Module Master M1

Systèmes temps réel et Informatique Industrielle
Chapitre II : Ordonnancement temps réel

Présenté par : Prof. Kholladi Mohamed-Khireddine
Département d'Informatique
Facultés des Sciences Exactes
Université Echahid Hamma Lakhdar d'El Oued
Tél. 0770314924
Email. kholladi@univ-eloued.dz et kholladi@yahoo.fr
Site Web. www.univ-eloued.dz
http://kholladi.doomby.com/ et http://kholladi.e-monsite.com/

II – Ordonnancement temps réel

II.1 – Introduction

a - Qu'est-ce qu'un ordonnanceur ?

L'ordonnanceur est la partie logicielle d'une application temps réel, qui est un ensemble de tâches synchronisées, communicantes et partageant des ressources critiques. Un rôle essentiel d'un STR est donc de gérer l'enchaînement et la concurrence des exécutions des tâches, en optimisant l'occupation des unités de traitement : c'est la fonction de l'ordonnanceur. On appelle ordonnanceur (scheduler) le processus système qui gère l'ordonnancement des processus. Un algorithme d'ordonnancement est une méthode ou une stratégie utilisée pour ordonnancer les processus. Un tel algorithme s'appuie sur la connaissance de certaines caractéristiques des processus ou du système :

1. Processus périodiques ou apériodiques;
2. Processus cyclique ou non cyclique;
3. Préemption possible ou pas;
4. Échéance et pire temps d'exécution des processus
5. Système à priorité fixe ou à échéance.

b - Nature des traitements

Sporadiques

- Arrêt d'urgence
- Changement de mode de fonctionnement
- Traitement par interruption

Périodiques

- Acquisition de capteurs (image, poids, température)
- Calcul de position
- Calcul de consigne

c - Classes d'ordonnancement

On distingue cinq classes d'ordonnancement comme sur le tableau ci-dessous. Une politique d'ordonnancement est un algorithme qui détermine en ligne l'activité à exécuter - propriétés requises : décidabilité et implémentabilité. Une politique est optimale si elle conduit à un ordonnancement faisable s'il en existe un.

N°	Classes d'ordonnancement	Signification
1	Hors-ligne	la séquence des exécutions est décidée à la conception puis déroulée pendant l'exécution.
2	En ligne	les choix d'ordonnancement sont faits en ligne, ils utilisent certaines caractéristiques des tâches.
3	Préemptif (respectivement non-préemptif)	une tâche peut être interrompue par une autre + prioritaire.
4	Non-oisif (respectivement oisif)	le processeur est nécessairement utilisé s'il y a du travail en attente d'exécution.
5	Déterministe	le hasard n'intervient pas dans les choix d'ordonnancement.

d - Deux algorithmes classiques d'ordonnancement

Il y a deux algorithmes classiques d'ordonnancement RMT et EDF comme sur le tableau suivant :

RTM	RaTe Monotonic	Algorithme à priorité fixe pour processus cycliques (le processus le plus prioritaire est celui de plus petite échéance).
EDF	Earlest Deadline First	Algorithme à priorité dynamique pour processus cycliques (le processus le plus prioritaire est celui de plus petite échéance).

L'ordonnanceur choisit d'exécuter le processus prêt de plus haute priorité. Au sein d'une même classe de priorité, le choix peut se faire par temps partagé (Round Robin) ou par ancienneté (gestion FIFO).

- Problème : Calculs à exécuter + contraintes de temps = dans quel ordre exécuter ?
- Définition : Ensemble des règles définissant l'ordre d'exécution des calculs sur le processeur.
- Pourquoi ordonnancer ? Parce que ça à un impact sur le respect des contraintes de temps.

Exemple

Tâche T_1 : arrivée en zéro, durée quatre, échéance sept.

Tâche T_2 : arrivée en deux, durée deux, échéance cinq.

Ordonnancement O_1 : premier arrivé, premier servi, on n'interrompt jamais une tâche.

Ordonnancement O_2 : priorité, T_2 plus prioritaire que T_1.

Solution

Ordonnancement **O_1** : **T_2** rate son échéance (comme sur la figure II.1).

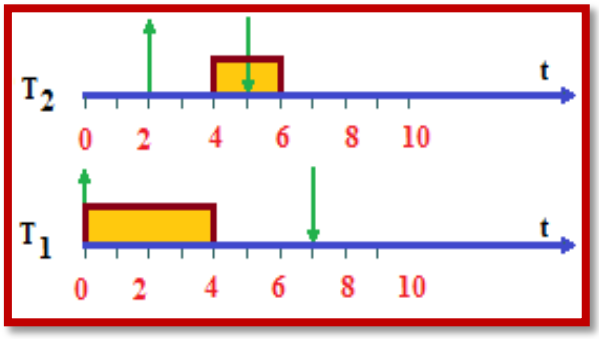


Figure II.1 – Ordonnancement O_1

Ordonnancement **O_2** : OK (comme sur la figure II.2).

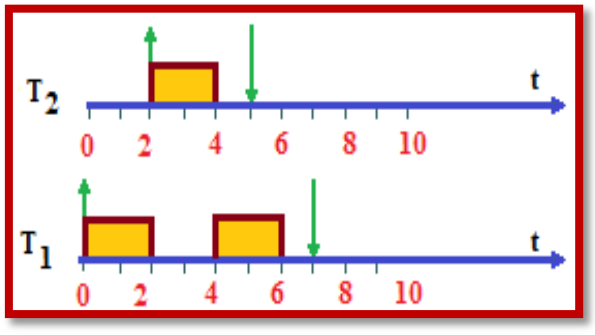


Figure II.2 – Ordonnancement O_2

e - Définition d'une tâche

Une tâche est une séquence d'instructions qui, en l'absence d'interruptions, sont exécutées sur un processeur jusqu'à son exécution finale. Une tâche sera généralement et concrètement représentée par un processus ou un thread (comme sur la figure II.3).

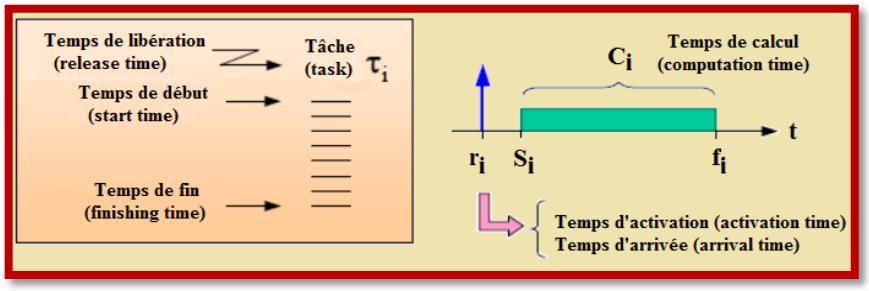


Figure II.3 – Description d'une tâche

e.1 - Tâches temps réel soumises à des contraintes de temps, plus ou moins strictes :

- Instant de démarrage,
- Instant de fin,
- Et absolus ou relatifs à d'autres tâches.

Le but de l'ordonnancement est de valider à priori la possibilité d'exécuter l'application tout en respectant ces contraintes, de permettre le respect de ces contraintes, lorsque l'exécution se produit dans un mode courant et permettre de borner les effets d'incidents ou de surcharges.

II.2 - Caractéristiques des tâches

Le tableau suivant énumère les différentes caractéristiques d'une tâche en termes de variables ou de formules. La figure II.4 illustre le diagramme d'exécution d'une tâche en fonction du temps.

Variables ou formules	Signification
r	Date de réveil - moment du déclenchement de la première requête d'exécution.
C	Durée d'exécution maximale (capacité).
D	Délai critique : délai maximum acceptable pour son exécution.

P	Période (si tâche périodique).
d = r+D	Échéance (si tâche à contraintes strictes).
$r_k = r_0 + k*P$	Tâche périodique : si D = P, tâche à échéance sur requête.

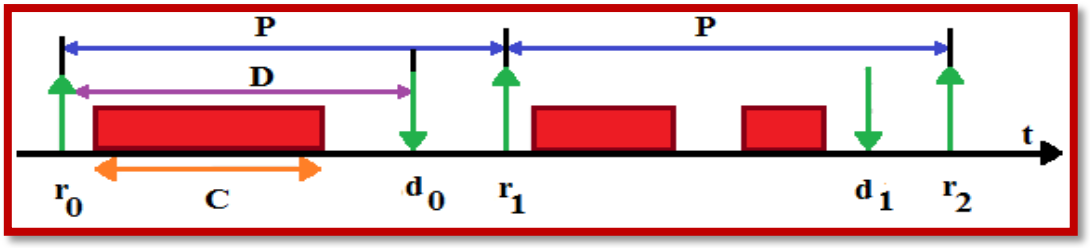


Figure II.4 – Diagramme d'exécution d'une tâche

On distingue des paramètres statiques et des paramètres dynamiques comme sur les deux tableaux suivants. La figure II.5 illustre le diagramme d'exécution d'une tâche avec l'intégration du délai critique résiduel.

a - Paramètres statiques

Formules	Signification
U = C/P	Facteur d'utilisation du processeur.
CH = C/D	Facteur de charge du processeur.

b - Paramètres dynamiques avec intégration du délai critique résiduel.

Variables ou formules	Signification
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 ($0 \leq D(t) \leq D$).

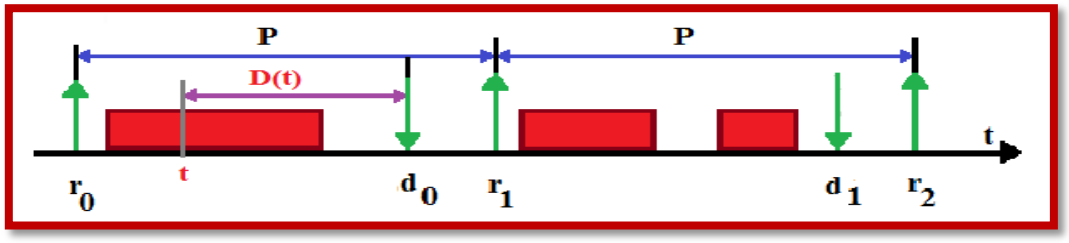


Figure II.5 – Diagramme d'exécution d'une tâche avec le délai critique

c - Paramètres dynamiques avec intégration de la durée d'exécution résiduelle (voir le tableau suivant et la figure II.6).

Variables ou formules	Signification
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 ($0 \leq C(t) \leq C$).

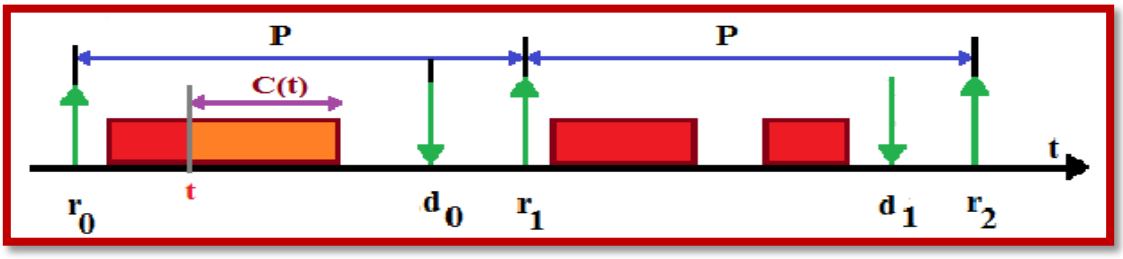


Figure II.6 – Diagramme d'exécution d'une tâche avec la durée d'exécution résiduelle

d - Paramètres dynamiques avec intégration de la laxité d'une tâche (voir le tableau suivant et la figure II.7).

Variables ou formules	Signification
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 : retard maximum pour son début d'exécution **s** (si elle est seule).

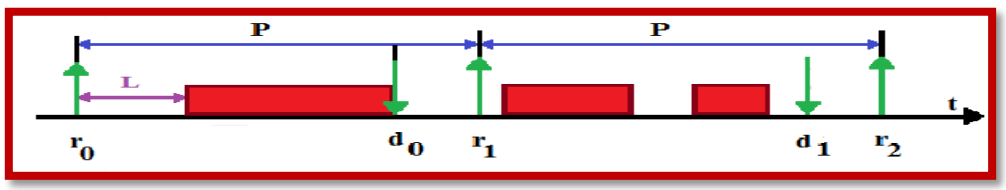


Figure II.7 - Diagramme d'exécution d'une tâche avec la laxité de la tâche

e - Paramètres dynamiques avec intégration de la laxité nominale résiduelle

(voir le tableau suivant et la figure II.8).

Variables ou formules	Signification
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 : retard maximum pour son début d'exécution s (si elle est seule).
L(t) = D(t) - C(t)	Laxité nominale résiduelle - retard maximum pour reprendre l'exécution.

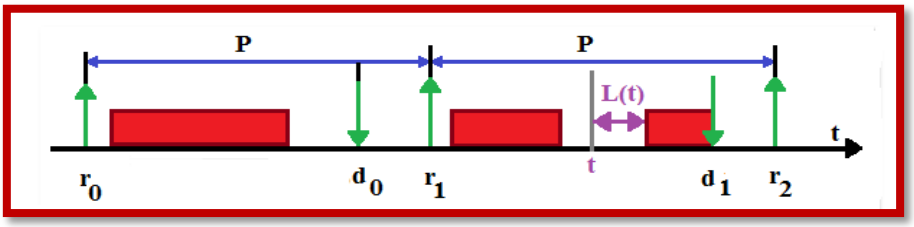


Figure II.8 - Diagramme d'exécution d'une tâche avec la laxité nominale résiduelle

f - Paramètres dynamiques avec intégration du temps de réponse (voir le tableau suivant).

Variables ou formules	Signification
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 : retard maximum pour son début d'exécution s (si elle est seule).
L(t) = D(t) - C(t)	Laxité nominale résiduelle - retard maximum pour reprendre l'exécution.
TR = e - r	Temps de réponse de la tâche.

g – Différentes caractéristiques d'une tâche

- Préemptibles ou non.

- Dépendance ou indépendance :
  - Ordre partiel prédéterminé ou induit,
  - Partage de ressources.
- Priorité externe :
  - Ordonnancement hors ligne déterminé à la conception.
- Gigue (jitter en anglais) maximale :
  - Variation entre la requête et le début de l'exécution.
- Urgence ↔ échéance.
- Importance.

II.3 - Etats d'une tâche

La figure II.9 illustre les différents états d'une tâche.

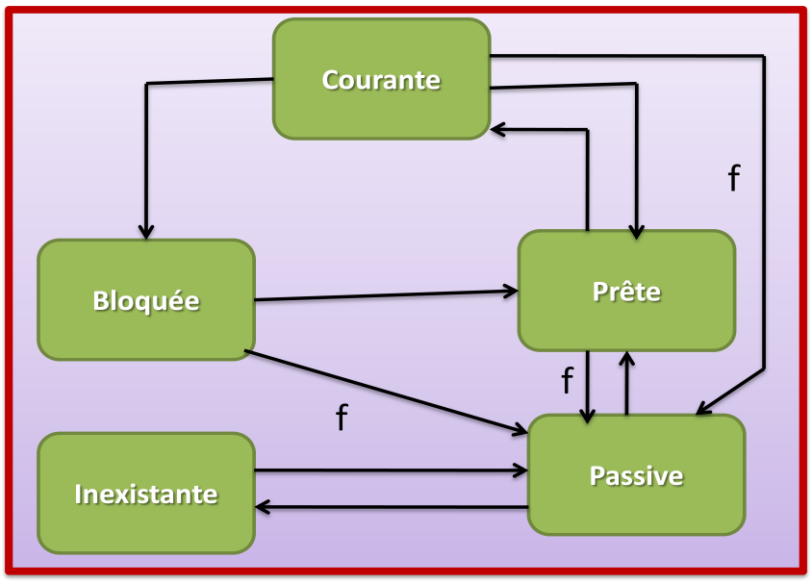


Figure II.9 – Les différents états d'une tâche

f : abandon de la requête pour cause de faute temporelle

a - Définitions

Configuration : ensemble de n tâches mises en jeu par l'application

- Facteur d'utilisation du processeur doit toujours être inférieur ou égal à un :

$$U = \sum_{i=1}^{n} \frac{Ci}{Pi}$$

- Facteur de charge :

$$CH = \sum_{i=1}^{n} \frac{Ci}{Di}$$

Intervalle d'étude : intervalle de temps minimum pour prouver l'ordonnançabilité d'une configuration :

- Le PPCM 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 $LC_i(t) = D_i - \Sigma C_i$ (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.4 - Buts de l'ordonnancement

Piloter l'application avec deux objectifs majeurs :

- En fonctionnement nominal : respect des contraintes temporelles.
- En fonctionnement anormal (par exemple pannes matérielles) : atténuer les effets des surcharges et maintenir un état cohérent et sécuritaire.

Ordonnancer = planifier l'exécution des requêtes de façon à respecter les contraintes de temps :

- De toutes les requêtes en fonctionnement nominal.
- D'au moins les requêtes les plus importantes (c'est-à-dire celles nécessaires à la sécurité du procédé) en fonctionnement anormal.

II.5 - Typologie des algorithmes

En ligne ou hors ligne :

- Choix dynamique ou prédéfini à la conception.

Préemptif ou non préemptif :

- Une tâche peut perdre le processeur (au profit d'une tâche plus prioritaire) ou non.
- Algorithme préemptif toutes ↔ les tâches préemptibles.

Stratégie du meilleur effort ou inclémence :

- En TR mou, meilleur effort = faire au mieux avec les processeurs disponibles.

- En TR dur, obligation des respecter les contraintes temporelles : inclémence aux fautes temporelles.

Centralisé ou reparti.

II.7 - Ordonnancement des tâches indépendantes

II.7.1 - Introduction

Tâches indépendantes :

- Pas de partage de ressources.
- Pas de contraintes de précédence.

Cas des algorithmes en ligne :

- Dynamique.
- Sur la base d'une priorité définie :
  - Soit de manière empirique,
  - Soit à partir d'un paramètre temporel de la tâche.
- Priorité constante ou variable avec le temps.
- À priorité identique, on évite la préemption.
- Test d'acceptabilité hors ligne si tous les paramètres sont connus.
- Sinon test de garantie au réveil des tâches.

II.7.2 - Rate Monotonic Analysis (RMA)

Algorithme à priorité constante

- Basé sur la période (tâches à échéance sur requête) : La tâche de plus petite période est la plus prioritaire.
- Test d'acceptabilité (condition suffisante).

$$\sum_{i=1}^{n} \frac{Ci}{Pi} \leq n(n^{1/n} - 1)$$

- Quand n est très grand : n $(2^{1/n} - 1)$ ~ ln 2 = 0,69.
- Dans la pratique, on peut rencontrer des ordonnancements valides qui vont jusqu'à 88%.
- On peut montrer 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 : La figure II.10 illustre un exemple d'application de l'algorithme Rate Monotonic Analysis (RMA).

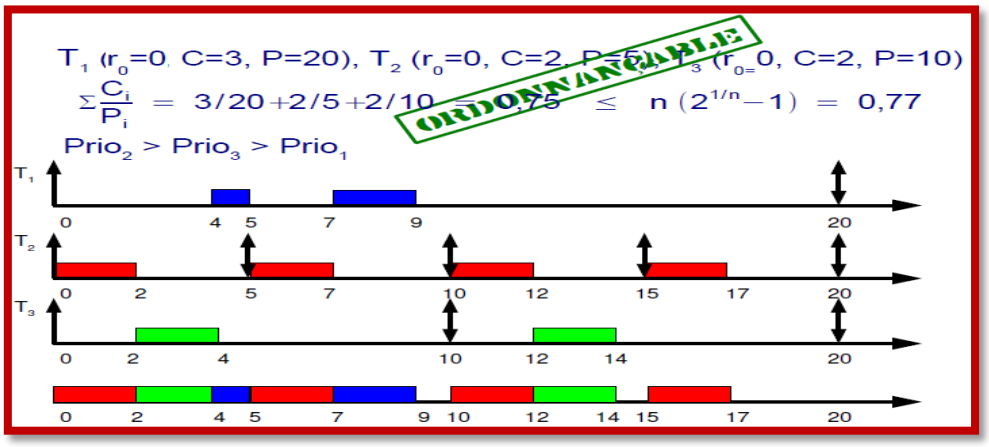


Figure II.10 – Exemple d'application de Rate Monotonic Analysis (RMA)

II.7.3 - Deadline Monotonic Analysis (DMA)

Algorithme à priorité constante

- Basé sur le délai critique : La tâche de plus petit délai critique est la plus prioritaire.
- Test d'acceptabilité (condition suffisante).

$$\sum_{i=1}^{n} \frac{Ci}{Pi} \leq n(n^{1/n} - 1)$$

- Équivalent à **RMA** dans le cas des tâches à échéance sur requête, meilleur dans les autres cas.

Exemple : La figure II.11 illustre un exemple d'application de l'algorithme Deadline Monotonic Analysis (DMA).

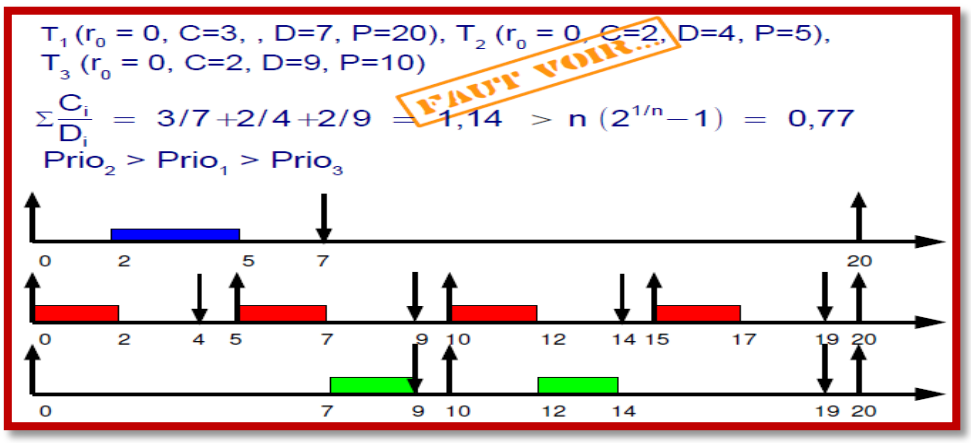


Figure II.11 – Exemple d'application de Deadline Monotonic Analysis (DMA)

II.7.4 - Earliest Deadline First (EDF)

Algorithme à priorité variable ou dynamique

- Basé sur l'échéance : à chaque instant (voir à chaque réveil de tâche), la priorité maximale est donnée à la tâche dont l'échéance est la plus proche.
- Test d'acceptabilité :
  - Condition nécessaire :

$$\sum_{i=1}^{n} \frac{Ci}{Pi} \leq 1$$

  - Condition suffisante :

$$\sum_{i=1}^{n} \frac{Ci}{Di} \leq 1$$

- On peut montrer qu'EDF est un algorithme optimal pour les algorithmes à priorité dynamique.

Exemple : La figure II.12 illustre un exemple d'application de l'algorithme Earliest Deadline First (EDF).

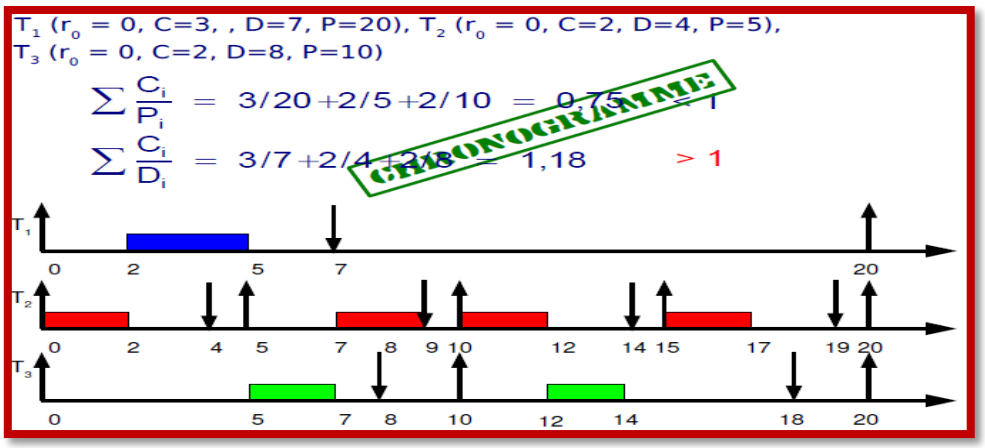


Figure II.12 – Exemple d'application de Earliest Deadline First (EDF)

II.7.5 - Least Laxity First (LLF)

Algorithme à priorité variable

- Aussi appelé Least Slack Time (LST).
- Basé sur la laxité résiduelle : La priorité maximale est donnée à la tâche qui a la plus petite laxité résiduelle L(t) = D(t) – C(t).
- Équivalent à EDF si on ne calcule la laxité qu'au réveil des tâches.

- Optimum à trouver entre la granularité du calcul et le nombre de changements de contexte provoqués.

Exemple : La figure II.13 illustre un exemple d'application de l'algorithme Least Laxity First (LLF).

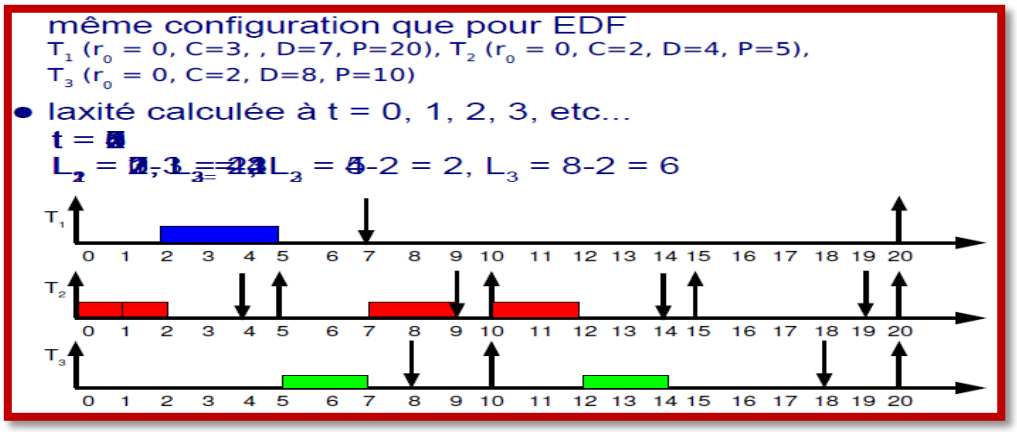


Figure II.13 – Exemple d'application de Least Laxity First (LLF)

II.8 - Tâches apériodiques à contraintes relatives

II.8.1 - Introduction

- Tâches prises en compte dans une configuration comprenant déjà des tâches périodiques critiques : Les tâches apériodiques peuvent être critiques (contrainte stricte) ou non (contrainte relative).
- À priori, on ne connait pas l'instant d'arrivée de la requête de réveil de la tâche apériodique :
  - Besoin d'une hypothèse sur le taux d'arrivée maximal des requêtes,
  - Les tâches apériodiques dont le taux d'arrivée est borné supérieurement sont dites sporadiques,
  - La charge due aux tâches apériodiques est bornée.
- On peut garantir la réponse :
  - Si le taux d'arrivée n'est pas borné : pas de garantie possible,
  - Mais garantie éventuelle au cas par cas.
- Test d'acceptabilité pour les tâches apériodiques critiques (fermes).

a - Buts à atteindre

- Si contraintes relatives : minimiser le temps de réponse,

- Si contraintes strictes : maximiser le nombre de tâches acceptées en respectant leurs contraintes.
- Deux grandes catégories de traitement :
  - Traitement en arrière-plan,
  - Traitement par serveurs,
  - Beaucoup d'études d'algorithmes.

II.8.2 - Traitement d'arrière-plan (comme sur la figure II.14)

- Tâches apériodiques ordonnancées quand le processeur est oisif : Les tâches périodiques restent les plus prioritaires (tâches critiques).
- Ordonnancement relatif des tâches apériodiques en mode FIFO.
- Préemption des tâches apériodiques par les tâches périodiques.
- Traitement le plus simple, mais le moins performant.

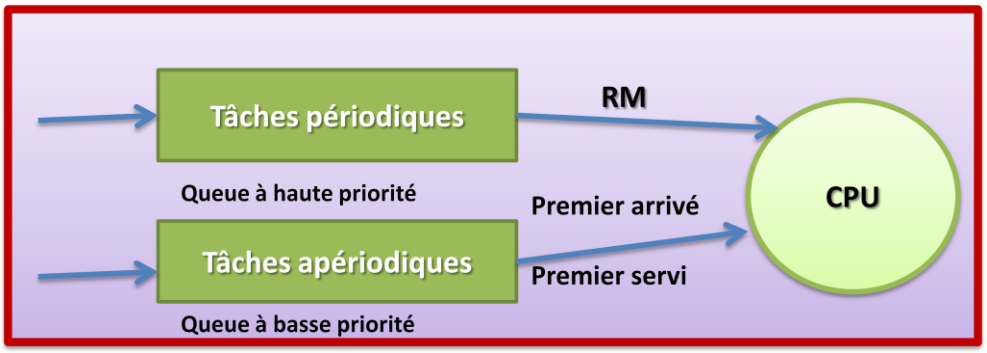


Figure II.14 – Traitement d'arrière plan

Exemple d'ordonnancement RMA des tâches périodiques (comme sur la figure II.15).

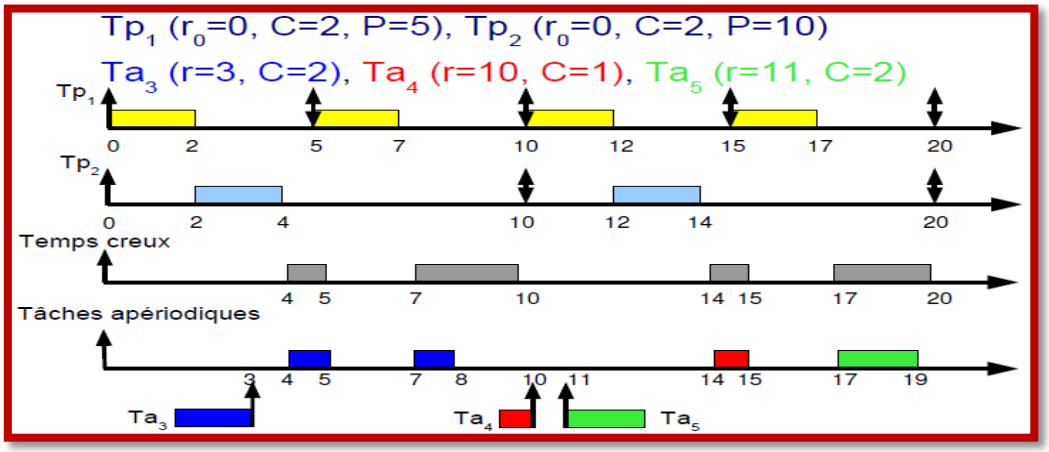


Figure II.15 – Ordonnancement RMA Des tâches périodiques

II.8.3 - Traitement par serveur

Un serveur est une tâche périodique créée spécialement pour prendre en compte les tâches apériodiques.

- Serveur caractérisé par :
  - Sa période,
  - Son temps d'exécution : capacité du serveur,
  - Serveur généralement ordonnance 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.

II.8.4 - Traitement par serveur par scrutation

a - Serveurs par scrutation (polling)

- À chaque activation, traitement des tâches en suspens jusqu'à épuisement de la 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).

Exemple de serveur par scrutation (ordonnancement RMA) comme sur les figures II.16 et II.17.

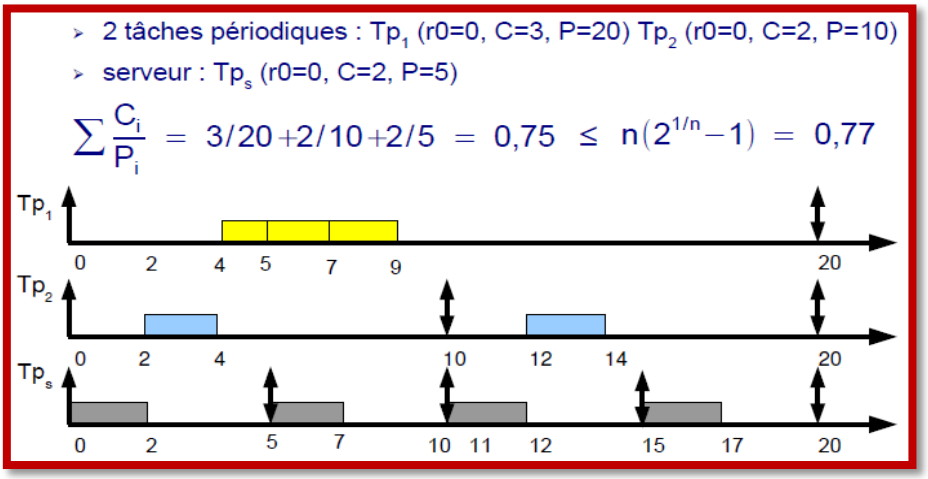


Figure II.16 – Serveurs par scrutation (polling)

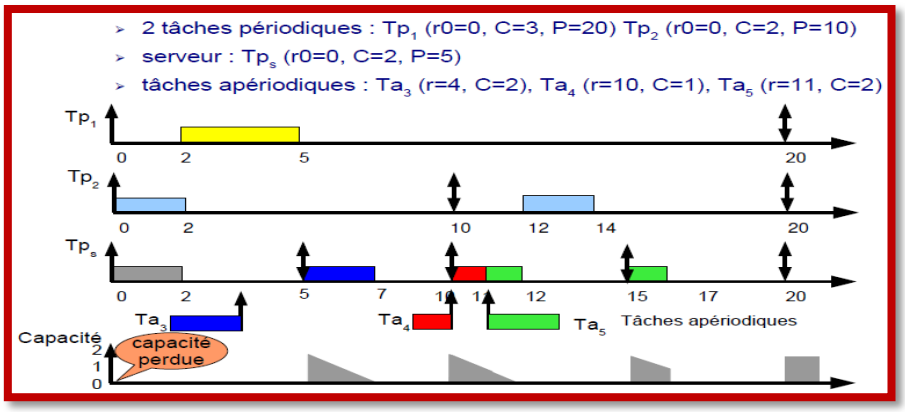


Figure II.17 – Serveurs par scrutation suite

b - Limitations du serveur par scrutation

- Perte de la capacité si aucune tâche apériodique en attente,
- Si occurrence d'une tâche apériodique alors que le serveur est suspendu, il faut attendre la requête suivante.

II.8.5 - Traitement par serveur sporadique

a - Serveur sporadique

- Améliore le temps de réponse des tâches apériodiques sans diminuer le taux d'utilisation du processeur pour les tâches périodiques,
- Comme le serveur ajournable mais ne retrouve pas sa capacité à période fixe,
- Le serveur sporadique peut être considéré comme une tâche périodique normale du point de vue des critères d'ordonnancement.

b - Calcul de la récupération de capacité

- Le serveur est dit « actif » quand la priorité de la tâche courante P_{exe} est supérieure ou égale à celle du serveur P_s
- Le serveur est dit inactif si $P_{exe} < P_s$
- RT : date de la récupération
  - Calculée dés que le serveur devient actif (t_A),
  - Égale à $t_A + T_s$.
- RA : montant de la récupération à effectuer à RT
  - Calculé à l'instant t_I ou le serveur devient inactif ou que la capacité est épuisée,
  - Égal à la capacité consommée pendant l'intervalle $[t_A, t_I]$.

Exemple de serveur sporadique à haute priorité comme sur la figure II.18.

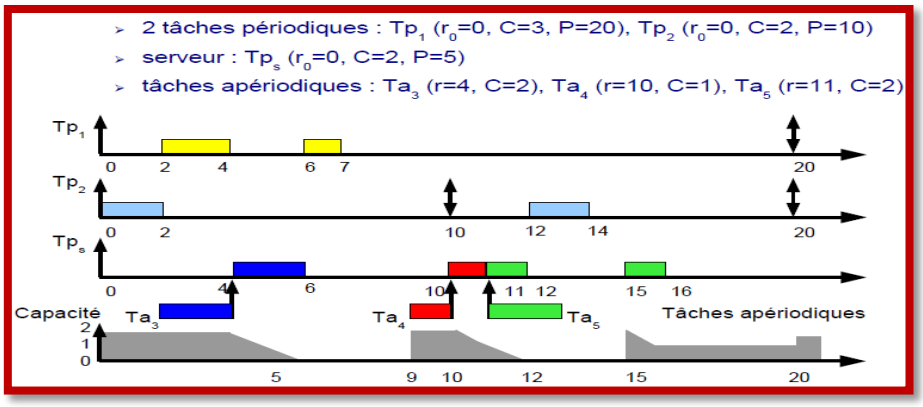


Figure II.18 – Serveurs sporadique à haute priorité

C - Exercice

- Deux tâches périodiques + 1 serveur sporadique
  - $Tp_1 : t_1 = 0, C_1 = 1, T_1 = 5$
  - $Tp_2 : t_2 = 0, C_2 = 4, T_2 = 15$
  - SS : $C_s = 5, T_s = 10$
- Tâches apériodiques :
  - $Ta_1 : t_{a1} = 4, C_{a1} = 2$
  - $Ta_2 : t_{a2} = 8, C_{a2} = 2$
- Ordonnancement ?

II.9 - Tâches apériodiques à contraintes strictes

II.9.1 – Introduction

Les algorithmes précédents restent possibles. Mais exécution d'un test d'acceptabilité avant l'acceptation ou le rejet de la tâche : suffisamment de temps CPU disponible (processeur oisif ou capacité serveur disponible) avant l'échéance de la tâche apériodique.

II.9.2 - Principe de l'ordonnancement

- Ordonnancer les tâches en EDF**.**
- À chaque nouvelle tâche apériodique, faire tourner une routine de garantie pour vérifier que toutes les contraintes temporelles seront respectées :
  - Si oui, accepter la tâche,
  - Si non, refuser la tâche.
- Deux politiques d'acceptation dynamique :
  - Acceptation dans les temps creux,
  - Ordonnancement conjoint.

- Favorise les tâches périodiques.

II.9.3 - Acceptation dans les temps creux

- Ordonnancement EDF des tâches périodiques.
- Les tâches apériodiques acceptées sont ordonnancées dans les temps creux des tâches périodiques (~ méthode d'arrière-plan) selon l'algorithme EDF.
- Routine de garantie (au réveil d'une tâche apériodique) :
  - Teste l'existence d'un temps creux suffisant entre le réveil et l'échéance de la tâche apériodique,
  - Vérifie que l'acceptation de la nouvelle tâche ne remet pas en cause le respect des contraintes temporelles des autres tâches apériodiques déjà acceptées et non encore terminées,
  - Si OK, la tâche est ajoutée à la liste des tâches apériodiques.

Exemple d'acceptation dans les temps creux comme sur les figures II.19 et II.20.

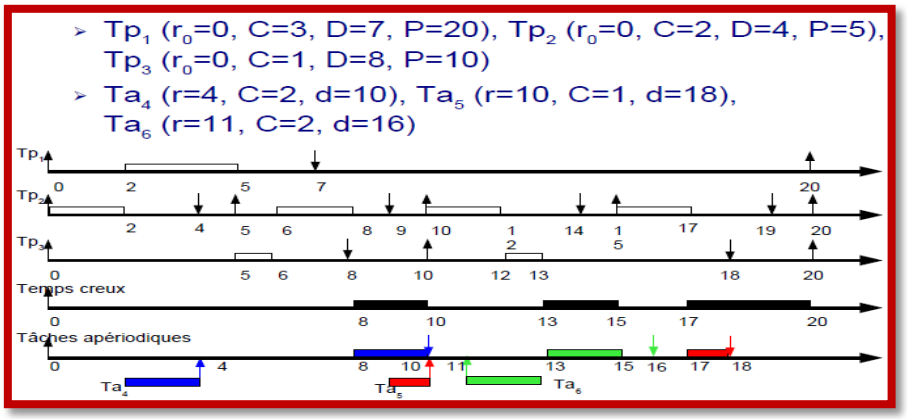


Figure II.19 – Acceptation dans les temps creux

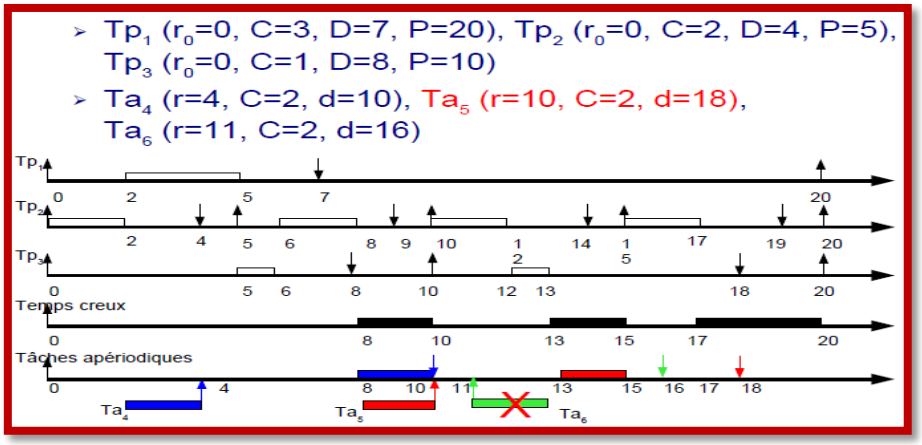


Figure II.20 – Acceptation dans les temps creux (suite)

II.9.4 - Ordonnancement conjoint

La séquence des tâches périodiques n'est plus immuable. À l'arrivée de chaque nouvelle tâche apériodique, construction d'une nouvelle séquence EDF : si la construction est possible : acceptation de la tâche, Sinon rejet.

Exemple d'ordonnancement conjoint comme sur les figures II.21 et II.22.

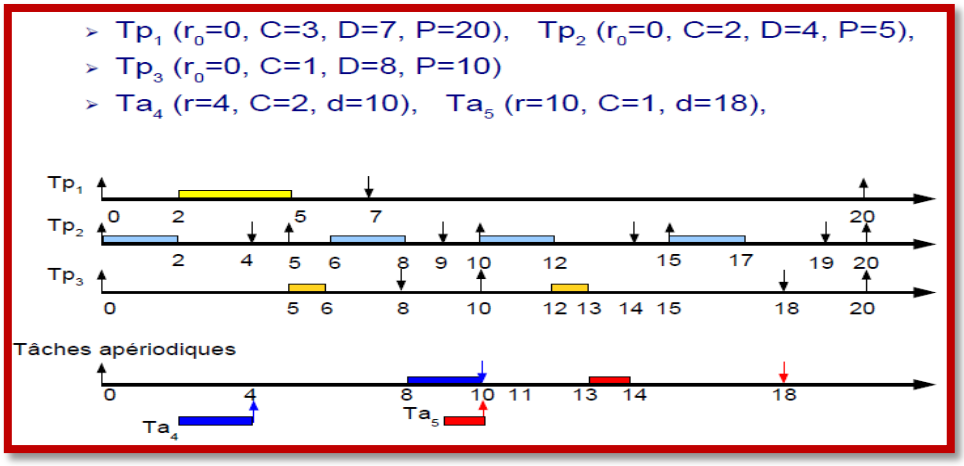


Figure II.21 – Ordonnancement conjoint)

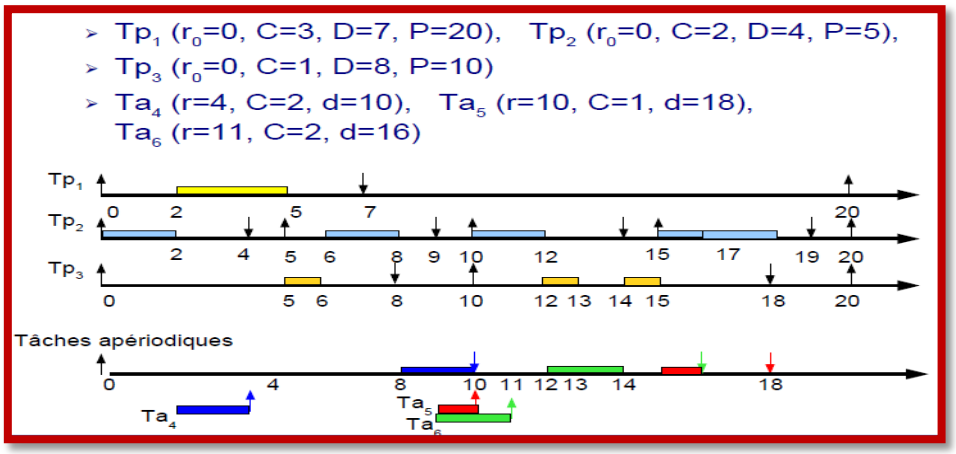


Figure II.22 – Ordonnancement conjoint)

II.10 - Ordonnancement de tâches dépendantes

II.10.1 - Présentation des contraintes

a - Contraintes de précédence (comme sur la figure II.23).

- Sur l'ordre d'exécution des tâches les unes par rapport aux autres.
- Généralement décrites par un graphe oriente G :
  - $T_a < T_b$ indique que la tâche T_a est un prédécesseur de T_b,
  - T_a ➔ T_b indique que la tâche T_a est un prédécesseur immédiat de T_b.

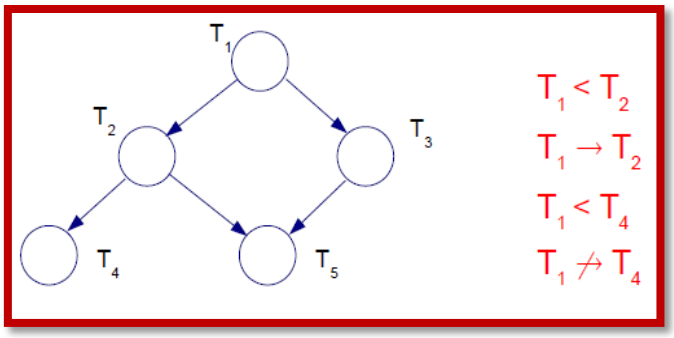


Figure II.23 – Exemple de contraintes de précédence

Exemple : reconnaissance de formes (comme sur la figure II.24).

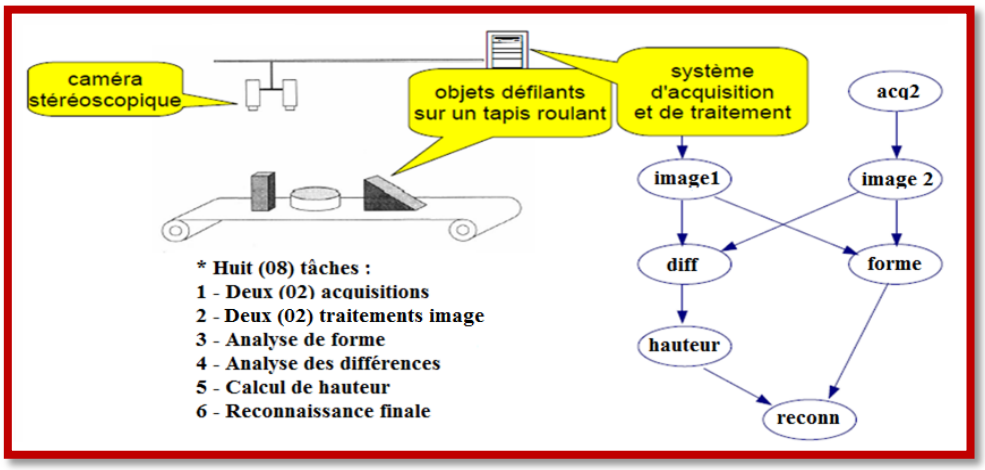


Figure II.24 – Exemple de reconnaissance de formes

b - Partage de ressources

Les contraintes sur les accès aux ressources :

Ressource	**Toute structure logicielle qui peut être utilisée par une tâche pour son exécution**
Privée	Si dédiée à la tâche.
Partagée	Si elle peut être utilisée par plusieurs tâches.
Exclusive	Si une seule tâche à la fois peut l'utiliser.

- Par l'accès séquentiel par plusieurs tâches à une ressource exclusive nécessitent un mécanisme de synchronisationEt les codes implémentant les opérations en exclusion mutuelle sont des sections critiques.
- Phénomènes de blocage et d'inversion de priorité.
- Le délai imposé par T_2 à T_1 est normal comme sur la figure II.25.

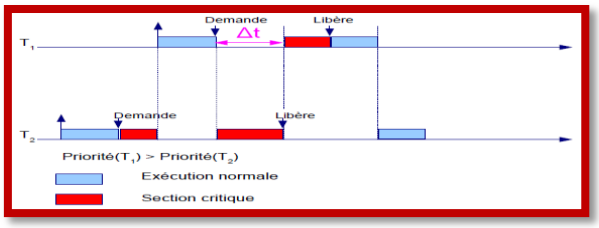


Figure II.25 – Partage de ressources

- Le délai supplémentaire apporté à T_1 par T_3 est dû au phénomène d'inversion de priorité comme sur la figure II.26.

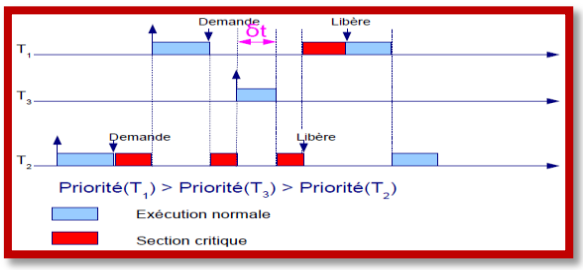


Figure II.26 – Partage de ressources (suite)

c - Théorème de Graham

"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".

II.10.2 - Quelques anomalies d'ordonnancement

Exemple : (comme sur la figure II.27)

- Neuf tâches de priorités décroissantes : (Prio(J_i) > Prio(J_j)) ⇔ i > j
- Contraintes de précédence et temps d'exécution :

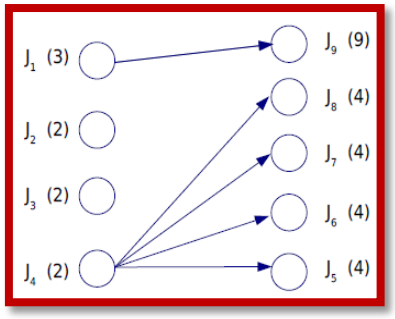


Figure II.27 – Exemple d'anomalies d'ordonnancement

a - Ordonnancement optimal sur un système à trois processeurs (comme sur la figure II.28)

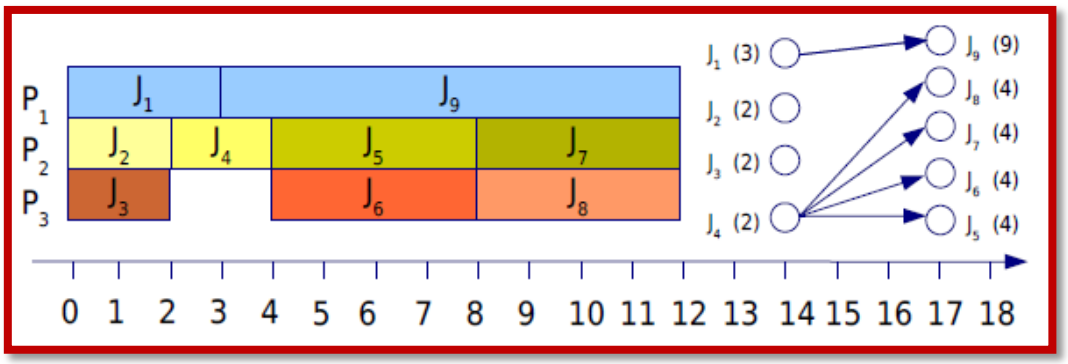


Figure II.28 - Ordonnancement optimale sur un système à trois processeurs

b - Ordonnancement optimal sur un système à quatre processeurs (comme sur la figure II.29)

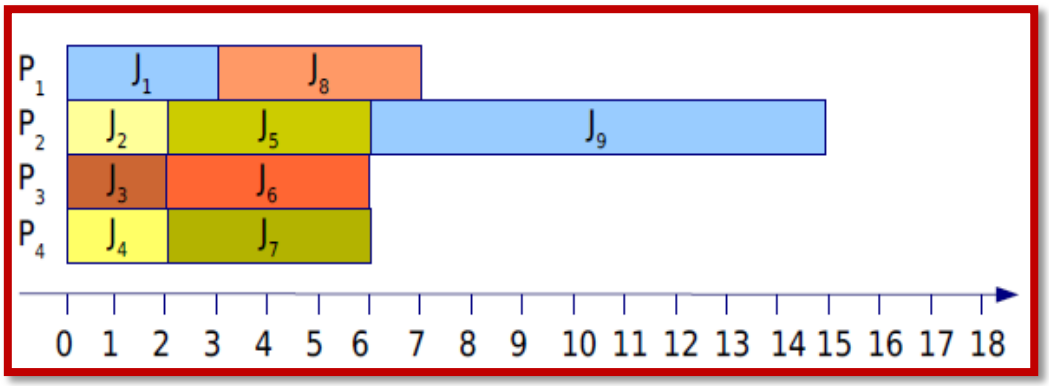


Figure II.29 - Ordonnancement optimale sur un système à quatre processeurs

c - Ordonnancement optimal sur un système à trois processeurs en réduisant d'une unité le temps de calcul des tâches (comme sur la figure II.30)

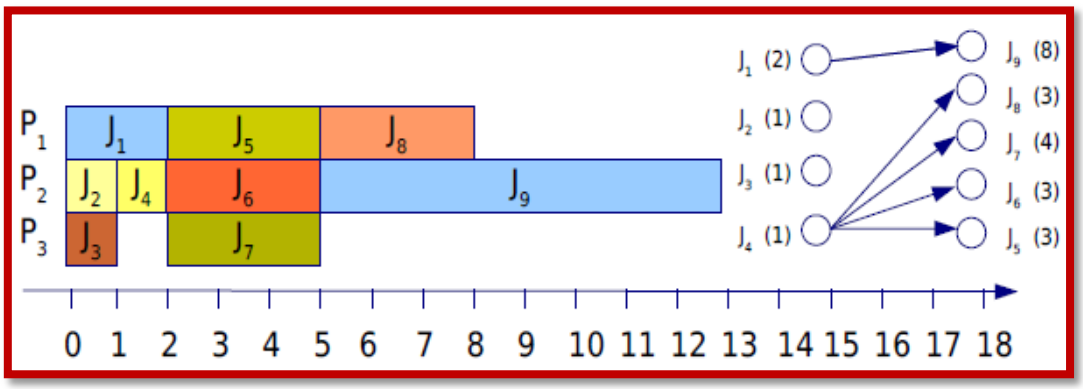


Figure II.30 - Ordonnancement optimale sur un système à trois processeurs en réduisant d'une unité le temps de calcul des tâches.

d - Ordonnancement optimal sur un système à trois processeurs en supprimant quelques relations de précédence (comme sur la figure II.31)

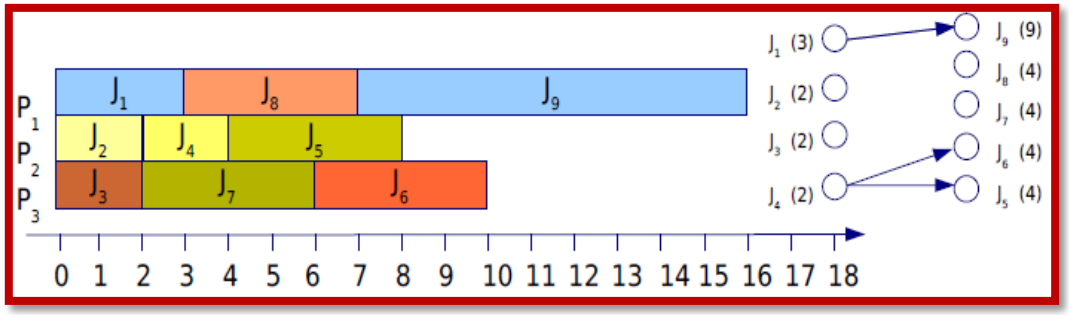


Figure II.31 - Ordonnancement optimale sur un système à trois processeurs en supprimant quelques relations de précédence.

e - Exemple d'anomalie sur des contraintes de ressources

Cinq tâches allouées statiquement à deux processeurs J_2 et J_4 partagent une ressource exclusive comme sur la figure II.32.

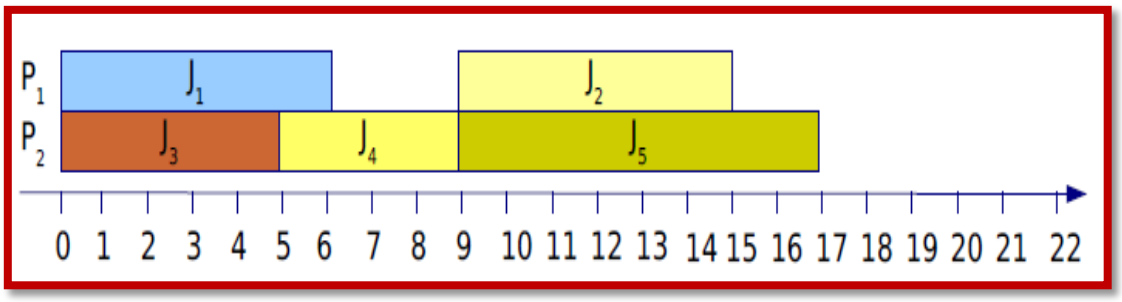


Figure II.32 - Cinq tâches allouées à deux processus partage une ressource exclusive

Si on réduit le temps d'exécution de J_1 comme sur la figure II.33.

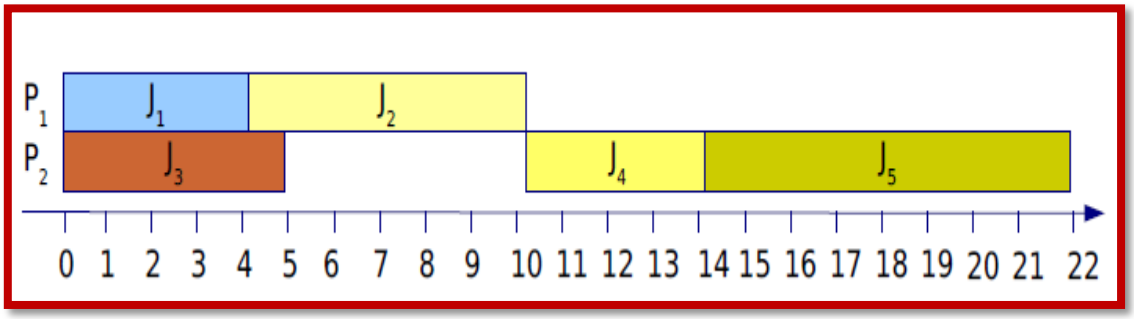


Figure II.33 - En réduisant le temps d'exécution de J_1

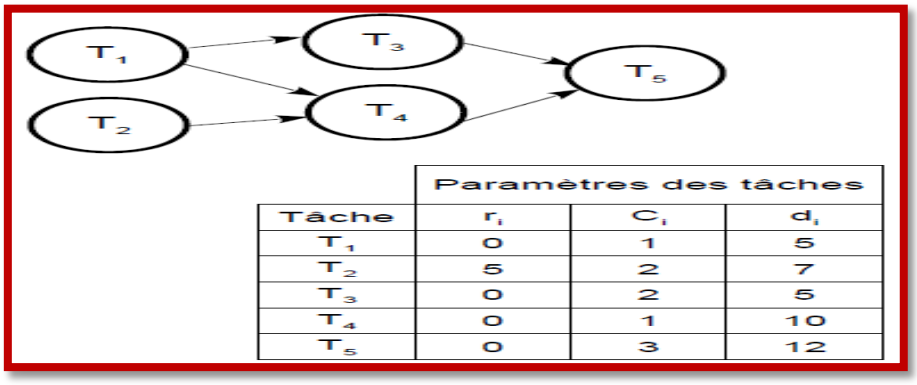
II.10.3 - Algorithmes d'ordonnancement

a - Contraintes de précédence

- Ici, seulement précédence simple, Si T_i est périodique de période P_i, alors T_j l'est aussi et $P_j = P_i$.
- Principe de l'établissement de l'ordonnancement :
    - Transformer l'ensemble des tâches dépendantes en un ensemble de tâches indépendantes que l'on ordonnancera par un algorithme classique.
    - Par des modifications des paramètres des tâches :
        - Si T_i ➔ T_j alors la règle de transformation doit respecter
        - r_i £ r_j
        - $Prio_i > Prio_j$
    - Validation de l'ordonnançabilité selon des critères utilisés pour des tâches indépendantes.
- Contraintes de précédence et Rate Monotonic.
- La transformation s'opère hors ligne sur la date de réveil et sur les délais critiques :
    - $r^*_i = Max \{r_i, r^*_j\}$ pour tous les j tels que T_j ➔T_i
    - Si T_i ➔T_j alors $Prio_i > Prio_j$ dans le respect de la règle d'affectation des priorités par RMA.
- Contraintes de précédence et Deadline Monotonic.
- La transformation s'opère hors ligne sur la date de réveil et sur les délais critiques :
    - $r^*_i = Max \{r_i, r^*_j\}$ pour tous les j tels que T_j ➔T_i
    - $D^*_i = Max \{D_i, D^*_j\}$ pour tous les j tels que T_j ➔T_i
    - Si T_i ➔ T_j alors $Prio_i > Prio_j$ dans le respect de la règle d'affectation des priorités par DMA.
- Contraintes de précédence et EDF :
    - Modification des échéances de façon à ce qu'une tâche ait toujours un d_i inférieur à celui de ses successeurs (algorithme de Chetto & al.),
    - Une tâche ne doit être activable que si tous ses prédécesseurs ont terminé leur exécution,
    - Modification de la date de réveil et de l'échéance :
    - $r^*_i = Max \{r_i, Max\{r^*_j + C_j\}\}$ pour tous les j tels que T_j ➔ T_i,
    - $d^*_i = Min \{d_i, Min\{d^*_j - C_j\}\}$ pour tous les j tels que T_i ➔T_j,
    - On itère sur les prédécesseurs et successeurs immédiats,

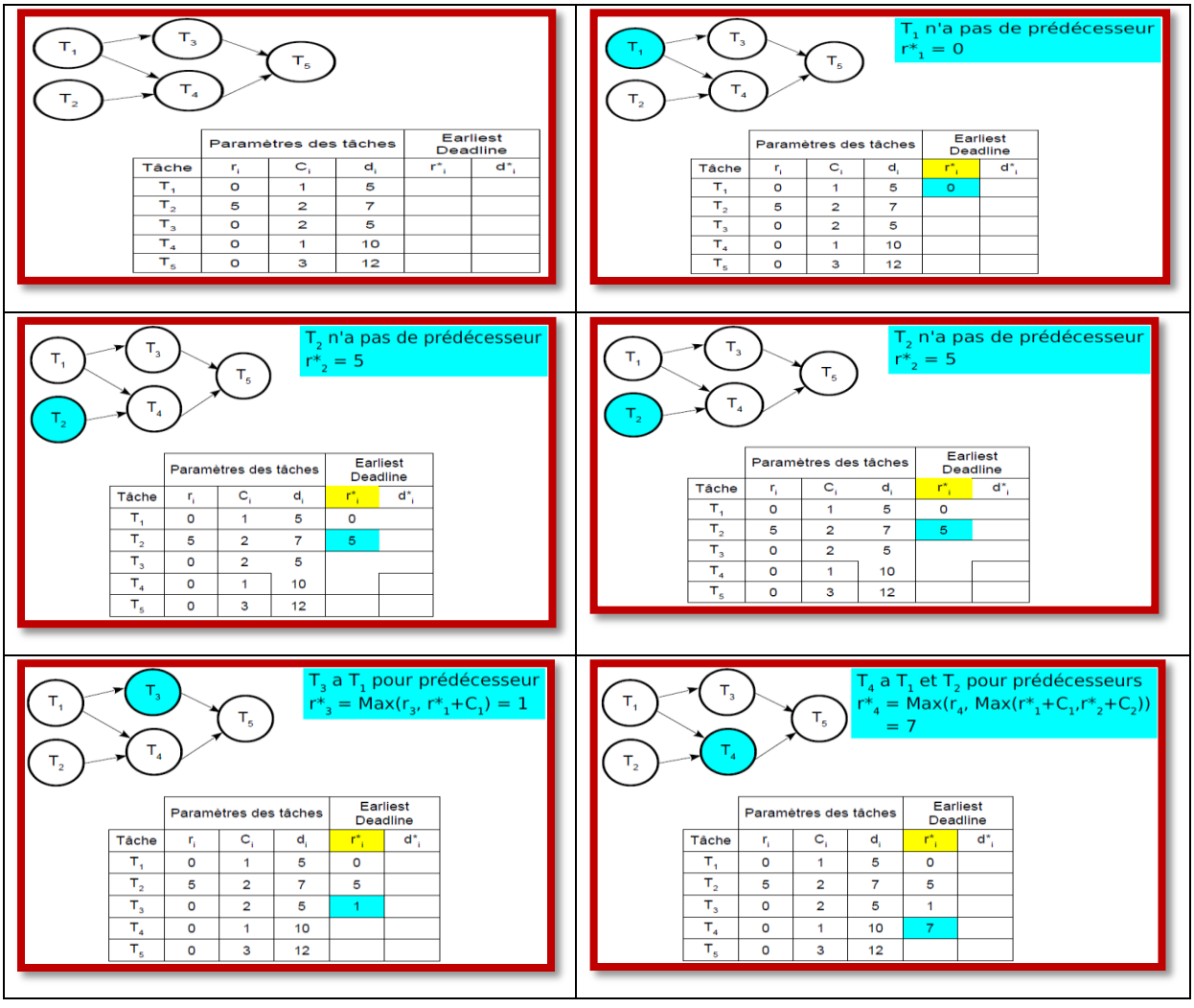
- On commence les calculs par les tâches qui n'ont pas de prédécesseurs pour le calcul des r et par les tâches qui n'ont pas de successeur pour le calcul des d.

b - Exemple d'exécution de contraintes de précédence (voir les figures II.34 et II.35)



	Paramètres des tâches		
Tâche	r_i	C_i	d_i
T_1	0	1	5
T_2	5	2	7
T_3	0	2	5
T_4	0	1	10
T_5	0	3	12

Figure II.34 - Exemple d'exécution de contraintes de précédence



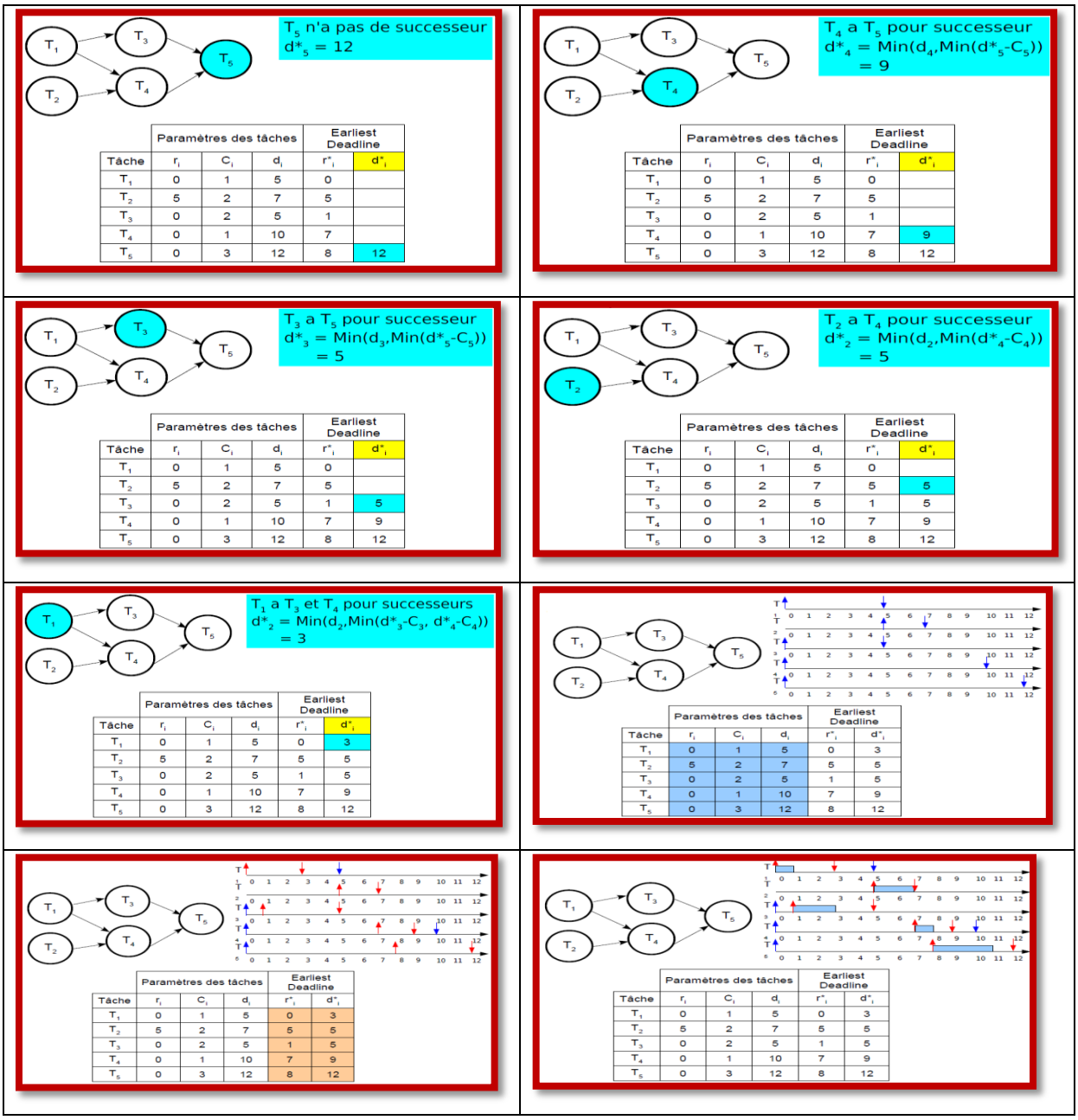


Figure II.35 - Séquence d'exécution de contraintes de précédence de l'exemple

II.10.4 - Partage de ressources critiques

Une ressource critique ne peut être utilisée simultanément par plusieurs tâches ou être réquisitionnée par une autre tâche, d'où la notion de section critique, qui est une séquence d'instructions pendant lesquelles on utilise une ressource critique sans problème dans le cas d'un ordonnancement non préemptif, mais c'est rarement le cas dans un environnement temps réel ➔ évaluation du temps de réponse très difficile, sinon impossible.

a - Inversion de priorité

Phénomène dû à la présence simultanée de priorités fixes et de ressources à accès exclusif dans un environnement préemptif :

- Exemple de quatre tâches de priorités décroissantes (comme sur la figure II.38).

$$\mathbf{prio(T_1) > prio(T_2) > prio(T_3) > prio(T_4).}$$

- Si pas de partage de ressource commune :

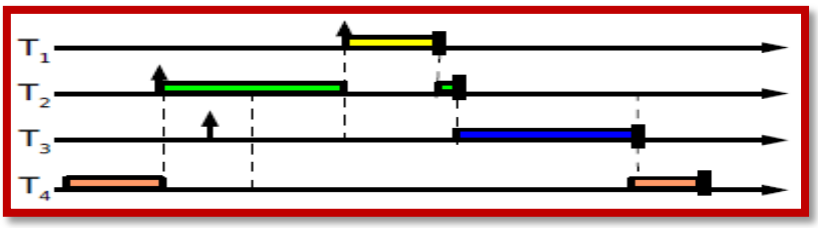


Figure II.38 – Exemple de partage de ressource commune entre quatre tâches

Phénomène dû à la présence simultanée de priorités fixes et de ressources à accès exclusif dans un environnement préemptif :

- Exemple de quatre (tâches de priorités décroissantes (comme sur la figure II.39).

$$\mathbf{prio(T_1) > prio(T_2) > prio(T_3) > prio(T_4)}$$

- Si partage d'une ressource commune,
- Le retard de $\mathbf{T_2}$ dû à $\mathbf{T_3}$ est une "inversion de priorité", non prévue.

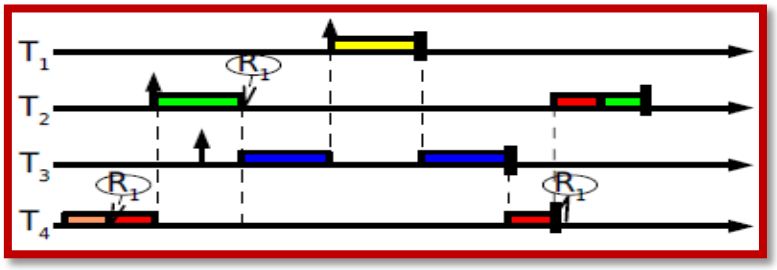


Figure II.39 – Exemple de partage de ressource avec inversion de priorité

II.10.5 - Héritage de priorité

a - Principe

Il faut attribuer à la tâche qui possède la ressource la priorité de la tâche de plus haute priorité qui attend la ressource.

b - Hypothèses de travail

- n tâches périodiques T_1, T_2, ... , T_n (période P_i, capacité C_i) à échéance sur requête,
- Partageant m ressources R_1, R_2, ... , R_m,
- Chaque ressource R_j est gardée par un sémaphore binaire S_j limitant une section critique $z_{i,j}$ (pour T_i),
- P_i : priorité nominale, p_i : priorité active
- $P_1 > P_2 > ... > P_n$,
- $z_{i,j} \subset z_{i,k}$ ou $z_{i,k} \subset z_{i,j}$ ou $z_{i,j} \cap z_{i,k} = \emptyset$ $\forall$ $z_{i,j}$ et $z_{i,k}$.

c - Définition du protocole d'héritage de priorité

Les tâches sont ordonnancées suivant leur priorité active, quand une tâche T_i cherche à entrer dans une section critique $z_{i,j}$ et que la ressource R_j est déjà possédée par une tâche T_k, de priorité plus faible, alors T_i se bloque (sinon T_i entre dans $z_{i,j}$), La tâche T_i, bloquée, transmet sa priorité à T_k qui peut alors redémarrer et terminer l'exécution de la section critique $z_{k,j}$, quand T_k sort de la section critique, il relâche le sémaphore, S_j et la tâche de plus haute priorité est réveillée. Si aucune tâche n'est encore bloquée par T_k, alors p_k est ramenée à P_k, sinon on donne à p_k la plus haute des priorités des tâches en attente.

d - Exemple de sans héritage de priorité (voir les figures II.40 et II.41)

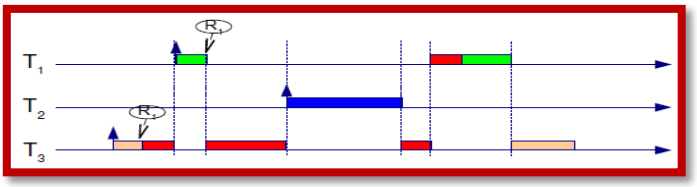


Figure II.40 – Exécution sans héritage de priorité

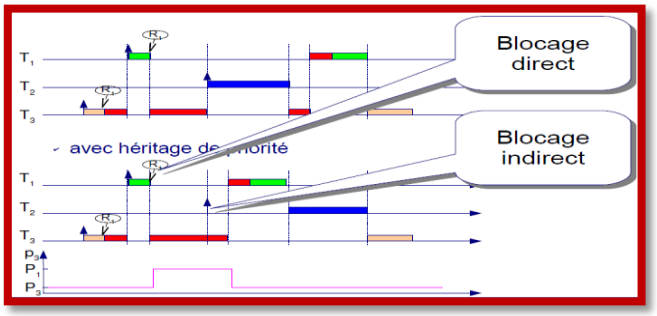


Figure II.41 – Exécution sans héritage de priorité avec blocage direct

e - Propriétés du protocole d'héritage des priorités

- 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,
- Condition nécessaire d'ordonnançabilité par un algorithme Rate Monotonic :

$$\forall i, 1 \leq i \leq n, \left[\sum_{k=1}^{i} \frac{Ck}{Tk}\right] + \frac{Bi}{Ti} \leq i\left(2^{1/i} - 1\right)$$

f - Problèmes rémanents

Blocages en chaîne comme sur la figure II.42.

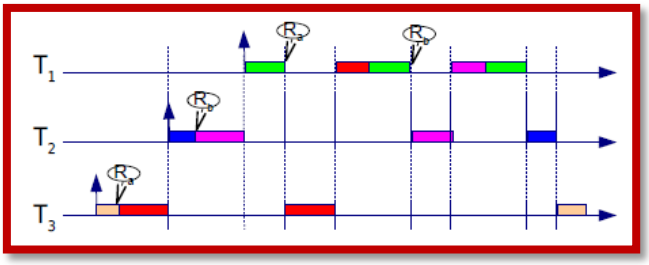


Figure II.42 – Blocages en chaîne

Étreinte fatale (deadlock), si deux tâches utilisent deux ressources imbriquées, mais dans l'ordre inverse comme sur la figure II.43.

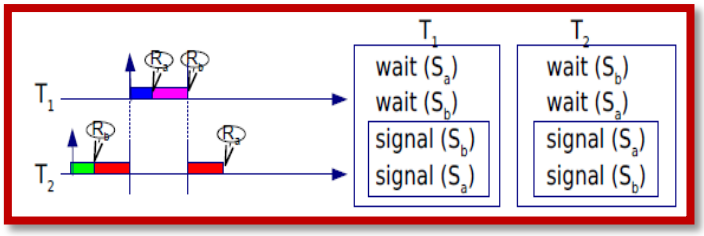


Figure II.43 – Etreinte fatale avec ressources imbriquées dans l'ordre inverse

II.10.6 - Priorité plafonnée

Elle a été introduite à la fin des années quatre vingt (80) pour résoudre le problème d'inversion de priorité tout en prévenant l'occurrence de deadlocks et de blocages en chaîne comme sur la figure II.43. Dans l'amélioration du protocole d'héritage de priorité, une tâche ne peut pas entrer dans une section critique s'il y a un sémaphore acquis qui pourrait la bloquer.

a - Principe

On attribue à chaque sémaphore une priorité plafond égale à la plus haute priorité des tâches qui pourraient l'acquérir. Une tâche ne pourra entrer dans la section critique que si elle possède une priorité supérieure à toutes celles des priorités plafond des sémaphores acquis par les autres tâches.

b - Définition du protocole

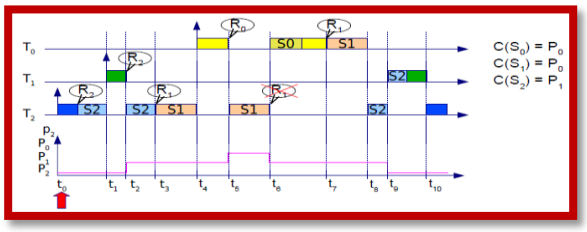
On attribue à chaque sémaphore S_k une priorité plafond $C(S_k)$ égale à la plus haute priorité des tâches susceptibles de l'acquérir.

- Soit T_i la tâche prête de plus haute priorité : l'accès au processeur est donné à T_i.
- Soit S* le sémaphore dont la priorité plafond C(S*) est la plus grande parmi tous les sémaphores déjà acquis par des tâches autres que T_i.
- Pour entrer dans une section critique gardée par un sémaphore S_k, T_i doit avoir une priorité supérieure à C(S*). Si $P_i \leq C(S^*)$, l'accès à S_k est dénié et on dit que T_i est bloquée sur S* par la tâche qui possède S*.
- Quand une tâche T_i est bloquée sur l'acquisition d'un sémaphore, elle transmet sa priorité à la tâche T_k qui possède le sémaphore (héritage par T_k de la priorité de T_i).
- Quand T_k sort de la section critique, elle libère le sémaphore et la tâche de plus haute priorité bloquée sur le sémaphore est réveillée. La priorité active de T_k est modifiée :
  - Si T_k ne bloque aucune autre tâche, elle revient à sa priorité nominale,
  - Sinon, T_k prend la priorité la plus élevée des tâches qu'elle bloque.
- L'héritage de priorité est transitif : si T_3 bloque T_2 et T_2 bloque T_1, alors J_3 récupère la priorité de T_1 via T_2.

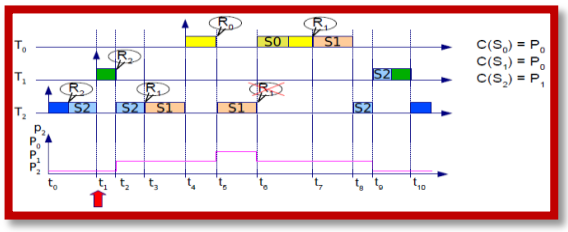
c - Exemple voir la solution dans les figures de la figure II.44

- Trois tâches T_0, T_1 et T_2 de priorités décroissantes avec $P_0 > P_1 > P_2$
  - T_0 accède séquentiellement à deux 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 S_2 et aussi, de manière imbriquée, à S_1.
- Les priorités plafond sont :
  - $C(S_0) = P_0$

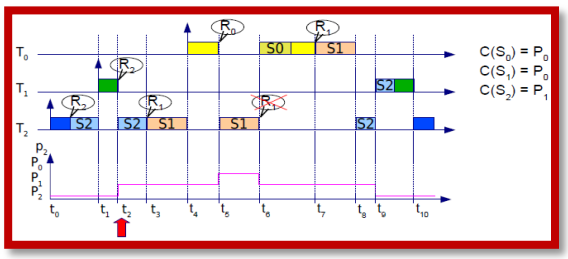
- $C(S_1) = P_0$
- $C(S_2) = P_1$



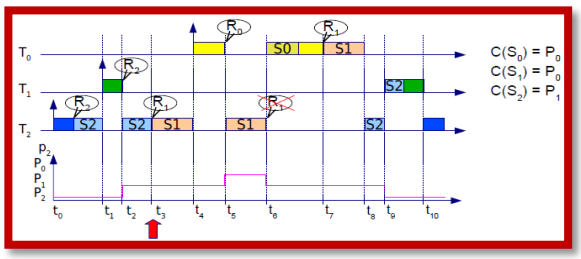
À t_0, T_2 est activée et démarre. Le sémaphore S_2 est demandé et obtenu (il n'y a pas d'autre ressource en cours d'utilisation).



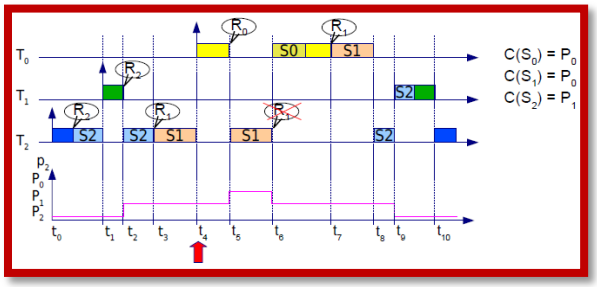
À t_1, T_1 est activée et préempte T_2.



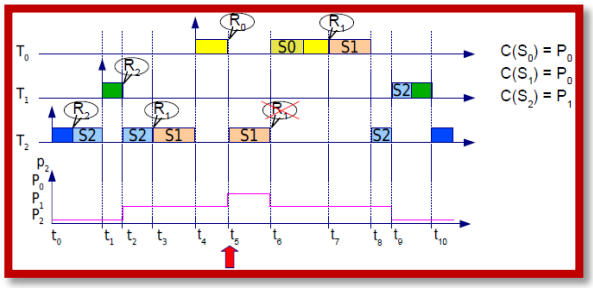
À t_2, T_1 demande S_2 et est bloquée par le protocole car la priorité de T_1 n'est pas supérieure à la priorité plafond $C(S_2)=P_1$ (S_2 est le seul sémaphore acquis à t_2) T_2 hérite de la priorité de T_1 et redémarre



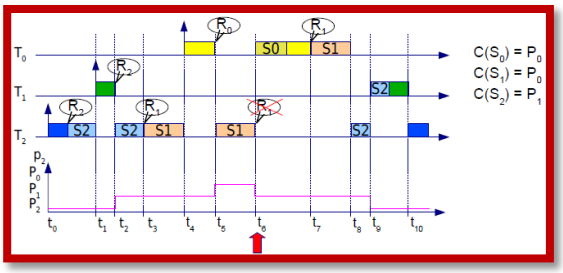
À t_3, T_2 demande et obtient le sémaphore S_1, car aucune autre tâche ne détient de ressource.



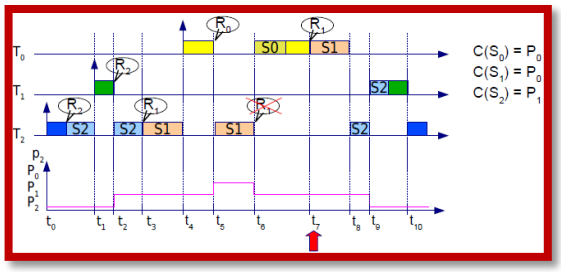
À t_4, T_0 est réveillée et préempte T_2.



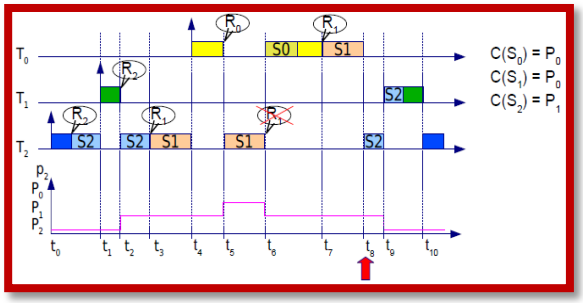
À t_5, T_0 demande le sémaphore S_0 qui n'est détenu par aucune autre tâche. Le protocole bloque néanmoins T_0 parce que sa priorité n'est pas supérieure à la plus grande priorité plafond des sémaphores déjà détenus (S_2 et S_1) : P_0, T_2 hérite de la priorité de T_0 et peut redémarrer.



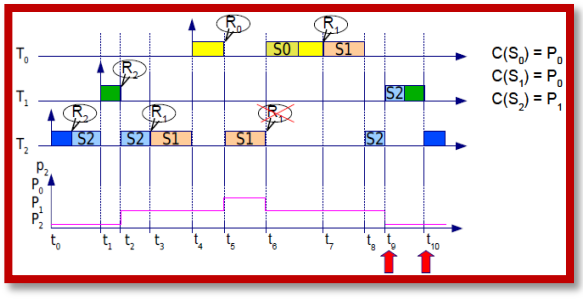
À t_6, T_2 relâche S_1. T_0 est réveillée et peut alors préempter T_2 et acquérir S_0 puisque la priorité plafond de S_2 (seul sémaphore encore pris) est égale à P_1. La priorité p_2 de T_2 est ramenée à P_1, plus haute priorité des tâches bloquées par T_2.



À 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.



À t_8, T_0 se termine. T_2 peut reprendre son exécution, toujours avec la même priorité P_1.



À 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.

À t_{10}, T_1 se termine, T_2 peut redémarrer et se terminer.

Figure II.44 – Exemple d'exécution de l'exercice

On a le même critère d'ordonnançabilité par RMA que dans le cas du protocole d'héritage de priorité :

$$\forall i, 1 \le i \le n, \left[\sum_{k=1}^{i} \frac{Ck}{Tk}\right] + \frac{Bi}{Ti} \le i\left(2^{1/i} - 1\right)$$

Mais le calcul du temps de blocage maximum de chaque tâche est plus simple. On peut 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 :

Exercice : tâches avec demandes de ressources imbriquées.

d - La mission Pathfinder (en 1997) comme sur la figure II.45

- Sonde sur Mars, arrivée le 4 juillet 1997
- Robot mobile Sojourner chargé de différentes tâches
  1. Photos
  2. Relevés météo
  3. Prélèvements
- Poids : 11.5kg
- Vitesse : 24m/h
- Puissance totale : 30W
- Liaison **UHF** avec la sonde Pathfinder
- Bug dans la gestion des ressources critiques

Figure II.45 – La mission Pathfinder

➔ Perte de données importantes

d.1 - Spécification fonctionnelle (la mission **Pathfinder** en 1997) comme sur la figure II.46.

Le système de gestion de la sonde communique avec l'extérieur par :

- La carte radio pour les liaisons avec la terre,
- La carte de liaison avec la camera.
- Et l'interface avec le bus 1553 pour les autres capteurs/actionneurs

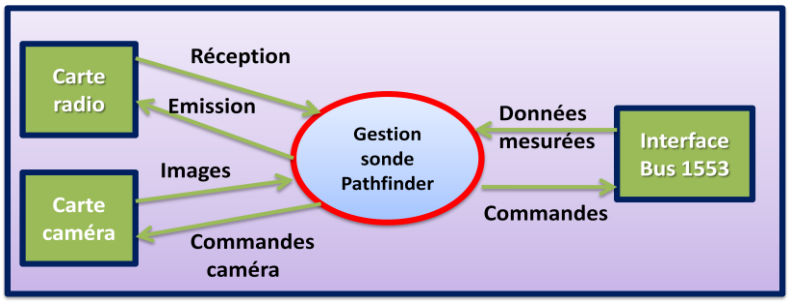


Figure II.46 – Spécification fonctionnelle de Pathfinder

d.2 - Architecture logicielle (la mission Pathfinder en 1997)

- Multitâche gérée par le noyau Vxworks (Wind River)
- 25 tâches

- Périodiques (ex. : gestion du bus 1553)
- Apériodiques (ex. : analyse des erreurs)
- Communication et synchronisation par des files de messages

- Suivant les phases de la mission (vol interplanétaire, atterrissages, exploration par le robot), toutes les taches ne sont pas utiles

d.3 - Architecture logicielle (la mission Pathfinder en 1997)

Liste des tâches et priorités relatives :

Priorité	Tâche	Nature de la tâche
Maximum	ORDO_BUS	Ordonnanceur du bus 1553
↑	DISTRIBUTION_DONNEES	Distribution des données du bus 1553
↑	TÂCHE_PILOTAGE	Pilotage de l'application (robot)
↑	TÂCHE_RADIO	Gestion des communications radio
↑	TÂCHE_CAMÉRA	Gestion de la caméra
↑	TÂCHE_MESURES	Mesures
Minimum	TÂCHE_METEO	Gestion des données météo

d.4 - Caractéristiques des tâches (la mission Pathfinder en 1997)

Tâches	Priorités	Paramètres (ms)		Paramètres réduits		Temps d'utilisation ressources
		C_i	P_i	C_i	P_i	
ORDO_BUS	7	25	125	1	5	
DISTRIBUTION_DONNEES	6	25	125	1	5	1
TÂCHE_PILOTAGE	5	25	250	1	10	1
TÂCHE_RADIO	4	25	250	1	10	
TÂCHE_CAMÉRA	3	25	250	1	10	
TÂCHE_MESURES	2	50	5000	2	200	2
TÂCHE_METEO	1	[50,75]	5000	[2,3]	200	[2,3]

II.11 - Ordonnancement en situations de surcharge

II.11.2 - Notion de surcharge

Quand la charge du processeur est telle qu'il est impossible de respecter toutes les échéances. Les origines possibles multiples :

- Matériel, par exemple une transmission défectueuse qui fait qu'un signal arrive en retard (réseau surcharge),
- Contention sur une ressource critique,
- Réveil de tâches apériodiques suite à des alarmes.

Les algorithmes à priorités classiques (du type EDF ou RMA) offrent des performances souvent médiocre en cas de surcharge. Effet domino dû à l'arrivée d'une tâche supplémentaire dans un ordonnancement EDF comme sur la figure II.47.

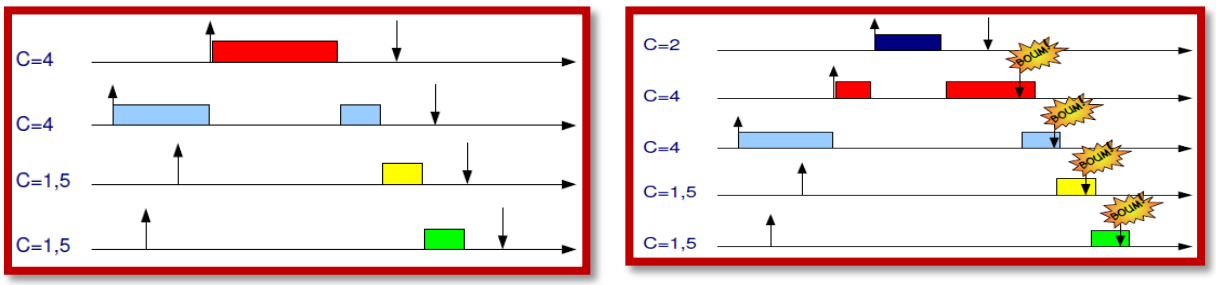


Figure II.47 – Effet Domino dû à la surcharge

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

$$U = \sum_{i=1}^{n} \frac{Ci}{Ti}$$

Le cas général est basé 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 :

$$\rho_i = \frac{Ci}{Di} \ (d_i = \ r_i + D_i)$$

Il faut calculer au réveil des tâches (date $t = r_i$)

$$\rho_i(t) = \frac{\sum_{d_k \leq d_i} C_k(t)}{(d_i - t)} \ et \ \rho = \max\left(\rho_i(t)\right)$$

Exemple de la notion de surcharge comme sur la figure II.48.

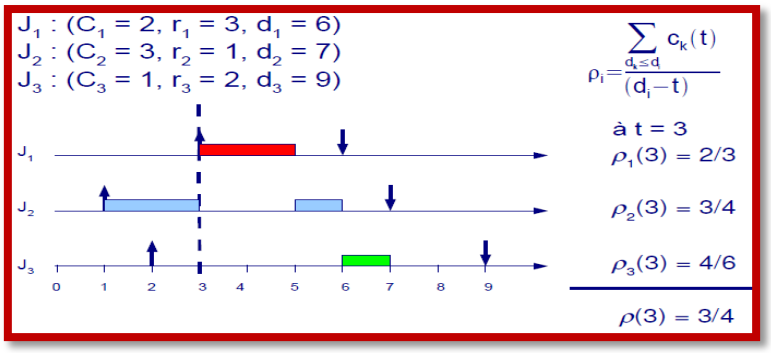


Figure II.48 – Exemple de notion de surcharge

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. 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. L'importance de l'exécution d'une tâche ne peut pas se définir seulement à l'aide de son échéance. Il est surement plus important de faire la mesure d'un capteur toutes les dix secondes que de mettre à jour l'heure sur l'écran de contrôle toutes les secondes. Cela nécessite 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, l'intérêt est de définir une fonction d'utilité qui décrit l'importance de la tâche en fonction du temps comme sur la figure II.49.

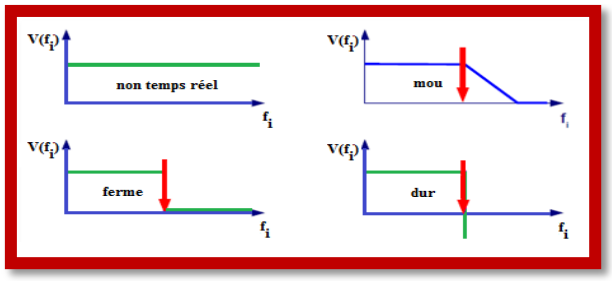


Figure II.49 – Fonction d'utilité

On fait une évaluation de la performance d'un algorithme par :

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

où $V(f_i)$ est l'utilité de la tâche à sa terminaison.

Remarque

Si une tâche temps réel dur dépasse son échéance, alors Γ_A vaut -∞. Il faut réserver les ressources (entre autres la CPU) pour garantir les tâches dures. Pour un ensemble de n tâches

J_i (C_i, D_i, V_i), ou V_i est l'utilité de la tâche quand elle se termine avant son échéance, la valeur maximale de Γ_A est en conditions de surcharge, la qualité d'un algorithme A se mesure en comparant Γ_A à Γ_{max}.

II.11.3 - Gestion des situations de surcharge

Rappel

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 et tâches quelconques.

II.11.4 - Tâches périodiques

- Pas d'arrivée dynamique de nouvelles tâches.
- Durées d'exécutions variables, non forcément que l'on peut bornées.
- Deux politiques présentées : méthode du mécanisme à échéance et méthode du calcul approche.

II.11.5 - Méthode du mécanisme à échéance

a - Principe

Chaque tâche a deux versions :

1. Version primaire assurant une bonne qualité de service mais dans un temps indéterminé,
2. Version secondaire fournissant un résultat suffisamment acceptable au bout d'un temps connu.

b - L'algorithme de tolérance aux fautes

Il doit assurer le respect de toutes les échéances, soit par le primaire soit par le secondaire :

- Si les deux marchent, on garde le primaire,
- Si le primaire ne réussit pas, le secondaire doit réussir,
- Ordonnancement = juxtaposition d'une séquence des primaires et d'une des secondaires + règle de décision pour commuter de l'une à l'autre.

Deux politiques possibles :

1. Politique de la première Chance :

Prio (Secondaires) > Prio (Primaires)

➔ Les primaires sont exécutés dans les temps creux du processeur après leur secondaire.

2. Politique de la Seconde Chance :
   - Primaires exécutés avant les secondaires,
   - Secondaires exécutés plus tard,
   - Si le primaire réussit, le secondaire n'est pas exécuté.

Pour obtenir un minimum de qualité de service, il faut un certain pourcentage de réussite des primaires et donc garder suffisamment de temps de processeur pour cela.

II.11.6 - Méthode du calcul approché

Chaque tâche est décomposée en deux parties :

1. Mandataire, fournissant un résultat approche et devant s'exécuter dans le respect de son échéance,
2. Optionnelle, affinant le résultat et exécutée seulement s'il reste assez de temps.

Évite les surcouts dus à la coordination entre plusieurs versions d'une même tâche. Même remarque que précédemment à propos de la qualité de service minimale.

II.11.7 - Tâches quelconques

Le modèle canonique des tâches insuffisant, basé sur Urgence ↔ délai critique. Un nouveau paramètre est nécessaire, d'où l'importance, codant le caractère primordial de la tâche dans l'application, et définition subjective (cahier des charges). La cause fréquente des fautes temporelles : l'occurrence d'une tâche apériodique et la possibilité de les rejeter par une routine de garantie, éventuellement vers un autre processeur moins chargé dans un environnement reparti. Mais le mécanisme est lourd,

➔ **Ordonnancement à importance**.

Trois classes de politiques pour décider l'acceptation (ou le rejet) des nouvelles tâches : meilleur effort, avec garantie et robuste.

a - Meilleur effort comme sur la figure II.50

- Pas de prédiction en cas de surcharge.
- Les nouvelles tâches sont toujours acceptées.
- Seul moyen d'action : les niveaux relatifs des priorités.

Figure II.50 – Politique de meilleur effort

b - Avec garantie

Un test d'acceptabilité est exécuté à l'arrivée de chaque nouvelle tâche, garantissant la bonne exécution de toutes les tâches et base sur les pires hypothèses. Si le test d'acceptabilité n'est pas positif, la tâche est rejetée. Éventuellement, recyclage des tâches rejetées pour profiter d'une exécution plus rapide que prévue d'une tâche acceptée.

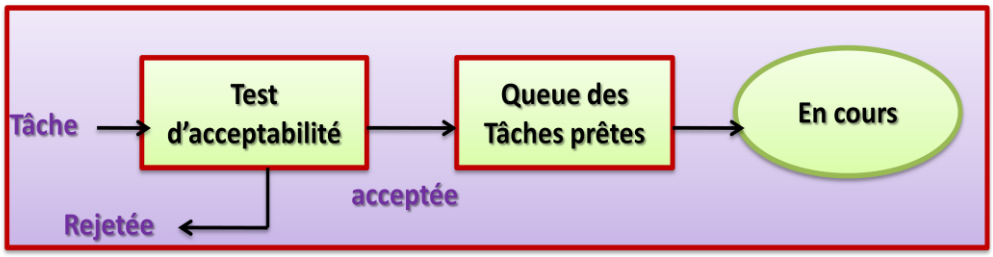


Figure II.51 – Politique avec garantie

c – Robuste comme sur la figure II.52

- Séparation des contraintes temporelles et de l'importance des tâches.
- Deux politiques : une pour l'acceptation de nouvelles tâches, une pour le rejet de tâches.
- Quand une nouvelle tâche arrive, on exécute un test d'acceptabilité :
  - Si le test passe, la tâche est mise dans la queue des tâches prêtes,
  - Si le test ne passe pas, on exécute l'algorithme de rejet qui permet d'éliminer une ou plusieurs tâches de moindre importance,
  - Éventuellement, recyclage des tâches rejetées pour profiter d'une exécution plus rapide que prévue d'une tâche acceptée.
- Séparation des contraintes temporelles et de l'importance des tâches.

- Deux politiques : une pour l'acceptation de nouvelles tâches, une pour le rejet de tâches.
- Quand une nouvelle tâche arrive, on exécute un test d'acceptabilité

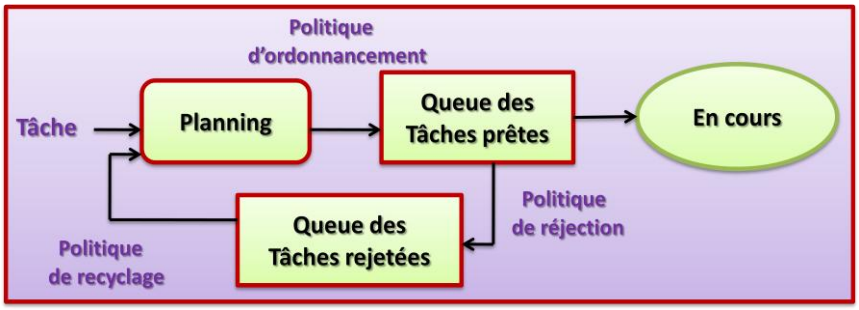


Figure II.52 – Politique de robustesse

II.11.8 - Test de garantie

Le texte de garantie est exécuté à chaque réveil d'une tâche. On vérifie si la nouvelle tâche peut être exécutée sans provoquer de faute temporelle, basé sur l'évaluation de la laxité.

$T = \{T_i (t, C_i(t), d_i)\}$ avec $d_i < d_j$ si $i < j$ → Tâches actives à t, classées par ordre décroissant d'échéance.

$$LC_i(t) = D_i - \sum Ci(t),\ d_j \leq d_i \text{ et } (D_i = d_i - r_i)$$

$$LP(t) = Min\ (LC_i(t))$$

La surcharge est détectée quand LP(t) <0 et la tâches fautives ↔ $LC_i(t) < 0$

II.11.9 - Ordonnancement par importance

Le critère d'importance utilise pour éliminer des exécutions de tâches parmi celles dont les échéances sont inférieures ou égales à celle de la tâche fautive. Plusieurs politiques possibles :

- Stabilisation de l'importance des tâches ordonnancées,
- Maximisation de la somme des importances des tâches garanties.

II.11.10 - Stabilisation de l'importance

- Assurer que ce sont les tâches les plus importantes qui font le moins de fautes temporelles.
- Amélioration du modèle des tâches :
  - Mode de fonctionnement normal,

- Modes de fonctionnement de survie, invoques quand le mode de fonctionnement normal doit être arrêté,
- Propriétés d'exécution du mode normal :
  - Ajournabilité, quand l'exécution de la tâche peut être supprimée avant qu'elle n'ait commencé,
  - Révocabilité, quand l'exécution de la tâche peut être supprimée alors qu'elle est déjà commencée,
  - Les quatre combinaisons sont possibles.

- Trois modes de survie :
  - Ajournement,
  - Révocation,
  - Faute temporelle.
- Les modes d'ajournement et de révocation contiennent les opérations à exécuter dans chaque cas :
  - Communication de résultats à d'autres tâches, exécution d'un travail palliatif, suppression de tâches liées par une précédence, etc.
- Le mode de faute temporelle est exécuté quand la tâche commet la faute, et doit être défini pour chacun des modes normaux, ajournement et révocation

Algorithme

Task T is

begin

Mode Normal

actions du mode normal(C_n,d_n,propriétés,Imp_n)

Mode Ajournement

actions du mode ajournement(C_{aj},d_{aj},exécution obligatoire)

Mode Révocation

actions du mode révocation(C_{rv},d_{rv},exécution obligatoire)

Mode Faute Temporelle

actions si Faute Temporelle mode normal(C_{nf})

actions si Faute Temporelle mode ajournement(C_{ajf})

actions si Faute Temporelle mode révocation(C_{rvf})

end

Exemple

```
Task T1 is
begin
Mode_Normal(C=10, Ajournable, Revocable, Imp=5)
        Acquerir(Capteur);
        Lire(Capteur, Temp);
        Restituer(Capteur);
        -- calculs sur Temp
        Temp := Calcul();
        -- envoi de Temp à la Tâche T2
                Send(Temp, {T2});
                old_Temp := Temp;
Mode_Révocation (C=3, Obligatoire,          Imp=5)
        -- ajournement de T1
        Restituer(Capteur);
        Ajourner(T2);
Mode_Ajournement(C=2,               Obligatoire, Imp=5)
        -- Calcul d'une valeur depuis          l'ancienne valeur
        Temp := old_Temp *                  facteur_correction
        Send(Temp, {T2});
end;
```

a - Politique de contrôle comme sur la figure II.53

La politique de contrôle est assurée par un contrôleur de charge en amont de l'ordonnanceur.

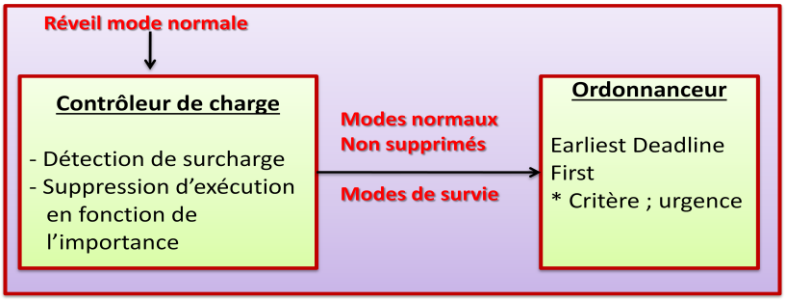


Figure II.53 – Politique de contrôle

b - Détection de la surcharge par évaluation de la laxité

À chaque réveil, calcul de la laxité courante de chaque mode actif

$$\mathbf{M = \{Mx_i\ (r, C_{xi}(r), d_{xi}, Imp_{xi}\}\ avec\ x = n, a_j\ ou\ r_v, i = 1, m}$$

et $d_{xi} < d_{xj}$ (i<j) l'ensemble des modes actifs à la date r triés par échéance croissante

laxité du mode M_{xi} :

- Si toutes les laxités sont positives, pas de surcharge ➔ ordonnancement de la tâche par EDF
- M n'est pas ordonnançable s'il existe un mode M_{xf} pour lequel la laxité $L_{xf}(r)$ est négative. Le mode M_{xf} est fautif et la surcharge est $|L_{xf}(r)|$.

II.11.11- Optimisation de la somme des importances

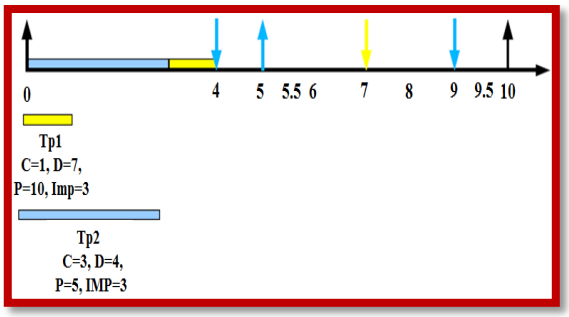
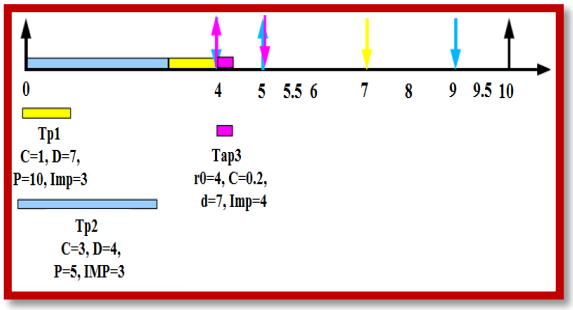
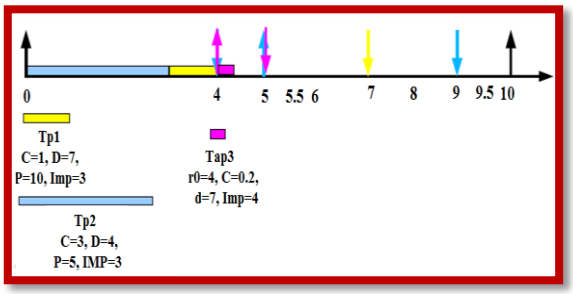
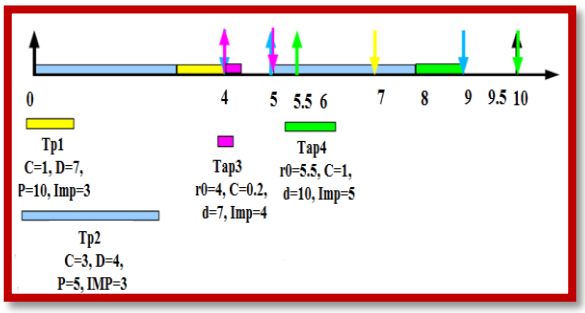
Algorithme RED (Robust Earliest Deadline) [Buttazzo]

- Gestion des tâches apériodiques dans un environnement temps réel ferme.
- Introduit la notion de tolérance sur l'échéance, c'est-à-dire le retard après l'échéance pendant lequel le résultat est encore acceptable.
- Tâches caractérisées par :
  - Le temps d'exécution dans le pire des cas C_i
  - L'échéance normale D_i (échéance primaire)
  - La tolérance sur l'échéance M_i (échéance secondaire)
  - La valeur V_i

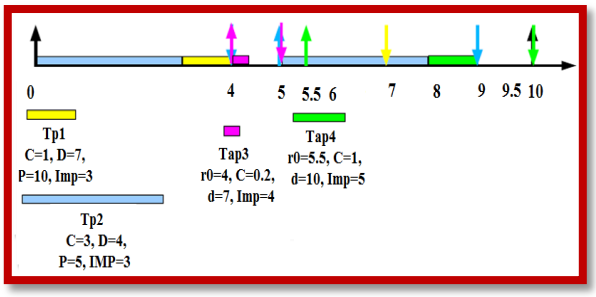
- Les tâches sont acceptées sur leur échéance secondaire et ordonnancées sur leur échéance primaire.
- Détection de la surcharge par les algorithmes classiques basés sur la laxité résiduelle.
- Pour un ensemble de tâches $J = \{J_1, J_2, \ldots, J_n\}$ classées par ordre d'échéances croissantes, la laxité résiduelle peut être calculée efficacement par la formule :
- $L_i = L_{i-1} + (d_i - d_{i-1}) - c_i(t)$
- on peut définir :
  - Le temps de dépassement : $E_i = \max(0, -(L_i + M_i))$
  - Le temps de dépassement maximal : $E_{max} = \max(E_i)$
- Un ordonnancement sera :
  - Faisable ssi $L_i \geq 0$ pour toutes les tâches
  - Tolérant ssi il existe des $L_i < 0$ mais que $E_{max} = 0$
- Plusieurs stratégies pour respecter les échéances des tâches critiques :
  - La plus simple : rejet d'une tâche, en choisissant la tâche de plus petite importance dont le temps d'exécution résiduel est plus grand que la surcharge.
- Les tâches rejetées sont mises dans une file d'attente, classées par ordre décroissant de valeur.
- Quand une tâche termine son exécution avant sa pire échéance, l'algorithme essaie de réinjecter les tâches de plus grande valeur et de laxité positive. Les tâches de laxité négative sont définitivement éliminées

Exemple : (comme sur la figure II.54)

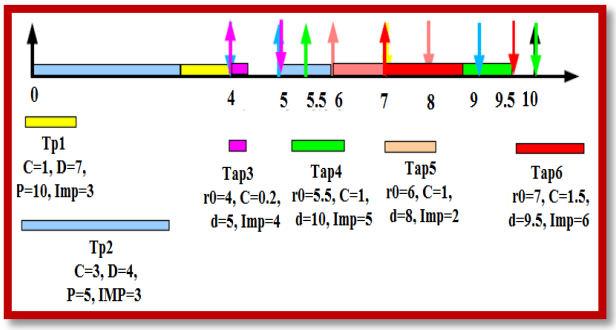
- Deux tâches périodiques
  - Tp_1 (r_0=0, C=1, D=7, P=10, I_{mp}=3)
  - Tp_2 (r_0=0, C=3, D=4, P=5, I_{mp}=1)
- Quatre tâches apériodiques
  - Tap_3 (r_0=4, C=0.2, d=5, I_{mp}=4)
  - Tap_4 (r_0=5.5, C=1, d=10, I_{mp}=5)
  - Tap_5 (r_0=6, C=1, d=8, I_{mp}=2)
  - Tap_6 (r_0=7, C=1.5, d=9.5, I_{mp}=6)
- Ordonnancées par EDF
- T(t) est la configuration des tâches actives à l'instant t
- d est la date de l'échéance (ne pas confondre avec le délai critique D).

À t=0, Tp_1 et Tp_2 se réveillent T(0)={Tp_2 (C(0)=3, d=4), Tp_1 (C(0)=1, d=7)} L(Tp_2) = 4 - 0 - 3 = 1 L(Tp_1) = 7 – 0 – 3 = 4 Pas de surcharge, Tp_2 est élue, puis Tp_1 est élue à t=3	
À t=4, Tap_3 se réveille T(4) = {Tap_3 (C(4)=0.2, d=5)} L(Tap_3) = 5 – 4 – 0.2 = 0.8 Pas de surcharge, Tap_3 est élue	
À t=5, Tp_2 se réveille T(5) = {Tp_2 (C(5)=3, d=9)} L(Tp_2) = 9 – 5 – 3 = 1 Pas de surcharge, Tp_2 est élue	
À t=5.5, Tap_4 se réveille T(5.5) = {Tp_2(C(5.5)=2.5, d=9), Tap_4 (C(5.5)=1, d=10)} L(Tp_2) = 9 – 5.5 – 2.5 = 1 L(Tap_4) = 10 – 5.5 – 1 – 2.5 = 1 Pas de surcharge, Tp_2 garde la **CPU**	

<table>
<tr>
<td>À t=6, Tap_5 se réveille

T(6) = {Tap_5 (C(6)=1, d=8), Tp_2(C(6)=2, d=9) Tap_4(C(6)=1, d=10)}
L(Tap_5) = 8 – 6 – 1 = 1
L(Tp_2) = 9 – 6 – 1 – 2 =0
L(Tap_4) = 10 – 6 – 1 – 2 – 1 = 0</td>
<td> 
</td>
</tr>
<tr>
<td colspan="2">À t=7, Tap_6 se réveille

T(7) = {Tp_2 (C(7)=2, d=9), Tap_6 (C(7)=1.5, d=9.5), Tap_4(C(7)=1, d=10)}
L(Tp_2) = 9 – 7 - 2 = 0
L(Tap_6) = 9.5 – 7 – 2 - 1.5 = -1
Surcharge, Tap_6 est fautive avec une surcharge de 1
Suppression de tâches dont les échéances sont inférieures ou égales à celles de Tap_6 par ordre croissant des importances ➔ suppression de Tp_2</td>
</tr>
<tr>
<td>Nouvelle configuration à t=7

T(7) = {Tap_6(C(7)=1.5, d=9.5), Tap_4(C(7)=1, d=10)}
L(Tap_6) = 9.5 – 7 – 1.5 = 1
L(Tap_4) = 10 – 7 – 1 – 1.5 = 0.5</td>
<td> 
</td>
</tr>
<tr>
<td>Si on avait utilise un simple algorithme à routine de garantie

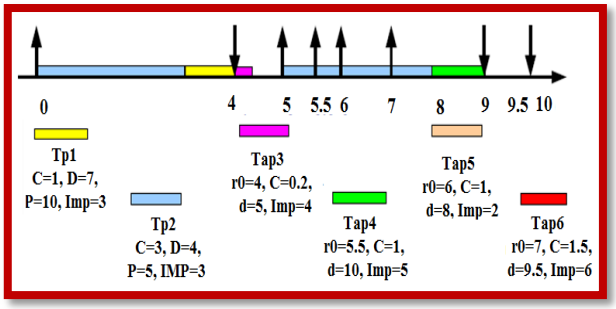
Temps creux [4,5] = 1: Tap_3 acceptée
Temps creux [5.5, 10] = 2 : Tap_4 acceptée
Temps creux [6, 8] = 0 : Tap_5 refusée
Temps creux [7, 9.5] = 1.5, mais Tap_4 doit rester garantie : Tap_6 refusée</td>
<td> 
</td>
</tr>
</table>

Figure II.54 – Exemple d'optimisation

II.12 – Conclusion

On a vu comment ordonnancer théoriquement des tâches ayant des contraintes temporelles de façon à respecter ces contraintes ou a minimiser les effets du non-respect, tout en ayant éventuellement des contraintes de précédence et/ou de partage de ressources.

On va pouvoir étudier maintenant les outils qui vont nous aider à satisfaire ces contraintes :

- Systèmes d'exploitation
- Supports physiques de communication (bus de terrain)
- Outils de programmation
    - Gestion des tâches (création, destruction, priorités, etc.)
    - Gestion du temps (alarmes, chronomètres, etc.)
    - Outils de communication (files de messages, mémoire partagée, etc.)
    - Outils de synchronisation (sémaphores, mutex, variables conditionnelles, etc.).