Noyau d'un système d'exploitation
INF2610

Chapitre 9 : Ordonnancement de processus

Département de génie informatique et génie logiciel

POLYTECHNIQUE MONTRÉAL
AFFILIÉE À L'UNIVERSITÉ DE MONTRÉAL

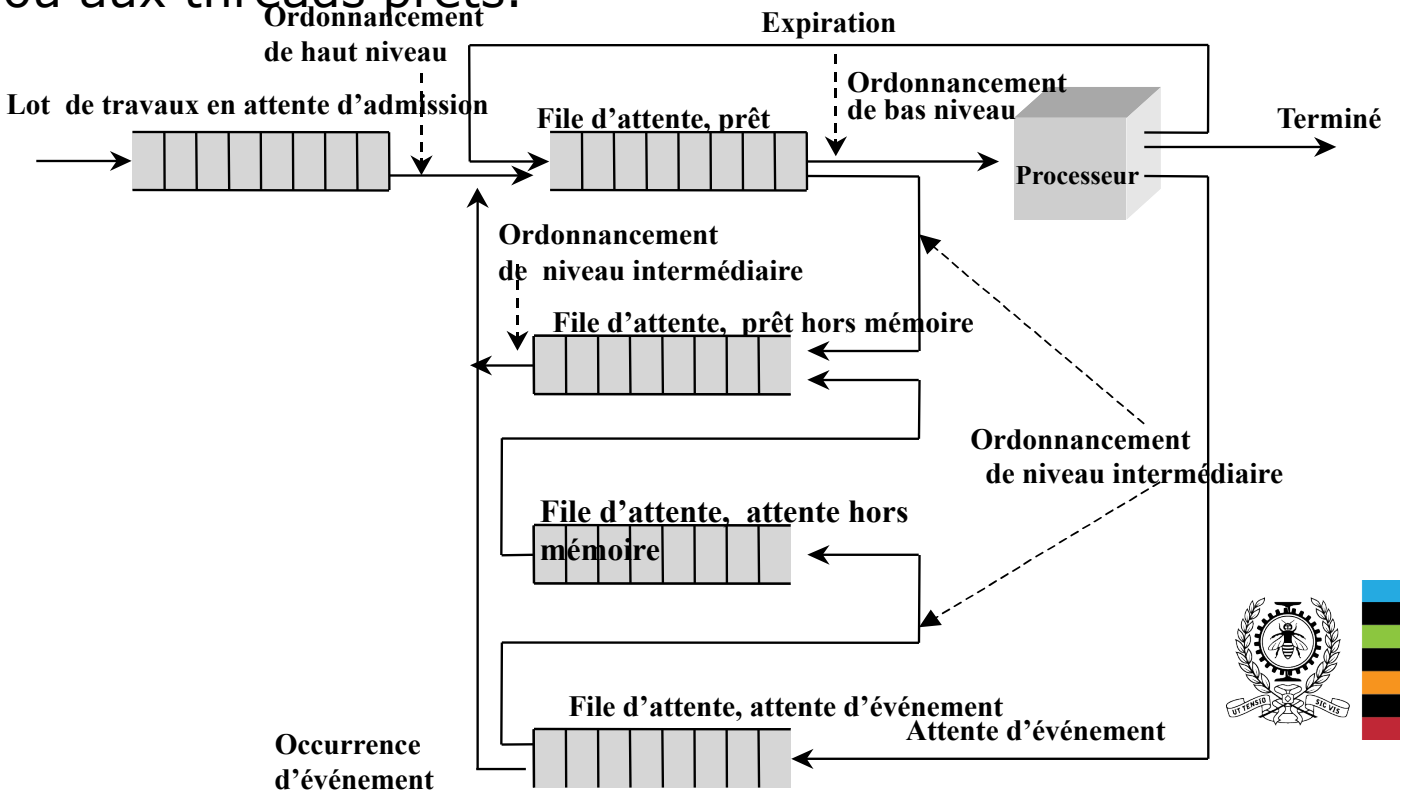
Automne 2017

Chapitre 9 - Ordonnancement de processus

- **Introduction**
- **Objectifs d'ordonnancement**
- **Politiques d'ordonnancement**
  - **Ordonnancement non préemptif**
  - **Ordonnancement préemptif**
- **Etudes de cas**
  - **Linux (noyau 2.6)**
  - **Windows**
- **Ordonnancement temps réel**
  - **Qu'est qu'un système temps réel**
  - **Qu'est qu'un système d'exploitation temps réel (RT-Linux)**
  - **Ordonnancement temps réel de tâches périodiques**

Introduction

- L'ordonnanceur (de bas niveau) c'est la partie du système d'exploitation qui se charge de gérer l'allocation de(s) processeur(s) aux processus ou aux threads prêts.



- L'ordonnancement doit se faire selon une politique bien réfléchie visant à répondre à des objectifs de performance.

Objectifs d'ordonnancement

- Pour le Système :
  - Maximiser le taux d'utilisation des processeurs et des autres ressources.
  - Maximiser la capacité de traitement (nombre de processus exécutés par unité de temps) ← systèmes de traitements par lots
  - Éviter le problème de famine (longue attente de temps CPU).
  - Minimiser le nombre et la durée des changements de contexte.
  - Respecter les échéances de tâches temps réel (terminer l'exécution avant leurs échéances (deadlines)) ← systèmes temps réel.
- Pour l'utilisateur :
  - Minimiser le temps de séjour des processus.
  - Minimiser le temps de réponse des processus.
  - Minimiser le temps d'attente d'exécution.

Objectifs d'ordonnancement (2)

Critères de performance :

- Taux d'utilisation d'un processeur.
- Capacité de traitement : nombre de processus traités par unité de temps.
- Temps de séjour d'un processus (temps de rotation ou de virement) : temps entre l'admission du processus et sa sortie.
- Temps de réponse: temps entre l'entrée d'un processus et le moment où il commence à être traité.
- Temps d'attente d'un processus : somme des périodes que le processus passe à l'état prêt.

=> temps moyen de séjour et temps moyen d'attente

Objectifs d'ordonnancement (3)

Difficultés :

- L'exécution d'un processus est une séquence alternée de calculs CPU (rafales CPU) et d'attentes d'événements (E/S, attente d'un sémaphore, etc.) ➔ Deux classes de processus :
  - CPU-Bound : beaucoup de calcul et peu d'E/S (interactifs ou non),
  - IO-Bound : peu de calcul et beaucoup d'attente d'E/S, et

  ➔ Faut-il favoriser les processus IO-Bound ? Si oui, comment déterminer la classe d'un processus ?

- Comment gérer la répartition des processus sur les différents processeurs ?
  - Problème de copies multiples dans les caches des processeurs, si les processus qui partagent des données sont affectés à des processeurs différents.
  - Problème de répartition équitable de la charge entre les processeurs (migration entre processeurs (différents)).

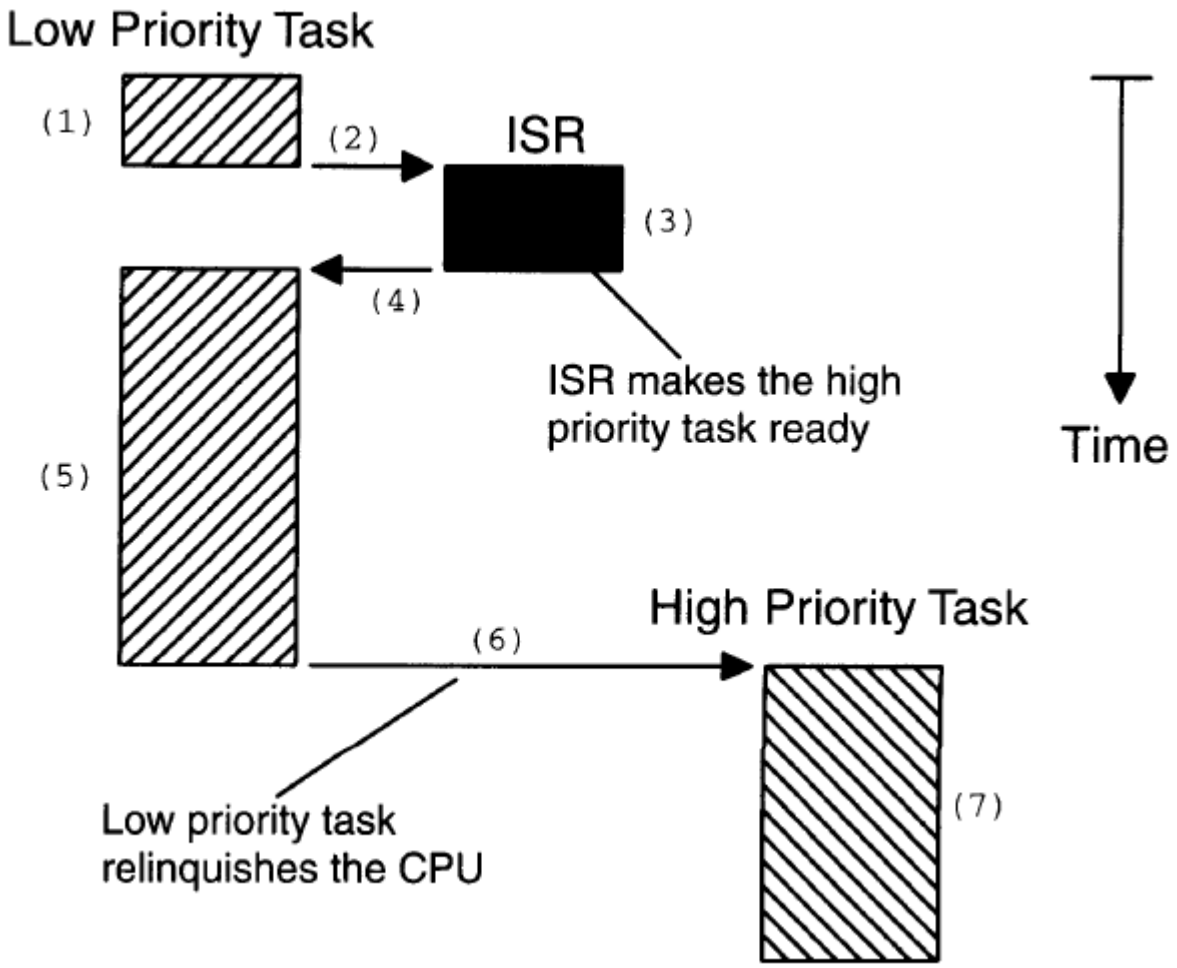
Politique d'ordonnancement

- Choix du processus à exécuter (cas d'un seul processeur) ?
  - Premier arrivé, premier servi.
  - Plus prioritaire (priorités statiques ou dynamiques).
  - Plus court temps ….
- Le temps d'allocation du processeur au processus choisi :
  - Allocation jusqu'à terminaison ou libération volontaire (ordonnancement non préemptif).
  - Temps d'allocation limité fixe ou variable (ordonnancement préemptif).
- Quand appeler l'ordonnanceur ?
  - Le processus en cours se termine, se bloque ou cède le processeur.
  - Le temps alloué a expiré.
  - L'arrivée d'un processus plus prioritaire.

Ordonnancement non préemptif

- Le système d'exploitation choisit le prochain processus à exécuter :
  - Premier arrivé, Premier servi (FCFS, First-Come First-Served) ou
  - Plus court d'abord (SPF, Short Process First ou SJF Short Job First).
  - Plus prioritaire d'abord (priorite = (temps d'attente + temps d'exécution) / temps d'exécution).
- Il lui alloue le processeur jusqu'à ce qu'il se termine, se bloque (en attente d'un événement) ou cède le processeur. Il n'y a pas de réquisition.

Ordonnancement non préemptif (2)



MicroC/OS-II: The Real Time Kernel , Jean J. Labrosse

Ordonnancement non préemptif (3) : FCFS

Processus	Exécution	Arrivée
A	3	0
B	6	1
C	4	4
D	2	6
E	1	7

AAABBBBBBCCCCDDE

Temps de séjour moyen : 7.6

Temps moyen d'attente : 4.4

Nombre de changements de contexte : 5

Remarque :

Temps moyen d'attente élevé si de longs processus sont exécutés en premier.

Ordonnancement non préemptif (4) : SPF
Plus court en premier (Short Process First)

Processus	Exécution	Arrivée
A	3	0
B	6	1
C	4	4
D	2	6
E	1	7

AAABBBBBBEDDCCCC
Temps de séjour moyen : 6.4
Temps moyen d'attente : 3.2
Nombre de changements de contexte : 5

Remarque : Meilleur temps moyen d'attente.

Ordonnancement non préemptif (5) : SPF
Plus court en premier (Short Process First)

- Parmi les processus prêts, le processus élu est celui dont la prochaine rafale CPU est la plus courte.

Comment estimer le temps de la prochaine rafale CPU ?

- Le temps de la prochaine rafale CPU de chaque processus est estimé en se basant sur le comportement passé du processus.

- Soient Si et Ti les temps d'exécution estimé et mesuré de la $i^{ième}$ rafale CPU. Les estimations successives des rafales CPU sont :

  S0 ; pour la première

  $S1 = \alpha T0 + (1- \alpha)S0$; pour la seconde

  $S2 = \alpha T1 + (1- \alpha)S1$; pour la troisième

  ....

$0 \leq \alpha \leq 1$

Ordonnancement non préemptif (6): à priorités

Processus	Exécution	Arrivée
A	3	0
B	6	1
C	4	4
D	2	6
E	1	7

- Priorité = (temps d'attente + temps d'exécution) / temps d'exécution
- AAABBBBBBEDDCCCC
- Le temps moyen de séjour : 6,4
- Le temps moyen d'attente : 3,2

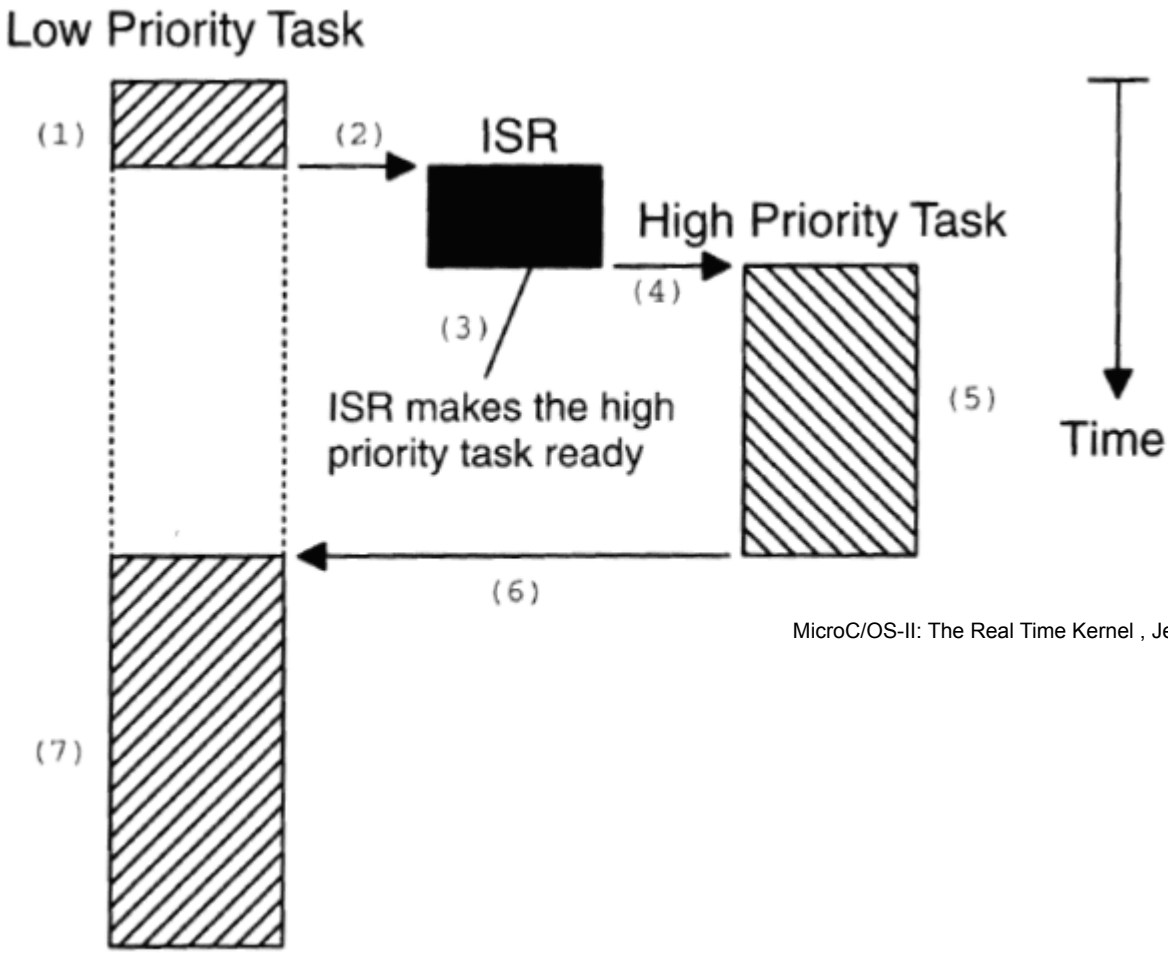
Ordonnancement préemptif (avec réquisition)

- Pour éviter qu'un processus monopolise le processeur, les ordinateurs ont une horloge électronique qui génère périodiquement une interruption.
- A chaque interruption d'horloge, le système d'exploitation reprend la main et décide si le processus courant doit :
  - poursuivre son exécution ou
  - être suspendu pour laisser place à un autre.
- S'il décide de suspendre l'exécution au profit d'un autre, il doit d'abord sauvegarder l'état des registres du processeur avant de charger dans les registres les données du processus à lancer (commutation de contexte).
- Cette sauvegarde est nécessaire pour pouvoir poursuivre ultérieurement l'exécution du processus suspendu.
-

Ordonnancement préemptif (2)

- Le processeur passe donc d'un processus à un autre en exécutant chaque processus pendant quelques dizaines de millisecondes.
- Le temps maximal d'allocation du processeur au processus est appelé quantum. (Linux : **int sched_rr_get_interval(pid_t** *pid*, **struct timespec *** *tp*);)
- La commutation entre processus doit être rapide (temps nettement inférieur au quantum).
- Le processeur, à un instant donné, n'exécute réellement qu'un seul processus.
- Pendant une seconde, le processeur peut exécuter plusieurs processus et donne ainsi l'impression de parallélisme (pseudo-parallélisme).
- Le système peut aussi suspendre l'exécution d'un processus en cours si un processus plus prioritaire passe à l'état prêt.

Ordonnancement préemptif (3)



MicroC/OS-II: The Real Time Kernel , Jean J. Labrosse

Ordonnancement préemptif (4) : SRT Shortest Remaining Time

- Lorsqu'un processus arrive, l'ordonnanceur compare le temps estimé de son exécution avec celui du processus actuellement en exécution (version préemptive du SPF).
- Si ce temps est plus petit, on exécute immédiatement le nouveau processus.

AAABCCCCEDDBBBBB

Temps moyen de séjour : 5.8

Temps moyen d'attente : 2.6

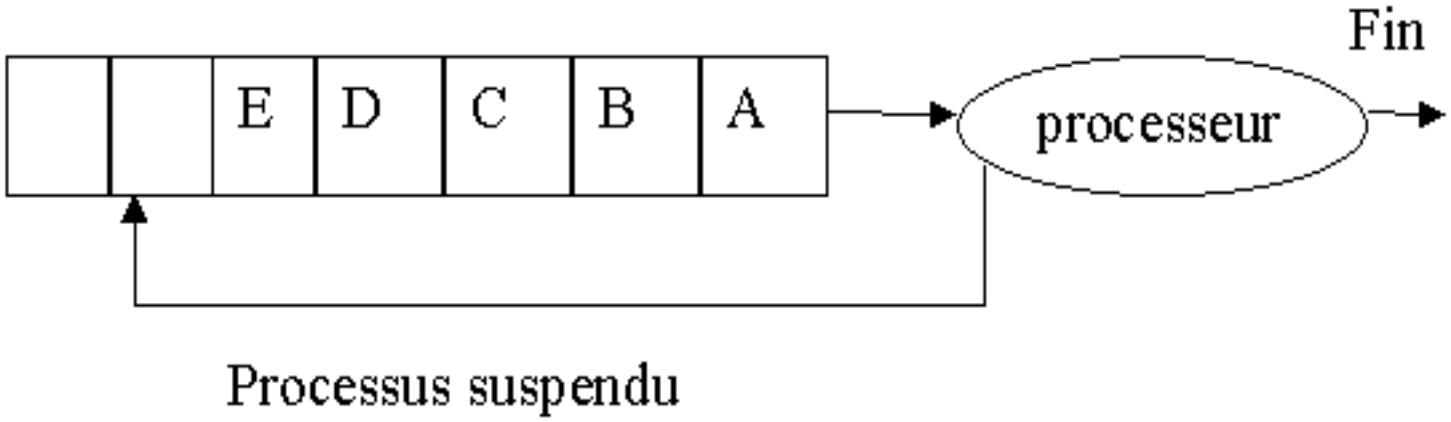
Nombre de changements de contexte : 6

Processus	Exécution	Arrivée
A	3	0
B	6	1
C	4	4
D	2	6
E	1	7

Ordonnancement préemptif (5) : circulaire (tourniquet/round robin)

- Algorithme ancien, simple, fiable et très utilisé en combinaison avec d'autres.

- Il mémorise dans une file (FIFO : First In First Out), la liste des processus en attente d'exécution (prêts).



Ordonnancement préemptif (6) : circulaire (tourniquet/round robin)

Choix de la valeur du quantum :

- Algorithme équitable mais sensible au choix du quantum.
- Un quantum trop petit provoque trop de commutations de processus et abaisse l'efficacité du processeur.
- Un quantum trop élevé augmente le temps de réponse des courtes commandes en mode interactif.
- Il était de 1 seconde dans les premières versions d'UNIX.
- Il varie entre 10 et 100 ms.

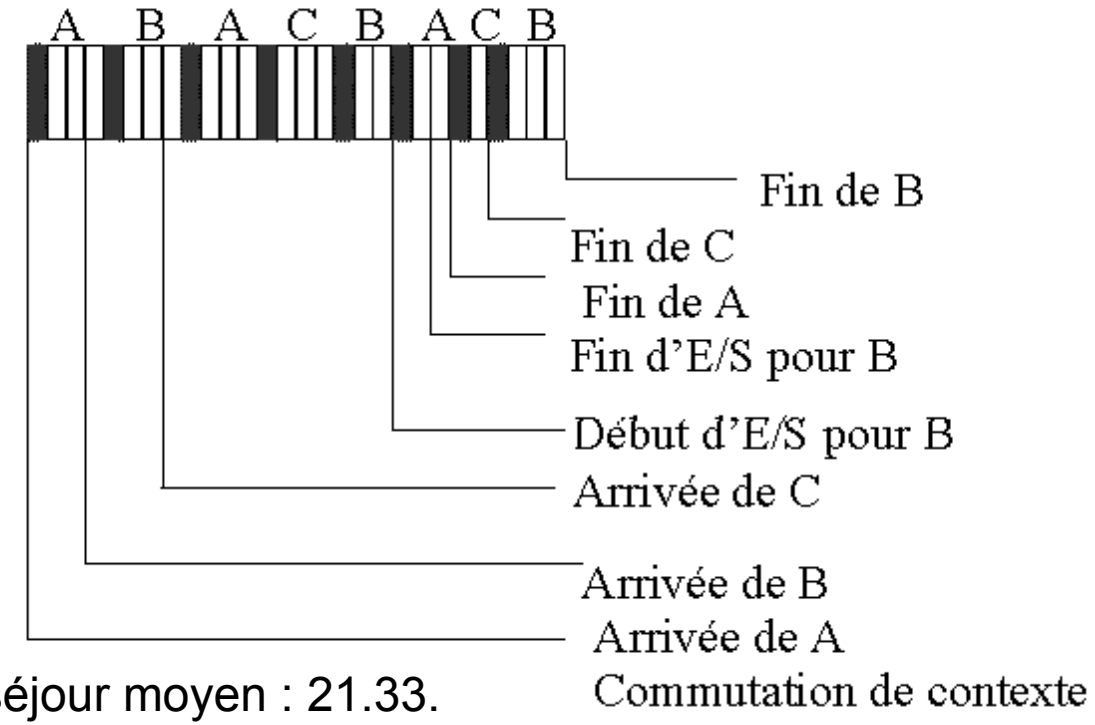
=> quantum = (10 à 100) fois le temps de commutation.

Ordonnancement préemptif (7) : circulaire (tourniquet/round robin)

Exemple 1 :

Quantum = 3 unités
Commutation de contexte = 1 unité

Processus	Exécution	Arrivée
A	8	0
B	5(2)3	3
C	4	7



- Le temps de séjour moyen : 21.33.
- Le temps moyen d'attente : 14
- Nombre de changements de contexte : 8

Ordonnancement préemptif (8) : circulaire (tourniquet/round robin)

Exemple 2 :

Quantum = 5 unités
Commutation de contexte = 1 unité

Processus	Exécution	Arrivée
A	8	0
B	5(2)3	3
C	4	4

Quantum = 5 unités

A A A A A B B B B B C C C C A A A B B B

- Le temps de séjour moyen : (21+22+13)/3 (soit 18,66).
- Le temps d'attente moyen : 11,33.
- Nombre de changements de contexte : 5

Ordonnancement préemptif (9) : circulaire (tourniquet/round robin)

Exemple 3 :

Quantum = 7 unités
Commutation de contexte = 1 unité

Processus	Exécution	Arrivée
A	8	0
B	5(2)3	3
C	4	4

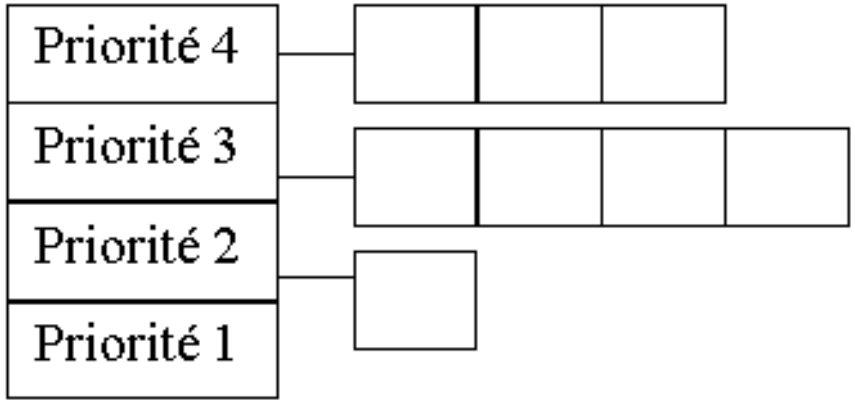
Quantum = 7 unités

A A A A A A A B B B B B C C C C A B B B

- Le temps de séjour moyen : (21+22+15)/3 (soit 19.33).
- Le temps d'attente moyen : 12.
- Nombre de changements de contexte : 5

Ordonnancement préemptif (10) : à files multiples

- Une priorité est attribuée à chaque processus (priorité statique ou dynamique).
- Les processus de même priorité sont dans une même file d'attente.
- Il y a autant de files d'attente qu'il y a de niveaux de priorités.
- Les processus de chaque file sont ordonnancés selon l'algorithme du Tourniquet (Round Robin avec un quantum variable ou fixe) ou FCFS.



Ordonnancement préemptif (11) : à files multiples

Problèmes

- Pas d'équité : l'exécution des processus moins prioritaires peut être constamment retardée par l'arrivée de processus plus prioritaires.

- Inversion des priorités :

  – un processus moins prioritaire détient une ressource nécessaire à l'exécution d'un processus plus prioritaire.

  – Le processus moins prioritaire ne peut pas s'exécuter car il y a constamment des processus actifs plus prioritaires.

Si l'attente de la ressource est active dans le processus plus prioritaire ➔ une attente active infinie.

Ordonnancement préemptif (12) : à files multiples

Comment attribuer les priorités aux différents processus ?

- Pour empêcher les processus de priorité élevée de s'exécuter indéfiniment, l'ordonnanceur :
  - diminue régulièrement la priorité du processus en cours d'exécution (priorité dynamique) et
  - augmente progressivement celles des processus en attente.
- La priorité du processus en cours est comparée régulièrement à celle du processus prêt le plus prioritaire (en tête de file). Lorsqu'elle devient inférieure, la commutation de contexte a lieu.
- Dans ce cas, le processus suspendu est inséré en queue de file correspondant à sa nouvelle priorité.

Ordonnancement préemptif (13) : à files multiples

Comment attribuer les priorités aux différents processus ?

- Pour favoriser les processus qui font beaucoup d'E/S (IO-bound), ils doivent acquérir le processeur dès qu'ils le demandent, afin de leur permettre de lancer leurs requêtes suivantes d'E/S.

- Pour éviter qu'il y ait beaucoup de commutations pour les processus consommateurs de temps CPU (CPU-bound), il est préférable d'allouer un plus grand quantum à ces processus (quantum variable).

- **Comment déterminer si un processus est CPU-bound ou IO-bound ?**

➔ mesure de différents temps de séjour du processus dans les états prêt, exécution et/ou bloqué.

Études de cas :
Linux et Windows
Algorithmes d'ordonnancement utilisés
Attribution et évolution des priorités

Cas de Linux (noyau 1.2 - noyau 2.2)

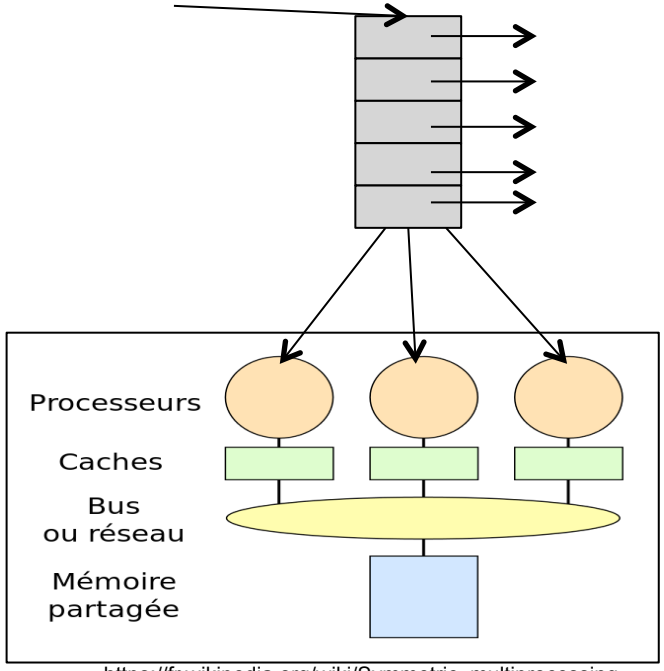
- **Linux 1.2** : utilise un ordonnancement circulaire et une file circulaire de processus prêts.
- **Linux 2.2** : introduit l'idée de combiner différentes politiques dans un ordonnancement à files multiples ➔ classes de processus (**sched_setscheduler, sched_getscheduler, etc.)** :
  - **Les FIFO temps réel** (SCHED_FIFO) : les plus prioritaires et préemptibles par des processus de la même classe ayant un niveau de priorité plus élevé.
  - **Les tourniquets temps réel** (SCHED_RR) : un quantum de temps est associé à chaque processus de cette classe (ordonnancement circulaire).
  - **Les processus en temps partagé** (SCHED_OTHER, classe par défaut) : un quantum de temps est associé à chaque niveau de priorité.

Les processus temps réel ont des priorités plus élevées que celles des processus en temps partagé.

⇒ Ordonnancement préemptif à files multiples de processus (priorité dynamiques).

Cas de Linux (noyau 1.2 - noyau 2.2) (2)

- **Linux 2.2** : supporte aussi des architectures multiprocesseur symétrique (SMP) avec une seule file d'attente « runqueue » commune à tous les processeurs.



https://fr.wikipedia.org/wiki/Symmetric_multiprocessing

Avantages : Meilleure utilisation de processeurs et équitable vis-à-vis des processus.

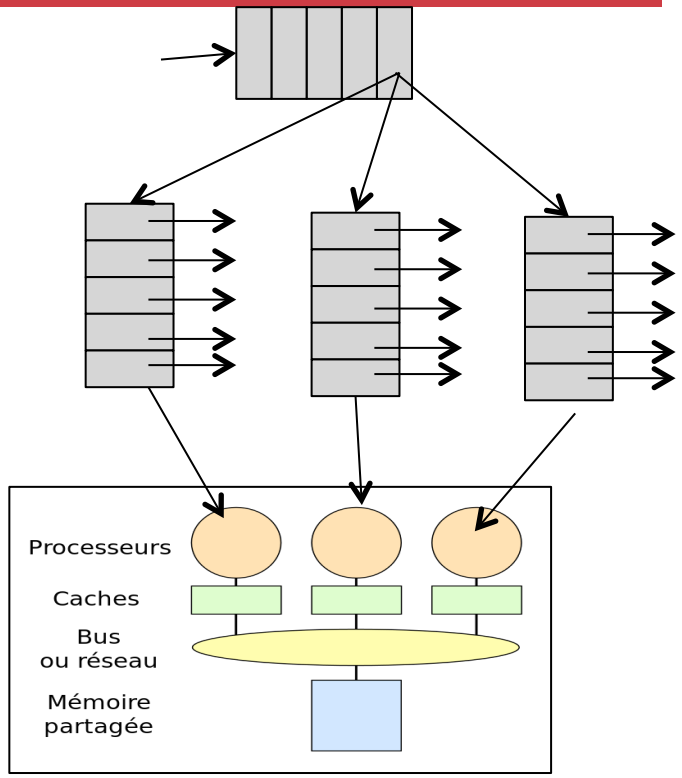
Inconvénients : pas « scalable » (accès en exclusion mutuelle à la file globale) et une plus faible localité au niveau des caches (plus de « cache miss »).

Cas de Linux (noyau 2.6) (3)

- L'ordonnancement préemptif et à priorité avec files multiples (**140 niveaux de priorité**). Les priorités sont dynamiques.
- L'ordonnanceur Linux **gère les threads** et non pas les processus : threads temps réel (**SCHED_FIFO, SCHED_RR**) et threads en temps partagé (**SCHED_OTHER**).
- Les threads temps réel ont des priorités allant de 0 à 99 (les plus élevées).
- Les threads en temps partagé (classe par défaut) ont des priorités allant de 100 à 139 (priorité statique = 120 + nice).
- Linux associe une valeur **nice** à chaque thread. La valeur par défaut est 0 et peut être modifiée par **nice(val**) avec val∈[-20,19]. Seul l'administrateur peut appeler nice(val) avec val <0.

Cas de Linux (noyau 2.6) (4)

- L'ordonnanceur utilise une file d'attente (runqueue) par processeur.
- **Avantages** : est plus « scalable » (pas de contention sur la runqueue de chaque processeur) et permet une meilleure localité au niveau des caches.
- **Inconvénients :** risque de charge déséquilibrée (avec sous utilisation de processeurs) et moins équitable vis-à-vis des processus.



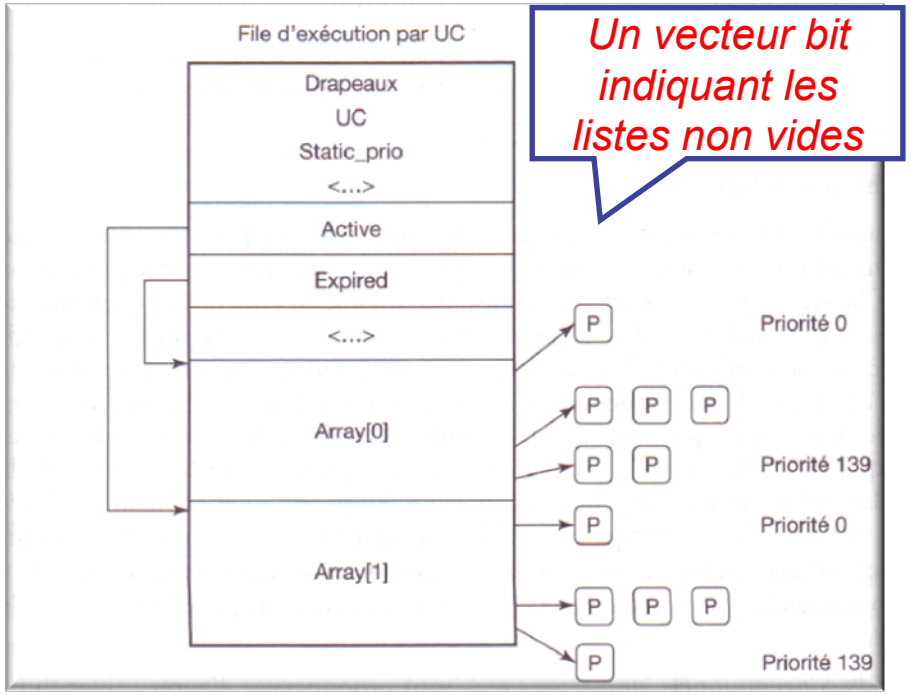
https://fr.wikipedia.org/wiki/Symmetric_multiprocessing

➔ « Load Balancer » pour rééquilibrer périodiquement la charge entre les processeurs avec migration possible entre les « runqueues ». Load Balancer est aussi appelé si une file se vide (thread idle).

➔ Chaque processus/thread a un masque d'affinité (1 bit par processeur) indiquant sur quel(s) processeurs il peut s'exécuter (ce masque est dupliqué lors d'un fork pour le processus fils).

Cas de Linux (noyau 2.6) (5)

- Chaque file d'attente est composée de deux tableaux «Active»/«Expired».
- Il sélectionne le thread le plus prioritaire, en tête de file Active.
- Si le thread consomme son quantum restant, il est déplacé vers le tableau Expired.
- Lorsque le tableau Active devient vide, l'ordonnanceur permute les pointeurs Active et Expired.
- Il attribue des quanta variables selon leurs priorités (quanta plus élevés pour les plus prioritaires (ex. pri.100 -> qt = 800ms, pri. 139 -> qt = 5ms).



If (SP < 120): qt = (140 − SP) × 20
if (SP >= 120): qt= (140 − SP) × 5

Cas de Linux (noyau 2.6) (6)

- Dans le but de favoriser les threads IO-bound (notamment les threads interactifs), l'ordonnanceur associe à chaque thread une **variable sleep_avg** qui est une heuristique utilisée. Sa valeur est entre 0 et MAX_SLEEP_AVG (10).

- Lorsqu'un thread passe de l'état bloqué à prêt, sleep_avg est incrémenté de la durée passée à l'état bloqué (jusqu'au maximum *MAX_SLEEP_AVG*).

- Lorsqu'il est suspendu, sleep_avg est décrémentée du temps CPU consommé (jusqu'au minimum 0).

- sleep_avg est utilisée pour mapper le bonus du thread sur des valeurs allant de **-5 à 5**.

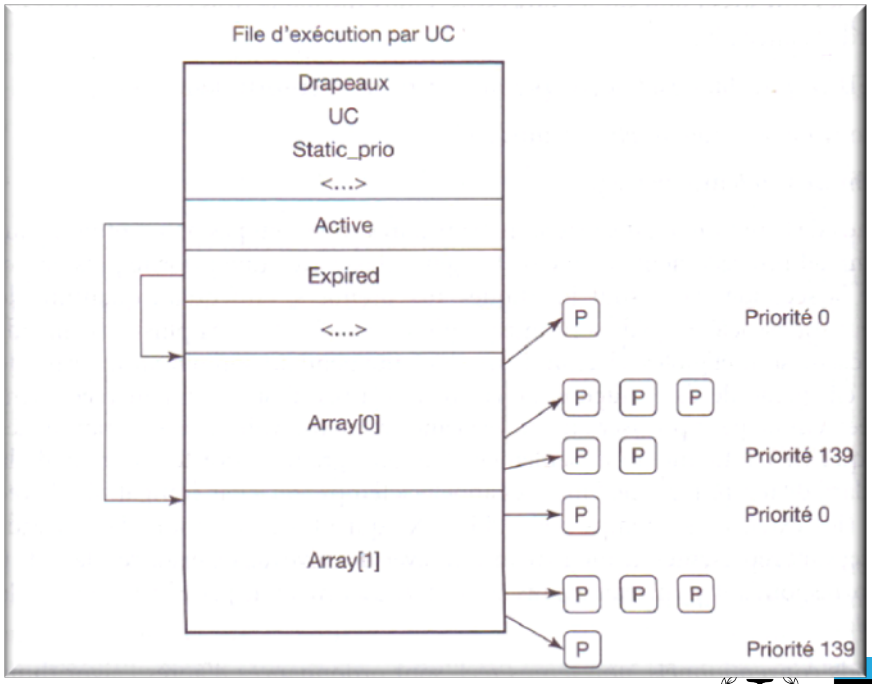
- L'ordonnanceur recalcule le niveau de priorité d'un thread au moment où il est déplacé vers le tableau Expired.

DP = SP − bonus + 5
DP = min(139, max(100, DP))

Cas de Linux (noyau 2.6) (7)

- Un thread qui passe à l'état bloqué est déplacé vers la « waitqueue ».
- La « waitqueue » contient une file d'attente par type d'événement.
- La décision d'ordonnancement est indépendante du nombre de threads dans le système (complexité en temps est O(1) par rapport au nombre de threads).

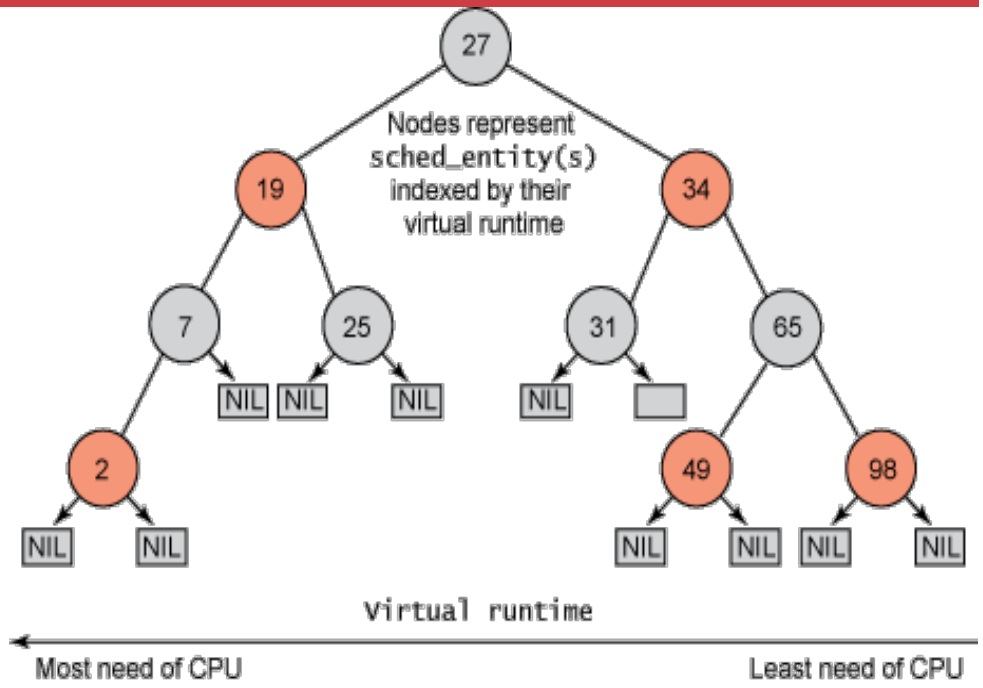
➔ Problème : sleep_avg nécessite un calcul compliqué et ne favorise pas toujours les threads interactifs.



Cas de Linux (noyau 2.6.21 ...) (8)

http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/

- L'ordonnanceur CFS (Completely Fair Scheduler) est utilisé pour les threads en temps partagé).
- Il vise à équilibrer le temps alloué à chaque thread en utilisant un arbre binaire de recherche.
- La clé de chaque nœud est « virtual runtime » d'un thread prêt = temps pondéré qu'il a passé en exécution.
- Ce temps croit moins vite pour les threads plus prioritaire.
- Le thread avec la plus petite clé (vruntime) est sélectionné.



Cas de Linux (noyau 2.6.21 ...) (9)

http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/

- Le quantum alloué dépend du nombre des threads prêts et de leurs poids.
- Le poids d'un thread = 1024/(1.25)^(nice)
- Soient deux A et B avec nice =0 (priorité statique = 120 + nice), la portion de temps CPU allouée à chaque thread est : 1024/(1024*2) = 0.5
- Si la priorité de A est 121 et celle de B est 120, son poids serait à 820 et sa portion de temps CPU devient : 820/(1024+820)) = 0.45.
  Celle de B est : (1024/(1024+820)) = 0.55.
  Le thread A va avoir 10% de temps CPU de moins que B.

```
static const int prio_to_weight[40] = {
/* -20 */     88761,     71755,     56483,     46273,    36291,
 /* -15 */     29154,     23254,     18705,     14949,   11916,
/* -10 */      9548,      7620,      6100,      4904,     3906,
/*  -5 */      3121,      2501,      1991,      1586,     1277,
/*   0 */      1024,       820,       655,       526,      423,
/*   5 */       335,       272,       215,       172,      137,
/*  10 */       110,        87,        70,        56,       45,
 /*  15 */        36,        29,        23,        18,       15, };
```

Cas Windows

http://msdn.microsoft.com/en-us/library/windows/desktop/ms685100%28v=vs.85%29.aspx

- L'ordonnanceur de Windows est **préemptif et à priorité avec files multiples** (32 niveaux de priorité). Les priorités sont dynamiques.

- Windows **ordonnance les threads** sans se préoccuper des processus.

- Windows ne contient pas un module autonome qui se charge de l'ordonnancement. Le code de l'ordonnanceur est exécuté par un thread ou un DPC (appel de procédure différé).

- Le code de l'ordonnanceur est exécuté par un thread dans les trois cas suivants :

  – Il exécute une fonction qui va **basculer son état vers bloqué** (E/S, sémaphore, mutex, etc). Cette fonction fait appel au code de l'ordonnanceur pour sélectionner un successeur et réaliser la commutation de contexte.

Cas de Windows (2)

- Il exécute une fonction qui **signale un objet** (sémaphore, mutex, événement, etc). Cette fonction fait appel au code de l'ordonnanceur pour vérifier l'existence d'un thread prêt plus prioritaire. Si c'est le cas, un basculement vers ce thread est réalisé.

- **Son quantum a expiré**. Le thread passe en mode noyau puis exécute le code de l'ordonnanceur pour vérifier l'existence d'un thread prêt plus prioritaire. Si c'est le cas, un basculement vers ce thread est réalisé.

- L'ordonnanceur est également appelé à la fin d'une E/S et à l'expiration d'un timeout.

  → L'interruption de fin d'E/S ou celle associée au timeout va programmer un **DPC** (appel de procédure différé). Ce DPC va exécuter le code de l'ordonnanceur pour vérifier si le thread « débloqué » est plus prioritaire que le thread courant. Si c'est le cas, un basculement vers le thread débloqué est réalisé.

Cas de Windows (3)

- L'API Win32 fournit deux fonctions pour influencer l'ordonnancement des threads :
  - SetPriorityClass qui définit la classe de priorité de tous les threads du processus appelant (realtime, high, above normal, normal, below normal, idle).
  - SetThreadPriority qui définit la priorité relative d'un thread par rapport à la classe de priorité de son processus (timecritical, highest, above normal, normal, below normal, lowest, idle).

Les combinaisons de classe de priorité et de priorité relative sont mappées sur 32 priorités de threads absolues

→ Priorités de base

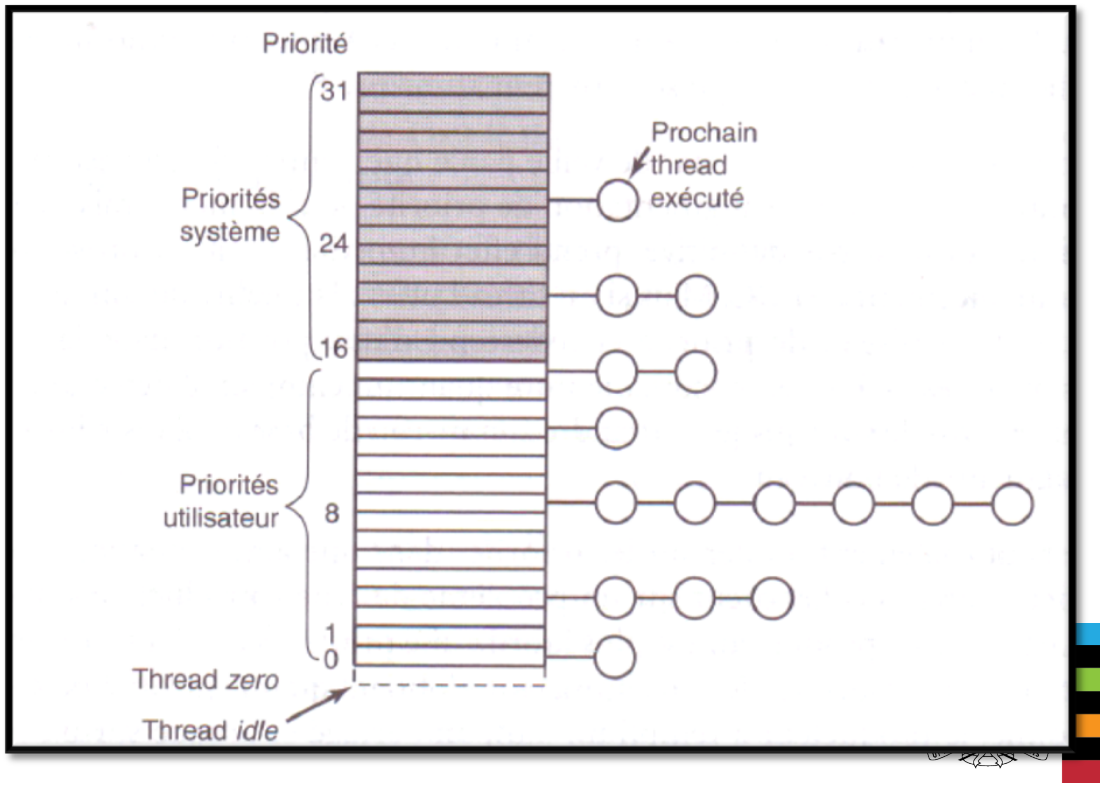
		Priorités de classe de processus Win32					
		Realtime	High	Above normal	Normal	Below normal	Idle
Priorités de thread Win32	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

Cas de Windows (4)

- Chaque thread a une priorité de base et une priorité courante qui peut être plus élevée que celle de base mais jamais plus basse.

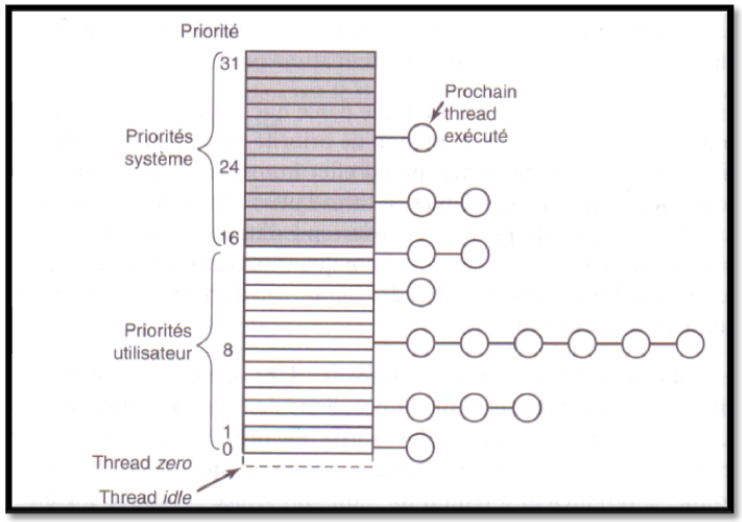
- L'ordonnanceur sélectionne le thread le plus prioritaire.

- Ce thread va s'exécuter pendant au maximum un quantum (20ms par défaut, 120 ms)



Cas de Windows (5)

- On distingue quatre classes de processus légers :
  - **Les threads temps réel** : chaque thread de cette classe a une priorité comprise entre 16 et 31. Sa priorité ne change jamais.
  - **Les threads utilisateur** : chaque thread de cette classe a une priorité comprise entre 1 et 15. Sa priorité varie durant sa vie mais reste toujours comprise entre 1 et 15. Il migre d'une file à une autre avec l'évolution de sa priorité.
  - **Le thread zéro (priorité 0)** : s'exécute, en arrière plan, lorsque toutes les files, de priorité supérieure, sont vides. Il se charge de la remise à zéro des pages libérées.
  - **Le thread idle** : s'exécute si toutes les files sont vides.

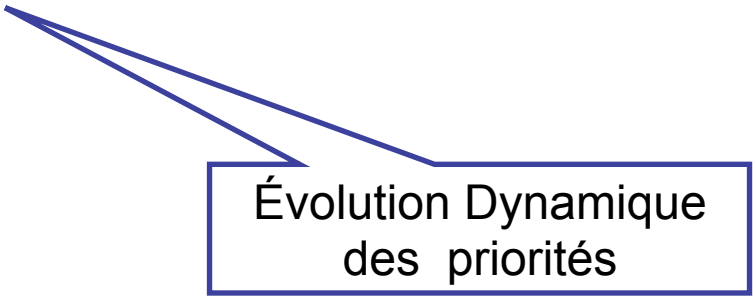


Cas de Windows (6)

- La priorité courante d'un thread utilisateur est augmenté lorsqu'il sort de l'état bloqué suite à :
  - une fin d'E/S (+1 pour un disque, +6 pour un clavier, +8 pour la carte son, etc)
  - Acquisition d'un sémaphore, mutex ou l'arrivée d'un événement (+2 ou +1) , etc.

- La priorité courante d'un thread utilisateur ne peut jamais dépasser 15.

- La priorité courante d'un thread utilisateur est diminué de 1 s'il consomme son quantum et sa priorité courante est supérieure à sa priorité de base.



Cas de Windows (7)

- **Problème d'inversion de priorité** → (producteur / consommateur) :

  Le producteur de priorité de base 12 est bloqué car le tampon est plein.

  Le consommateur de priorité de base 4 est en attente car il y a un thread de priorité de base 8 qui monopolise le processeur.

- Si un thread à l'état prêt reste trop longtemps en attente du processeur (ce temps dépasse un certain seuil), il se voit attribuer une priorité 15 pendant 2 quanta (accélérer le thread pour éviter les problèmes de famine et d'inversion de priorité).

- À la fin des deux quanta, le thread reprend sa priorité de base.

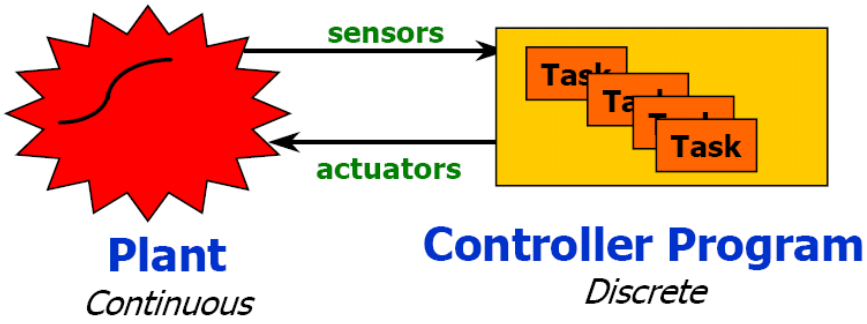
Cas de Windows (8)

- C'est le « balance set manager » qui se charge de la recherche de tels threads (toutes les secondes).
- Afin de minimiser le temps CPU consommé par le « balance set manager », un nombre limité (par exemple 16) de threads sont examinés à chaque passage. Il examinera les suivants à son prochain passage.
- Le « balance set manager » limite le nombre de threads à accélérer (par exemple à 10).

Ordonnancement temps réel

- **Qu'est ce qu'un système temps réel ?**
- **Qu'est ce qu'un système d'exploitation temps réel ? (RT-Linux)**
- **Ordonnancement temps réel de tâches périodiques**
  - **Tâches indépendantes**
  - **Tâches dépendantes**

Qu'est ce qu'un système temps réel ?



- Un *système temps réel (STR)* est un système qui doit répondre à un ensemble de stimuli provenant de son environnement, dans un intervalle de temps dicté par ce même environnement (i.e. contraintes temporelles).
- Les systèmes temps réel se caractérisent par :
  - de fortes interaction avec le procédé ou l'environnement (données avec une durée de validité limitée, **temps de réponse stricts**…) et
  - des contraintes de coût, d'espace, de consommation d'énergie, de matériel

  → Environnement d'exécution spécifique
- Un STR est généralement composé d'une partie continue et d'une partie discrète (un ensemble de tâches qui commandent la partie continue).

Qu'est ce qu'un système temps réel ? (2)

Il existe deux types de systèmes temps réel (STR) :

- STR stricts : Le système doit impérativement respecter les contraintes temporelles (temps de réponse, échéances). Sinon, il peut y avoir des conséquences majeures.
- STR souples : Les contraintes temporelles sont importantes, mais le système fonctionne correctement même si celles-ci ne sont pas respectées (échéances manquées, retard sur les réponses). Il en résulte seulement une dégradation de performance.

Exemples de STR stricts :

- Avionique et aérospatial;
- Gestion du trafic aérien;
- Contrôle d'une centrale nucléaire.

Exemples de STR souples :

- Jeux vidéos;
- Télé-conférences.

Qu'est ce qu'un système d'exploitation temps réel ?

- "Real time in operating systems:
  The ability of the operating system to provide a required level of service in a bounded response time."
  POSIX Standard 1003.1

- Le noyau d'un RTOS assure les fonctions de base d'un OS:
  - gestion des tâches (création, ordonnancement, communication, synchronisation);
  - gestion des interruptions;
  - gestion du temps (minuteries);
  - gestion de la mémoire….

  ➔ Mais il doit garantir en plus des temps de réponses bornés et acceptables aux demandes de services.

- L'objectif est d'exécuter des applications temps réel

  { tâches concurrentes, communicantes avec éventuellement des contraintes temporelles strictes et/ou souples }

- Noyau non préemptif ou préemptif -> préemptif

Qu'est ce qu'un système d'exploitation temps réel ? (2)

Comment garantir des temps de réponse bornés et satisfaire les contraintes temporelles ?

➔ Interdire tous les appels bloquants dans les ISR ou dans les tâches associées aux interruptions.

➔ Réduire et borner les durées de masquage des interruptions, le temps nécessaire pour déterminer la prochaine tâche à élire….

➔ Exécuter les tâches temps réel dans l'espace noyau.

➔ Ordonnancer les tâches en tenant compte de leurs contraintes temporelles (ordonnancement temps réel) :

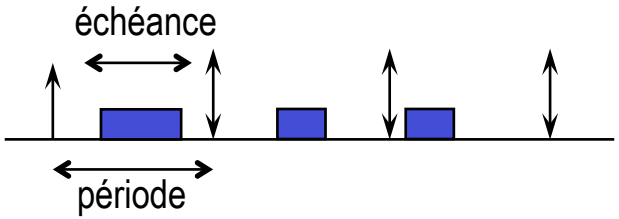
- ❑ Ordonnancement préemptif à priorité
- ❑ L'attribution des priorités doit tenir compte des contraintes temporelles.

→ Élire la tâche la plus prioritaire (« la plus urgente »)

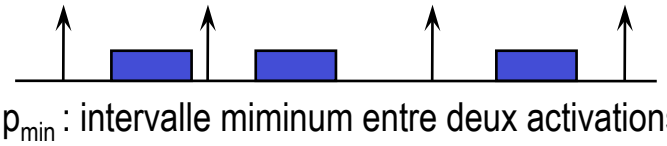
Qu'est ce qu'un système d'exploitation temps réel ? (3)

Tâches temps réel : Contraintes temporelles

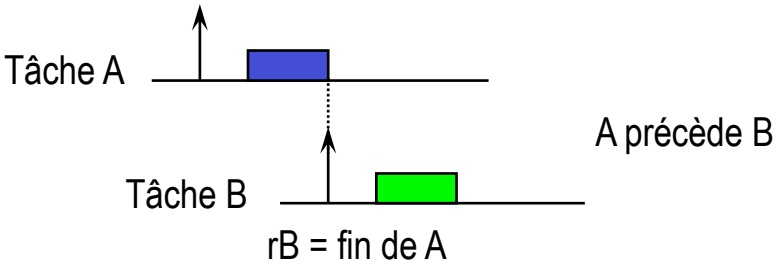
- Tâche périodique



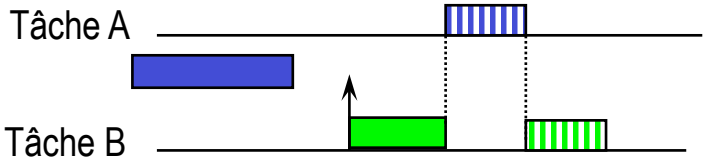
- Tâche apériodique (sporadique)



- Relations de précédence



- Exclusion mutuelle



Section critique

Qu'est ce qu'un système d'exploitation temps réel ? (4)

Tâches temps réel : Priorités

- La priorité d'une tâche est une métrique qui caractérise son importance.
- Principe général : plus une tâche est importante et critique du point de vue temporel, plus la priorité qu'on lui assigne est élevée.
- Les priorités des tâches peuvent être :
  - *statiques* (les priorités sont fixes, ne changent pendant l'exécution de l'application).
  - *dynamiques* (les priorités peuvent changer pendant l'exécution de l'application).
- La complexité des systèmes temps réel rend difficile le processus d'assignation des priorités aux tâches.
- Il est nécessaire de faire une analyse précise, afin de déterminer les tâches critiques et non critiques du système (surtout pour les systèmes temps réel avec contraintes strictes).

Ordonnancement temps réel de tâches périodiques

Ordonnancement temps réel de tâches périodiques

- Les tâches ont des contraintes temporelles, de précédence, de partage de ressources….

- L'ordonnancement doit se faire de manière à respecter toutes ces contraintes.
  - Ordonnancement préemptif / non préemptif => Ordonnancement préemptif
  - Ordonnancement à priorités statiques / dynamiques => Priorités statiques / dynamiques, mais assignées en tenant compte des contraintes temporelles, de précédence, de partage de ressources.

- La séquence d'exécution des tâches est établie dynamiquement par l'ordonnanceur en fonction des événements qui surviennent.

- L'ordonnanceur choisit la prochaine tâche à exécuter en fonction d'un critère de priorité qui dépend en général des contraintes temporelles.

Ordonnancement temps réel de tâches périodiques (2)

Contraintes sur les tâches périodiques

- Période d'activation : Pi
- Temps d'exécution de chaque tâche (au pire cas) à chaque activation : Ci
- Échéance (deadline) de chaque tâche : Di
- Tâches qui peuvent démarrer durant toute la période ou doivent démarrer durant les x premiers cycles de la période.
- Date d'activation initiale : Oi
- Tâche à échéance sur requête *:* Di = Pi
- Tâches synchrones Oi = Oj, pour tout (i, j)

Ordonnancement préemptif à priorités fixes de tâches périodiques indépendantes

Rate monotonic priority assignment (RMA) [Liu & Layland, 1973]

- La priorité d'une tâche est inversement proportionnelle à sa période. Plus la période est petite, plus la priorité est grande => Priorité statique

- La résolution de conflit est arbitraire.

- Un ensemble de n tâches périodiques indépendantes (pas de section critique) où Di=Pi, est ordonnançable si (**condition suffisante** de *Liu et Layland*) :

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

n	Utilization Bound
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.8%

n → ∞ => borne = 69.3 %

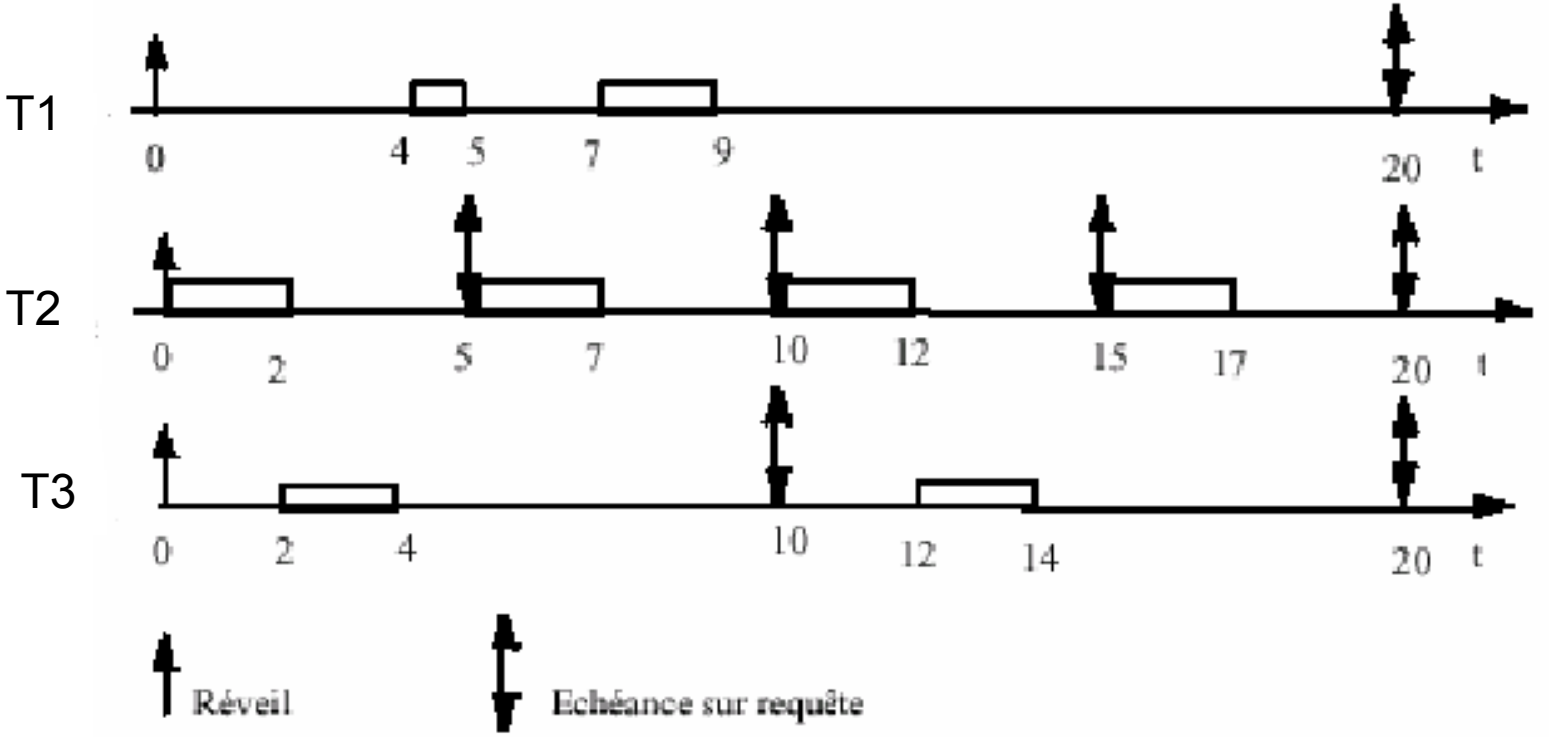
Ui = Ci / Pi Taux d'occupation processeur (ou charge) par tâche.

Rate monotonic priority assignment (RMA) (2)
Exemple 1

T1 : C1=3 P1=20 T2 : C2=2 P2 = 5
T3 : C3=2 P3 = 10 prio(T1) < prio(T3) < prio(T2)

3/20 + 2/5 + 2/10 = 75 % < 78 % => ordonnançable

Simulation
Intervalle d'étude :
[0, PPCM(P1,P2,P3)]



Deadline monotonic priority assignment (DMA) [Leung & Whitehead, 1985]

- La priorité d'une tâche est inversement proportionnelle à son *deadline (*plus le *deadline* est petit, plus la priorité est grande) → *Priorité statique*.

- Le conflit est résolu de façon arbitraire.

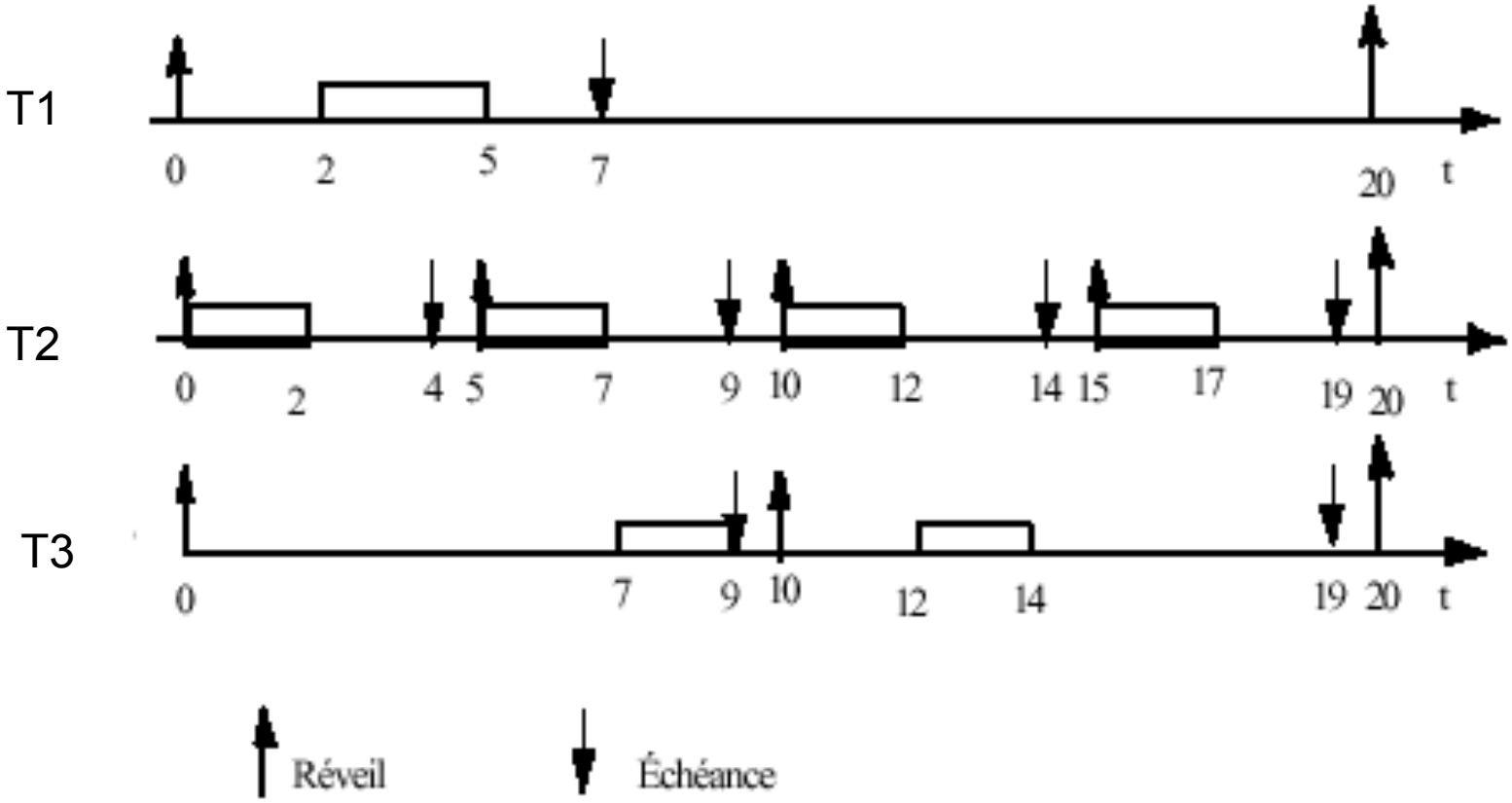
- Un ensemble de n tâches périodiques indépendantes est ordonnançable *si :*

$$\sum_{i=1}^{n} \frac{Ci}{Di} \leq n(2^{\frac{1}{n}} - 1)$$

Deadline monotonic priority assignment (DMA) (2)
Exemple 2

T1 : C1=3 D1=7 P1=20 T2 : C2=2 D2=4 P2 = 5
T3 : C3=2 D3=9 P3 = 10 prio(T3) < prio(T1) < prio(T2)

3/7 + 2/4 + 2/9 = 115 % non < 78 % on ne peut rien dire



Ordonnancement préemptif à priorités dynamiques de tâches périodiques indépendantes

Earliest Deadline First (EDF) [Liu & Layland, 1973]

- La tâche la plus prioritaire (parmi les tâches prêtes) est celle dont l'échéance absolue est la plus proche.

- Il est applicable aussi bien pour des tâches périodiques qu'apériodiques.

- Ordonnançable :

  Pour les tâches à échéance sur requête (Pi=Di pour toute tâche Ti), CNS : $\sum_{i=1}^{n} \frac{Ci}{Pi} \leq 1$

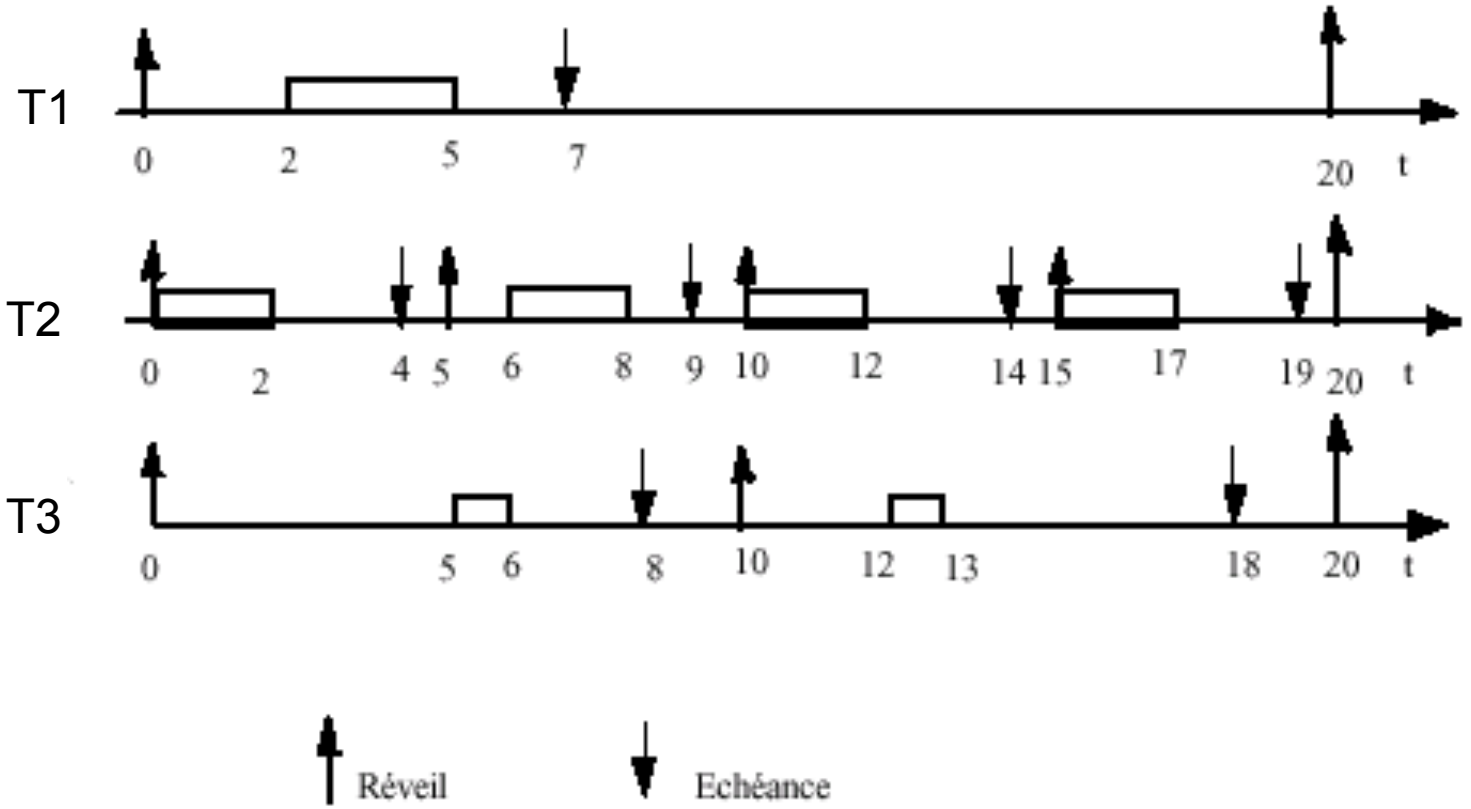
  Pour les autres cas:

  CN : $\sum_{i=1}^{n} \frac{Ci}{Pi} \leq 1$

  CS : $\sum_{i=1}^{n} \frac{Ci}{Di} \leq 1$

Earliest Deadline First (EDF) (2)
Exemple 3

T1 : C1=3 D1=7 P1=20 T2 : C2=2 D2=4 P2 = 5
T3 : C3=1 D3=8 P3 = 10
3/7 + 2/4 + 1/8 = 105 % non < 1 on ne peut rien dire



Ordonnancement en présence de ressources partagées

Problèmes de partages de ressources

- Les tâches partagent des ressources à accès exclusif.
- Lorsqu'une tâche Tx demande une ressource déjà allouée à une autre tâche Ty, Tx se met en attente de la ressource.
- Ordonnancement à priorité

⇒ Inversion de priorités possible : Une tâche de basse priorité empêche une tâche plus prioritaire d'accéder à sa section critique (bloque une tâche plus prioritaire).

⇒ Interblocage

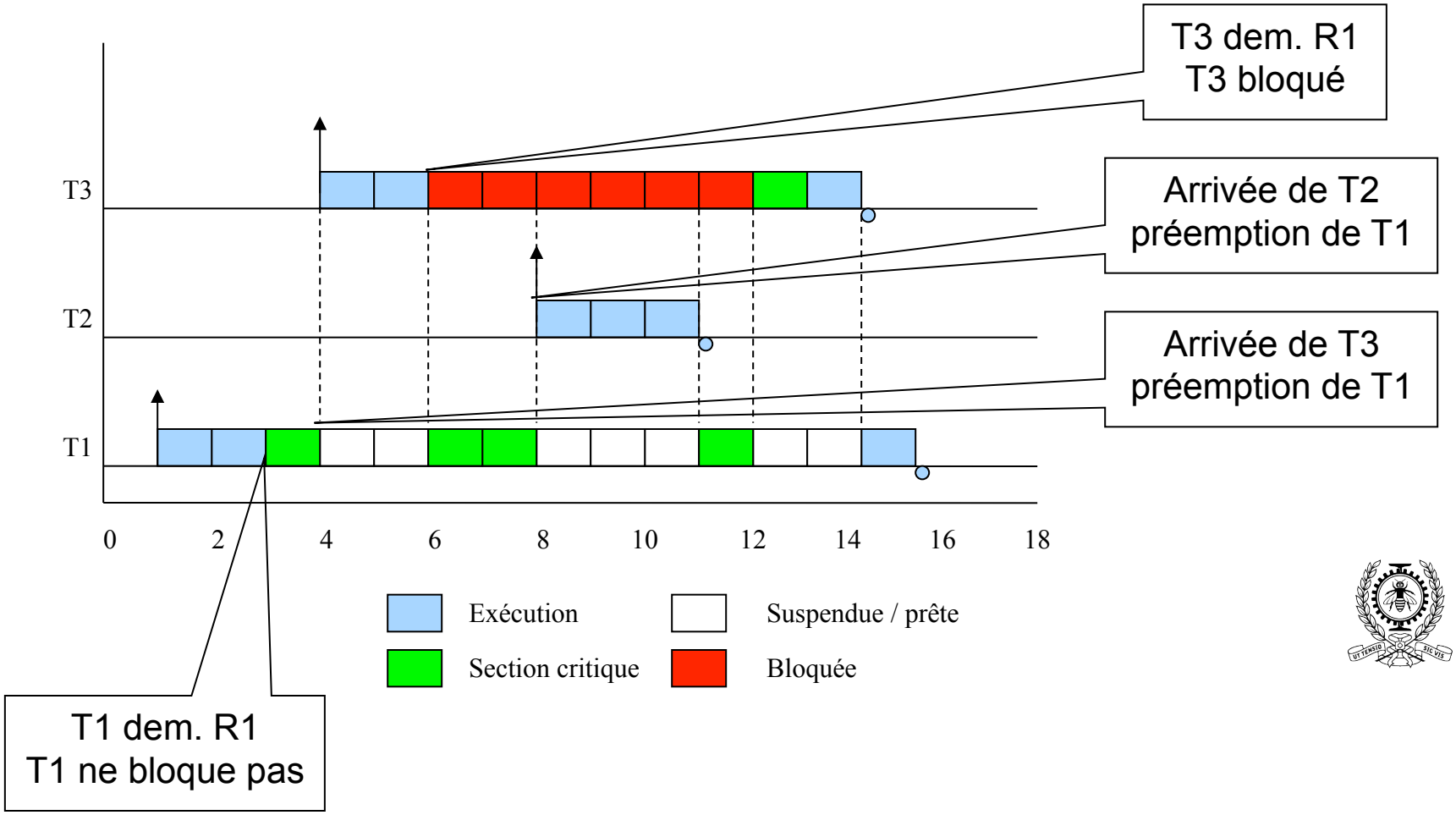
Problèmes de partages de ressources
Inversion de priorités

Exemple 4 :

Tâche	Date de départ	Priorité	Séquence d'exécution
T3	4	3	EER1E
T2	8	2	EEE
T1	1	1	EER1R1R1R1E

Problèmes de partages de ressources
Inversion de priorités (2)

Exemple 4 : Trois tâches : T1, T2, T3 → prio(T1)<prio(T2)< prio(T3)



Problèmes de partages de ressources
Inversion de priorités (3)

Traitement du problème d'inversion de priorités :

- Protocole d'héritage de priorités [Kaiser, 1982] [Sha, 1990]
- OCPP (Original Ceiling Priority Protocol) [Chen, 1990] [Sha, 1990]
- ICPP (Immediate Ceiling Priority protocol)

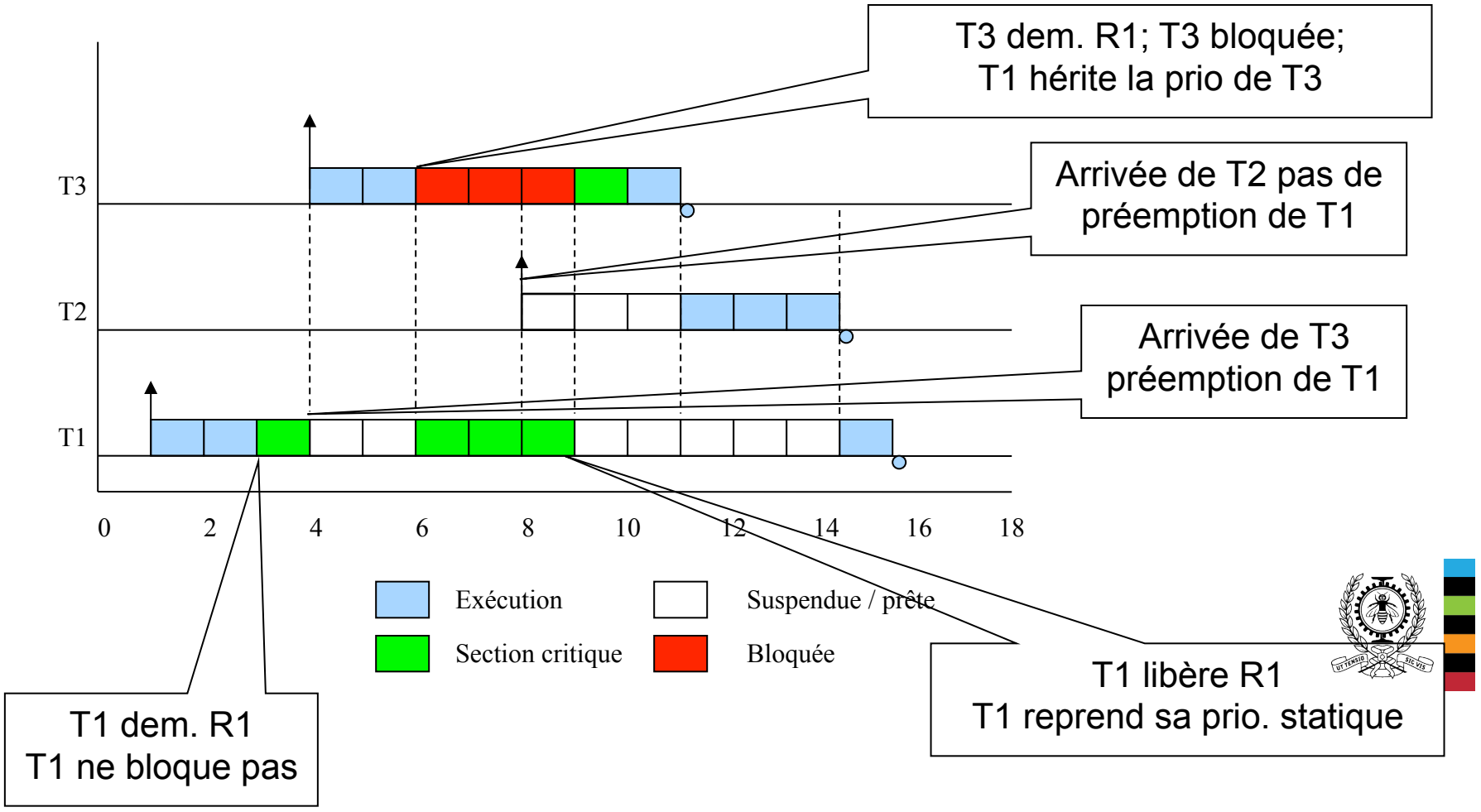
Protocole d'héritage de priorités (PIP = Priority Inheritance Protocol)

- Lorsqu'une tâche Tx bloque suite à une demande d'accès à une ressource R, la tâche bloquante Ty (celle qui détient R) hérite la priorité courante de Tx.

- La tâche Ty s'exécutera avec la priorité héritée jusqu'à la libération de la ressource.

- Elle retrouve ensuite sa priorité.

Protocole d'héritage de priorités (2) (PIP = Priority Inheritance Protocol)



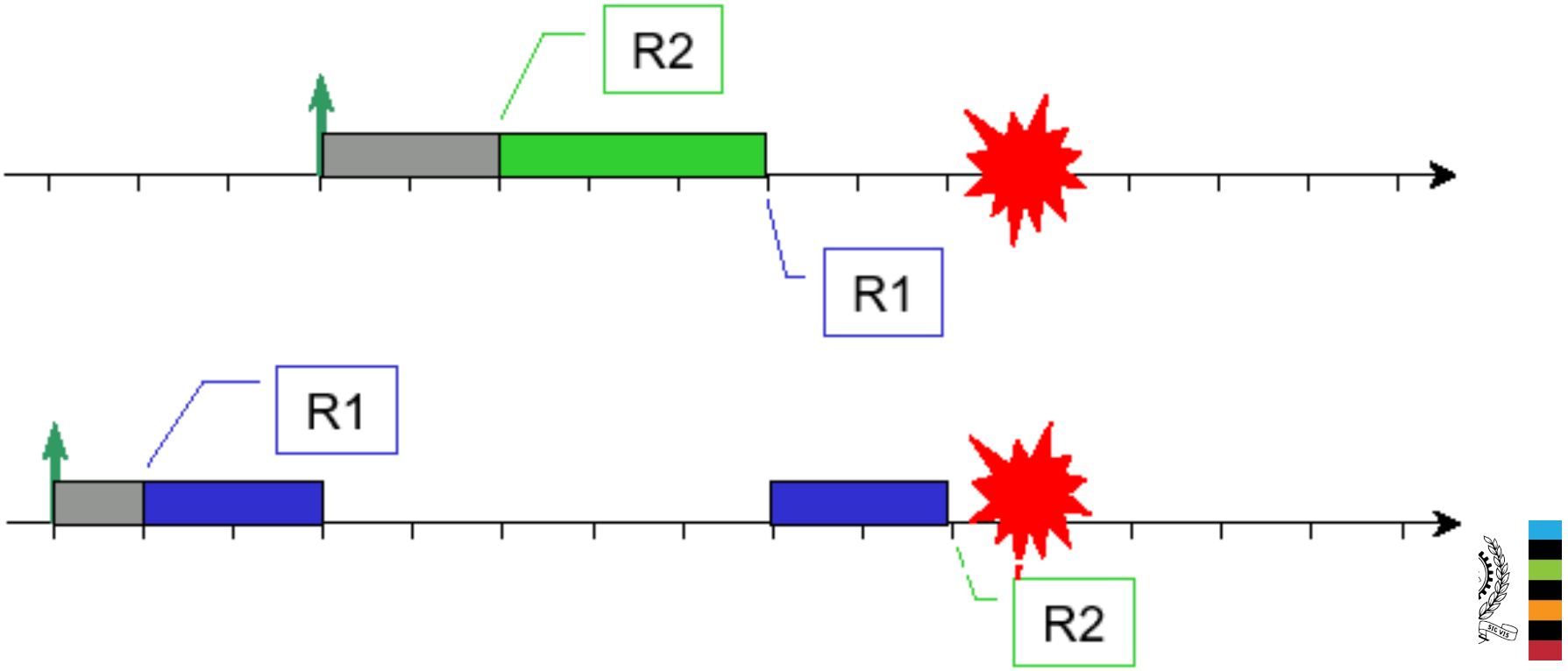
Protocole d'héritage de priorités (3) (PIP = Priority Inheritance Protocol)

- Ce protocole peut mener à un Interblocage : Deux tâches T1 et T2 utilisant deux ressources R1 et R2; prio(T1) < prio(T2); T1 est lancée avant T2.

Tâche T1	Tâche T2
Demander(R1);	Demander(R2);
/* utiliser R1	/* utiliser R2
Demander(R2) ;	Demander(R1) ;
/* utiliser R1 et R2	/* utiliser R1 et R2
Liberer(R2);	Liberer(R1);
Liberer(R1);	Liberer(R2) ;

- Temps de blocage est borné (sauf en cas d'interblocage) → PIP permet d'évaluer cette borne Bi (au pire cas) pour chaque tâche Ti.
- PIP n'élimine pas l'inversion de priorité mais il permet de la réduire.

Protocole d'héritage de priorités (4)
Problème d'interblocage



Exercice 1

Soit l'ensemble de tâches suivant à ordonnancer :

	Period T	Computation Time, C	Priority P	Utilization U
Task_1	80	40	1	0.50
Task_2	40	10	2	0.25
Task_3	20	5	3	0.25

100% non < 78% donc on ne peut rien dire

Vérifiez par représentation graphique (diagramme de Gantt) si les tâches peuvent être ordonnancées RMA.

Exercice 2 (RMA + PIP)

Tâche	Date d'arrivée	Temps d'exécution Ci	Période Pi
T3	3	3: ER1E	5
T2	4	2: EE	10
T1	1	5 : ER1R1R1E	20

- Les tâches T1, T2 et T3 sont-elles ordonnançables, selon RMA ?
- A-t-on un problème d'inversion de priorité ? Si oui, donnez le diagramme de Gantt correspondant au cas où ce problème est traité en utilisant le protocole PIP (Priority Inheritance Protocol). A-t-on des échéances manquées, dans ce cas ?
- Mission Mars Pathfinder lancée par la NASA (décembre 1996)
  http://www.fil.univ-lille1.fr/~nebut/portail/svl/fichiers/tp/tpPathfinder/tpPathfinder.pdf

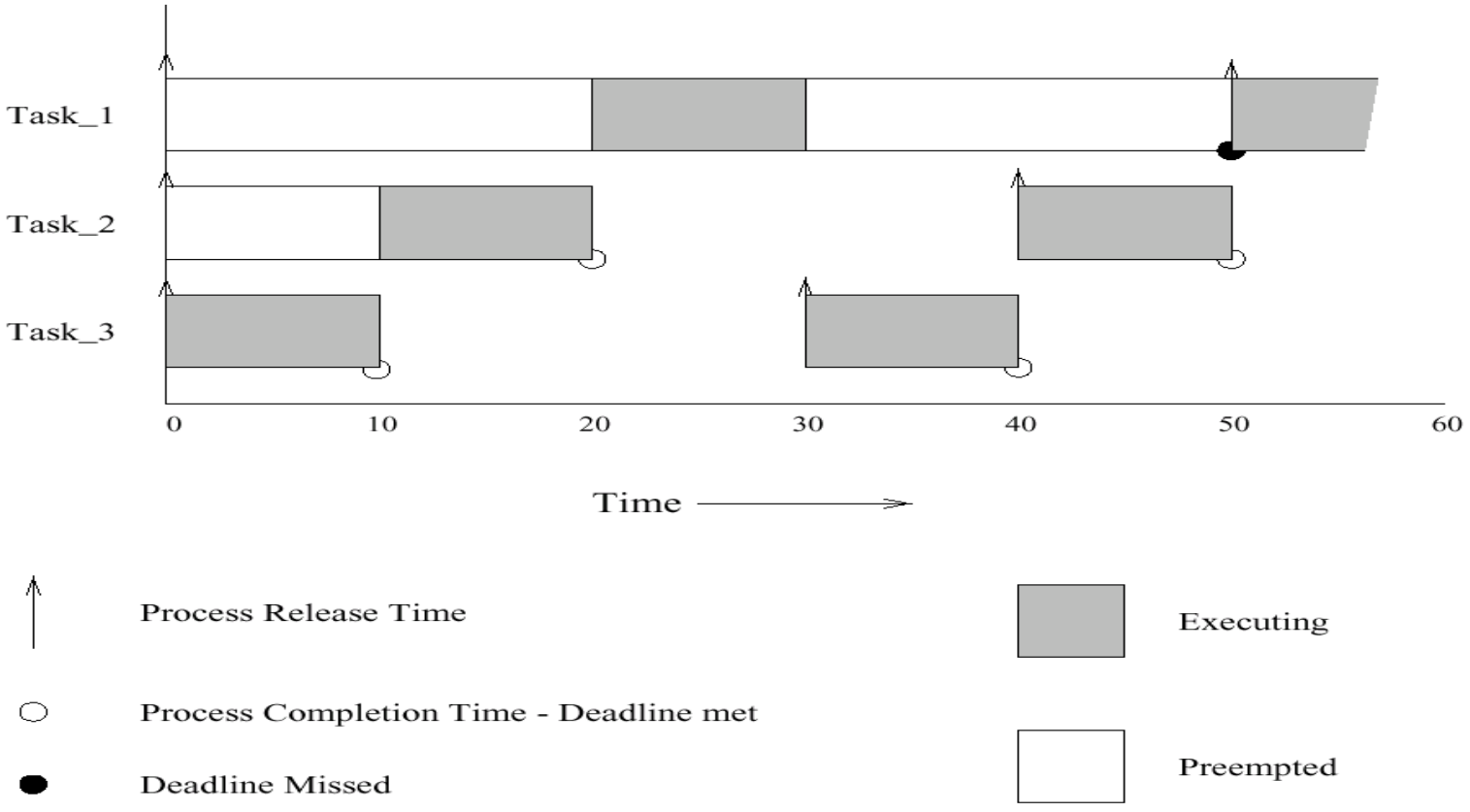
  Tâche météo (la moins prioritaire)
  Tâche gestion du bus la plus prioritaire
  Tâche de communication (priorité intermédiaire)
  (http://degeeter.pagesperso-orange.fr/inv_fr.htm)

Autres exemples
Rate monotonic priority assignment (RMA)

	Period P	Computation Time, C	Priority	Utilization U
Task_1	50	12	1	0.24
Task_2	40	10	2	0.25
Task_3	30	10	3	0.33

$82\% < 78\%$ on ne peut donc rien dire

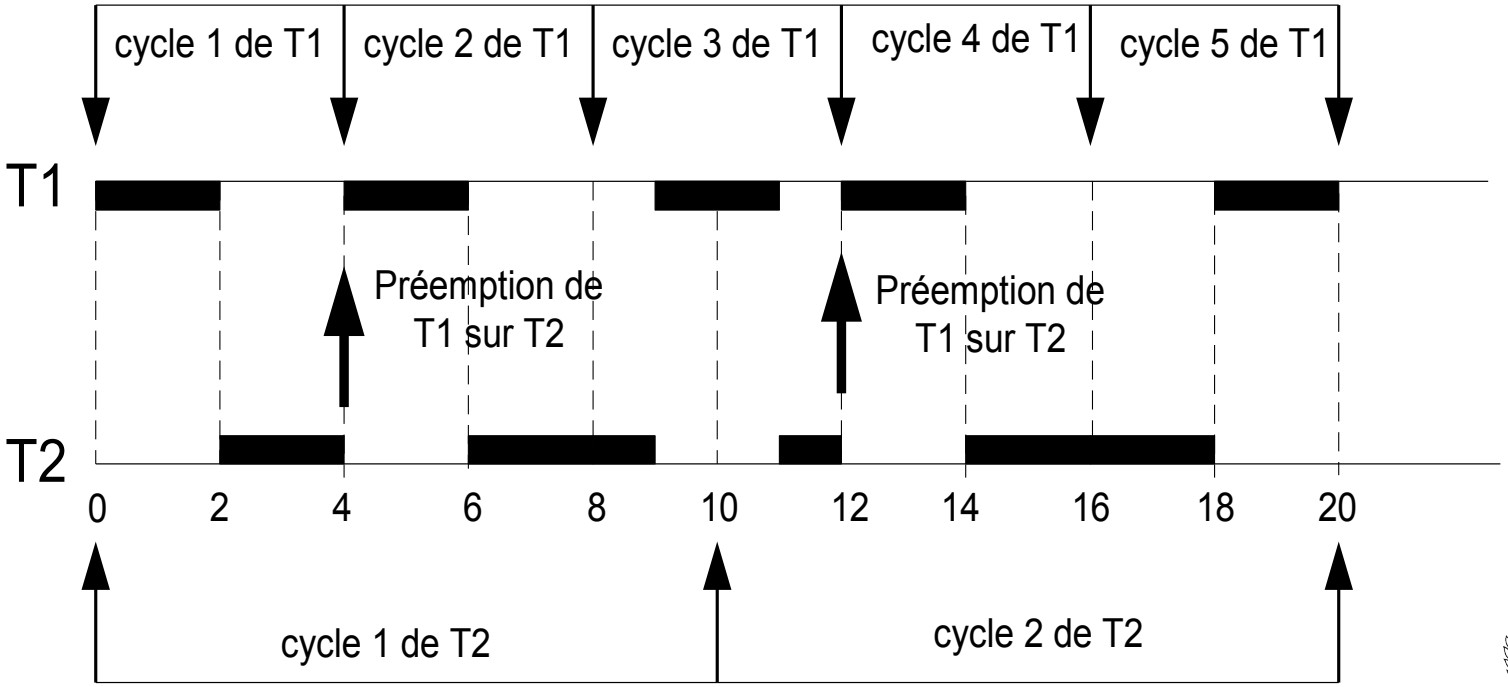
Autres exemples
Rate monotonic priority assignment (RMA)



Par contre, graphiquement on voit qu'on ne peut pas ordonnancer

Autres exemples
Earliest Deadline First (EDF)

L'ordonnancement EDF de ces tâches:



Autres exemples
Earliest Deadline First (EDF)

Soient les tâches suivantes :

Task	Period, P	Deadline, *D*	Computation Time, *C*
T1	4	4	2
T2	10	10	5

2/4 + 5/10 = 1, donc les tâches peuvent être ordonnancées EDF (Liu and Layland).

Autres exemples
Earliest Deadline First (EDF)

Remarques :

- T1 démarre en premier, car elle possède le plus petit *deadline* (4 vs 10);
- À l'instant 1, le *deadline* de T1 (3) est toujours plus petit que celui de T2 (9), donc T1 poursuit son exécution;
- À l'instant 2, T1 a terminé son cycle 1 et son *deadline* devient 0. Par contre, T2 qui n'a pas encore démarré, a maintenant un *deadline* de 8. T2 a donc le plus petit *deadline* non nul → elle démarre;
- À l'instant 3, le *deadline* de T1 est encore 0, tandis que celui de T2 est 7, donc T2 poursuit son exécution;
- À l'instant 4, T2 a maintenant un *deadline* de 6 et T1 retrouve son deadline de 4 (début de son 2^{e} cycle). Donc T1 a une préemption sur T2;

Autres exemples
Earliest Deadline First (EDF)

- À l'instant 5, le deadline de T1 (3) est toujours plus petit que celui de T2 (5), donc T1 poursuit son exécution;

- À l'instant 6, T1 a terminé son cycle 2 et son *deadline* devient 0. Par contre, T2 a maintenant un deadline de 4. T2 a donc le plus petit *deadline* non nul → elle redémarre son exécution;

- À l'instant 7, T2 a maintenant un *deadline* de 3 et T1 a encore un deadline de 0, donc T2 poursuit son exécution;

- À l'instant 8, T2 a maintenant un *deadline* de 2 et T1 retrouve son deadline de 4 (début de son 3e cycle). Cependant puisque 2 < 4, T2 poursuit son exécution.

- À l'instant 9, T2 a terminé son cycle 1 et donc son *deadline* devient 0. Par conséquent, T1 qui a un deadline de 3 démarre son cycle 3;
- etc.

Autres exemples
Earliest Deadline First (EDF)

- On remarque aussi que ce n'est qu'à la période 20 que les deux tâches terminent un cycle en même temps (ce qui n'est pas le cas à la période 10).

- On peut donc considérer que le même ordonnancement se répétera entre les périodes 20 et 40, 40 et 60, 60 et 80, etc.

- C'est en fait le plus petit multiple commun entre les deux périodes qui détermine le point de rencontre.

- Finalement on remarque que le CPU est utilisé à 100% du temps (dans la plupart des cas, le CPU est utilisé moins que 100%).

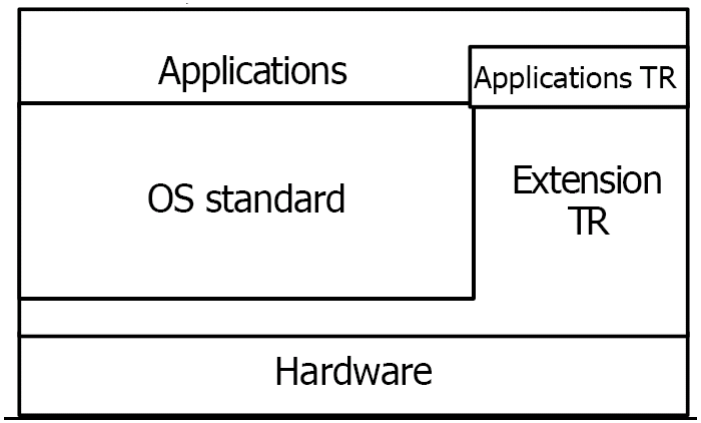
RT-Linux

http://www.mnis.fr/ocera_support/rtos/c1450.html

- **Linux** n'est pas un système d'exploitation Temps Réel (dur) car le noyau Linux possède de longues sections de code où tous les événements extérieurs sont masqués (non interruptibles).

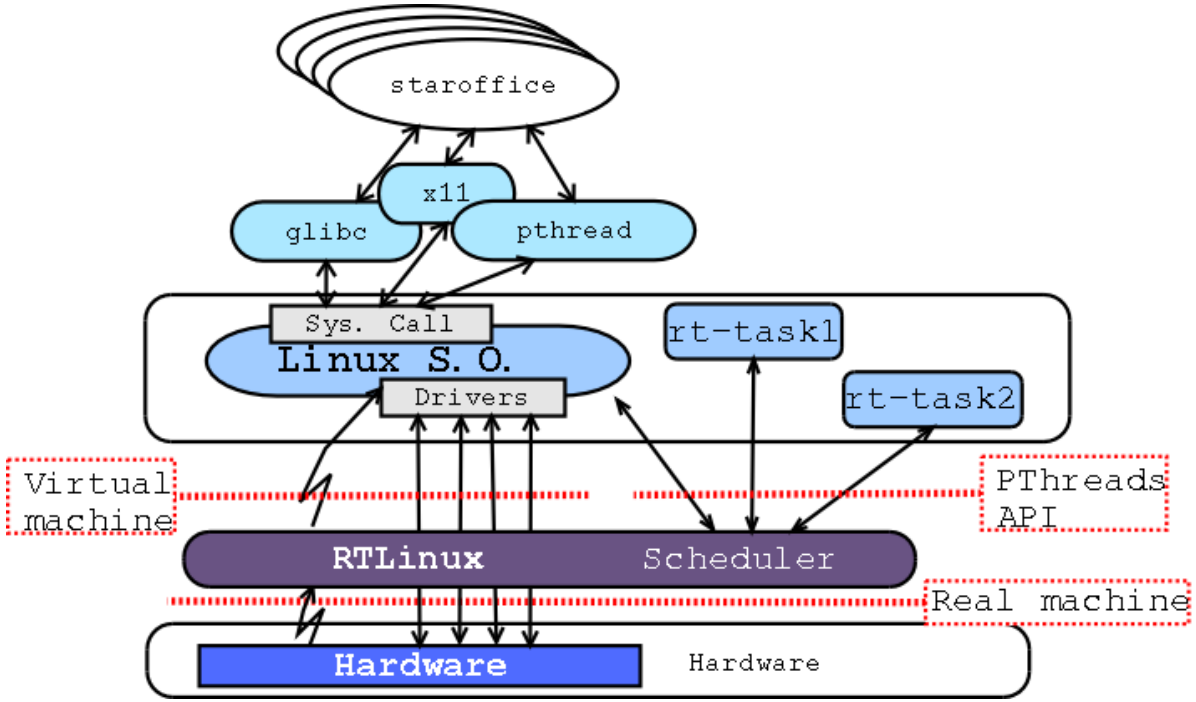
- Systèmes d'exploitation temps réel dédiés :
  - VxWorks,
  - MicroC,
  - QNX,
  - LynuxWorks …

- Les extensions existantes de cohabitation du noyau Linux avec un micronoyau :
  - **RT-Linux**,
  - RTAI….



RT-Linux (2)

http://www.mnis.fr/ocera_support/rtos/c1450.html



- RT-Linux est une extension du système Linux classique vers le temps réel.
- Il est constitué d'un noyau temps réel RT-Linux qui partage le processeur avec le noyau de base Linux et exécute des tâches temps réel.
- Les tâches TR sont chargées comme des modules Linux. Linux apparaît comme une tâche de fond.

RT-Linux (3)

- Application = partie TR (tâches temps réel) + partie non TR (processus Linux).

- Les processus Linux et les tâches TR peuvent communiquer par des FIFOS (pipes) ou par mémoire partagée.

- Les tâches RT-Linux s'exécutent dans l'espace d'adressage du noyau au même titre que l'ordonnanceur temps réel.

  → élimine la charge occasionnée par tout changement de mode de protection (mode utilisateur, mode noyau);

  → réduit les latences et les temps de réponse;

  → comporte un risque : un « Bug » dans une tâche temps réel peut mettre en danger tout le système.

RT-Linux (4)
Les tâches

- Les tâches sont, en général, créées lors du chargement du module (fonction init_module()) et supprimer uniquement au déchargement (fonction cleanup_module).

- La structure « rt_task_struct » (ou RT_TASK par « typedef struct rt_task_struct RT_TASK;») des tâches temps réel dans RT-Linux contient notamment les champs suivants :
  - État (Inexistant, Dormant, Prêt, Élu, Attente de la prochaine activation, etc);
  - Priorité;
  - Période;
  - Date de la prochaine activation.

RT-Linux (5)
Les tâches

- Deux types de tâches sont gérées dans RT-Linux :

- Tâches périodiques :
  - exécutent périodiquement un traitement.
  - sont activées périodiquement
- Le traitement périodique réalisé par une tâche doit se faire à l'intérieure de la période (avant la prochaine activation).

- Tâches apériodiques :
  - exécutent à des intervalles de temps irréguliers un traitement.
  - sont généralement activées par d'autres tâches (relation de précédence, arrivée d'un événement, …).

RT-Linux (6)
Les tâches

- Création d'une tâche :

  int rt_task_init (RT_TASK *task, void (fn)(int data), int data,

  int stack_size, int priority);

- Activation d'une tâche :

  int rt_task_wakeup (RT_TASK *task);

- Rendre une tâche inactive :

  int rt_task_suspend (RT_TASK *task);

RT-Linux (7)
Les tâches

- Transformation d'une tâche créée en une tâche périodique :

  int rt_task_make_periodic(RT_TASK **task*, RTIME *start_time*, RTIME *period*

Cette fonction doit être appelée après sa création (pas après son activation).

- Suspension d'une tâche périodique jusqu'à sa prochaine date de réveil :

  int rt_task_wait (void);

- Suppression d'une tâche RT-Linux (passe à l'état dormant) :

  int rt_task_delete (RT_TASK *task);

RT-Linux (8)
Tâches périodiques

- Lorsqu'une tâche est créée rt_task_init(), elle est à l'état **Dormant**.
- L'appel à la fonction rt_task_make_periodic() rend la tâche périodique et la met en attente jusqu'à la date du prochain réveil (état **Delayed**)→ À son réveil, elle passe à l'état prêt (état **Ready**).
- Si elle est élue, elle passe à l'état **Active**. Elle revient à l'état prêt, si elle est préemptée, et ainsi de suite.
- La fonction rt_task_wait() permet à la tâche de se mettre en attente de la prochaine activation périodique → elle passe à l'état **Delayed**.
- La fonction rt_task_suspend() permet de suspendre la tâche → elle passe à l'état **Dormant** (elle est inactive).
- La fonction rt_task_delete() permet la suppression de la tâche → elle passe à l'état Inexistant (**Zombie**).

RT-Linux (9)
Tâches apériodiques

- Lorsqu'une tâche est créée rt_task_init(), elle est à l'état **Dormant**.
- La fonction rt_task_wakeup permet de réveiller la tâche → elle passe à l'état prêt (**Ready**).
- Si elle est élue, elle passe à l'état **Active**. Elle revient à l'état prêt, si elle est préemptée, et ainsi de suite.
- La fonction rt_task_wait() permet à la tâche de se mettre en attente de la prochaine activation → elle passe à l'état **Delayed**.
- La fonction rt_task_suspend() permet de suspendre la tâche → elle passe à l'état **Dormant**.
- La fonction rt_task_delete() permet la suppression de la tâche → elle passe à l'état Inexistant (**Zombie**).

RT-Linux (10)
Ordonnancement de tâches

- Ordonnancement (par défaut) préemptif à priorités statiques

  → Il sélectionne la tâche prête la plus prioritaire.

- Ordonnancement Earliest Deadline First (EDF)

- Ordonnancement Rate Monotonic priority assignment (RMA)

Lectures suggérées

- Notes de cours: Chapitre 8

(http://www.groupes.polymtl.ca/inf2610/documentation/notes/chap8.pdf)