PIM-INF4 et IN4R21

Ordonnancement Temps Réel

Damien MASSON

http://esiee.fr/~massond/Teaching/IN4R21

http://esiee.fr/~massond/Teaching/PIM-INF4

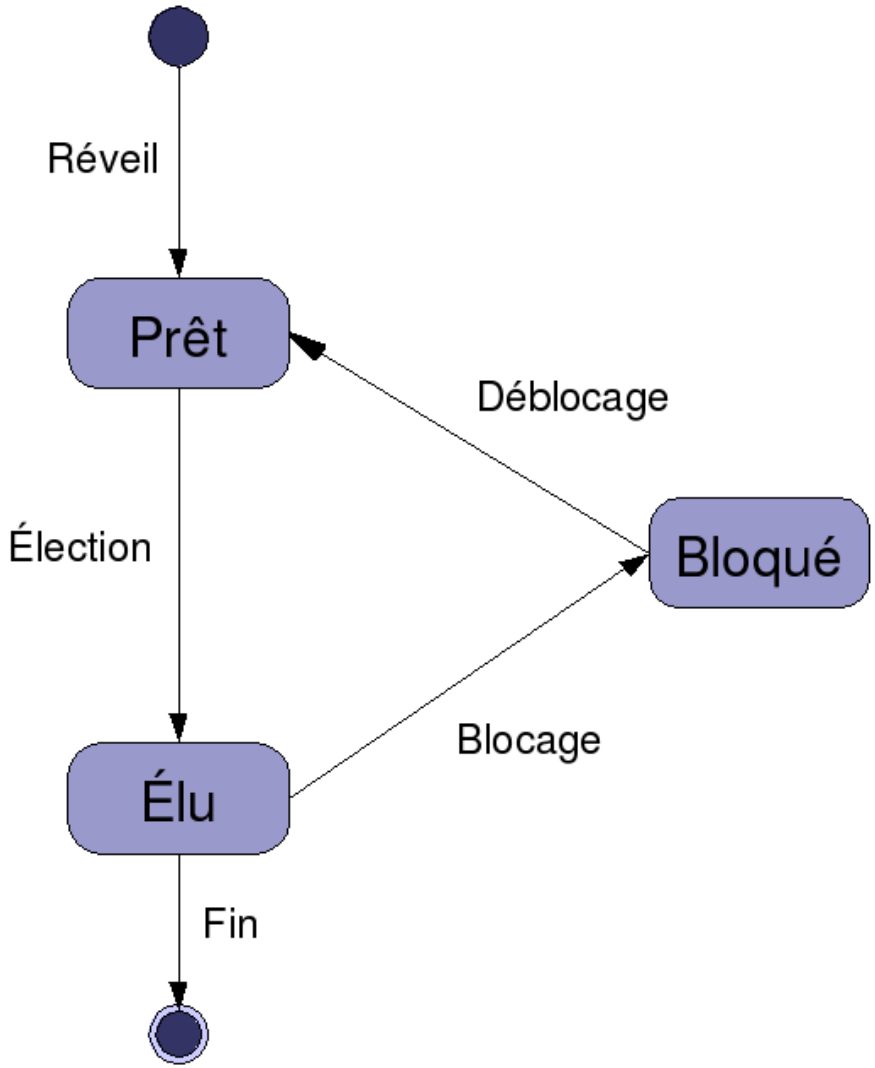
Dernière modification: 19 novembre 2015

Références

- Hard real-time computing systems : predictable scheduling algorithms and applications, Giorgio C. Buttazzo, Springer, 2005 - 425 pages (http://books.google.com/books/about/Hard_real_time_computing_systems.html?id=fpJAZM6FK2sC)
- Cours de Serge Midonnet : http://igm.univ-mlv.fr/~midonnet/Polyamid/JavaTempsReel/Cours/CoursJTR.pdf
- Cours de Nicolas Pernet : http://esiee.fr/~massond/Teaching/PIM-INF4/np.pdf
- Ma thèse, partie I : http://esiee.fr/~massond/Research/thesis/
- Internet

Concurrence

Depuis pas mal de temps les OS sont multi tâches, même sur des architectures mono processeur. Comment est-ce possible ? Grâce aux processus !



Temps réel

Temps réel : différent de rapide, plutôt synonyme de prévisible, ou "garantissable".

Temps réel

Plusieurs niveaux de contraintes :

- STR à contraintes strictes (temps réel dur) : le non respect des contraintes temporelles peut conduire à des catastrophes (avionique)
- STR à contraintes relatives (temps réel souple) : le non respect de contraintes temporelles est toléré, sans conséquence catastrophique (multimédia)
- STR à contraintes mixtes

Temps réel

Plusieurs niveaux de contraintes :

- STR à contraintes strictes (temps réel dur) : le non respect des contraintes temporelles peut conduire à des catastrophes (avionique)
- STR à contraintes relatives (temps réel souple) : le non respect de contraintes temporelles est toléré, sans conséquence catastrophique (multimédia)
- STR à contraintes mixtes

Exemple

Le standard DO-178B développé aux USA pour l'avionique définit 5 niveaux de conséquence en cas de faute répartis entre

- Safety Critical : l'échec conduit à des pertes humaines
- Mission Critical : Systèmes de navigation, Système d'affichage

Ordonnancement

- algorithme d'ordonnancement : c'est l'algorithme utilisé pour décider quelle tâche doit s'exécuter
- ordonnancement : c'est le résultat de l'algorithme d'ordonnancement
- ordonnanceur : c'est la tâche chargée d'appliquer l'algorithme d'ordonnancement

- Deux familles : préemptif, non préemptif
- Deux méthodes : hors ligne ou en ligne

Ordonnancement

- algorithme d'ordonnancement : c'est l'algorithme utilisé pour décider quelle tâche doit s'exécuter
- ordonnancement : c'est le résultat de l'algorithme d'ordonnancement
- ordonnanceur : c'est la tâche chargée d'appliquer l'algorithme d'ordonnancement

- Deux familles : préemptif, non préemptif
- Deux méthodes : hors ligne ou en ligne

On s'intéresse dans ce cours aux algorithmes préemptifs en ligne.

Modèle Périodique

Temps réel : garantie des contraintes. Quelles contraintes ?

Le modèle le plus étudié (vient du contrôle) : les systèmes de tâches périodiques.

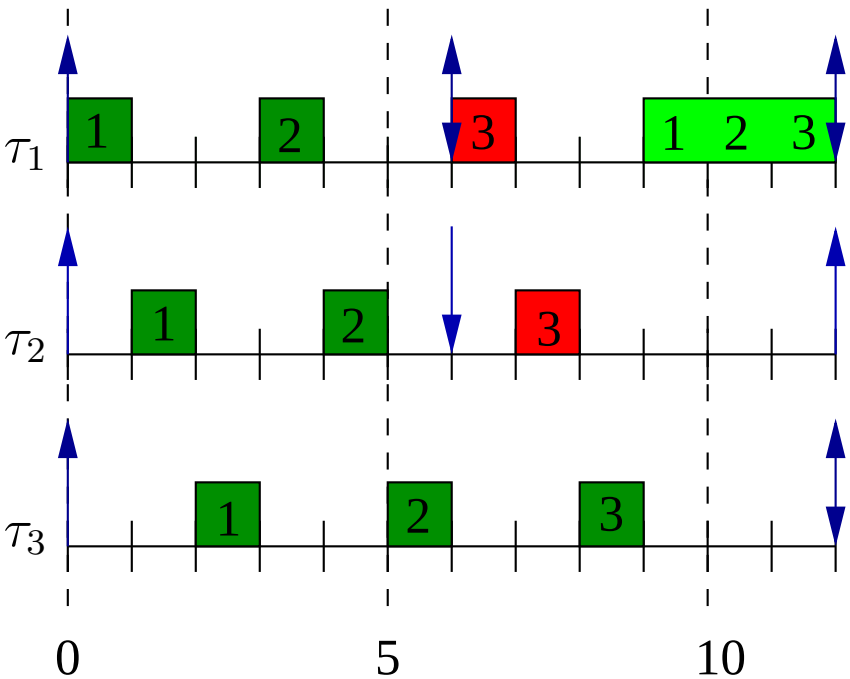
Une tâche périodique τ_i est définit par :

- date de première activation (release) : r_i
- pire temps d'exécution (WCET) : C_i
- période : T_i
- échéance relative : D_i
- échéance absolue de l'instance k : $d_{i,k}$
- convention : majuscules pour les durées, minuscules pour les dates
- ... (Modèle extensible)

Exemple

	Coût C_i	Période T_i	Échéance D_i
τ_1	3	6	6
τ_2	3	12	6
τ_3	3	12	12

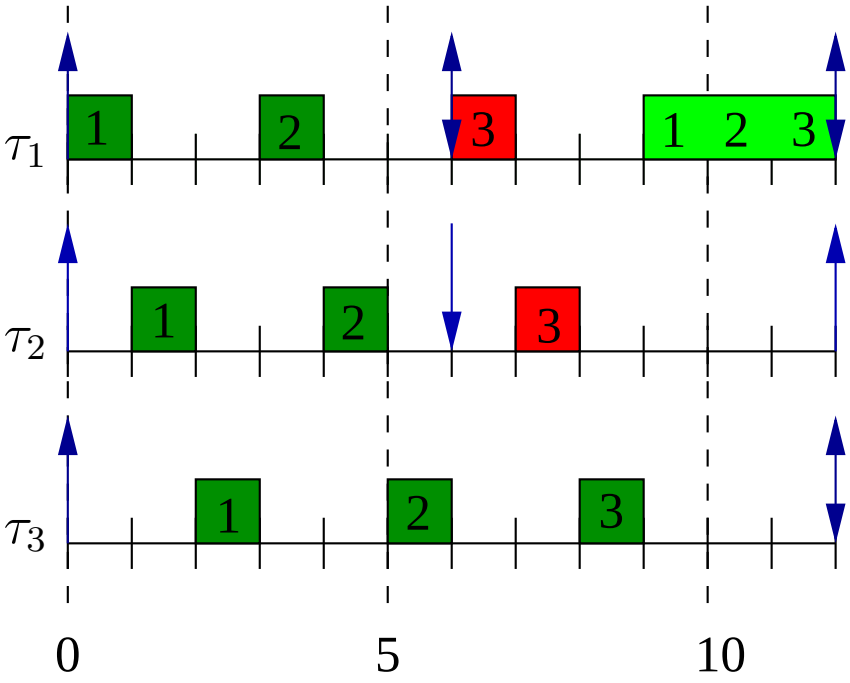
Round Robin



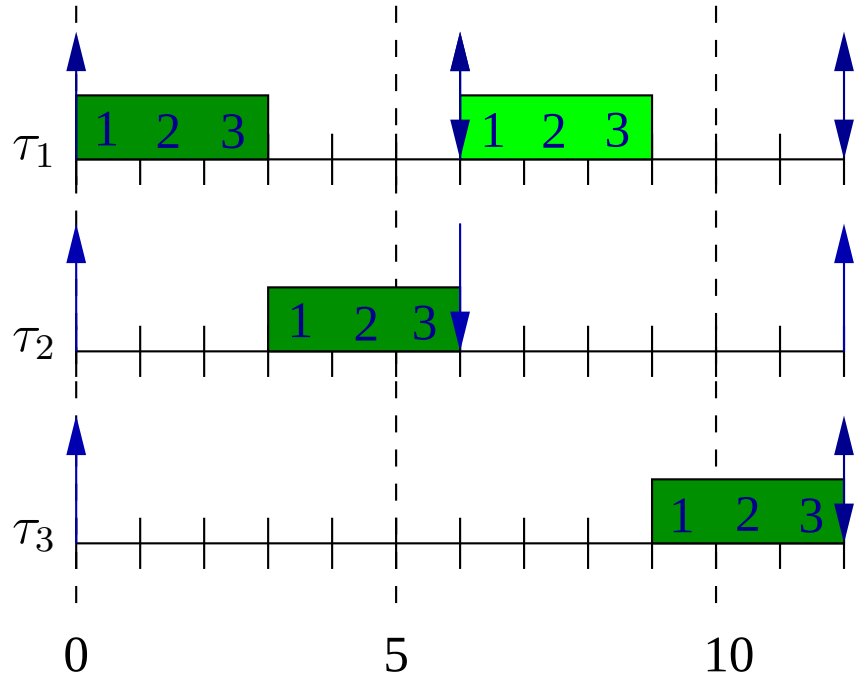
Exemple

	Coût C_i	Période T_i	Échéance D_i
τ_1	3	6	6
τ_2	3	12	6
τ_3	3	12	12

Round Robin



Priorités fixes



Ordonnancement temps-réel en ligne préemptif

(Mono Processeur)

Les instances des tâches (job) sont classées par priorité. À chaque instant, l'ordonnanceur attribue le processeur à la tâche la plus prioritaire.

- priorités fixes : les priorités des tâches sont fonction d'une constante (période, échéance, importance...) ;
- priorités dynamiques : les priorités sont fonction d'une variable (prochaine échéance, laxité, ...).

Qualité d'un algorithme d'ordonnancement ? ?

Ordonnancement temps-réel en ligne préemptif

(Mono Processeur)

Les instances des tâches (job) sont classées par priorité. À chaque instant, l'ordonnanceur attribue le processeur à la tâche la plus prioritaire.

- priorités fixes : les priorités des tâches sont fonction d'une constante (période, échéance, importance...) ;
- priorités dynamiques : les priorités sont fonction d'une variable (prochaine échéance, laxité, ...).

Qualité d'un algorithme d'ordonnancement ? ?

- optimalité (définitions à connaître)
- borne d'ordonnançabilité
- facilité d'implantation
- surcout (overhead) à l'exécution
- gigue, stabilité, temps de réponses moyens...

Les principaux algorithmes

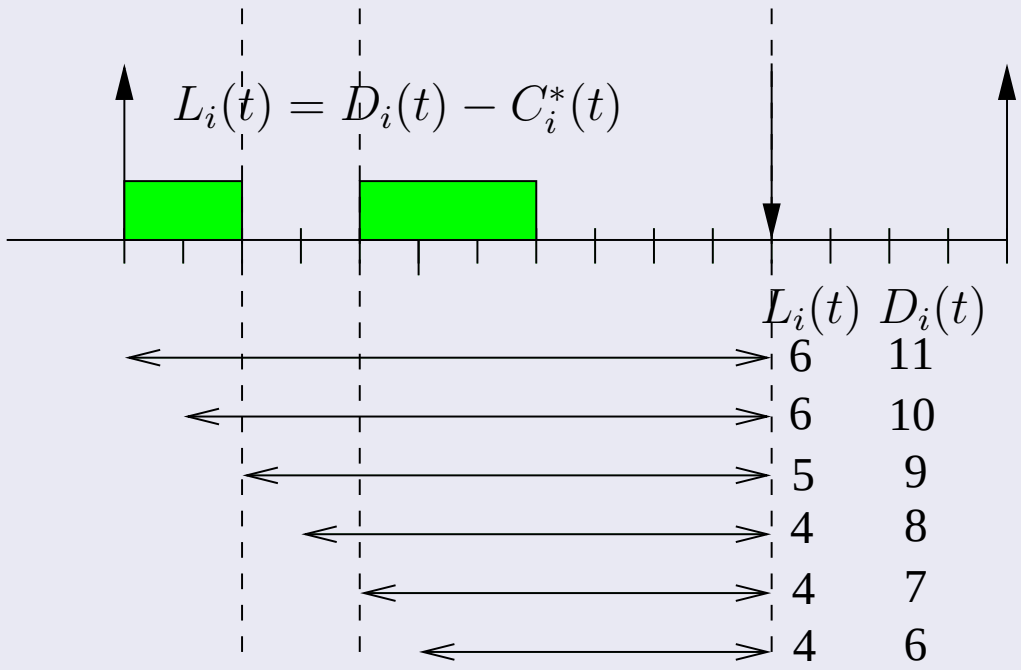
- Rate Monotonic (RM) : priorité à la plus petite période
- Deadline Monotonic (DM) : priorité à la plus petite échéance relative
- EDF : priorité à l'échéance absolue la plus proche
- LLF : priorité à la laxité la plus petite

Les principaux algorithmes

- Rate Monotonic (RM) : priorité à la plus petite période
- Deadline Monotonic (DM) : priorité à la plus petite échéance relative
- EDF : priorité à l'échéance absolue la plus proche
- LLF : priorité à la laxité la plus petite

attention : erreur dans la figure

Laxité

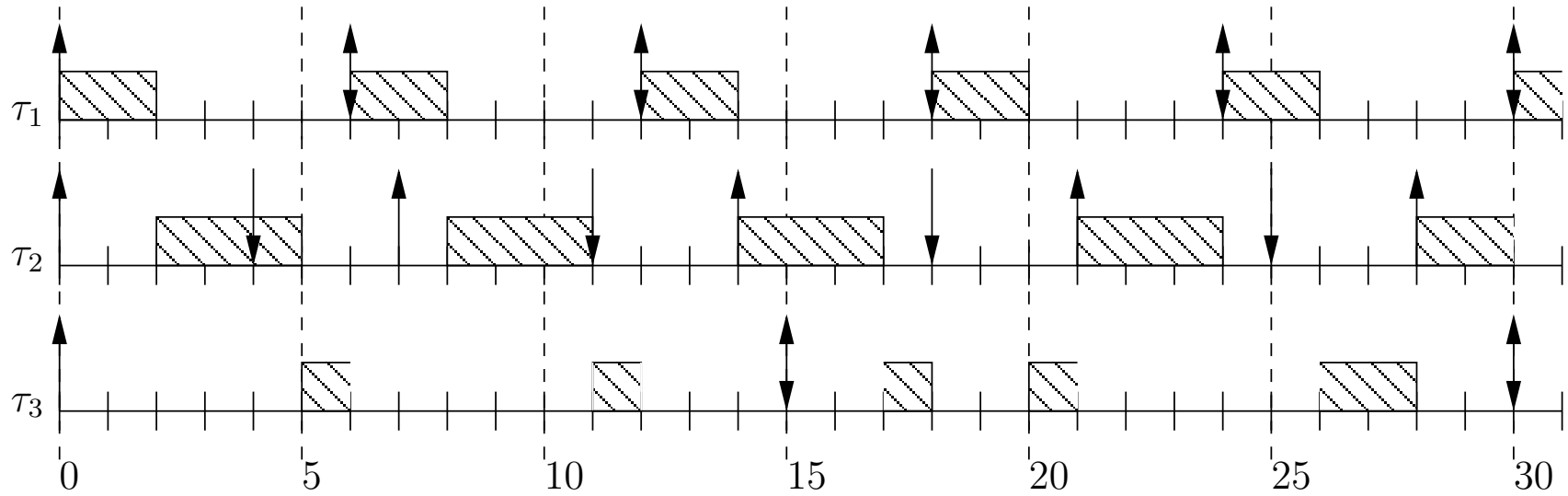


Exercice

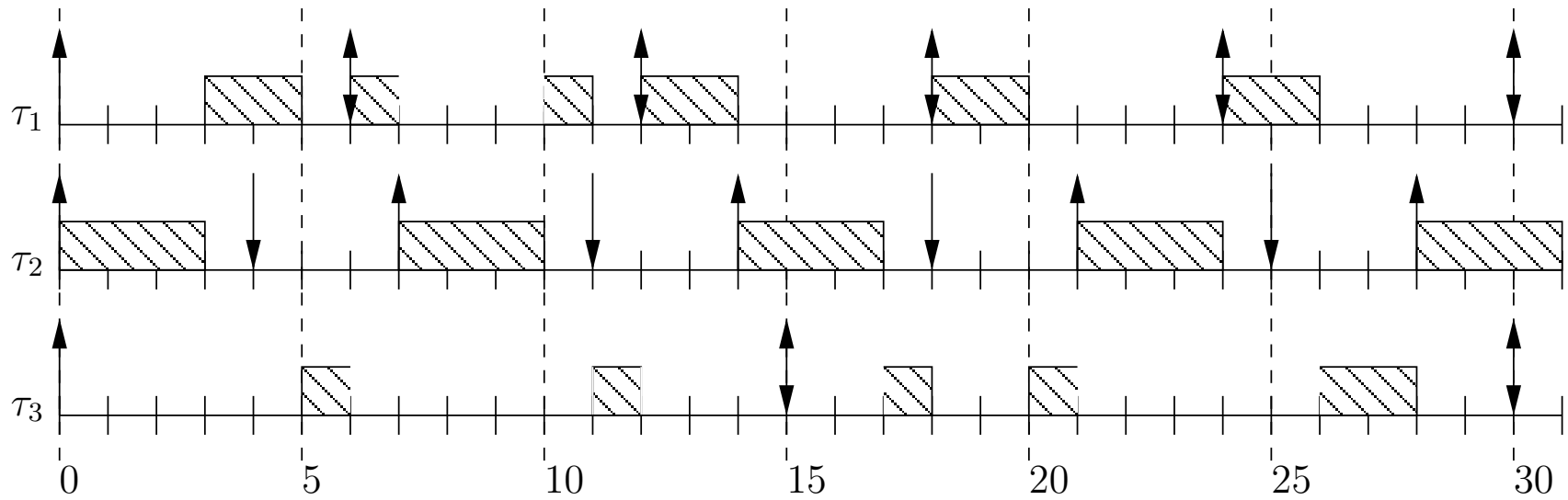
Donnez les chronogrames des ordonnancements produits par ces 4 algorithmes entre $t = 0$ et $t = 30$ pour le jeux de tâches :

	r_i	T_i	D_i	C_i
τ_1	0	6	6	2
τ_2	0	7	4	3
τ_3	0	15	15	3

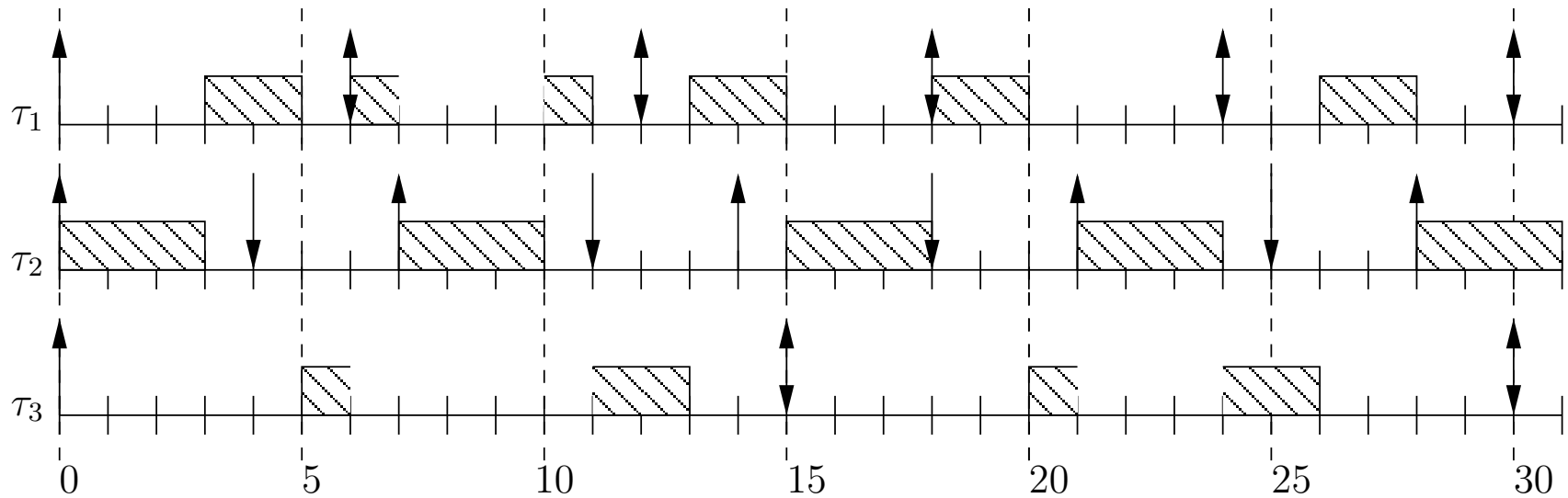
RM



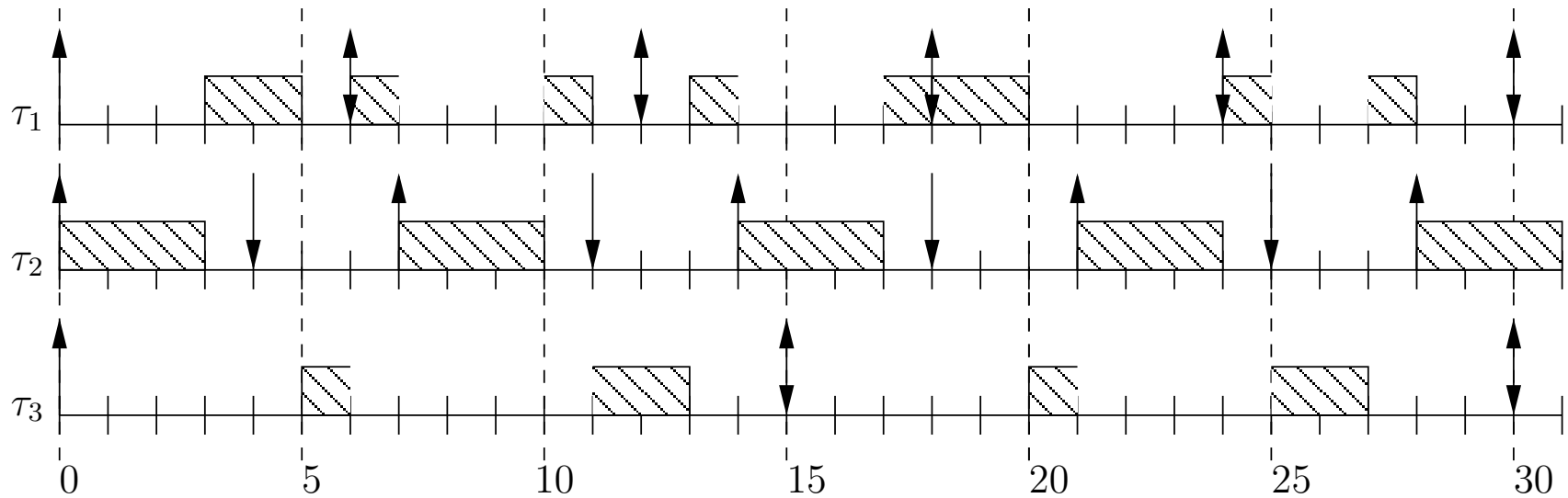
DM



EDF



LLF



Ordonnançabilité

$\neq$ Faisabilité

- **Faisabilité :** Soit un jeu de tâches, existe-t-il un ordonnançement qui respecte toutes les contraintes ?
- **Ordonnançabilité :** Soit un jeu de tâches et un algorithme d'ordonnancement, l'ordonnançement produit respecte-t-il toutes les contraintes ?

Plusieurs approches selon la criticité du système étudié et les moyens disponibles :

- condition suffisante mais pas nécessaire pour un contrôle d'admission en ligne,
- détection de fautes ou de surcharge,
- vérification exacte à l'aide de la théorie de l'analyse de faisabilité/ordonnançabilité.

Étude de la charge

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

l'étude de la charge peut permettre de conclure sur la faisabilité et/ou l'ordonnançabilité dans certains cas :

- $U \leq n(2^{\frac{1}{n}} - 1)$ est une condition suffisante d'ordonnançabilité pour un système à échéances sur requêtes ordonnancé en priorités fixes. Attention, ce n'est pas une condition nécessaire.
- $U \leq 1$ est une condition nécessaire et suffisante pour un système à échéances sur requêtes ordonnancé en EDF.

Malheureusement, lorsque $D_i \leq T_i$, ou pire lorsqu'on autorise $D_i > T_i$ (tâches dites ré-entrantes), les choses se compliquent !

Étude de l'ordonnançabilité d'un système à échéances sur requêtes

(priorités fixes)

- on l'a vu, $U \leq n(2^{\frac{1}{n}} - 1)$ est une condition suffisante d'ordonnançabilité, et $U > 1$ est une condition suffisante (évidente ?) de non ordonnançabilité
- cas $1 > U > n(2^{\frac{1}{n}} - 1)$?
  - étude de la demande : on cherche un instant inférieur à l'échéance où la demande cumulée est plus faible que le temps écoulé
  - étude du pire temps de réponse : le pire temps de réponse est-il plus petit que l'échéance ?
    - temps de réponse d'une instance R_i^j : temps entre la requête et la fin du traitement
    - pire temps de réponse d'une tâche ($WCRT_i$) : maximum des temps de réponse de chaque instance

Étude de la demande

Un système où les tâches réentrantes sont interdites ($\forall i, D_i \leq T_i$) est ordonnançable en priorités fixes ssi il existe un instant t dans l'intervalle $]0, Di]$ tel que $t = w_i(t)$

- avec $w_i(t) = \sum_{k \leq i} \left\lceil \frac{t}{T_k} \right\rceil C_k$
- algorithme récursif : calcul de $t_1 = w_i(0)$, puis $t_2 = w_i(t_1)$, ..., $t_n = w_i(t_{n-1})$
- on s'arrête si l'on dépasse l'échéance ou si l'on trouve t tel que $t = w_i(t)$.

Étude de la demande

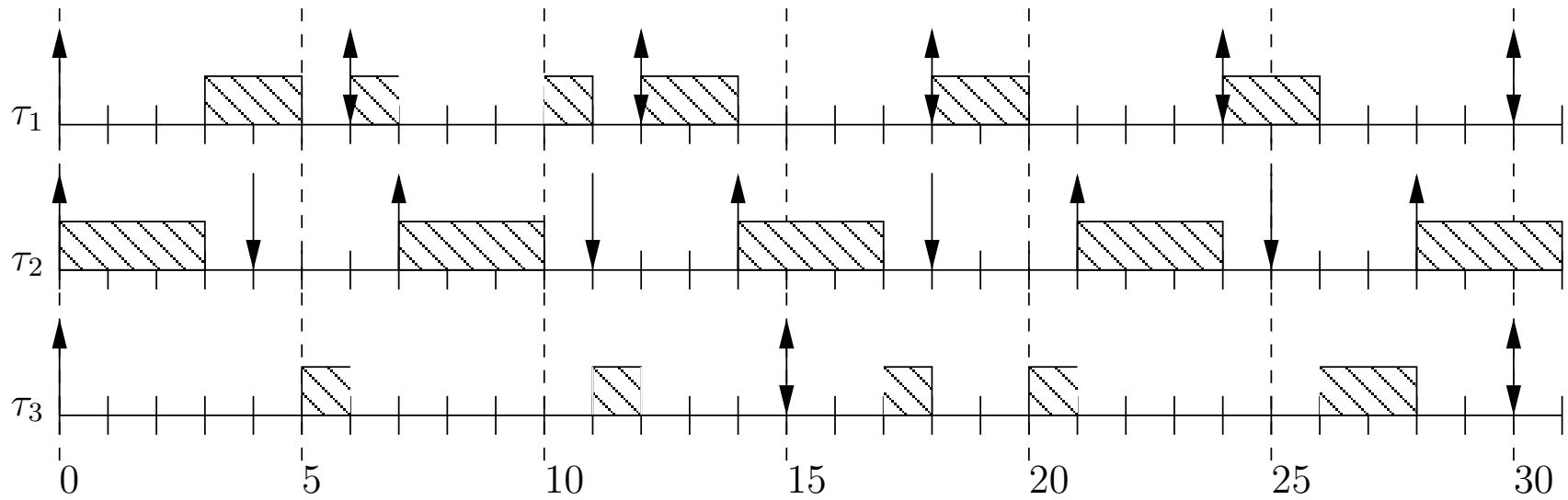
Un système où les tâches réentrantes sont interdites ($\forall i, D_i \leq T_i$) est ordonnançable en priorités fixes ssi il existe un instant t dans l'intervalle $]0, Di]$ tel que $t = w_i(t)$

- avec $w_i(t) = \sum_{k \leq i} \left\lceil \frac{t}{T_k} \right\rceil C_k$
- algorithme récursif : calcul de $t_1 = w_i(0)$, puis $t_2 = w_i(t_1)$, ..., $t_n = w_i(t_{n-1})$
- on s'arrête si l'on dépasse l'échéance ou si l'on trouve t tel que $t = w_i(t)$.

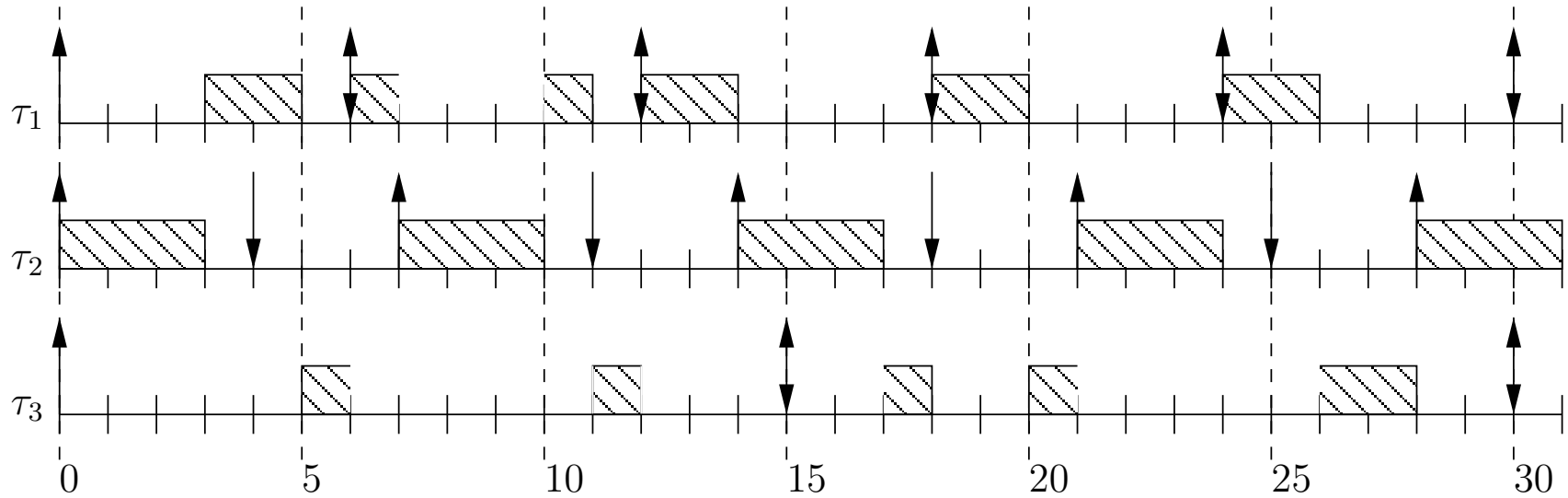
Exercice

- étudiez la demande au niveau 2 de l'exemple RM précédant
- étudiez la demande au niveau 2 de l'exemple DM précédant
- étudiez la demande au niveau 3 de l'exemple RM précédant
- la demande au niveau 3 avec DM est-elle différente ?

DM



DM



t	8	13	15
$w_3(t)$	13	15	18

Limites

- Ce test ne permet que de conclure sur l'ordonnançabilité, il ne donne pas d'autres informations. Il peut être intéressant de calculer les temps de réponses, pour avoir une idée du comportement de la tâche (gigue, temps de réponse moyen...)
- Ce test ne marche plus pour le cas $D_i > T_i$
- essayez sur l'exemple suivant pour vous en convaincre :

	r_i	C_i	T_i	D_i	Priorité P_i
τ_1	0	4	8	10	forte
τ_2	0	3	6	8	basse

  Que vaut $w_2(7)$? Cela a t il un sens de le comparer à D_2 ?
- ce système n'est d'ailleurs pas ordonnançable, mais il faut attendre $t = 21$ pour voir la deuxième instance de la tâche τ_2 dépasser son échéance (la pire instance n'est plus la première !)

Calcul de temps de réponse

- calcul récursif très similaire à l'étude de la demande
- une tache n'est retardée que par les plus prioritaires
- on cherche à calculer le tdr de l'instance j de la tâche τ_i, en numérotant les instances à partir de 1. Sa date de terminaison, notée F_i^j, est donnée par l'équation :

$$F_i^j = \min_{t>0}\{t = w_{i-1}(t) + j * C_i\} \tag{1}$$

- Son tdr, R_i^j, est alors la différence entre sa date de terminaison et sa date d'activation. Il est donné par l'équation :

$$R_i^j = F_i^j - (r_i + (j-1)T_i) \tag{2}$$

Busy Period

période occupée

- une période occupée de niveau i est un intervalle entre deux instants d'inactivité au niveau de priorité i du processeur $(w_i(t) = t)$
- pour calculer celle qui commence à $t = 0$ (la pire dans le scénario synchrone) on réalise l'étude de la demande, mais on ne s'arrête plus si l'échéance est dépassée

Il suffit d'étudier une tâche durant sa busy period synchrone pour rencontrer son pire temps de réponse.

Busy Period

période occupée

- une période occupée de niveau i est un intervalle entre deux instants d'inactivité au niveau de priorité i du processeur $(w_i(t) = t)$
- pour calculer celle qui commence à $t = 0$ (la pire dans le scénario synchrone) on réalise l'étude de la demande, mais on ne s'arrête plus si l'échéance est dépassée

Il suffit d'étudier une tâche durant sa busy period synchrone pour rencontrer son pire temps de réponse.

Exercice

- calculez la busy period de niveau 2 (bp_2) de l'exemple précédant
- calculez Q_i le nombre d'activations de τ_2 durant bp_2
- calculez les Q_i premiers temps de réponse de τ_2
- conclure

Faisabilité sous EDF

- $D_i = T_i$: ordonnancable ssi $\forall t \geq 0, t \geq \sum_{i=1}^{n} \left\lfloor \frac{t}{T_i} \right\rfloor C_i$
- équivalent au test necessaire et suffisant $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$
- $D_i \leq T_i$: ordonnancable ssi
  $\forall t \geq 0, t \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1 \right)_0 C_i$
- on se limite à vérifier la propriété lors des échéances contenues dans la busy-period synchrone

Quelques métriques et remarques sur DM vs EDF

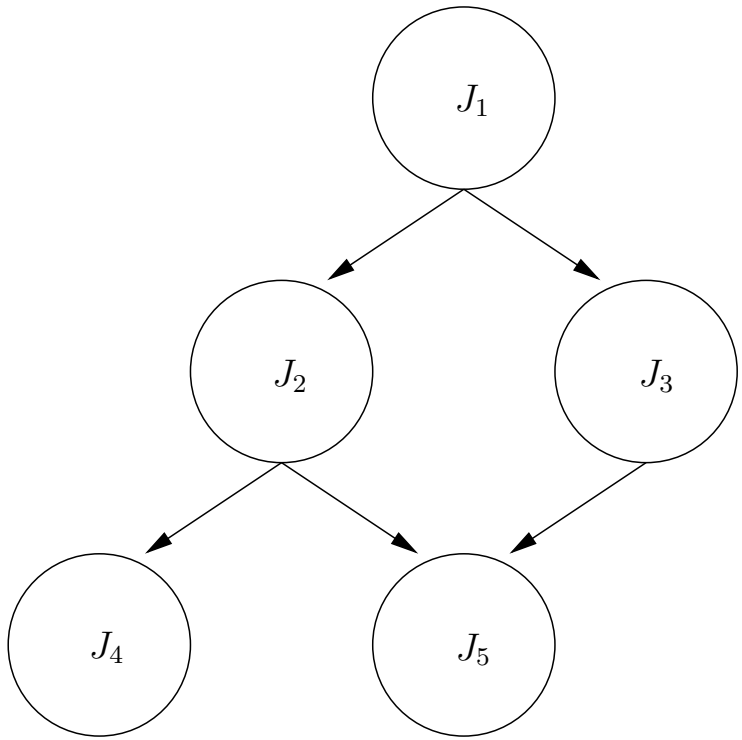
- DM est optimal pour la classe des ordonnanceurs à priorités fixes préemptif en ligne
- EDF est optimal pour la classe des ordonnanceurs préemptif en ligne
- cela veut juste dire que si un système n'est pas ordonnançable avec DM/EDF, il n'est pas ordonnançable avec un autre algorithme de la même classe
- cela ne veut pas dire que DM/EDF est le meilleur algorithme pour répondre à notre problème (autres critères fonction de l'application visée)

Autres modèles de tâches

- Jusqu'à présent, on a fait l'hypothèse de tâches indépendantes les unes des autres, mais il peut exister d'autres contraintes :
  - relations de précédence entre les tâches
  - partage de ressources avec protection par sémaphores entre les tâches
- Des tâches non périodiques doivent également pouvoir être traitée :
  - temps d'inter arrivée de deux instances minimal et étude pire cas (modèle sporadique)
  - traitements encapsulés dans un serveur avec des ressources réservées
  - traitement sur le temps libre du système (background ou slack stealing)

Tâches à contraintes de précédence

On peut représenter les contraintes par un DAG :



- J_1 précède toutes les autres tâches
- J_4 succède à J_2 et donc à J_1
- J_5 succède à J_2 et J_3, et donc à J_1

Contraintes de précédence

- on va transformer notre jeu de tâche Ψ en un nouveau jeu de tâches indépendantes Ψ^* indépendantes
- il faut assurer la propriété : Ψ^* ordonnançable ssi Ψ ordonnançable.
- il faut modifier pour chaque tâche sa date d'activation et son échéance
- soit J_a prédécesseur direct de J_b
  - J_b ne doit pas commencer avant r_b
  - J_b ne doit pas commencer avant la fin au plus tôt de J_a

  On a donc $r_b^* = \max(r_b, r_a + C_a)$
  - J_a doit terminer avant son échéance
  - J_a doit terminer avant le début au plus tard de J_b

  On a donc $d_a^* = \min(d_a, d_b - C_b)$

Modification des tâches

1. Pour chaque tâche sans prédécesseur : $r_i^* = r_i$
2. Choisir J_i parmi les tâches non traitées dont tous les prédécesseurs immédiats ont été traités, si inexistant GOTO 5
3. $r_i^* = \max\left(r_i, \max_{J_h \to J_i}(r_h + C_h)\right)$
4. GOTO 2
5. Pour chaque tâche sans successeur : $d_i^* = d_i$
6. Choisir J_i parmi les tâches non traitées dont tous les successeurs immédiats ont été traités, si inexistant FIN
7. $d_i^* = \min\left(d_i, \min_{J_i \to J_h}(d_h - C_h)\right)$
8. GOTO 7

Partage de ressources

- lorsque deux tâches concurrentes accèdent à des ressources, il faut protéger l'accès à cette ressource par un verrou (sémaphore, mutex, ...)
- pour accéder à une ressource, une tâche doit obtenir le verrou associé
- une seule tâche peut posséder un verrou donné
- les tâches qui demandent un verrou sont donc bloquées si le verrou a été obtenu par une autre tâches
- il faut donc faire attention avant d'accorder le verrou à une tâche

Enjeux et solutions en priorité fixe

- Borner les inversions de priorité
- Empêcher les interblocages (deadlocks)
- Éviter les chaînes de blocage

Enjeux et solutions en priorité fixe

- Borner les inversions de priorité

une tâche s'exécute alors qu'une tâche plus prioritaire est bloquée

- Empêcher les interblocages (deadlocks)
- Éviter les chaînes de blocage

Enjeux et solutions en priorité fixe

- Borner les inversions de priorité
- Empêcher les interblocages (deadlocks)

une première tâche attend un verrou possédé par une autre tâche qui attend un verrou possedé par la première

- Éviter les chaînes de blocage

Enjeux et solutions en priorité fixe

- Borner les inversions de priorité
- Empêcher les interblocages (deadlocks)
- Éviter les chaînes de blocage

Une tâche est bloquée plusieurs fois durant une instance

Enjeux et solutions en priorité fixe

- Borner les inversions de priorité
- Empêcher les interblocages (deadlocks)
- Éviter les chaînes de blocage

Trois algorithmes :

- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)
- Priority Ceiling Emulation (PCE)

Enjeux et solutions en priorité fixe

- Borner les inversions de priorité
- Empêcher les interblocages (deadlocks)
- Éviter les chaînes de blocage

Trois algorithmes :

- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)
- Priority Ceiling Emulation (PCE)

Enjeux et solutions en priorité fixe

- Borner les inversions de priorité
- Empêcher les interblocages (deadlocks)
- Éviter les chaînes de blocage

Trois algorithmes :

- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)
- Priority Ceiling Emulation (PCE)

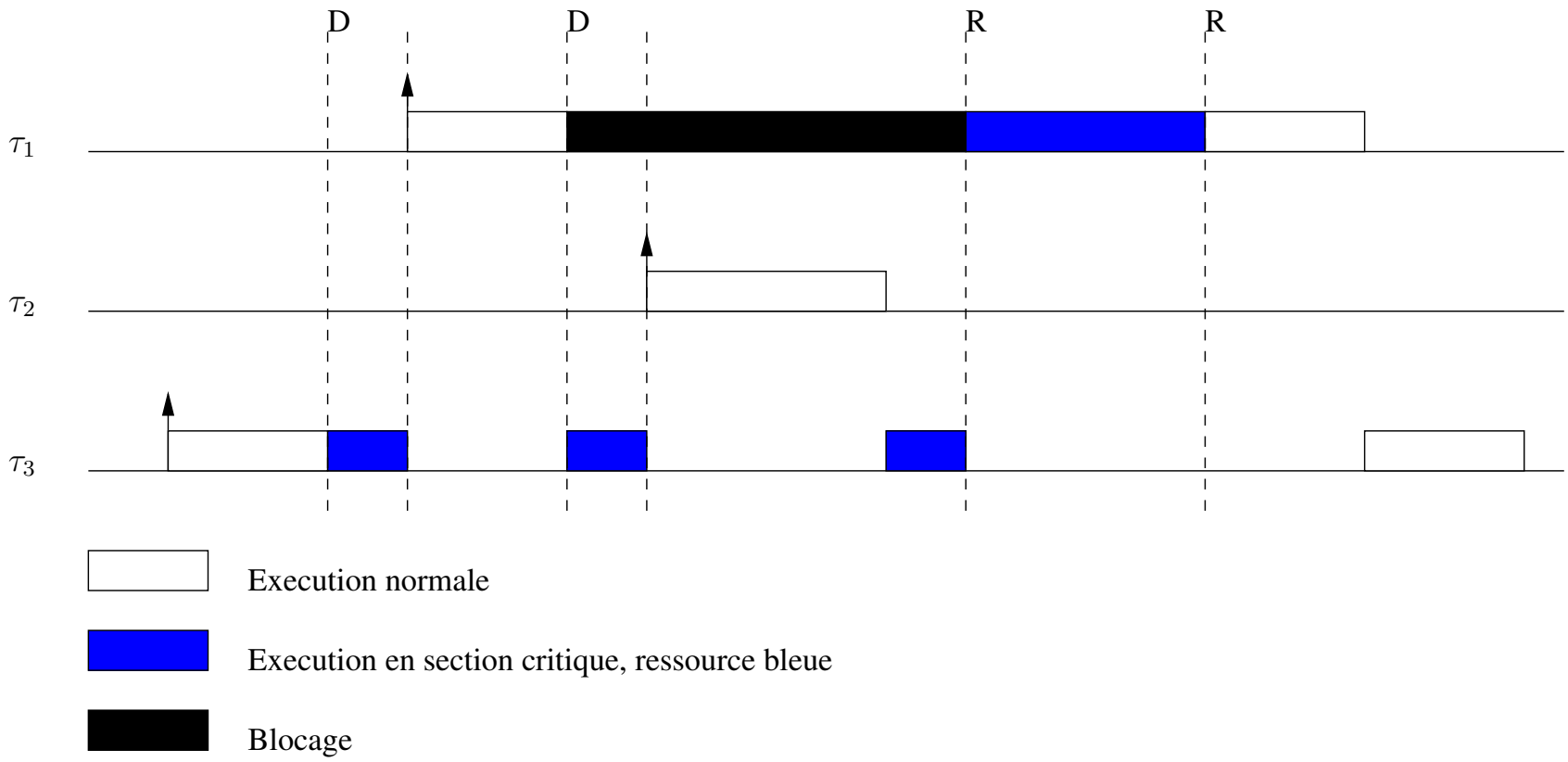
Enjeux et solutions en priorité fixe

- Borner les inversions de priorité
- Empêcher les interblocages (deadlocks)
- Éviter les chaînes de blocage

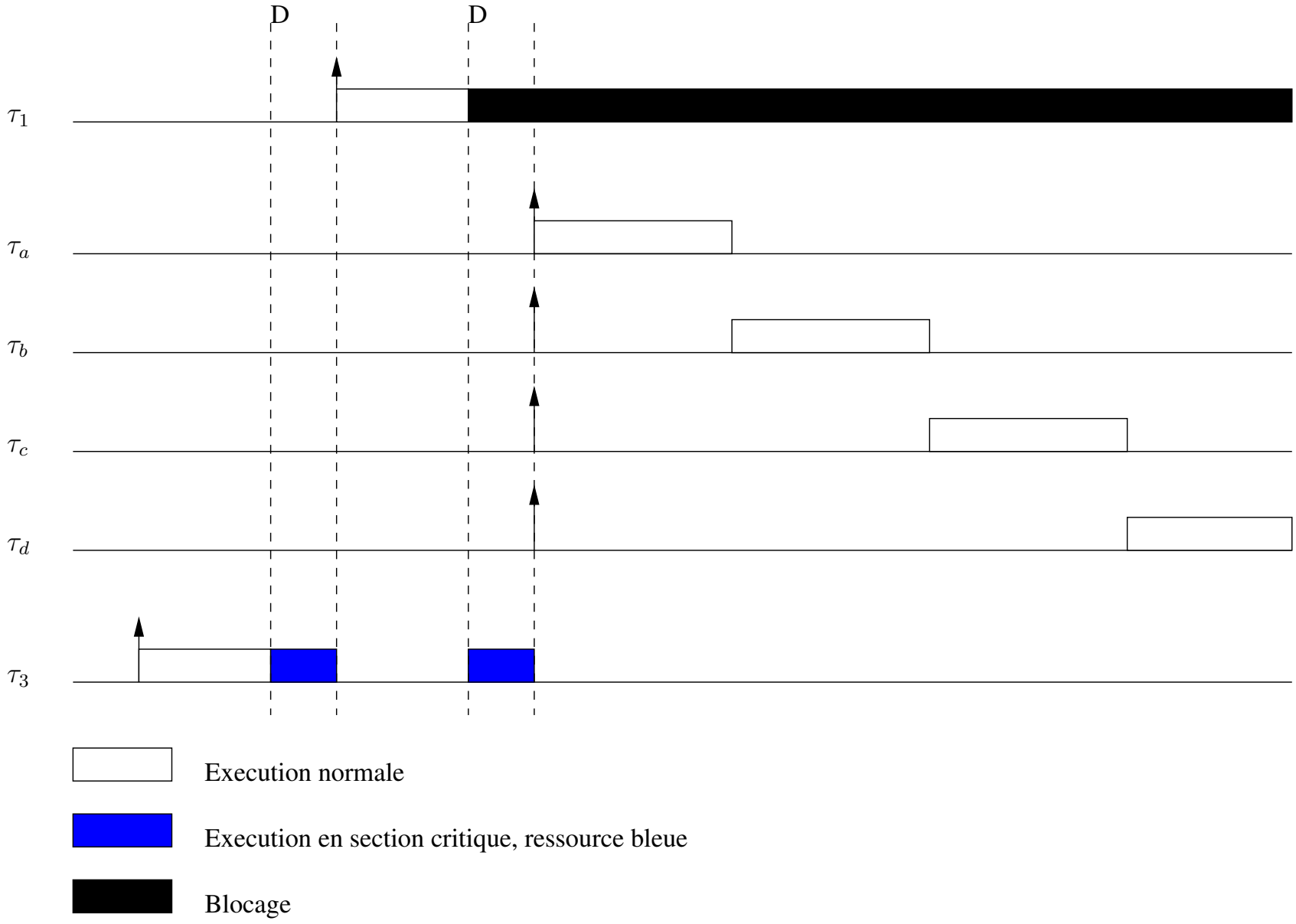
Trois algorithmes :

- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)
- Priority Ceiling Emulation (PCE)

Inversion non bornée



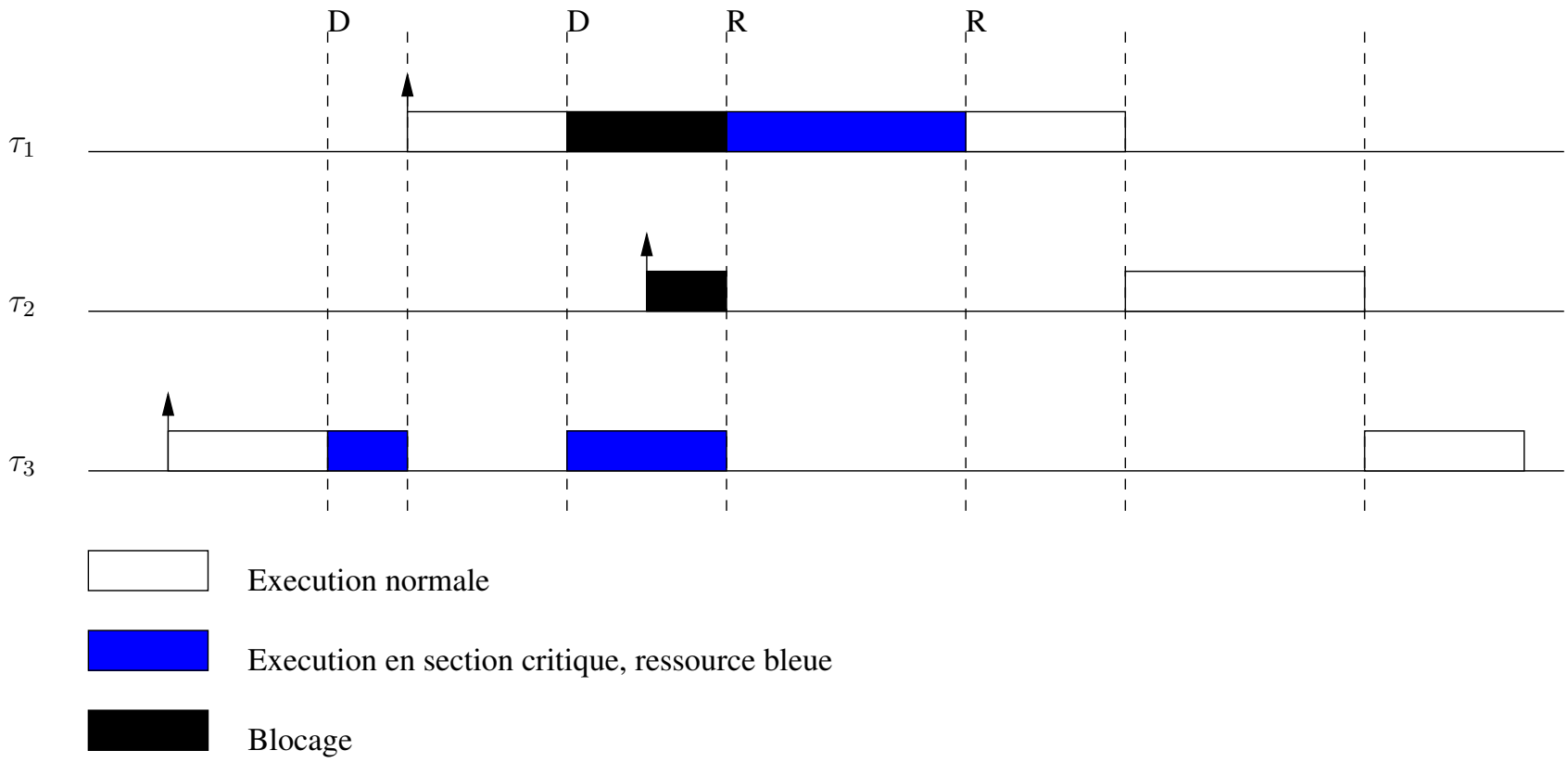
Inversion non bornée



Priority Inheritance Protocol (PIP)

- Lorsqu'une tâche τ_1 demande l'accès à une ressource utilisée par une tâche moins prioritaire τ_2, τ_1 se bloque et τ_2 hérite de la priorité de τ_1
- L'héritage de priorité est transitif, lorsque τ_3 bloque τ_2 et que τ_2 bloque τ_1, alors τ_3 hérite de la priorité de τ_1 via τ_2 (avec $P_1 > P_2 > P_3$)
- Lorsque τ_2 libère la ressource elle récupère sa priorité initiale
- Lorsque la ressource est libérée, elle est allouée à la tâche la plus prioritaire (la file d'attente de la ressource est ordonnée par priorité décroissantes)

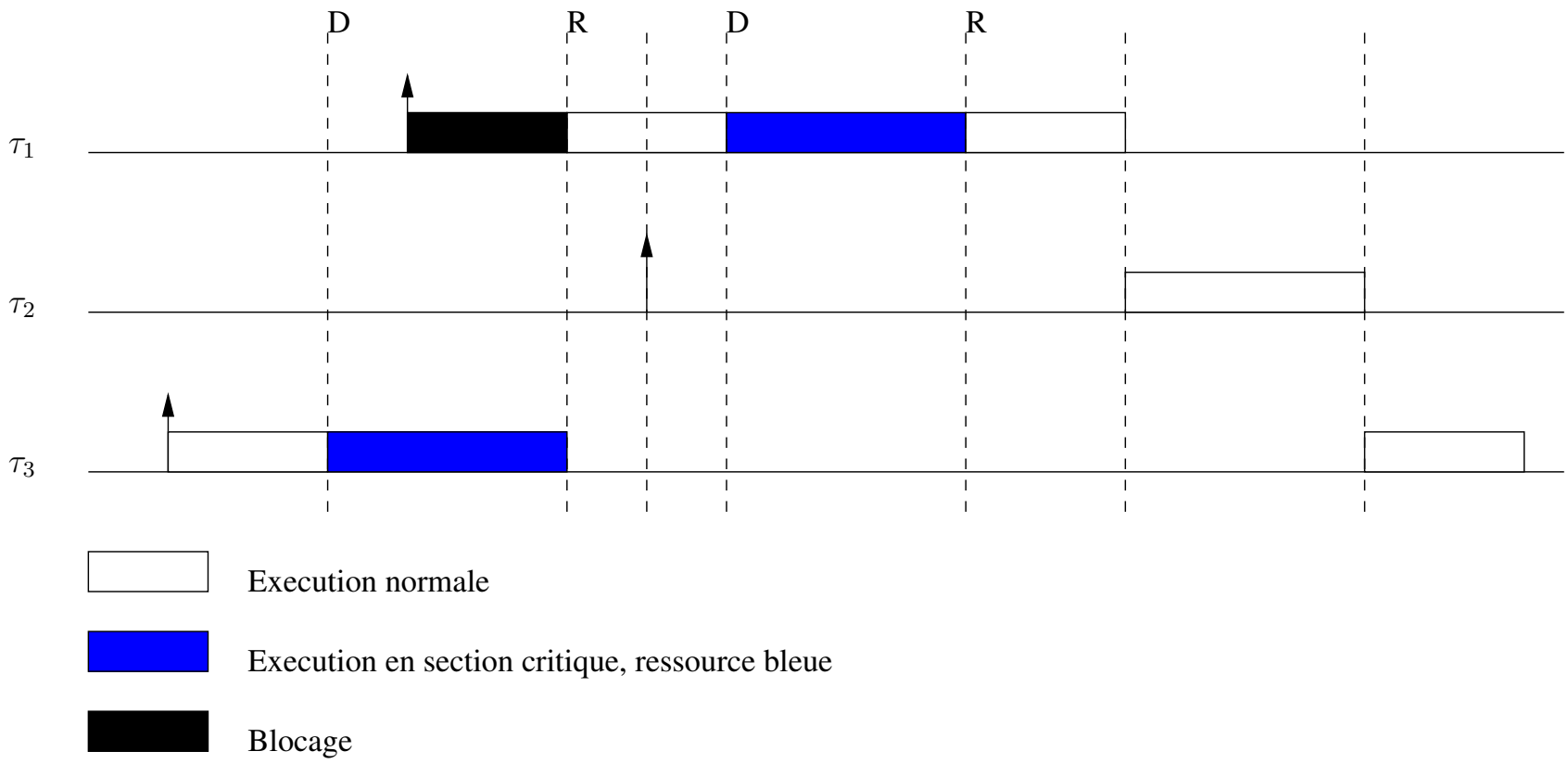
Priority Inheritance Protocol (PIP)



Priority Ceiling Emulation (PCE)

- Une priorité plafond est calculée statiquement pour chaque ressource
- Lorsqu'une tâche entre en section critique, elle prend immédiatement la priorité plafond de la ressource

Priority Ceiling Emulation (PCE)

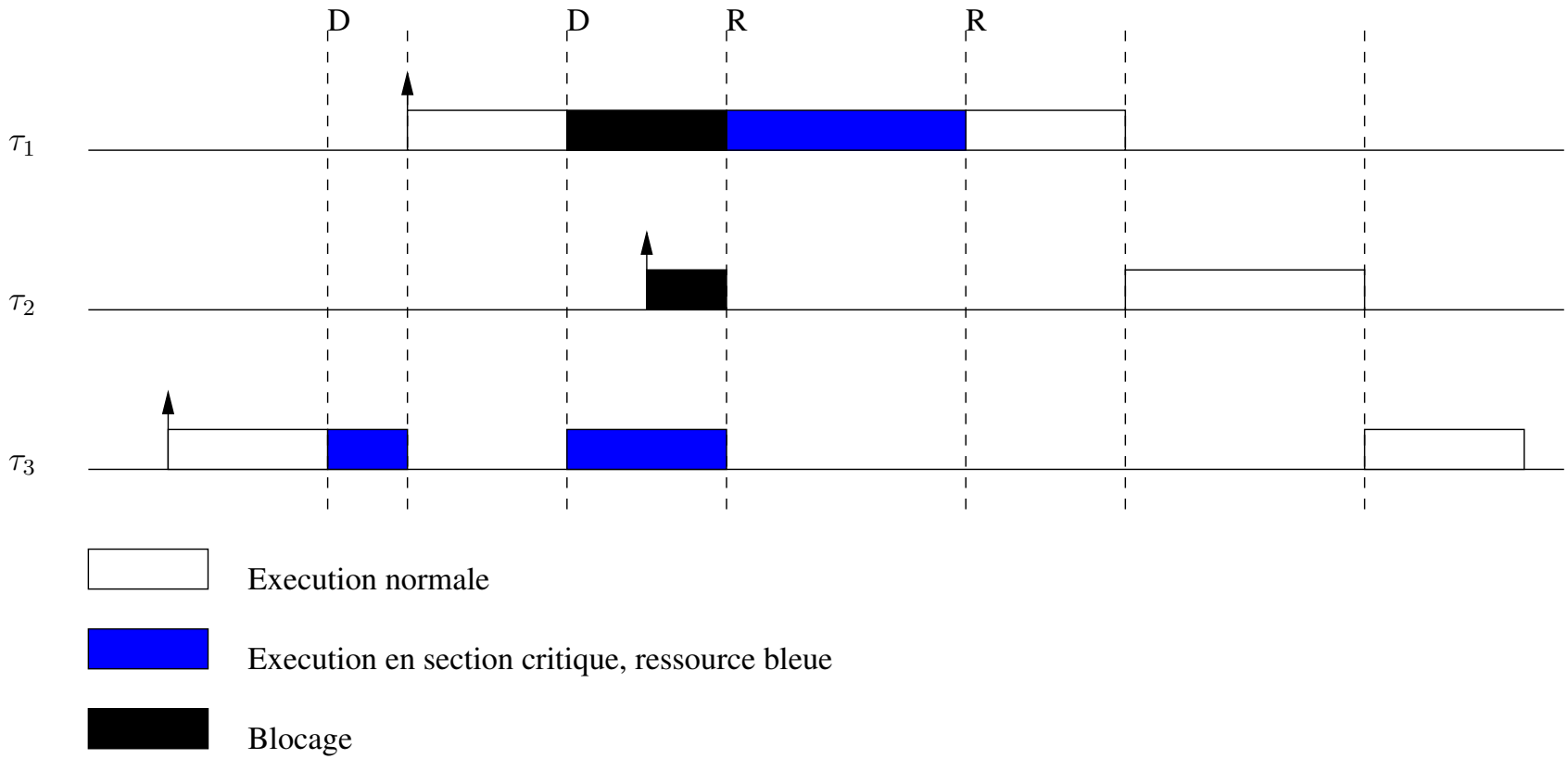


Priority Ceiling Protocol (PCP)

- Une priorité plafond est calculée statiquement pour chaque ressource
- Une tâche ne peut entrer en section critique que si sa priorité est supérieure à toutes les priorités plafond des ressources en cours d'utilisation
- Il y a héritage de priorité comme pour PIP
- Si une tâche est déjà en section critique, elle peut acquérir d'autres verrous

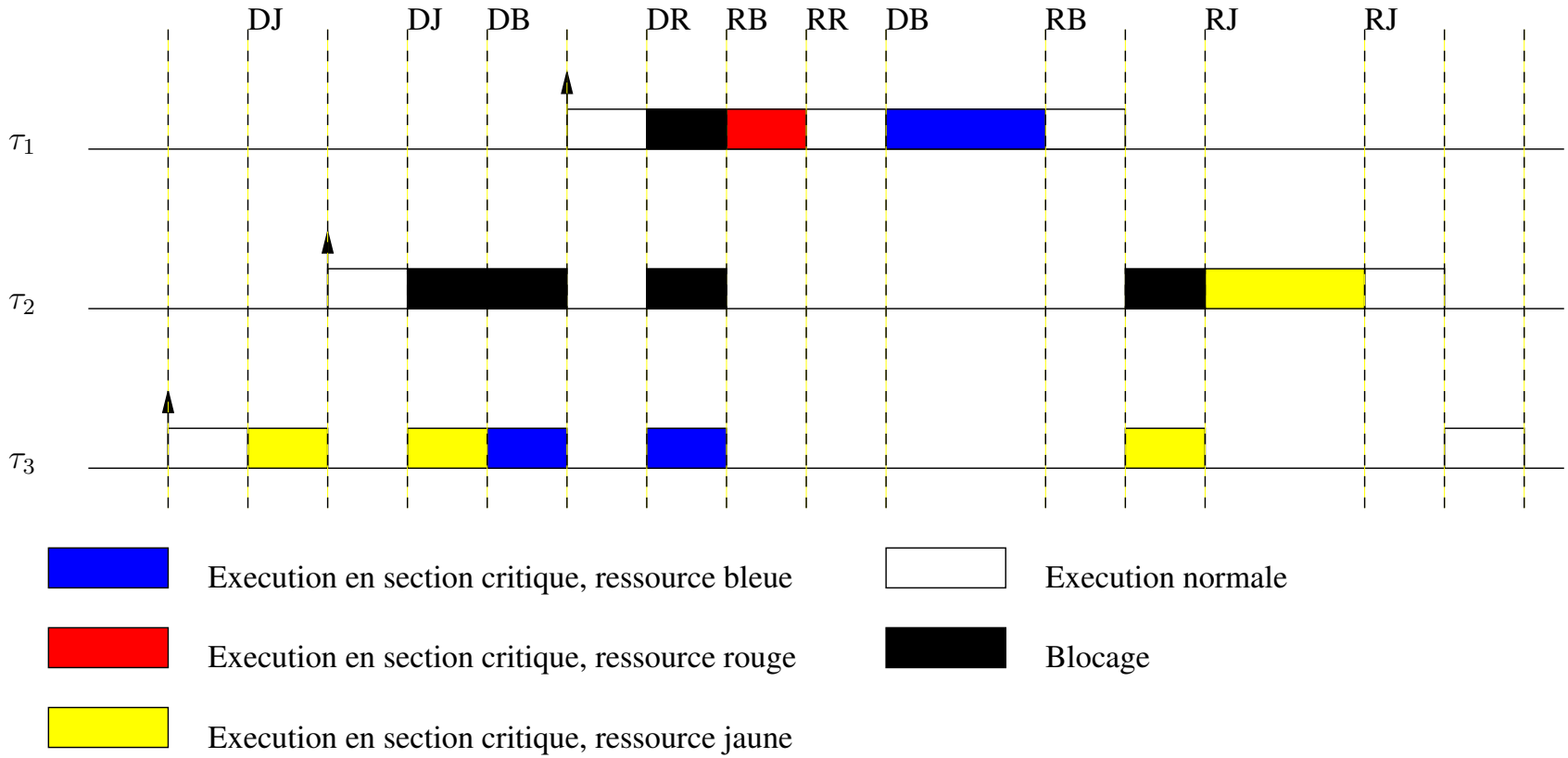
Priority Ceiling Protocol (PCP)

Exemple 1



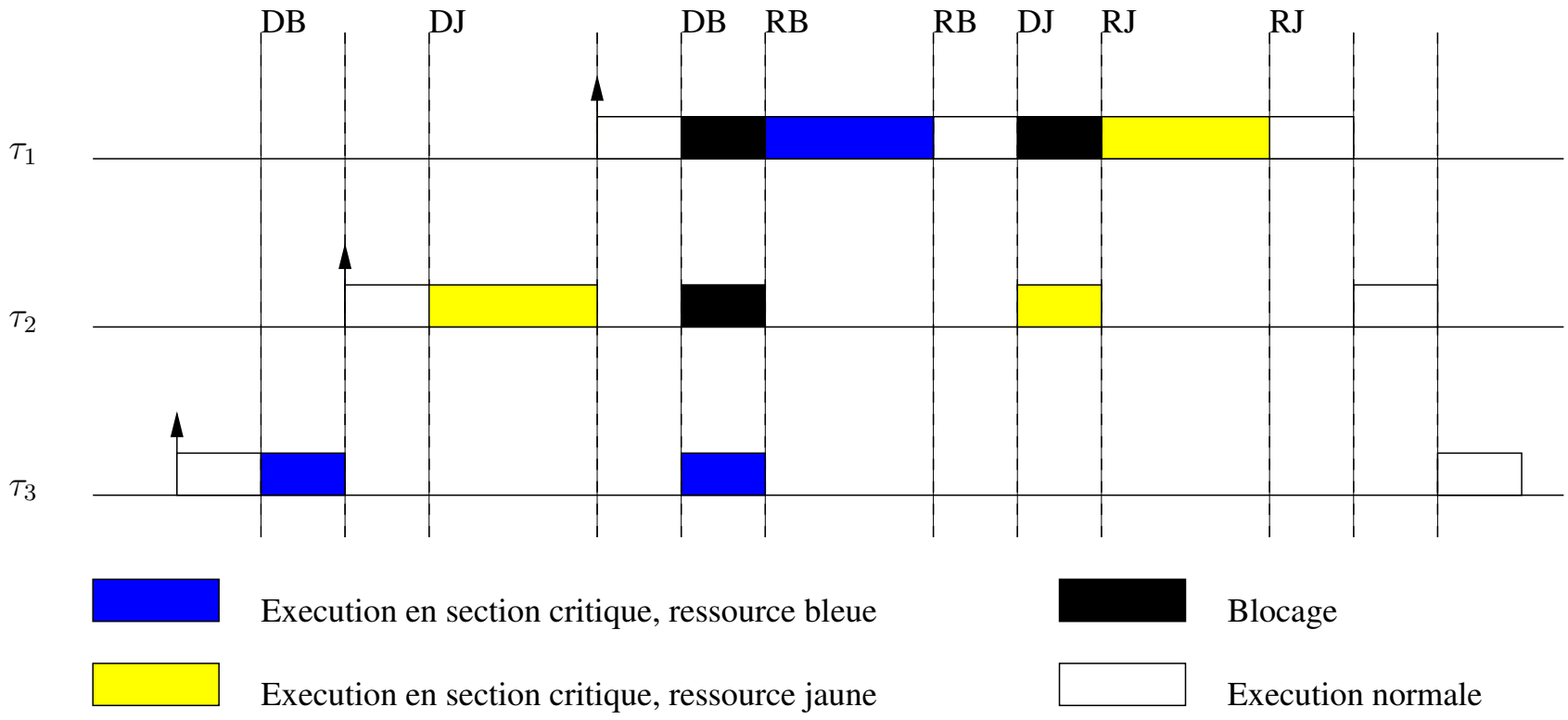
Priority Ceiling Protocol (PCP)

Exemple 2



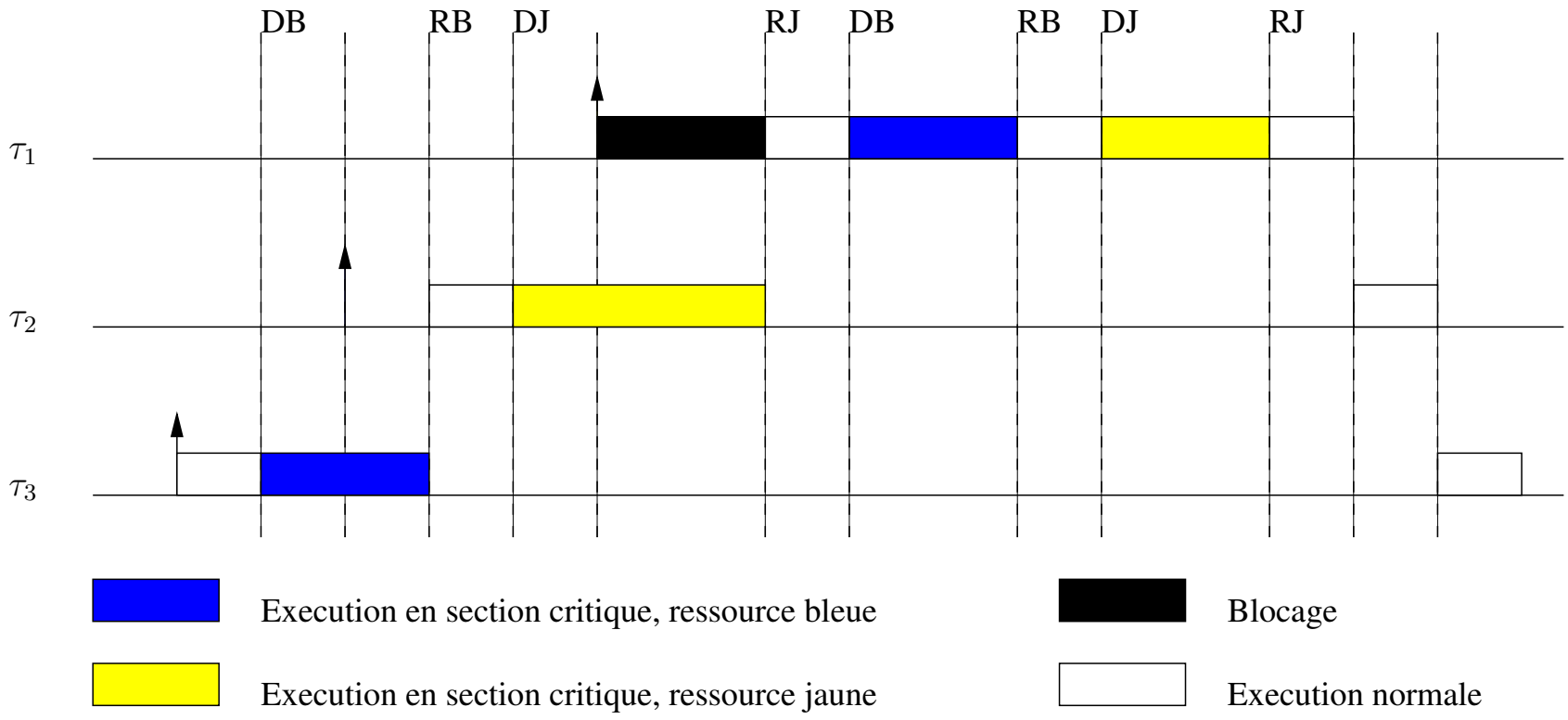
Chaîne de blocage

PIP



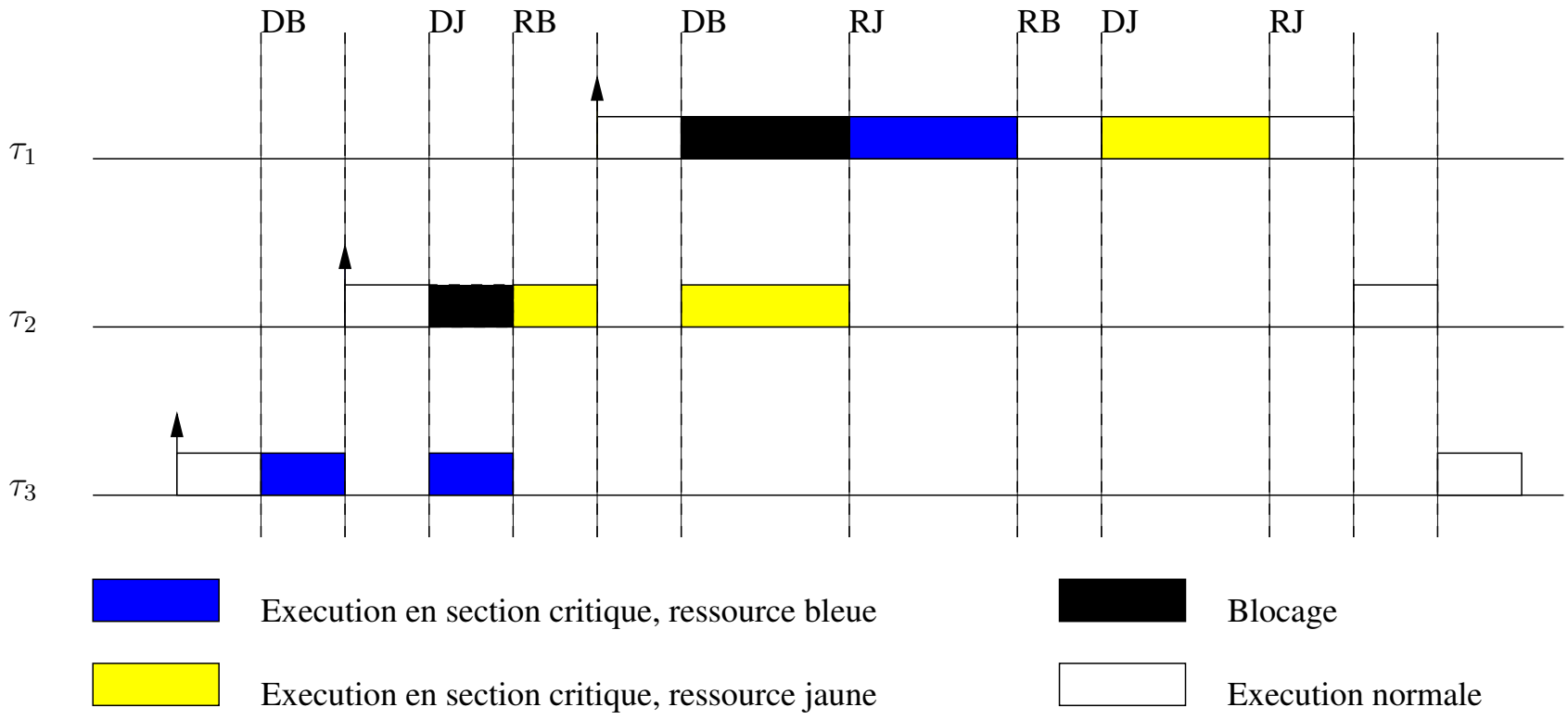
(Pas de) Chaîne de blocage

PCE



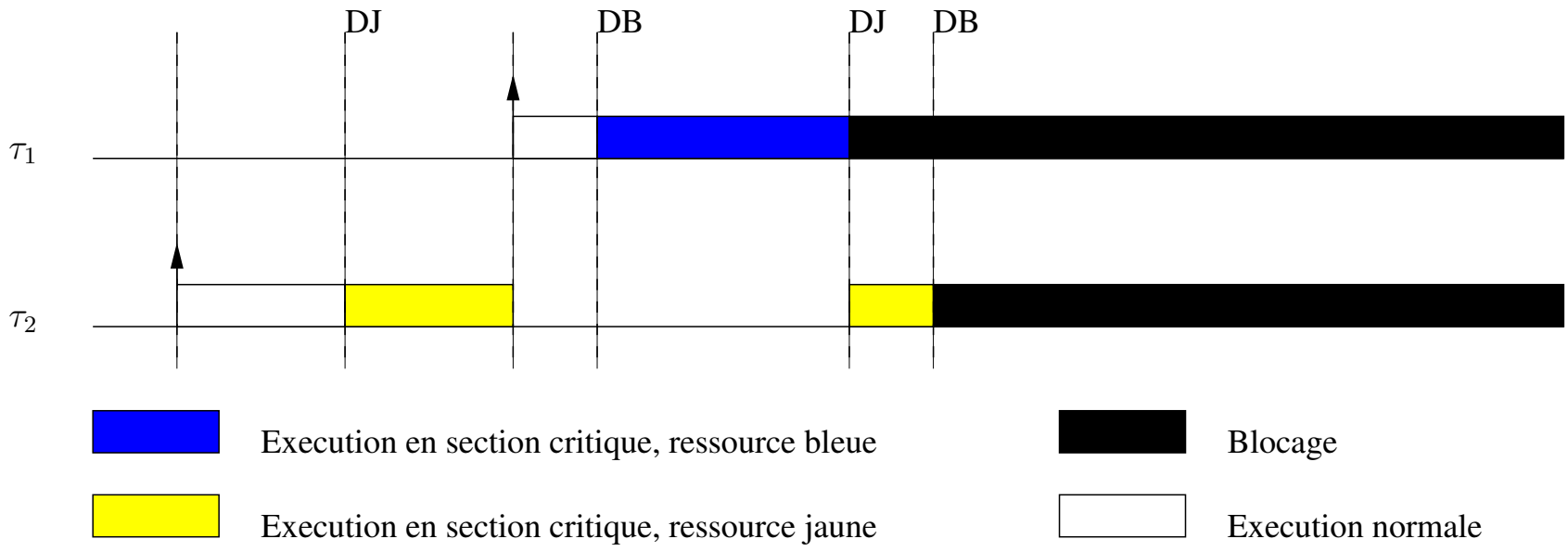
(Pas de) Chaîne de blocage

PCP



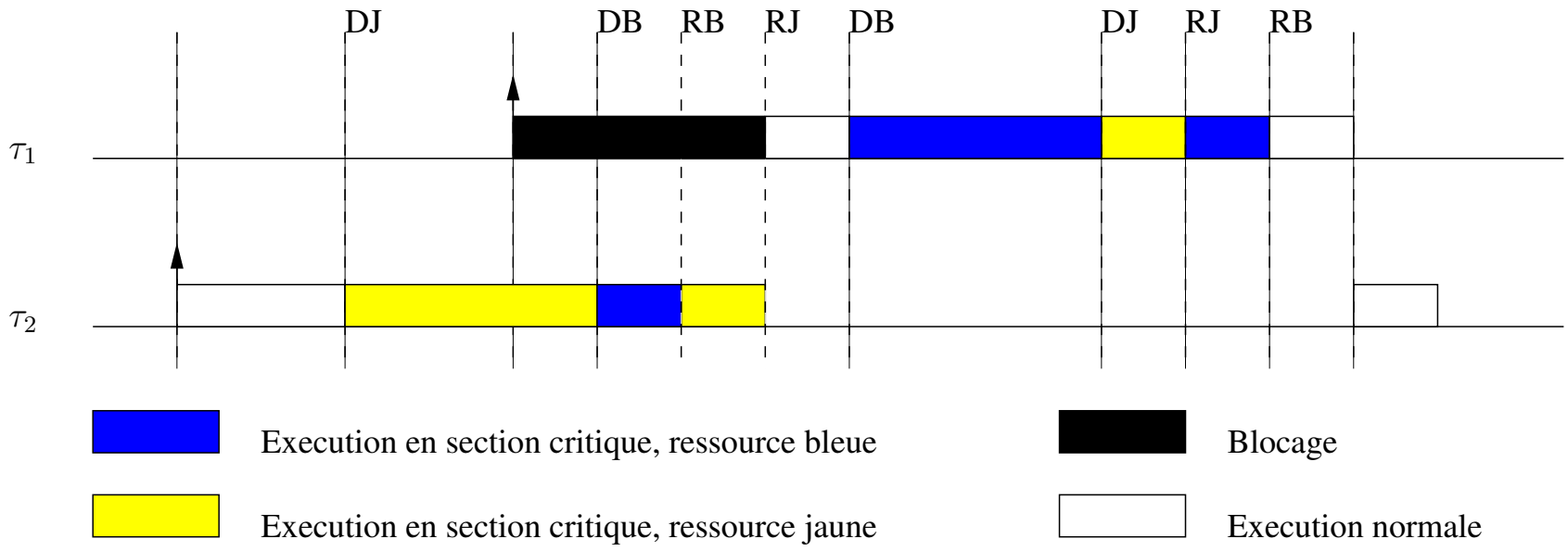
Interblocage

PIP



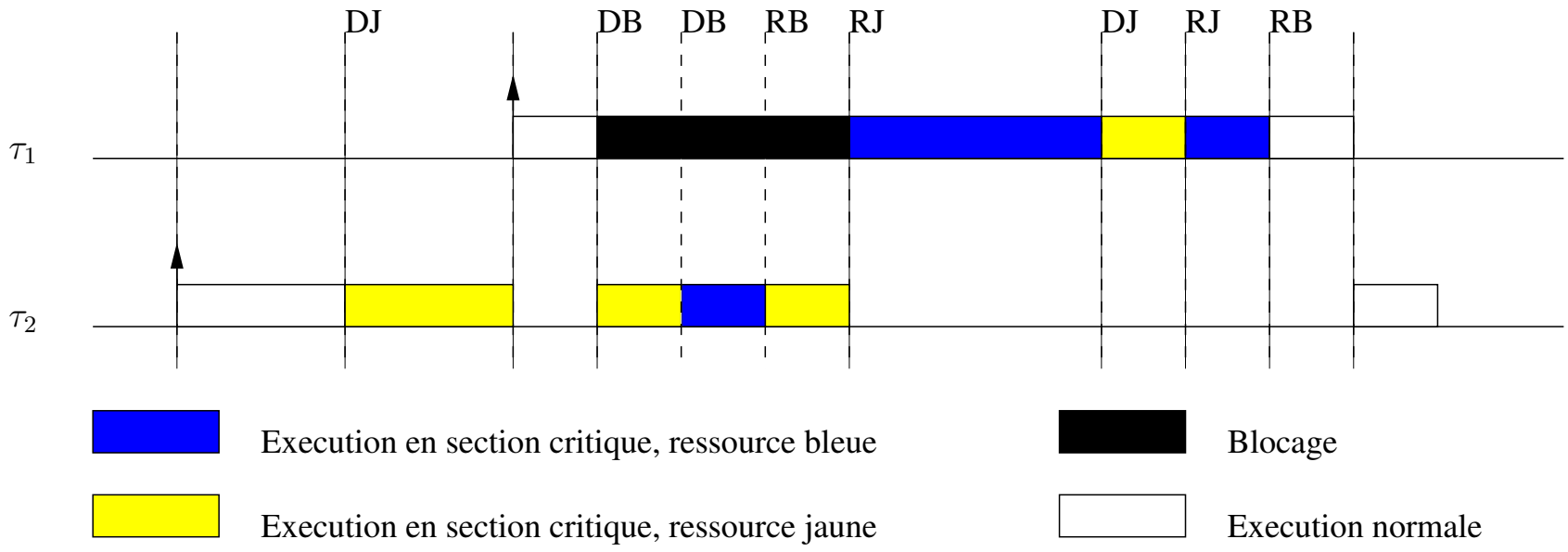
(Pas d')Interblocage

PCE



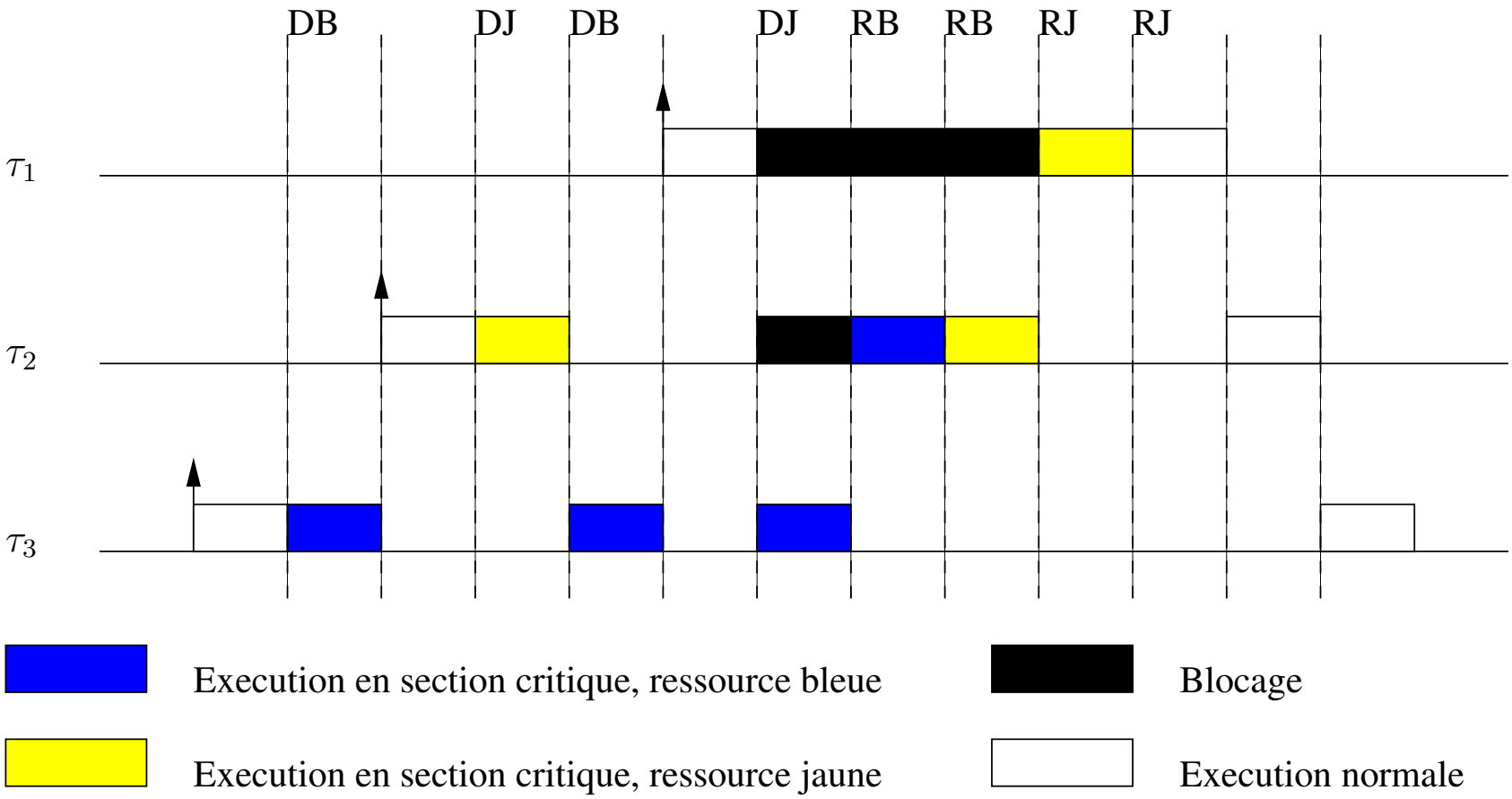
(Pas d')Interblocage

PCP



PIP & PCP : Héritage transitif

PIP



Modèle de trafic apériodique

Ordonnancement mixte périodique / apériodique

1. garantir le respect des échéances des tâches composant le trafic périodique
2. minimiser le temps de réponse des tâches composant le trafic apériodique

- impossible de garantir une contrainte temporelle pour une tâche apériodique :
  - elle peut arriver n'importe quand
  - il peut en arriver simultanément un nombre imprévisible

Traitement en tâche de fond (BS)

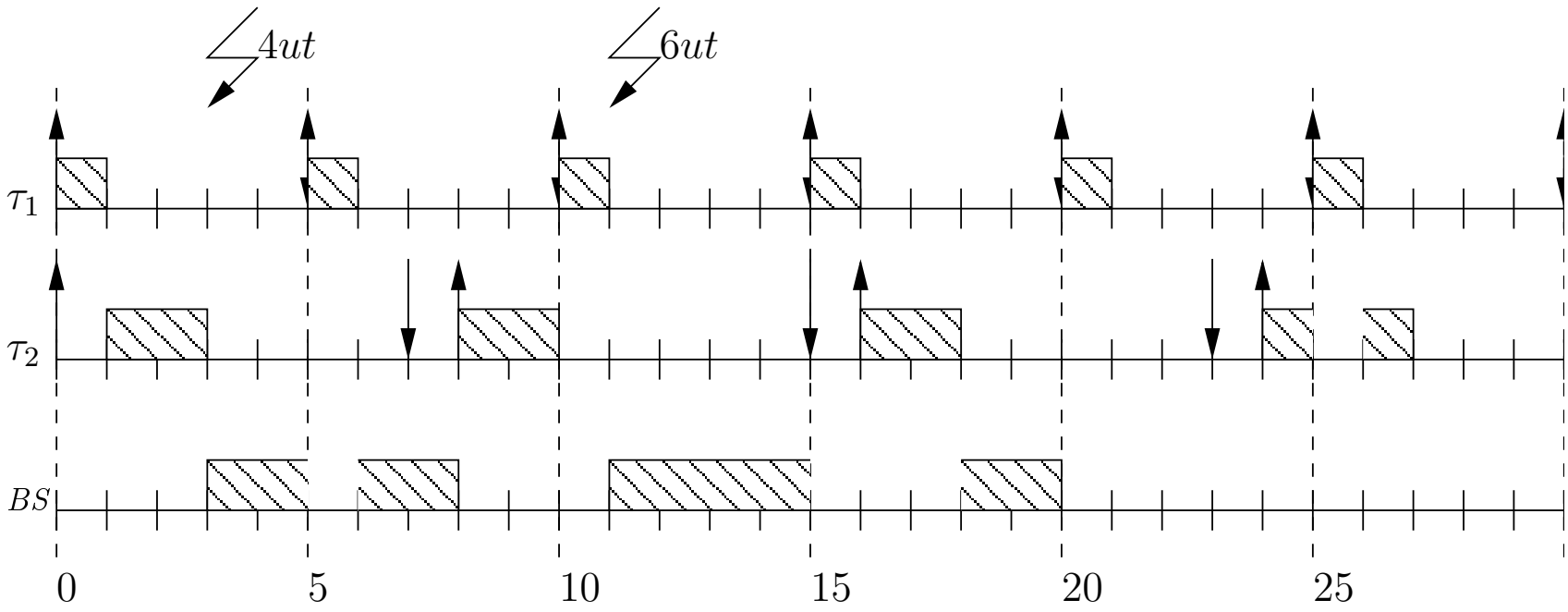
Background Scheduling

reserver une plage de priorité faible pour les apériodiques

- très simple à mettre en œuvre
- garantie qu'elles ne gênent jamais les périodiques

- ne répond qu'à 1) ! rien pour les temps de réponse des apériodiques

BS



Serveurs de tâches apériodiques

Serveurs de tâches

- ressources reservées
- influence maximale sur les autres tâches identifiée

Délégation du traitement des apériodiques à une tâche particulière qui dispose :

- d'un budget
- d'une politique de renouvellement de ce budget

De nombreux algorithmes possibles : PS, DS, SS, PE, EPE...

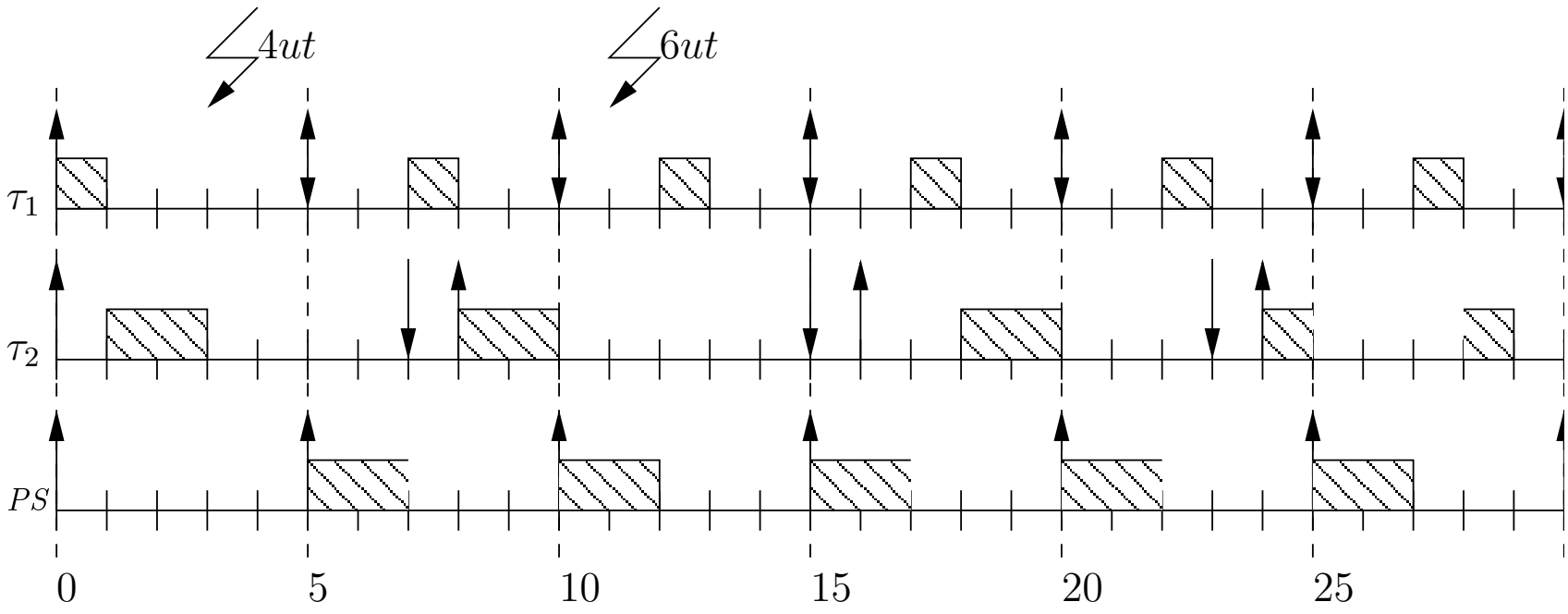
Polling Server (PS)

Ou *serveur à scrutation*

Il s'agit d'une tâche périodique et il s'analyse comme telle.

- $PS = \{(r_s, C_s, T_s)\}$
- éventuellement : $PS = \{(r_s, C_s, T_s, P_s)\}$
- les apériodiques sont ajoutées dans une file d'attente lors de leur activation,
- lorsque le serveur obtient le processeur, il exécute les tâches de la file dans la limite de sa capacité,
- la capacité revient au maximum périodiquement,
- si la file est vide alors que le serveur a la main, la capacité tombe à zéro.

Polling Server (PS)

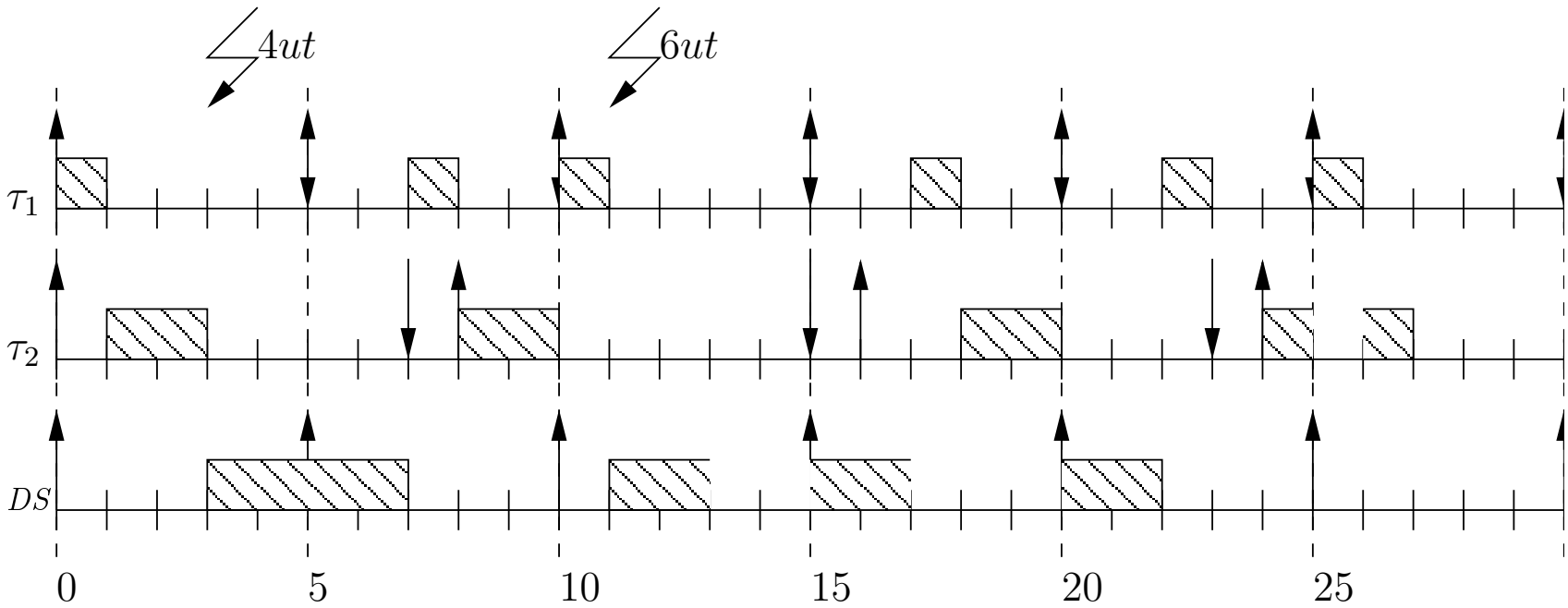


Deferrable Server (DS)

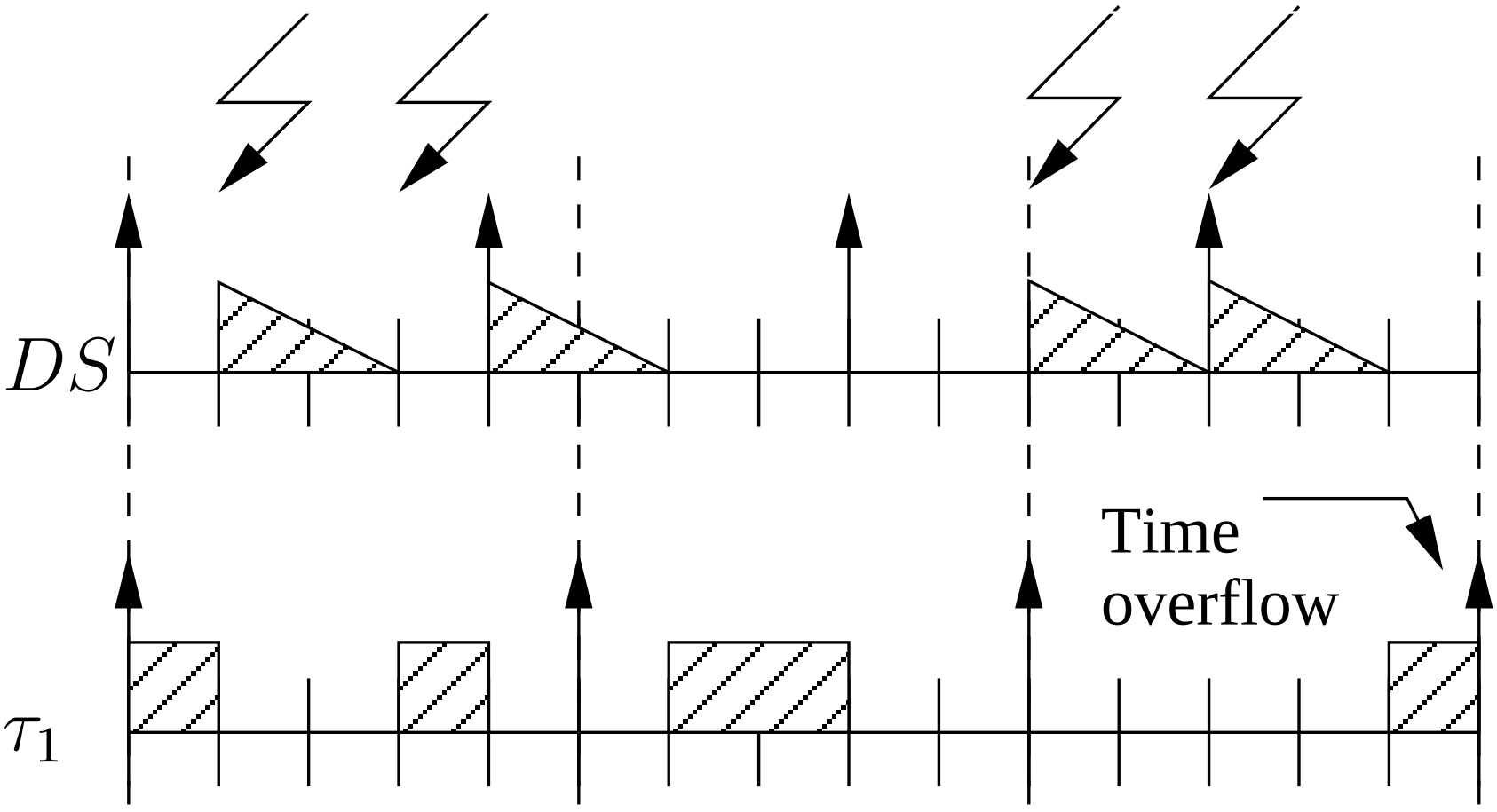
Ou *serveur ajournable*

- $DS = \{(r_s, C_s, T_s)\}$
- éventuellement : $DS = \{(r_s, C_s, T_s, P_s)\}$
- identique au PS, mais conserve sa capacité lorsque la file est vide,
- n'est plus une tâche périodique, et ne s'analyse plus comme telle.

Deferrable Server (DS)



Deferrable Server (DS)



DS : Analyse d'ordonnançabilité

- Condition suffisante sur la charge en Rate Monotonic :

$$U = U_s + \sum_{i=1}^{n} \frac{C_i}{T_i} \leq U_s + n \times \left(\left(\frac{U_s + 2}{2U_s + 1} \right)^{\frac{1}{n}} - 1 \right) \quad (3)$$

  avec $U_s = \frac{C_s}{T_s}$ et n n'incluant pas le serveur.
- en prenant la limite quand n tend vers l'infini, ca donne :

$$U \leq U_s + ln \left(\frac{U_s + 2}{2U_s + 1} \right) \quad (4)$$

- Pour l'analyse des temps de réponse on change le pire cas :
  - pour les tâches périodiques moins prioritaires que le serveur, activation synchrone à $t = r_s + T_s - C_s$,
  - pour le serveur, activation synchrone avec les tâches plus prioritaires,
  - équivalant à l'analyse d'ordonnançabilité de tâches périodiques avec gigue d'activation.

Vol de temps creux

Slack Stealing

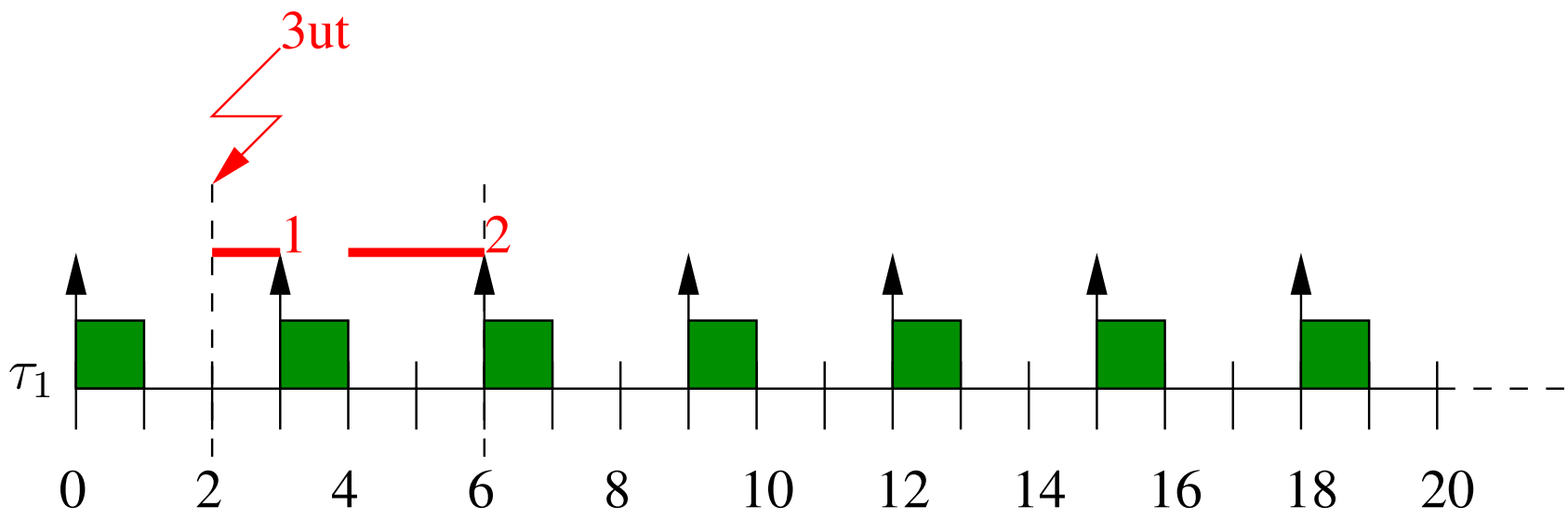
- pas de reservation
- calcul dynamique de la laxité du système (charge supplémentaire acceptable à un instant donné)

- meilleures performances que les serveurs
- mais plus de réservation

Laxité

Définitions

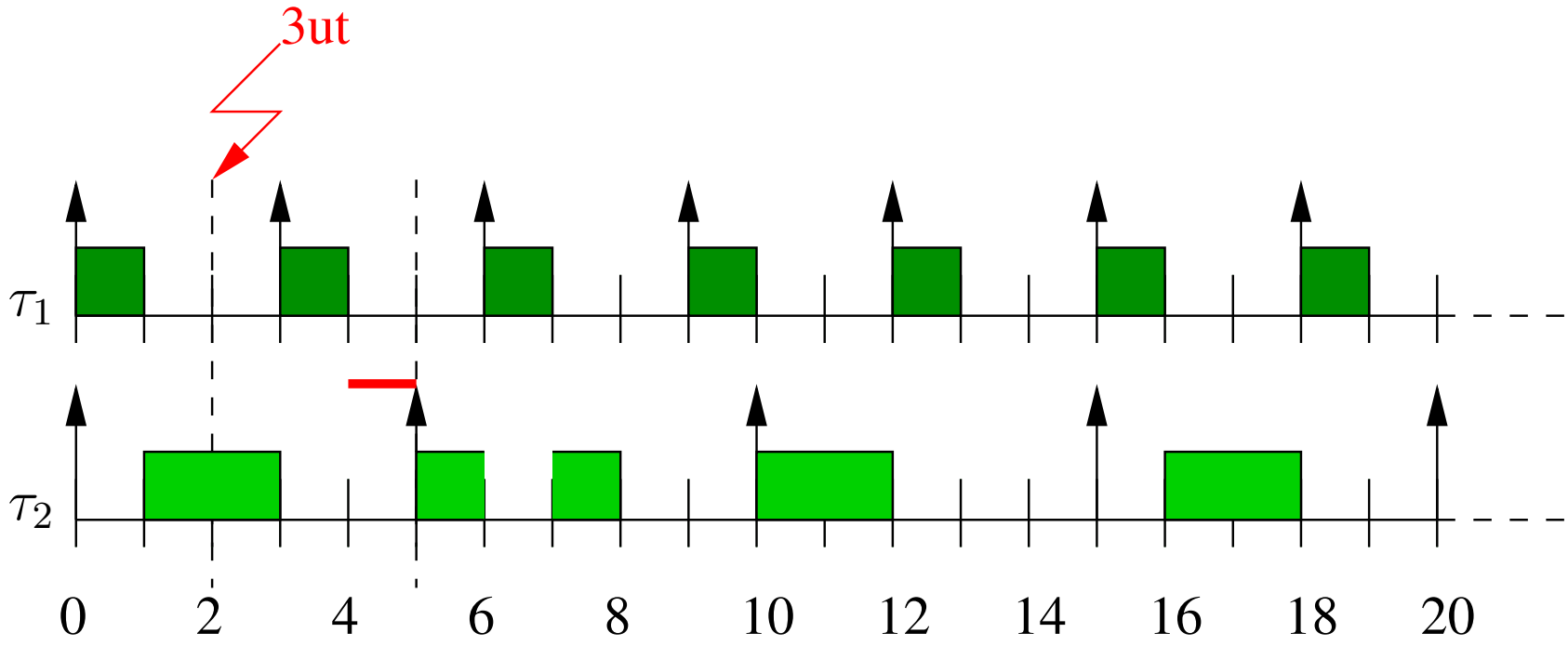
- $S_i(t)$: travail supplémentaire possible aux niveaux $\geq i$ jusqu'à la prochaine échéance de τ_i



Laxité

Définitions

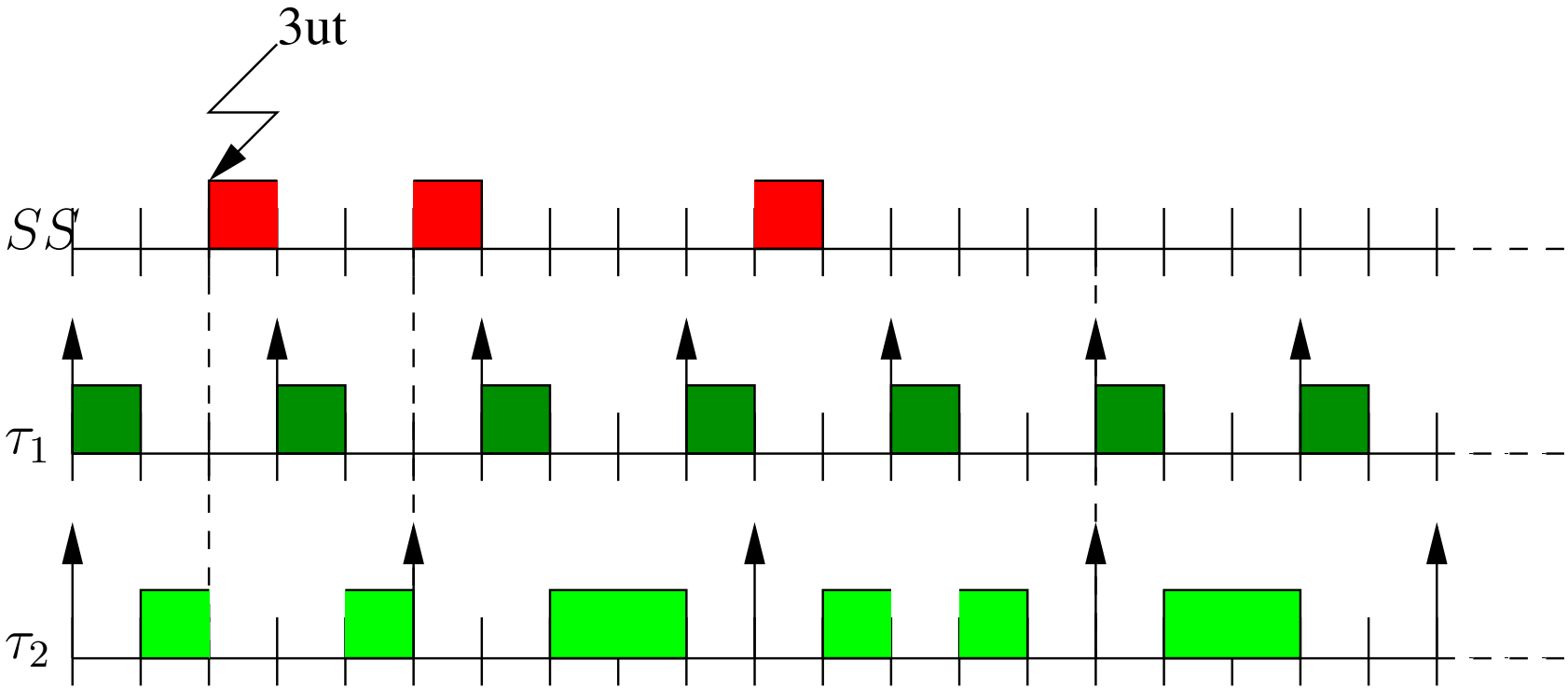
- $S_i(t)$: travail supplémentaire possible aux niveaux $\geq i$ jusqu'à la prochaine échéance de τ_i
- $S_t = \min S_i(t)$



Laxité

Définitions

- $S_i(t)$: travail supplémentaire possible aux niveaux $\geq i$ jusqu'à la prochaine échéance de τ_i
- $S_t = \min S_i(t)$



Calcul Dynamique

Calcul des $S_i(t)$

- complexité en temps polynomiale
- principe : calculer la durée de la prochaine période d'inactivité puis celle de la prochaine période d'activité etc. jusqu'à la prochaine échéance

- non implantable
- solution : utiliser une approximation pessimiste

Principe

- $S_i(t) \geq \left(d_i(t) - t - \sum_{j \leq i} I_j^i(t, d_i(t)) \right)_0$
- $\mathcal{O}(n)$

- $S_i(t)$ n'augmente que si τ_i termine une instance
- entre deux instants, S_i diminue du temps passé à exécuter des tâches moins prioritaires

- il faut donc réévaluer S_i à la fin de chaque tâche périodique
- et savoir à tout instant combien de temps chaque tâche a consommé

Pour aller plus loin

- Ordonnancement temps réel multiprocesseur,
- Modèle d'arrivée des tâche plus complexes : gigues d'activation...
- Modèle de contraintes plus complexes : modèle (m,k)firm ...
- Tolérance aux fautes temporelles, détection de surcharges, ...