La gestion de tâches est faite par l'ordonnanceur (Scheduler) du système d'exploitation. Il permet d'allouer le processeur à chaque tâche.

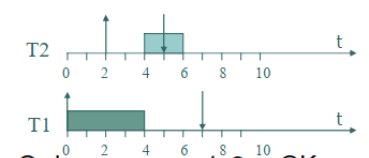
**Contraintes de temps**

* Choix de l’ordre d’exécution des fonctions important : Ordonnancement des calculs
* Déterminisme d’exécution (predictability) : On doit connaître pour tous les calculs effectués (application, OS) leur temps de calcul de manière sûre (non sous-estimée)
* Validation des contraintes de temps à partir de la charge de travail : Tests pas toujours suffisants (exhaustivité). Modèle du système et de la charge : analyse d’ordonnançabilité

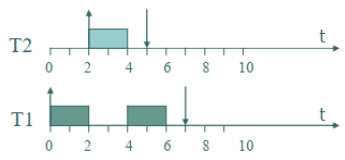
**Exemple d’ordonnancement**

Tâche T1 : arrivée en 0, durée 4, échéance 7 | Tâche T2 : arrivée en 2, durée 2, échéance 5

Ordonnancement 1 : premier arrivé - premier servi. Pas d’interruption de tâche.

 T2 Rate son échéance

Ordonnancement 2 : priorité. T2 plus prioritaire que T1



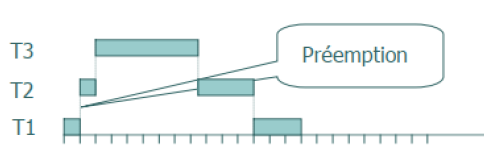
**Types d'ordonnanceurs**

**Préemptif** (avec réquisition) : L'ensemble des Unix et Windows sont des systèmes de ce type. Le processus courant peut être suspendu par décision de l'ordonnanceur (e.g. suite à une interruption).

**Non-préemptif**: Une commutation de contexte ne peut avoir lieu que lorsque le système d'exploitation est appelé. Une tâche qui s'exécute doit demander explicitement l'intervention de l'ordonnanceur.

*Exemple d'ordonnancement préemptif : conduit par la priorité*

* Algorithme d'ordonnancement : sélectionner la tâche la plus prioritaire
* Prio (T3) > Prio (T2) > Prio (T1) (ici, priorités fixes)
* T1, T2 et T3 arrivent respectivement aux dates 1, 2 et 3



**Principe** : Planifie l'exécution des tâches, en fonction de l'état de l'environnement, de telle sorte que les échéances temporelles soient respectées.

**Mécanisme** : A chaque instant où l'ordonnanceur est activé :

* Examine tous les événements arrivés et l'état du système.
* Exécute l'algorithme d'ordonnancement mettant en œuvre la stratégie de partage des ressources,
* Retire les processeurs aux tâches en activité qui ne sont plus prioritaires
* Sauvegarde leur contexte
* (Re)lance les tâches prioritaires

**Quels algorithmes d'ordonnancement ?**

Problème complexe sans solution dans le cas général

**Hypothèses simplificatrices :**

* Monoprocesseur
* Tâches indépendantes
* Le temps pris par l'ordonnanceur lui-même est négligé

**Plusieurs algorithmes :**

* Ordonnancement cyclique (les tâches sont exécutées dans un ordre défini à l'avance – pas de préemption)
* Ordonnancement par priorité (le problème revient alors à choisir les priorités)
* « Rate monotonic » : ordo par priorité où la priorité est inversement proportionnelle à la périodicité des tâches :
* Optimal
* Correction mathématiquement prouvée
* Mais ne s'applique qu'aux tâches périodiques et indépendantes
* Ordonnancement par échéances...

**A quel moment ordonnancer ?**

* Trop fréquent : risque de surcharge du processeur
* Peu fréquent : risque de manque de réactivité par rapport à l'environnement

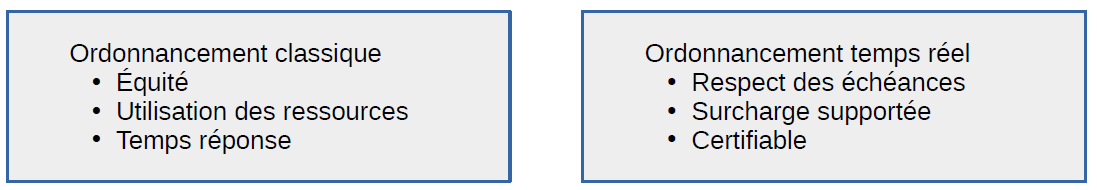
**Plusieurs stratégies :**

* Stratégie préemptive : activation de l'ordonnanceur chaque fois qu'une action est susceptible de faire évoluer l'état du système => bonne réactivité mais risque d'augmenter l'overhead (temps pris par le système pour se gérer lui-même)
* Stratégie préemptive cyclique : activation de l'ordonnanceur lors d'instants prédéfinis (par exemple à chaque top d'horloge…)
* Stratégie non préemptive : le processeur ne peut être retiré aux tâches en activité ; l'ordonnanceur ne peut être activé que lorsque la tâche en activité rend le processeur

**Définitions :**

Faute temporelle : non-respect d'une contrainte temporelle associée à une tâche (dépassement d'échéance) → Notion d'urgence (choix d'une tâche à exécuter par rapport aux paramètres temporels)

Surcharge : occurrence d'une ou plusieurs fautes temporelles → Notion d'importance (choix d'une tâche à exécuter par rapport aux spécifications fonctionnelles de l'application)



Une séquence d'exécution fournie par l'ordonnanceur est valide si **les échéances des tâches sont respectées**. Un algorithme est fiable pour une configuration de tâches s'il produit une séquence valide sur une durée infinie quelles que soient les valeurs des premières dates de déclenchement des différentes tâches.

Une configuration est ordonnançable s'il existe au moins un algorithme fiable. L'algorithme d'ordonnancement peut être qualifié selon deux aspects principaux :

1. Optimalité : un algorithme est optimal s'il trouve une solution d'ordonnancement du système (si cette solution existe) ; s'il ne trouve pas de solution à ce système alors aucun autre algorithme ne peut en trouver une.
2. Ordonnançabilité : c'est la capacité de pouvoir prévoir l'ordonnancement de la configuration de tâches en se basant sur les conditions nécessaires et/ou suffisantes ou des simulations de l'exécution.

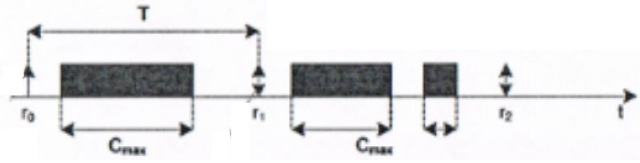
**Modélisation des tâches indépendantes périodiques (avec ordonnance sur requête)**

La modélisation des tâches périodiques repose sur trois paramètres temporels :

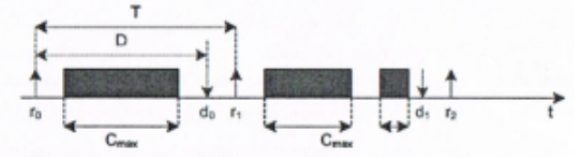
* r0 : date de réveil de la tâche
* C = Cmax : durée d'exécution maximale ;
* T : période d’exécution

La date de réveil rk de la kème instance d'une tâche est définie par : rk = r0 + kT

Pour limiter les indéterminismes d'exécution, on supposera que les tâches sont non- réentrantes : la tâche doit avoir terminé avant la fin de sa période.



L'échéance de la tâche peut être plus courte que la fin de sa période :



La modélisation des tâches périodiques strictes va donc reposer sur quatre paramètres temporels :

* r0 : data de réveil ;
* C = Cmax : durée d'exécution maximale ;
* D : délai critique ;
* T : période d'exécution.

L'échéance dk de la kème instance est : dk = rk + D = r0 + kT + D, 0 ≤ Ci ≤ Di ≤ Ti

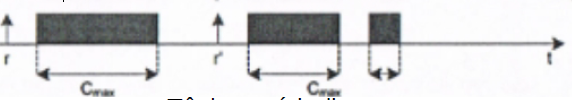
La zone d'exécution de la tâche se situe entre les dates rk et dk ;

La zone entre dk et rk+1 ne peut pas être utilisée pour exécuter la tâche.

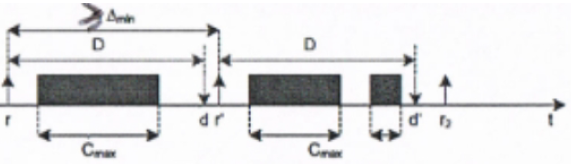
**Modélisation des tâches indépendantes apériodiques**

Pour les tâches apériodiques, le seul paramètre connu est la durée d'exécution C.

La date de réveil est aléatoire car elle dépend du contexte d'évolution du procédé :



**Tâche apériodique**



**Tâche sporadique**

En se plaçant dans un environnement d’exécution stricte, la tâche apériodique, appelée apériodique stricte ou sporadique, doit posséder un délai critique D qui conduit à des échéances strictes pour chaque instance d'exécution : d = r + D ou d' = r' + D

Pour limiter les indéterminismes, on suppose que les tâches possèdent un délai minimum Δmin :

r' – r ≥ Δmin, 0 ≤ C ≤ D ≤ Δmin

**Paramètres temporels statiques**

* **Laxité L** : c'est le temps restant entre la fin d'exécution de la tâche et son échéance.

L ≤ D – C et Lmax = D – C

* **Délai de latence Dl** : c'est le temps avant l'exécution de la tâche

Dl ≤ D – C et Dlmax = D – C = Lmax

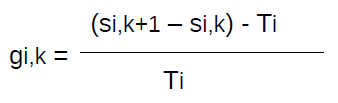
* **Début d'exécution** si,k de la kème instance de la tâche τi.
* **Fin d'exécution** ei,k de la kème instance de la tâche τi. Si si,k correspond au délai de latence, alors : si,k ≥ τi,k ; ei,k ≤ di,k et ei,k – si,k ≥ C

Pour une exécution au plus tôt si,k = ri,k et pour une exécution au plus tard ei,k = di,k. Lorsqu'il n'y a aucune préemption ei,k – si,k = C

* **Temps de réponse** de la kème instance de la tâche τi :

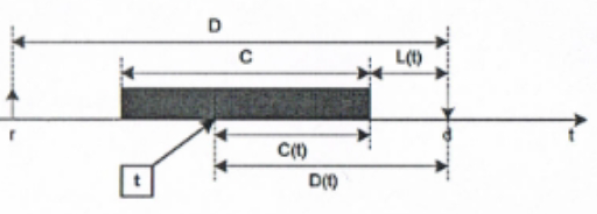
TRi,k = ei,k – τi,k

* Gigue ou **régularité d'exécution** :



**Paramètres temporels dynamiques**

Ce sont les paramètres en fonction du temps :

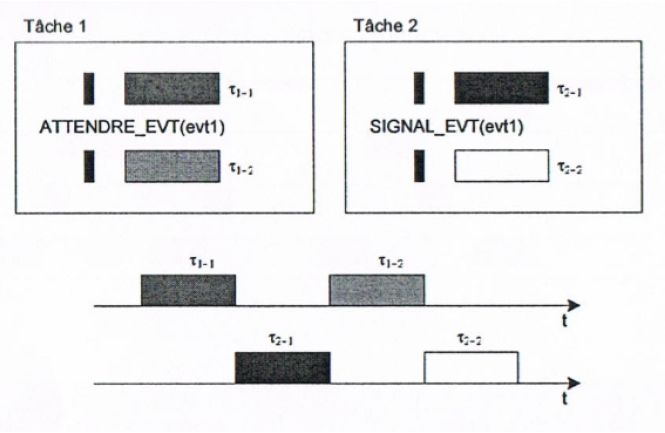


* Temps d'exécution restant C(t) : C(t) = C – C exécuté(t) ;
* Délai critique dynamique D(t) : temps restant avant la prochaine échéance D(t) = d – t ;
* Laxité dynamique L(t) : c'est le temps avant la fin de l'exécution de la tâche :

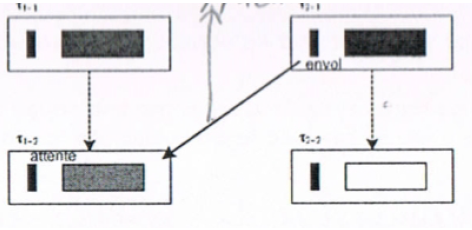
L(t) = D(t) – C(t) = d – t – C(t)

**Modélisation des tâches dépendantes : précédence**

La précédence est mise en œuvre par des primitives de synchronisation ou de communication intégrées dans le code des tâches, *Exemple : deux tâches non atomiques*.

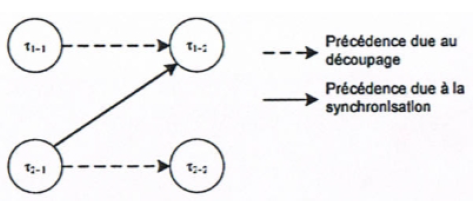


Pour transformer ces tâches en tâches atomiques, il suffit de décomposer les deux tâches au niveau des primitives de synchronisation



* Attente de synchronisation ou de communication en début de tâche ;
* Envoi d'événement de synchronisation ou de communication en fin de tâche

L'ensemble des tâches obtenu reste rigoureusement équivalent aux tâches initiales au niveau de l'exécution. Pour signifier les relations entre les tâches, il faut leur adjoindre un graphe de relation :

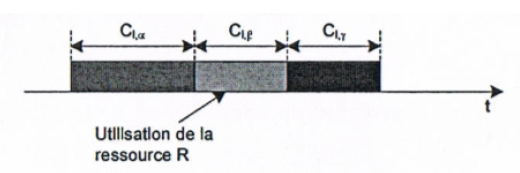


**Modélisation des tâches dépendantes : partage**

***Partage des ressources critiques***

L'exécution d'une tâche utilisant une ressource critique R peut être décrite par trois valeurs : le temps avant la section critique Ci,α, la durée de la section Ci,β et le temps après la section Ci,γ :

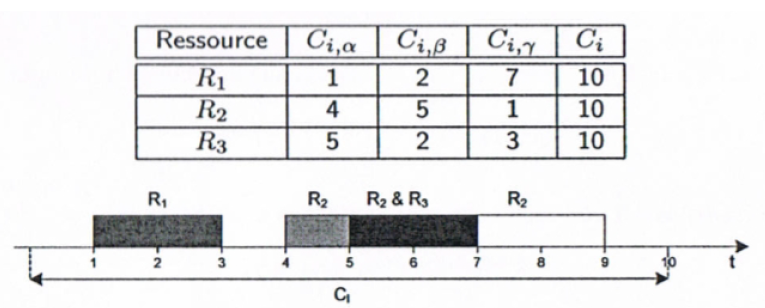
Ci = Ci,α + Ci,β + Ci,γ



Si une tâche τi utilise plusieurs ressources, les sections critiques doivent être correctement imbriquées. Par exemple, pour deux ressources R1 et R2 :

Sci,1 ⊂ Sci,2 ou Sci,2 ⊂ Sci,1 ou Sci,1 ∩ Sci,2 = ∅

*Exemple : une tâche τi utilisant trois ressources. On vérifie bien que toutes les sections critiques vérifient l'une des trois conditions précédentes.*



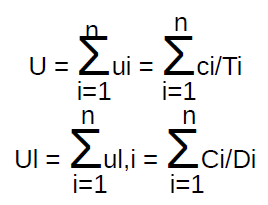
**Analyse de configuration des tâches périodiques**

***Charge processeur***

Pour une tâche périodique τi, définie avec les paramètres de base (ri, Ci, Di, Ti), on définit :

* Le facteur d'utilisation ui : pourcentage du processeur nécessaire à son exécution sur une période Ti : (la durée de la tache sur la période) : ui = Ci/Ti
* Le facteur de charge ul,i : pourcentage du processeur sur le délai critique Di : ul,i = Ci / Di

Pour une configuration de n tâches périodiques T = {τ1, …, τn}, on définit le facteur d'utilisation U et le facteur de charge Ul par :



Si toutes les tâches sont à échéance sur requête, alors U = Ul.

Une condition nécessaire d'ordonnançabilité sur un système monoprocesseur : Facteur d'utilisation U < 1

***Période d’étude***

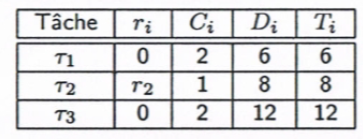
Pour une configuration de n tâches T = {τ1, …, τn} à départ simultané (τi = 0, ∀i), i), l'étude de la séquence d'exécution peut se limiter à un temps H appelé période d'étude ou méta-période ou cycle majeur : H = PPCM{Ti} 1 ≤ i ≤ n

Si ∃i|ri ≠ rj, j ≠ I, la séquence d’exécution commence par une phase transitoire de i|ri ≠ rj, j ≠ I, la séquence d’exécution commence par une phase transitoire de durée maximale Ht :

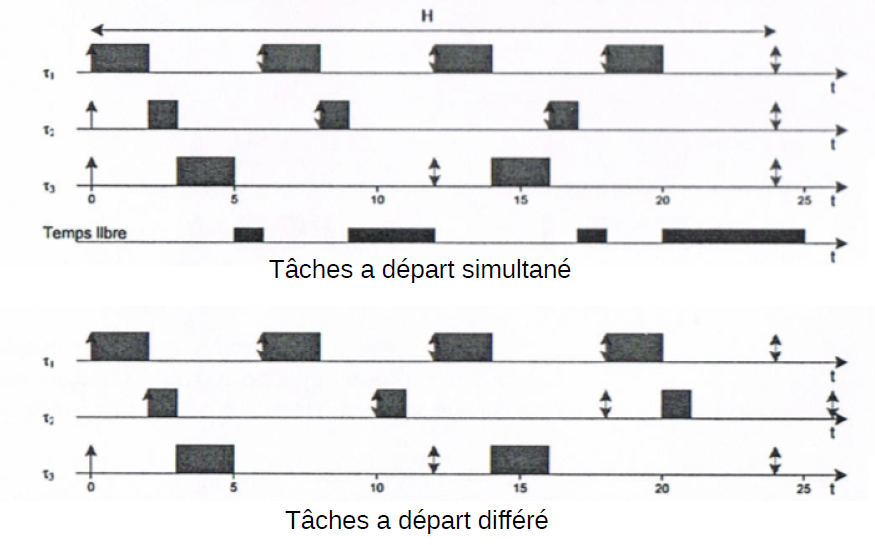
Ht = max{ri}i ≤ i ≤ n + PPCM{Ti} 1 ≤ i ≤ n

A partir d'un instant situé dans l'intervalle [0, Ht], la séquence d'exécution devient périodique

*Exemple :*

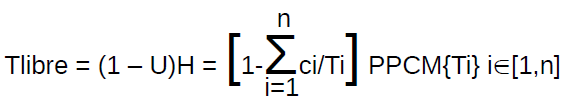


* Dans le cas où r2 = 0, on a des tâches à départ simultané et H = 24
* Dans le cas où r2 = 2, les tâches sont à départ différé et Ht = 26

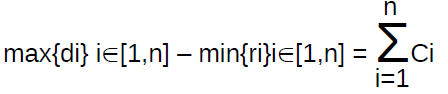


***Temps libre du processeur***

Pour une configuration de n tâches périodiques à départ simultané s'exécutant sur un système monoprocesseur, on définit le temps libre ou temps creux ou temps oisif processeur par :



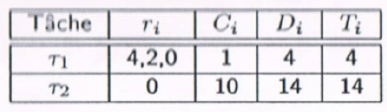
Le temps libre processeur est également défini pour les tâches à départ non simultané en considérant la partie périodique qui se situe après la phase transitoire. Une séquence est dite saturée si l'allocation du processeur est complète (U = 1) :

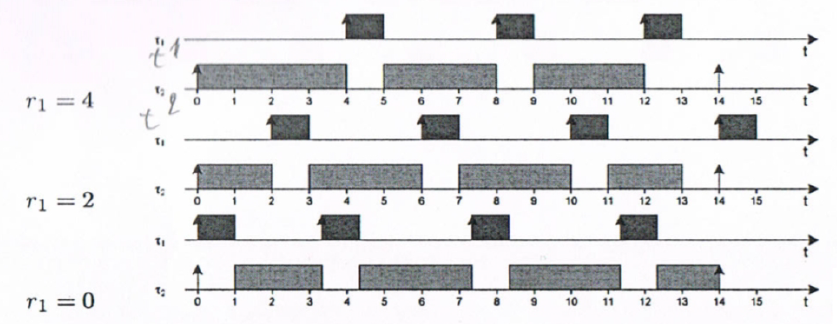


***Instance critique***

En terme d'ordonnançabilité, la situation de pire cas (temps de réponse le plus long) se produit lorsque toutes les tâches sont à départ simultané (activation au même instant, appelé Instant critique). Si une configuration est ordonnançable dans le pire des cas, elle le sera obligatoirement si une ou plusieurs tâches sont à départ différé.

*Exemple :*



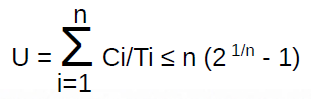


***Algorithmes à priorité fixe : Rate monotonic***

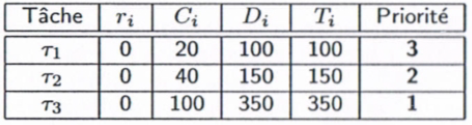
L'algorithme Rate Monotonic (RM) consiste à affecter aux tâches une priorité qui dépend de la période. Plus la période est petite, plus la priorité est grande.

Lorsque τi = 0 et Di = Ti, ∀i), 1 ≤ i ≤ n :

* L'algorithme RM est optimal dans la classe des algorithmes à priorité fixe ;
* Une CS d'ordonnançabilité est obtenue avec le facteur d'utilisation :



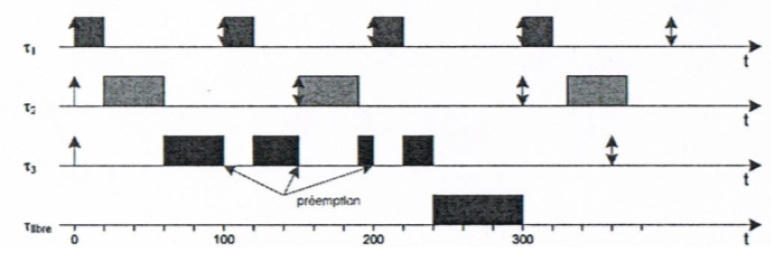
*Exemple :*



On vérifie bien qu'on est dans une configuration de tâches indépendantes, périodiques, à échéance sur requête et à départ simultané. Le facteur d'utilisation est :

U = 20/100 + 40/150 + 100/350 ≈ 0.75 ≤ 3 (2 - 1) ≈ 0.779

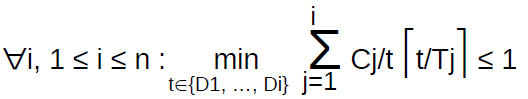
La configuration est ordonnançable



Partie de la séquence d'exécution

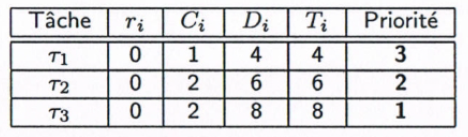
* La tâche de plus forte priorité τ1 s'exécute dès sa demande (date de réveil) ;
* La tâche de plus faible priorité τ3 est préemptée 3 fois dans la partie de la séquence illustrée par la figure ;
* Le temps libre processeur sur l'ensemble de la période d'étude est :

Tlibre = (1 – 0.75238) x 2100 = 520

Lorsque les tâches sont à départ simultané, une CNS d'ordonnançabilité par RM est donnée par le théorème de la zone critique. Celui-ci exprime le fait que si les tâches respectent leur première échéance, alors la configuration est ordonnançable quel que soit l'instant d'arrivée des tâches par la suite. Théorème de la zone critique, Un ensemble de n tâches T = {τ1, …, τn} ordonnées suivant les priorités décroissantes définies par les paramètres (r1, C1, D1, T1) est ordonnançable si et seulement si : Avec ⌈x⌉ est l'entier supérieur le plus proche de x

Cette condition est une CS quand les tâches sont à départ différé.

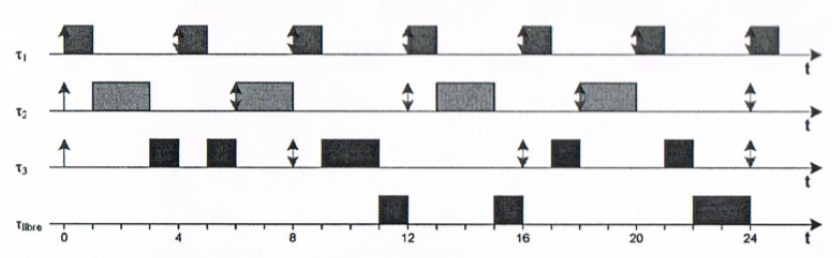
*Exemple : U ≈ 0.833 > 3 (2 - 1) ≈ 0.779*



Utilisation de la condition dite de zone critique :

* Pour la première tâche avec D1 = 4
* Pour la deuxième tâche avec D2 = 6
* Pour la troisième tâche avec D3 = 8

**Conclusion :**

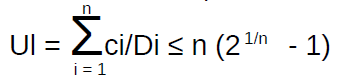


***Algorithmes à priorité fixe : Deadline monotonic***

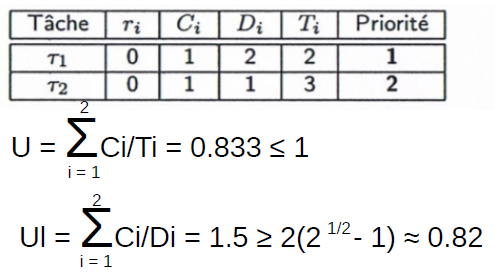
S'il existe au moins une tâche qui n'est pas a échéance sur requête, alors on utilise un algorithme d'affectation des priorités basé sur les délais critiques. Cet algorithme est appelé Deadline Monotonic (DM) ou Inverse Deadline (ID)

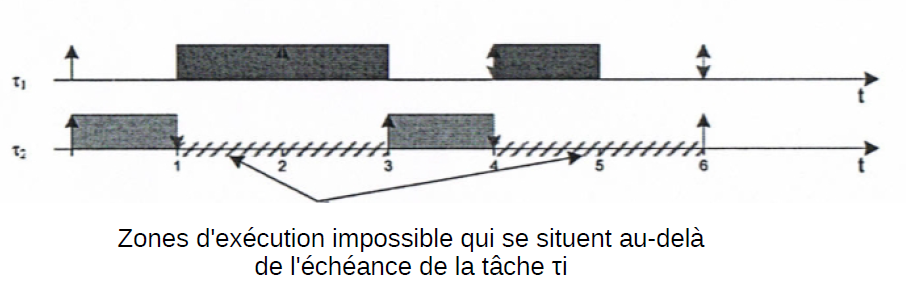
Pour une configuration avec ri = 0, ∀i), i et ∃i|ri ≠ rj, j ≠ I, la séquence d’exécution commence par une phase transitoire de i|Di ≠ Ti

* L'algorithme DM est optimal dans la classe des algorithmes à priorité fixe
* Une CS d'ordonnançabilité est donnée par le facteur de charge Ul :



Exemple :





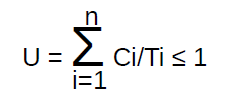
***Algorithmes à priorité variables : Earliest Deadline first***

Pour une instance k de la tâche τi, la priorité est fonction de la prochaine échéance di,k de cette tâche. A l'instant t, la priorité peut être calculée à partir du délai critique dynamique Di(t) :

Di(t) = di,k – t = ri,k + Di – t

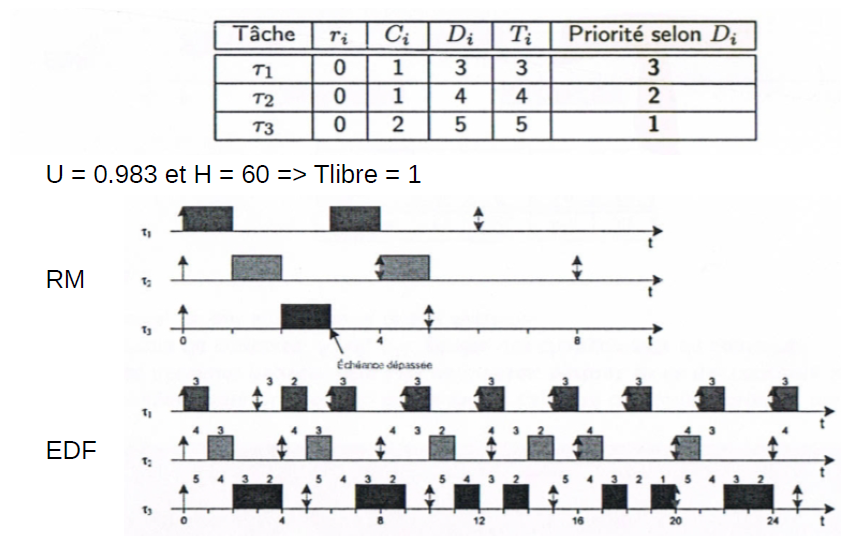
La priorité dynamique augmente quand le délai critique dynamique diminue.

La CNS d'ordonnançabilité d'une configuration de tâches indépendantes, périodiques, à échéance sur requête et à départ simultané avec l'algorithme EDF est :



EDF est optimal dans la catégorie des algorithmes à priorité variable.

Exemple :



***Algorithmes à priorité variables: Minimum Laxity***

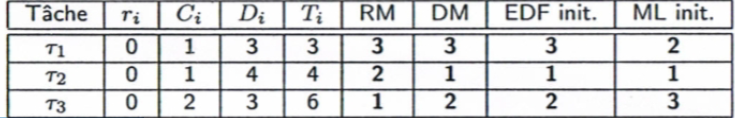
Pour une instance k d'une tâche τi, la priorité est liée à la laxité dynamique :

Li,k(t) = di,k – Ci(t) – t = ri,k + Di – Ci(t) – t

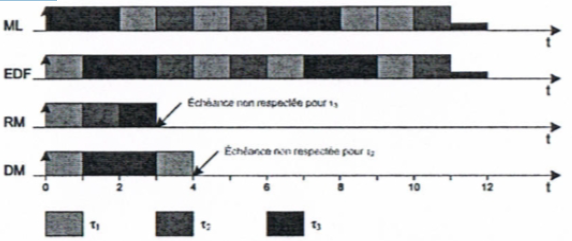
L'algorithme ML a les mêmes caractéristiques que EDF : optimalité et ordonnançabilité.

Cet algorithme est appelé Minimum Laxity (ML) ou Least Laxity (LL).

Exemple : Comparaison des algorithmes d'ordonnancement étudiés.

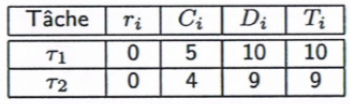


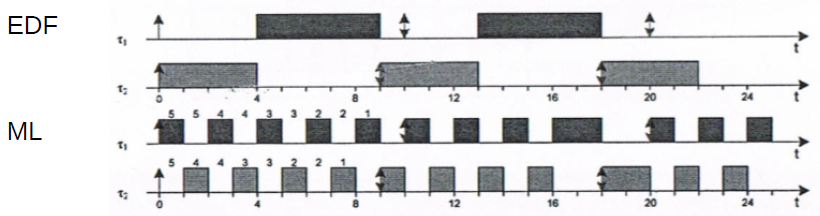
U = 0.93 et H = 12



Les deux algorithmes EDF et ML ont la même capacité d'ordonnançabilité, mais les séquences obtenues peuvent être très différentes, surtout en ce qui concerne les changements de contexte. Ainsi, ML génère des changements de contexte intempestifs, ce qui augmente le temps système.

Exemple :





L'algorithme ML contient 5 fois plus de changements de contexte que EDF ! En environnement monoprocesseur, il faut préférer EDF.