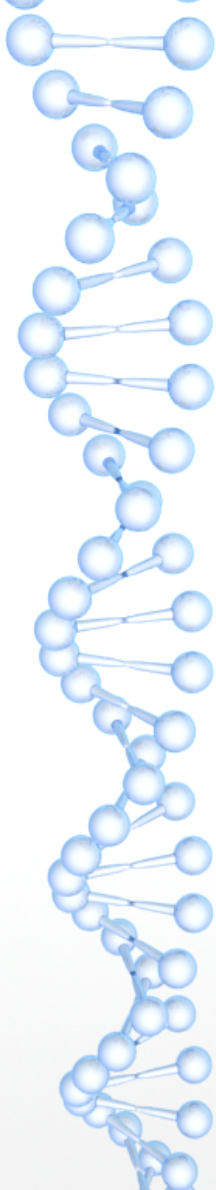
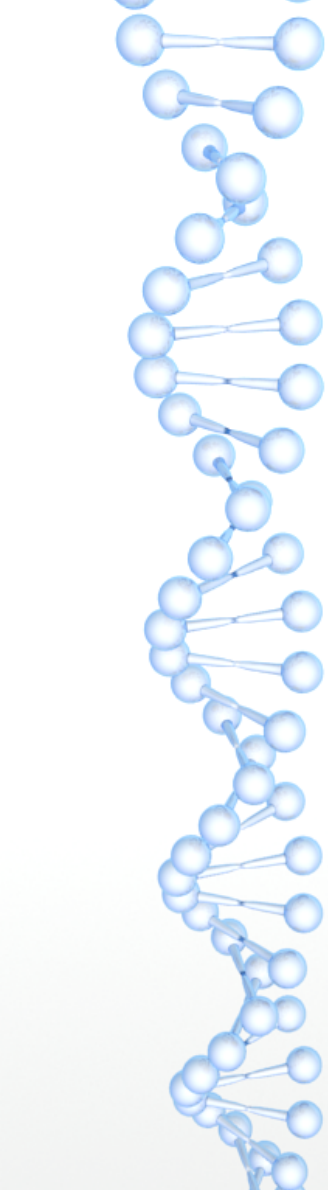


M1 : Systèmes temps réel et Ordonnancement

Université du Havre - UFR des Sciences et Techniques
Laboratoire LITIS

25 rue Philippe Lebon - BP 540
76058 LE HAVRE CEDEX
bruno.sadeg@univ-lehavre.fr

http://litis.univ-lehavre.fr/~sadeg/enseignement/enseignement.html
(voir titre : master1)



SOMMAIRE

- Historique
- Ordonnancement de tâches temps réel indépendantes :
  - Tâches périodiques
  - Tâches périodiques en présence de tâches apériodiques
- Ordonnancement de tâches temps réel dépendantes :
  - Dépendance statique (graphe de précédence)
  - Dépendance dynamique (partage de ressources)

- le temps a toujours joué un rôle important dans les services et les applications (informatique industrielle, télécommunications, …).
  Actuellement, le temps devient encore de plus en plus important.

Exemples :

- En informatique industrielle : c'était les prémisses de *l'informatique temps réel,*

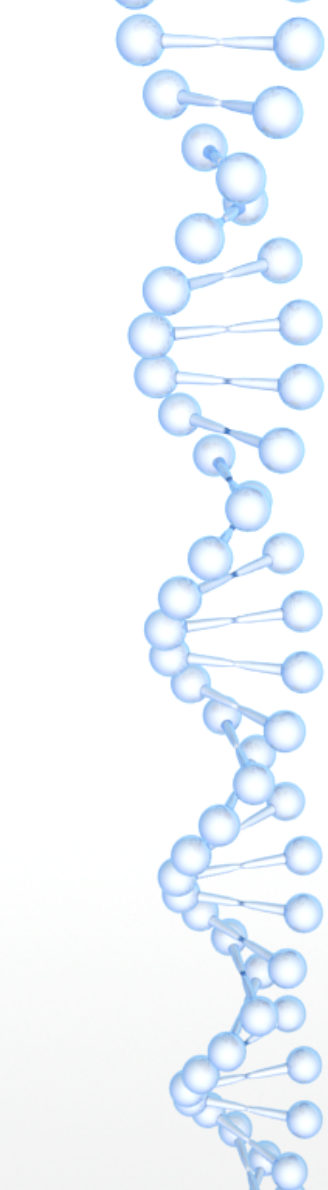
- Dans les télécommunications :
  - (1) on cherche à a obtenir des débits élevés (borner la durée du trafic),
  - (2) on doit gérer plusieurs flux d'information => une synchronisation entre les flux est nécessaire (comme dans les applications multimédia),

Quand on ne peut pas respecter les échéances : on associe, à chaque service, une **qualité de service** (QoS) qui doit être respectée : temps d'établissement d'une connexion ou temps de réponse d'une requête, etc.

Définition d'un système temps réel (STR)

STR = système dont le comportement dépend non seulement de **l'exactitude** des traitements effectués, mais également de **l'instant** où les résultats de ces traitements sont fournis,

c-à-d qu'un retard dans la production d'un résultat est considéré comme une erreur (pouvant entraîner de graves conséquences)



Exemples d'applications temps réel

- La commande de procédés industriels,
- Les systèmes embarqués,
- Le guidage de mobiles,
- La surveillance de centrales (thermiques, nucléaires, ...)
- Conduite d'expériences scientifiques,
- La fourniture d'images et de son pour les applications multimédia,
- Le suivi d'informations boursières,
- Le suivi opératoire en milieu médical,
- ...

Un exemple détaillé

- Un Système de vidéo-conférence sur un réseau local :

  => le système numérise l'image, puis la séquence. Il faut que le système traite

  30 images/sec pour que l'image soit acceptable.

  => Plusieurs opérations vont s'enchaîner dans le temps :

  (1) numérisation, (2) compression, (3) transmission.

  La durée de ces opérations est le temps de latence du système.

  (La voix doit également être (1) numérisée, (2) compressée et (3) transmise).

  Il faut que le son et l'image soient synchronisés : un retard léger dans l'arrivée de l'un des flux est acceptable. Mais si le décalage est trop long, la scène devient incompréhensible

  => c'est le temps réel mou, ou échéances molles, relatives.

Types d'échéances et de systèmes temps réel

- Échéance dure (hard) : le dépassement de l'échéance provoque une exception dans le système (qui peut engendrer des dommages)

  ==> STR durs : les échéances ne doivent en aucun cas être dépassées

  (sinon catastrophe humaine ou industrielle ou économique, ...)

- Échéance molle, lâche (soft) : un dépassement d'échéance ne provoque pas d'exception dans le système

  ==> STR mous : les échéances peuvent être dépassées occasionnellement

  (ex. usine automatisée)

Caractéristique d'un système temps réel

- **La prévisibilité**, qui est un aspect important d'un STR.

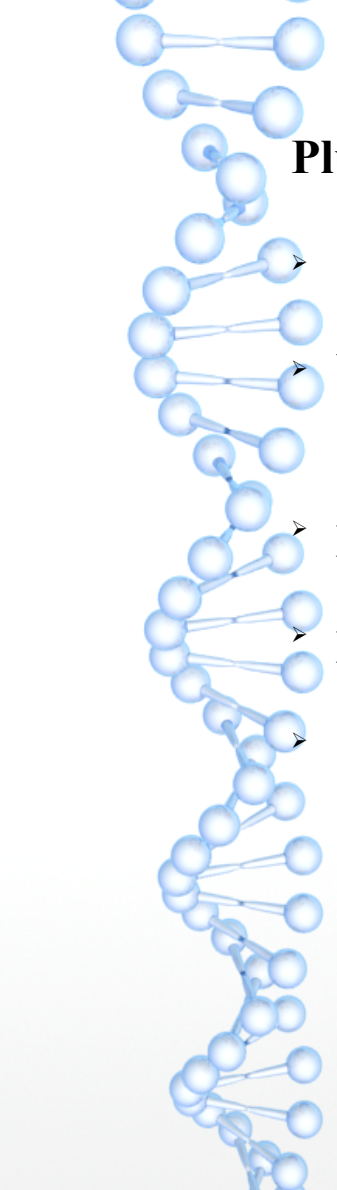
  (Un STR doit être conçu tel que ses performances soient définies dans le **pire cas**, alors que dans les systèmes classiques, les performances sont définies en termes de **moyenne**).

- La prévisibilité est le fait de savoir à **l'avance** si un système va respecter ses contraintes temporelles (C.T.). Pour cela, il est nécessaire de connaître avec précision les paramètres des tâches : temps global de calcul de chaque activité, périodicité, date de réveil, etc.

  Ces éléments définissent le choix d'une **politique d'ordonnancement** des activités de manière à ce que le système soit prévisible.

Autre caractéristique

- **Le déterminisme** d'un STR : est le but à atteindre pour assurer la prévisibilité. Pour qu'un système soit prévisible, il faut qu'il soit déterministe. Donc enlever toute incertitude sur le comportement des activités (individuelles et mises ensemble).
  - Dans un STR dur : on cherche à ce que toutes les échéances soient respectées
  - Dans un STR mou : on cherche, par exemple, à minimiser le retard moyen des activités (ou à maximiser le nombre de tâches qui respectent leurs échéances).



Plusieurs états d'une tâche

- élue : un processeur est alloué à la tâche
- bloquée : la tâche attend une ressource, un message ou un signal de synchronisation
- prête (éligible) : la tâche attend d'être élue
- passive : la tâche n'a pas de requête en cours
- inexistante : la tâche n'est pas créée

Principales politiques d'ordonnancement classique

- **Objectifs** : maximiser le taux d'utilisation du processeur (temps où le processeur est actif sur le temps total) => Déterminer quelle tâche prête doit être élue (s'exécuter sur le processeur)

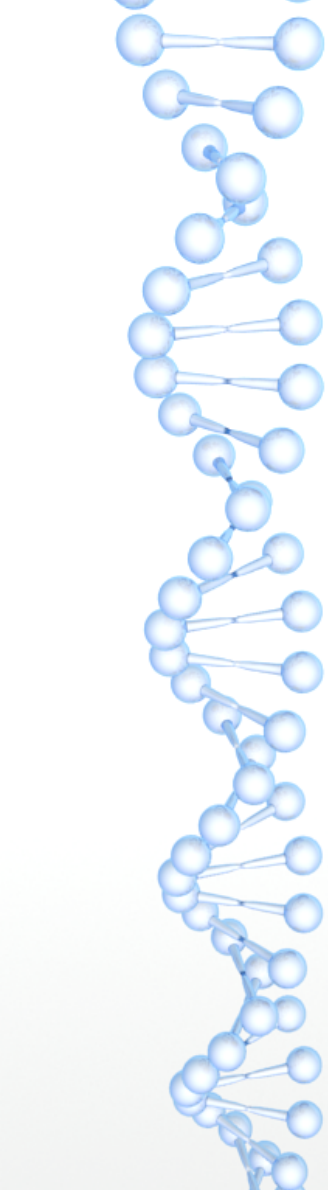
1) Premier arrivé, premier servi (PAPS ou FIFO) : les tâches de faible temps d'exécution sont pénalisées si elles sont précédées dans la file par une (ou des) tâche(s) de longue durée d'exécution.
2) Plus court d'abord : remédie à l'inconvénient précédent. Minimise le temps de réponse moyen. Mais pénalise les travaux longs. De plus, elle impose d'estimer les durées d'exécution des tâches (connues difficilement).
3) Temps restant le plus court d'abord : la tâche en exécution restitue le processeur lorsqu'une nouvelle tâche ayant un temps d'exécution inférieur à son temps d'exécution restant devient prête.
4) Tourniquet (Round Robin) : un quantum de temps est alloué pour chaque tâche. Chaque tâche s'exécute pendant son quantum, ...

1)

(5) Files de priorités constantes multiniveaux : on définit plusieurs files de tâches prêtes. Chaque file correspond à un niveau de priorité (n files de priorités variant de 0 à n-1 : file i, tâches de même priorité). Elles sont gérées, soit par ancienneté, soit par tourniquet Le quantum peut être différent selon la file. L'ordonnanceur sert d'abord les tâches de la file 0 , puis celles de la file 1, moins prioritaire (dès que la 0 se vide), etc. Deux variantes :

- les priorités des tâches sont constantes tout au long de leur exécution : une tâche en fin de quantum est réinsérée dans la même file.
- les priorités des tâches évoluent dynamiquement en fonction des services dont elles ont déjà bénéficié. Une tâche de la file i, qui n'a pas terminé son exécution à la fin de son quantum est réinsérée dans la file i + 1 (moins prioritaire), etc.

On minimise ainsi les risques de famine des tâches de faible priorité en diminuant petit à petit les priorités des tâches ayant de fortes priorités initiales.



Constat

- Aucune des politiques classiques ne permet de remplir l'objectif principal d'un ordonnancement temps réel, c-à-d pas de prise en compte de l'urgence des tâches (délai critique)

==> Ordonnancement spécifique aux tâches temps réel

Terminologie pour l'ordonnancement des tâches temps réel

- **tâche** = unité de base de l'ordonnancement temps réel. Une tâche peut être :

  (1) périodique ou apériodique, (2) à contraintes temporelles strictes ou relatives.

- **Modèle de tâche** : paramètres chronologiques (des délais) et chronométriques (des dates). Les paramètres de base d'une tâche (périodique) sont :
  - **r** : sa date de réveil, moment de déclenchement de la 1ère requête d'exécution
  - **C** : sa durée maximale d'exécution (si elle dispose du processeur à elle seule)
  - **D** : son délai critique (délai maximal acceptable pour son exécution)
  - **P** : sa période (si tâche périodique)
- **Remarque** : Si une tâche est à contraintes strictes : l'échéance d = r + D, est la date dont le dépassement entraîne une faute temporelle.

Illustration : diagramme temporel d'exécution

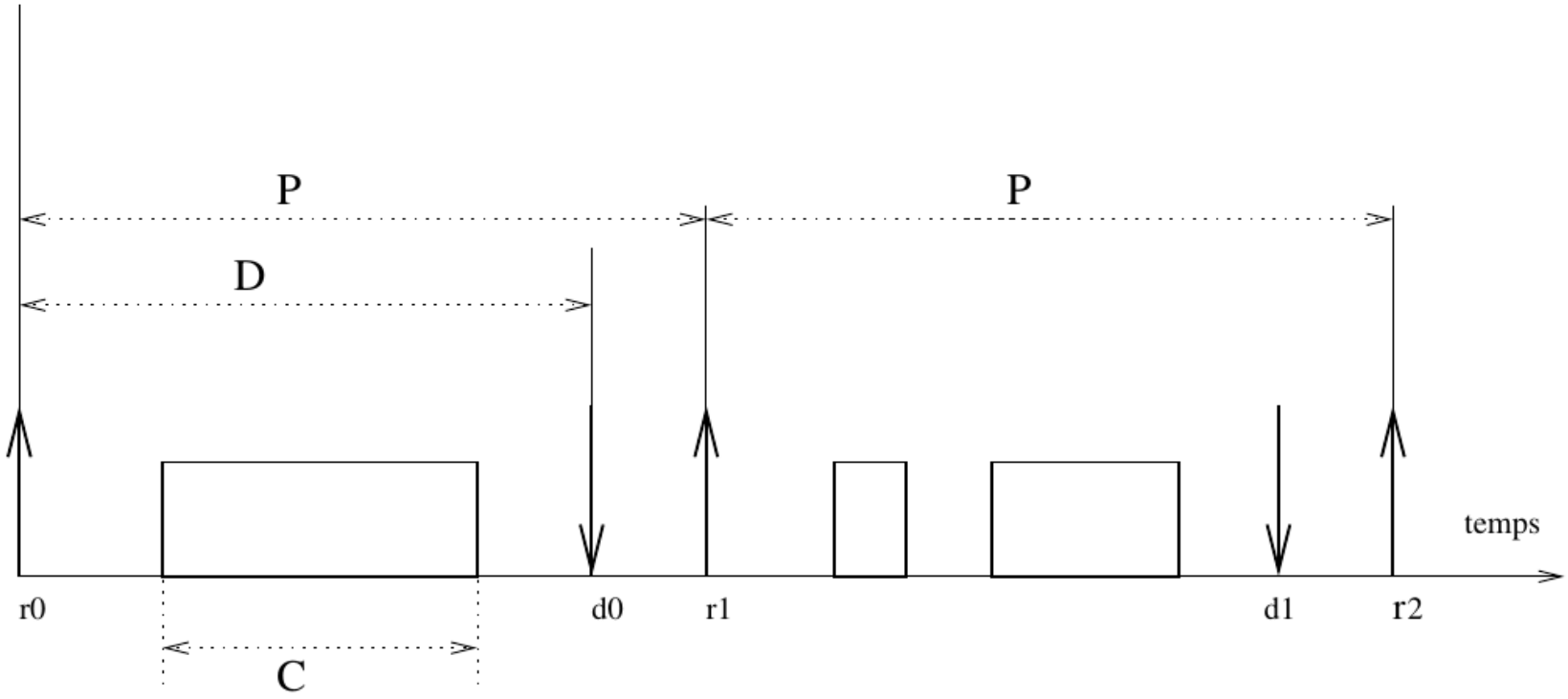


Figure: Modèle de tâches

Commentaires

T (r_0 , C, D, P) avec $0 < C \leq D \leq P$

- **r_0** : date de réveil, C : durée max d'exécution, D : délai critique, P : période

- **r_k** : date de réveil de la k-ème requête de la tâche. On a :

  r_k = r_0 + kP, représentée par ↑

- **d_k** : échéance de la k-ème requête de la tâche. On a :

  d_k = r_k + D, représentée par ↓

Remarque : une tâche est *à* **échéance sur requête** quand D = P, représentée par ↕

Autres paramètres déduits

U = C / P (doit être ≤ 1) : facteur d'utilisation du processeur par la tâche

Ch = C / D (doit être ≤ 1) : facteur de charge du processeur

Paramètres dynamiques (pour suivre l'exécution d'une tâche dans le temps) :

s : date de début de l'exécution d'une tâche

e : date de fin d'exécution d'une tâche

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 ($0 \leq C(t) \leq C$)

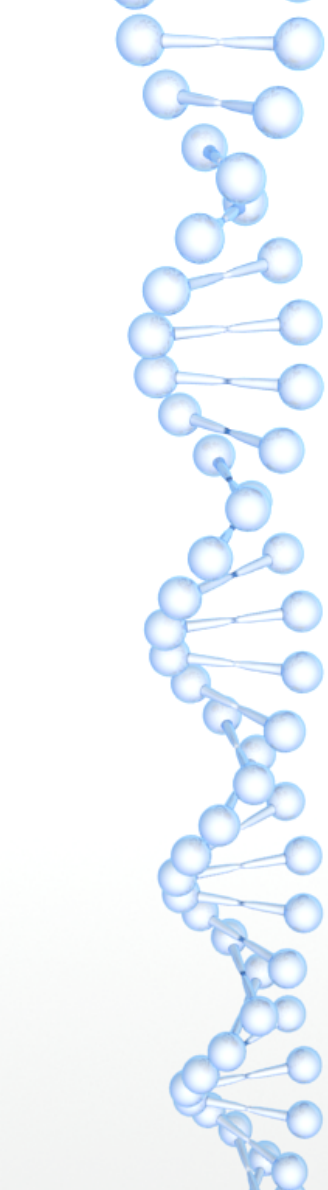
Autres paramètres déduits

L = D – C : laxité nominale de la tâche, retard maximum pour son début d'exécution s quand la tâche s'exécute seule.

L(t) = D(t) – C(t) : laxité nominale résiduelle à l'instant t = retard maximum pour reprendre l'exécution d'une tâche à partir de t, si elle s'exécute seule,

TR = e – r : temps de réponse de la tâche,

CH (t) = C(t) / D(t) : charge résiduelle au temps t



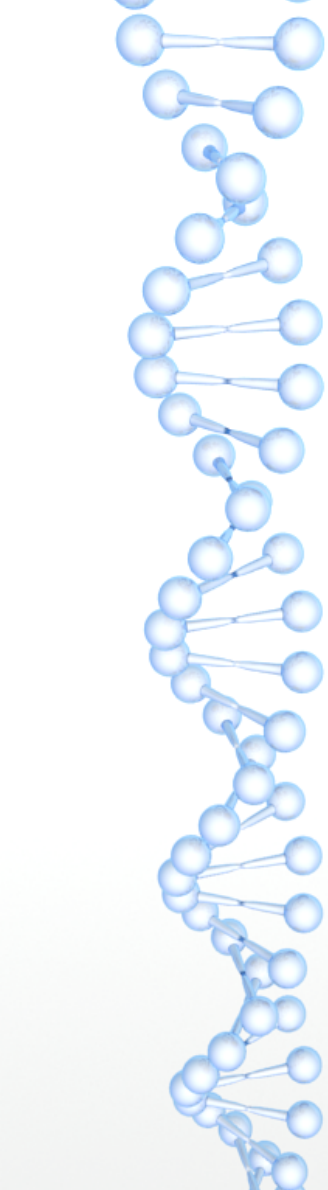
Autres caractéristiques d'une tâche

- **préemptible** : tâche élue qui peut être arrêtée et remise à l'état prêt, pour allouer le processeur à une autre tâche,

- **non préemptible** : une fois élue, la tâche ne doit plus être interrompue jusqu'à la fin de la requête,

- **tâches indépendantes** : lorsqu'elles n'ont pas de relations de précédence, ni ne partagent des ressources critiques (sauf le processeur)

- **tâches dépendantes** : dépendance statique (on peut tracer le graphe de précédence) ou dynamique (les tâches partagent des ressources)

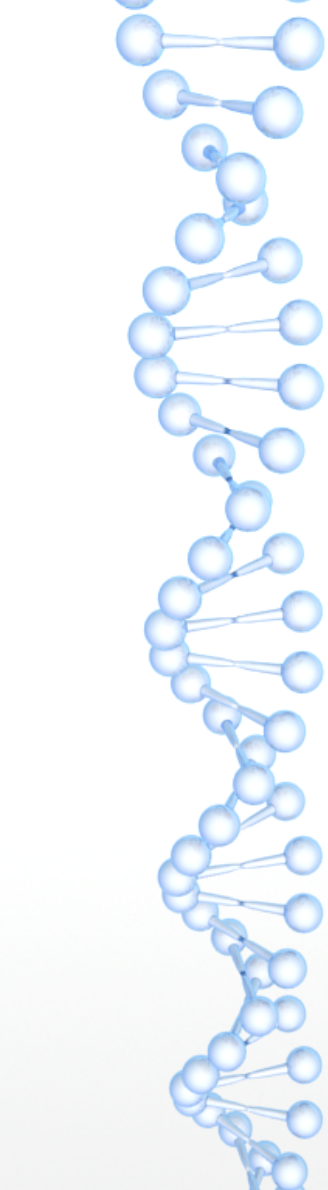


Ordonnancement temps réel

Son but : permettre le respect des contraintes temporelles des tâches d'une application quand elles s'exécutent en mode courant,

L'ordonnancement doit être **certifiable**, c-à-d prouver a priori le respect des C.T. des tâches d'une application (en régime courant)

En régime de **surcharge** (tâches supplémentaires suite à des anomalies : alarmes, variations des temps d'exécution, … => fautes temporelles) : l'ordonnancement doit offrir une tolérance aux surcharges (permettre une exécution dégradée, mais sécuritaire du système)



Définitions

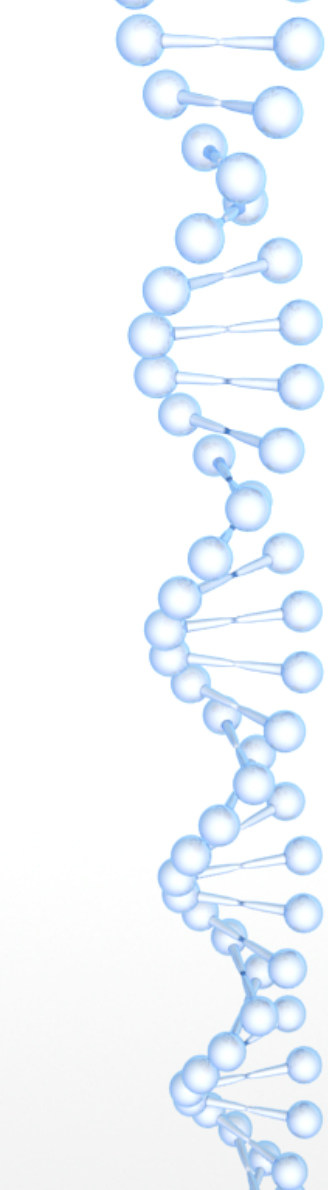
Configuration de tâches : mise en jeu d'un ensemble de n tâches qui s'exécutent.

Départ simultané : même date de réveil (sinon départ **échelonné**).

Facteur d'utilisation du processeur pour n tâches périodiques : U = $\sum_{i=1}^{i=n} \frac{C_i}{P_i}$

Facteur de charge du processeur pour n tâches périodiques : CH = $\sum_{i=1}^{n} \frac{C_i}{D_i}$

Laxité du processeur à l'instant t, LP (t) = intervalle de temps à partir de t pendant lequel le processeur peut rester inactif sans remettre en cause le respect des échéances.



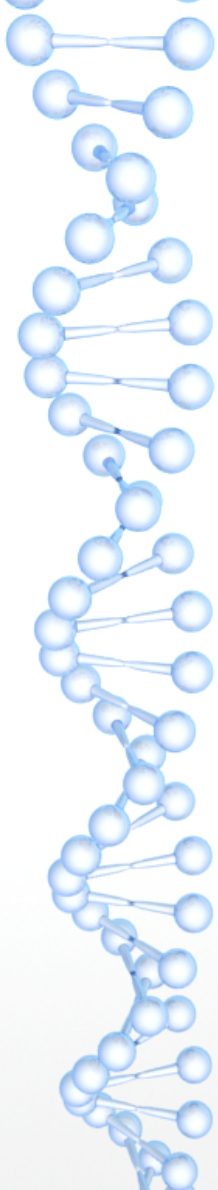
Définition du problème d'ordonnancement

Le système de conduite d'une application temps réel doit piloter le procédé en ordonnançant les tâches avec 2 objectifs :

1) assurer le respect des C.T. en fonctionnement nominal,

2) atténuer les effets des surcharges et maintenir le procédé dans un état cohérent et sûr en fonctionnement anormal (pannes matérielles ou logicielles)
=> respecter au moins les CT des requêtes vitales pour le procédé.

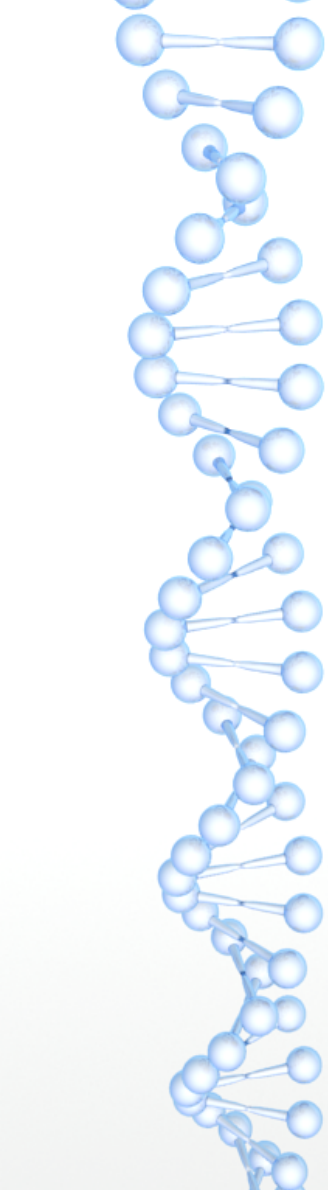
L'algorithme d'ordonnancement fournit une description de la **séquence de planification** des tâches à effectuer de manière à respecter les CT.



Typologie des algorithmes d'ordonnancement

Hors ligne : l'algorithme construit la séquence complète de planification des tâches sur la base de tous les paramètres temporels des tâches.
=> Séquence connue avant l'exécution. Très efficace, mais cette approche statique est rigide, donc elle ne s'adapte pas aux changements de l'environnement (aux tâches apériodiques)

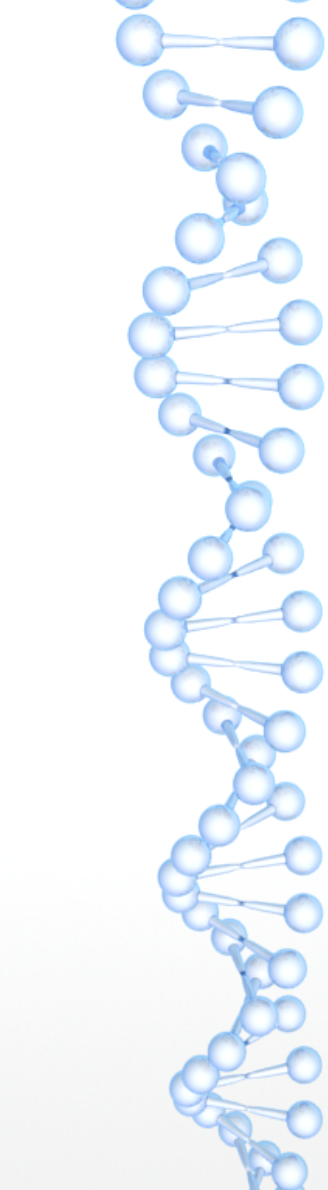
En ligne : capable à tout instant de l'exécution d'une application de choisir la prochaine tâche à ordonnancer, en utilisant les informations des tâches déclenchées à cet instant. Ce choix peut être remis en cause par l'occurrence d'un nouvel événement.
Cette approche dynamique offre des solutions moins bonnes (car elle utilise moins d'informations) et entraîne des surcoûts de mise en œuvre, MAIS possède des **avantages** : permet l'arrivée imprévisible des tâches, autorise la construction progressive de la séquence d'ordonnancement (gère les tâches apériodiques).



Propriétés des algorithmes d'ordonnancement

Séquence valide : un algorithme d'ordonnancement délivre une séquence d'ordonnancement valide pour une configuration de tâches donnée si toutes les tâches respectent leur CT.

Une configuration est ordonnançable dès qu'il existe au moins un algorithme capable de fournir une séquence valide pour cette configuration.



Tests

Test d'ordonnançabilité : vérifie qu'une configuration de tâches périodiques donnée soumise à un algorithme d'ordonnancement peut être ordonnancée selon une séquence valide.

Test d'acceptabilité : un algorithme en ligne modifie dynamiquement la séquence d'ordonnancement à l'arrivée de nouvelles tâches (ou à la suite du dépassement d'échéances). Une tâche nouvelle peut être ajoutée à la configuration s'il existe au moins une séquence valide avec la nouvelle configuration (tâches existantes + celle ajoutée).

L'ensemble des conditions à satisfaire est le **test d'acceptabilité** ou **routine de garantie**.

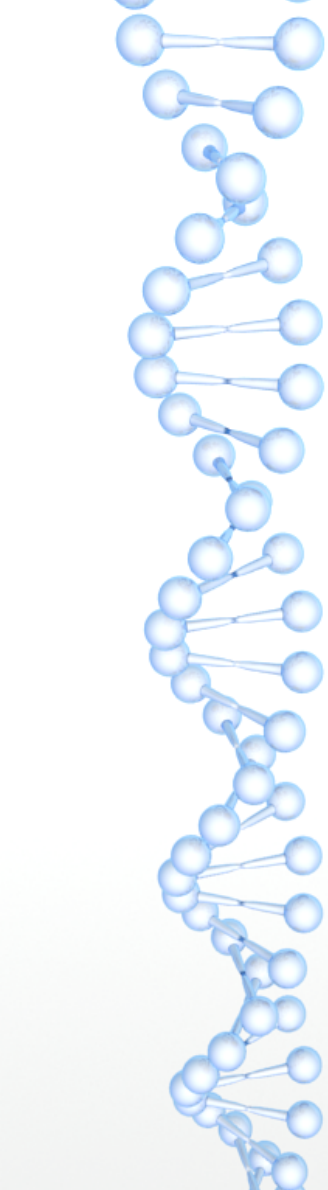
Propriétés

Période d'étude ou intervalle d'étude : Pour valider une configuration de tâches périodiques et apériodiques, on effectue une analyse temporelle de l'exécution de la configuration durant cet intervalle.

=> Pour les tâches **périodiques, cet intervalle est :** $[\min\{r_{i0}\}, PPCM(P_i)]$

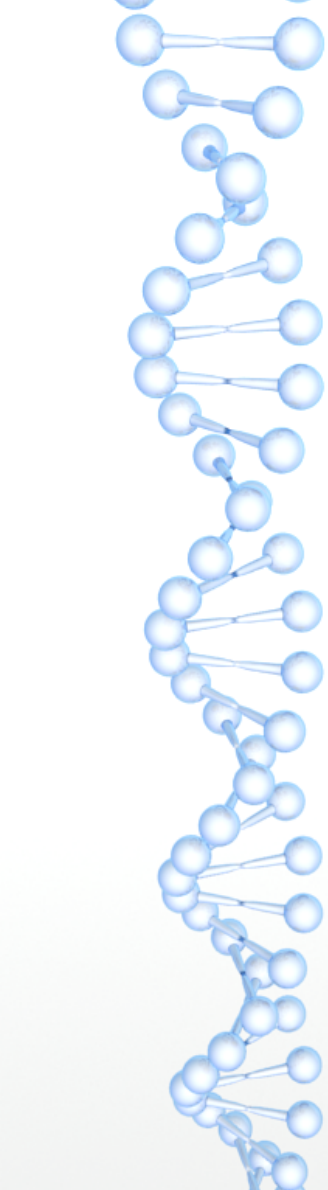
où P_i = période de la tâche i.

Souvent, $\min\{r_{i0}\} = 0$, donc on a l'intervalle d'étude : $[0, PPCM(P_i)]$.



Algorithmes en ligne, à priorité constante

- Rate Monotonic (RM)
- Inverse Deadline (ID) ou Deadline Monotonic (DM)



Algorithme en ligne, à priorité variable)

- Earliest Deadline (ED) ou Earliest Deadline First (EDF)
- Least Laxity (LL) ou Least Laxity First (LLF)

Ordonnancement de tâches indépendantes

A. Algorithme En ligne, à priorité constante (où D=P)

A.1. Rate Monotonic (ou RM) : priorité d'une tâche = f(sa période). La tâche de plus petite période est la plus prioritaire. L'algorithme est optimal dans la classe des algorithmes à priorité constante pour une configuration de tâches à échéances sur requête. Dans ce cas, on connaît une condition suffisante d'existence d'une borne minimale pour l'acceptation d'une configuration de n tâches :

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

- **Remarque** : Cette borne est le pire cas. Elle tend vers log 2 quand n est très grand
- (c-à-d 69%). En pratique, elle peut être dépassée (en moyenne, elle est de 88%), c-à-d qu'en pratique, le taux d'utilisation du processeur = 88%.

Valeurs de $U(n) = n * (2^{1/n} - 1)$

$U(1) = 1$ - $U(2) \approx 0.828$ - $U(3) \approx 0.779$

$U(4) \approx 0.756$ - $U(5) \approx 0.743$ - $U(6) \approx 0.734$

Pour $n \to \infty$, $U \to \ln 2 \approx 69\%$

Exemple d'ordonnancement avec RM

La figure 3 montre l'exemple d'un ordonnancement "Rate Monotonic" pour 3 tâches périodiques à échéance sur requête (échéance = période).
T1 (r0 = 0, C = 3, P = 20), T2 (r0 = 0, C = 2, P = 5), T3 (r0 = 0, C = 2, P = 10).

Tâche de plus haute priorité : T2 , tâche de plus basse priorité : T1.
La période d'étude est l'intervalle [0, ppcm(20, 5, 10)] = [0, 20].

Les 3 tâches respectent leur CT car la condition suffisante est vérifiée :
on a bien 3/20 + 2/5 + 2/10 = 0.75, qui est < à 3 $(2^{1/3} - 1) = 0.779$,
MAIS on trace le chronogramme pour vérifier (voir figure).

Remarque : l'utilisation de la période comme critère d'ordonnancement limite l'application de "Rate Monotonic" aux seules tâches où l'échéance est égale à la période (échéances sur requête).

Chronogramme de ces tâches avec ’Rate Monotonic’

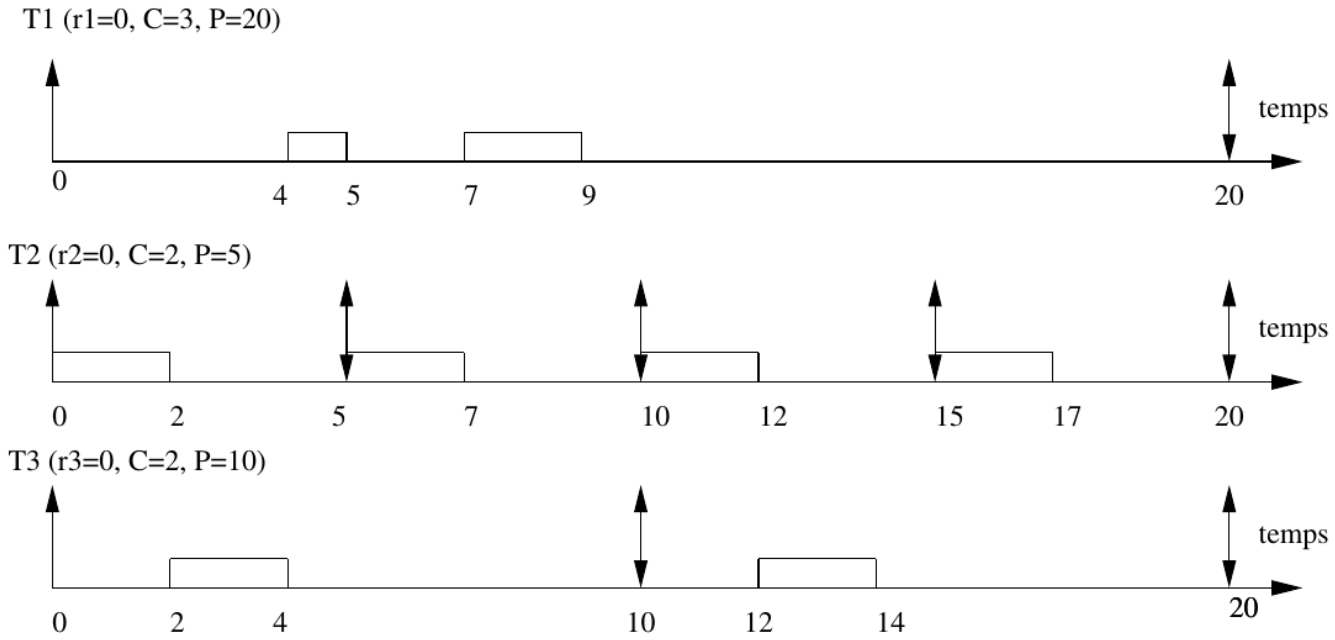
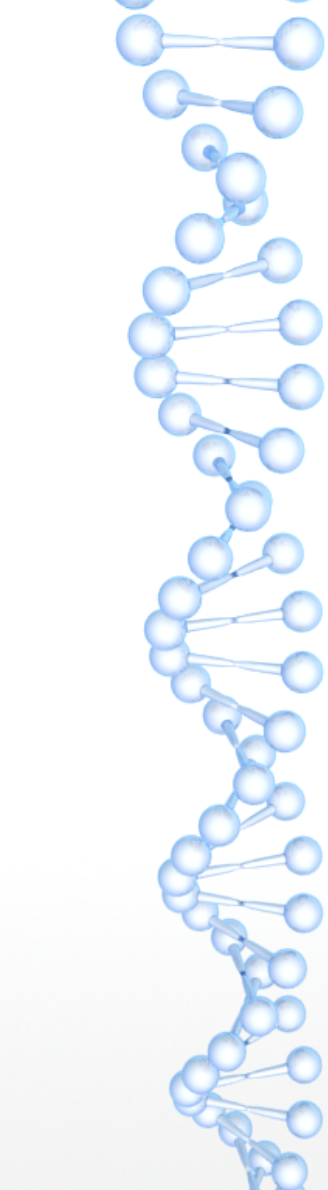


Figure: Ordonnancement “Rate Monotonic”

Figure 3 : Ordonnancement “Rate Monotonic”



Ordonnancement de tâches indépendantes

Algorithme En ligne, à priorité constante (où D <> P)

A.2. Inverse Deadline ou Deadline Monotonic :

priorité d'une tâche = f(son délai critique).

La tâche la + prioritaire est celle de + petit délai critique.

Cet algorithme est égal en performances à "Rate Monotonic" pour les tâches à échéances sur requête, et meilleur pour les autres types de configurations de tâches.

Condition suffisante d'acceptabilité :

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

Exemple d'ordonnancement avec 'DM' (ou 'ID')

La figure 4 montre l'exemple d'un ordonnancement "Deadline Monotonic" pour 3 tâches périodiques :

T1 (r0 = 0, C = 3, D = 7, P = 20), T2 (r0 = 0, C = 2, D = 4, P = 5),
T3 (r0 = 0, C = 2, D = 9, P = 10).
Tâche de plus haute priorité : T2 , tâche de plus basse priorité : T3 .

La condition suffisante n'est pas vérifiée : $3/7 + 2/4 + 2/9 = 1.15 > 3\,(2^{1/3} - 1) = 0.779$.

On doit tracer le chronogramme sur l'intervalle d'étude,
où l'intervalle d'étude est [0, ppcm(20, 5, 10)] = [0, 20] (voir figure).

Conclusion : On constate que le chronogramme construit sur la période d'étude montre que les 3 tâches sont ordonnançables, sans faute temporelle.

Chronogramme de ces tâches avec 'Deadline Monotonic'

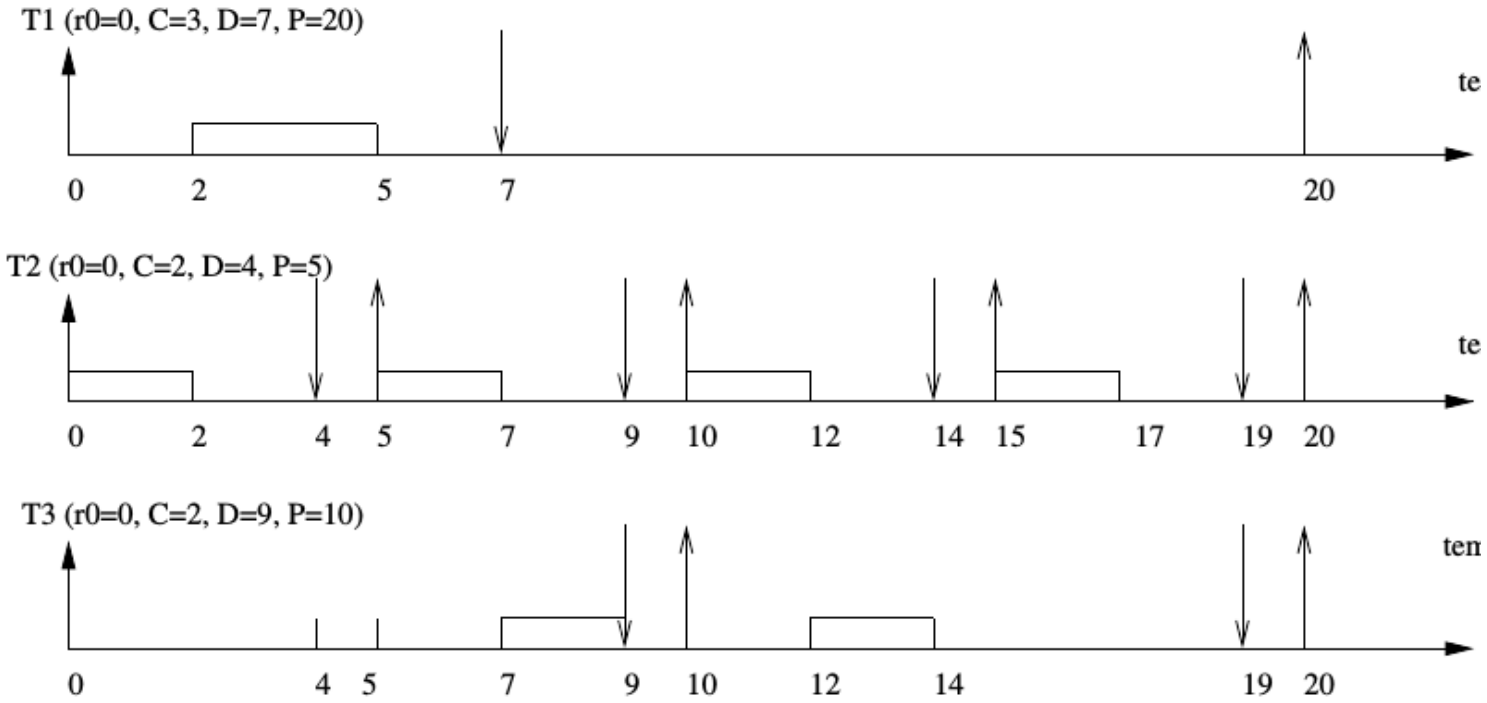


Figure: Ordonnancement "Inverse Deadline ou Deadline Monotonic"

Figure 4 : Ordonnancement "Inverse Deadline" ou "Deadline Monotoni

Ordonnancement de tâches indépendantes

B. Algorithmes en ligne à priorité variable : ED ou EDF

(La priorité d'une tâche varie dans le temps.)

B.1. Earliest Deadline : (ED) : La plus haute priorité à l'instant t est accordée à la tâche dont l'échéance est la plus proche (EDF : Earliest Deadline First).

- Pour les tâches à **échéances sur requête :**
  - une **CNS** d'ordonnancement est : $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$

- Pour des tâches **périodiques quelconques :**
  - une **condition suffisante** est : $\sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1$
  - et une **condition nécessaire** est : $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$

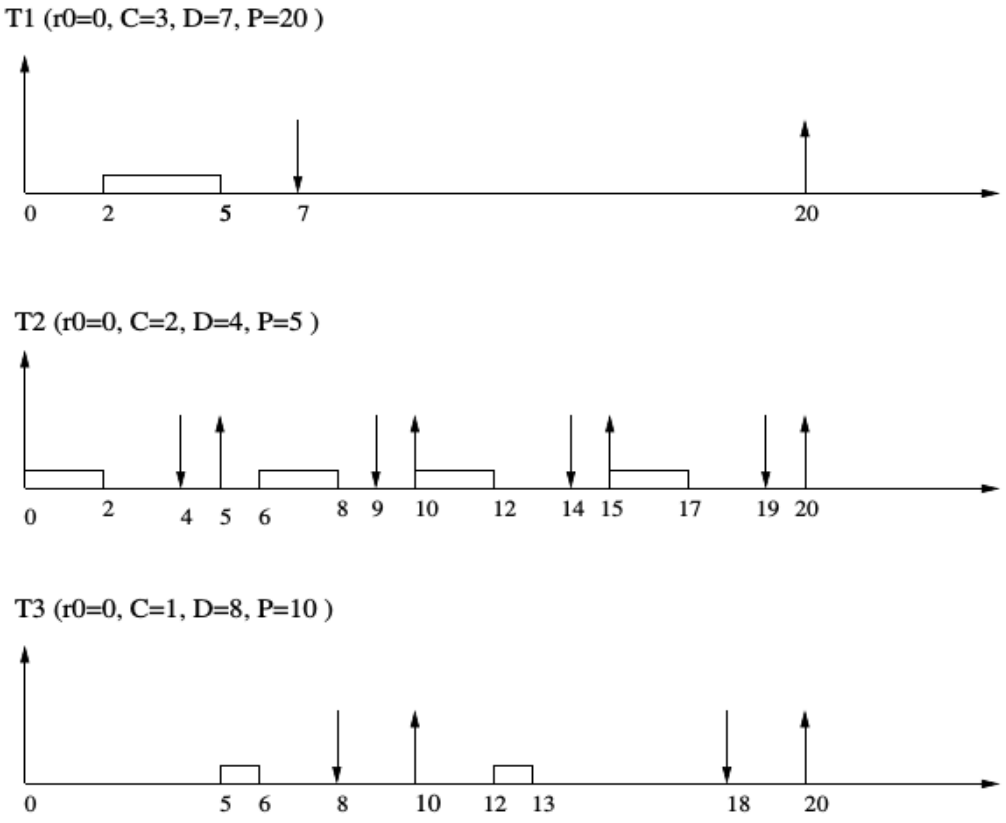
Exemple

La figure 5 montre l'exemple d'un ordonnancement "Earliest Deadline" pour 3 tâches périodiques. T1 (r0 = 0, C = 3, D = 7, P = 20),
T2 (r0 = 0, C = 2, D = 4, P = 5), T3 (r0 = 0, C = 1, D = 8, P = 10).

La CS : 3/7 + 2/4 + 1/8 est > 1, donc condition suffisante non respectée. On trace le chronogramme sur l'intervalle d'étude [0, ppcm(20, 5, 10)] = [0, 20] :

- à t=0 : au réveil des 3 tâches, T2 prioritaire, elle s'exécute pendant 2 unités.
- à t=2, T2 termine. C'est T1 la + prioritaire. Elle s'exécute pendant 3 unités de temps.
- à t=5, T1 se termine et T2 se réveille de nouveau (car période 5), mais c'est T3 qui est cette fois la + prioritaire car son échéance est 8, elle s'exécute pendant une unité.
- Ici, on voit que les priorités des tâches varient dans le temps : par exemple, à t=0, c'est T2 qui est plus prioritaire (que T1 et T3), mais à t=5, c'est T3 la plus prioritaire.

Chronogramme de ces tâches avec 'ED' (ou 'EDF')



Ordonnancement ED (ou EDF)

Figure 5 : Ordonnancement

"Earliest Deadline"

Ordonnancement de tâches indépendantes

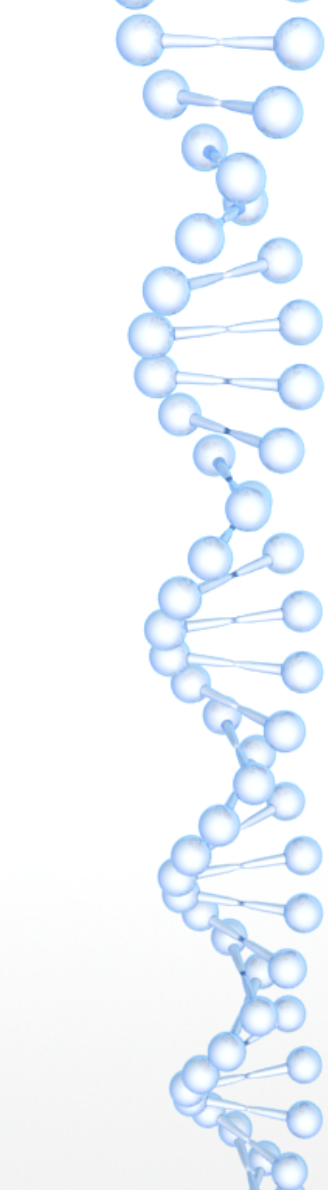
B. Algorithmes en ligne à priorité variable : LL ou LLF

(La priorité d'une tâche varie dans le temps.)

B.2. Least Laxity : (LL) : La plus haute priorité à l'instant t est accordée à la tâche qui a la plus petite laxité dynamique $L_i(t)$ (LLF : Least Laxity First). Comme EDF :

- Pour les tâches à **échéances sur requête :**
  - une **CNS** d'ordonnancement est : $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$

- Pour des tâches **périodiques quelconques :**
  - une **condition suffisante** est : $\sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1$
  - et une **condition nécessaire** est : $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$



Remarque : cet algorithme est optimal et les conditions d'ordonnançabilité des tâches sont les mêmes que pour Earliest Deadline.

Les séquences produites par LL sont équivalentes à celles produites par ED lorsque les valeurs des laxités sont calculées aux dates de réveil.

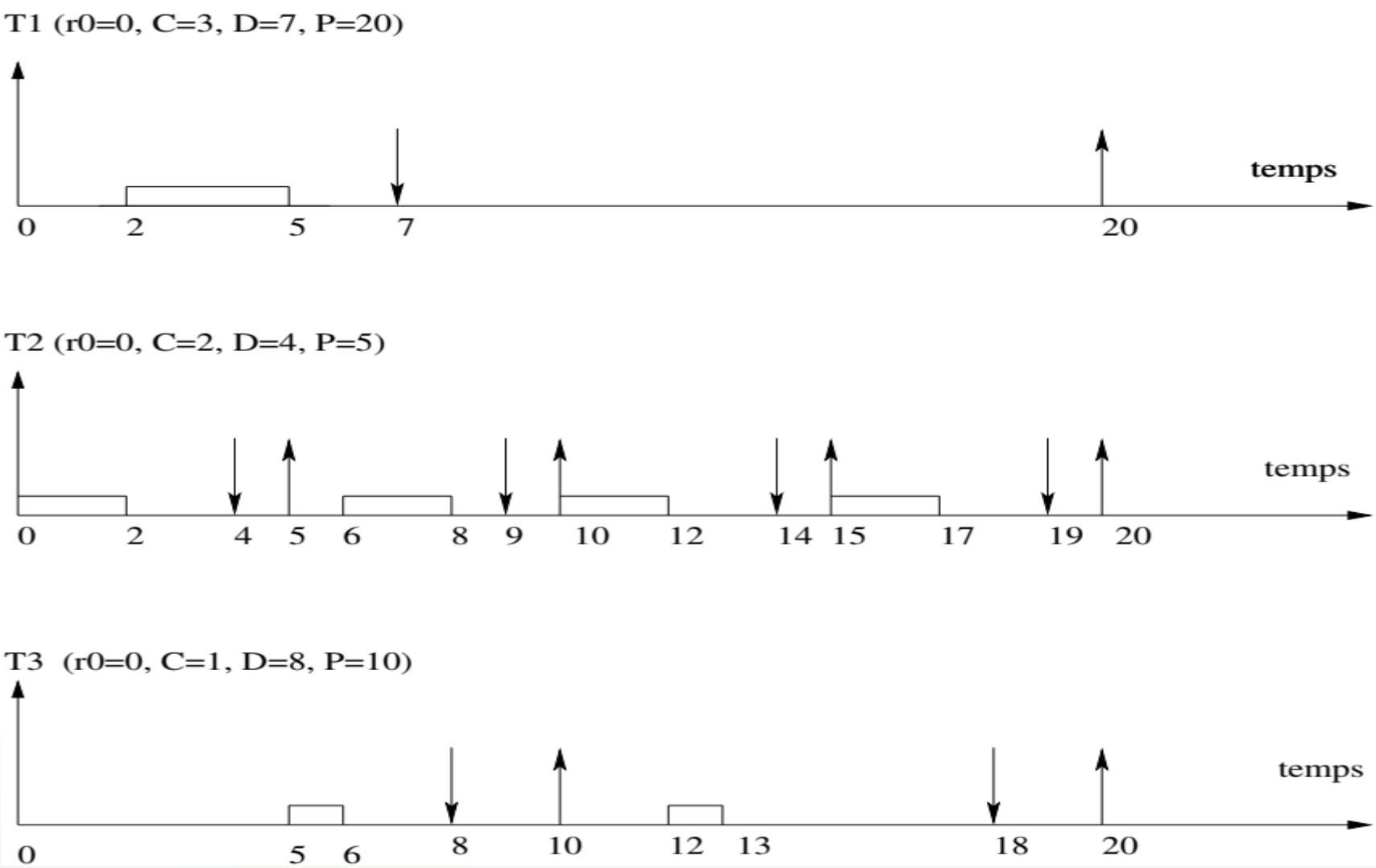
Par contre, si on calcule la laxité à chaque instant alors la séquence produite par LL entraîne plus de changements de contextes que par ED.

Exemple

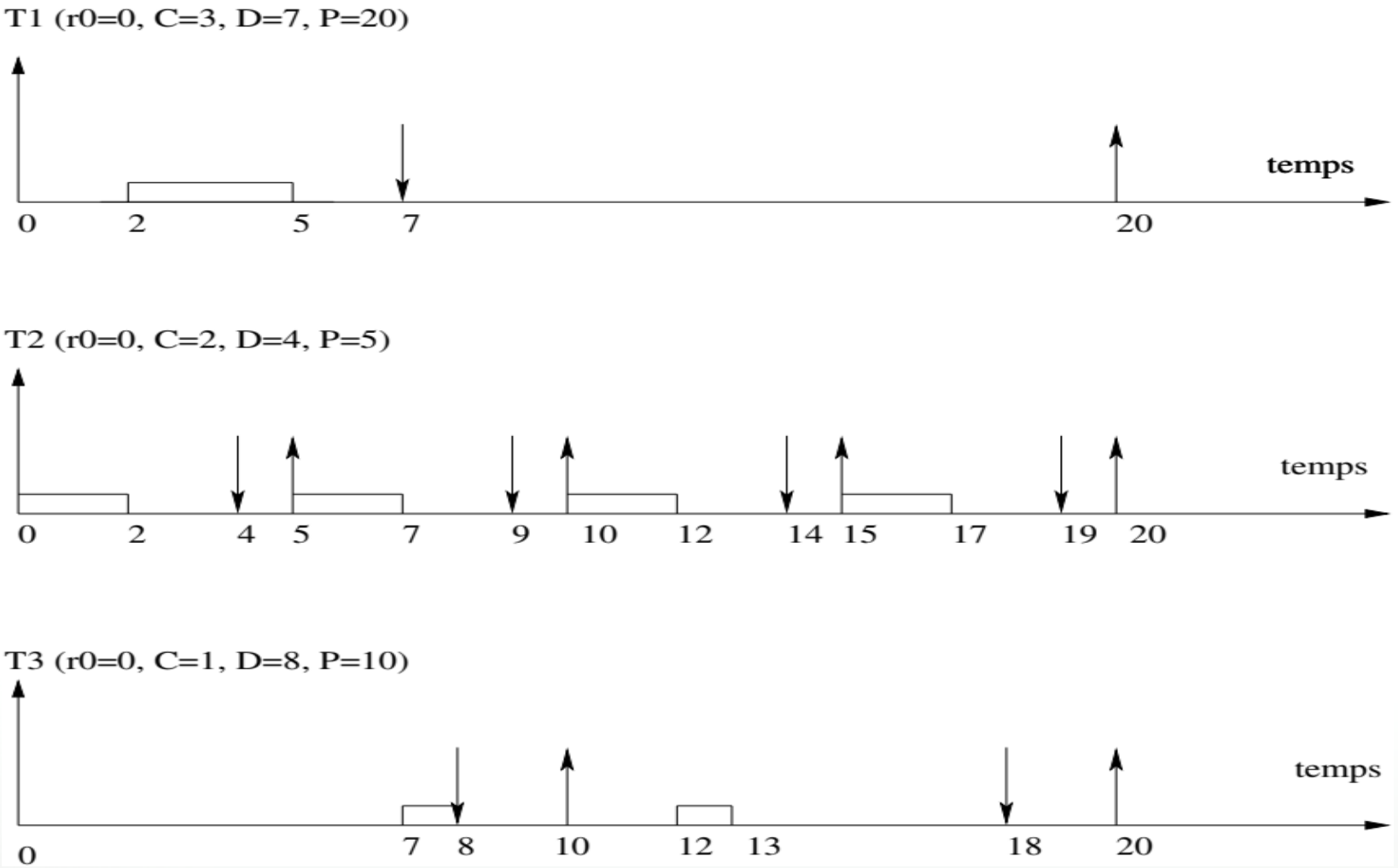
Les figures 11 et 12 montrent l'ordonnancement avec LL pour les 3 tâches de l'exemple sur ED, avec un calcul de laxité effectué aux seules dates de réveil.

- A t=0, les 3 tâches sont réveillées. Laxité dynamique de T1 =7-3=4.
  Laxité de T2 =4-2=2. Laxité de T3 =8-1=7. C'est donc T2 (la + prioritaire) qui s'exécute à t=0 (pour 2 unités).
- A t=2, c'est T1 qui s'exécute (sa laxité [7-2-3] est plus petite que celle de T3 [8-2-1]).
- A t=5, T2 se réveille de nouveau. Sa laxité dynamique
  (9-5-2 = 2ème échéance - temps courant - durée exécution)
  est égale à 2, celle de T3 (8-5-1) est de 2 également (retard 'toléré' tel que l'échéance soit toujours respectée). On peut choisir T3 (cas a) ou T2 (cas b) pour être ordonnancée et s'exécuter.

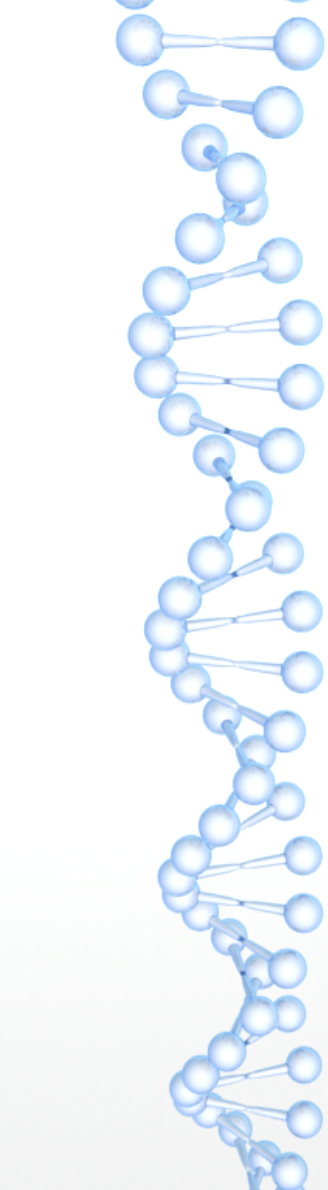
Rappel : L(t) = D (t) – C (t) laxité nominale résiduelle, retard maximum pour reprendre l'exécution d'une tâche si elle s'exécute seule.



Cas (a) : T3 puis T2



Cas (b) : T2 puis T3



Tâches périodiques et tâches apériodiques

Ce sont des algorithmes qui prennent en compte les tâches apériodiques.

Objectifs :

- les CT des tâches périodiques doivent être respectées et
- 1. soit minimiser le temps de réponse des tâches apériodiques à contraintes relatives (Best Effort),
- 2. soit maximiser le nombre de tâches apériodiques à contraintes strictes qui respectent leurs CT .

Tâches apériodiques à contraintes relatives :

1. Traitement en arrière-plan (BackgroundProcessing)

Principe : les tâches apériodiques sont ordonnancées lorsque le processeur est oisif (pas de tâche périodique prête).

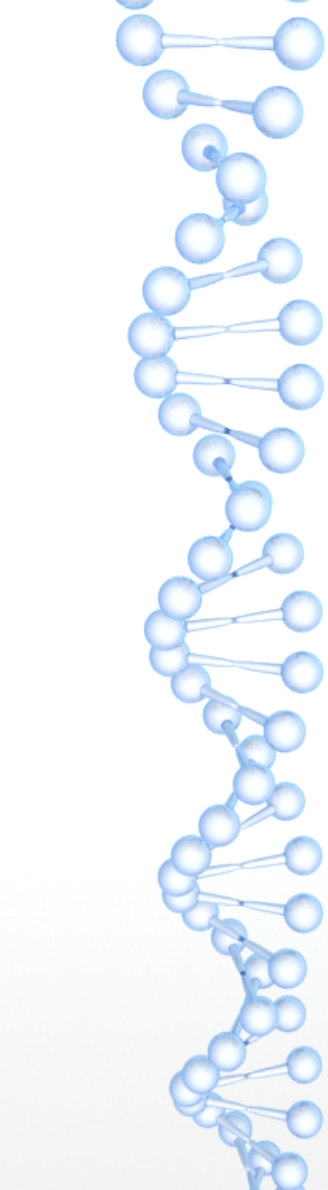
S'il y a plusieurs tâches apériodiques, les tâches sont servies par dates de réveil (FIFO).

Ordonnancement “en arrière plan”

La figure montre un tel ordonnancement sur un intervalle égale à 2 fois la période d’étude d’une configuration de 2 tâches périodiques à échéances sur requête : T1 (r0 = 0, C = 2, P = 5), T2 (r0 = 0, C = 2, P = 10).

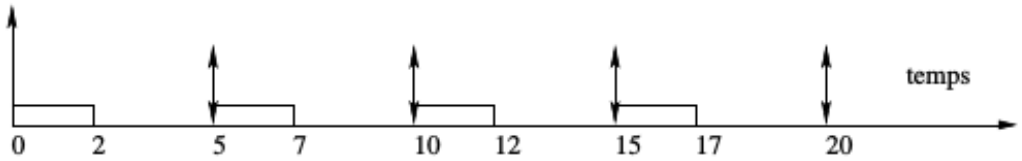
L’application de ’Rate Monotonic’ à cette configuration laisse le processeur oisif sur les intervalles [4,5], [7,10], [14,15] et [17,20].

La tâche apériodique Ta3 (r = 4, C = 2) survenant à t = 4 peut immédiatement commencer son exécution qu’elle finit sur le temps creux suivant, c-à-d entre les temps t = 7 et t = 8.
La tâche apériodique Ta4 (r=10, C=1) qui survient à t=10 doit attendre le temps creux [14,15] pour s’exécuter. La tâche apériodique Ta5 (r=11, C=2) doit attendre le temps creux [17,20] pour s’exécuter.

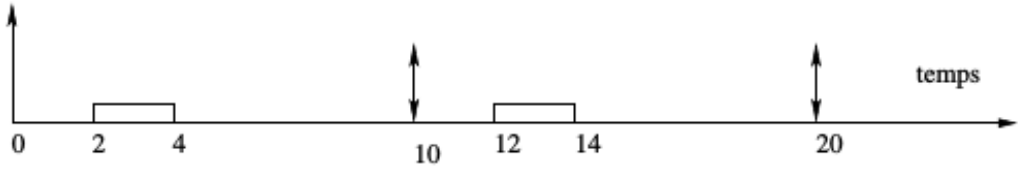


Tâches apériodiques à contraintes relatives

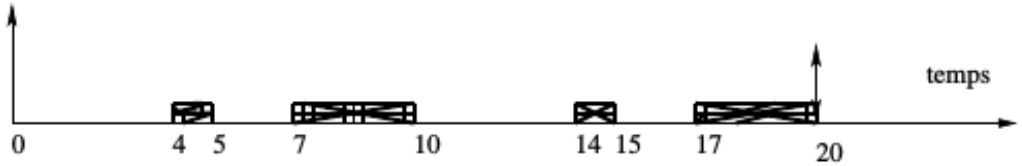
T1 (r0=0, C=2, P=5)



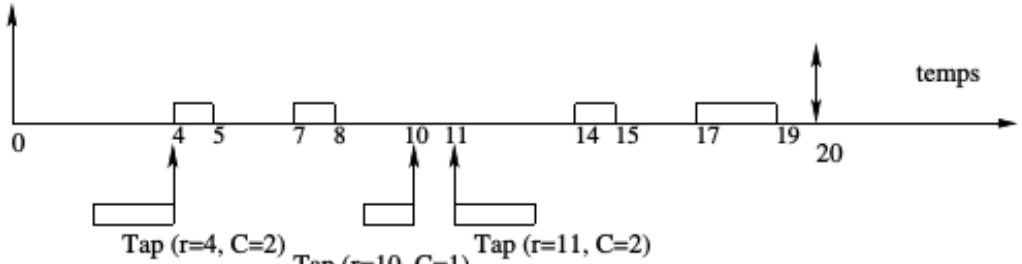
T2 (r0=0, C=2, P=10)

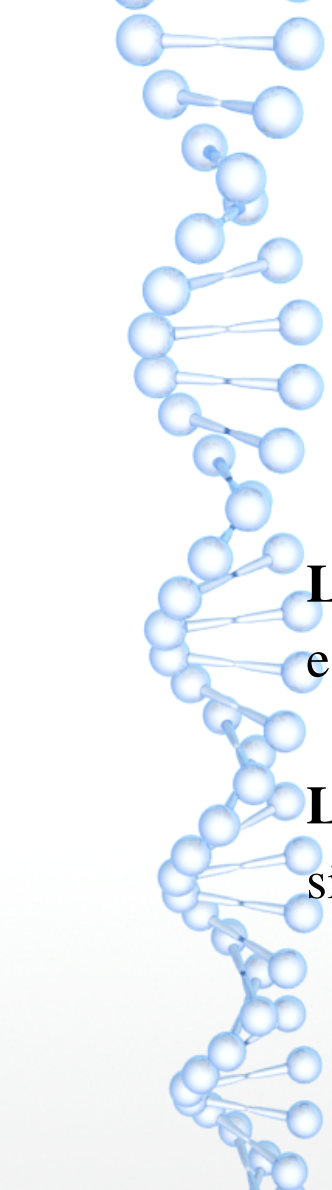


Temps creux



Taches apériodiques





Remarque

Les + : la méthode d'ordonnancement en arrière-plan est simple et peu coûteuse et est applicable ∀ l'algorithme d'ordonnancement des tâches périodiques.

Les - : les temps de réponse des tâches apériodiques peuvent être mauvais, surtout si la charge des tâches périodiques est importante.

Tâches apériodiques à contraintes relatives :

2. Les serveurs de tâches

un serveur est **une tâche périodique** créée spécialement pour veiller à ordonnancer les tâches apériodiques. Le serveur est caractérisé par une période et un temps d'exécution (la capacité du serveur).

Cette tâche-serveur souvent ordonnancée avec le même algo que les autres tâches périodiques.

Une fois active, la tâche-serveur sert les tâches apériodiques dans la limite de sa capacité. L'ordre de service des tâches apériodiques ne dépend pas de l'algo d'ordonnancement des tâches périodiques (il peut être FIFO ou f(échéances) ou f(temps d'exécution), ...).

Il existe plusieurs types de serveurs : le plus simple est le serveur par scrutation.

Les serveurs de tâches : Traitement par 'scrutation' ("Polling")

- à chaque activation, le serveur traite les tâches apériodiques en attente depuis son activation précédente, jusqu'à épuisement de sa capacité (ou plus de tâches en attente).
- Si, lors d'une nouvelle activation, il n'y a aucune tâche apériodique en attente, le serveur se suspend jusqu'à la prochaine occurrence : sa capacité (tps d'exécution) est récupérée par les tâches périodiques.
- La figure montre le fonctionnement d'un serveur par scrutation.

Configuration : 2 tâches périodiques à échéances sur requête :
T1 (r0 =0, C=3, P=20) et T2 (r0 =0, C=2, P=10).

- La tâche serveur est Ts (r0 =0, C=2, P=5) est de plus grande priorité (car plus petite période) avec "Rate Monotonic" appliqué pour ordonnancer les 3 tâches. On vérifie que la condition d'acceptabilité est remplie :

$3/20 + 2/10 + 2/5 = 0.75 < 077$, c-à-d somme$(C_i/P_i) < 2\,(2^{1/3}-1)$

Chronogramme : Ordonnancement “avec un serveur par scrutation”

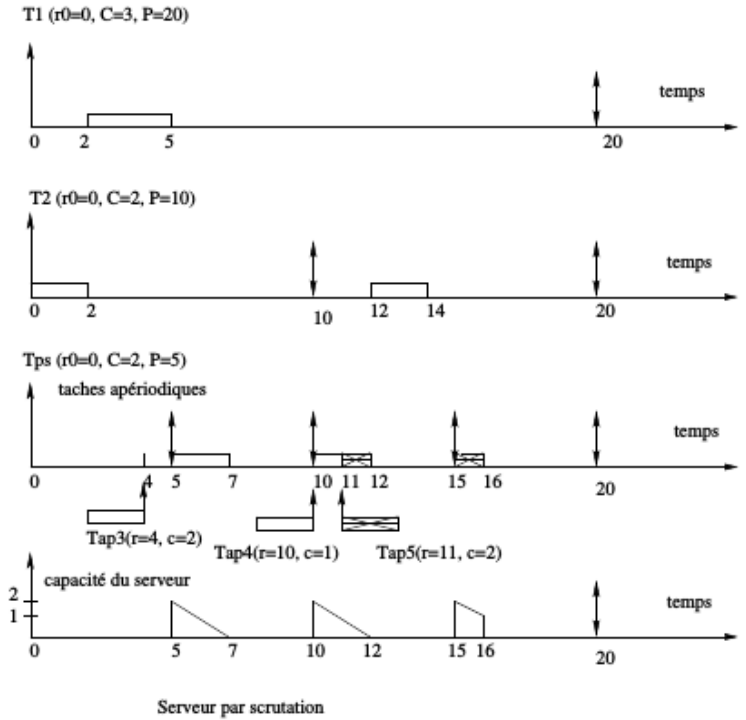


FIGURE: Ordonnancement “avec un serveur par scrutation”

Commentaires : avec “un serveur par scrutation”

- à t=0, aucune tâche apériodique, donc le serveur se suspend immédiatement et perd sa capacité. C’est T2, de priorité élevée suivante, qui s’exécute, puis c’est T1 .
- à t=4, la tâche apériodique Ta3 survient. elle doit attendre l’occurrence suivante du serveur (t=5) pour s’exécuter. Elle épuise alors toute la capacité (C=2) du serveur.
- à t=10, le serveur se réveille de nouveau, il sert la tâche apériodique Ta4 qui vient d’être activée. Puis il sert la tâche apériodique Ta5 . La tâche Ta5 ne peut pas s’exécuter entièrement car elle épuise la capacité restante du serveur. Elle doit attendre l’occurrence suivante du serveur à t=15 pour finir son exécution. Il reste encore une unité au serveur, il la perd car il n’y a aucune tâche apériodique à servir.

Commentaires : avec “un serveur par scrutation” (suite)

Faiblesses : lorsque le serveur est prêt, s’il n’y a pas de tâches apériodiques à traiter, sa capacité est perdue. Si des tâches apériodiques surviennent ensuite, elles doivent attendre l’occurrence suivante du serveur.

Les serveurs de tâches : Serveur sporadique

Il améliore le temps de réponse des tâches apériodiques, sans diminuer le taux d'utilisation du processeur des tâches périodiques.

Ce serveur est également une tâche périodique de plus haute priorité, qui **conserve sa capacité** de traitement si, à son réveil, il n'y a pas de tâche apériodique à servir.

Commentaires : “Serveur sporadique”

La figue montre le fonctionnement d’un serveur sporadique sur le même ensemble de tâches que précédemment : T1 (r0 =0, C=3, P=20) et T2 (r0 =0, C=2, P=10).
La tâche serveur est Ts (r0 =0, C=2, P=5).

- à t=0, le serveur se réveille, mais se suspend, car il n’a pas de tâches apériodiques à traiter (mais il conserve sa capacité, 2).
- à t=4, la tâche apériodique Tap3 survient. Elle est servie immédiatement entre t=4 et t=6 et consomme toute sa capacité (=2). Comme le serveur s’est exécuté, une date de réinitialisation de sa capacité est calculée : ce sera t=4+5 = 9. La valeur de réinitialisation est 2 (sa capacité).

Commentaires : “Serveur sporadique” (suite)

- à t=9, le serveur retrouve toute sa capacité, mais il se suspend car il n’y a pas de tâche apériodiques en attente.
- à t=10, la tâche Tap4 arrive et s’exécute tout de suite pendant une unité (C=1). Puis Tap5 arrive et commence à s’exécuter pendant une unité et elle est suspendue jusqu’à nouvelle occurrence du serveur (car sa capacité est épuisée). Une nouvelle date de réinitialisation de la capacité est calculée : t=10+5. La tâche Tap5 poursuivra son exécution pour une unité quand le serveur aura récupéré sa capacité à t=15. Il reste au serveur une unité qu’il récupérera à t=20.

Chronogramme : Ordonnancement “avec un serveur sporadique”

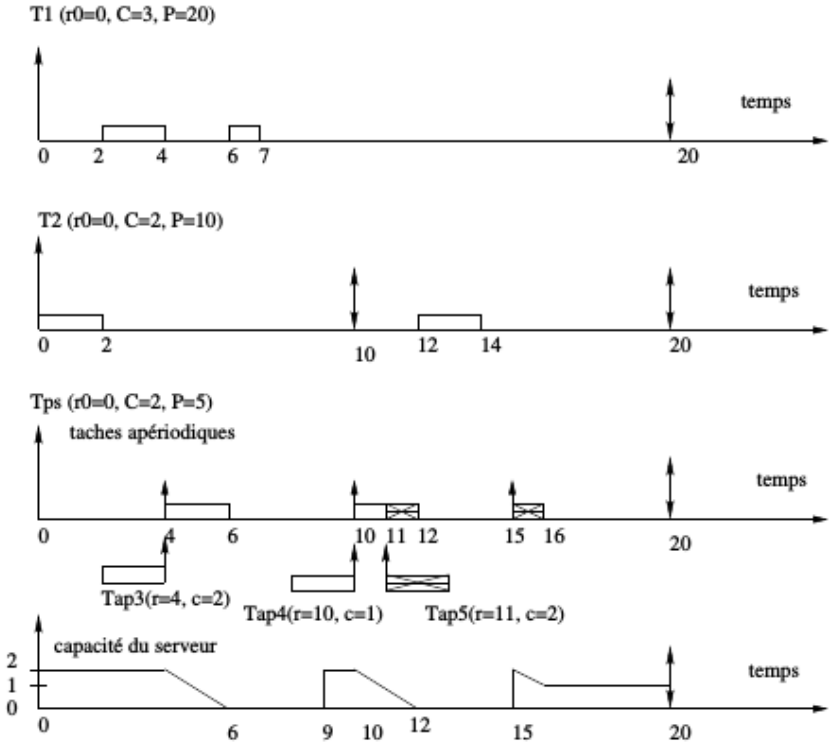
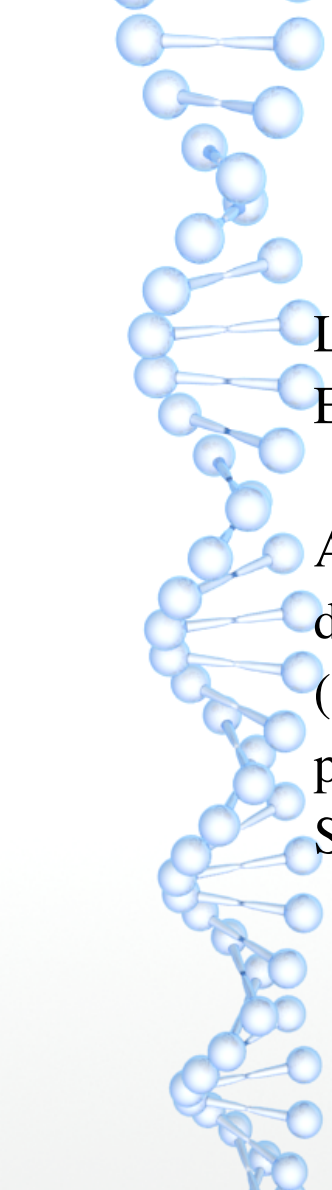


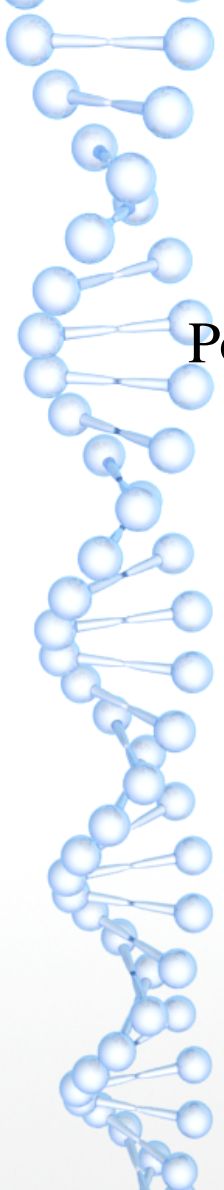
FIGURE: Ordonnancement “avec un serveur sporadique”



Tâches apériodiques à contraintes strictes

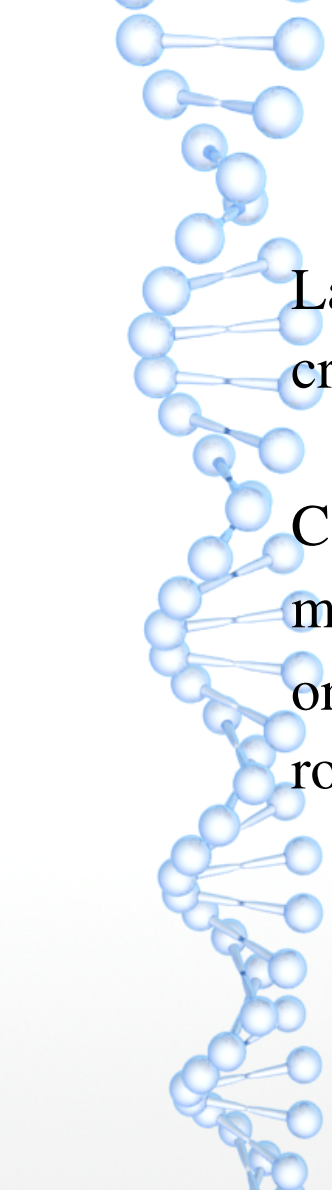
Les tâches apériodiques sont traitées sans hypothèse sur leur rythme d'arrivée. Elles sont ordonnancées selon "Earliest Deadline".

A chaque arrivée d'une tâche apériodique, une "routine de garantie" teste dynamiquement si la nouvelle tâche peut s'exécuter en respectant (1) ses CT, (2) celles des tâches périodiques et (3) celles des autres tâches apériodiques précédemment acceptées mais non terminées :
Si le test est positif, la tâche est acceptée.



Méthode pour tâches apériodiques à contraintes strictes

Politiques d'Acceptation dans les temps creux d'une séquence rigide de tâches :



Tâches apériodiques à contraintes strictes

La méthode consiste à ordonnancer les tâches apériodiques dans les temps creux de la séquence d'ordonnancement "ED" des tâches périodiques.

C'est une politique proche de celle en 'arrière-plan' vue précédemment, mais ici les tâches apériodiques ont **des CT** à respecter et sont elles-mêmes ordonnancées selon "ED" : à chaque réveil d'une tâche apériodique, une routine de garantie est exécutée.

Routine de garantie :

1) La routine teste l'existence d'un temps creux suffisant pour l'exécution de la tâche apériodique entre sa date de réveil et son échéance.

2) **Si ce temps creux existe**, alors elle vérifie que l'acceptation de la nouvelle tâche ne remet pas en cause le respect des CT de tâches apériodiques précédemment acceptées, non encore achevées (cette vérification ne concerne que les tâches apériodiques dont les échéances sont ≥ à celle de la nouvelle tâche).

3) Si au moins une des condition 1 ou 2 n'est pas satisfaite, alors la tâche est rejetée, sinon elle est ajoutée à l'ensemble des tâches apériodiques acceptées pour être exécutée selon son échéance.

Exemple

La figure 9 montre que la configuration périodique “ED” présente, durant la période d’étude, trois intervalles de creux : [8,10], [13,15] et [17,20]. Les 3 tâches apériodiques Ta4 , Ta5 et Ta6 peuvent être acceptées dans ces temps creux :

Déroulement

- à t=4, Ta4 se réveille. Le temps creux existant entre sa date de réveil et son échéance est l'intervalle [8,10] = 2 unités, juste suffisantes pour exécuter la tâche. Ta4 dispose donc d'assez de temps pour s'exécuter dans le respect de ses CT, et comme il n'y a pas d'autre tâches apériodiques précédemment acceptées mais non terminée, elle est acceptée.
- à t=10, Ta5 se réveille. Le temps creux existant entre sa date de réveil 10 et son échéance 18, est de 3 unités (tout l'intervalle [13,15] + 1 unité de l'intervalle [17,20]). Il y a donc d'assez de temps pour exécuter Ta5 qui a un temps d'exécution égal à 1. Et comme il n'y a pas de tâches apériodiques acceptées et non terminées (Ta4 s'était exécutée entre 8 et 10), alors Ta5 peut être acceptée.

Déroulement (suite)

- à t=11, Ta6 se réveille. Le temps creux existant entre sa date de réveil 11 et son échéance 16 est de 2 unités ([13,15]), donc juste suffisant pour l'exécuter dans le respect de son échéance. Mais, il faut vérifier que son acceptation ne remet pas en cause les CT des autres tâches apériodiques acceptées non terminées (et d'échéances plus éloignées que la sienne). Ici Ta5, qui n'a pas encore commencé son exécution et dont l'échéance est 18. L'ordre d'exécution devient Ta6 puis Ta5 et on constate que chacune peut s'exécuter dans le respect de ses CT (car C=2 pour Ta6 et C=1 pour Ta5).

Ordonnancement dans les temps creux : Routine de garantie

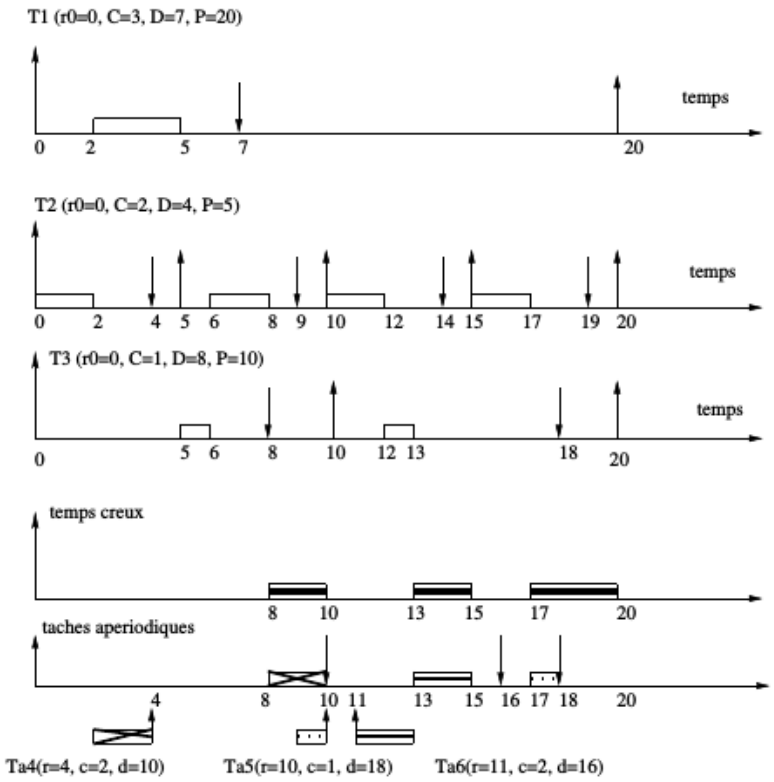
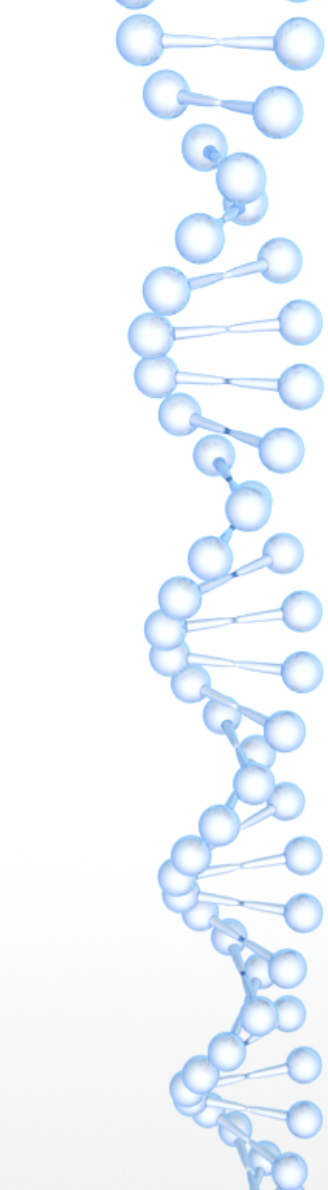


Figure: Ordonnancement “ dans les temps creux”



Ordonnancement de tâches dépendantes

A. Précédence statique : on peut tracer le graphe de précédence (une tâche attend la fin d'exécution d'une ou plusieurs autres tâches), car elle a besoin de ses (leurs) résultat(s).

- Ti → Tj, si Tj doit attendre la fin de l'exécution de Ti pour commencer son exécution.

- Si Ti précède Tj alors {(1) $r_{Tj} \geq r_{Ti}$, r étant la date de réveil,
  (2) si Ti périodique alors Tj l'est également et
  (3) priorité(Ti) > priorité (Tj)}

Exemple : Caméra pour la reconnaissance et la vérification

T1 : acquisition

T2: pré-traitement

T3, T4 : extraction de caractéristiques, détection de contours

T5 : estimation de la hauteur

T6 : Reconnaissance finale

Graphe de précédence de tâches

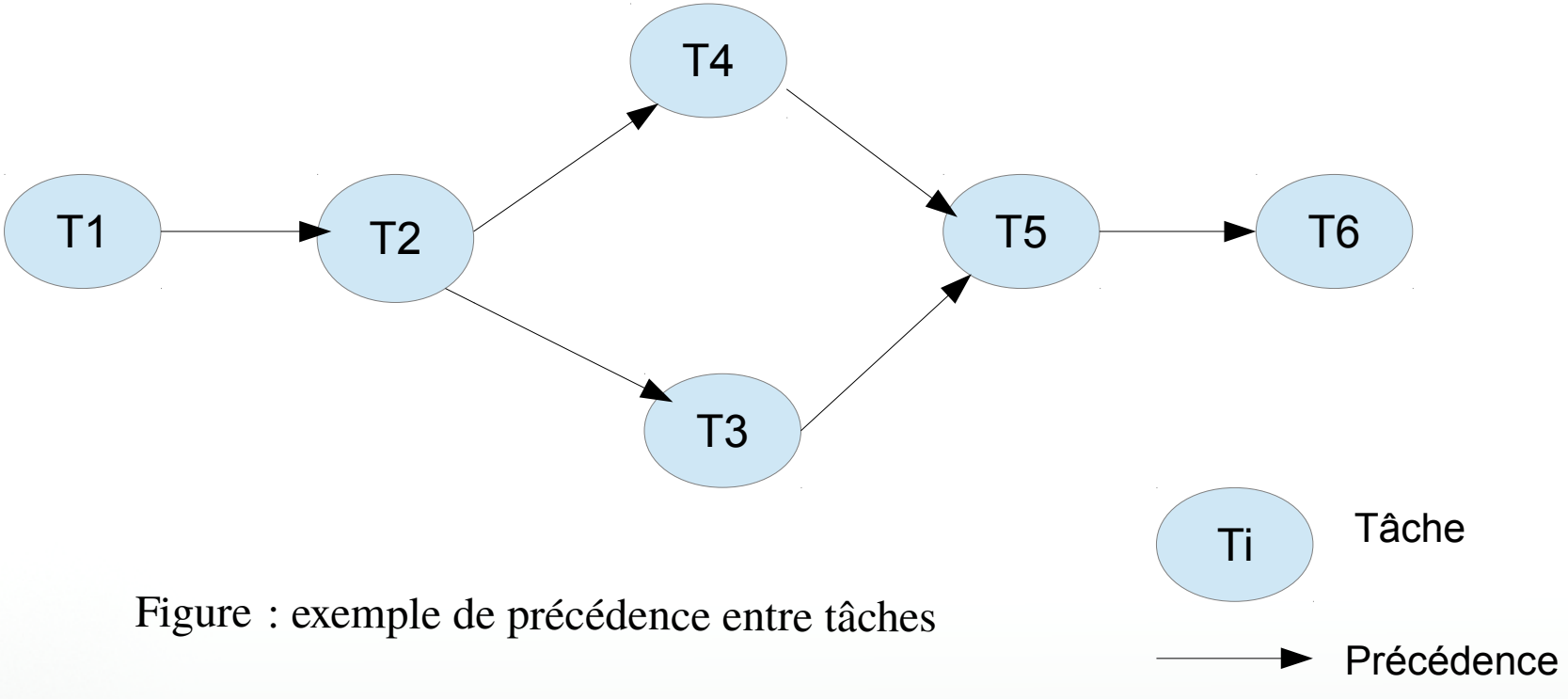
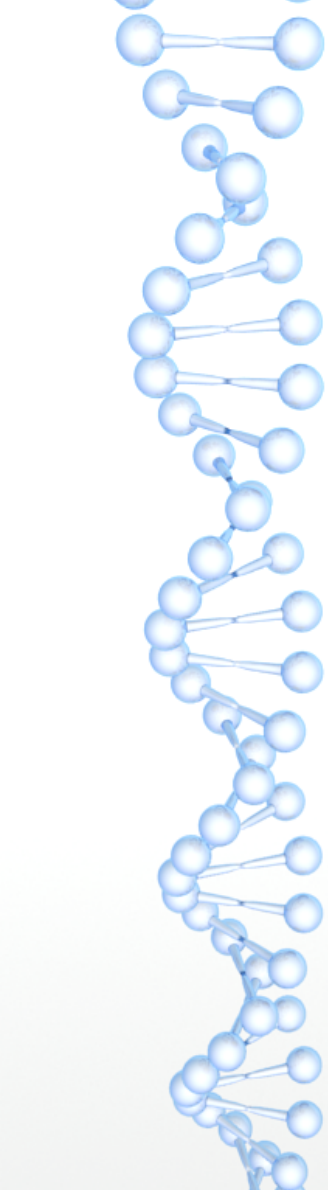


Figure : exemple de précédence entre tâches



Commentaires

Tx → Ty : Tx est un prédécesseur direct de Ty . Ex. T1 → T2

Tx < Ty : Tx est un prédécesseur direct ou indirect de Ty . Ex. T1 <T 4

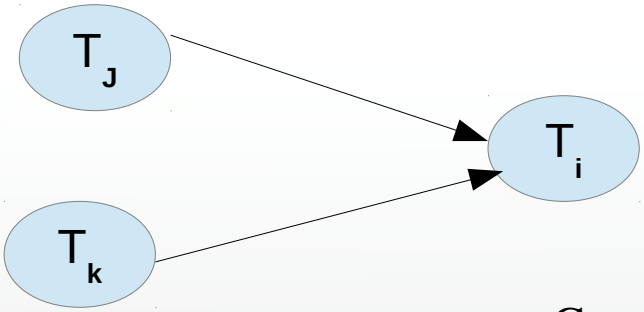
Contraintes de précédence et 'Rate Monotonic'

- Idée : Une tâche ne doit pas commencer avant son prédécesseur, et ne doit préempter son successeur.

- dans le cas de l'ordonnancement selon "RM", on transforme hors ligne l'ensemble des tâches liées par une contrainte de précédence en un ensemble de tâches indépendantes pour lesquelles le test d'acceptabilité de "RM" est valide, en faisant :

  - Calcul des dates de réveil : $r_i^* = \max\{r_i , r^*_{pred}\}$ avec r^*_{pred} date de réveil de la tâche précédant la tache i, et
  - Si T_i et T_j ont même période avec T_i qui précède T_j
    Alors priorité (T_i) > priorité (T_j)

Précédence	C_i^*	r_i^*	P_i^*
$\varnothing \rightarrow T_i^*$	C_i	r_i	P_i
$T_J^* \rightarrow T_i^*$	C_i	$\max(r_i, r_J^*)$	P_i et > P_J^*
$T_J^* \rightarrow T_i^*$ $T_k^* \rightarrow T_i^*$	C_i	$\max(r_i, r_J^*, r_k^*)$	P_i et $\max(P_J^*, P_k^*)$



Graphe 3 noeuds

Exemple/Exercice : précédence avec RM

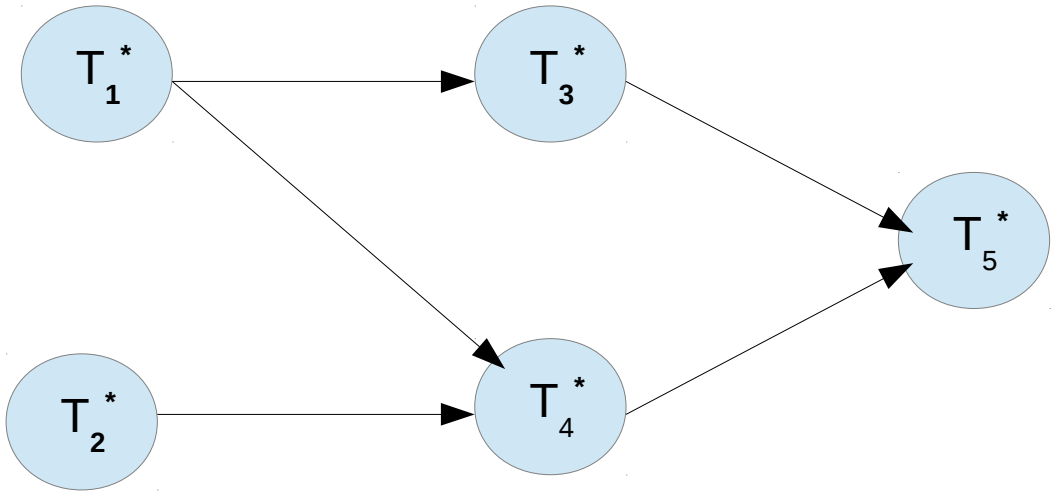


Figure : précédence entre 5 tâches

tâche	C_i	r_i	P_i	r_i^*
T_1^*	1	0	12	$r_1^* = r_1 = 0$
T_2^*	2	5	14	$r_2^* = r_2 = 5$
T_3^*	2	0	10	$r_3^* = \max(r_3, r_1^*) = \max(0,0) = 0$
T_4^*	1	0	8	$r_4^* = \max(r_4, r_1^*, r_2^*) = \max(0,0,5) = 5$
T_5^*	3	0	16	$r_5^* = \max(r_5, r_3^*, r_4^*) = \max(0,0,5) = 5$

Calcul des r_i^* (nouvelles dates de réveil des tâches)

tâche	C_i	r_i	P_i	P_i^*
T_1^*	1	0	12	$P_1^* = P_1 \rightarrow 12$
T_2^*	2	5	14	$P_2^* = P_2 \rightarrow 14$
T_3^*	2	0	10	$P_3^* = (P_3=10)$ et $> (P_1^* = 12) \rightarrow 13$
T_4^*	1	0	8	$P_4^* = (P_4=8)$ et $> \max(P_1^* = 12), P_2^* = 14) \rightarrow 15$
T_5^*	3	0	16	$P_5^* = (P_5=16)$ et $> \max(P_3^* = 13), P_4^* = 15) \rightarrow 16$

Calcul des P_i^* (nouvelles périodes des tâches)

On vérifie qu'il y a conservation des conditions après transformation :

Somme $(C_i/P_i) \leq ? \; n \; (2^{1/n} - 1)$ avec $n = 5$

$1/12 + 2/14 + 2/10 + 1/8 + 3/16 \sim 0.738$

$5 \; (2^{1/5} - 1) \sim 0.743$

On a : $0.738 \; < \; 0.743$

Somme $(C_i^*/P_i^*) \leq ? \; n \; (2^{1/n} - 1)$ avec $n = 5$

$1/12 + 2/14 + 2/13 + 1/15 + 3/16 \sim 0.634$

$5 \; (2^{1/5} - 1) \sim 0.743$

On a également : $0.634 \; < \; 0.743$

Contraintes de précédence et 'Earliest Deadline'

La prise en compte de la précédence est réalisée en modifiant r_i, d_i, r_i^*, d_i^* selon les formules suivantes :

- $r_i^* = \max \{r_i, \max(r^*_{pred} + C_{pred}) \}$, c-à-d la date de début d'une tâche doit être supérieure à toutes les dates de début de ses prédécesseurs augmentée de leurs durées d'exécution.

- $d_i^* = \min \{d_i, \min(d^*_{succ} - C_{succ}) \}$, c-à-d pour une instance donnée, l'échéance d'une tâche doit être inférieure à toutes les échéances de ses successeurs diminuées de leurs temps d'exécution.

C_i^* =	prec	r_i^* =	succ	D_i^*=
C_i	$\varnothing \rightarrow T_i$	r_i	$T_i \rightarrow \varnothing$	D_i
C_i	$T_j \rightarrow T_i$	$\max (r_i, r^*_j + C_j)$	$T_i \rightarrow T_j$	$\min \{D_i, D^*_j - C_j)$
C_i	$T_j \rightarrow T_i$ et $T_k \rightarrow T_i$	$\max (r_i, r^*_j + C_j, r^*_k + C_k)$	$T_i \rightarrow T_j$ et $T_i \rightarrow T_k$	$\min \{D_i, D^*_j - C_j, D^*_k - C_k)$

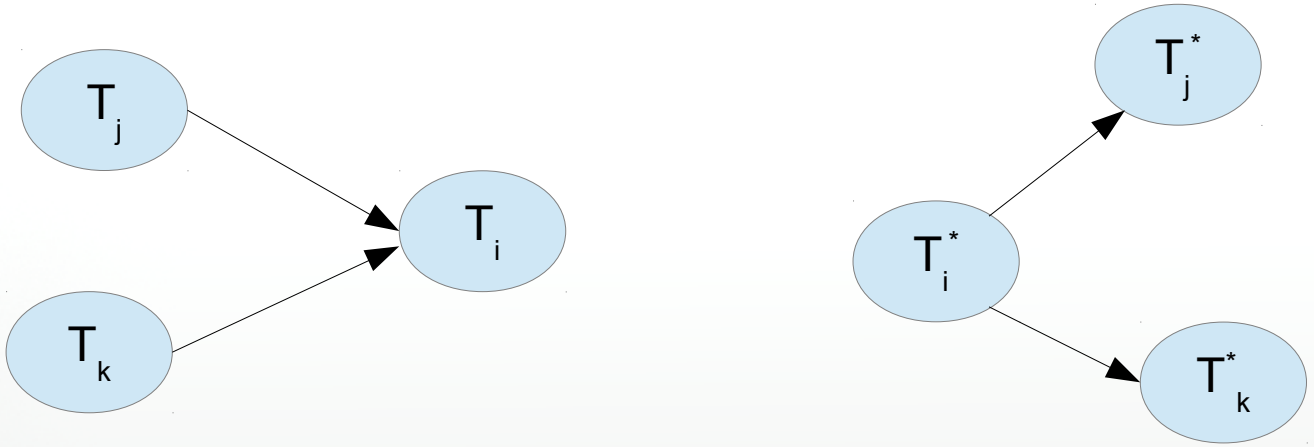


Figure : graphes 3 nœuds pour EDF

Exemple/Exercice : précédence avec EDF

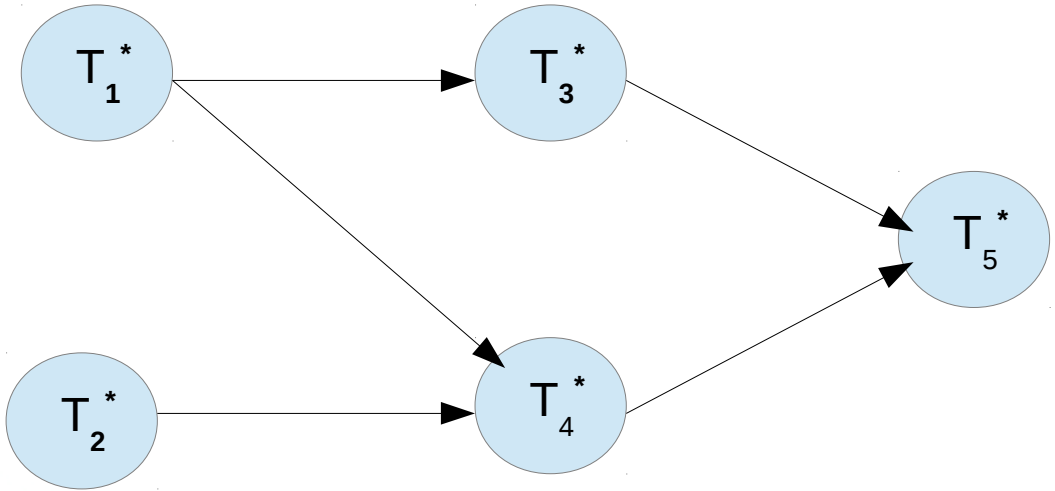


Figure : précédence entre 5 tâches

tâche	$c_i^* = c_i$	r_i	D_i	r_i^*
T_1^*	1	0	12	$r_1^* = r_1 = 0$
T_2^*	2	5	14	$r_2^* = r_2 = 5$
T_3^*	2	0	10	$r_3^* = \max(r_3, r^*_1 + C_1) = \max(0, 1) = 1$
T_4^*	1	0	8	$r_4^* = \max(r_4, r^*_1 + C_1, r^*_2 + C_2) = \max(0, 0+1, 5+2) = 7$
T_5^*	3	0	16	$r_5^* = \max(r_5, r^*_3 + C_3, r^*_4 + C_4) = \max(0, 1+2, 7+1) = 8$

Calcul des nouvelles dates de réveil r_i^*

Remarque : on commence par calculer r_1, puis r_2, puis r_3, etc.

(contrairement aux d_i^*)

Tâche	$C_i^* = C_i$	r_i	D_i	r_i^*	D_i^*
T_1^*	1	0	12	0	$D_1^* = \min \{D_1, D^*_3 - C_3, D^*_4 - C_4) = \min (12, 10-2, 8-1) = 7$
T_2^*	2	5	14	5	$D_2^* = \min \{D_2, D^*_4 - C_4) = \min (14, 8-1) = 7$
T_3^*	2	0	10	1	$D_3^* = \min \{D_3, D^*_5 - C_5) = \min (10, 16-3) = 10$
T_4^*	1	0	8	7	$D_4^* = \min \{D_4, D^*_5 - C_5) = \min (8, 16-3) = 8$
T_5^*	3	0	16	8	$D_5^* = D_5 = 16$

Remarque : On commence par calculer D_5, puis D_4, puis D3, etc.

La transformation conserve les conditions d'ordonnançabilité :

On a des tâches qui ne sont pas à échéance sur requête

=>

Avant transformation :

Somme $(C_i/D_i) \leq 1$?

On a : 1/12 + 2/14 + 2/10 + 1/8+ 3/16 = 0.738 < 1.

Après transformation :

Somme $(C_i^*/D_i^*) \leq 1$?

On a bien : 1/7 + 2/7 + 2/10 + 1/8 + 3/16 = 0.941 < 1.

B. Précédence dynamique (partage de ressources)

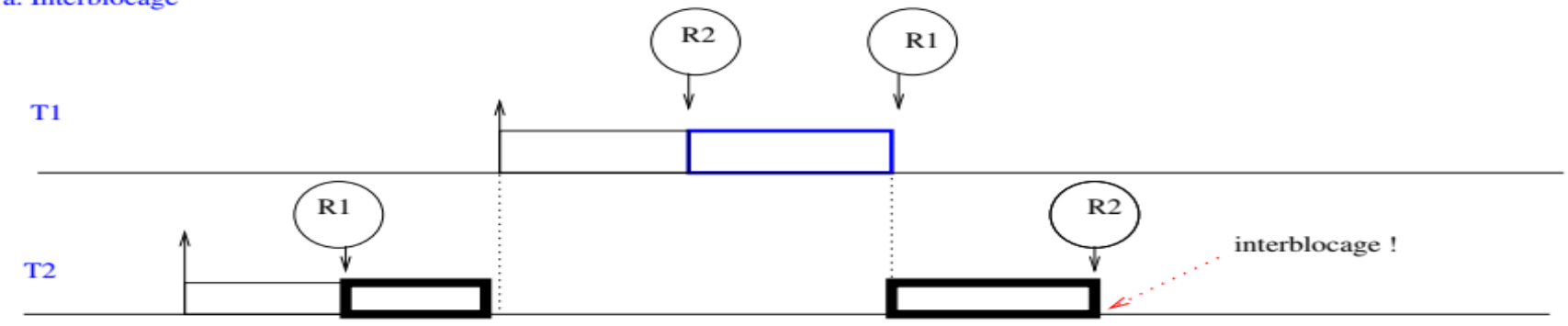
Les tâches partagent des ressources critiques (en exclusion mutuelle)
La précédence est dynamique car la tâche qui accède à la ressource critique en premier va s'exécuter., en exclusion mutuelle. L'autre doit attendre (donc l'ordre d'exécution n'est pas défini à l'avance).

Un des protocoles pour ordonnancer ce type de tâches est PIP (Priority Inheritance Protocol), aussi appelé PHP (Protocole d'Héritage de Priorité)

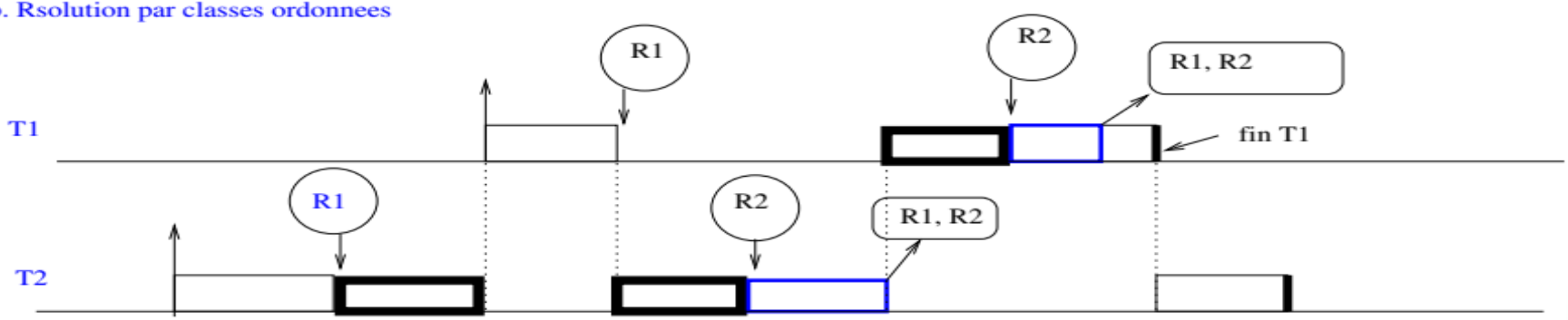
Principe du PIP : Si une tâche de forte priorité arrive et trouve une tâche de faible priorité (qui est dans sa section critique), la tâche de faible priorité hérite de la priorité de la tâche de haute priorité. Ainsi, elle peut continuer à s'exécuter et sortir plus vite de sa section critique.
Ce protocole ne prévient pas l'interblocage, que l'on peut résoudre avec les classes ordonnées (voir figure).

Problème de l'interblocage

a. Interblocage



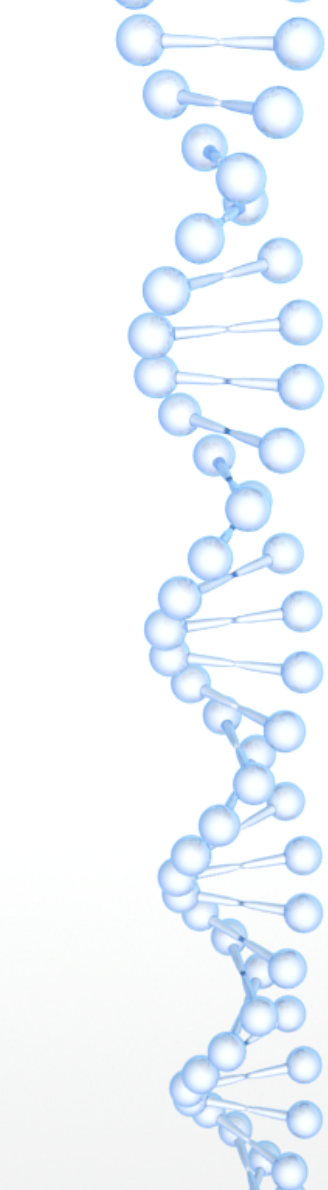
b. Rsolution par classes ordonnees



tache elue

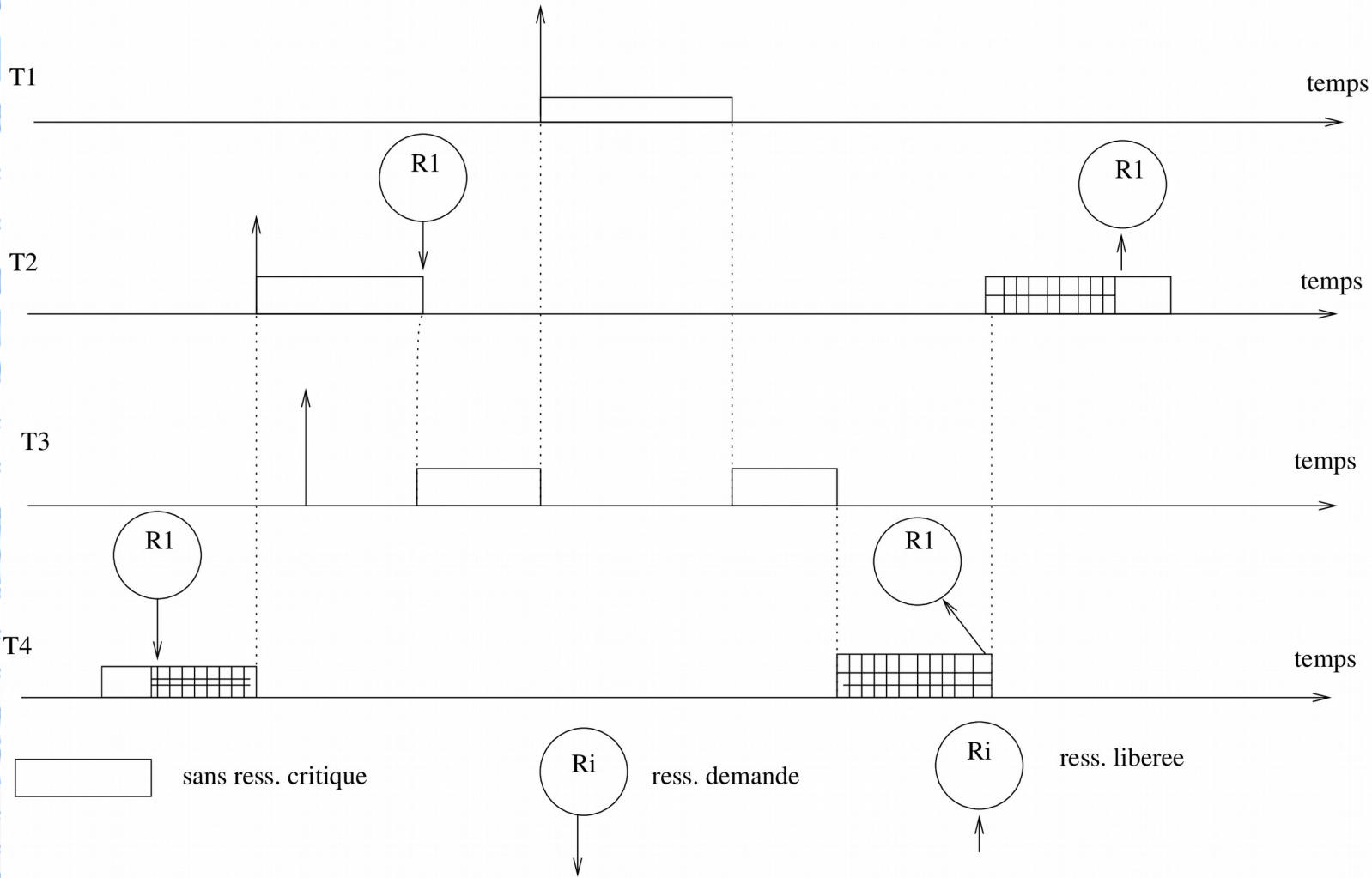
tache utilisant la ress. R1

tache utilisant les ress. R1 et R2



Un autre problème survient souvent dans le contexte temps réel : l'INVERSION DE PRIORITÉ (une tâche de faible priorité qui bloque une tâche de priorité plus élevée). Voir la figure suivante.
Ce problème est survenu au cours de la mission vers la planète MARS ; le robot *PathFinder* est bloqué à cause de ce problème. Il a fallu une intervention depuis la terre pour résoudre le problème (mais beaucoup de temps a été perdu pour trouver l'erreur!) :

On a : Pri(T1) > Pri (T2) > Pri(T3) > Pri(T4)

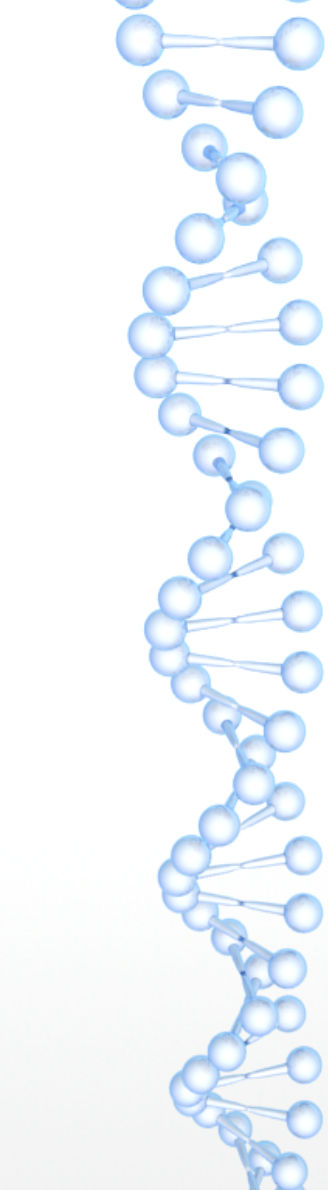


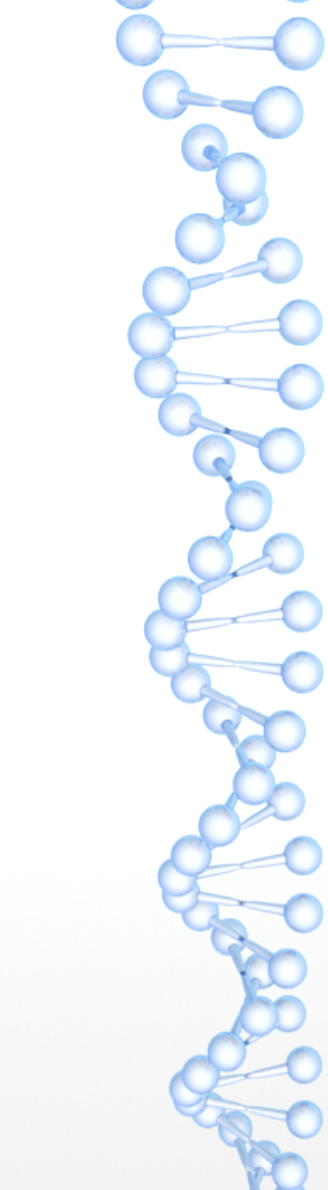
Commentaires : La tâche T2 est bloquée :

- par la tâche T1 : c'est normal, car Prio(T1) > Prio(T2) et
- par la tâche T4 : c'est normal, car T4 détient une ressource critique que T2 voudrait utiliser, et
- par la tâche T3 : **CE N'EST PAS NORMAL** ! Car Prio(T3) < Prio(T2) et (T3 et T2) ne partagent pas de ressources critique.

⇒ ce phénomène est appelé INVERSION DE PRIORITÉ.

Remarque : le protocole précédent PIP résout le problème d'inversion de priorité.



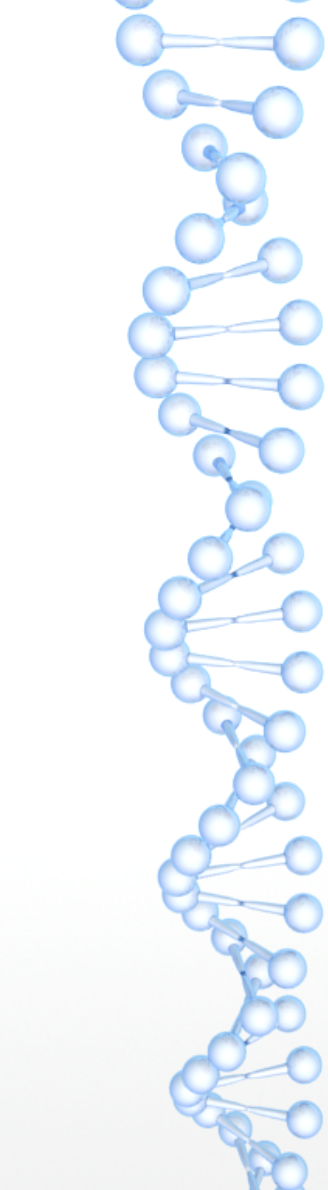


Situations de surcharge

Quelques éléments :

quand la charge du processeur devient telle qu'il est impossible de respecter toutes les échéances
Ex. EDF et RM –> mauvaises performances en situations de surcharge

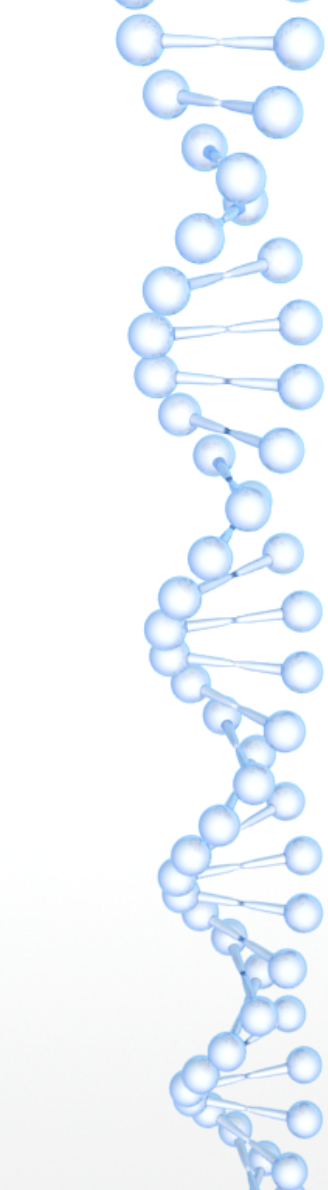
Pour éviter ces fautes temporelles => plusieurs politiques, qui se basent sur des modèles de tâches plus élaborés



A. Tolérance aux surcharges pour tâches périodiques

1. Méthode du mécanisme à échéance

2. Méthode du calcul approché

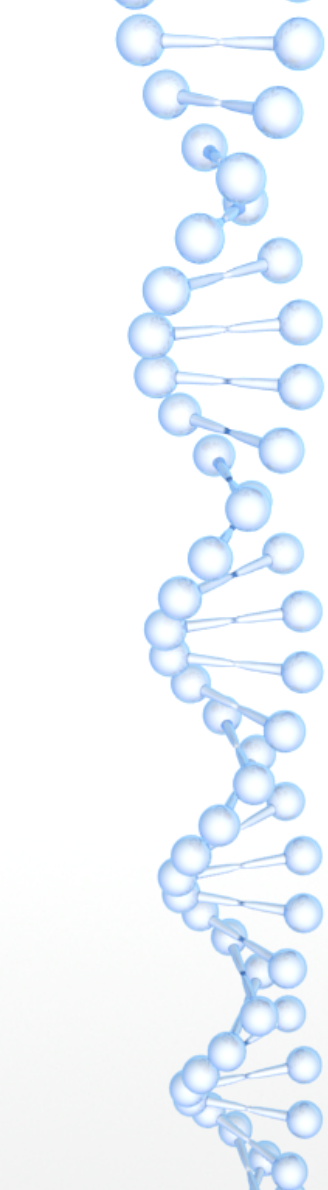


1. Méthode du mécanisme à échéance

Chaque tâche possède une version primaire et une version secondaire

La version primaire : fournit un résultat avec une bonne QdS mais au bout d'un temps indéterminé.

La version secondaire : fournit un résultat acceptable en un temps borné (connu à l'initialisation)



1. Méthode du mécanisme à échéance

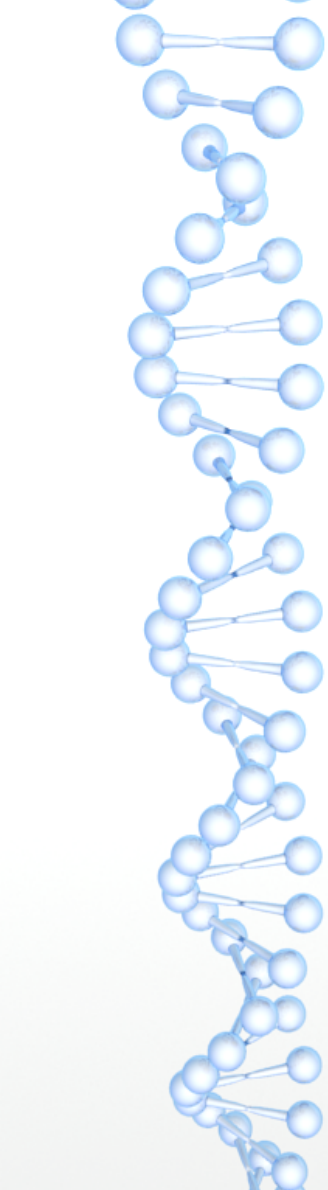
L'algorithme de tolérance aux fautes :

- doit assurer le respect des échéances, soit par le primaire, soit par le secondaire.

- Si le primaire et le secondaire sont exécutés => le résultat du primaire est utilisé.

- l'ordonnancement consiste en la juxtaposition d'une séquence des primaires et d'une séquence des secondaires et d'une règle de décision pour commuter de l'une à l'autre.

- Il existe 2 politiques : 1ère chance et 2ème chance



Tâches périodiques-Mécanisme à échéance :

- Politique 1ère chance :

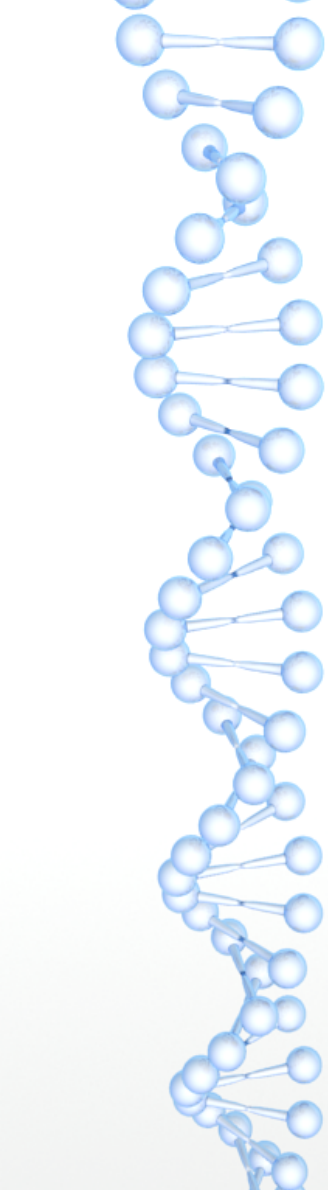
Si Priorité(Secondaires) > priorité (Primaires)

=> Primaires ordonnancées dans les temps creux des secondaires associées

- Politique 2ème chance :

Si Priorité(Primaires) > priorité (Secondaires)

=> Primaires exécutées avant leurs secondaires associées, lesquelles sont ordonnancées plus tard.
Si l'exécution de la primaire réussit, la secondaire n'est pas exécutée.

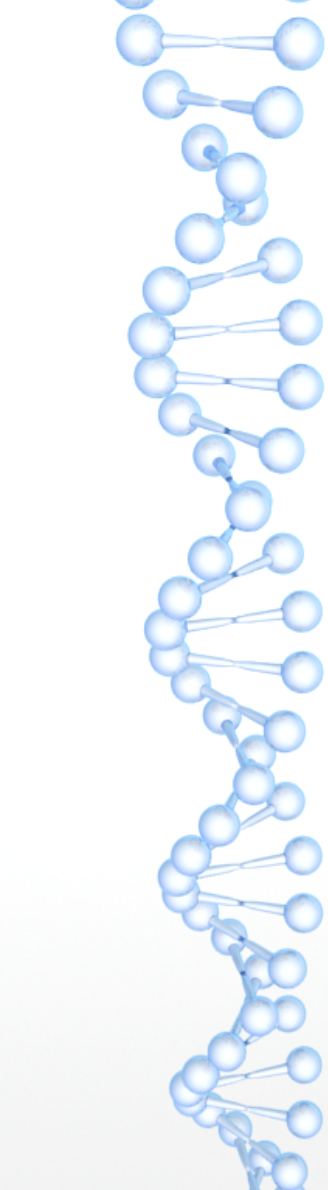


2. Méthode du calcul approché

Chaque tâche = partie obligatoire + partie optionnelle

Partie obligatoire : fournit un résultat approché et doit s'exécuter avant l'échéance.

Partie optionnelle : affine le résultat et s'exécute s'il reste assez de temps avant l'échéance.



B. Éléments sur la tolérance aux surcharges pour tâches quelconques

En plus de l'échéance, ces politiques prennent en compte le critère d'importance d'une tâche, qui définit son caractère primordial dans l'application => 2 tâches de même échéance peuvent avoir des importances différentes.

Ordonnancement à importance : à chaque réveil d'une tâche (périodique ou non), un test de garantie définit si la nouvelle tâche engendre une faute temporelle ou non.

Ce test est basé sur la laxité dynamique du système LP(t), qui devient négative si une surcharge est détectée.