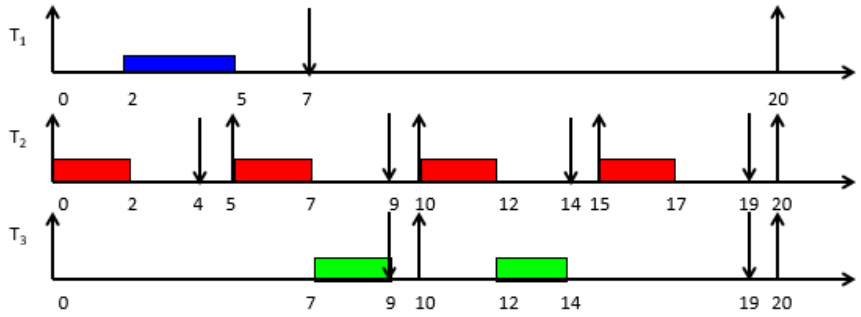
La configuration est-elle ordonnançable avec DMA ?

Pour répondre à cette question on commence par vérifier la condition si elle est satisfaite :

$$\sum \frac{C_i}{D_i} = \frac{3}{7} + \frac{2}{4} + \frac{2}{9} = 1{,}14 \quad \geq n\left(2^{\frac{1}{n}} - 1\right) = 0{,}77$$

La condition n'est pas satisfaite mais on ne peut rien conclure, alors il faut vérifier sur le graphe d'exécution si les tâches peuvent être ordonnancées :

La priorité d'exécution des tâches est : $Prio_2 > Prio_1 > Prio_3$



Conclusion : La tâche T_3 livré à son deadline lors de sa première période, donc les tâches ne peuvent pas être ordonnancées DMA s'il s'agit de temps réel dur.

IV.3. Earliest Deadline First (EDF)

Earliest deadline first scheduling ("échéance proche = préparation en premier") est un algorithme d'ordonnancement préemptif, à priorité variable ou dynamique, utilisé dans les systèmes temps réel. Il attribue une priorité à chaque requête en fonction de l'échéance de cette dernière selon la règle : Plus l'échéance d'une tâche est proche, plus sa priorité est grande. De cette manière, au plus vite le travail doit être réalisé, au plus il a de chances d'être exécuté.

Test d'acceptabilité des tâches :

- Condition nécessaire :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$$

- Condition suffisante :

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1$$

On peut montrer que EDF est un algorithme optimal pour les algorithmes à priorité dynamique.

a. Exemple :

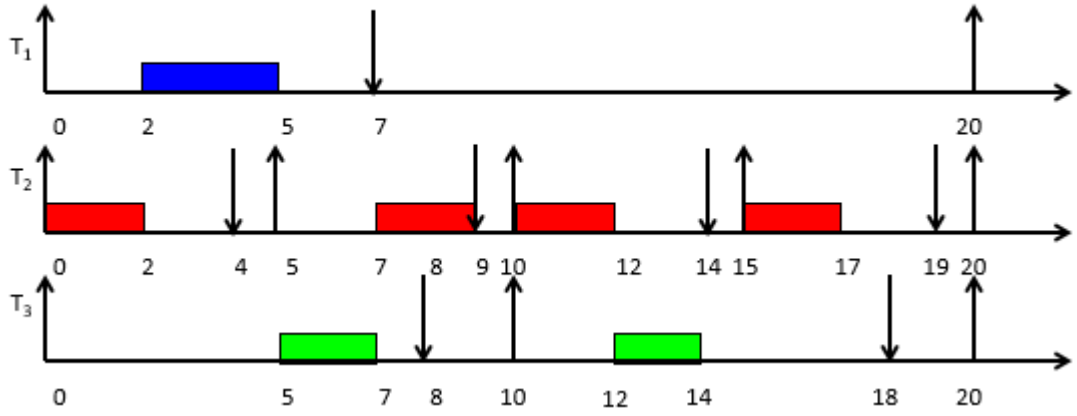
Soit trois tâches :

- T_1 (r_0 = 0, C=3, D=7, P=20)
- T_2 (r_0 = 0, C=2, D=4, P=5)
- T_3 (r_0 = 0, C=2, D=8, P=10)

Condition nécessaire Liu and Layland : $\sum \frac{C_i}{P_i} = 3/20 + 2/5 + 2/10 = 0{,}75 < 1$

Condition suffisante Liu and Layland : $\sum \frac{C_i}{D_i} = 3/7 + 2/4 + 2/8 = 1{,}18 > 1$

On ne peut rien conclure avec le test d'ordonnancement Liu and Layland. Le chronogramme illustre la trace d'ordonnancement selon l'algorithme EDF (ce qui prouve qu'un ordonnancement existe), on remarque que T_2 est à la limite du dedline :



IV.4. Least Laxity First (LLF)

L'algorithme d'ordonnancement dynamique LLF (aussi appelé Least Slack Time (LST)) équivalent à EDF si on ne calcule pas la laxité qu'au réveil des tâches. L'idée derrière le LLF est de calculer, à chaque période d'ordonnancement, la priorité maximale est donnée à la tâche qui a la plus petite laxité résiduelle est défini comme suit :

$$\mathbf{L(t) = D(t) - C(t)}$$

Ou D est le deadline, t est le temps réel depuis le début du cycle courant et c'est le temps d'exécution restant dans le cycle courant. Finalement, le processus avec la plus petite laxité se voit attribuer par l'ordonnanceur la plus grande priorité.

a. Exemple :

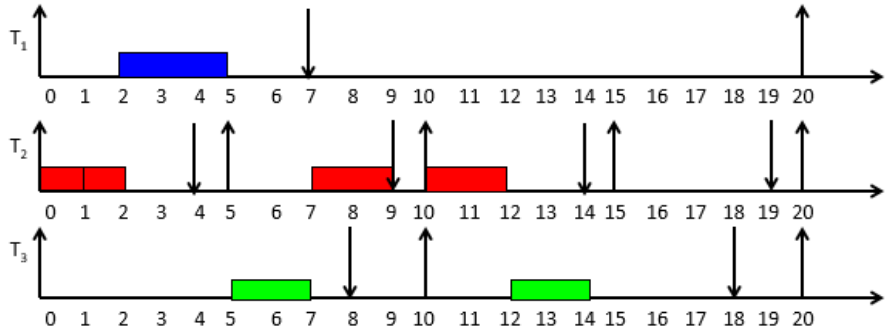
Soit la même configuration que pour EDF :

- T_1 (r_0 = 0, C=3, D=7, P=20)
- T_2 (r_0 = 0, C=2, D=4, P=5)
- T_3 (r_0 = 0, C=2, D=8, P=10)

Pour ce faire, on va calculer la laxité pour t=0,1 ,2, 3 ,4 ,5 ….

Temps	Laxité du T1	Laxité du T2	Laxité du T3
0	7 - 3 = 4	**4 - 2 = 2**	8 - 2 = 6
1	6 - 3 = 3	**3 – 1 = 1**	7- 2 = 5
2	**5 - 3 = 2**	**Terminé**	6 - 2 = 4
3	**4 – 2 =2**		5 - 2 = 3
4	**3 - 1 =2**		4 - 2 = 2
5		9 – 7 = 2	**3 – 2 =1**
6		8 – 7 = 1	**2 – 1 =1**
7		**7 – 7 = 0**	**Terminé**
…	**….**	**…..**	**…..**

Les chronogrammes :



On remarque que le LLF est optimum à trouver entre la granularité du calcul et le nombre de changements de contexte provoqués.

V. Traitement des tâches apériodiques

L'objectif est toujours d'ordonnancer les tâches apériodiques à contrainte souple dans un contexte de tâches périodiques à contrainte stricte, la réponse est garantie.

Les tâches apériodiques dont le taux d'arrivée est borné supérieurement sont dites « sporadiques ».

Pour tenir compte de tâches apériodiques parmi l'ensemble des tâches à ordonnancer, il existe plusieurs approches.

But à atteindre dans ce cas est :

- Si contraintes relatives : **minimiser le temps de réponse.**
- Si contraintes strictes : **maximiser le nombre de tâches acceptées en respectant leurs contraintes.**

Parmi les approches existantes, deux grandes catégories de traitement ont couramment employées :

- Le traitement en arrière-plan.
- Le traitement par serveur.

Sans oublier qu'il y'a beaucoup d'études d'algorithmes.

VI. Tâches apériodiques à contraintes relatives

VI.1. Traitement d'arrière-plan :

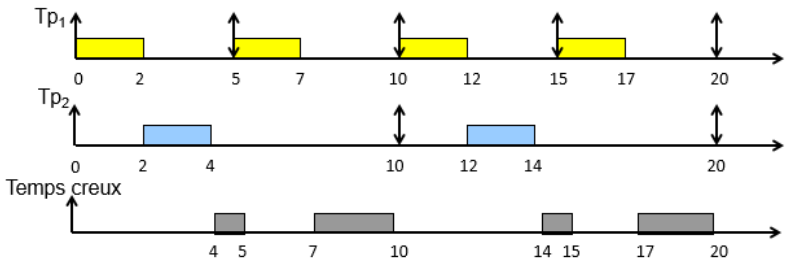
Les tâches apériodiques seront ordonnancées quand le processeur est oisif[3], les tâches périodiques restent les plus prioritaires peuvent aussi préempter les tâches apériodiques, ce traitement reste le plus simple, mais le moins performant.

a. Exemple :

La priorité est donnée aux tâches périodiques par une des politiques qu'on a étudiés (RMA, DMA, EDF, LLF). Quand le processeur est **oisif (IDLE STATE)** les tâches apériodiques prennent la main pour exécuter par une des politiques.

Soit deux Tâches périodique sont ordonnancer par l'algorithme RMA :

- Tp_1 (r_0=0, C=2, P=5)
- Tp_2 (r_0=0, C=2, P=10)

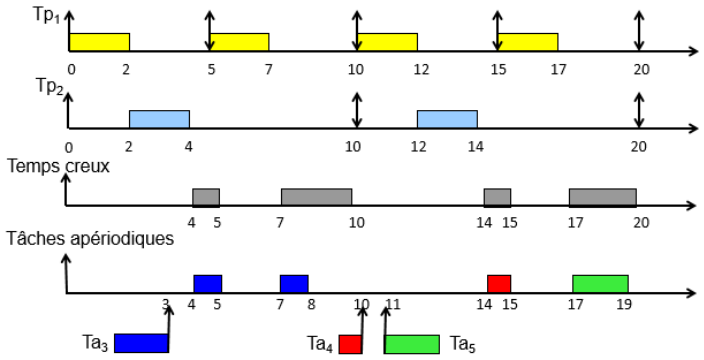


On remarque que le processeur est oisif pendant les intervalles du temps : [4,5], [7,10], [14,15] et [17,20], on peut dans ce cas intercaler les tâches apériodiques en mode FIFO.

Soit trois tâches apériodiques contraintes relatives :

- Ta_3 (r=3, C=2)
- Ta_4 (r=10, C=1)
- Ta_5 (r=11, C=2)

[3] Qui est dépourvu d'occupation, ne traite pas de tâche.



L'ordonnanceur de ce système manipule deux files d'attente (Ready Queue (R.Q)) une pour les tâches périodiques et l'autre pour les tâches apériodiques.

VII. Traitement par serveur

Le principe est de définir une tâche périodique spéciale, appelée serveur apériodique, active durant son temps d'exécution c'est sa capacité, et une période les tâches apériodiques en attente , c'est une forme de réservation garantie de temps CPU consacré uniquement aux tâches apériodiques, le serveur généralement ordonnancé suivant le même algorithme que les autres tâches périodiques (RMA), une fois actif, le serveur sert les tâches apériodiques dans la limite de sa capacité, l'ordre de traitement des tâches apériodiques ne dépend pas de l'algorithme général. . Deux types de serveurs existent :

- Le serveur par scrutation.
- Le serveur sporadique.
-

VII.1. Traitement par serveur par scrutation

A chaque activation du serveur par scrutation (polling), traitement des tâches en suspens jusqu'à épuisement de sa capacité ou jusqu'à ce qu'il n'y ait plus de tâches en attente, si aucune tâche n'est en attente (à l'activation ou parce que la dernière tâche a été traitée), le serveur se suspend immédiatement et perd sa capacité qui peut être réutilisée par les tâches périodiques (amélioration du temps de réponse).

a. Exemple :

Soit deux tâches périodiques :

- Tp_1 (r0=0, C=3, P=20)
- Tp_2 (r0=0, C=2, P=10)

Soit la tâche du serveur par scrutation :

- Tp_s (r0=0, C=2, P=5)

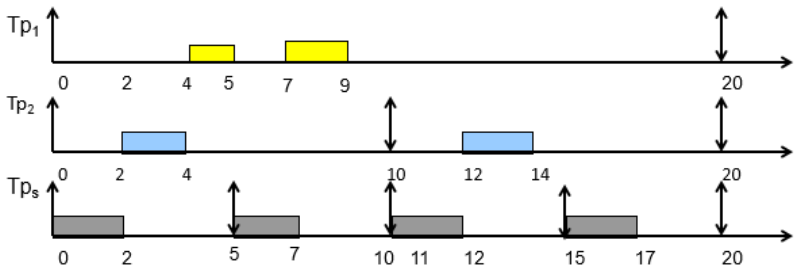
Ordonnancements RMA qui sera utilisé.

On vérifie le test d'acceptabilité :

$$\sum \frac{C_i}{P_i} = 3/20 + 2/5 + 2/10 = 0{,}75 \quad \leq n(2^{\frac{1}{N}} - 1)$$

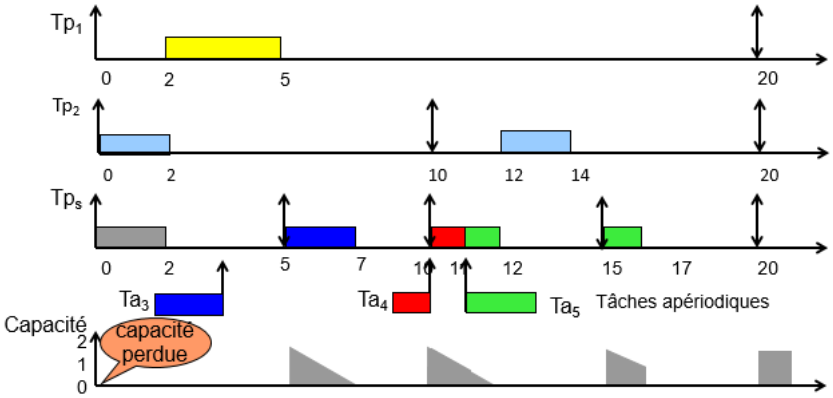
Le système est ordonnançable

On remarque que la tâche du serveur peut s'exécuter sans influencer sur les performances des autres tâches.



Soit trois tâches apériodiques :

- Ta_3 (r=4, C=2)
- Ta_4 (r=10, C=1)
- Ta_5 (r=11, C=2)



La perte de la capacité si aucune tâche apériodique en attente et si occurrence d'une tâche apériodique alors que le serveur est suspendu, il faut alors attendre la requête suivante sont les limitations du serveur par scrutation.

VIII. Traitement par serveur sporadique :

Le serveur sporadique améliore le temps de réponse des tâches apériodiques sans diminuer le taux d'utilisation du processeur pour les tâches périodiques, le serveur est ajournable mais ne retrouve pas sa capacité à période fixe ; il peut être considère comme une tâche périodique « normale » du point de vue des critères d'ordonnancement.
Comment le serveur calcul la date de la récupération de la capacité ?

- Le serveur est dit « actif » quand la priorité de la tâche courante P_x est supérieure ou égale à celle du serveur P_s

- Le serveur est dit « inactif » quand la priorité de la tâche courante est inferieur à celle du serveur Ps
- On note RT comme date de la récupération
  - La date de récupération est calculée dès que le serveur devient actif (t_A)
  - Égale à $t_A + T_s$
- RA : le nombre à récupérer à RT
  - Calculée à l'instant t_I où le serveur devient inactif ou que la capacité est épuisée
  - Egal à la capacité consommée pendant l'intervalle [t_A, t_I]

Exemple de serveur sporadique à haute priorité

Soit deux tâches périodiques :

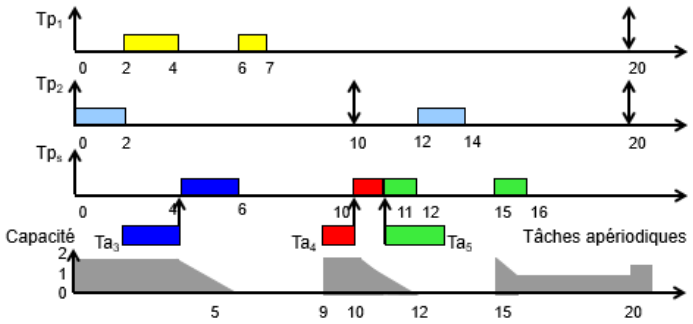
- Tp_1 (r_0=0, C=3, P=20)
- Tp_2 (r_0=0, C=2, P=10)

Le serveur sporadique:

- Tp_s (r_0=0, C=2, P=5)

Les tâches apériodiques :

- Ta_3 (r=4, C=2)
- Ta_4 (r=10, C=1)
- Ta_5 (r=11, C=2)



Le serveur haute priorité, inactif quand une tâche périodique (forcément de plus petite priorité) prend la CPU actif à t=4 avec une capacité >0 quand la tâche apériodique vient préempter et donc activer le serveur -> RT = 9
À t=9, on rétablit la capacité à 2

À t=10, le serveur redevient actif -> RT = 15

À t=15 capacité rétablie à 2. On peut servir la tâche tap3 et RT = 20

À t=20, on redonne 1 unité à la capacité

VIII.1. Exercice :

Soit deux tâches périodiques et un serveur sporadique (SS)

- Tp_1 : $t_1 = 0$, $C_1 = 1$, $T_1 = 5$
- Tp_2 : $t_2 = 0$, $C_2 = 4$, $T_2 = 15$
- SS : $C_s = 5$, $T_s = 10$

Soit les tâches apériodiques :

- $Ta_1 : t_{a1} = 4, C_{a1} = 2$
- $Ta_2 : t_{a2} = 8, C_{a2} = 2$

Ordonnancer ce système.

IX. Tâches apériodiques à contraintes strictes

Il existe deux méthodes : la première consiste à rapprocher le modèle des tâches **apériodiques** de celui des tâches **périodiques** en dotant celles-ci d'une pseudo-période, représentant l'intervalle minimal entre deux occurrences successives d'une tâche apériodique. La seconde traite les tâches apériodiques sans faire aucune supposition sur le rythme de leurs arrivées : à chaque occurrence de tâches apériodiques, un algorithme appelé « routine de garantie » **teste** si la nouvelle tâche pourra être garantie sans mettre en péril les tâches périodiques et les tâches apériodiques précédemment acceptées. Sinon, elle est rejetée.

Les types de contraintes qui seront étudiées dans cette section sont :

- Contraintes de précédence
- Contraintes liées au partage de ressources critiques

VIII.2. Ordonnancement de tâches liées par des contraintes de précédence :

Un premier type de contraintes entre les tâches d'une application temps réel est représenté par la notion de précédence. On dit qu'il existe une contrainte de précédence entre la tâche Ti et la tâche Tj ou Ti précède Tj, si Tj doit attendre la fin d'exécution de Ti pour commencer sa propre exécution. Les précédences entre tâches d'une application peuvent être modélisées sous forme d'un graphe de précédence (figure ci-dessous) tel qu'il existe un arc entre Ti et Tj si Ti précède Tj.

L'objectif des algorithmes de prise en compte des relations de précédence est de transformer l'ensemble de tâches avec relations de précédence en un ensemble de tâches indépendantes, en intégrant explicitement ou implicitement les relations de précédence dans l'affectation de priorité. Ainsi, on obtient les règles de précédence suivantes :

Si Ti précède Tj, alors $r_j \geq r_i$;
Si l'une des deux tâches est périodique, l'autre l'est obligatoirement et on a Pi = Pj ;
Priorité (Ti) > priorité (Tj) dans le respect de la politique d'ordonnancement utilisée.

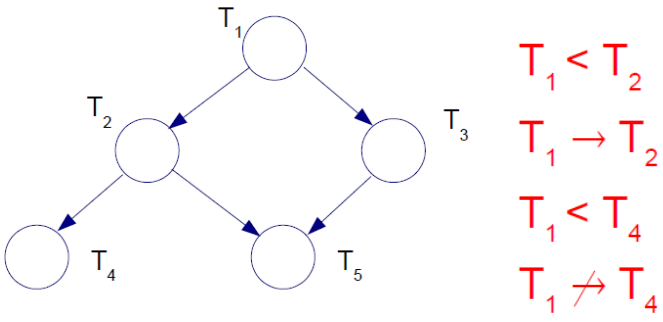


Figure 4 : Graphe de précédence

- $T_a < T_b$ indique que la tâche T_a est un prédécesseur de T_b
- $T_a \rightarrow T_b$ indique que la tâche T_a est un prédécesseur immédiat de T_b

X. Ordonnancement multiprocesseur

VIII.3. Position et formulation du problème

Définition d'un système multiprocesseur

L'environnement d'exécution des systèmes temps réel étudiés jusqu'à présent correspond à une architecture opérationnelle monoprocesseur.

Mais de nombreux domaines d'applications peuvent nécessiter plusieurs unités de traitement pour essentiellement deux raisons :

- Raison applicative : les contraintes de temps spécifiées dans le cahier des charges de l'application conduisent à utiliser plusieurs processeurs pour une exécution simultanée de certaines tâches. Après un parallélisme de conception (principe de la programmation multitâche), il peut être nécessaire de mettre en place un parallélisme d'exécution pour pouvoir répondre à la dynamique du procédé. Ce choix d'un environnement multiprocesseur concerne en général des applications complexes demandant le suivi de nombreux stimulus externes, comme les simulateurs de vol, par exemple.

- Raison sécuritaire : la sûreté de fonctionnement conduit à multiplier les équipements de contrôle pour diminuer la défaillance de l'ensemble du procédé. Ainsi, en cas de panne d'une ou plusieurs unités de traitement, il est alors possible d'incorporer des méthodes de dégradation élégante dans le sens où l'on obtient une continuité de service (exemple du domaine de l'aéronautique).

Nous nous plaçons dans le contexte des systèmes multiprocesseurs à contrôle centralisé ou encore appelés systèmes fortement couplés : leurs principales caractéristiques sont d'offrir une base de temps commune (ordonnancement global des événements et des tâches), une mémoire unique (vecteur de la communication entre les tâches) et, par conséquent, d'avoir une vue globale de l'état du système à chaque instant d'observation (figure 12). En plus de la mémoire commune qui contient l'ensemble des codes et des données des différentes tâches, les processeurs peuvent posséder des mémoires locales utilisées à l'exécution (pile, données transitoires, mémoires cache, etc.). Ces systèmes présentent des analogies fortes avec les systèmes centralisés (monoprocesseur) en se différenciant essentiellement par la capacité à mettre en œuvre un parallélisme d'exécution.

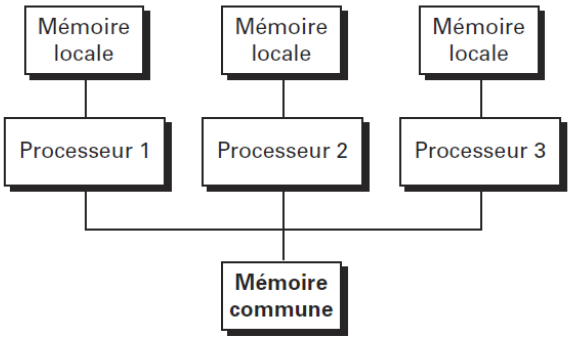


Figure 5 : Architecture matérielle d'un système multiprocesseur

Définition de l'ordonnancement multiprocesseur

Sur une architecture multiprocesseur, un algorithme d'ordonnancement est dit fiable si toutes les tâches de la configuration s'exécutent en respectant leur échéance. Cette définition, identique au cas monoprocesseur, est étendue avec les deux conditions suivantes :

- Un processeur ne peut traiter qu'une seule tâche à la fois ;
- Une tâche n'est traitée que par un seul processeur, à tout instant.

Critères de classification

Le choix d'un algorithme d'ordonnancement va être déterminé par la meilleure adéquation aux caractéristiques des tâches à ordonnancer et de l'architecture matérielle cible. Pour classer les algorithmes d'ordonnancement utilisés dans un environnement multiprocesseur, nous nous limitons aux principaux critères suivants :

- Architectures matérielles : nombre de processeurs (biprocesseur ou multiprocesseur), vitesses des processeurs (identiques, différentes) ;
- Caractéristiques des tâches : périodique/apériodique, avec contrainte de précédence, avec ou non-possibilité de préemption, possibilité ou non de changement de processeur en cours d'exécution (transfert du contexte d'exécution de la tâche : registres du processeur et mémoire locale) ;
- Exécution de l'algorithme : hors ligne, en ligne ;
- Critère optimisé : longueur de la séquence d'exécution, équilibrage de charge, etc.

Le cadre de l'étude présentée ici a été limité à la configuration la plus fréquente constituée de processeurs identiques (même vitesse de traitement considérée unitaire) avec un ordonnancement en ligne préemptif. Nous n'avons pas traité dans cette présentation les algorithmes d'ordonnancement hors ligne, souvent très complexes, qui sont, par définition, mal adaptés au temps réel. Il est toutefois important de souligner que les algorithmes hors ligne sont les seuls algorithmes qui permettent d'obtenir un ordonnancement optimal (résolution de problème d'optimisation de système linéaire) et de traiter certaines configurations non résolues par un algorithme en ligne.

VIII.4. Premiers résultats et comparaison avec l'ordonnancement monoprocesseur

Premiers résultats

Le premier résultat important est un théorème décrivant l'absence d'optimalité des algorithmes en ligne :

Théorème *: il ne peut exister d'algorithme en ligne qui construise une séquence valide sur une configuration matérielle comportant $m \geq 2$ processeurs pour une configuration de tâches temps réel à date critique.*

Ainsi nous pouvons en déduire que l'ordonnancement temps réel centralisé sur multiprocesseurs ne pourra pas être un ordonnancement optimal dans le cas général. Dans le cas d'une configuration T de tâches périodiques et indépendantes {Ti (ri, Ci, Ri, Pi) avec i ∈[1,n]} devant s'exécuter sur m processeurs, un deuxième résultat trivial est :

Condition nécessaire : la condition nécessaire d'ordonnançabilité faisant référence à la charge maximale *Uj* de chacun des processeurs ($U_j \leq 1 \; avec \; j \in [1, m]$) est que :

$$U = \sum_{j=1}^{m} U_j = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq m$$

Avec u_i facteur d'utilisation processeur par la tâche T_i.

Un troisième résultat concerne la période d'étude qui est identique à celui de l'environnement monoprocesseur :

Théorème *: il existe une séquence valide pour une* ***configuration T de tâches périodiques et indépendantes*** *si, et seulement si, il existe une séquence valide dans l'intervalle :*

$$[r_{min}, r_{max} + \Delta]$$

Avec :

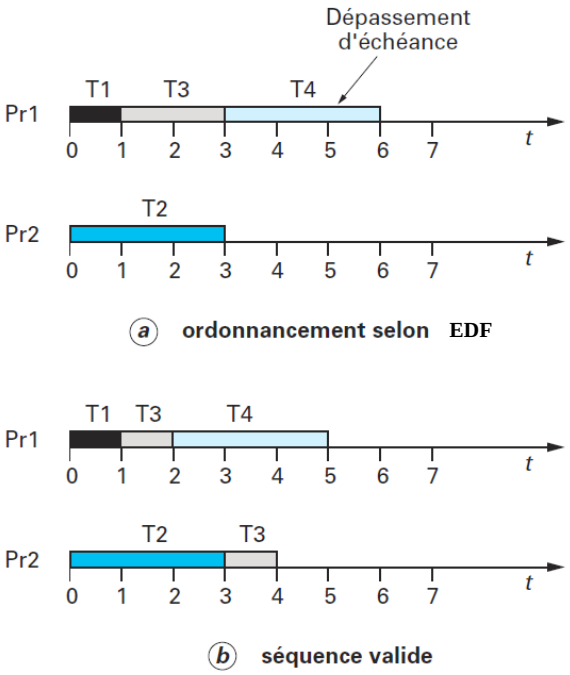
$$r_{min} = \min(r_i),$$
$$r_{max} = \max(r_i),$$
$$\Delta = \text{PPCM}(P_i),$$

$$i \in [1, n],$$

Ordonnancements monoprocesseur et multiprocesseur

Si nous considérons la stratégie d'ordonnancement EDF (Earliest Deadline First) optimale dans le cas monoprocesseur, celle-ci ne l'est plus dans le cas multiprocesseur. Soit l'exemple d'une configuration de quatre tâches Ti à ordonnancer sur deux processeurs (Pr1 et Pr2).

La séquence obtenue par ordonnancement selon EDF ne respecte pas l'échéance de la tâche 4 alors qu'il existe des séquences valides (figure 13).



T1 ($r = 0$, $C = 1$, $R = 2$, $P = 10$) T3 ($r = 1$, $C = 2$, $R = 4$, $P = 10$)
T2 ($r = 0$, $C = 3$, $R = 3$, $P = 10$) T4 ($r = 2$, $C = 3$, $R = 5$, $P = 10$)

Figure 6: Exemple montrant que l'algorithme EDF est non optimal en environnement multiprocesseur

VIII.5. Anomalies de l'ordonnancement multiprocesseur :

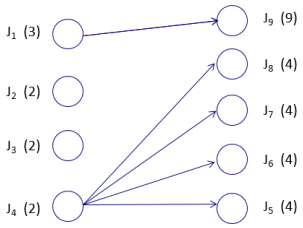
Il est très important de souligner que les applications, exécutées dans un environnement multiprocesseur, peuvent conduire à des **anomalies d'exécution** lors de changements apparemment positifs de paramètres. Ainsi, il a été démontré que :

> *« Si un ensemble de tâches est ordonnancé de façon optimale sur un système multiprocesseur avec des priorités assignées, des temps d'exécution fixées et des contraintes de précédence, alors le fait d'augmenter le nombre de processeurs, de réduire les temps d'exécution ou de relaxer les contraintes de précédence peut conduire à détériorer la durée d'exécution de l'algorithme ».* ***Théorème de Graham***

Exemple :

Afin de mettre en évidence toutes les anomalies qui peuvent subvenir dans un ordonnancement multiprocesseur, considérons l'exemple d'une configuration de neuf tâches préemptibles, mais sans possibilité de changement de processeur en cours d'exécution, qui sont à exécuter sur la première fois trois processeurs identiques et la deuxième fois sur quatre.

Soit neuf tâches de priorités décroissantes avec (Prio (J_i) > Prio (J_j)) ⇔ i > j



Calculer le temps d'exécution avec un ordonnancement multiprocesseur :

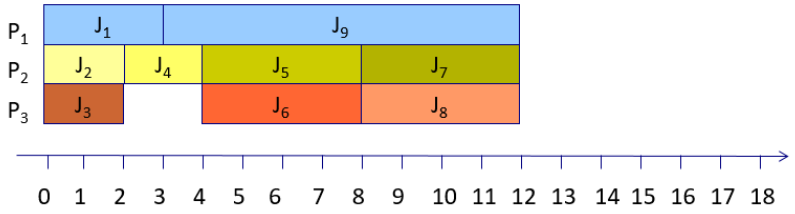


Figure 7 : Ordonnancement optimal sur un système à 3 processeurs :

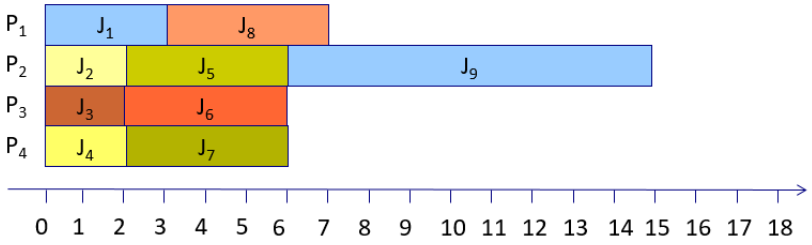


Figure 8 Figure 5 : Ordonnancement optimal sur un système à 4 processeurs :

La durée d'exécution de ces neuf tâches sur un système à trois processeurs est plus réduite que celui sur un système à 4 processeurs.

VIII.6. Contraintes de précédence et ordonnancement selon Rate Monotonic Analysis :

On transforme hors ligne l'ensemble de tâches liées par des contraintes de précédence en un ensemble de tâches indépendantes que l'on ordonnancera par un algorithme classique.

Cette transformation s'opère sur les paramètres date de réveil r et sur le délais critique pour donner :

$$r_i^* = \max\{(r_i, (r_j^* \, prédécesseur)\}$$
$$D_i^* = \max\{(D_i, (D_j^* \, prédécesseur)\}$$

Si Ti précède Tj, Þ Priorité (Ti) > Priorité (Tj) dans le respect de Rate Monotonic.

La validation de l'ordonnançable selon des critères utilisés pour des tâches indépendantes, contraintes de précédence selon et Rate Monotonic.

VIII.7. Contraintes de précédence et ordonnancement selon Earliest Deadline First :

On transforme hors ligne l'ensemble de tâches liées par des contraintes de précédence en un ensemble de tâches indépendantes pour lequel le test d'acceptabilité d'Earliest Deadline est valide.

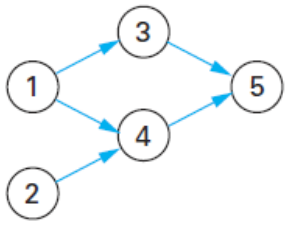
Cette transformation s'opère sur les paramètres date de réveil **r** et échéance **d** des tâches de l'ensemble pour donner :

$$d_i^* = \text{Min}(d_i, Min(d_j^* - C_j)) \quad \text{pour tous les j tels que } T_i \rightarrow T_j$$
$$r_i^* = \text{Max}(r_i, Max(r_j^* + C_j)) \quad \text{pour tous les j tels que } T_j \rightarrow T_i$$

On itère sur les prédécesseurs et successeurs immédiats et on commence les calculs par les tâches qui n'ont pas de prédécesseurs pour le calcul des **r** et par les tâches qui n'ont pas de successeur pour le calcul des **d.**

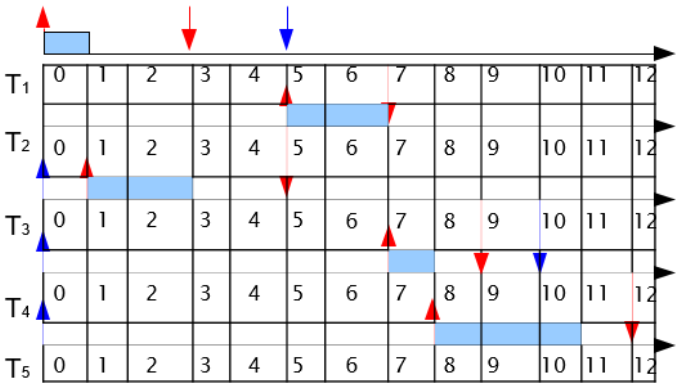
Exemple :

On considère les cinq tâches T1, T2, T3, T4 et T5 liées par le graphe de précédence donné à la figure ci-dessous. Le tableau **1** donne les paramètres des tâches et leurs transformations soit pour un ordonnancement Earliest Deadline First.



	Paramètres des tâches			Earliest Deadline First			
Tâche	r_i	C_i	d_i	r_i^*		d_i^*	
T1	0	1	5	0	T_1 n'a pas de prédécesseur : $r^*_1 = 0$	3	$d^*1 = \text{Min}(d_2, \text{Min}(d^*_3 - C_3, d^*_4 - C_4)) = 3$
T2	5	2	7	5	T_2 n'a pas de prédécesseur $r^*_2 = 5$	7	$d^*_2 = \text{Min}(d_2, \text{Min}(d^*_4 - C_4)) = 5$
T3	0	2	5	1	$r^*_3 = \text{Max}(r_3, r^*_1 + C_1) = 1$	5	$d^*_3 = \text{Min}(d_3, \text{Min}(d^*_5 - C_5)) = 5$
T4	0	1	10	7	$r^*_4 = \text{Max}(r_4, \text{Max}(r^*_1 + C_1, r^*_2 + C_2)) = 7$	9	$d^*_4 = \text{Min}(d4, \text{Min}(d^*_5 - C_5)) = 9$
T5	0	3	12	8	$r^*_5 = \text{Max}(r_5, \text{Max}(r^*_3 + C_3, r^*_4 + C_4)) = 8$	12	T_5 n'a pas de successeur : $d^*_5 = 12$

Ordonnancement de tâches liées par des contraintes de précédence :



VIII.8. Contraintes de précédence et ordonnancement selon Rate Monotonic

On transforme hors ligne l'ensemble de tâches liées par des contraintes de précédence en un ensemble de tâches indépendantes pour lequel le test d'acceptabilité de Rate Monotonic est valide.

Cette transformation s'opère sur les paramètres date de réveil $_r$ pour donner :

$$r_i^* = \text{Max}(r_i, (r_j^* \; predecesseur))$$

Si T_i précède T_j donc Priorité (Ti) > priorité (Tj) dans le respect de Rate Monotonic.

	Paramètres des tâches			Rate Monotonic	
Tâche	r_i	C_i	d_i	r_i^*	Priorité
T1	0	1	5	0	2
T2	5	2	7	5	1
T3	0	2	5	0	3
T4	0	1	10	5	3
T5	0	3	12	5	4

VIII.9. Ordonnancement de tâches liées par des contraintes de ressources :

L'ordonnancement de tâches avec contraintes de ressources pose différents problèmes dans tout type de systèmes informatiques :

- Des problèmes de synchronisation entre tâches et, notamment, d'inversion de priorité pour les ressources accédées en exclusion mutuelle ;
- Des problèmes d'interblocage.

Dans un système temps réel, la réservation et la pré allocation des ressources sont des méthodes simples pour éviter ces problèmes, méthodes par lesquelles une tâche dispose dès le début de son exécution de toutes les ressources qui lui seront nécessaires au cours de celle-ci. Cependant, ces méthodes engendrent généralement un très faible taux d'utilisation des ressources ; aussi des protocoles spécifiques ont été développés pour permettre d'effectuer une allocation dynamique des ressources.

Lorsqu'une ressource est accédée en exclusion mutuelle, il peut se produire le phénomène qualifié ***d'inversion de priorité***, phénomène au cours duquel une tâche de haute priorité voulant accéder à la section critique associée à la ressource est retardée par des tâches de moindre priorité dont une au moins possédant la section critique. Ce phénomène est inhérent à tout système informatique, mais il devient problématique dès que des contraintes temporelles sont associées aux tâches, car il induit des retards non maîtrisés. La figure ci-dessous donne un exemple de ce phénomène pour trois tâches T_1, T_2, T_3 telles que la tâche T_1 est la tâche la plus prioritaire et T_2 la tâche la moins prioritaire, T_1, T_2 partageant la section critique et T_3 n'utilisant pas la ressource.

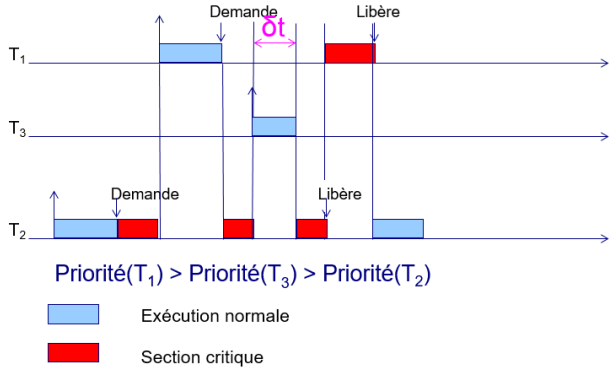


Figure 9: Partage de ressources

À l'instant 0, seule la tâche T_2 est réveillée et donc elle s'exécute. Au bout d'un certain temps d'exécution, la tâche T_2 demande la ressource et comme celle-ci est disponible, la tâche T_2 obtient la ressource. T_2 poursuit son exécution jusqu'au réveil de T_1 qui est plus prioritaire et donc préempte T_2. T_1 s'exécute jusqu'à l'instant où elle demande à accéder à la ressource. Comme est déjà allouée à T_2, la tâche T_1 est bloquée et T_2 reprend son exécution. Lorsque la tâche T_3 se réveille à son tour, elle préempte la tâche T_2 et s'exécute entièrement, retardant d'autant la tâche T_2 et, en conséquence, la tâche T_1.

On voit ainsi que T_1 est retardée par T_3 et éventuellement par toute autre tâche de priorité intermédiaire n'accédant pas à ressource. Plusieurs protocoles ont été définis pour pallier ce problème d'inversion de priorité, les tâches étant ordonnancées soit par l'algorithme Rate Monotonic, soit par l'algorithme Earliest Deadline.

Chacun de ces protocoles permet de définir, pour chaque tâche, des temps de blocage maximaux **δt** induits par l'accès aux sections critiques.

Ces temps de blocage maximaux sont ensuite intégrés aux tests d'acceptabilité des algorithmes Rate Monotonic et Earliest Deadline, en considérant qu'une tâche possède un temps d'exécution allongé de cette durée maximale de blocage, soit $C_i + \mathbf{\delta t}$. Par ailleurs, ces protocoles tentent également de prévenir la formation des interblocages.

VIII.10. Inversion de priorité

Dans un ordonnancement préemptif, la présence simultanée de priorités fixes et de ressources à accès exclusif peut entraîner un phénomène appelé inversion de priorité figure ci-dessous.

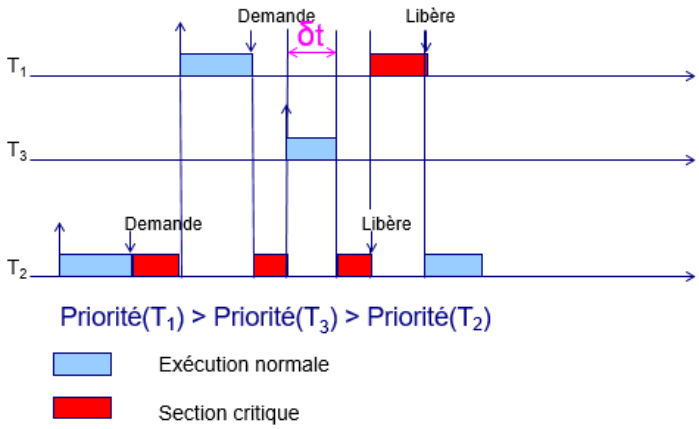


Figure 10 : Inversion de priorité

Exemple :

Soit trois tâches de priorité de décroissantes, T_1, T_3 et T_2, A un instant donné seule T_2 est déclenchée et a obtenu une ressource critique. Puis T_1 est réveillée par un évènement externe par exemple. L'ordonnanceur préemptif à priorités élit T_1. Supposons que T_1 réclame la ressource critique déjà allouée à T_2, Comme cette ressource ne peut être réquisitionnée pour T_1, cette tâche doit attendre que T_2 la libère. L'ordonnanceur à priorité élit donc T3, qui s'exécute. Ce n'est qu'après son exécution que l'ordonnanceur peut élire T_2 qui peut alors finir d'utiliser la ressource critique et, enfin, la libérer. Et seulement alors, T_1, la tâche la plus prioritaire, peut s'exécuter. Le temps de réponse de T_1 s'est rallongé du temps d'exécution de T3, moins prioritaire qu'elle. Cela peut parfois entraîner un **dépassement** d'échéance.

Comme les utilisations de ressources critiques sont souvent imbriquées, toute solution à ce problème doit en tenir compte.

VIII.11. Interblocage

Supposons que les deux tâches T_1 et T_2 ont besoin chacune de deux ressources exclusives R_a et R_b et que ces ressources ne peuvent pas être retirées à une tâche qui les utilise en section critique.

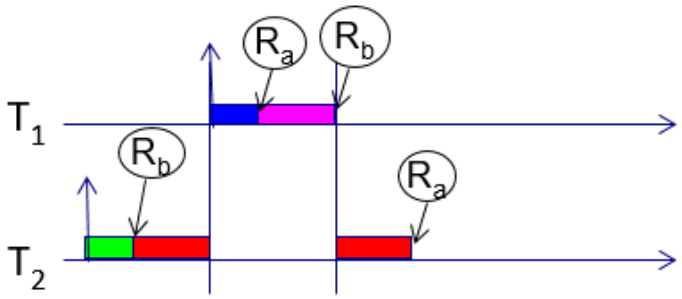


Figure 11: Interblocage

Si T1 demande successivement Ra et R_b et si T_2, au contraire, demande R_b puis R_a, les deux tâches peuvent parfois s'attendre mutuellement indéfiniment comme dans le cas décrit sur la figure 9. C'est une situation d'interblocage, qui peut se généraliser à

tous les cas d'attente circulaire entre plusieurs tâches, et dont on ne peut sortir qu'en détruisant une ou plusieurs tâches.

Pour une application temps réel, c'est une situation inacceptable.
Une méthode, appelée la méthode des classes ordonnées, permet d'éviter que l'interblocage puisse se produire, en programmant la demande des ressources selon le même ordre pour toutes les tâches. Elle n'est applicable que si l'on connaît, dès la programmation, toutes les ressources que les tâches vont appeler. C'est pourquoi on dit que c'est une méthode de prévention statique.

VIII.12. Protocole de l'héritage de priorité

Dans ce protocole, la tâche T′ qui possède la section critique prend la priorité de la tâche T en attente sur la section critique, si la priorité de T est supérieure à la sienne, ce jusqu'à libérer la section critique. Ainsi, la tâche T′ est ordonnancée à un niveau plus élevé que son niveau initial, afin de libérer au plus vite la section critique et, donc, afin de minimiser l'attente de la tâche T.

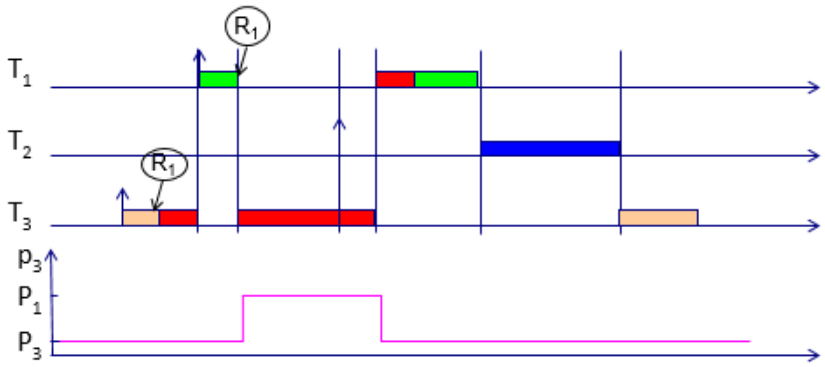


Figure 12: Inversion de priorité

La figure 10 donne un exemple de l'application de ce protocole pour les trois mêmes tâches T_1, T_2, T_3. Comme précédemment, la tâche T_3, réveillée à l'instant 0, commence son exécution, demande la ressource et l'obtient. Un peu plus tard, la tâche T_1 se réveille et préempte la tâche T_3. Lorsque T_1 demande la ressource, la tâche T_1 se bloque puisque est allouée à T_3 mais, dans le même temps, la tâche T_2 hérite de la priorité de T_1 et poursuit son exécution avec cette nouvelle priorité. Ainsi, lorsque la tâche T_2 se réveille, elle se trouve maintenant être de priorité inférieure à T_3 et ne peut plus préempter T_3. La tâche T_3 poursuit donc son exécution, libère la ressource et retrouve sa priorité initiale.

La tâche T_1 débloquée prend alors la main. On montre que, pour une tâche T_i, le temps de blocage maximal B_i sur l'attente d'une ressource est égal au plus à la somme des durées des sections critiques partagées avec des tâches de plus faible priorité. Ainsi, on peut voir sur l'exemple de la figure 10, que l'exécution de la tâche T_1 est au plus retardée d'une durée égale à la section critique de T_2.

Dans ce protocole, chaque ressource critique possède initialement une priorité qui est celle de la tâche de plus haute priorité pouvant demander son accès. Cette priorité se

comporte comme un seuil. Dès qu'une tâche T accède à la ressource, elle prend comme priorité d'ordonnancement la priorité de la ressource, et ce jusqu'à la libération de celle-ci. Par ailleurs, afin de prévenir les interblocages et les chaînes de blocage, la tâche T ne peut accéder à une ressource que si la priorité de la ressource est supérieure à celles de toutes les ressources déjà possédées par d'autres tâches T_i.

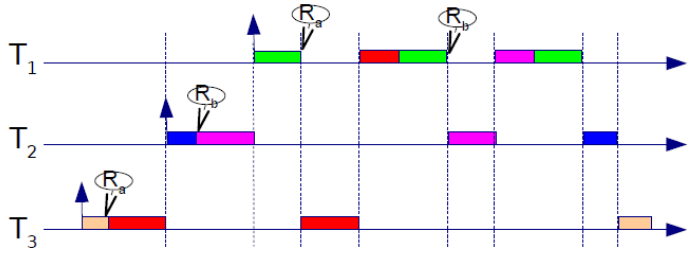
On montre que, pour une tâche T_i, le temps de blocage maximal $\delta\mathbf{t}$ sur l'attente d'une ressource est égal à la durée de la plus grande des sections critiques d'une tâche moins prioritaire dont la priorité plafond d'une de ses ressources est au moins aussi grande que la priorité de la tâche.

le temps de blocage B d'une tâche de haute priorité par une tâche de plus basse priorité nominale est borné, mais le calcul de ce temps de blocage maximum peut être très complexe et la condition nécessaire d'ordonnançable par un algorithme Rate Monotonic:

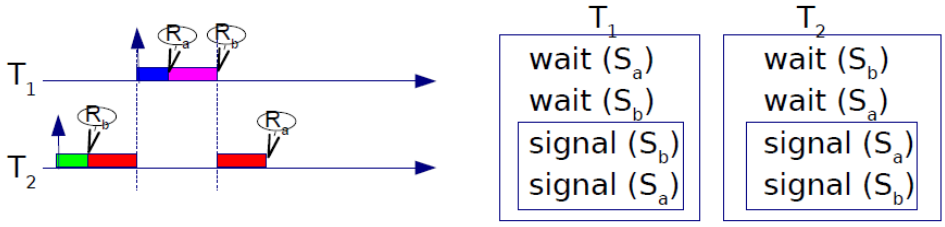
$$\forall i, 1 \leq i \leq n, \left(\sum_{k=1}^{i} \frac{C_k}{T_k}\right) + \frac{B_i}{T_i} \leq i(2^{\frac{1}{i}} - 1)$$

Exemple de problèmes rémanents

- Blocage en chaine :



- Etreinte fatale (deadlock) :
  Si deux tâches utilisent deux ressources imbriquées, mais dans l'ordre inverse.



VIII.13. La priorité plafonnée

Dans ce protocole, chaque ressource critique possède initialement une priorité qui est celle de la tâche de plus haute priorité pouvant demander son accès. Cette priorité se comporte comme un seuil. Dès qu'une tâche T accède à la ressource, elle prend comme priorité d'ordonnancement la priorité de la **ressource**, et ce jusqu'à la libération de celle-ci. Par ailleurs, afin de prévenir les interblocages et les chaînes de blocage, la tâche T ne peut accéder à une ressource que si la priorité de la ressource est supérieure à celles de toutes les ressources déjà possédées par d'autres tâches Ti. Les deux

chercheurs Chen et Lin[4] présentent une adaptation du protocole initialement conçu pour des algorithmes à priorités constantes tels que Rate Monotonic aux ordonnancements à priorités variables tels que Earliest Deadline First.

Dans ce cas, le seuil de chaque ressource est recalculé à chaque fois que l'ensemble des tâches actives du système est modifié, ce qui correspond aux instants de terminaison et de réveil de tâches.

L'intégration de ce protocole dans sa version statique a été réalisée dans le système Mach par l'équipe de Carnegie Mellon[5]. Il faut, par ailleurs, noter que les propriétés de ce protocole ne sont vraies que pour un **système monoprocesseur**.

Rappel :

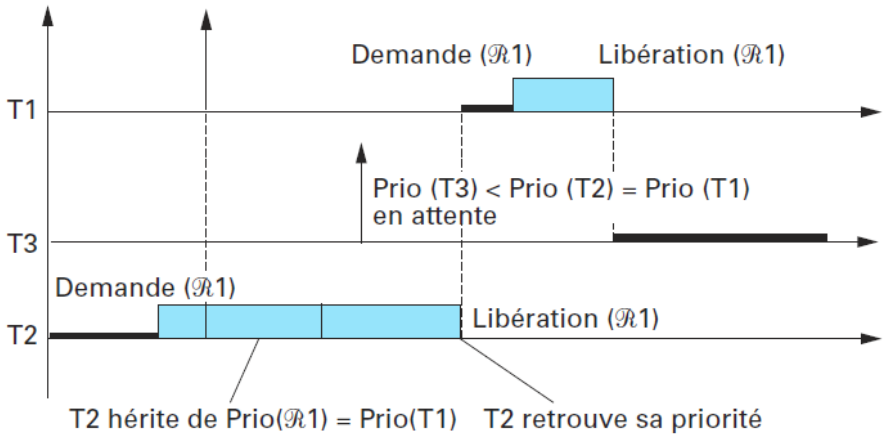


Figure 13: Inversion de priorité : application du protocole de la priorité plafonnée

On montre que, pour une tâche Ti, le temps de blocage maximal Bi sur l'attente d'une ressource est égal à la durée de la plus grande des sections critiques d'une tâche moins prioritaire dont la priorité plafond d'une de ses ressources est au moins aussi grande que la priorité de la tâche.

La figure 11 donne un exemple de l'application de ce protocole pour les trois mêmes tâches T_1, T_2, T_3. La ressource critique R_1 reçoit comme priorité la priorité de T_1.

Comme précédemment, la tâche T_2, réveillée à l'instant 0, commence son exécution, demande la ressource et l'obtient. Elle prend alors, comme nouvelle priorité d'exécution, la priorité de R_1. Un peu plus tard, la tâche T_1 se réveille, sa priorité est égale à celle de T_2 et donc T_2 poursuit son exécution. Lorsque la tâche T_3 se réveille, elle est mise en attente puisque sa priorité est plus petite que celle de T_2. T_2 poursuit donc son exécution, libère la ressource R_1 et retrouve sa priorité initiale.

La tâche T_1, qui est à présent la tâche de plus haute priorité, prend alors la main.

4 CHEN (M.I.) et LIN (K.J.). – Dynamic priority ceilings: A concurrency control protocol for real-time systems. Real-Time systems Journal, 2 (4) p. 325-346 (1990).
5 L'université de Carnegie Mellon

Rappel : le concept de sémaphore

Un sémaphore est une variable entière partagée. Sa valeur est positive ou nulle et elle est uniquement manipulable à l'aide de deux opérations wait(s) et signal(s), ou s est un identificateur désignant le sémaphore.

- wait(s) décrémente s si s > 0, si non, le processus exécutant l'opération wait(s) est mis en attente.
- signal(s) incrémente s. L'exécution de signal(s) peut avoir pour effet (pas nécessairement immédiat) qu'un processus en attente sur le sémaphore s reprend son exécution.
- L'exécution d'une opération wait(s) ou signal(s) se fait sans interaction possible (de façon atomique).

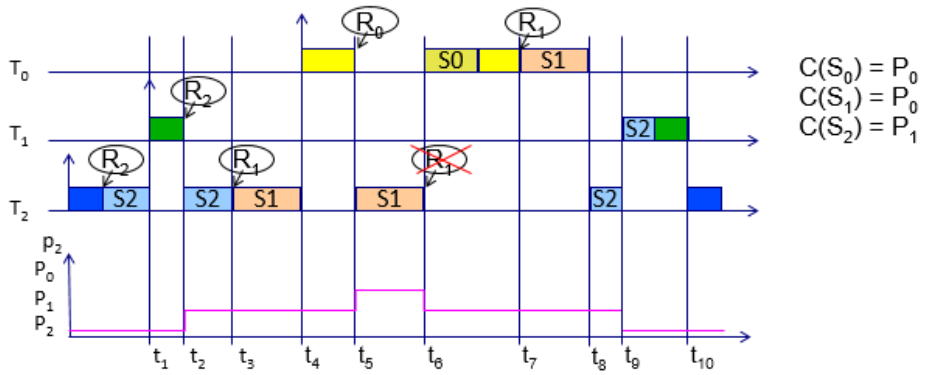
Example :

Trois tâches T_0, T_1, T_2 de priorités décroissantes $P_0 > P_1 > P_2$

- T_0 accède séquentiellement à 2 sections critiques gardées par les sémaphores S_0 et S_1.
- T_1 accède à une section critique gardée par S_2.
- T_2 accède à la section critique gardée par S2 et aussi, de manière imbriquée, a S_1.

Les priorités plafond sont :

- $C(S_0) = P_0$
- $C(S_1) = P_0$
- $C(S_2) = P_1$



- A t_0, T_2 est activée et démarre. Le sémaphore S_2 est demandé et obtenu (il n'y a pas d'autre ressource en cours d'utilisation)
- A t_2, T_1 demande S_2 et est bloquée par le protocole car la priorité de T_1 n'est pas supérieure à la priorité plafond $C(S_2)=P_1$ (S_2 est le seul sémaphore acquis à t_2) T_2 hérite de la priorité de T_1 et redémarre
- A t_3, T_2 demande et obtient le sémaphore S_1 , car aucune autre tâche ne détient de ressource
- A t_4 T_0 est réveillée et préempte T_2
- A t_5, T_0 demande le sémaphore S_0 qui n'est détenu par aucune autre tâche. Le protocole bloque néanmoins T_0 parce que sa priorité n'est pas supérieure à la

plus grande priorité plafond des sémaphores déjà détenus (S_2 et S_1) : P_0 T_2 hérite de la priorité de T_0 et peut redémarrer

- A t_6, T_2 relâche S_1. Sa priorité p_2 est ramenée à P_1, plus haute priorité des tâches bloquées par T_2
  T_0 est réveillée et peut alors préempter T_2 et acquérir S_0 puisque la priorité plafond de S_2 (seul sémaphore encore pris) est égale à P_1
- A t_7 quand T_0 demande S_1, le seul sémaphore encore tenu est S_2 avec une priorité plafond $P_1 => T_0$ (priorité $P_0 > P_1$) obtient S_1
- A t_8, T_0 se termine. T_2 peut reprendre son exécution, toujours avec la même priorité P_1
- A t_9, T_2 relâche S_2. Sa priorité est ramenée à sa valeur nominale P_2. T_1 peut alors préempter T_2 et redémarrer
- A t_{10}, T_1 se termine, T_2 peut redémarrer et se terminer

On remarque qu'on a le même critère d'ordonnançable par RMA que dans le cas du protocole d'héritage de priorité mais le calcul du temps de blocage maximum de chaque tâche est plus simple :

$$\forall i, 1 \leq i \leq n, \left(\sum_{k=1}^{i} \frac{C_k}{T_k} \right) + \frac{B_i}{T_i} \leq i(2^{\frac{1}{i}} - 1)$$

On peut aussi démontrer que le temps de blocage maximum B_i d'une tâche T_i est la durée de la plus longue des sections critiques parmi celles appartenant à des tâches de priorité inférieure à P_i et gardées par des sémaphores dont la priorité plafond est supérieure ou égale à P_i

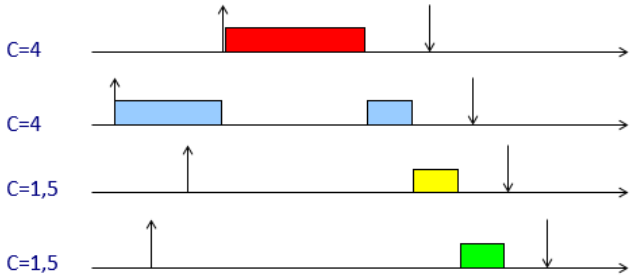
$$B_i = max_{j,k}\{D_{j,k} \mid P_j < P_i, C(S_k) \geq P_i\}$$

XI. Ordonnancement de tâches périodiques et apériodiques avec prise en compte des surcharges :

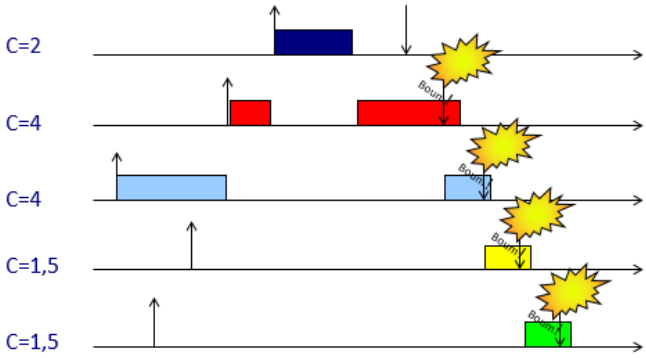
Une faute temporelle est générée lorsqu'une tâche dépasse son échéance. On dit alors que le système est en situation de surcharge.

Une faute temporelle peut être générée, par exemple, par suite d'une transmission défectueuse de communications, et donc de la réception tardive d'un signal nécessaire à la poursuite de l'exécution d'une tâche ou encore par suite d'une contention sur une ressource critique entraînant un allongement du temps d'exécution de la tâche. Une faute temporelle peut encore être générée par des réveils de tâches apériodiques, dus à une situation d'alarme sur le procédé contrôlé. Par ailleurs, lors d'une faute temporelle, il est souhaitable de supprimer la tâche fautive, sinon la faute risquerait de se propager à d'autres tâches.

Cette escalade de fautes temporelles est appelée *l'effet domino* dû à l'arrivée d'une tâche supplémentaire dans un ordonnancement par exemple EDF.



Les algorithmes à priorités classiques (du type EDF ou RMA) offrent des performances souvent médiocres en cas de surcharge :



Plusieurs études portent sur la tolérance aux fautes temporelles. Les premières concernent exclusivement les configurations de tâches périodiques et permettent de supporter des temps d'exécution variables, non forcément bornables. Les autres, qui utilisent un critère d'importance, image du caractère primordial des tâches dans l'application, s'appliquent à des configurations de tâches périodiques et apériodiques.

Dans un contexte temps réel préemptible de n tâches périodiques indépendantes à échéance sur requête, la charge est équivalente au facteur d'utilisation U :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

Dans le cas général elle est basée sur le fait que pour une tâche unique de capacité C_i et de délai critique D_i, la charge dans l'intervalle $[r_i, d_i]$ est $\rho_i = C_i/D_i$ avec $(d_i = r_i + D_i)$. Donc pour calculer au réveil des dates à des dates $t=r_i$

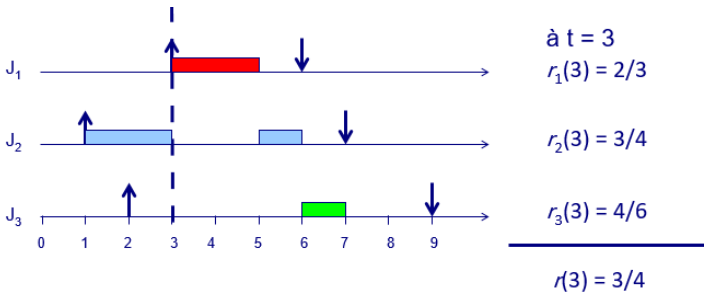
$$\rho_i(t) = \frac{\sum_{d_k \le d_i} c_k(t)}{(d_i - t)}$$

$$\rho = \max(\rho_i(t))$$

Exemple :

Soit trois Tâches :

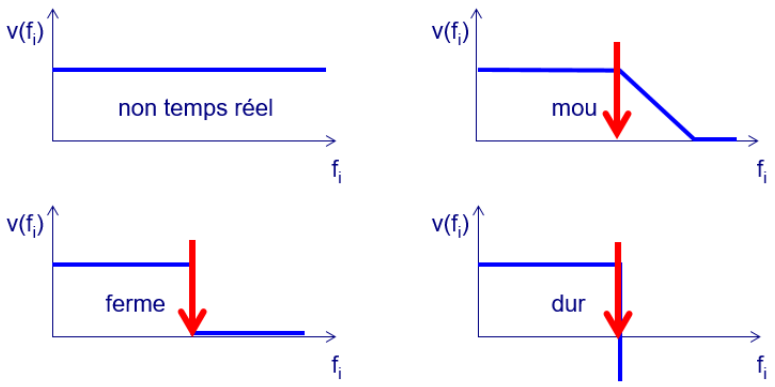
- J_1 : ($C_1 = 2$, $r_1 = 3$, $d_1 = 6$)
- J_2 : ($C_2 = 3$, $r_2 = 1$, $d_2 = 7$)
- J_3 : ($C_3 = 1$, $r_3 = 2$, $d_3 = 9$)



En l'absence de possibilité de surcharge, l'optimalité d'un algorithme est facile à définir et l'importance relative des tâches n'a pas d'intérêt mais en présence d'un système où l'arrivée dynamique de tâches est autorisée, il n'y a pas d'algorithme qui puisse garantir la bonne exécution de l'ensemble des tâches par exemple l'importance de l'exécution d'une tâche ne peut pas se définir seulement à l'aide de son échéance par exemple il est sûrement plus important de faire la mesure d'un capteur toutes les 10s que de mettre à jour l'heure sur l'écran de contrôle toute les secondes.

Paracerque Notion de surcharge nécessitée de définir une valeur associée à la tâche qui reflète son importance relative par rapport aux autres tâches dans les tâches temps réel, l'échéance fait partie aussi de la définition de la valeur de la tâche dans les tâches temps réel, l'échéance fait partie aussi de la définition de la valeur de la tâche d'où intérêt de définir une "**fonction d'utilité**" qui décrit l'importance de la tâche en fonction du temps.

Exemple des Tâches :



Nous pouvons évaluer la performance d'un algorithme par :

$$\Gamma_A = \sum_{i=1}^{n} v(f_i) \quad (1)$$

Avec $v(f_i)$ est l'utilité de la tâche à sa terminaison.

Remarque : si une Tâche temps réel du dépasse son échéance, alors Γ_A $vaut$ $-\infty$.

Il faut réserver les ressources (entre autres la CPU) pour garantir les exigences des Tâches dures. Alors pour un ensemble de n tâches J_i (C_i, D_i, V_i), où V_i est l'utilité de la tâche quand elle se termine avant son échéance, la valeur maximale de Γ_A est (1).

En conditions de charges, la qualité d'un algorithme A se mesure en comparant Γ_A à Γ_{max} .

VIII.14. Gestion des situations de surcharge

Prenant par exemple une gestion de trafic urbain repose sur trois processeurs qui déroulent périodiquement, toutes les 0,3 secondes, trois tâches d'analyse occupant chacune jusqu'à 60 % du temps de chaque processeur. Devant le succès de cette gestion du régime permanent et la charge raisonnable des processeurs, on souhaite l'étendre à la gestion de situations de surcharge causées par des cas urgents (police, pompiers, officiels) ou par des pointes de trafic. Pour cela, il faut ajouter une tâche de recherche d'itinéraire, une tâche de commande des feux et une tâche de suivi des voitures de pompiers. Ces tâches ajoutées font passer la charge totale à 275 % (pour trois processeurs, on est au-dessous de 300 %), mais les contraintes de période et d'échéance sont telles qu'on ne peut trouver un placement statique des tâches et qu'on ne peut s'en sortir qu'en plaçant dynamiquement sur des processeurs différents les requêtes successives des tâches ajoutées. Cette adaptation aux surcharges n'est possible que si l'on dispose d'un exécutif qui sache gérer le placement dynamique des requêtes.

Donc en présence d'un système où l'arrivée dynamique de tâches est autorisée, il n'y a pas d'algorithme qui puisse garantir la bonne exécution de l'ensemble des tâches, seulement des politiques plus ou moins bien adaptées aux circonstances

- ➢ Tâches périodiques
- ➢ Tâches quelconques

Tâches périodiques

Les tâches périodiques sont déclenchées à la date de réveil des requêtes successives et redeviennent passives une fois la requête terminée. Les tâches apériodiques peuvent avoir le même comportement si elles peuvent se déclencher plusieurs fois ; elles peuvent aussi plus rarement être créées au moment du réveil. Autrement dit, il n'y a pas d'arrivée dynamique de nouvelle tâche et les durées d'exécutions variables, non forcément bornables, seulement deux politiques présentées :

- ➢ Méthode du mécanisme à échéance
- ➢ Méthode du calcul approché

Méthode du mécanisme à échéance

Plusieurs modèles de tâches ont été introduits pour éviter qu'en cas de faute temporelle tout le travail réalisé par la tâche avant la panne ne soit perdu.

Dans le plus ancien modèle proposé, chaque tâche a deux versions, une **version primaire** qui fournit le service demandé mais dont l'exécution peut entraîner une faute temporelle parce que la durée d'exécution peut varier considérablement, et **une version secondaire** qui fournit un service dégradé mais acceptable et sans faute temporelle. Selon les politiques d'ordonnancement, on essaie d'abord l'une ou l'autre version mais, dans tous les cas, il faut que la version secondaire s'exécute sans faute.

Une autre approche est celle du calcul imprécis (ou progressif ou par contrat) où chaque tâche est décomposée en deux parties, la partie mandataire et la partie optionnelle. La partie mandataire doit obligatoirement s'exécuter dans le respect de son échéance et elle fournit un résultat approché. La partie optionnelle affine ce premier résultat et est exécutée pendant le temps restant avant l'échéance.

L'algorithme de tolérance aux fautes doit assurer le respect de toutes les échéances, soit par le primaire soit par le secondaire par exemple si les deux marchent, on garde le primaire, si le primaire échoue, le secondaire doit réussir Donc :

Ordonnancement = juxtaposition d'une séquence des primaires et d'une des secondaires + règle de décision pour commuter de l'une à l'autre.

Dans une approche récente, une tâche présente plusieurs modes de fonctionnement, un mode normal et un mode de survie qui est exécuté quand une requête est supprimée, soit parce que son échéance est arrivée, soit parce qu'elle a été sacrifiée au profit d'une autre tâche plus importante. Le mode de survie a une durée très courte mais il permet à la tâche de rendre une ressource partagée ou de terminer un calcul dans un état exploitable. L'échéance de la tâche est fixée pour permettre l'exécution du mode normal et du mode de survie.

Tâches quelconques

Ce modèle canonique des tâches insuffisant, basé sur urgence (délai critique) un nouveau paramètre est nécessaire est *importance*, codant le caractère primordial de la tâche dans l'application et sa définition est subjective selon les exigences exprimes au cahier des charges.

La cause fréquente des fautes temporelles : occurrence d'une tache apériodique il y'a une possible de les rejeter par une routine de garantie, éventuellement vers un autre processeur moins charge dans un environnement reparti mais le mécanisme est lourd et l'ordonnancement a importance.

On distingue trois classes de politiques pour décider l'acceptation (ou le rejet) des nouvelles taches :

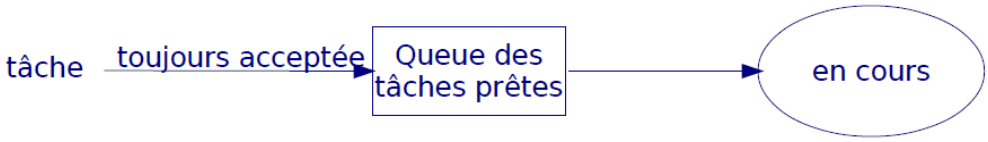
- Meilleur effort
- Avec garantie

- Robuste

Meilleur effort

Pour une application à contraintes temporelles relatives, on dit que la stratégie d'ordonnancement est celle du meilleur effort : elle essaie de faire au mieux avec les processeurs disponibles, pas de prédiction en cas de surcharge, les nouvelles tâches sont toujours acceptées, le seul moyen d'action : les niveaux relatifs des priorités

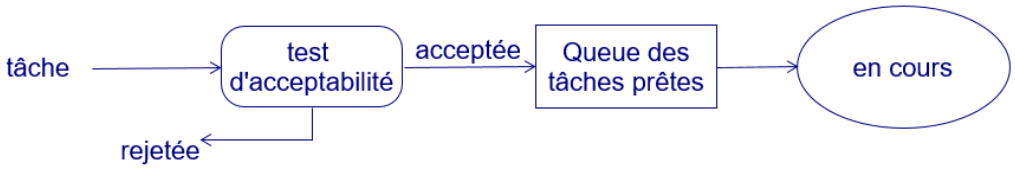
.



Avec garantie

L'ensemble des conditions à satisfaire est appelé ***test d'acceptabilité***. On dit parfois test de garantie car, si les tâches ne dépassent pas leurs durées d'exécution (auxquelles s'ajoutent les durées d'attente pour obtenir les ressources critiques), on peut garantir alors que les tâches ne font pas de faute temporelle.

Dans un ordonnancement réparti, le rejet d'une tâche par un test d'ordonnançabilité sur un site peut avoir comme conséquence d'essayer de faire migrer la tâche.



Robuste

On parle d'un système robuste (il n'est pas sujet à des anomalies d'ordonnancement pour le système de tâches considéré). Il y'a une séparation des contraintes temporelles et de l'importance des taches , il existe deux politiques : une pour l'acceptation de nouvelles taches, une pour le rejet de taches, quand une nouvelle tâche arrive, on exécute un test d'acceptabilité , si le test passe, la tâche est mise dans la queue des taches prêtes, si le test ne passe pas, on exécute l'algorithme de rejet qui permet d'éliminer une ou plusieurs taches de moindre importance, et éventuellement, recyclage des taches rejetées pour profiter d'une exécution plus rapide que prévue d'une tache acceptée

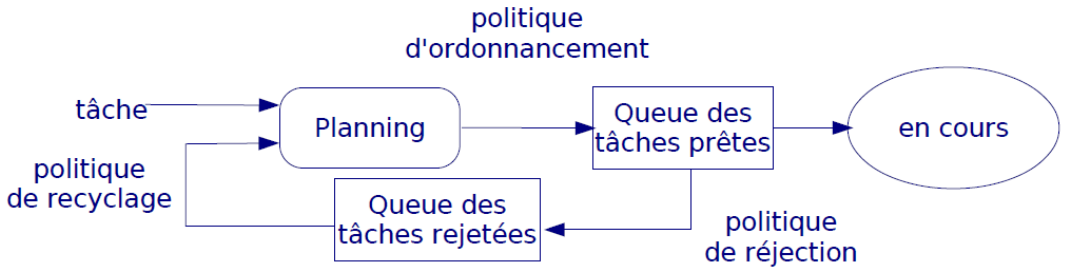

politique
d'ordonnancement
tâche
Planning
Queue des
tâches prêtes
en cours
politique
de recyclage
Queue des
tâches rejetées
politique
de réjection

Table des matières :

Bibliographie :

- Solutions temps réel sous Linux, 2ème édition, éditions Eyrolles
- Le temps réel en milieu industriel, éditions Dunod-Systèmes
- Temps réel de contrôle-commande, éditions Dunod