Real-time systems

Ordonnancement temps-réel

Laure Gonnord

University of Lyon/ LIP

MIF18 - M1 optional course. Feb-June 2019

Plan

Lyon 1

Source

- Ordonnancement et systèmes classiques : cours de T. Excoffier.
- Ordonnancement et systèmes Temps-Réel : cours de M. Pouzet. Comparaison entre algos : P. de Oliveira.

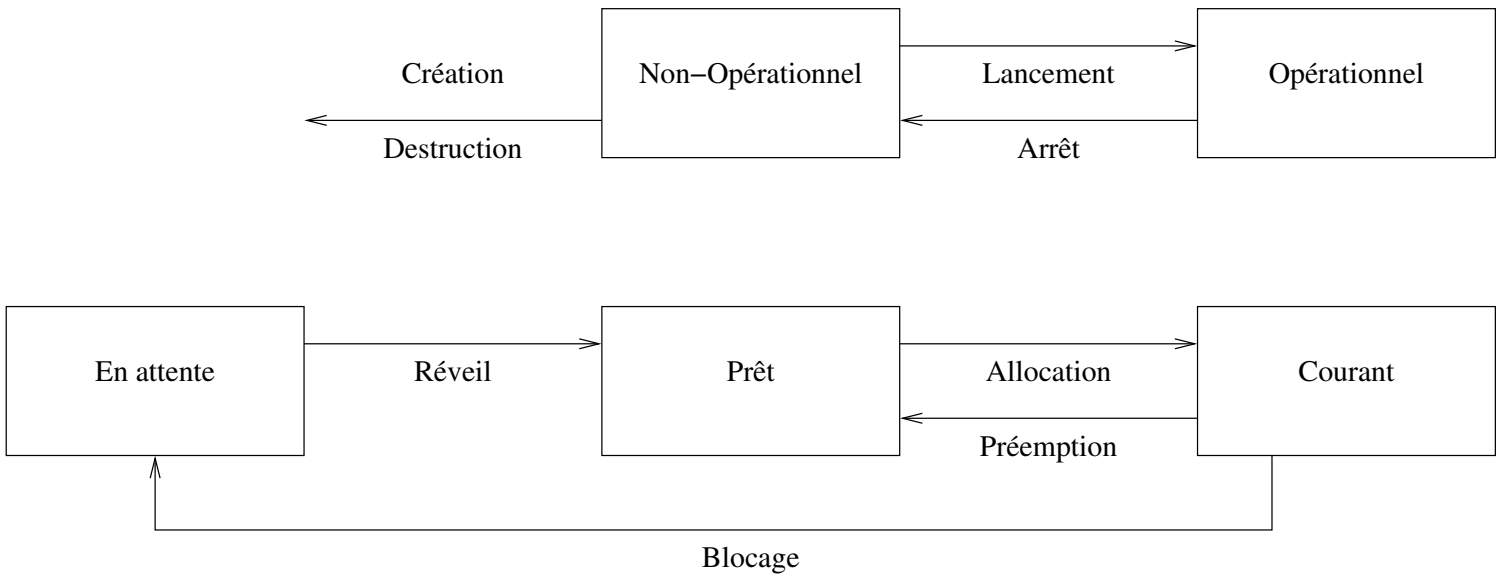
Plan

Lyon 1

Généralités

Il n'existe pas d'ordonnanceur optimum pour tous les cas d'utilisation d'un système d'exploitation.

Schéma général des tâches système :



Un ordonnanceur, pourquoi ?

C'est l'ordonnanceur qui gère l'allocation et la suspension des tâches. L'ordonnanceur peut être :

- "hors-ligne" ou "en ligne".
- Préemptif ou non-préemptif.

Ordonnanceurs classiques :

- PAPS (FIFO) Premier arrivé, premier servi.
- Tourniquet (Round Robin) à tour de rôle avec un *quantum* de temps.
- Priorité : plus prioritaire d'abord.

Les priorités Unix

Priorités $\neq$ respect des échéances :

- Les priorités UNIX, laissent la main aux tâches moins prioritaires afin qu'elles avancent un peu.
- Si les tâches faiblement prioritaires n'avançaient jamais cela pourrait introduire des *dead-lock*.

▶Mais les priorités ne permettent pas forcément le respect des échéances.

Algo d'ordo PAPS (FIFO)

Principe : premier arrivé premier servi.

Algo PAPS, durée : A=4, B=6, C=2														
Temps	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Qui ?	A	A	A	A	B	B	B	B	B	B	C	C		D
Démarrage	A	B	C											D

Algo d'ordo Tourniquet (Round Robin)

Principe : donner à manger à tout le monde "en tournant".

Algo tourniquet, durée : A=4, B=6, C=2														
Temps	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Qui ?	A	B	C	A	B	C	A	B	A	B	B	B		D
Démarrage	A	B	C											D

Algo d'ordo PAPS + prio

Principe : idem PAPS, mais en mettant des priorités (choix du processus à ordonnancer si deux sont ordonnançables).

PAPS + Priorités : A=+, B=++, C=++														
Temps	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Qui ?	A	B	B	B	B	B	B	C	C	A	A	A		D
Démarrage	A	B	C											D

Algo d'ordo Tourniquet + prio

Principe : idem tourniquet + priorités.

Tourniquet + Priorités : A=+, B=++, C=++														
Temps	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Qui ?	A	B	C	B	C	B	B	B	B	A	A	A		D
Démarrage	A	B	C											D

Inconvénients des ordonnanceurs classiques

- Le processeur est toujours utilisé et ne peut se mettre en veille. Le passage à l'état de veille du processeur est souvent long.
- Cela fait perdre du temps CPU. (très gênant quand on fait tourner des machines virtuelles.)
- Le *quantum* de temps doit plutôt être choisi en fonction de la charge de la machine.

Solution : au lieu d'être lancé périodiquement, l'ordonnanceur indique dans combien de temps il doit être réveillé.
Évidemment, il peut être réveillé par une tâche prioritaire qui devient prête.

Plan

Lyon 1

Problématique

Respect des contraintes TR : échéances, périodicité, ...

Approches **"bare metal"** :

- Pas d'OS : produire un code décrivant la boucle de réaction.
- Penser en pire cas.
- On vérifie que l'implémentation est suffisamment rapide par "Analyse du temps d'exécution pire-cas".
- Pour les applis les + critiques.

▶Tend à **sur-dimensionner** dans le cas de systèmes non critiques. Difficile dans dans le cas où l'on veut du **parallélisme** (ex : moteur).

▶Dans la suite : approches construisant un OS temps-réel.

Lyon 1

OS temps réel, problématique

Un OS temps réel disposera d'un ordonnanceur de tâches :

- il gère les priorités, traite le temps de manière ad-hoc en donnant des garanties de temps de traitement des interruptions.
- avec des garanties.

Des garanties de temps réel à condition que soient connus statiquement :

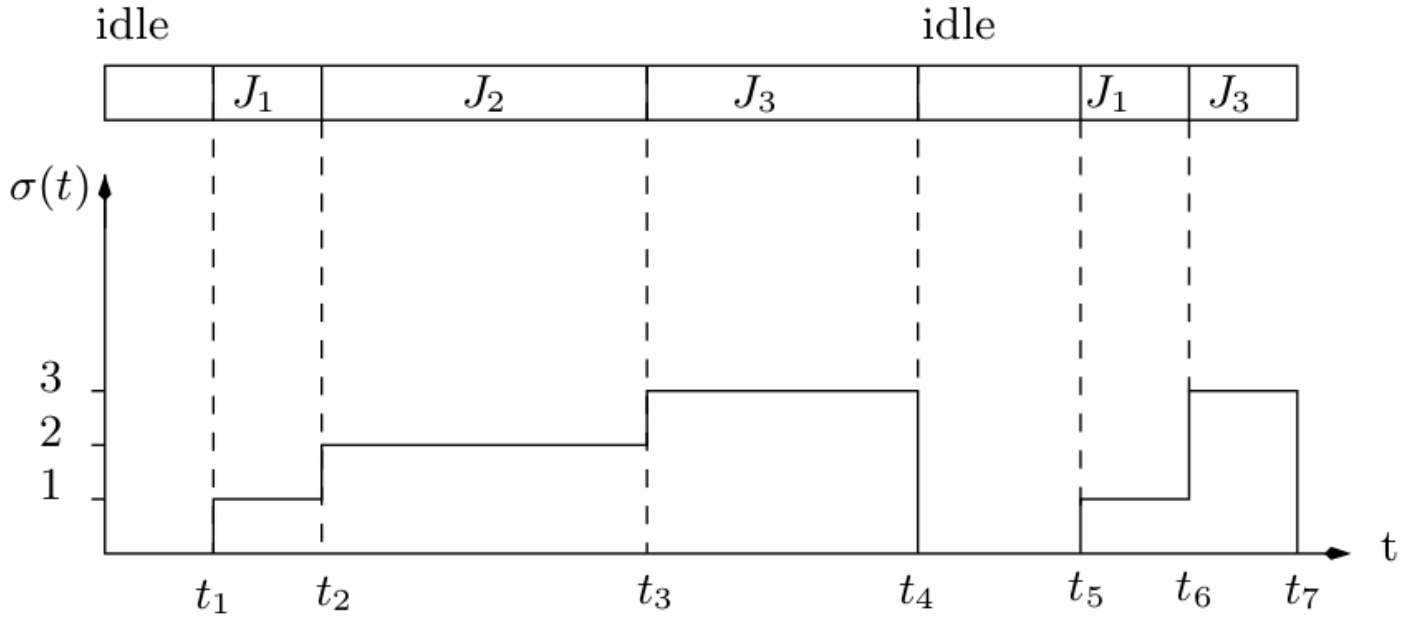
- le nombre de tâches
- les durées/coûts de chaque tâche
- les priorités entre tâches.

Lyon 1

Ordonnancement temps réel sur monoprocesseur

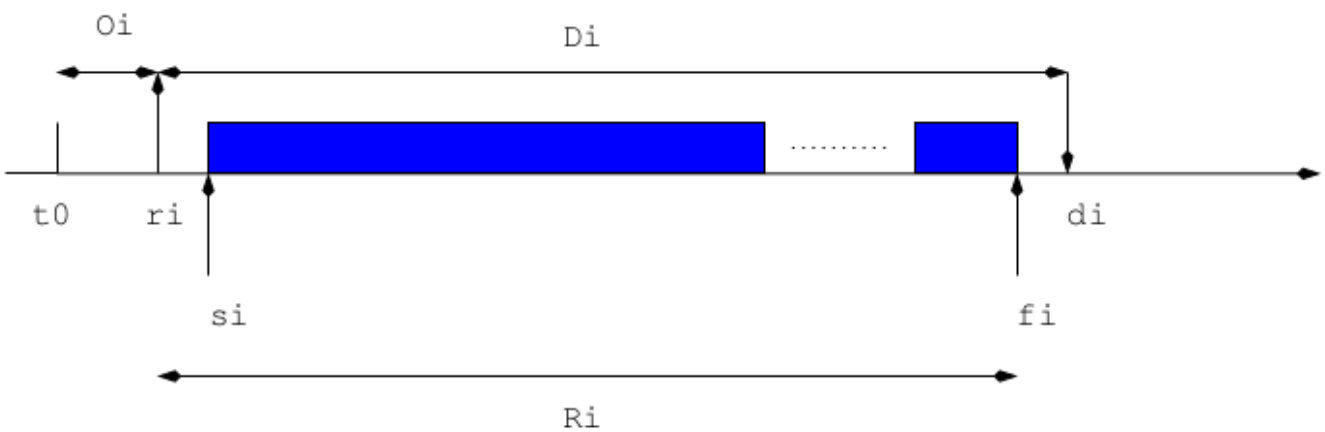
Soit un ensemble de tâches $J = \{J_1, \ldots J_n\}$. Un **ordonnancement** est une fonction $\sigma : \mathbb{R}^+ \to \mathbb{N}$ qui assigne des dates à des tâches :

$$\forall t \in \mathbb{R}^+, \exists t_1, t_2 \text{ tq } t \in [t_1, t_2[\text{ et } \forall t' \in [t_1, t_2[, \sigma(t) = \sigma(t')$$



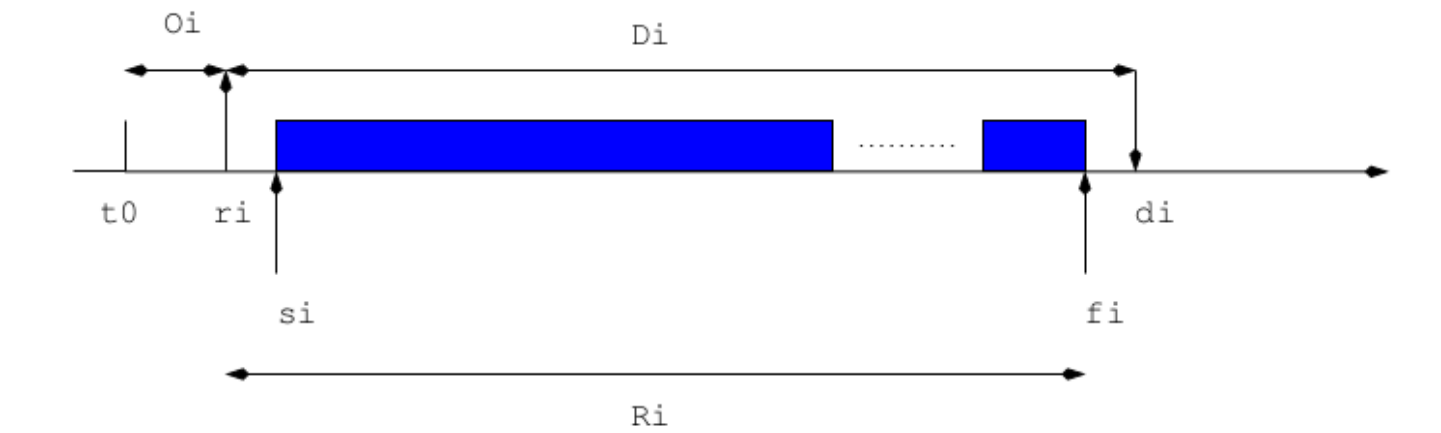
Ordo TR : terminologie, notations 1/3

- Date de début d'exécution :
  - r_i instant où tâche prête à être exécutée
  - O_i décalage par rapport au lancement.
  - (tâche synchrone si $r_i = t_0$).
- Échéance (deadline) :
  - D_i durée à ne pas dépasser pour une exécution
  - C_i borne sup estimée du pire temps d'exec. (WCET, ou capacité)
  - $d_i = r_i + D_i$ date avant laquelle la tâche doit être terminée (échéance absolue).



Ordo TR : terminologie, notations 2/3

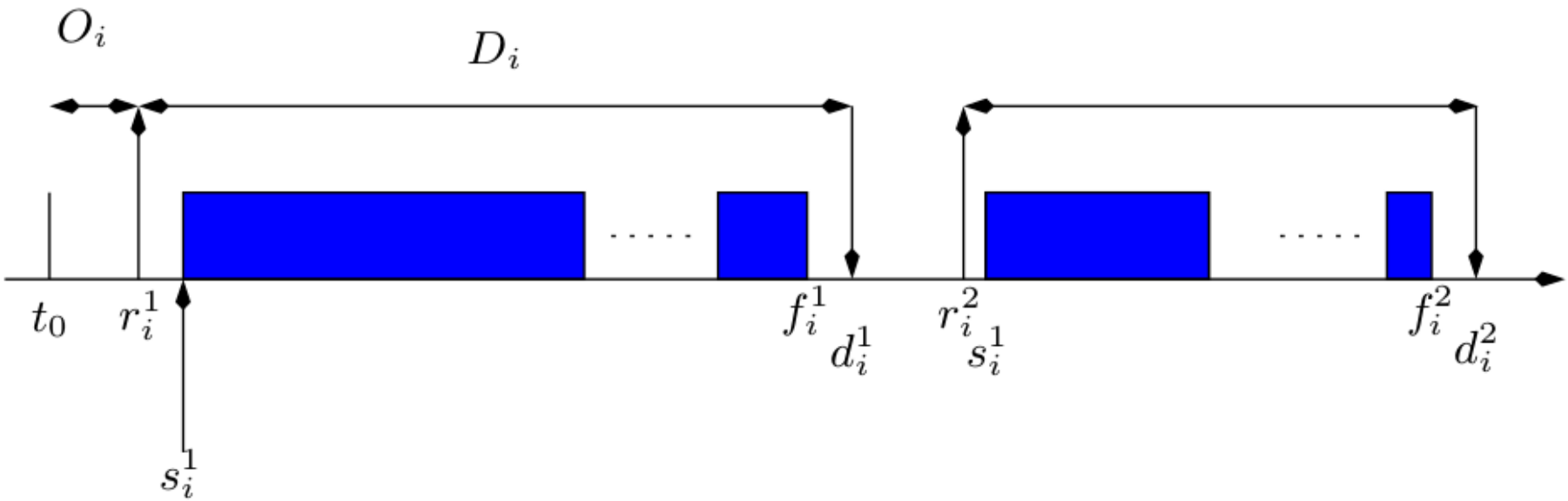
- Dates de début et de fin d'exec : s_i, f_i
- Temps de réponse $R_i = f_i - r_i$.



Ordo TR : terminologie, notations 3/3

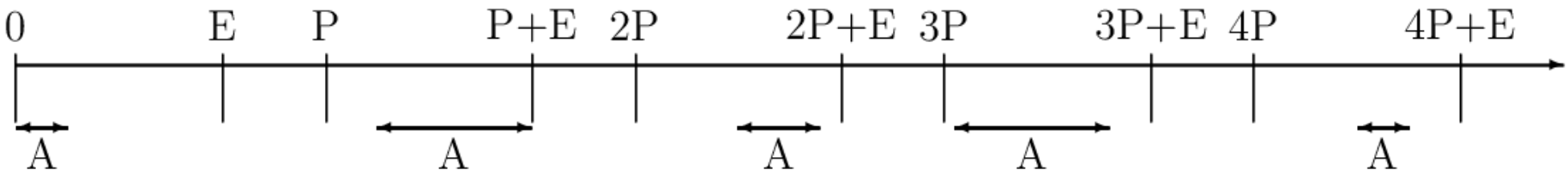
La k-ième instance d'une tâche i est notée t_i^k.

- tâche périodique : l'intervalle entre deux activations est constant, de période T_i, donc $r_i^k = r_i^1 + (k-1)T_i$.
- tâche sporadique : l'intervalle entre deux activations est sup à une certaine valeur, ie $r_i^k - r_i^{k-1} \geq T$.
- tâche apériodique : aucune contrainte sur les dates d'activation.



Notations : exemple

La tâche A respecte période et échéance :



Rque on peut rajouter certaines contraintes temporelles : bornes sur la durée max d'exec, écart max entre deux événements (synchro image/son), taux de production (flux vidéo)...

Algorithmes en-ligne et hors-ligne

Il y a deux classes d'algorithmes d'ordonnancement :

- **Statique** (off-line) L'ordonnancement est pré-calculé statiquement et peut être stocké dans une table qui détermine qui est activé et quand. Cela s'applique seulement lorsque :
  - Le nombre de tâches est connu statiquement.
  - Les prioriétés sont fixes et connues statiquement.

  ▶Absence de flexibilité.
- **Dynamique** (on-line) L'ordonnancement est calculé dynamiquement : meilleure utilisation (charge) du processeur, et on peut prendre en compte des événements sporadiques et apériodiques.

Lyon 1

Plan

Lyon 1

Le sous-problème traité

Ordonnancement de tâches **périodiques** :

- r_i date de réveil
- C_i capacité (WCET)
- D_i échéance (deadline)
- T_i période

On se place dans le cas particulier $r_i = 0$ et $D_i = t_i$.

Remarque (test simple) La charge du processeur est $U = \sum_i \frac{C_i}{T_i}$. Donc, si $U > 1$ il n'existe aucun ordonnancement monoprocesseur.

Ordo avec priorité 1/2

Caractéristiques (ici)

- Une horloge périodique temps réel ($\simeq$ 1 ms)
- On précalcule statiquement des priorités.
- Durant l'exec, l'ordonnanceur choisit la tâche à activer de plus haute priorité. (choix arbitraire si deux égales)

Implem : tableau avec les périodes (donc priorités). Chaque tâche est "runnable" ou pas, et on met à jour le temps restant pour la tâche en cours...

2 critères de choix pour select : RM et EDF

Les deux algorithmes d'ordonnancement les plus connus (et utilisés).

- **Rate Monotonic** (RM) On choisit la tâche de plus forte priorité statique (période minimale). (fixed priority scheduling)

  $$select(\ell) = \text{choose } j \in \ell \text{ such that } T(j) = min_{k \in \ell}(T(k))$$

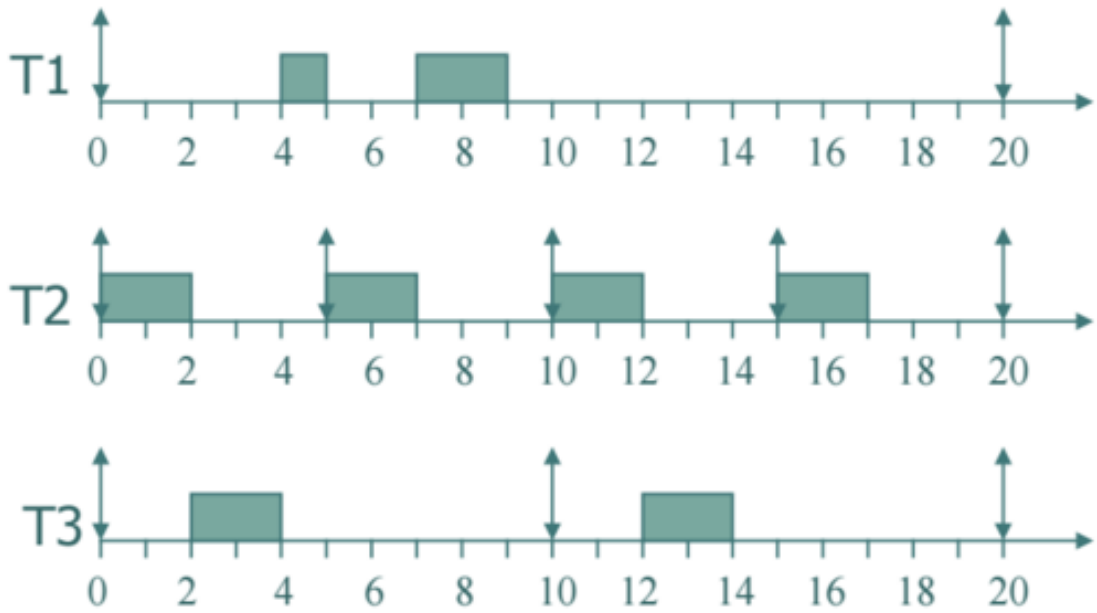
- **Earliest Deadline First** (EDF) On choisit la tâche dont la deadline est la plus proche (dynamic priority scheduling). Marche aussi pour tâches non périodiques.

  $$select(\ell) = \text{choose } j \in \ell \text{ such that } cpt(j) = min_{k \in \ell}(cpt(k))$$

Lyon 1

Rate Monotonic, exemple

Tâche	Période	Échéance	Capacité
T1	20	20	3
T2	5	5	2
T3	10	10	2



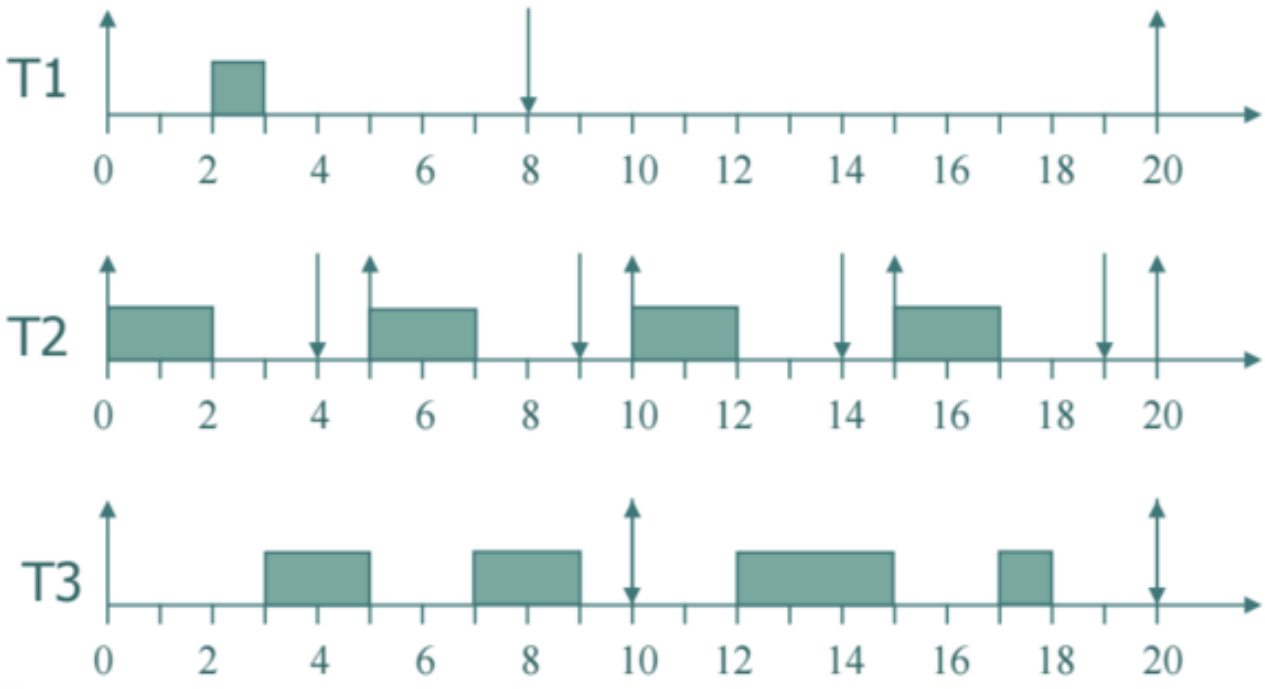
$$Prio(T2) > Prio(T3) > Prio(T1)$$

Exécution cyclique (ppcm des périodes).

Earliest Deadline First, exemple

Tâche	Période	Échéance	Capacité
T1	20	8	1
T2	5	4	2
T3	10	10	4

2 préemptions à $t = 5$ et 15



Lyon 1

Notion de faisabilité d'un (algo) d'ordo.

Il existe des tests simples permettant de savoir si un ensemble de tâches est ordonnançable (pour un algo donné)
▶**test d'ordonnançabilité**.

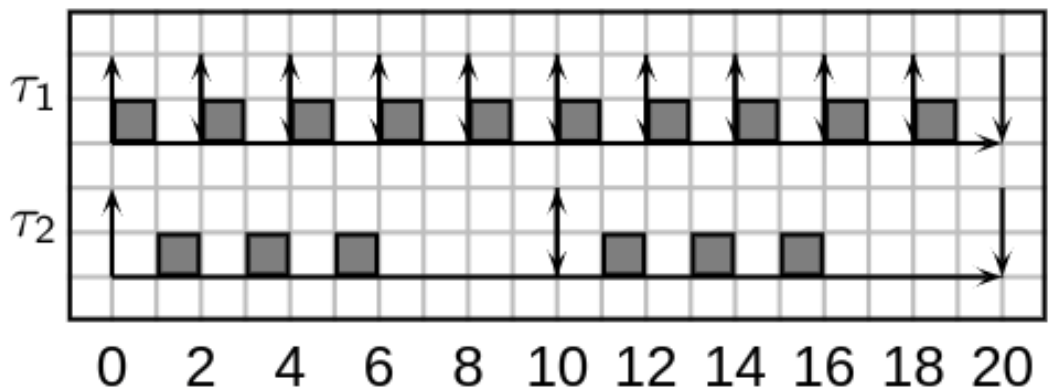
Critères suffisants :

- Pour RM (Liu and Layland, 1973) n tâches indep : $U \leq n(2^{1/n-1})$. La limite de cette quantité est .69, donc si la charge est inférieure à 69%, le système admet un ordo RM.
- pour RM + tâches harmoniques : $U \leq 1$ est CNS.
- Pour EDF : $U \leq 1$ (mais pas nécessaire).

Lyon 1

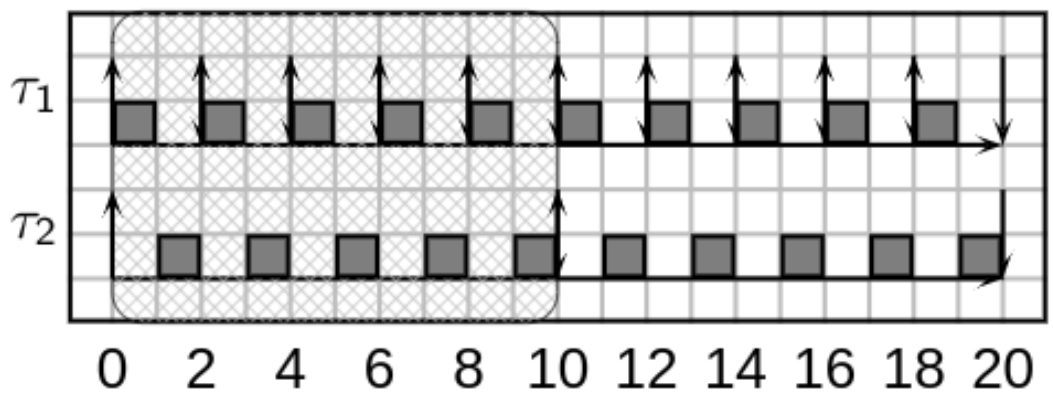
Analyses et exemples pour RM 1/3

- 2 tâches $T_1 = 2s, C_1 = 1s,\ T_2 = 10s, C_2 = 3s$.
- Utilisation : $1/2 + 3/10 = 80\% < 2^{1/2-1} = 83\%$ donc ordonnançable !
- Système harmonique donc on aurait pu utiliser $U \leq 1$.



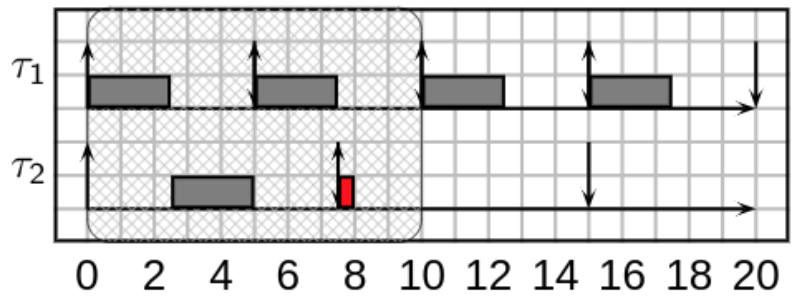
Analyses et exemples pour RM 2/3

- 2 tâches $T_1 = 2s, C_1 = 1s,\ T_2 = 10s, C_2 = 5s$.
- Utilisation : 100% et harmonique, donc OK !

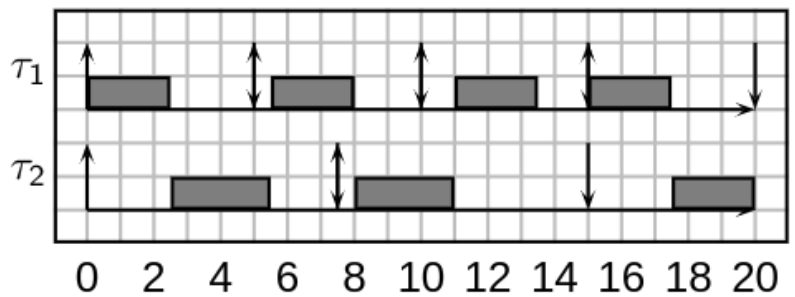


Analyses et exemples pour RM 3/3

- 2 tâches $T_1 = 5s, C_1 = 2,5s,\ T_2 = 7,5s, C_2 = 3s$.
- Utilisation : 90% on ne peut pas conclure, ici NOK.



Mais si on s'affranchit de RM :

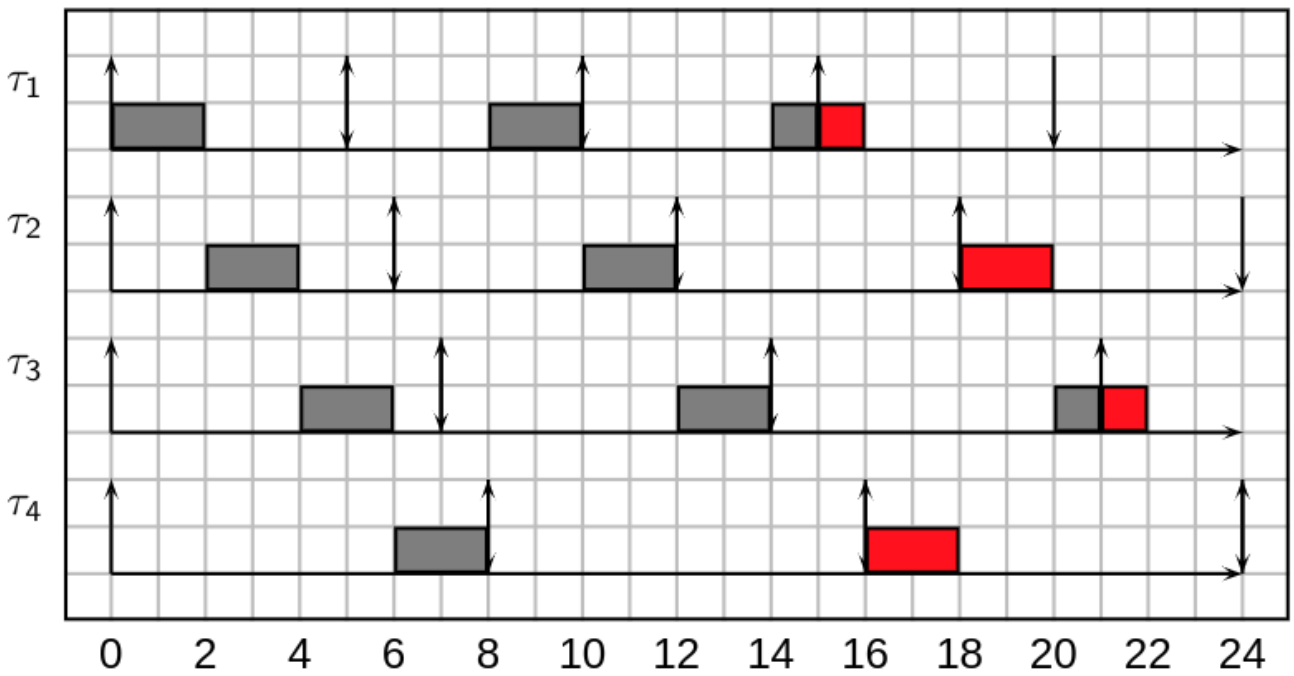


Conditions, encore

- CNS Pour un algorithme donné faire une simulation pire-cas sur une hyper-période $T = ppcm(T_i)$.
- CNS pour EDF $U \leq 1$.
- Plein d'autres dans la littérature.

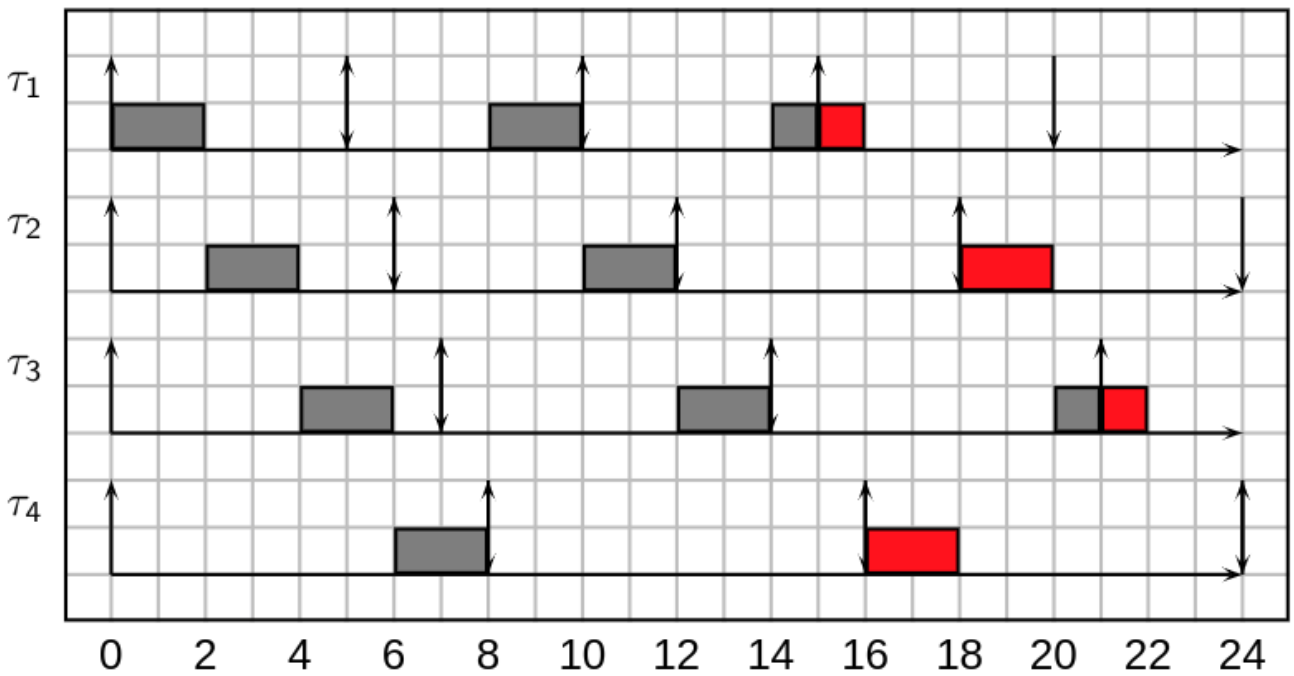
Comparaison RM/EDF 1/3 - effet domino

EDF : à tout moment, le travail prioritaire est celui dont l'échéance est la plus courte. Mais comportement mauvais en cas de surcharge, cela peut provoquer une avalanche d'échéances manquées :



RM / EDF 2/2

RM a un comportement meilleur, le souci affecte les tâches les moins prioritaires, mais certaines tâches peuvent ne jamais être exécutées :



Plan

Lyon 1

Le partage des ressources, le début des ennuis

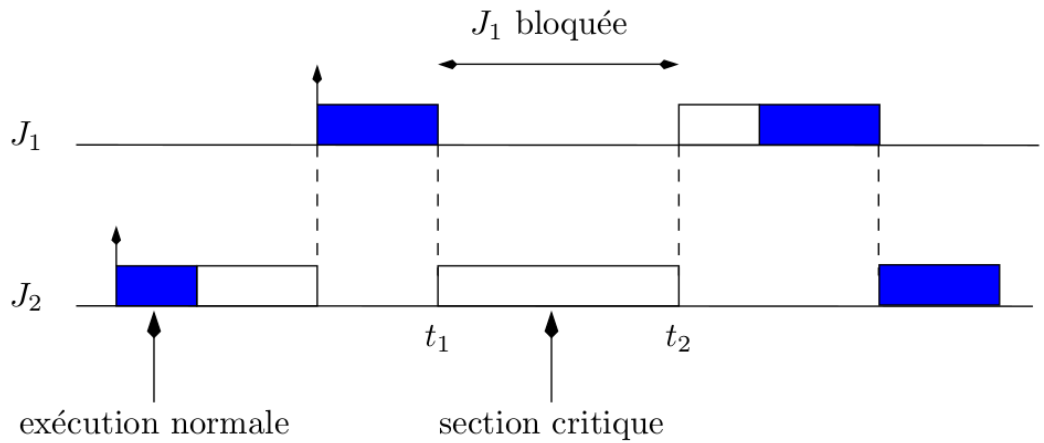
On a considéré le cas de tâches sans synchronisation entre elles et donc sans accès à une ressource partagée nécessitant un mécanisme de verrou.
Considérons deux tâches J_1 et J_2 accédant à une ressource partagée R_k devant être accédée en exclusion mutuelle en utilisant un sémaphore S_k :

- *wait*(S_k) : demande d'accès à la ressource partagée
- *signal*(S_k) : libération de la ressource

$$
\begin{array}{lll}
J_1 = & \ldots & J_2 = \quad \ldots \\
& wait(S_k); & \quad wait(S_k); \\
& \ldots & \quad \ldots \\
& R_k; & \quad R_k; \\
& signal(S_k); & \quad signal(S_k); \\
& \ldots & \quad \ldots
\end{array}
$$

Inversion de priorité

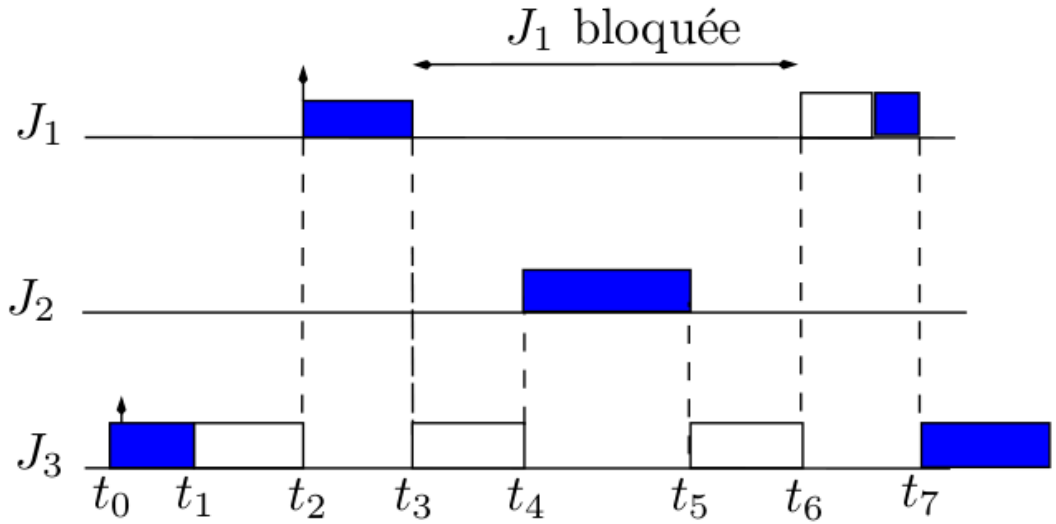
- J_1 a une priorité plus forte que J_2.
- J_2 est activée puis entre en section critique (bloque le sémaphore).
- J_1 arrive. Parce que sa priorité est plus forte, préempte J_2.
- A $t = t_1$, J_1 est bloquée et donc J_2 reprend.
- J_1 doit attendre jusqu'à $t = t_2$, lorsque J_2 quitte la section critique.



▶Le temps d'attente maximum pour J_1 est égale à la durée de la section critique de J_2.

Cas plus grave : Mars Pathfinder, 1997 [1]

Le temps d'attente ne peut pas toujours être borné par la durée de la section critique exécutée par la tâche de priorité la plus faible.

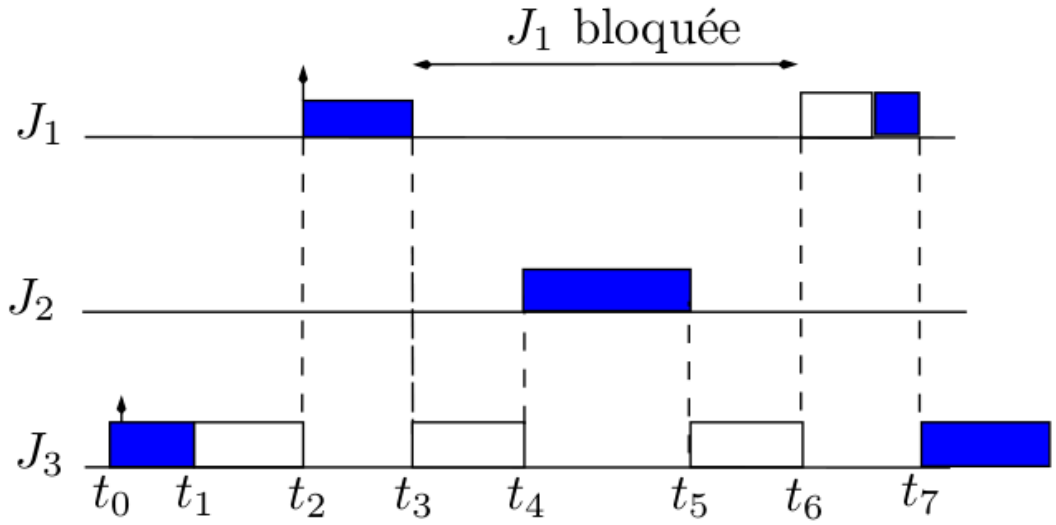


- J_1 arrive au temps t_2 et préempte J_3 durant sa section critique.
- A l'instant t_3 , J_1 tente d'utiliser la resource mais est bloquée sur une sémaphore S.
- Donc J_3 continue son exécution en section critique.
- Si J_2 arrive à $t = t_4$, il préempte J_3 (car il a une priorité plus forte).
- Cela augmente la durée pendant laquelle J_1 est bloquée.

1. `http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html`

Lyon 1

Inversion 3 : toujours Pathfinder



- Le temps de blocage maximum pour J_1 ne dépend plus seulement du temps de la section critique de J_3 ; il dépend du temps d'exécution maximum de J_2 .
- Une inversion de priorité a lieu dans l'intervalle $[t_3, t_6]$

▶Cette durée n'est pas bornée statiquement puisque n'importe quelle tâche de prioriété intermédiaire préemptant J_3 bloque indirectement J_1 .

Solutions à l'inversion de priorité

Plusieurs possibilités :

- Interdire la préemption durant l'exécution d'une section critique : réaliste à condition que celles-ci soient courtes.
- **Héritage de priorité** : modifier la priorité d'une tâche qui cause un bloquage. Quand une tâche J_i bloque une ou plusieurs tâches de plus forte priorité, elle hérite temporairement de la priorité la plus forte de la tâche bloquée.

Héritage de priorité, mise en œuvre

- Lorsque J_i essaie d'entrer en section critique $z_{i,j}$ et que la ressource $R_{i,j}$ est utilisée par une tâche de priorité plus faible, J_i reste bloqué. Sinon, il entre en section critique.
- Lorsque J_i est bloqué sur une sémaphore, il transmet sa priorité à la tâche J_k qui tient le sémaphore. Donc, J_k continue et exécute la suite de sa section critique avec la priorité $p_k = p_i$. J_k hérite de la priorité de J_i.
- Lorsque J_k sort de sa section critique, il débloque le sémaphore et la tâche de plus forte priorité bloquée sur le sémaphore est libérée. La nouvelle priorité de J_k est modifiée : si aucune autre tâche n'est bloquée par J_k , p_k est réinitialisé à sa priorité initiale , sinon, il hérite de la priorité la plus haute des tâches bloquées par J_k .
- L'héritage de priorité est transitif : si une tâche J_3 bloque une tâche J_2 et que J_2 bloque J_1 , alors J_3 hérite de la priorité J_1.

Lyon 1

Plan

Lyon 1

Take out message

- Des algorithmes spécifiques au temps-réel.
- Des algorithmes classiques à connaître et à savoir mettre en oeuvre. (cf “services POSIX pour l’ordonnancement”)

Le cas multiprocesseur, est largement plus complexe.

Lyon 1