

# INF2610

## Noyau d'un système d'exploitation

---

Chapitre 8 – Ordonnancement des processus

# Chapitre 9 - Ordonnancement de processus

- Introduction
- Politiques d'ordonnancement
  - Ordonnancements non préemptifs
  - Ordonnancements préemptifs
- Cas de Linux (et Windows en annexe)
- Qu'est qu'un système d'exploitation temps réel ?
- Ordonnancement de tâches temps réel
  - Tâches périodiques indépendantes
  - Tâches périodiques dépendantes

# Introduction



- Dans un système multiprogrammé, plusieurs processus ou threads peuvent être prêts (en attente de temps CPU) en même temps.
- L'ordonnanceur (ordonnanceur de bas niveau) est la partie du système d'exploitation (SE) qui se charge de gérer l'allocation de(s) processeur(s) aux processus/threads prêts.
- La majorité des systèmes d'exploitation actuels (Linux, Unix, Windows, MacOS, etc.) ordonnance les threads (dans ce qui suit, les termes threads et processus sont interchangeables).
- Comment **gérer efficacement** l'allocation de(s) processeur(s) ?

# Introduction - Objectifs

## Pour le Système :

- **Maximiser le taux d'utilisation** des processeurs et des autres composants du système.
- **Éviter la famine** (longue attente de temps CPU).
- **Maximiser la capacité de traitement** (nombre de processus exécutés par unité de temps).
- **Minimiser le nombre et la durée des changements de contexte.** ← systèmes en temps partagé.
- **Répartir équitablement la charge entre les différents processeurs.** ← systèmes multi-processeur.
- **Respecter les échéances des tâches temps réel** (terminer les traitements avant leurs échéances (deadlines)) ← systèmes temps réel.

## Pour les utilisateurs :

- **Minimiser les temps de réponse et de séjour des processus.**

# Introduction – Critères d'évaluation

- **Taux d'utilisation d'un processeur** (taux d'occupation).
- **Capacité de traitement d'un processeur** : nombre de processus traités par unité de temps.
- **Temps de séjour d'un processus (temps de rotation ou de virement)** : temps entre son admission et sa terminaison.
- **Temps de réponse d'un processus** : temps entre son admission et le début de son exécution.
- **Temps d'attente d'un processus** : somme des temps qu'il a passé à l'état prêt.
- ➔ **Temps moyen de séjour, temps moyen de réponse et temps moyen d'attente.**

# Introduction - Questions clés

- Quels sont les critères de sélection du processus à exécuter ?
  - Premier arrivé, premier servi.
  - Plus prioritaire d'abord (priorités statiques ou dynamiques), etc.
- Quel est le temps d'allocation du processeur au processus sélectionné ?
  - Allocation jusqu'à la terminaison, le blocage, libération volontaire ou l'arrivée d'un processus/thread plus prioritaire.
  - Temps d'allocation limité fixe ou variable.
- L'exécution d'un processus est une séquence alternée de calculs CPU (rafales CPU) et d'attentes d'événements (E/S, sémaphores etc.). Un processus est dit :
  - **CPU-Bound**, s'il réalise beaucoup de calcul et peu d'E/S et
  - **IO-Bound**, s'il réalise peu de calcul et beaucoup d'E/S.

# Introduction - Questions clés

- Les **IO-Bound** processus passent beaucoup de temps à attendre des E/S. Ils ont en général besoin de temps CPU pour réaliser un traitement court qui inclut le lancement d'une opération d'E/S. Il est donc avantageux de les favoriser.
- Les **CPU-Bound** processus consomment beaucoup de temps CPU. Ils peuvent monopoliser le(s) processeur(s). Comment éviter ce problème sans dégrader les performances ?
- Comment déterminer la classe d'un processus ?
- **Comment favoriser les processus IO-Bound ?**

# Introduction - Questions clés

- Comment gérer la répartition des processus prêts sur les différents processeurs (si le SE gère plusieurs processeurs) ?
  - Est-il préférable d'utiliser une seule file d'attente ou une file d'attente par processeur ?
  - Comment gérer le problème de copies multiples dans les caches des processeurs, si des processus (qui partagent des données) sont affectés à des processeurs différents ?
  - Comment répartir équitablement la charge entre les différents processeurs dans le cas où chaque processeur a sa propre file d'attente ?

INF8601 - Systèmes informatiques parallèles

Recherches en cours d'algorithmes  
de répartition plus efficaces.

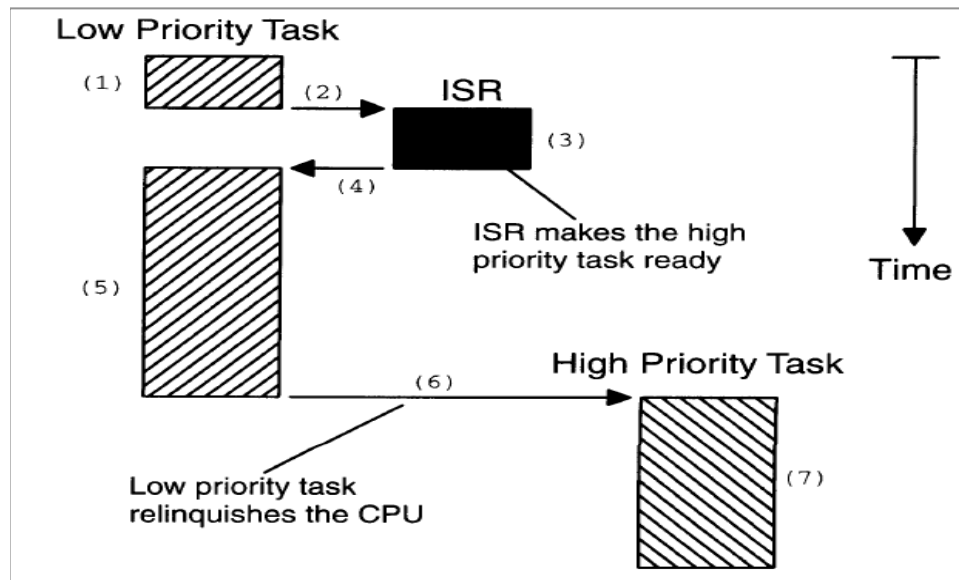


# Politiques d'ordonnancement

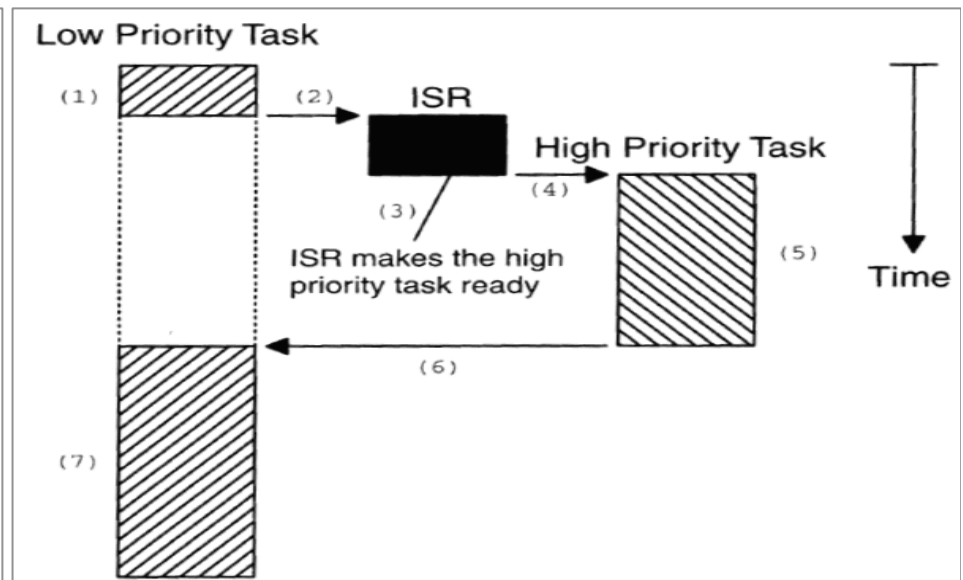
- On distingue deux classes de politiques d'ordonnancement :  
Ordonnancements non préemptifs et Ordonnancements préemptifs.  
La différence entre les deux classes est dans la réponse à la question suivante :  
Quand est-ce l'ordonnanceur est appelé pour sélectionner un processus prêt ?
- D'une manière générale, un ordonnanceur non-préemptif est appelé lorsque :
  - le processus en cours d'exécution se bloque (en attente d'E/S, d'un sémaphore, de la fin d'un fils, etc.),
  - le processus en cours a terminé son exécution ou cède le processeur.
- Un ordonnanceur préemptif est appelé lorsque :
  - le processus en cours d'exécution se bloque (en attente d'E/S, d'un sémaphore, de la fin d'un fils, etc.),
  - le processus en cours a terminé son exécution,
  - l'état du processus en cours bascule vers l'état prêt, ou
  - l'état d'un autre processus bascule vers l'état prêt.

# Politiques d'ordonnancement

## Ordonnancement non préemptif



## Ordonnancement préemptif



# Ordonnancements non préemptifs

Quels sont les critères de sélection du processus à exécuter ?

- Le système d'exploitation choisit le prochain processus à exécuter selon un critère :
  - Premier arrivé, Premier servi (FCFS, First-Come First-Served appelé aussi FIFO),
  - Plus prioritaire d'abord, etc.
- 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.

## Ordonnancements non préemptifs (2) : FIFO

Exemple 1 :

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

AAABBBBBBBCCCCDDE

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.

## Ordonnancements non préemptifs (3): À priorités

### Exemple 2 :

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

- $\text{Priorité} = (\text{temps d'attente} + \text{temps d'exécution}) / \text{temps d'exécution}$

AAABBBBBBEDDCCCC

Le temps moyen de séjour : 6,4

Le temps moyen d'attente : 3,2

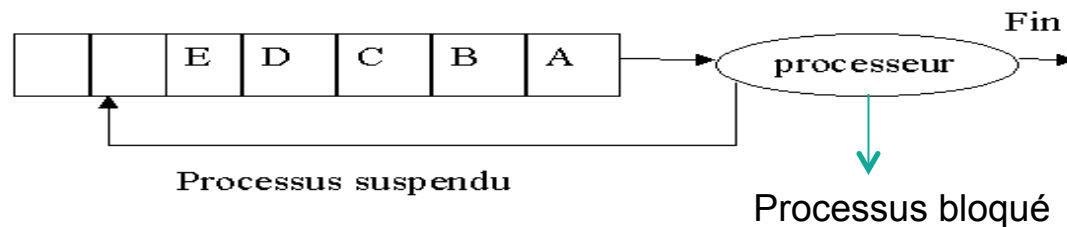
Un processus peut monopoliser l'utilisation d'un processeur si l'ordonnancement est non préemptif.

# Ordonnancements préemptifs (avec réquisition)

- Le système peut suspendre l'exécution d'un processus avant qu'il se termine ou se bloque, si, par exemple, un processus plus prioritaire devient prêt.
  - Il peut aussi limiter le temps d'allocation du processeur aux processus, pour éviter qu'un processus monopolise le processeur.
- les ordinateurs ont une horloge électronique qui génère périodiquement une interruption (période appelée tic (tick) d'horloge). La fréquence en Hz donne le nombre de tics par seconde.
- 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.

## Ordonnancements préemptifs (2) : circulaire (tourniquet/round robin)

- C'est un 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).
- Il sélectionne le processus prêt en tête de file pour lui allouer le processeur pendant au plus un **quantum de temps**.
- Si le processus est toujours en exécution à la fin du quantum, il est suspendu et inséré à la fin de la file.



## Ordonnancements préemptifs (3) : circulaire (tourniquet/round robin)



- L'ordonnanceur est appelé dans ce cas si le processus en cours se bloque, se termine, cède le processeur ou a consommé tout son quantum.
- Le processeur passe donc d'un processus à un autre en exécutant chaque processus pendant au plus un quantum.
- La commutation entre processus doit être rapide (temps de commutation de contexte doit être nettement inférieur au quantum).
- Un processeur, à un instant donné, n'exécute réellement qu'un seul processus. Pendant une seconde, le processeur peut exécuter plusieurs processus et donner ainsi l'impression de parallélisme (pseudo-parallélisme).



## Ordonnancements préemptifs (4) : circulaire (tourniquet/round robin)



### Choix de la valeur du quantum :

- L'algorithme est 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 les temps d'attente moyens (comme FIFO).
  - 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.

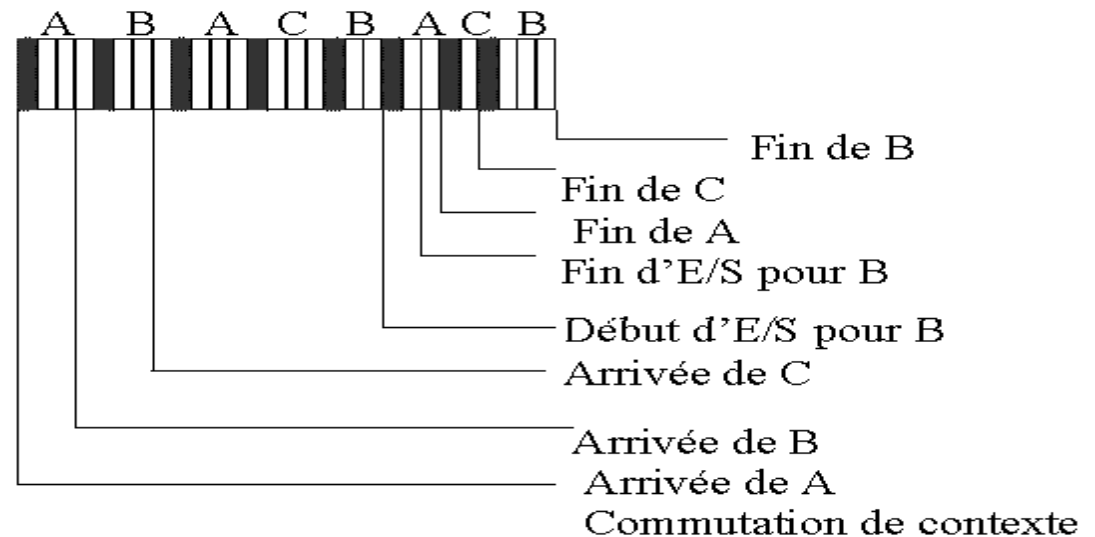
## Ordonnancements préemptifs (5) : circulaire (tourniquet/round robin)

### Exemple 3 :

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

## Ordonnancements préemptifs (6) : circulaire (tourniquet/round robin)

### Exemple 4 :

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

## Ordonnancements préemptifs (7) : circulaire (tourniquet/round robin)

### Exemple 5 :

Quantum = 7 unités

Commutation de contexte = 1 unité

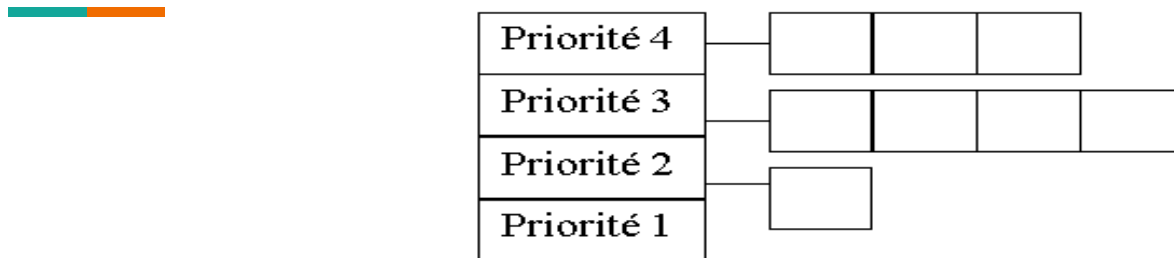
Processus	Exécution	Arrivée
A	8	0
B	5(2)3	3
C	4	4

- 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

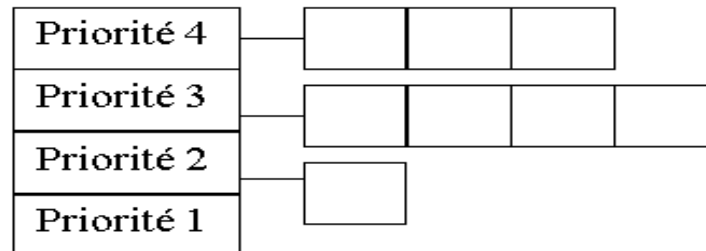
- **Comment favoriser les processus plus importants que d'autres ?**
- **Comment favoriser les I/O- bound processus ?**

## Ordonnancements préemptifs (8) : à files multiples



- Une priorité est attribuée à chaque processus.
- Il y a autant de files d'attente qu'il y a de niveaux de priorités. Les processus qui partagent la même priorité sont dans une même file d'attente.
- Les processus de chaque file sont ordonnancés selon l'algorithme RR (circulaire avec un quantum variable ou fixe) ou FIFO.  
→ On peut combiner plusieurs politiques d'ordonnancement.
- Le processus prêt le plus prioritaire en tête de file est sélectionné.

## Ordonnancements préemptifs (9) : à files multiples



- Le processus sélectionné est exécuté jusqu'à ce qu'il se bloque, se termine, cède le processeur, consomme son quantum (s'il s'agit d'un processus RR) ou encore un processus plus prioritaire devient prêt.
- Les nouveaux processus ou les processus qui basculent vers l'état prêt sont insérés à la fin de leurs files d'attente.

# Ordonnancements préemptifs (10) : 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 prêts plus prioritaires.
  - Si l'attente de la ressource est active dans le processus plus prioritaire → une attente active infinie.

## Ordonnancements préemptifs (11) : à files multiples

- Pour empêcher les processus de priorités élevées de s'exécuter indéfiniment, l'ordonnanceur devrait diminuer la priorité de tout processus au fur et à mesure qu'il consomme du temps CPU et l'augmenter au fur et à mesure qu'il attend.
- Pour favoriser les processus **IO-Bound**, il devrait augmenter leurs priorités lorsqu'ils sont en attente d'E/S, afin de leur permettre de lancer leurs requêtes suivantes d'E/S.  
→ Chaque processus aurait donc **une priorité de base** et **une priorité dynamique**.
- Par exemple, la priorité d'un processus est **diminuée de 1 s'il a consommé tout son quantum ou d'une quantité qui dépend du temps CPU consommé**. Elle est **augmentée de 1 s'il se met en attente d'un sémaphore, de 2 s'il se met en attente d'E/S sur le disque, de 5 s'il se met en attente du clavier**, etc. Elle est aussi **augmentée s'il reste trop longtemps à l'état prêt**.
- Pour éviter qu'il y ait beaucoup de commutations pour un processus **CPU-Bound**, il est préférable de lui allouer un plus grand quantum tout en diminuant sa priorité. Windows et Solaris associent des quanta plus grands aux processus moins prioritaires.



## Exercice 1 : Ordonnancement circulaire



Supposez trois processus P1, P2 et P3 avec les caractéristiques suivantes :

Processus	Date d'arrivée ( $O_i$ )	Durée d'exécution ( $C_i$ )
P1	0	10
P2	1	5
P3	2	7

Donnez le digramme de Gant de l'exécution de ces processus dans le cas d'un ordonnancement circulaire de quantum de 4. Donnez les temps de séjour et d'attente moyens (TMS et TMA).

Les temps de commutation de contexte sont supposés nuls.

## Exercice 1 (suite) : Ordonnancement circulaire

Supposez que les processus P1 et P2 ont chacun une section critique (le processus P3 n'a pas de section critique). Le triplet  $(x_i, y_i, z_i)$  du temps d'exécution du processus  $P_i$  (pour  $i=1,2$ ) signifie que :  $P_i$  réalise un calcul de  $x_i$  unités de temps CPU avant la demande d'entrer en section critique. Pour exécuter et quitter sa section critique,  $y_i$  unités de temps CPU sont nécessaires. Après sa section critique,  $P_i$  se termine au bout de  $z_i$  unités de temps CPU. L'ordonnancement de ces processus est circulaire avec un quantum de 4 et les temps de commutation de contexte sont supposés nuls.

Processus	Date d'arrivée ( $O_i$ )	Durée d'exécution ( $C_i$ )
P1	0	10 (3, 6, 1)
P2	1	5 (2, 2, 1)
P3	2	7

Donnez le digramme de Gant de l'exécution des processus P1, P2 et P3, pour chacun des cas suivants :

- a) une attente passive d'accès aux sections critiques.
- b) une attente active d'accès aux sections critiques.

Indiquez sur les diagrammes les dates d'entrées et de sorties des sections critiques ainsi que les attentes (passives ou actives). Donnez le temps de séjour moyen pour chaque cas.

## Exercice 2 : Ordonnancement à files multiples

Considérez les 3 processus P1, P2 et P3 suivants :

<i>Processus</i>	<i>Priorité de base</i>	<i>Temps d'exécution</i>
<i>P1</i>	<i>13</i>	<i>10 unités de CPU</i>
<i>P2</i>	<i>15</i>	<i>6 unités de CPU, 6 unités d'E/S, 2 unités de CPU</i>
<i>P3</i>	<i>6</i>	<i>5 unités de CPU</i>

Supposez ce qui suit :

- L'ordonnancement des processus est préemptif et à files multiples avec 20 niveaux de priorité de 0 à 19. La priorité 0 est la priorité la plus basse.
- Les processus de même priorité (d'une même file d'attente) ont un ordonnancement circulaire avec un quantum de 4 unités,
- La priorité d'un processus augmente de 4 lorsqu'il est mis en attente d'E/S et diminue de 4 lorsqu'il consomme son quantum, tout en maintenant sa priorité dans l'intervalle  $[0, 19]$ .
- Le temps de commutation de contexte est égal à 0.
- Tous les processus arrivent dans le système à l'instant 0.
- L'ordonnanceur met à jour la priorité d'un processus à la fin de son quantum ou lorsqu'il se bloque.

**Donnez** le diagramme de Gantt montrant l'ordonnancement des processus. Pour chaque processus, indiquez l'évolution de sa priorité.



## **Cas de Linux**

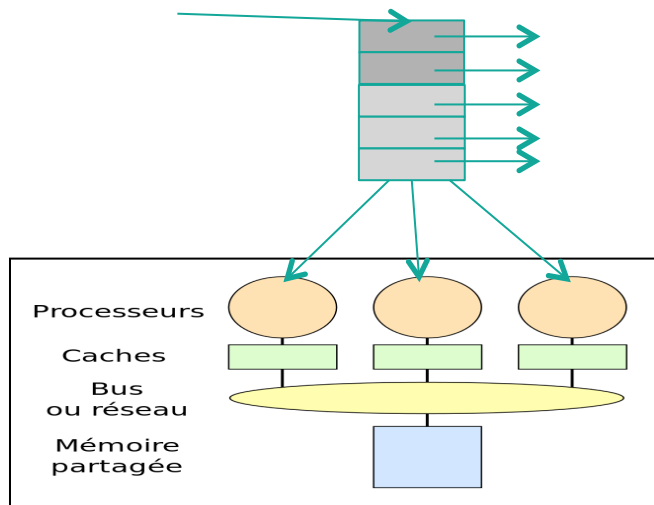
## Cas de Linux (noyau 1.2 - noyau 2.2)

- 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 préemptif à files multiples (ordonnancements FIFO et circulaire)**.
  - ➔ au moins 3 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 (100 ms par défaut) et une priorité fixe sont associés à chaque processus de cette classe.
    - **Les processus en temps partagé** (SCHED\_OTHER, classe par défaut) : un quantum de temps et deux priorités de base et dynamique sont associées à chaque processus.

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

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

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




[https://fr.wikipedia.org/wiki/Symmetric\\_multiprocessing](https://fr.wikipedia.org/wiki/Symmetric_multiprocessing)

**Avantages** : il permet une meilleure utilisation des processeurs et une répartition équitable vis-à-vis des processus.

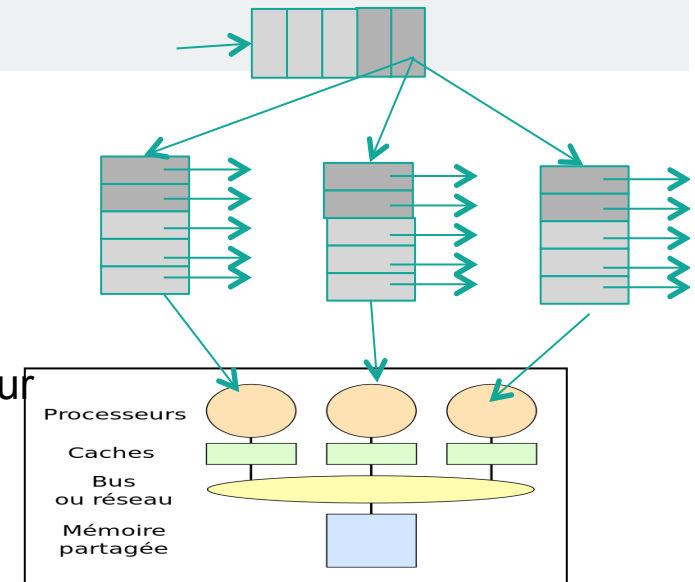
**Inconvénients** : il n'est pas « scalable » (car il doit assurer un accès en exclusion mutuelle à la file d'attente commune) et a une plus faible localité au niveau des caches (plus de « cache miss »).

## Cas de Linux (noyau 2.6) (3)

-  Linux 2.6 : implémente un **ordonnancement préemptif avec files multiples (140 niveaux de priorité)**.
  - Il ordonnance **les threads** et non pas les processus avec différentes classes de threads : threads temps réel (**SCHED\_FIFO**, **SCHED\_RR**), threads en temps partagé (**SCHED\_OTHER**), etc.
  - Les threads temps réel ont des priorités fixes allant de 0 à 99 (les plus élevées).
  - Les threads en temps partagé (classe par défaut) ont chacun une priorité statique (SP) comprise entre 100 à 139 et une priorité dynamique (DP).
  - Chaque thread en temps partagé a une valeur **nice** qui vaut 0 par défaut mais qui peut être modifiée par **nice(val)** avec val dans [-20,19]. Seul l'administrateur peut appeler nice(val) avec val <0. La priorité statique du thread est : **SP = 120 + nice**.

## Cas de Linux (noyau 2.6) (4)

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



[https://fr.wikipedia.org/wiki/Symmetric\\_multiprocessing](https://fr.wikipedia.org/wiki/Symmetric_multiprocessing)

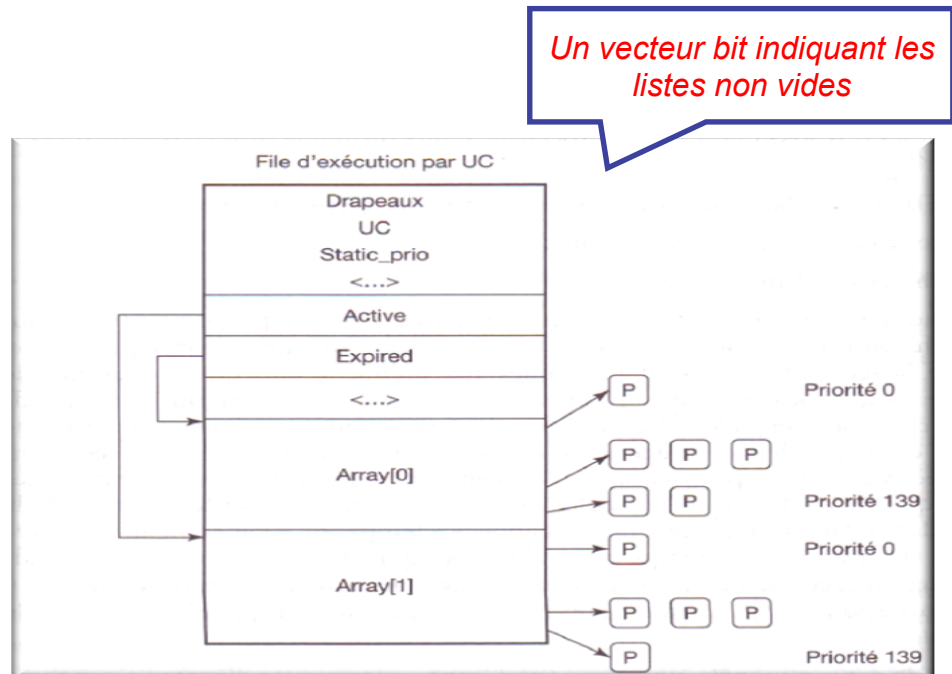
➔ « Load Balancer » pour rééquilibrer périodiquement la charge entre les processeurs avec migration possible entre les « runqueues ». Load Balancer est aussi appelé si une file se vide (thread idle).

Toujours à la recherche d'un meilleur algorithme ?



## Cas de Linux (noyau 2.6) (5)

- Chaque file d'attente est composée de deux tableaux «Active»/«Expired».
- L'ordonnanceur sélectionne du tableau Active le thread le plus prioritaire en tête de file.
- Si le thread consomme son quantum, il est déplacé vers le tableau Expired.
- Lorsque le tableau Active devient vide, l'ordonnanceur permute les pointeurs Active et Expired.



## Cas de Linux (noyau 2.6) (5)

- Il attribue des quanta variables selon les priorités (quanta plus élevés pour les plus prioritaires) :

$$\text{If } (SP < 120): qt = (140 - SP) \times 20$$
$$\text{if } (SP \geq 120): qt = (140 - SP) \times 5$$

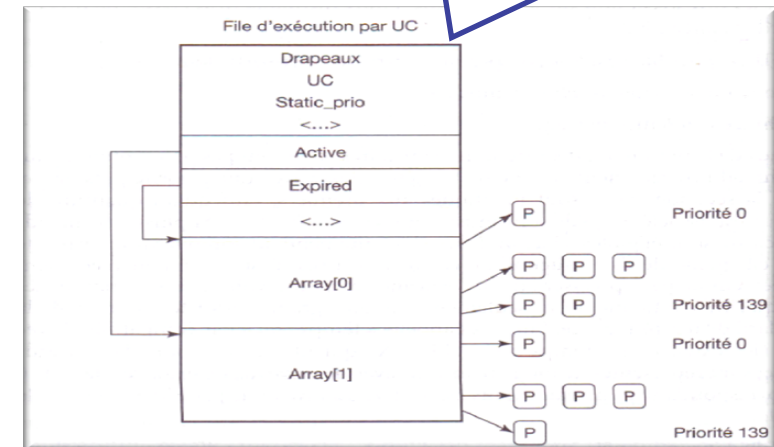
- La priorité dynamique d'un thread est recalculé juste avant de le déplacer vers le tableau Expired :

$$DP = \min(139, \max(100, SP + \text{Bonus}))$$

Bonus  $\in [0,5]$  pour les threads qui consomment beaucoup de temps CPU. Bonus  $\in [-5,0[$  pour les threads qui attendent beaucoup.

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

Un vecteur bit indiquant les listes non vides

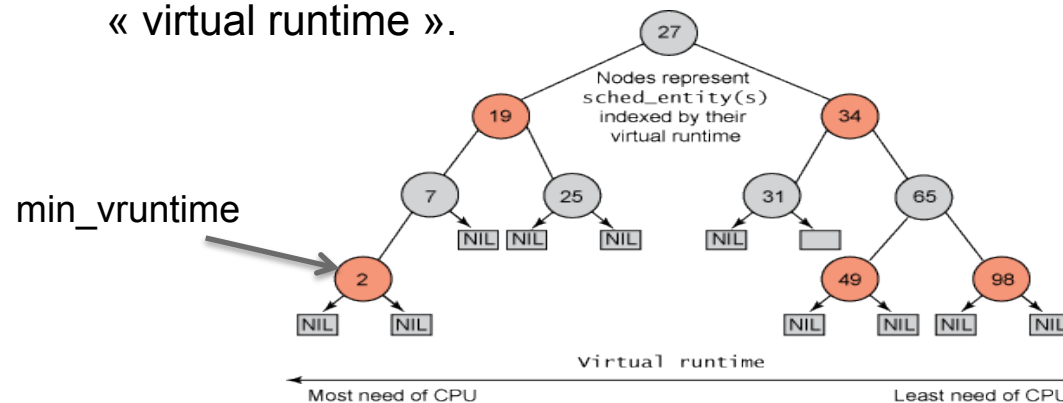


**Contrairement aux algorithmes précédents (en  $O(n)$ ), la décision d'ordonnancement est indépendante du nombre de threads dans le système (complexité en temps est  $O(1)$  par rapport au nombre de threads  $n$ ).**

# Cas de Linux (noyau 2.6.21 ...) (6)

<http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/>  
<http://www.ece.ubc.ca/~sasha/papers/eurosys16-final29.pdf>

- **Linux 2.6.21 ... (à partir de 2007)** : implémente un nouvel ordonnanceur appelé Completely Fair Scheduler (CFS) pour les threads **SCHED\_OTHER**.
  - CFS utilise des **arbres binaires de recherche** pour représenter les threads prêts (un arbre binaire de recherche par processeur).
  - Chaque nœud d'un arbre correspond à un thread prêt et la clé du nœud est le **temps pondéré que le thread a passé en exécution**. Ce temps est désigné par **vruntime** pour « virtual runtime ».



**CFS sélectionne le thread qui a le plus petit vruntime.**

**Comment vruntime est calculé ?**

## Cas de Linux (noyau 2.6.21 ...) (7)

- CFS associe à **chaque tâche une valeur nice  $\in [-20,19]$  et un poids.**
- La valeur nice est considérée comme une sorte de priorité. Une tâche avec une valeur nice élevée a une priorité plus basse que celle qui a une valeur plus grande. Cette valeur est utilisée pour associer un poids à la tâche **prio\_to\_weight [nice+20]**.

```
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, };
```

- Si la valeur nice d'une tâche est 0, son poids est 1024.
- Si la valeur nice d'une tâche est 10, son poids est 110.
- **Les tâches plus prioritaires ont des poids plus élevés.**
- **À quoi servent ces poids ?**

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

- Si un thread vient de consommer x nanosecondes de temps CPU, son temps d'exécution virtuel est mis à jour :

$$\text{vruntime} = \text{vruntime} + x / \text{prio\_to\_weight}[\text{nice}+20].$$

- vruntime** croît moins vite pour les threads prioritaires.
- vruntime** d'un nouveau thread est :  
 $\text{min\_vruntime} * 1024 / \text{prio\_to\_weight}[\text{nice}+20].$
- CFS cherche à favoriser les threads qui ont été les moins servis en temps CPU et qui, en même temps, sont prioritaires.

```
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, };
```

Quel est le temps d'allocation du processeur ?

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

- Les poids des threads servent aussi à déterminer le pourcentage de temps à allouer à chaque thread :  
**poids du thread / somme des poids des threads prêts.**

- Ils servent à répartir équitablement le temps CPU entre les threads prêts.

```
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, };
```

- Supposons 3 tâches A, B, C prêtes. Leurs valeurs de nice sont respectivement 0, 0 et 2. Les pourcentages de temps à allouer aux tâches sont :

A :  $1024 / (1024 + 1024 + 820) = 35.7\%$

B :  $1024 / (1024 + 1024 + 820) = 35.7\%$

C :  $820 / (1024 + 1024 + 820) = 28,6\%$

- Les temps maximums alloués aux tâches sont (quanta) :


A :  $35.7\% * \text{period} = 4,28 \text{ ms}$

B :  $35.7\% * \text{period} = 4,28 \text{ ms}$

C :  $28,6\% * \text{period} = 3,43 \text{ ms}$

Où  $\text{period} = 4 \text{ ms} * \text{nb}$ , nb est le nombre de tâches prêtes.

## Cas de Linux (noyau 2.6.21 ...) (10)

- 
- Le thread en cours d'exécution est préempté s'il y a un thread plus prioritaire (un thread avec un plus petit vruntime) devient prêt ou s'il a consommé son quantum.
  - Le thread préempté est inséré dans l'arbre binaire de recherche après avoir mis à jour son vruntime.
  - Plusieurs améliorations depuis concernant, notamment, les algorithmes de répartition des temps CPU et d'équilibrage des charges des processeurs.



# **Ordonnancement temps réel**



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

- “Real time in operating systems (RTOS): POSIX Standard 1003.1  
The ability of the operating system to provide a required level of service in a bounded response time.”
- Le noyau d'un RTOS assure les fonctions de base d'un système d'exploitation (gestion des tâches, gestion des interruptions, gestion du temps, gestion de la mémoire, etc.).
- Il doit, en plus,
  - **garantir des temps de réponses bornés et acceptables aux demandes de services, et**
  - **permettre de développer et de mettre en œuvre des applications temps réel :**  
{ tâches concurrentes, communicantes avec éventuellement des **contraintes temporelles** }.

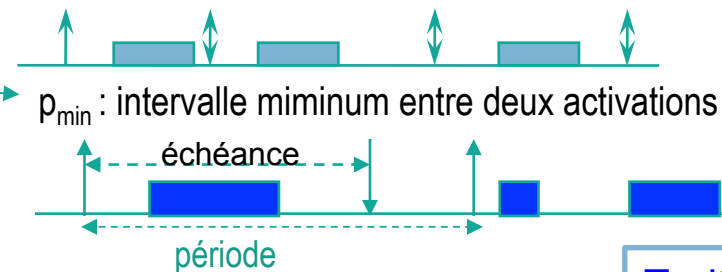
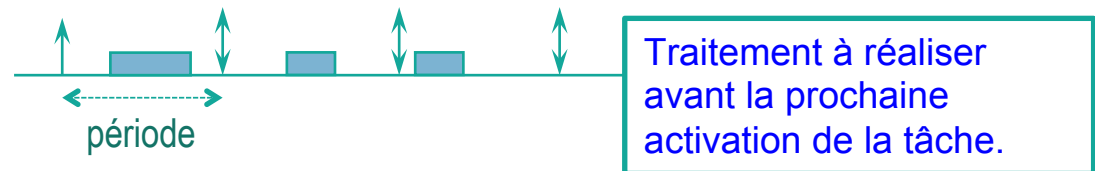
## Qu'est ce qu'un système d'exploitation temps réel ? (2)

- Une tâche temps réel est une tâche qui est activée à des **intervalles de temps réguliers ou irréguliers** → **tâche périodique, sporadique ou apériodique (par une autre tâche)** ;
- A chaque activation, elle réalise un traitement puis se met en attente de la prochaine activation;
- Le traitement réalisé par une tâche suite à son activation **doit être complété avant la prochaine activation, avant une échéance relative à la date de cette activation.**
- Le traitement réalisé par une tâche suite à son activation **doit être commencé avant ou après un certain délai relatif à la date de cette activation.**
- Les tâches temps réel peuvent être aussi soumises à des contraintes de dépendances (tâches dépendantes et indépendantes).

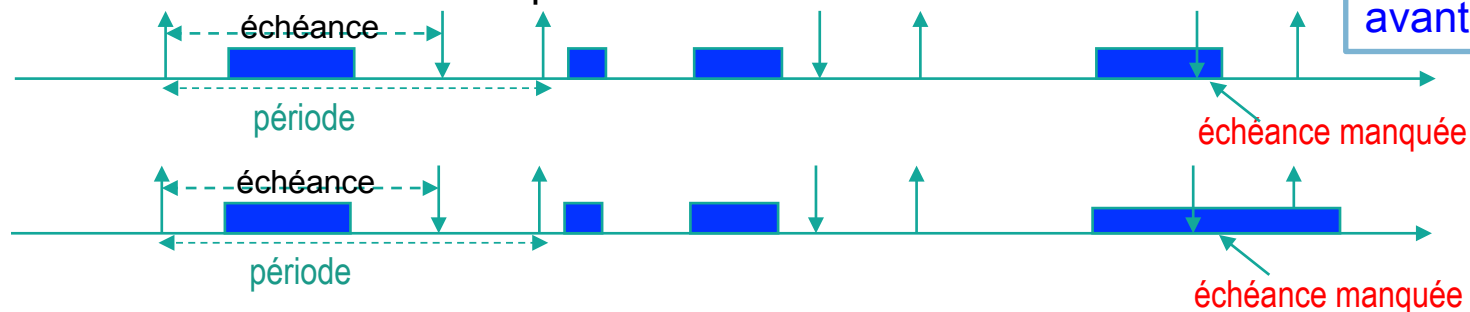
# Qu'est ce qu'un système d'exploitation temps réel ? (3)

## Tâches temps réel : Contraintes temporelles

- Tâche périodique
- Tâche apériodique (sporadique)
- Tâche périodique avec échéance



### → Problème d'échéance manquée



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

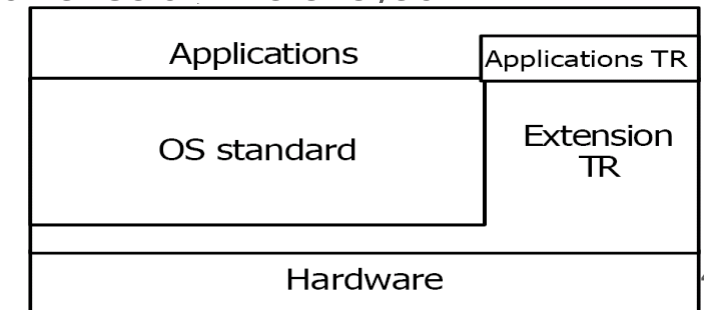
- **Contraintes temporelles d'une tâche périodique  $T_i$  :**
  - Période d'activation :  $P_i$
  - Temps d'exécution (au pire cas) à chaque activation (traitement périodique) de  $T_i$  :  $C_i$
  - Échéance (deadline) :  $D_i$
  - Date d'activation initiale :  $O_i$
  - Délai de début de traitement  $[0, x_i]$  avec  $x_i < P_i - C_i$ .
  - $T_i$  est à échéance sur requête si  $D_i = P_i$
- Tâches synchrones si  $O_i = O_j$ , pour toutes tâches  $T_i$  et  $T_j$

## Qu'est ce qu'un système d'exploitation temps réel ? (5)

- Comment garantir des temps de réponse bornés et satisfaire les contraintes temporelles ?
  - Interdire tous les appels bloquants dans les ISR (Interrupt Service Routine) 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, etc.
  - Exécuter les tâches temps réel dans l'espace noyau.
  - noyau préemptif.
  - Ordonnancement préemptif à base de priorités qui permet de tenir compte des contraintes temporelles (activation périodique d'une tâche, date de première activation, etc.).
- Le bon fonctionnement d'une application temps réel est, notamment, conditionné par le respect de leurs contraintes temporelles.

# Qu'est ce qu'un système d'exploitation temps réel ?(6)

- **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 (INF3610),
  - MicroC (INF3610),
  - QNX,
  - LynuxWorks, etc.
- Les extensions existantes de cohabitation du noyau Linux avec un micronoyau :
  - **RT-Linux**,
  - RTAI, etc.



# Ordonnancement de tâches temps réel

## RT-Linux ([http://www.mnis.fr/ocera\\_support/rtos/c1450.html](http://www.mnis.fr/ocera_support/rtos/c1450.html))

- Les tâches RT-Linux s'exécutent dans l'espace noyau au même titre que l'ordonnanceur temps réel.
- La structure « `rt_task_struct` » (ou `RT_TASK`) des tâches temps réel dans RT-Linux contient notamment les champs suivants :
  - État (inactive, ready, active, delayed, zombie);
  - Priorité;
  - Période;
  - Date de la prochaine activation.

# Ordonnancement de tâches temps réel

## RT-Linux (2)



Deux types de tâches sont gérées dans RT-Linux :

- Tâches périodiques :
  - exécutent périodiquement un traitement.
  - sont activées périodiquement par le système.
- Tâches apériodiques :
  - exécutent à des intervalles de temps irréguliers un traitement.
  - sont généralement activées par d'autres tâches (relation de précedence ou arrivée d'un événement).



# Ordonnancement de tâches temps réel

## RT-Linux (3)

- Création d'une tâche (état inactive) :  
`int rt_task_init ( RT_TASK *task, void (fn)(int data), int data,  
int stack_size, int priority );`
- Activation d'une tâche (état ready) :  
`int rt_task_wakeup ( RT_TASK *task );`
- Rendre une tâche inactive (état inactive) :  
`int rt_task_suspend (RT_TASK *task);`

# Ordonnancement de tâches temps réel

## RT-Linux (4)

- 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 (état delayed) : **int rt\_task\_wait (void);**
- Suppression d'une tâche RT-Linux (état zombie) :  
**int rt\_task\_delete (RT\_TASK \**task*);**

# Ordonnancement de tâches temps réel

## RT-Linux (5)

- Ordonnancement (par défaut) préemptif à priorités statiques  
→ Il sélectionne la tâche prête la plus prioritaire.
- Ordonnancement Rate Monotonic priority assignment (RMA)
- Ordonnancement Earliest Deadline First (EDF)



# Ordonnancement de tâches indépendantes

Soient  $n$  tâches périodiques  $T_1, T_2, \dots, T_n$ . Si elles sont ordonnançables sur l'intervalle  $[0, \text{Max}(O_1, O_2, \dots, O_n) + \text{PPCM}(P_1, P_2, \dots, P_n)]$  alors elles sont ordonnançables.

J. Y. T. Leung, M. L. Merrill, «A note on Preemptive Scheduling of Periodic Real-Time Tasks», Information Processing Letters, vol. 11 n°3, pp. 115-118, 1980.

# Rate monotonic priority assignment (RMA)

## [Liu & Layland, 1973]

- RMA est un ordonnancement préemptif à priorités statiques où la priorité d'une tâche est inversement proportionnelle à sa période. Plus la période est petite, plus la priorité est grande.
- Un ensemble de  $n$  tâches périodiques indépendantes telles que  $D_i = P_i$  pour  $i=1$  à  $n$ , est ordonnançable si (**condition suffisante de Liu et Layland**)(<https://www.di.ens.fr/~pouzet/cours/systeme/bib/liu73.pdf>) :

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

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

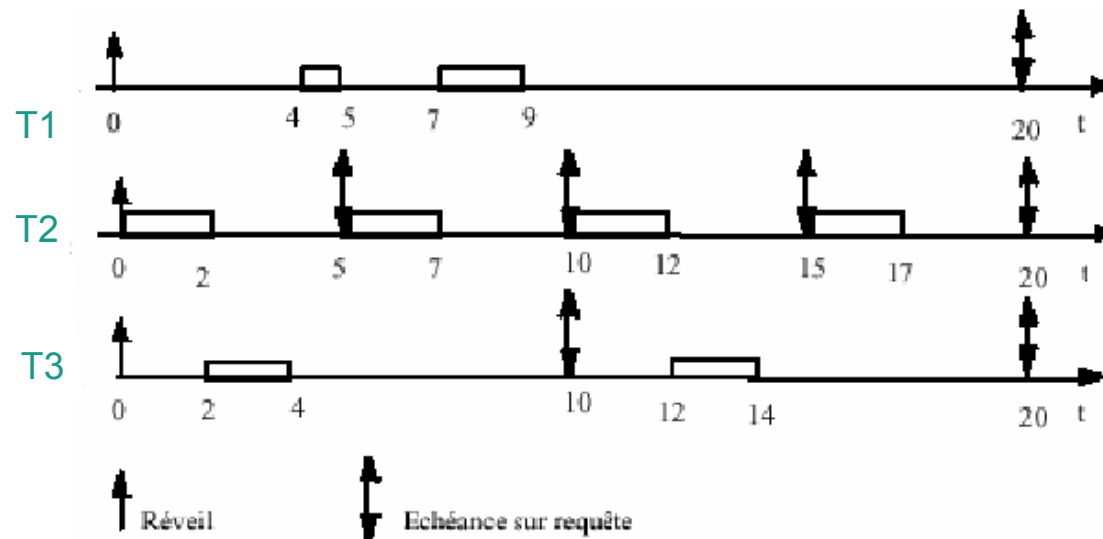
$n \rightarrow \infty \Rightarrow$  borne = 69.3 %

$U_i = C_i / P_i$  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  
 $3/20 + 2/5 + 2/10 = 75 \% < 78 \% \Rightarrow$  Ordonnançable



Simulation : Intervalle d'étude :  
 $[0, \text{Max}(O1, O2, O3) + \text{PPCM}(P1, P2, P3)]$

$\text{prio}(T1) < \text{prio}(T3) < \text{prio}(T2)$

$\Rightarrow$  Ordonnançable

# Deadline monotonic priority assignment (DMA)

## [Leung & Whitehead, 1985]

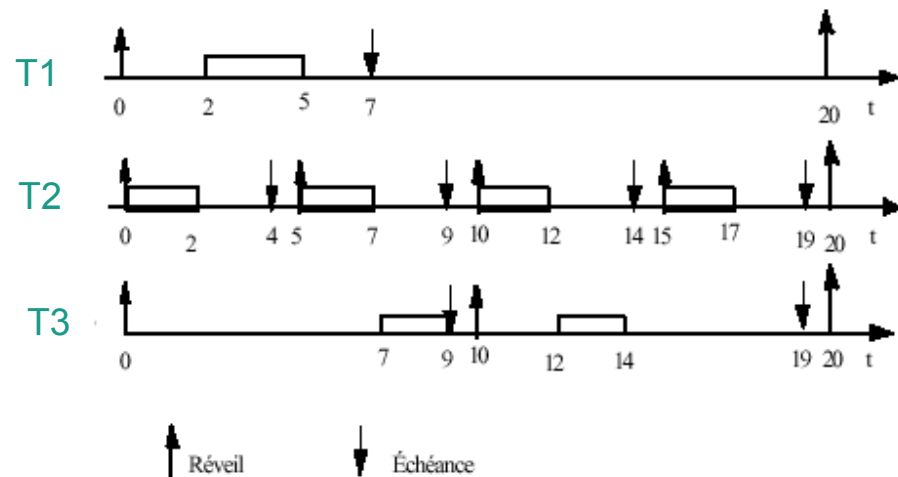
- DMA est un ordonnancement préemptif à priorités statiques où la priorité d'une tâche est inversement proportionnelle à son *échéance* (plus l'échéance est petite, plus la priorité est grande).
- Un ensemble de  $n$  tâches périodiques indépendantes est ordonnançable si :

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

## 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  
 $3/7 + 2/4 + 2/9 = 115\%$  non  $< 78\%$       on ne peut pas conclure



prio(T3) < prio(T1) < prio(T2)

→ Ordonnançable



# Earliest Deadline First (EDF)

## [Liu & Layland, 1973]

- EDF est un ordonnancement préemptif à priorités dynamiques.
- La tâche la plus prioritaire (parmi les tâches prêtes) est celle dont l'échéance absolue est la plus proche → priorités dynamiques. Il est applicable aussi bien pour des tâches périodiques qu'apériodiques.
- Ordonnançable :
  - Pour les tâches à échéance sur requête ( $P_i = D_i$  pour toute tâche  $T_i$ ),

$$\text{CNS : } \sum_{i=1}^n \frac{C_i}{P_i} \leq 1$$

- Pour les autres cas :

$$\text{CN : } \sum_{i=1}^n \frac{C_i}{P_i} \leq 1$$

$$\text{CS : } \sum_{i=1}^n \frac{C_i}{D_i} \leq 1$$

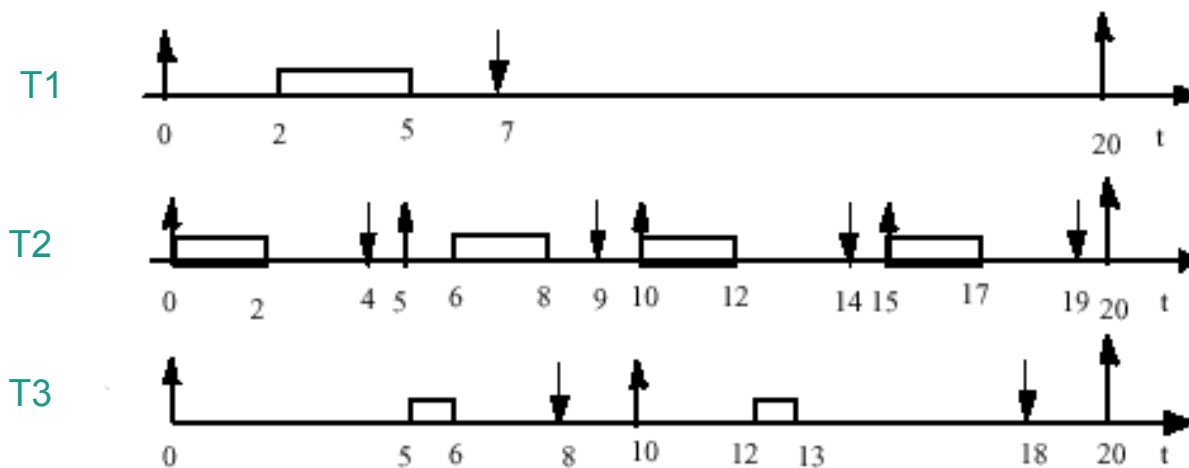
## Earliest Deadline First (EDF) (2)

### 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  $\leq 1$  on ne peut pas conclure

→ ordonnançable

↑ Réveil

↓ Echéance



## **Ordonnancement de tâches périodiques dépendantes (en présence de ressources partagées)**

# Problèmes de partages de ressources

- Les tâches partagent des ressources à accès exclusif.
- Lorsqu'une tâche  $T_x$  demande une ressource déjà allouée à une autre tâche  $T_y$ ,  $T_x$  se met en attente de la ressource.
- Ordonnancement à base de priorités
  - ⇒ 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 :** Soient les tâches T1, T2 et T3 suivantes :

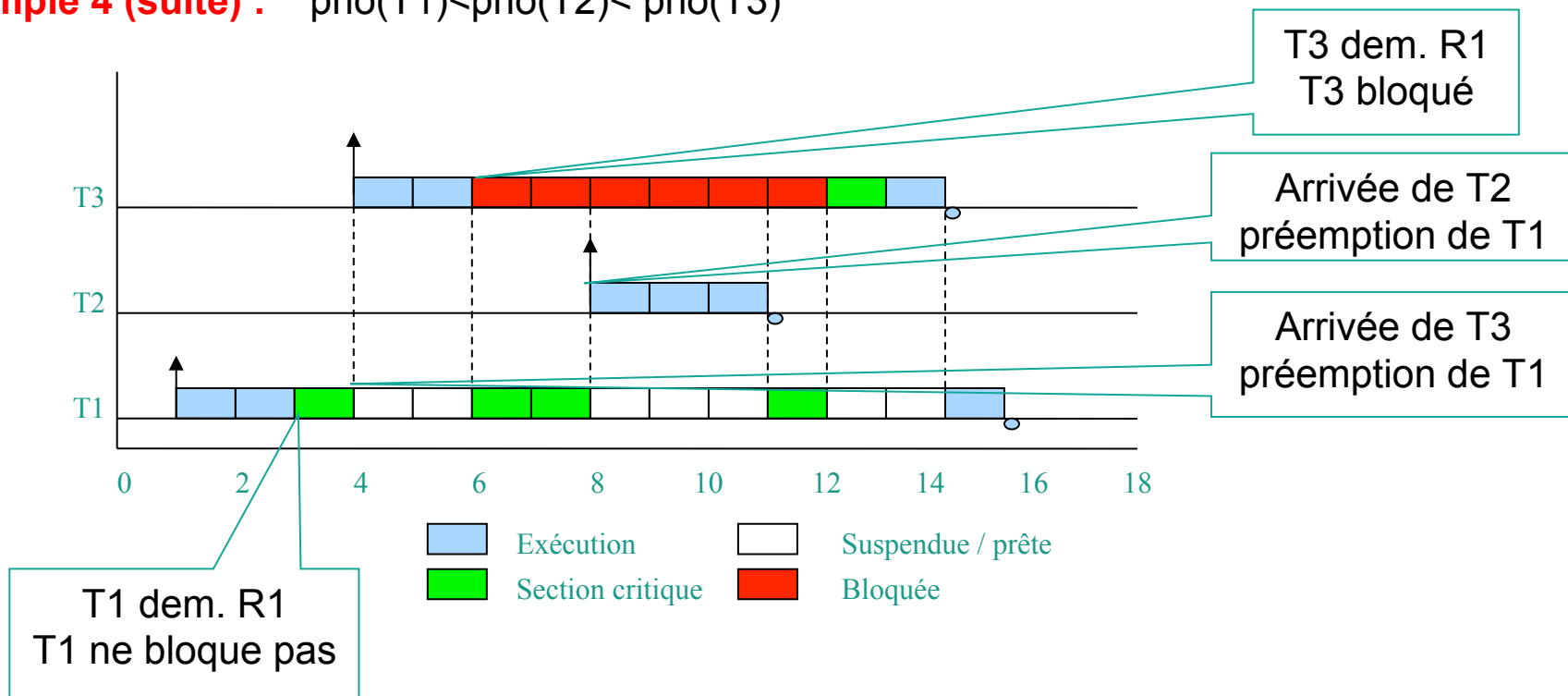
Tâche	Date de départ	Priorité	Séquence d'exécution
T3	4	3	EER1E
T2	8	2	EEE
T1	1	1	EER1R1R1R1E

Donnez le diagramme de Gantt dans le cas d'un ordonnancement préemptif à priorités.

# Problèmes de partages de ressources

## Inversion de priorités (2)

**Exemple 4 (suite) :**  $\text{prio}(T1) < \text{prio}(T2) < \text{prio}(T3)$



# Problèmes de partages de ressources

## Inversion de priorités (3)



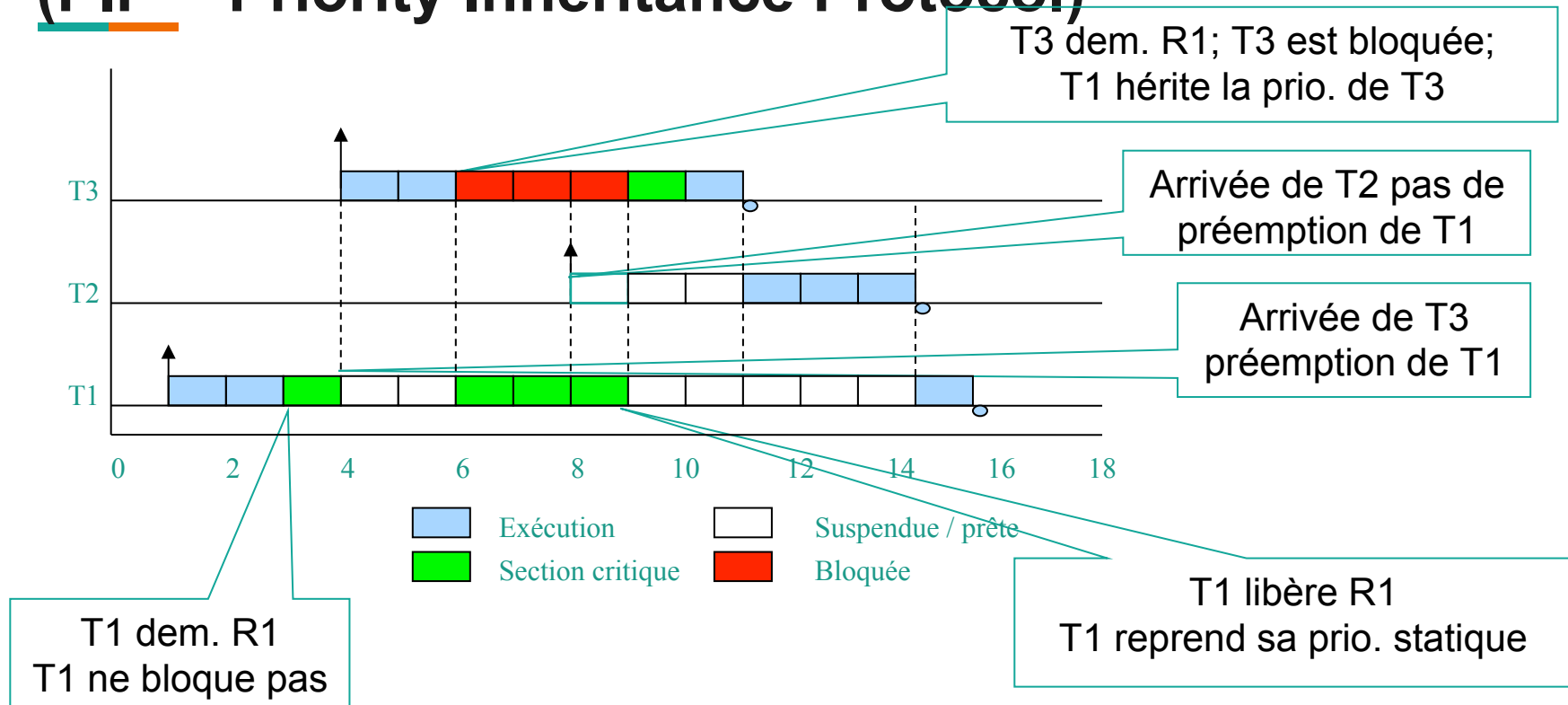
- Traitement du problème d'inversion de priorités :
    - Protocole d'héritage de priorités [Kaiser, 1982] [Sha, 1990]
    - OCPP (Original Ceiling Priority Protocol) [Chen, 1990] [Sha, 1990]
    - ICPP (Immediate Ceiling Priority protocol).
- ➔ Ils permettent de borner la durée de l'inversion de priorités.

# 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'exécute alors avec la priorité héritée jusqu'à la libération de la ressource R.
- Elle retrouve sa priorité d'origine (de base) à la libération de la ressource R.



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



## Exercice (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 ordonnancables selon RMA ?
- A-t-on un problème d'inversion de priorités ? 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)

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](http://degeeter.pagesperso-orange.fr/inv_fr.htm))

Suite : Cours INF3610

## Lectures suggérées



- Sections 2.4 (Scheduling) et 10.3.4 (Scheduling in Linux) (pp 148-165, 746-751) *Modern operating Systems*, 4<sup>nd</sup> edition, Andrew S. Tanenbaum, publié par Pearson Education, Prentice-Hall, 2016. **Livre disponible dans le dossier Slides Aut 2019 du site moodle du cours.**

# Annexe : 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 files multiples** (32 niveaux de priorité). Les priorités sont dynamiques.
- Windows **ordonne 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)

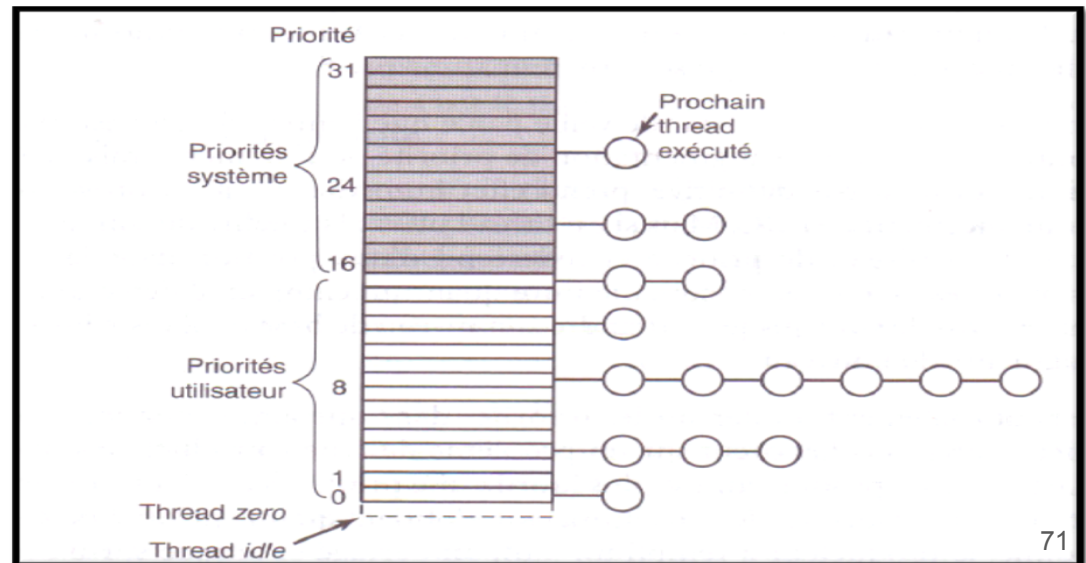
- L'API Win32 fournit deux fonctions pour influencer l'ordonnancement des threads :
  - SetPriorityClass qui définit la classe de priorité de tous les threads du processus appelant (realtime, high, above normal, normal, below normal, idle).
  - SetThreadPriority qui définit la priorité relative d'un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

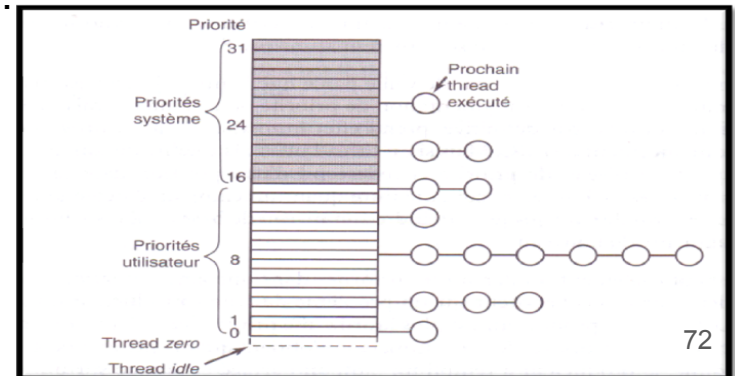
## Cas de Windows (4)

- 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.




Évolution Dynamique des priorités

# Cas de Windows (7)

- **Problème d'inversion de priorités :**
  - Supposez un producteur et un consommateur sont dans le système.
  - 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).