

Ordonnancement temps réel

B. Miramond

Polytech Nice Sophia

Plan du cours

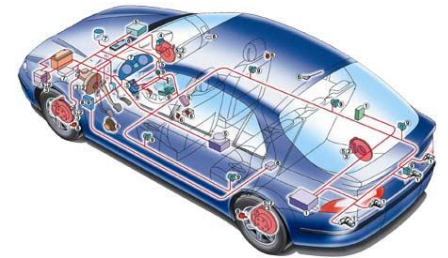
- Modèles de tâches : description non fonctionnelle de l'application
- Politiques à priorités fixes
 - RMS
- Politiques à priorités dynamiques
 - EDF
 - LLF
- Conditions d'ordonnancabilité
- Ressources critiques et inversion de priorités
 - deadlock
 - Héritage de priorités
 - Plafond de priorités
- Hyper-période
- Estimation de WCET

Séance 1

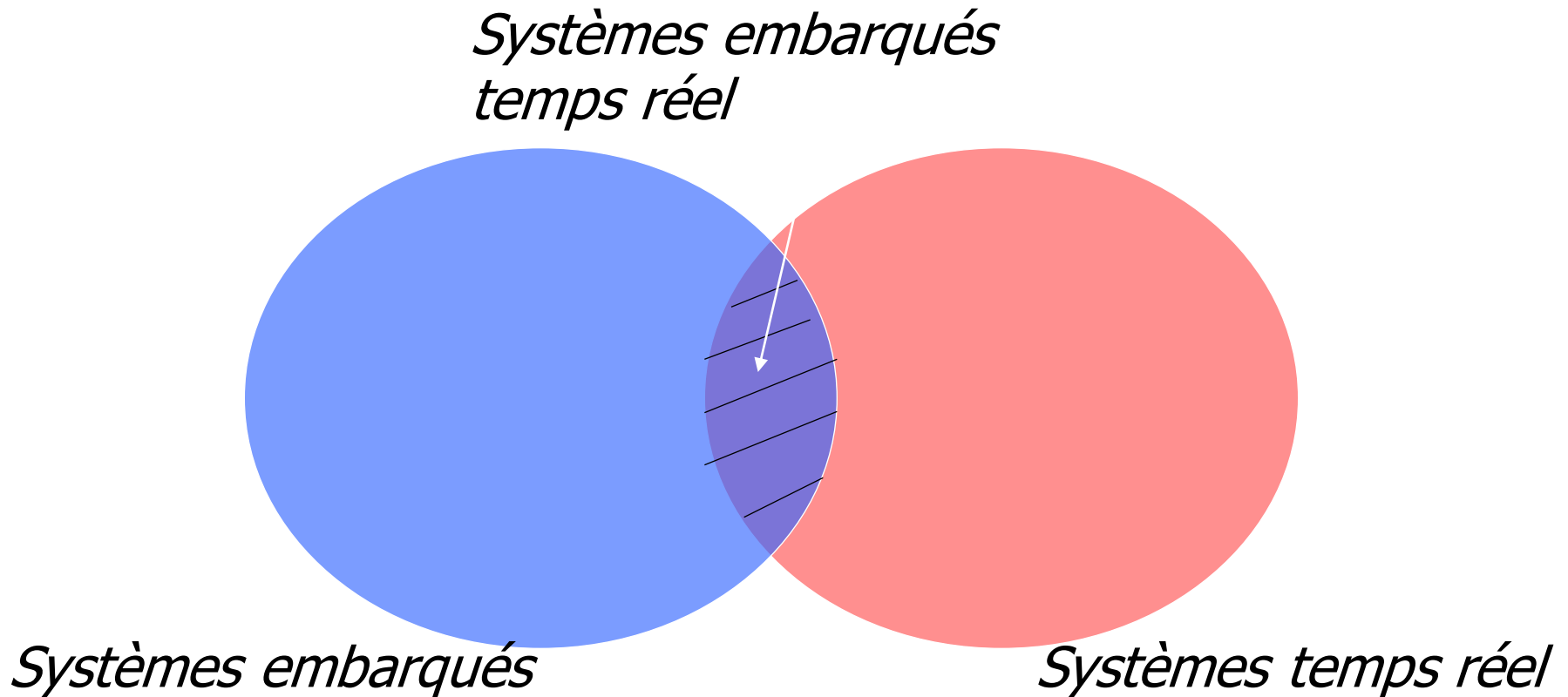
ALGORITHMES PRINCIPAUX D'ORDONNANCEMENT

Secteurs d'application du temps réel

- Automobile (Contrôle moteur, ABS, Airbag, etc.),
- Aéronautique et aérospatial,
- Systèmes militaires,
- Systèmes médicaux,
- Processus industriels (centrales nucléaires, robotique, production chimique, etc.),
- Systèmes de surveillance et d'alarme,
- Systèmes de télécommunication.



Systèmes embarqués temps réel



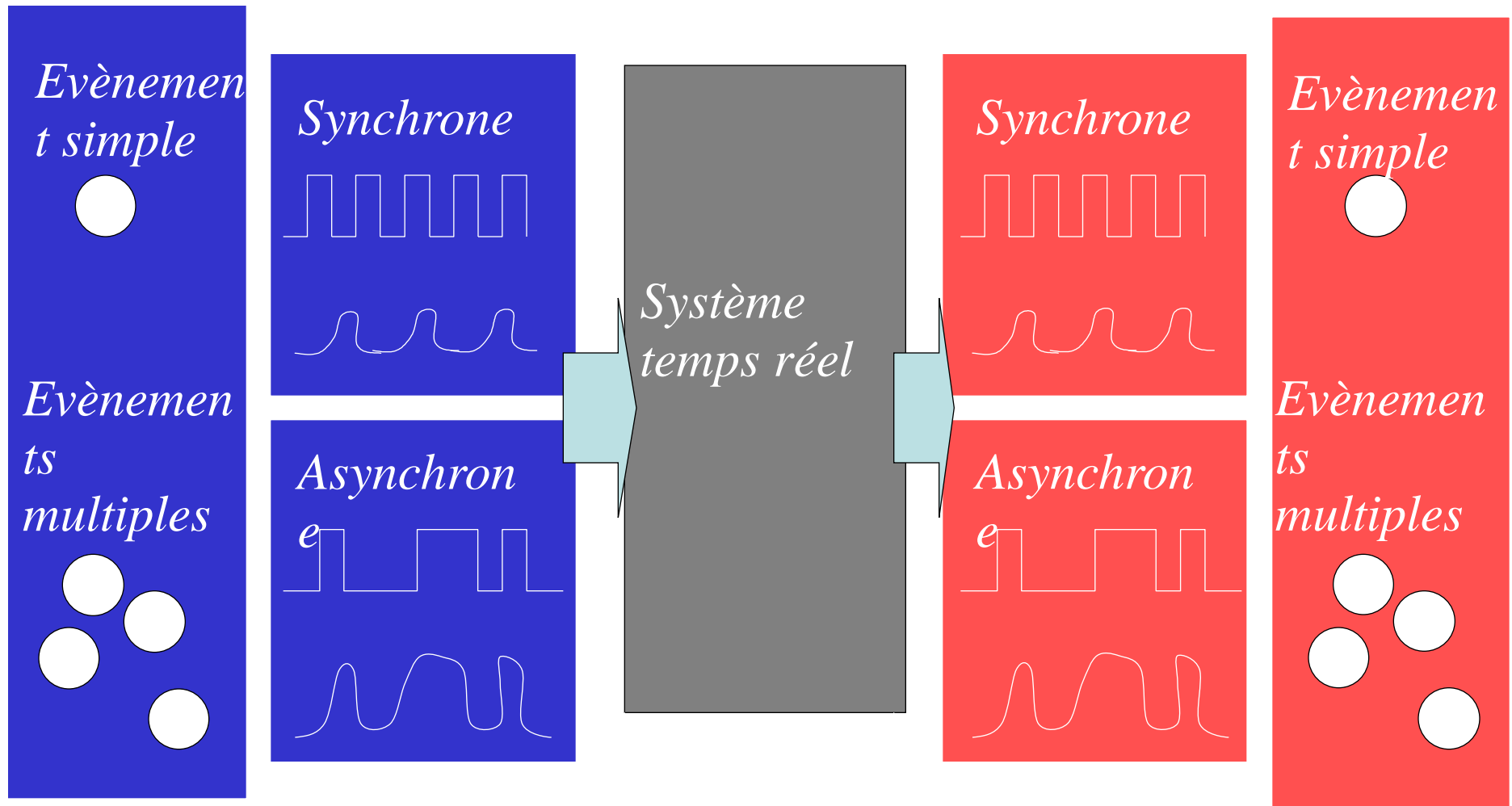
Ce sont des systèmes liés au contrôle de procédés du monde (temps) **réel**.

L'exécution de traitements dans ces systèmes doit terminer avant une date butoir appelée **échéance** définie par la vitesse d'évolution de l'environnement **réel** au-delà de laquelle les résultats ne sont plus valides.

Systèmes temps réel

- Il ne s'agit pas de rendre le résultat le plus rapidement possible, mais simplement à temps.
- L'échelle de temps de l'échéance peut varier d'une application à l'autre
 - microseconde en contrôle radar
 - milliseconde pour la synchronisation image/son (mpeg)
 - minute pour les distributeurs automatiques
- Tous peuvent être temps réel souples ou durs

La notion/modélisation du temps est fondamentale



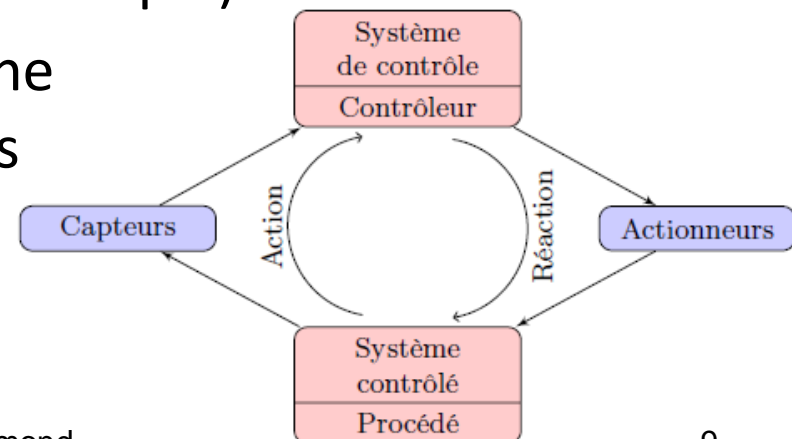
Description des applications temps réel

- On ne s'intéresse qu'aux aspects temporels de l'application,
- On s'abstrait des aspects fonctionnels,
- On décrit
 - la décomposition fonctionnelle en tâches
 - Les temps d'exécution de ces fonctions
 - Les dépendances de données entre ces fonctions
 - Les périodes de réactivation de ces fonctions

Systèmes multi-périodes

La plupart des systèmes embarqués sont dit *mutlirate* ou multi-période

- Les données sont capturées à un certain rythme (du monde réel) : tour de roue de l'automobile, fps d'une caméra, ...
- Les traitements sur ces données ne sont pas forcément de même granularité (1 pour 64)
- Différents traitements peuvent intervenir de manière indépendante (périodique et apériodique)
- Les actionneurs fonctionnent à une fréquence différente des capteurs



Modèle formel de tâches

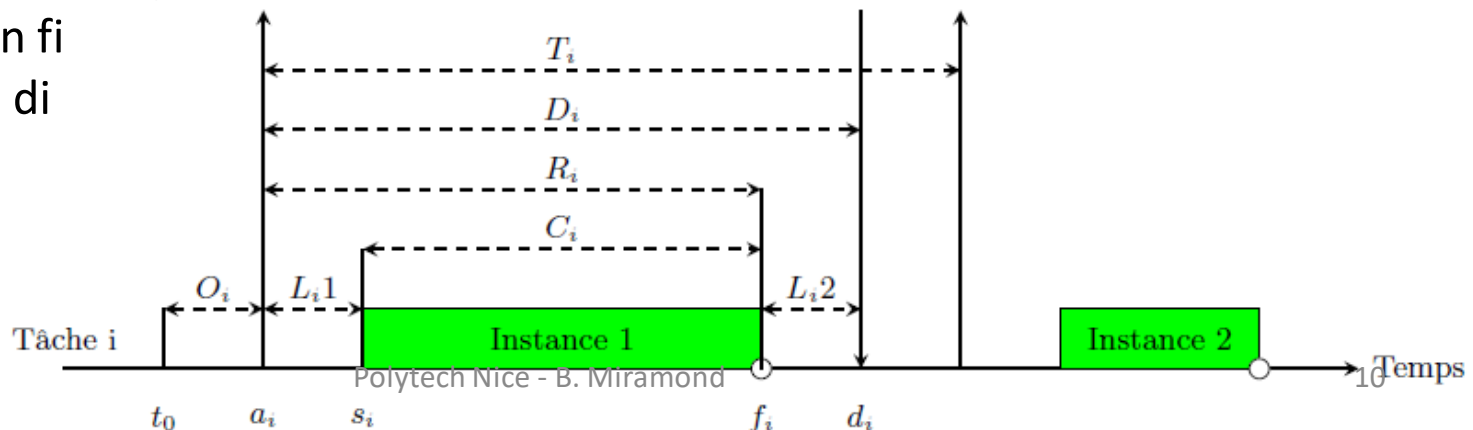
(sans dépendances de données)

Une tâche T_i est modélisée par :

- Un temps d'exécution C_i estimé à partir du WCET
- Un temps de réponse R_i , date séparant le réveil de la fin d'exécution ($f_i - a_i$)
- Une échéance d_i , date avant laquelle la tâche doit s'achever
- Une période de réactivation T_i d'une nouvelle instance de la tâche
- Un Offset $O_i = a_i - t_0$. Si l'offset est connu a priori la tâche est dite concrète, sinon elle est dite non-concrète.
- Une laxité L_i , marge pendant laquelle la tâche peut être retardée sans dépasser l'échéance : $L_i = D_i - C_i$

Son ordonnancement est représenté par :

- Une date de réveil a_i
- Une date de démarrage s_i
- Une date de fin f_i
- Une échéance d_i



Modèle formel de tâches

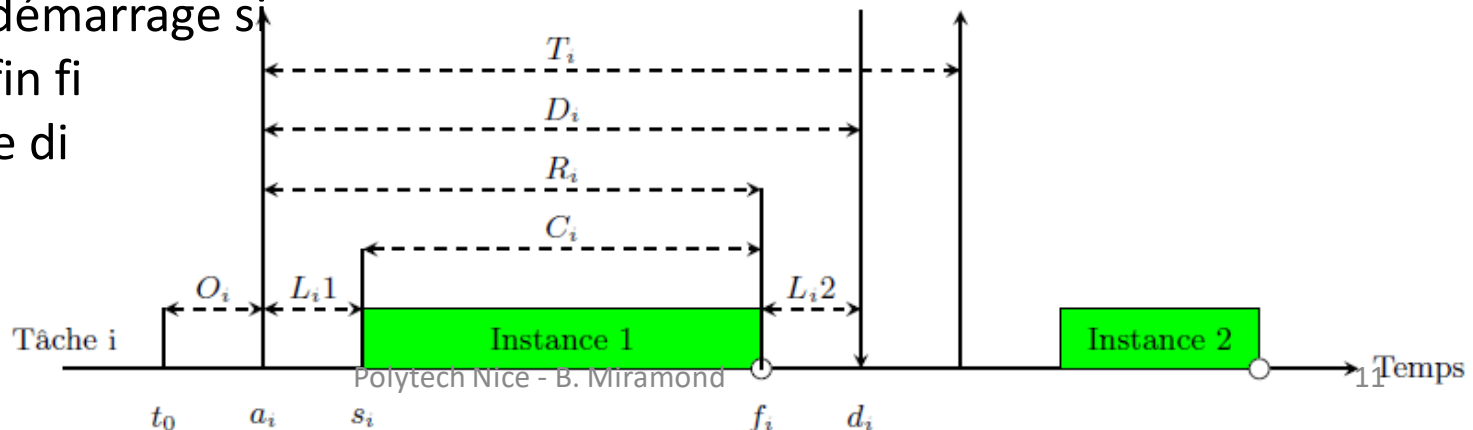
(sans dépendances de données)

Dans ce cours, une tâche T_i sera modélisée par :

- Un temps d'exécution C_i estimé à partir du WCET
- Un temps de réponse R_i , date séparant le réveil de la fin d'exécution ($f_i - a_i$)
- Une échéance d_i , date avant laquelle la tâche doit s'achever
- Une période de réactivation T_i d'une nouvelle instance de la tâche
- Un Offset O_i connu et nul
- Une laxité L_i , marge pendant laquelle la tâche peut être retardée sans dépasser l'échéance : $L_i = D_i - C_i$

Son ordonnancement est représenté par :

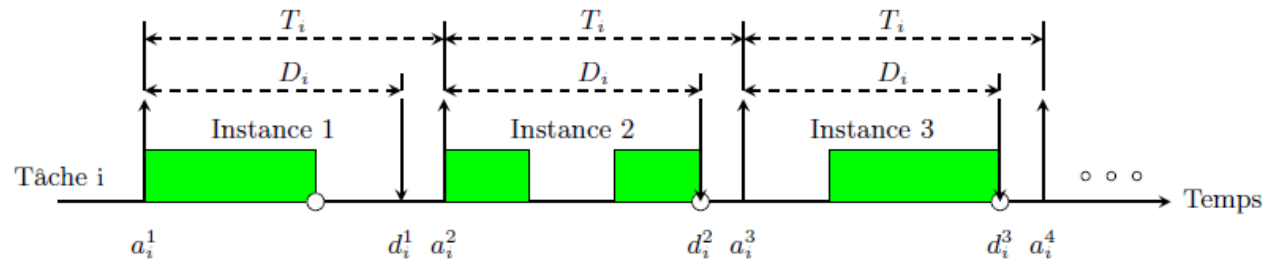
- Une date de réveil a_i
- Une date de démarrage s_i
- Une date de fin f_i
- Une échéance d_i



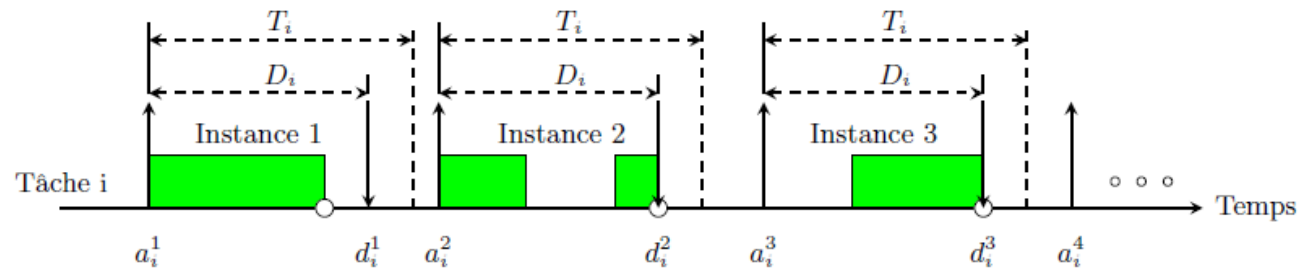
Types de tâches

- Tâches périodiques

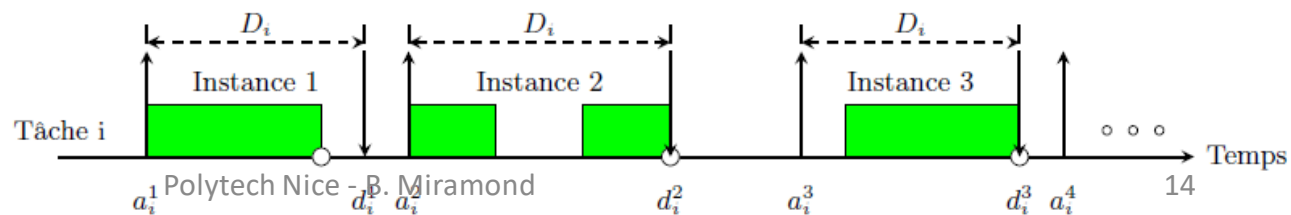
$$a_i^m = a_i^1 + (m-1)T_i \quad ; \quad m \geq 1$$



- Tâches sporadiques (cycliques mais réveil irrégulier)

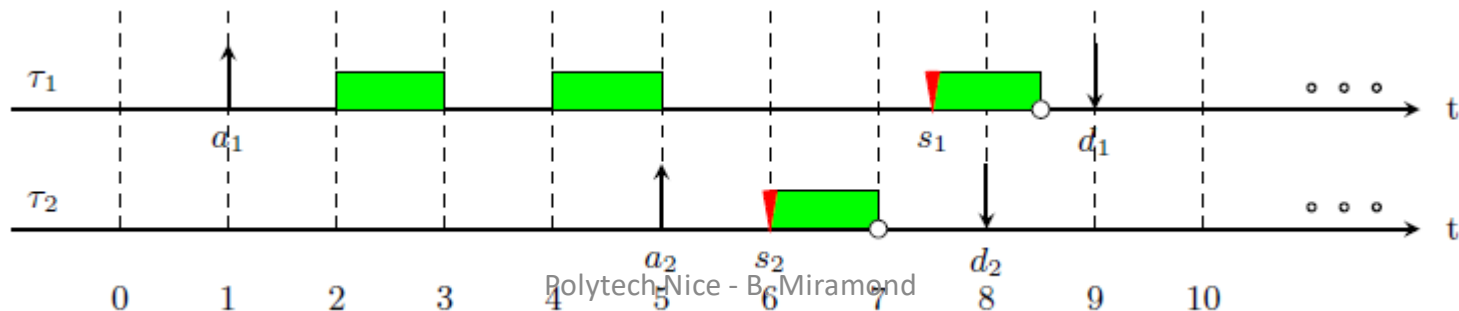


- Tâches apériodiques (non cycliques, réveil imprévisible)



Ordonnancement

- L'ordonnancement représente la planification dans le temps de l'exécution des tâches d'une application
- Le résultat de l'ordonnancement est une séquence d'exécutions des tâches généralement représenté par un chronogramme ou diagramme de Gantt



Classification des politiques d'ordonnancement

- Ordonnancement hors-ligne / en-ligne
 - Déterminé avant l'exécution dans une table, ordonnancement statique
 - Déterminé pendant l'exécution, plus flexible mais plus coûteux, on parle aussi d'ordonnancement dynamique
- Ordonnancement préemptif / non-préemptif
 - Préemptif lorsque l'exécution d'une tâche peut être interrompue par d'autres : surtout d'exécution
 - Non préemptif : plus simple
- Ordonnancement à priorités statiques / dynamiques
 - L'ordonnancement se base sur la notion de priorité. A chaque instant la tâche de plus haute priorité (HPT – Highest Priority Task) qui est prête (état Ready) est élue par l'ordonnanceur pour l'exécution.
 - Si ces priorités peuvent changer au cours du temps, on parle de priorités dynamiques
- Ordonnancement monoprocesseur / multiprocesseur

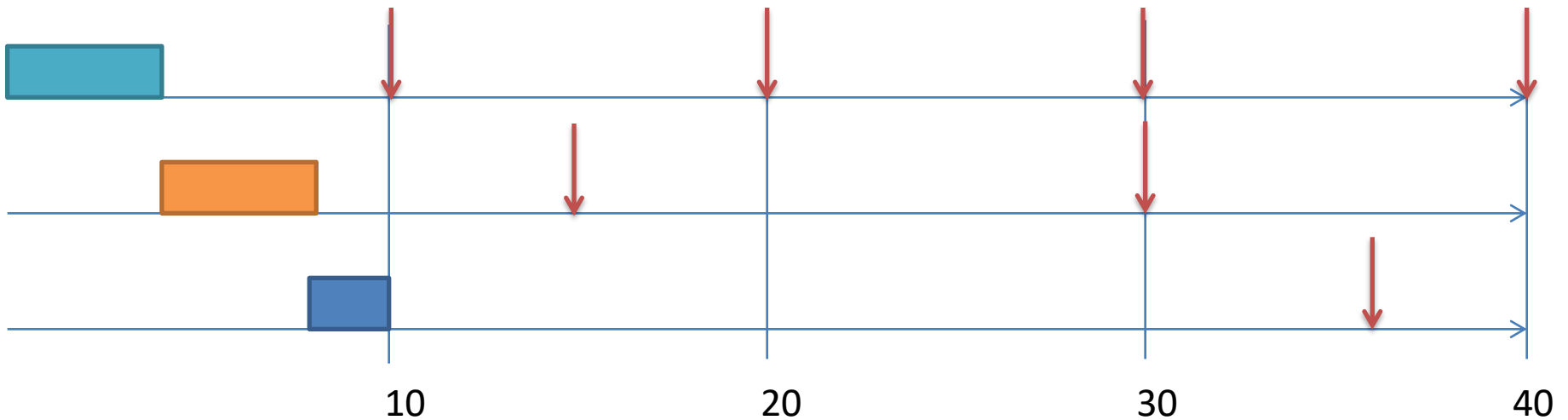
Politiques d'ordonnancement de tâches périodiques

- EDF (Earliest Deadline First)
 - Algorithme à priorités dynamiques : la priorité est d'autant plus forte que la date d'échéance est proche
 - EDF est dit optimal au sens où si un jeu de tâches ne peut être ordonnancé par EDF, alors il ne pourra l'être par aucun autre algorithme
 - Un jeu de tâches périodiques est ordonnancable par EDF si son facteur de charge U est inférieur ou égal à 1

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

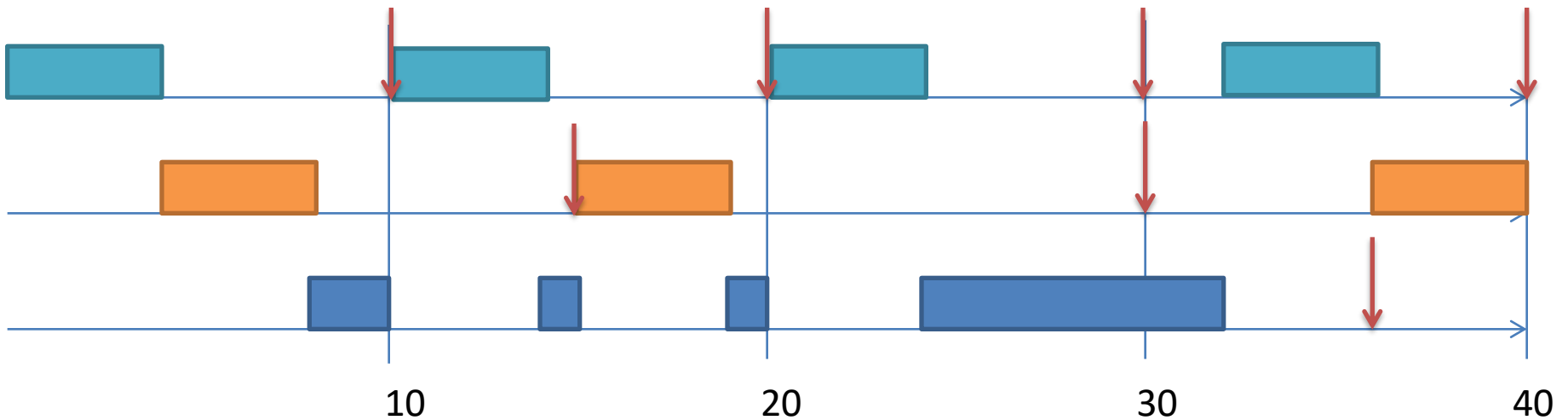
Exemple d'ordonnancement EDF

| | période | calcul | utilisation |
|----------|---------|--------|-------------|
| τ_1 | 10 | 4 | 0.400 |
| τ_2 | 15 | 4 | 0.267 |
| τ_3 | 36 | 12 | 0.333 |



Exemple d'ordonnancement EDF

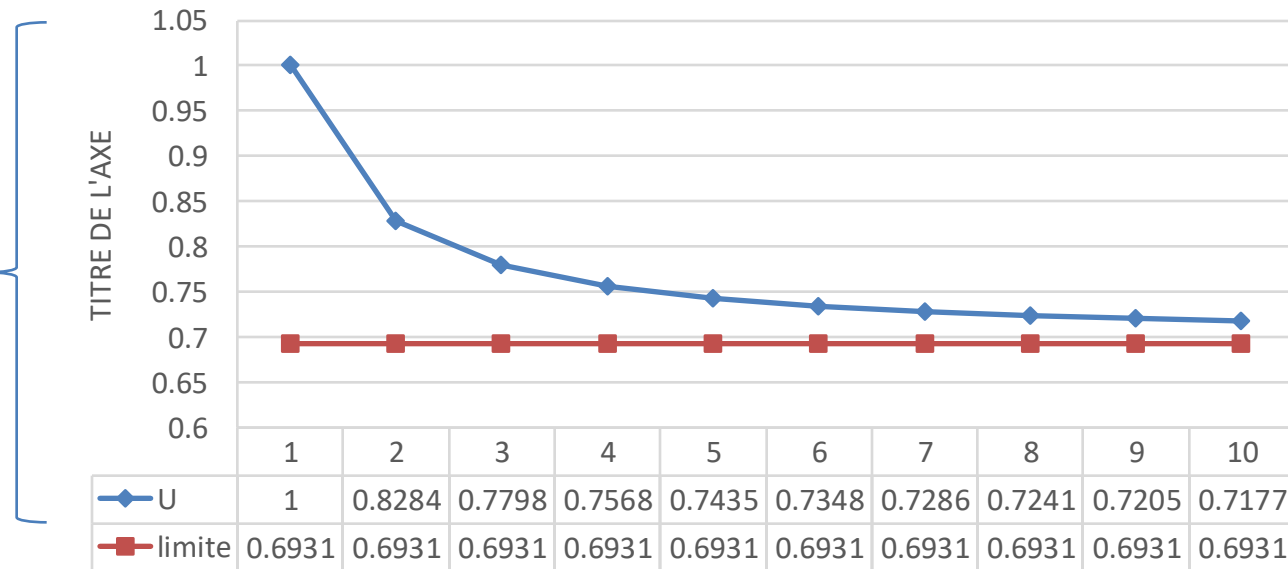
| | période | calcul | utilisation |
|----------|---------|--------|-------------|
| τ_1 | 10 | 4 | 0.400 |
| τ_2 | 15 | 4 | 0.267 |
| τ_3 | 36 | 12 | 0.333 |



Politiques d'ordonnancement de tâches périodiques

- Rate Monotonic Analysis / Scheduling (RMA/RMS)
 - RMS est un algorithme à priorités fixes, une tâche est d'autant prioritaire que sa période est petite
 - La condition d'ordonnancabilité de RMS, pour n tâches, est

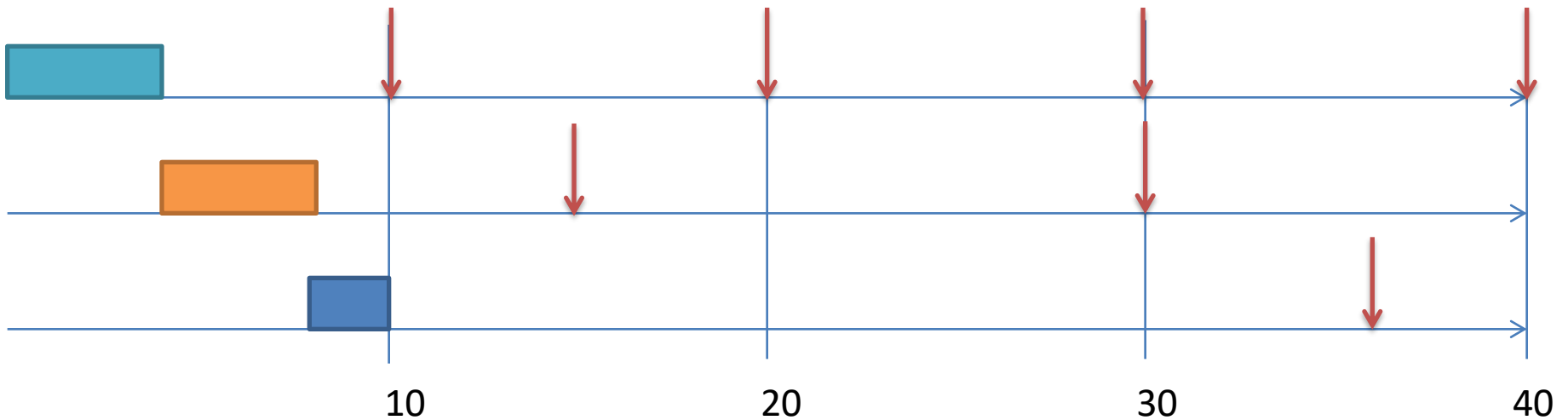
$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$



[RMS] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM (JACM), 20(1):46–61, 1973.

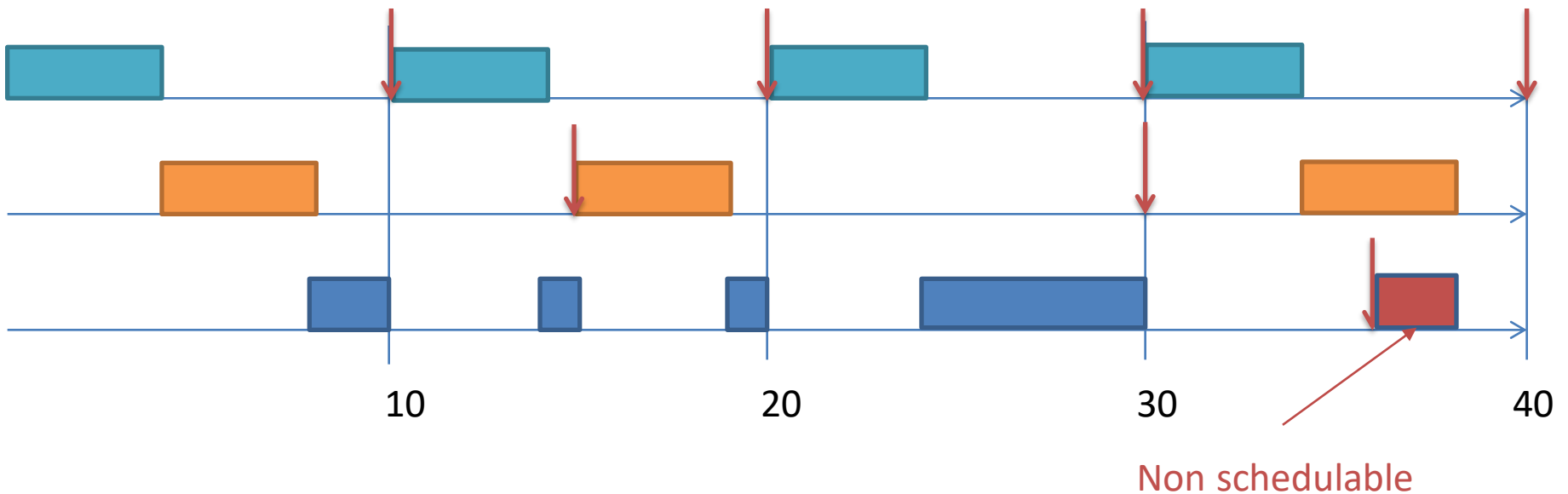
Exemple d'ordonnancement RMS

| | période | calcul | utilisation |
|----------|---------|--------|-------------|
| τ_1 | 10 | 4 | 0.400 |
| τ_2 | 15 | 4 | 0.267 |
| τ_3 | 36 | 12 | 0.333 |



Exemple d'ordonnancement RMS

| | période | calcul | utilisation |
|----------|---------|--------|-------------|
| τ_1 | 10 | 4 | 0.400 |
| τ_2 | 15 | 4 | 0.267 |
| τ_3 | 36 | 12 | 0.333 |

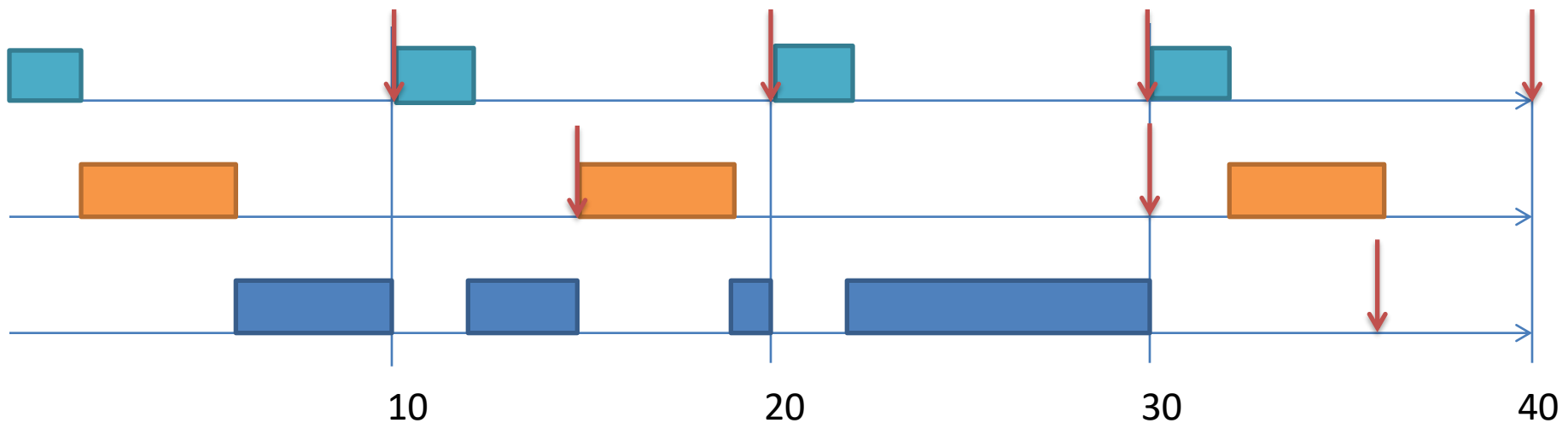


Exemple d'ordonnancement RMS

$U_{\max}(3) = 0,77976315,$

$U = 0,8$

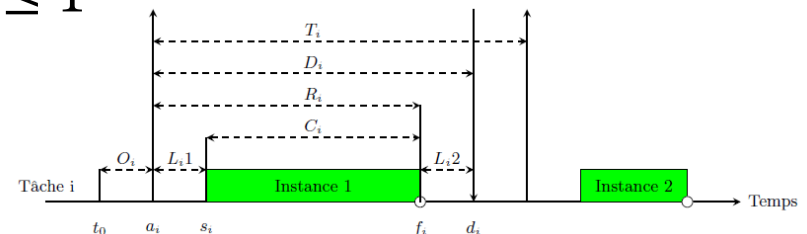
| $3 \times (2^{1/3} - 1) \approx 0.78$ | période | calcul | utilisation |
|---------------------------------------|---------|--------|-------------|
| τ_1 | 10 | 2 | 0.200 |
| τ_2 | 15 | 4 | 0.267 |
| τ_3 | 36 | 12 | 0.333 |



Politiques d'ordonnancement de tâches périodiques

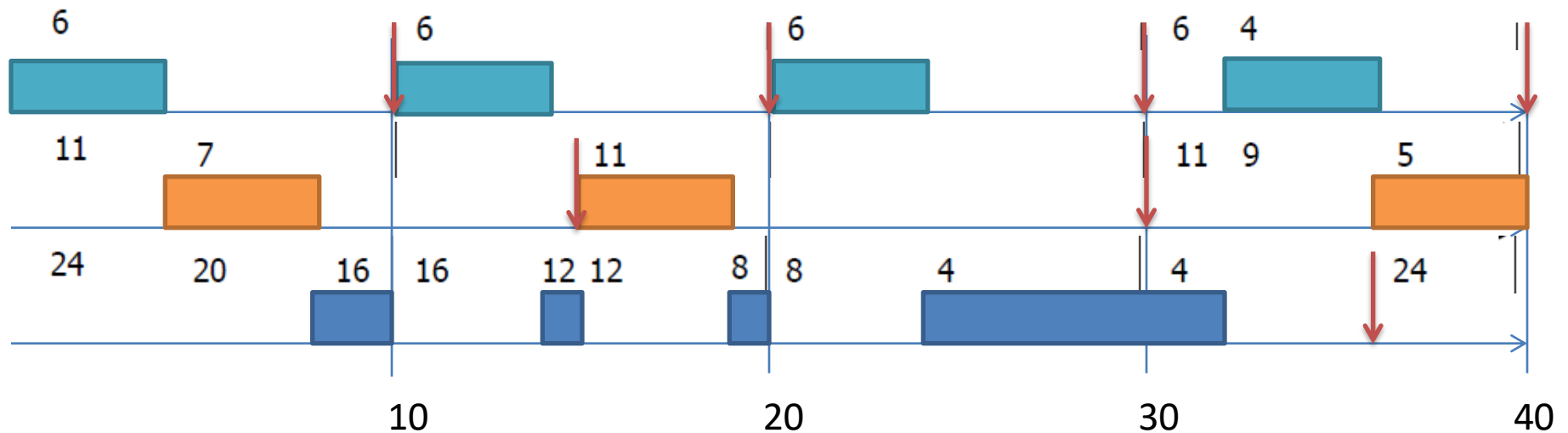
- Least Laxity First (LLF)
 - Algorithme à priorité dynamique, la priorité est d'autant plus forte que la laxité est faible
 - La laxité à l'instant présent est calculée comme le temps avant la prochaine échéance diminuée du temps d'exécution restant de la tâche
 - Inconvénient : le nombre prohibitif de préemptions et donc de changements de contextes, surtout élevé
 - Même condition que EDF

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$



Exemple d'ordonnancement LLF

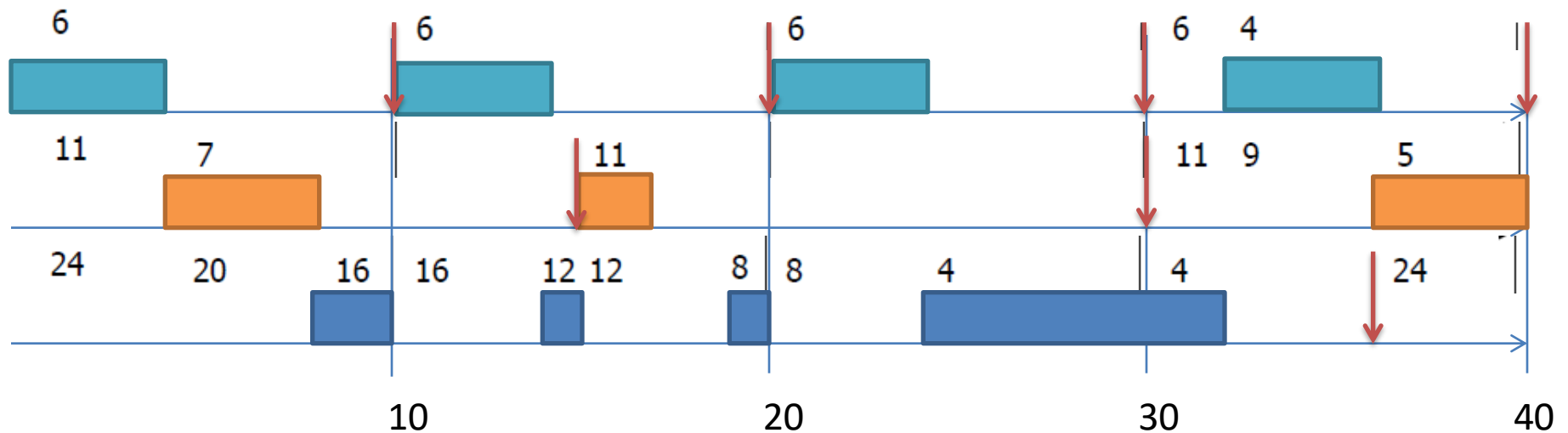
| | période | calcul | utilisation |
|----------|---------|--------|-------------|
| τ_1 | 10 | 4 | 0.400 |
| τ_2 | 15 | 4 | 0.267 |
| τ_3 | 36 | 12 | 0.333 |



Exemple d'ordonnancement LLF

Illustration de la laxité à partir de la date 15

| date | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 |
|------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| T1 | - | - | - | - | - | 6 | 6 | 6 | 6 | - | - | - | - | - | - | 6 | 5 | 4 | 4 | 4 | 4 | - |
| T2 | 11 | 11 | 11 | 11 | - | - | - | - | - | - | - | - | - | - | - | 11 | 10 | 9 | 8 | 7 | 6 | 5 |
| T3 | 12 | 11 | 10 | 9 | 8 | 8 | 7 | 6 | 5 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | - | - | - | - | 24 |



Politiques d'ordonnancement de tâches périodiques

- Deadline monotonic scheduling (DMS)
 - Algorithme à priorité statique, priorité haute si le délai critique D_i de la tâche est petit
 - DMS = RMS lorsque $D_i = T_i$
 - Algorithme le plus utilisé en pratique

[DMS] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. Performance Evaluation, 2(4) :237–250, 1982

Critères d'ordonnancement

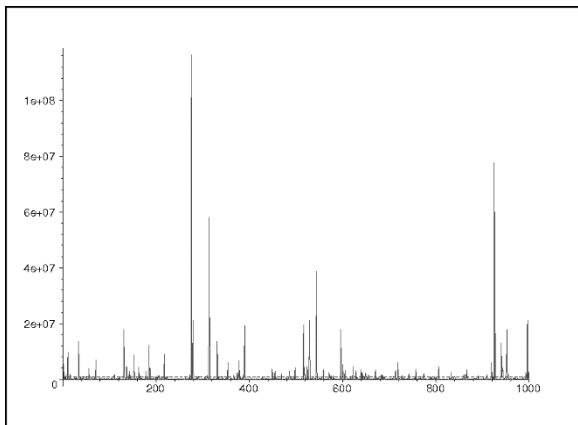
| Critère d'ordonnancement | Priorités fixes | Priorités dynamiques |
|--------------------------|-----------------|----------------------|
| Temps d'exécution | | |
| Période | | |
| Échéance | | |
| Utilisation du CPU | | |
| Consommation d'énergie | | |
| Laxité | | |

Critères d'ordonnancement

| Critère d'ordonnancement | Priorités fixes | Priorités dynamiques |
|--------------------------|-----------------|----------------------|
| Temps d'exécution | X | |
| Période (RMS) | X | |
| Échéance | X (DMS) | X (EDF) |
| Utilisation du CPU | | X |
| Consommation d'énergie | | X |
| Laxité (LLF) | | X |

Périodicité de l'ordonnancement

- La séquence produite par tout algorithme d'ordonnancement préemptif sur un jeu de tâches périodiques est elle-même périodique de période égale au Plus Petit Commun Multiple (PPCM) des périodes des tâches de la configuration, noté P , Hyper-période
- L'ordonnancement sera donc dans le même état à la date $t + kP$, $k \in \mathbb{N}$
- Ainsi, si une condition d'ordonnancement n'est pas valide, il faut simuler l'ordonnancement sur P pour vérifier si ce jeu de tâches est faisable
- Mais cette valeur P croît exponentiellement avec le nombre de tâches et la valeur de la plus grande période

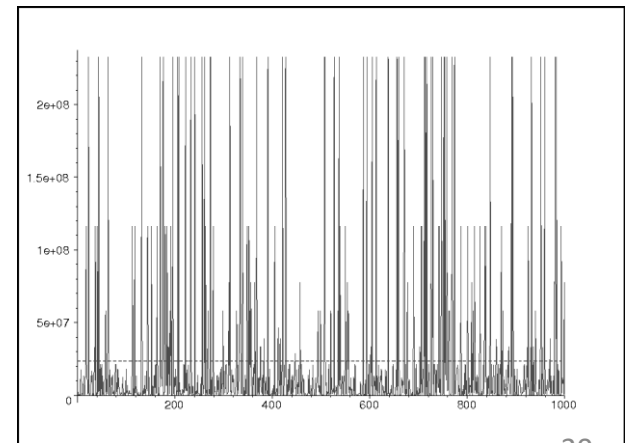


Jeu de tâches dont les périodes sont tirées aléatoirement dans $[1,10]$



10 tâches

20 tâches



Théorème de la zone critique

- Si toutes les tâches arrivent initialement dans le système simultanément et si elles respectent leur première échéance, alors toutes les échéances seront respectées par la suite, quel que soit l'instant d'arrivée des tâches.
- Pour cela, il faut trouver le temps de terminaison t , avant la première échéance de la tâche de plus grande période :

$$\forall i, 1 \leq i \leq n \quad \min_{0 < t < D_i} \sum_{j=1}^i C_j / t * \lceil t / T_j \rceil \leq 1$$

- Les tâches sont indicées de la plus petite période à la plus grande
- Pour résoudre le problème, on applique une méthode itérative

Méthode itérative pour le théorème de la zone critique

On recherche le temps t tel que :

$$\forall i, 1 \leq i \leq n, \exists t \leq D_i, \quad t = \sum_{j=1}^i C_j * \lceil t / T_j \rceil$$

$$\text{Soit } W_i(t) = \sum_{j=1}^i C_j * \lceil t / T_j \rceil$$

$W_i(t)$ représente la demande cumulée de temps processeur de toutes les tâches jusqu'à la tâche i dans l'intervalle $[0, t]$.

On recherche $W_i(t) / W_i(t)=t$ par itérations successives et ce pour toutes les tâches i

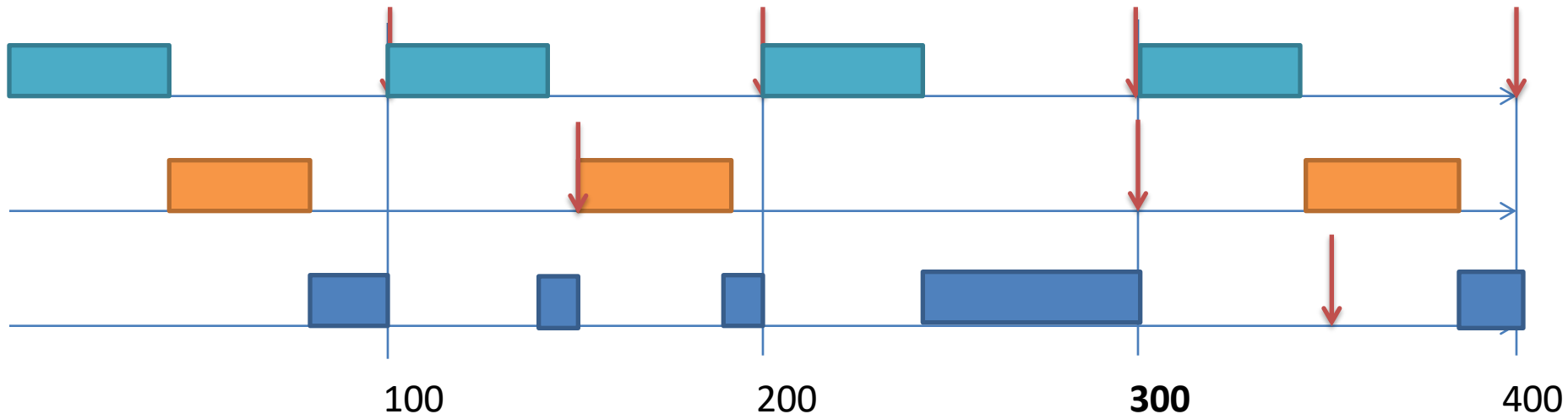
On part de $t_0 = \sum_{j=1}^i C_j$, si l'instant t ne respecte pas la condition, on itère en prenant comme nouveau temps $t = W_i(t)$

Exemple

| Pi | T | C |
|----|-----|-----|
| P1 | 100 | 40 |
| P2 | 150 | 40 |
| P3 | 350 | 100 |

$$\sum U = 0.779,$$

Mais on va montrer que l'exemple
répond au théorème de la zone critique.



Exemple

| Pi | T | C |
|----|-----|-----|
| P1 | 100 | 40 |
| P2 | 150 | 40 |
| P3 | 350 | 100 |

$$\sum_i U = 0.779,$$

Mais on va montrer que l'exemple répond au théorème de la zone critique.

Pour i de 1 à 3 :

Pour i=1 $t_0 = C_1 = 40$, $W_1(0) = 40 * 1 = 40$, $W_1(40) = 40$, i++

Pour i=2 $t_0 = C_1 + C_2 = 80 \Rightarrow W_2(0) = 80$, $W_2(80) = 80$, i++

Pour i=3 $t_0 = C_1 + C_2 + C_3 = 180$ (>100 et 150)

$$W_3(180) = 2C_1 + 2C_2 + C_3 = 260 > 180$$

$$W_3(260) = 3C_1 + 2C_2 + C_3 = 300 > 260$$

$$W_3(300) = 3C_1 + 2C_2 + C_3 = 300$$

et < **350 (T3)**, c'est la condition recherchée

$$t = \sum_{j=1}^i C_j * \lceil t / T_j \rceil$$

$$\min_{0 < t < D_i} \sum_{j=1}^i C_j / t * \lceil t / T_j \rceil \leq 1$$

$$3 * 40 / 300 + 2 * 40 / 300 + 1 * 100 / 300 = 300 / 300 = 1$$

Les trois tâches sont donc ordonnançables selon le théorème de la **zone critique**. La troisième tâche terminera l'exécution de sa première instance au temps **300**.

Séance 2

RESSOURCES PARTAGÉES

Ressources partagées

Problème d'accès à une ressource partagée

- Ressource protégée par un mécanisme d'accès à exclusion mutuelle (Mutex ou Sémaphore)
- Un ordonnancement préemptif peut conduire à une situation **d'inversion de priorité** où une tâche de faible priorité bloque une tâche de forte priorité pendant un temps supérieur à celui de l'exclusion mutuelle
- On ne peut évaluer la borne supérieure de ce temps

Solution du problème : **Protocole d'héritage de priorité**

- L'héritage de priorité **change la priorité** de la tâche bloquante au niveau de celle bloquée
- Une fois le sémaphore libéré, la tâche bloquée retrouve sa priorité initiale

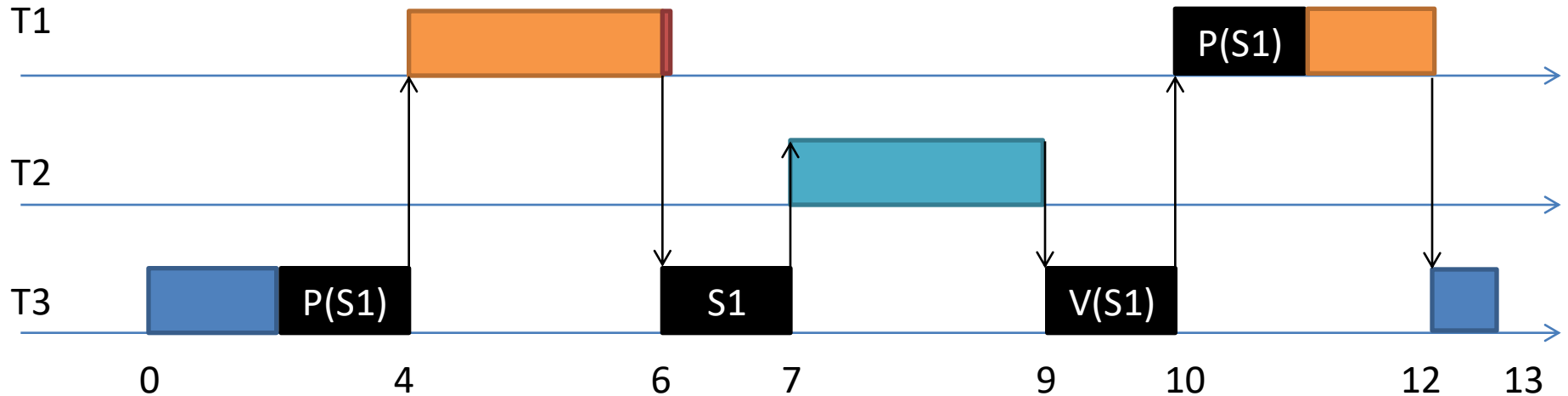
Sémaphore

- Objectif :
 - restreindre l'accès à des ressources partagées (par exemple un espace de stockage)
 - synchroniser les processus dans un environnement de programmation concurrente
 - ...
- inventé par Edsger Dijkstra en 1965
- Les trois opérations prises en charge sont **Init**, **P** et **V**.
 - L'opération **P** est en attente jusqu'à ce qu'une ressource soit disponible, ressource qui sera immédiatement allouée au processus courant.
 - **V** est l'opération inverse; elle rend simplement une ressource disponible à nouveau après que le processus a terminé de l'utiliser.
 - **Init** est seulement utilisée pour initialiser le sémaphore. Cette opération ne doit être utilisée qu'une seule et unique fois
- Le sémaphore binaire est une exclusion mutuelle (ou mutex). Il est toujours initialisé avec la valeur 1.

Le problème d'inversion de priorités

- Les tâches accèdent par mécanisme d'exclusion mutuelle à des ressources partagées
- Les tâches deviennent dépendantes les unes des autres
- **Problème** : une tâche de plus faible priorité peut bloquer une autre tâche de priorité plus forte quand elle est en section critique, c'est le problème de l'inversion de priorité.

Exemple



t0 – la tâche T3 démarre

t2 – T3 accède au sémaphore S1

t4 – T1, plus prioritaire, préempte T3

t6 – T1 demande S1 pris par T3. T3 reprend la main

t7 – T2, plus prioritaire, préempte à son tour T3

t9 – T2 termine et redonne la main

t10 – T3 relache le sémaphore, T1 peut s'exécuter, il y a eu inversion de priorité

t12 – T1 termine avec un retard

Solution : Méthode d'héritage de priorité

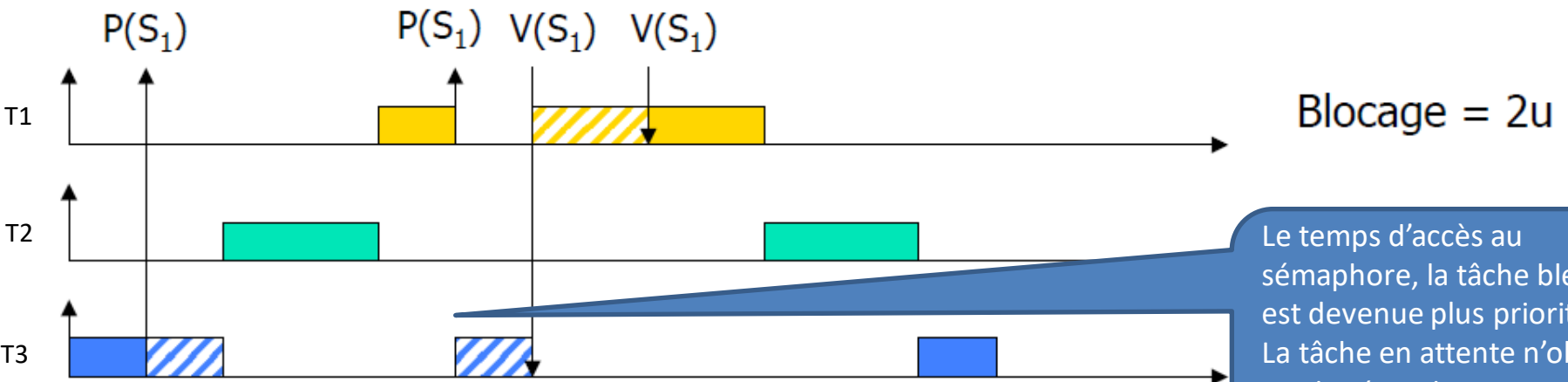
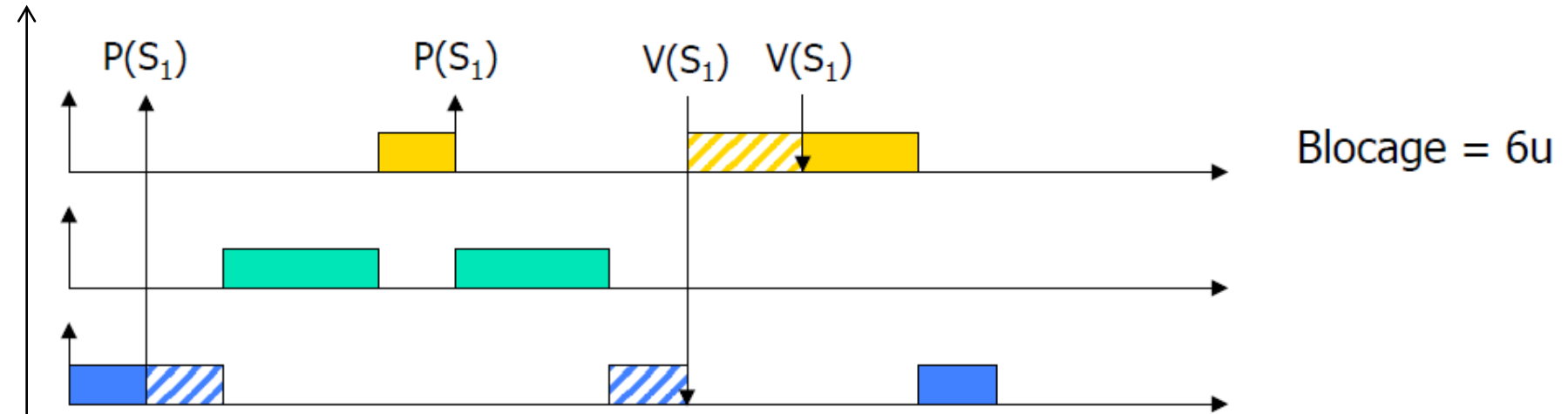
Héritage de priorité

(priority inheritance)

- Le système effectue une gestion dynamique des priorités. L'idée est de rajouter aux sémaphores la notion de **possesseurs**.
 - Le possesseur d'un sémaphore est la tâche qui a demandé et obtenu le droit de rentrer dans la zone d'exclusion mutuelle protégé par le sémaphore.
 - Dans l'héritage de priorité simple, les sémaphores gèrent leur file d'attente en tenant compte des priorités des tâches qui désirent prendre le sémaphore.
 - Lorsqu'une tâche détient un sémaphore et qu'une autre le réclame, la priorité de la tâche qui possède le sémaphore est **augmentée** au niveau de la priorité de la tâche qui le réclame.

Exemple d'héritage de priorité

Priorités



Le temps d'accès au sémaphore, la tâche bleue est devenue plus prioritaire. La tâche en attente n'obtient pas le sémaphore mais attend moins longtemps.

Priority Ceiling Protocol

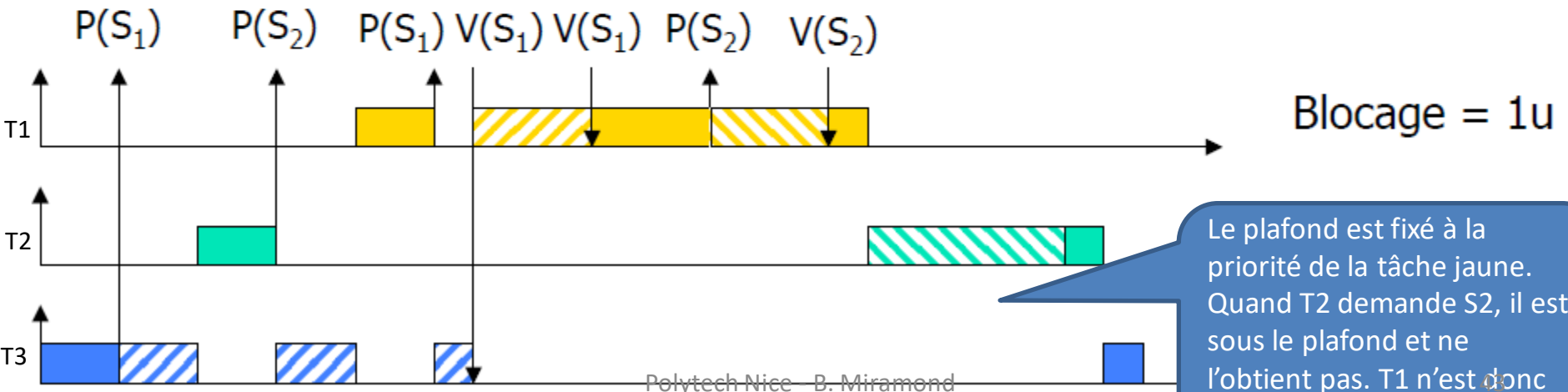
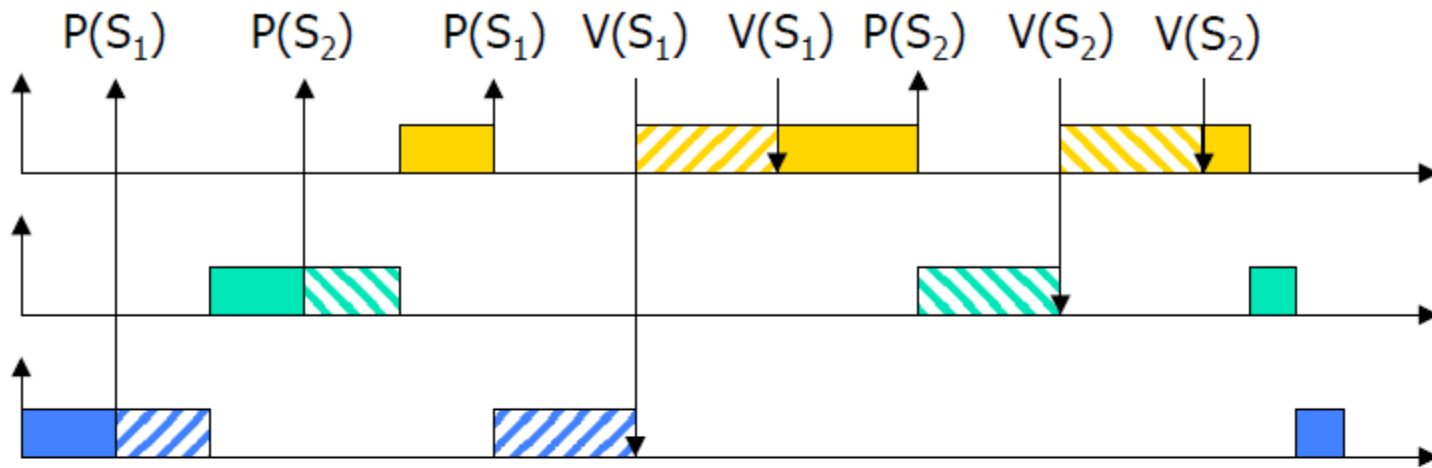
Malgré ce premier protocole,

- Les temps de blocage peuvent s'enchaîner
- Les élévations de priorité peuvent s'enchaîner
- Les **interblocages** restent possibles

Solution : **Plafond de priorité**

- La priorité plafonnée (statique) représente la priorité maximum des tâches qui l'utilisent
- Une tâche accède à un sémaphore lorsque sa priorité est strictement supérieure à toutes les priorités plafonnées **des sémaphores utilisés**
- Autrement dite, l'OS maintient une valeur qui représente la valeur maximale du plafond courant.
- Lorsqu'une tâche essaye d'exécuter une section critique, elle est suspendue sauf si sa priorité est supérieure au plafond de priorité de tous les sémaphores pris par les autres tâches.

Priority Ceiling Protocol

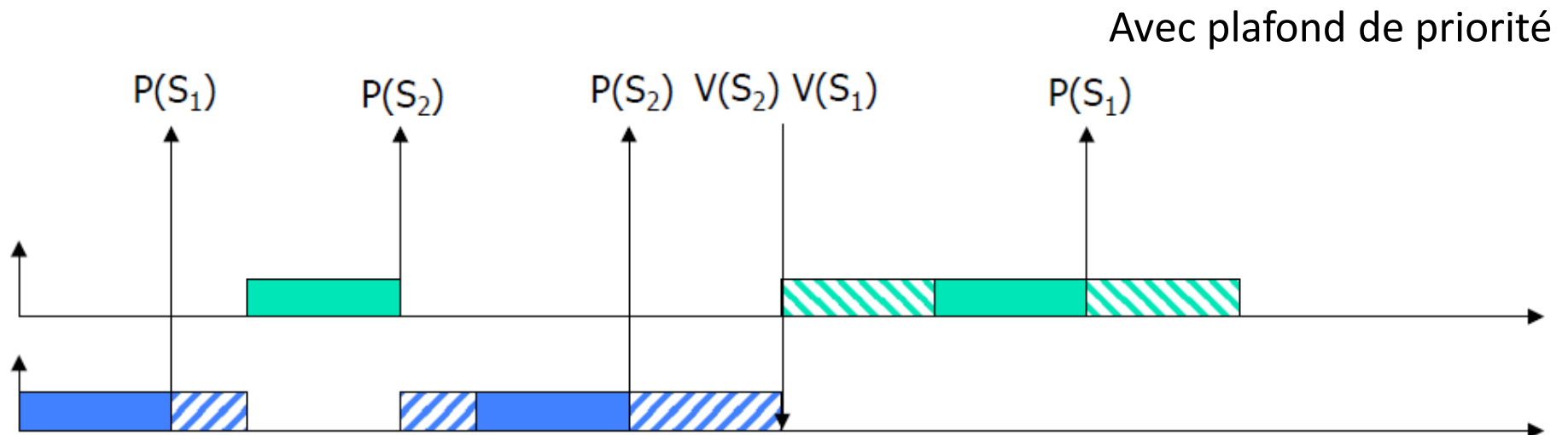
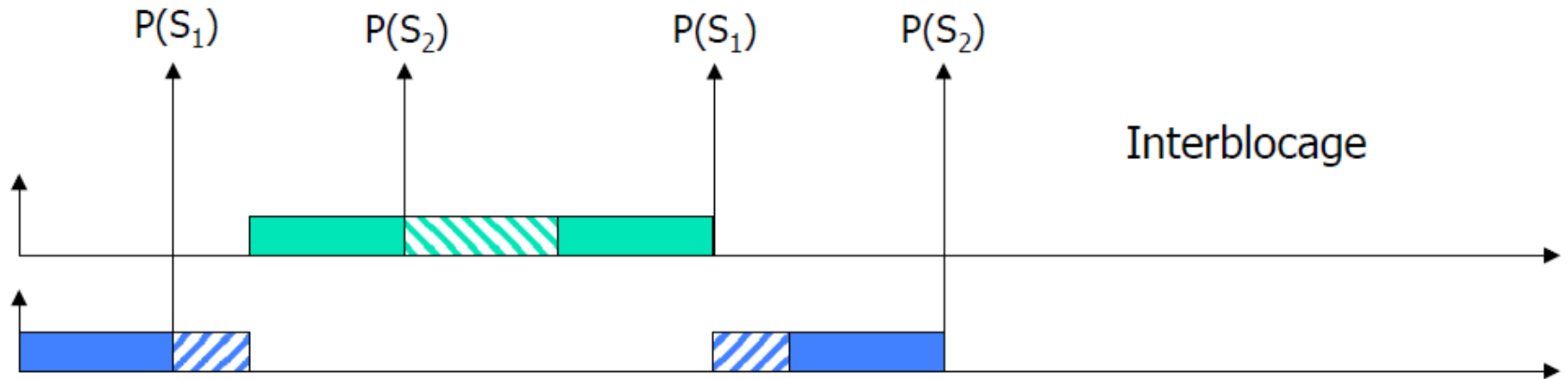


Le plafond est fixé à la priorité de la tâche jaune. Quand T2 demande S2, il est sous le plafond et ne l'obtient pas. T1 n'est donc bloquée que par T3.

Deadlock

- Un **interblocage**, **étreinte fatale**, ou ***deadlock*** est un phénomène qui peut survenir lorsque deux processus concurrents s'attendent mutuellement par l'intermédiaire de plusieurs accès exclusifs

Exemple d'interblocage (deadlock)



Conclusion















- Plusieurs méthodes d'ordonnancement selon les critères à respecter par l'application
 - Fréquences de tâches, RMS
 - Échéances, EDF
 - Laxité, LLF
- Dans le cas d'accès à des ressources partagées, les séquençements sont modifiés
- Des méthodes de gestion dynamique des priorités sont alors nécessaires

EXEMPLE DE LINUX RT


Comment rendre linux temps réel ?

- Distributions TR
 - Montavista
 - LynuxWorks
 - Redhat
 - Ubuntu
- Patch RT (CONFIG_PREEMPT_RT) de Linux :
 - Scheduler
 - Interruption préemptibles (thread-IRQ)
 - Horloge système plus précise (nanosec)...
 - sur une certaine plate-forme, le temps de latence de réponse à une interruption ne sera jamaïs supérieur à 20µs, par exemple
 - http://rt.wiki.kernel.org/index.php/Main_Page
- Micro-noyau
 - Xenomai
 - Windriver RT Linux (RTAI)

Fonctionnalités TR du patch config_preempt_rt

| Architecture | x86 | x86/64 |
|----------------------------------|---------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| Feature | | |
| Deterministic scheduler |  |  |
| Preemption Support |  |  |
| PI Mutexes |  |  |
| High-Resolution Timer |  |  1 |
| Preemptive Read-Copy Update |  2 |  2 |
| IRQ Threads |  4 |  4 |
| Full Realtime Preemption Support |  |  |

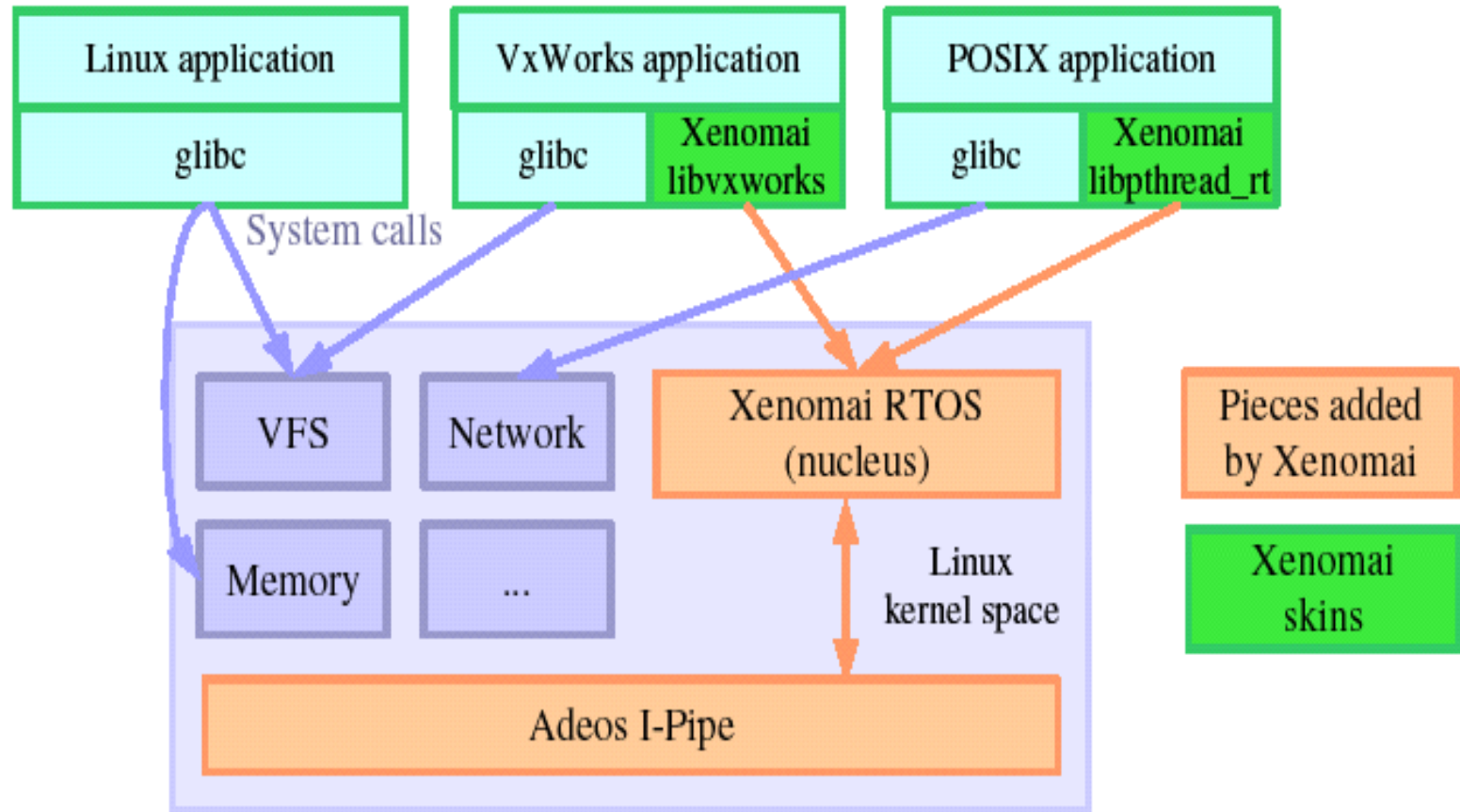
 Available in mainline Linux

 Available when Realtime-Preempt patches applied

1) Since kernel 2.6.24

2) Since kernel 2.6.25

Exemple du micro-noyau Xenomai



Ordonnanceur 2.6

Grande nouveauté : Ordonnanceur en $O(1)$

2 types de process

Process non-Temps Réel : chaque *process* se voit attribuer une valeur, appelée *nice value*, qui détermine la priorité et le temps d'exécution – ce temps d'exécution est appelé *timeslice* et correspond au temps maximum qu'un *process* peut rester en exécution avant de « laisser la place » à un autre. La valeur de *nice* est comprise entre -20 et 19 inclus, 0 étant la valeur par défaut et -20 correspond à la priorité la plus haute et au *timeslice* le plus élevé

Process Temps Réel : chaque *process* se voit attribuer une priorité comprise entre 1 et 99 inclus – les *process* non-Temps Réel se voient attribuer également cette priorité mais avec la valeur 0. 99 étant la priorité la plus élevée, on comprend alors pourquoi un *process* Temps Réel sera toujours prioritaire devant un autre non-Temps Réel.

Pour les *process* Temps Réel, il existe **deux politiques d'ordonnement** :

SCHED_FIFO : le *process* Temps Réel restera en exécution tant qu'aucun autre *process* Temps Réel de priorité supérieure strictement ne sera prêt ou tant qu'il ne passera pas à l'état bloqué. S'il est préempté, il reste tout de même en tête de la liste d'exécution des *process* de même priorité que lui. C'est donc lui qui sera de nouveau exécuté quand le *process* de priorité supérieure aura terminé son exécution. Idem s'il change de priorité, il sera placé en tête de la liste des *process* exécutables ayant cette priorité. Il ne passe en fin de cette liste que par appel explicite de sa part à une certaine fonction.

SCHED_RR : la politique est identique en tout point à la précédente, exception faite à la présence d'un *timeslice* pour les *process* de même priorité. Ainsi, s'il n'y a pas de *process* de plus forte priorité pour préempter celui en cours d'exécution, au bout d'un certain temps ce dernier sera placé en queue de la liste des *process* de même priorité.

Trois tests pour les linux RT

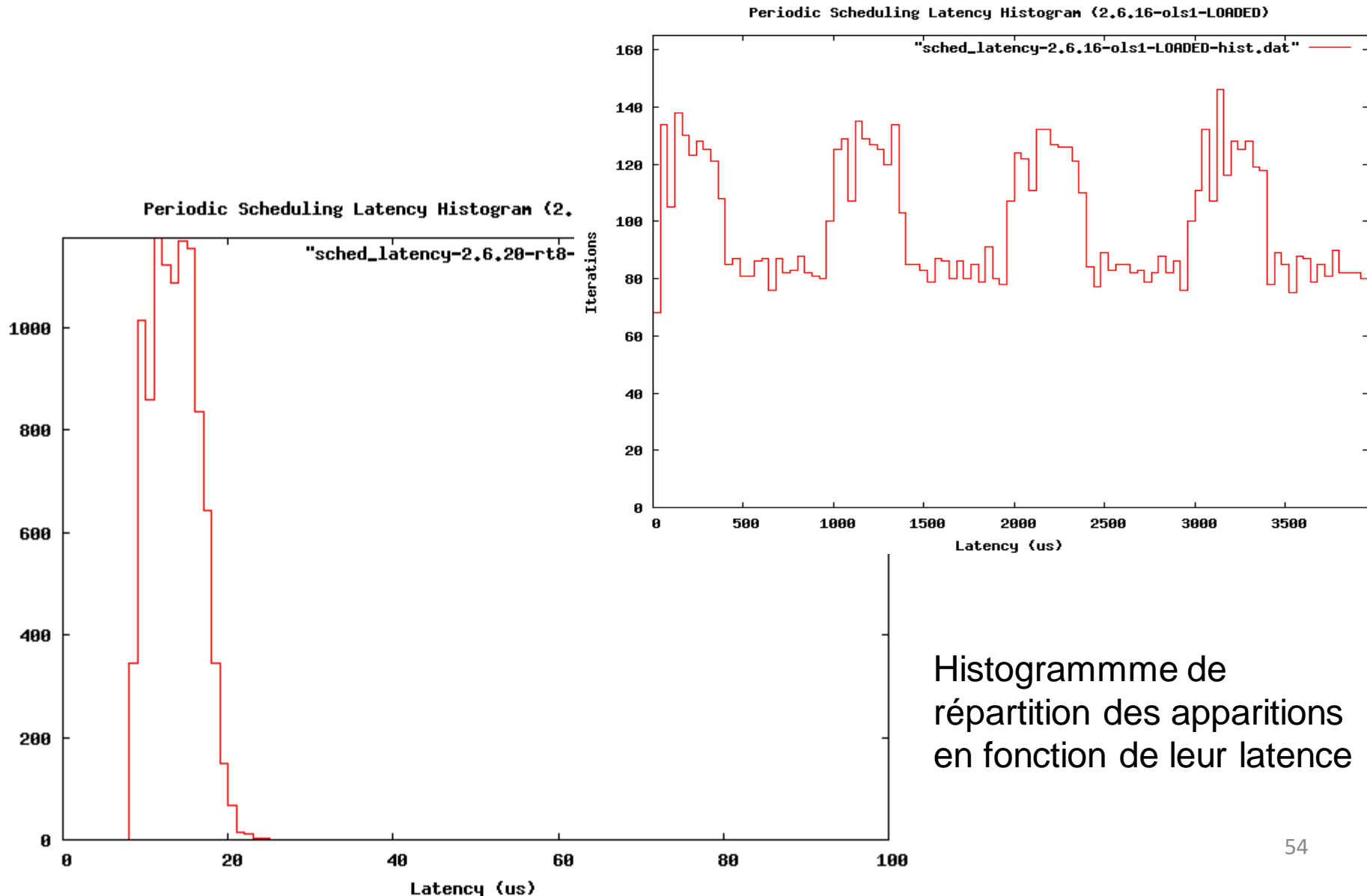
- Latence d'une tâche de forte priorité
 - Cyclic_test
- Inversion de priorité (cf. cours suivant)
 - classic_pi
- Préemption et latence de l'ordonnanceur
 - preempt_test
- Temps d'accès noyau
- ...

<http://www.kernel.org/pub/linux/kernel/people/tglx/rt-tests/rt-tests-0.49.tar.bz2>

Le bench cyclic_test

- sur un système non Temps Réel :
- `$> ./cyclictest -a -t -n -p99`
 - T:0(3431) P:99 I:1000 C: 100000 Min: 5 Act: 10 Avg: 14 Max: 39242
 - T: 1(3432) P:98 I:1500 C: 66934 Min: 4 Act: 10 Avg: 17 Max: 39661
- Ici, le résultat du même test sur un système Temps Réel :
- `$> ./cyclictest -a -t -n -p99`
 - T: 0 (3407) P:99 I:1000 C: 100000 Min: 7 Act:10 Avg: 10 Max: 18
 - T: 1 (3408) P:98 I:1500 C: 67043 Min: 7 Act: 8 Avg: 10 Max: 22

Latence d'ordonnancement



Histogramme de répartition des apparitions en fonction de leur latence

