Ordonnancement

Fabien Rico

Univ. Claude Bernard Lyon 1

séance 6

Pédro Silva	pedro.silva@ens-lyon.fr	TP
Léo Le Taro	leo.le-taro@inria.fr	TD + TP
Fabien Rico	fabien.rico@univ-lyon1.fr	CM+ TD + TP

Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.

Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.

Cas d'école

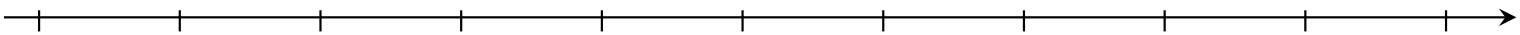
- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.

Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute

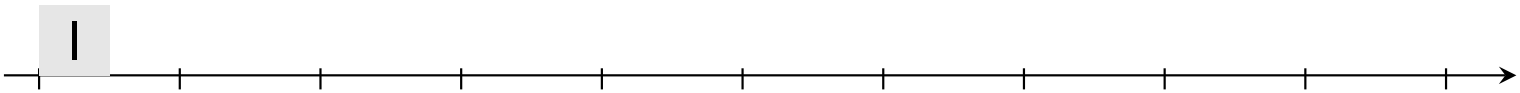
Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



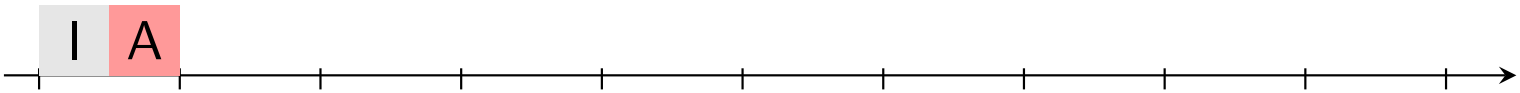
Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



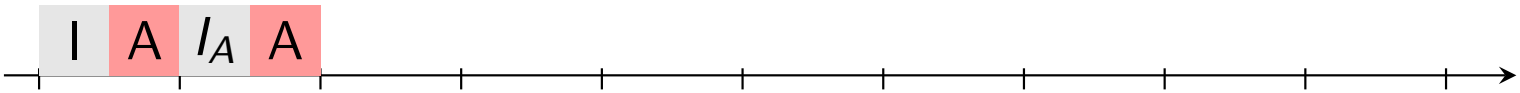
Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :

Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :

Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :

Cas d'école

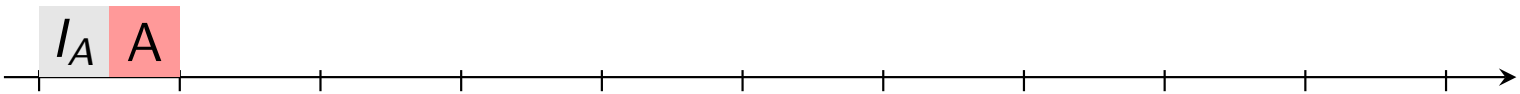
- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :

  - en même temps il faut $5,5$ secondes

Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :

  - en même temps il faut $5,5$ secondes



Cas d'école

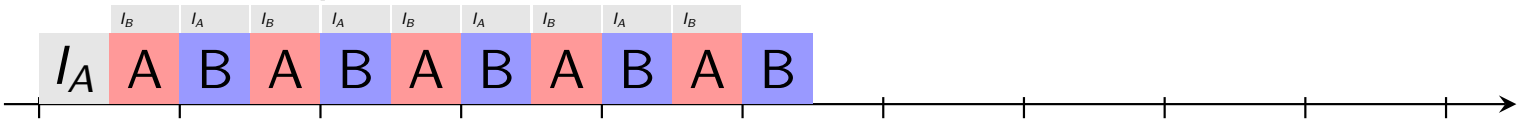
- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :

  - en même temps il faut $5,5$ secondes

Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :

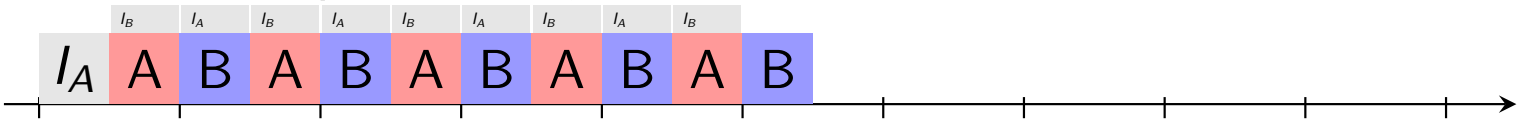
  - en même temps il faut $5,5$ secondes



Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :

  - en même temps il faut 5, 5 secondes

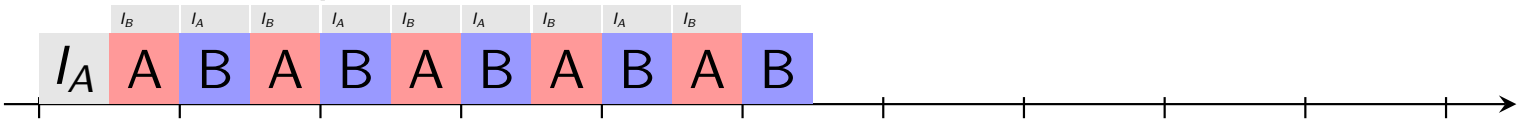


- Si un processus est bloqué, il faut exécuter un autre processus

Cas d'école

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :

  - en même temps il faut $5,5$ secondes



- Si un processus est bloqué, il faut exécuter un autre processus
- Mais ce n'est pas si simple.

Un peu plus de réalisme

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).

Un peu plus de réalisme

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas à priori leur durée.

Un peu plus de réalisme

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas à priori leur durée.
- Plusieurs tâches peuvent être disponibles et assez souvent aucune ne l'est.

Un peu plus de réalisme

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas à priori leur durée.
- Plusieurs tâches peuvent être disponibles et assez souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).

Un peu plus de réalisme

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas à priori leur durée.
- Plusieurs tâches peuvent être disponibles et assez souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).
- Elles peuvent avoir des dépendances.

Un peu plus de réalisme

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas à priori leur durée.
- Plusieurs tâches peuvent être disponibles et assez souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).
- Elles peuvent avoir des dépendances.
- Il y a parfois des contraintes externes (pilote).

Un peu plus de réalisme

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas à priori leur durée.
- Plusieurs tâches peuvent être disponibles et assez souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).
- Elles peuvent avoir des dépendances.
- Il y a parfois des contraintes externes (pilote).
- Le comportement des tâches est différent :

Un peu plus de réalisme

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas à priori leur durée.
- Plusieurs tâches peuvent être disponibles et assez souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).
- Elles peuvent avoir des dépendances.
- Il y a parfois des contraintes externes (pilote).
- Le comportement des tâches est différent :
  - Processus interactifs (court, temps de réponse important)

Un peu plus de réalisme

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas à priori leur durée.
- Plusieurs tâches peuvent être disponibles et assez souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).
- Elles peuvent avoir des dépendances.
- Il y a parfois des contraintes externes (pilote).
- Le comportement des tâches est différent :
  - Processus interactifs (court, temps de réponse important)
  - Processus avec beaucoup d'entrée/sortie (analyse de disque)

Un peu plus de réalisme

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas à priori leur durée.
- Plusieurs tâches peuvent être disponibles et assez souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).
- Elles peuvent avoir des dépendances.
- Il y a parfois des contraintes externes (pilote).
- Le comportement des tâches est différent :
  - Processus interactifs (court, temps de réponse important)
  - Processus avec beaucoup d'entrée/sortie (analyse de disque)
  - Long calculs (rare)

Ordonnancement

Définition (Ordonnancement)

La sélection dans le temps des processus ou threads qui peuvent accéder à un processeur est *l'ordonnancement*. Le but est de maximiser :

- Le débit (nombre de processus traités par unité de temps)
- Le taux utile (le taux du processeur effectivement utilisé pour les tâches utilisateurs)

Il y a plusieurs types d'ordonnancement en fonction de la possibilité d'interrompre une tâche :

- *Ordonnancement collaboratif* : les tâches ne sont pas interruptibles.
- *Ordonnancement préemptif* : le système peut interrompre une tâche à tout moment.

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex **read**, sleep, thread_yield, SwitchToFiber)

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex **read**, sleep, thread_yield, SwitchToFiber)
  - Les tâches critiques sont protégées (écriture, périphérique)

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex **read**, sleep, thread_yield, SwitchToFiber)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais le système est plus instable. Si une tâche non intérruptible entre dans une boucle infinie...

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex **read**, sleep, thread_yield, SwitchToFiber)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais le système est plus instable. Si une tâche non intérruptible entre dans une boucle infinie...
- L'ordonnancement préemptif est plus fiable

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex **read**, sleep, thread_yield, SwitchToFiber)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais le système est plus instable. Si une tâche non intérruptible entre dans une boucle infinie...
- L'ordonnancement préemptif est plus fiable
  - Une erreur dans un programme ne compromet pas le système

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex **read**, sleep, thread_yield, SwitchToFiber)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais le système est plus instable. Si une tâche non intérruptible entre dans une boucle infinie...
- L'ordonnancement préemptif est plus fiable
  - Une erreur dans un programme ne compromet pas le système
  - Le système n'est pas bloqué par certaines tâches trop longues

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex **read**, sleep, thread_yield, SwitchToFiber)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais le système est plus instable. Si une tâche non intérruptible entre dans une boucle infinie...
- L'ordonnancement préemptif est plus fiable
  - Une erreur dans un programme ne compromet pas le système
  - Le système n'est pas bloqué par certaines tâches trop longues
  - Mais cela pose des difficultés pour les tâches critiques (driver, écritures)

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex **read**, sleep, thread_yield, SwitchToFiber)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais le système est plus instable. Si une tâche non intérruptible entre dans une boucle infinie...
- L'ordonnancement préemptif est plus fiable
  - Une erreur dans un programme ne compromet pas le système
  - Le système n'est pas bloqué par certaines tâches trop longues
  - Mais cela pose des difficultés pour les tâches critiques (driver, écritures)
- Par exemple :

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex **read**, sleep, thread_yield, SwitchToFiber)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais le système est plus instable. Si une tâche non intérruptible entre dans une boucle infinie...
- L'ordonnancement préemptif est plus fiable
  - Une erreur dans un programme ne compromet pas le système
  - Le système n'est pas bloqué par certaines tâches trop longues
  - Mais cela pose des difficultés pour les tâches critiques (driver, écritures)
- Par exemple :
  - MsDos et Windows 3.1 utilisaient un ordonnancement collaboratif ;

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex **read**, sleep, thread_yield, SwitchToFiber)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais le système est plus instable. Si une tâche non intérruptible entre dans une boucle infinie...
- L'ordonnancement préemptif est plus fiable
  - Une erreur dans un programme ne compromet pas le système
  - Le système n'est pas bloqué par certaines tâches trop longues
  - Mais cela pose des difficultés pour les tâches critiques (driver, écritures)
- Par exemple :
  - MsDos et Windows 3.1 utilisaient un ordonnancement collaboratif ;
  - Windows 95 Windows 98 utilisaient un mode collaboratif pour certaines tâches.

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex **read**, sleep, thread_yield, SwitchToFiber)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais le système est plus instable. Si une tâche non intérruptible entre dans une boucle infinie...
- L'ordonnancement préemptif est plus fiable
  - Une erreur dans un programme ne compromet pas le système
  - Le système n'est pas bloqué par certaines tâches trop longues
  - Mais cela pose des difficultés pour les tâches critiques (driver, écritures)
- Par exemple :
  - MsDos et Windows 3.1 utilisaient un ordonnancement collaboratif ;
  - Windows 95 Windows 98 utilisaient un mode collaboratif pour certaines tâches.
  - Windows Nt, Xp et les suivants, MacOs, Unix, Linux... utilisent le préemptif.

Critère de performance

Un algorithme d'ordonnancement peut être jugé selon plusieurs critères :

- taux utile : utilisation du processeur ;

Critère de performance

Un algorithme d'ordonnancement peut être jugé selon plusieurs critères :

- taux utile : utilisation du processeur ;
- débit : nombre de travaux réalisés ;

Critère de performance

Un algorithme d'ordonnancement peut être jugé selon plusieurs critères :

- taux utile : utilisation du processeur ;
- débit : nombre de travaux réalisés ;
- temps de réponse : surtout pour les processus interactifs ;

Critère de performance

Un algorithme d'ordonnancement peut être jugé selon plusieurs critères :

- taux utile : utilisation du processeur ;
- débit : nombre de travaux réalisés ;
- temps de réponse : surtout pour les processus interactifs ;
- temps d'exécution moyen d'un ensemble de tâches ;

Critère de performance

Un algorithme d'ordonnancement peut être jugé selon plusieurs critères :

- taux utile : utilisation du processeur ;
- débit : nombre de travaux réalisés ;
- temps de réponse : surtout pour les processus interactifs ;
- temps d'exécution moyen d'un ensemble de tâches ;
- temps de traitement total d'un ensemble de tâches ;

Critère de performance

Un algorithme d'ordonnancement peut être jugé selon plusieurs critères :

- taux utile : utilisation du processeur ;
- débit : nombre de travaux réalisés ;
- temps de réponse : surtout pour les processus interactifs ;
- temps d'exécution moyen d'un ensemble de tâches ;
- temps de traitement total d'un ensemble de tâches ;
- équité entre les tâches (au moins assurer l'absence de famine).

Critère de performance

Un algorithme d'ordonnancement peut être jugé selon plusieurs critères :

- taux utile : utilisation du processeur ;
- débit : nombre de travaux réalisés ;
- temps de réponse : surtout pour les processus interactifs ;
- temps d'exécution moyen d'un ensemble de tâches ;
- temps de traitement total d'un ensemble de tâches ;
- équité entre les tâches (au moins assurer l'absence de famine).

Critère de performance

Un algorithme d'ordonnancement peut être jugé selon plusieurs critères :

- taux utile : utilisation du processeur ;
- débit : nombre de travaux réalisés ;
- temps de réponse : surtout pour les processus interactifs ;
- temps d'exécution moyen d'un ensemble de tâches ;
- temps de traitement total d'un ensemble de tâches ;
- équité entre les tâches (au moins assurer l'absence de famine).

Ces critères sont mutuellement contradictoires par exemple pour minimiser le temps de réponse moyen il faut favoriser les petites tâches.

Ordonnancement à plusieurs niveaux

- Un processus élu n'utilise le processeur que pendant un court laps de temps (p. ex. 10ms).
- Si le choix du processus demande trop de temps (p. ex. 1ms) le taux utile diminue.
- On peut utiliser plusieurs niveaux d'ordonnancement :
  - Ordonnancement à cours terme rapide pour choisir un nouveau processus (p. ex. FIFO dans une liste de priorité)
  - Ordonnancement à long terme (calcul des priorités pour les processus, swap)

FIFO

Le plus simple, on exécute les tâches dans l'ordre d'arrivée : Par exemple :

	T_1	T_2	T_3
Durée	10	2	1
Arrivée	0	0,001	2

Temps de traitement moyen $\backsimeq \frac{10+12+11}{3} = 11\ldots$

Cette méthode a les mêmes problèmes que toutes les queues.

Le plus court d'abord (Short Job First)

C'est la tâche la plus courte qui est exécutée en premier sans tenir compte de l'ordre d'arrivée. Avec le même exemple :

	T_1	T_2	T_3
Durée	10	2	1
Arrivée	0	0,001	2

Le temps de traitement total est plus élevé mais le temps de traitement moyen plus faible :

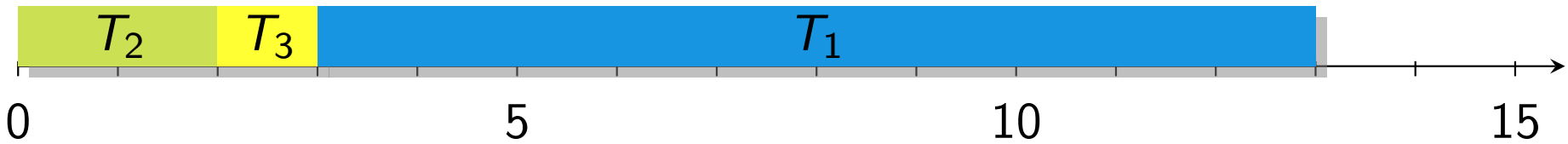
$$\text{Temps de traitement moyen} \simeq \frac{1+5+15}{3} = 7$$

Cet algorithme est efficace, il est optimal pour le temps de traitement moyen si les tâches arrivent en même temps. Mais il demande de connaître ou estimer la durée d'une tâche.

Avec priorité

Selon l'importance de la tâche, le système ou l'utilisateur attribue une priorité. C'est toujours la tâche la plus prioritaire à un instant donné qui est élue :

	T_1	T_2	T_3
Durée	10	2	1
Arrivée	0	0	0
Priorité	10	3	4

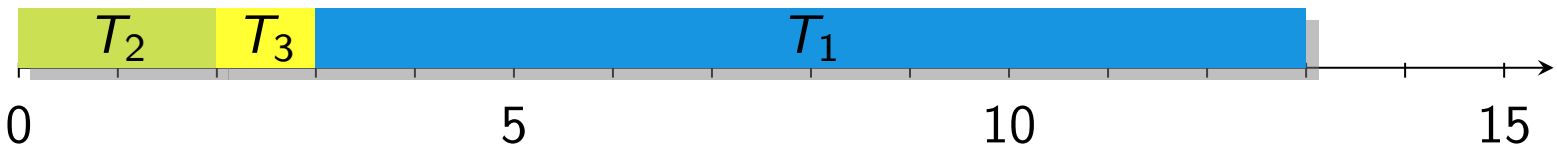


Problème

Les 2 derniers ordonnancements ne sont pas équitables :

- Si un processus est peu prioritaire (ou long pour SJF), il est bloqué par tous les autres

	T_1	T_2	T_3
Durée	10	2	1
Arrivée	0	0	0
Priorité	10	3	4



Problème

Les 2 derniers ordonnancements ne sont pas équitables :

- Si un processus est peu prioritaire (ou long pour SJF), il est bloqué par tous les autres

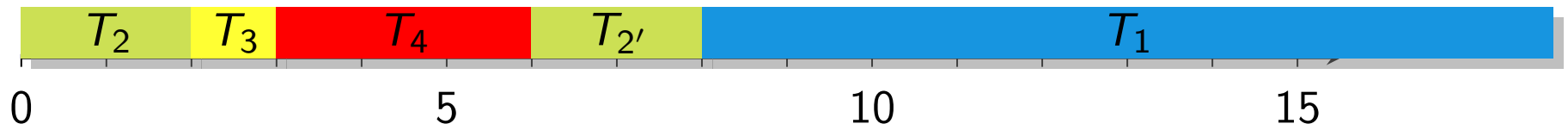
	T_1	T_2	T_3	T_4
Durée	10	2	1	3
Arrivée	0	0	0	2
Priorité	10	3	4	6

Problème

Les 2 derniers ordonnancements ne sont pas équitables :

- Si un processus est peu prioritaire (ou long pour SJF), il est bloqué par tous les autres

	T_1	T_2	T_3	T_4	T_2'
Durée	10	2	1	3	2
Arrivée	0	0	0	2	5
Priorité	10	3	4	6	3



- Il y a famine

Problème

Les 2 derniers ordonnancements ne sont pas équitables :

- Si un processus est peu prioritaire (ou long pour SJF), il est bloqué par tous les autres

	T_1	T_2	T_3	T_4	T_2'
Durée	10	2	1	3	2
Arrivée	0	0	0	2	5
Priorité	10	3	4	6	3

- Il y a famine
- *Idée de solution :* on peut augmenter la priorité des processus qui ne sont pas élus

Problème

Les 2 derniers ordonnancements ne sont pas équitables :

- Si un processus est peu prioritaire (ou long pour SJF), il est bloqué par tous les autres

	T_1	T_2	T_3	T_4	T'_2
Durée	10	2	1	3	2
Arrivée	0	0	0	2	5
Priorité	10	3	4	6	3

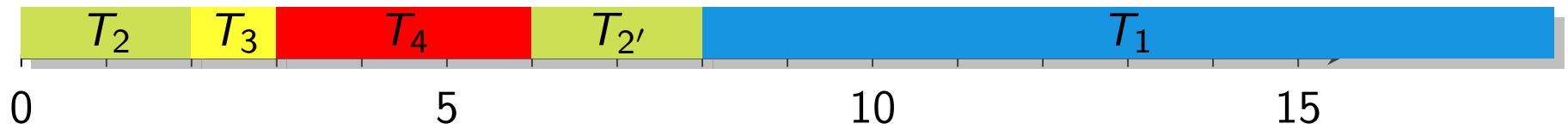
- Il y a famine
- *Idée de solution :* on peut augmenter la priorité des processus qui ne sont pas élus
- *Nouveau problème :* une tâche longue devient prioritaire puis bloque le processeur (mauvais temps de réponse).

Problème

Les 2 derniers ordonnancements ne sont pas équitables :

- Si un processus est peu prioritaire (ou long pour SJF), il est bloqué par tous les autres

	T_1	T_2	T_3	T_4	T'_2
Durée	10	2	1	3	2
Arrivée	0	0	0	2	5
Priorité	10	3	4	6	3



- Il y a famine
- *Idée de solution :* on peut augmenter la priorité des processus qui ne sont pas élus
- *Nouveau problème :* une tâche longue devient prioritaire puis bloque le processeur (mauvais temps de réponse).
- $\Rightarrow$ il faut une préemption pour découper la tâche.

Algorithmes précédents

- FIFO ne peut pas être préemptif (la tâche qui serait élue en cas de préemption est déjà exécutée).
- Les algorithmes SJF et à priorité peuvent l'être. Par ex :

	T_1	T_2	T_3
Durée	10	2	1
Arrivée	0	1	2
Priorité	10	2	5

  - SJF

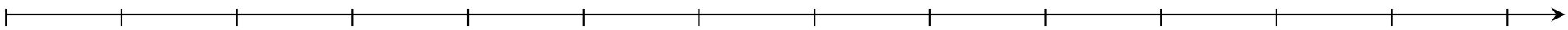
  - Avec priorités

Tourniquet (Round Robin) - I

- On définit un *Quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou interruption par la tâche elle même (appel système, défaut de page, fin...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Interruption	–	3	1

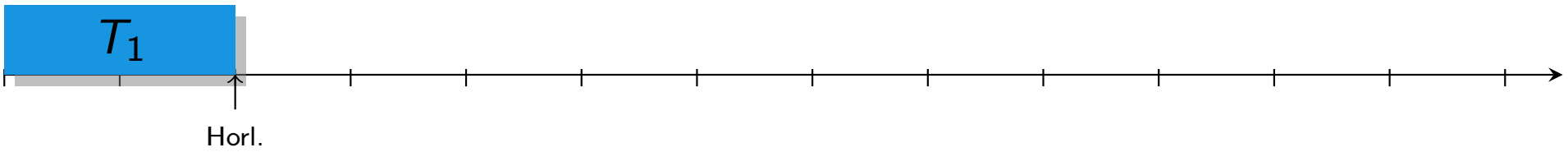


Tourniquet (Round Robin) - I

- On définit un *Quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou interruption par la tâche elle même (appel système, défaut de page, fin...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Interruption	–	3	1

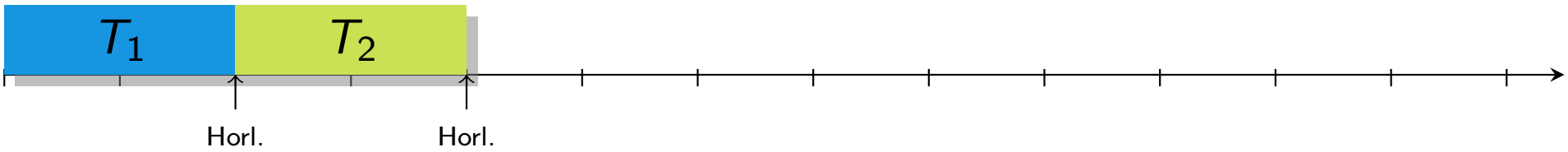


Tourniquet (Round Robin) - I

- On définit un *Quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou interruption par la tâche elle même (appel système, défaut de page, fin...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Interruption	–	3	1

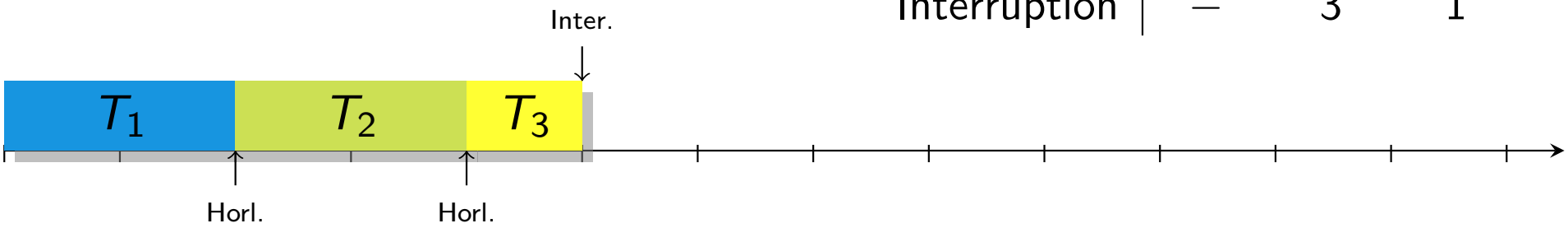


Tourniquet (Round Robin) - I

- On définit un *Quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou interruption par la tâche elle même (appel système, défaut de page, fin...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Interruption	–	3	1

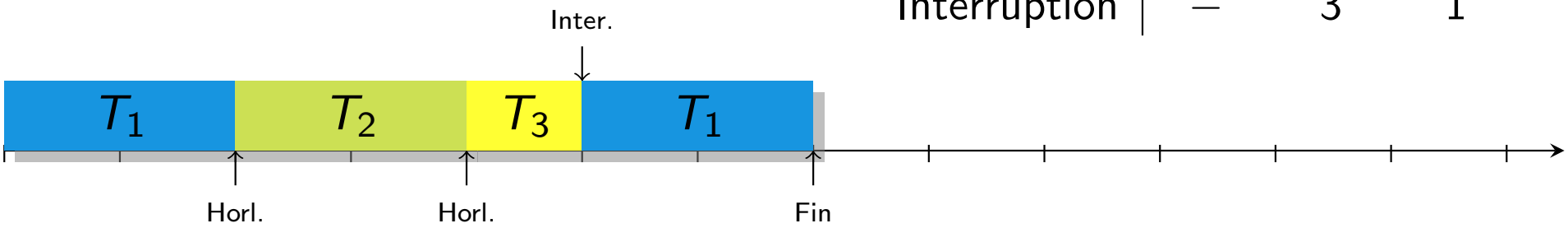


Tourniquet (Round Robin) - I

- On définit un *Quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou interruption par la tâche elle même (appel système, défaut de page, fin...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Interruption	–	3	1

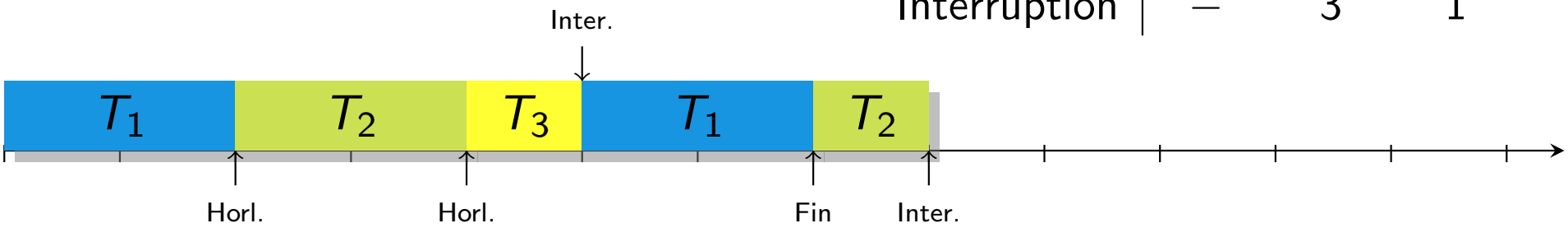


Tourniquet (Round Robin) - I

- On définit un *Quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou interruption par la tâche elle même (appel système, défaut de page, fin...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Interruption	–	3	1

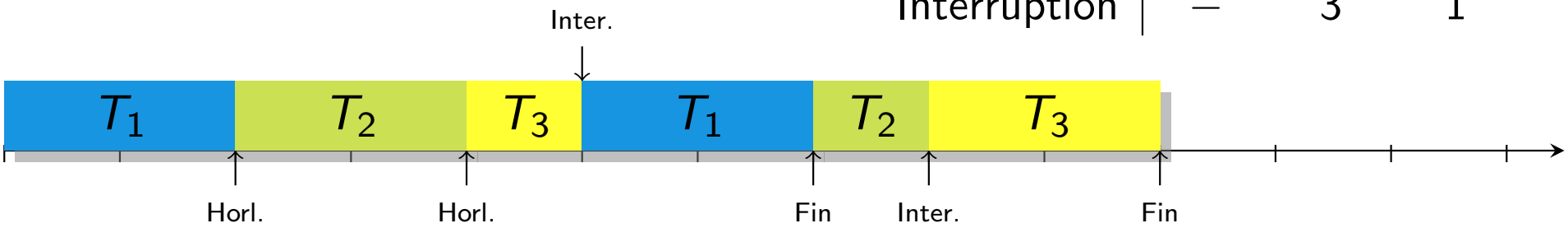


Tourniquet (Round Robin) - I

- On définit un *Quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou interruption par la tâche elle même (appel système, défaut de page, fin...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Interruption	–	3	1

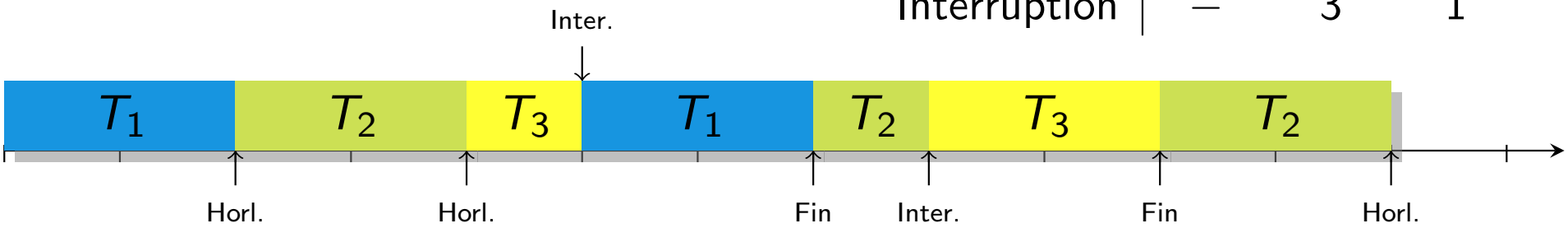


Tourniquet (Round Robin) - I

- On définit un *Quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou interruption par la tâche elle même (appel système, défaut de page, fin...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Interruption	–	3	1

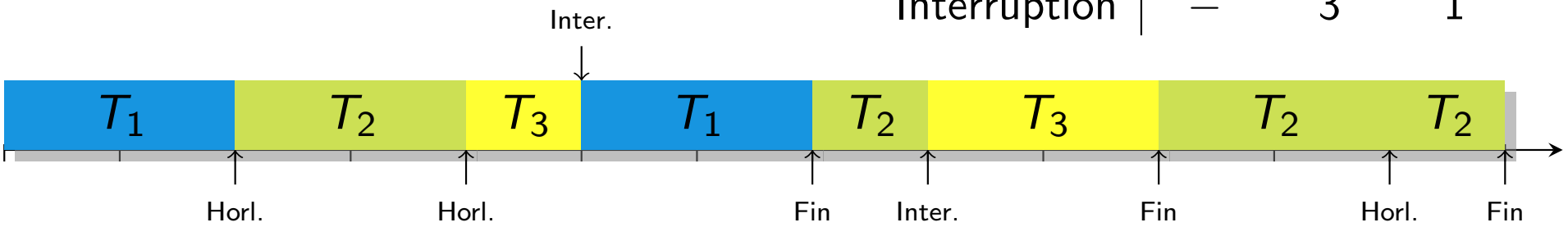


Tourniquet (Round Robin) - I

- On définit un *Quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou interruption par la tâche elle même (appel système, défaut de page, fin...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Interruption	–	3	1



Tourniquet (Round Robin) - II

- Toutes les tâches ont accès au processeur.

Tourniquet (Round Robin) - II

- Toutes les tâches ont accès au processeur.
- Le temps d'attente est borné par $(n-1) \times q$ (n le nb. de tâches, q le quantum).

Tourniquet (Round Robin) - II

- Toutes les tâches ont accès au processeur.
- Le temps d'attente est borné par $(n-1) \times q$ (n le nb. de tâches, q le quantum).
- Si le quantum est trop grand on revient à l'algorithme FIFO.

Tourniquet (Round Robin) - II

- Toutes les tâches ont accès au processeur.
- Le temps d'attente est borné par $(n-1) \times q$ (n le nb. de tâches, q le quantum).
- Si le quantum est trop grand on revient à l'algorithme FIFO.
- Si le quantum est trop court, il y a baisse du taux utile (trop de changements de contexte).

Tourniquet (Round Robin) - II

- Toutes les tâches ont accès au processeur.
- Le temps d'attente est borné par $(n-1) \times q$ (n le nb. de tâches, q le quantum).
- Si le quantum est trop grand on revient à l'algorithme FIFO.
- Si le quantum est trop court, il y a baisse du taux utile (trop de changements de contexte).
- Par exemple, on fixe le quantum de manière à ce que 80% des tâches terminent d'elles-mêmes.

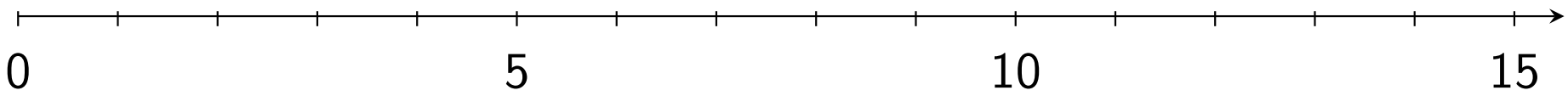
Ordonnancement à queues multiples

- Le système gère plusieurs listes de tâches.
- Chaque liste correspond à une priorité.
- Le scheduler choisit une tâche dans la première liste non vide.
- Par exemple :
  - Processus critique (swap, I/O...).
  - Processus système.
  - Processus interactif.
  - Processus de calcul.
  - Tâche de fond.
- L'ordre utilisé est très important. Par exemple, si le swap est moins prioritaire qu'une tâche qui a besoin de mémoire...
- On peut utiliser des algorithmes différents selon la liste.

Exemple

Avec un quantum de $q = 2$, ordonnancez :

	T_1	T_2	T_3	T_4
Durée	6	3	4	3
Arrivée	0	2	3	8
Priorité	4	2	2	4



Évolution de la priorité - I

Les algorithmes utilisant une priorité fixe sont susceptibles de conduire à la famine des processus peu prioritaires.

- On fait évoluer la priorité en fonction du temps passé hors du processeur.

Évolution de la priorité - I

Les algorithmes utilisant une priorité fixe sont susceptibles de conduire à la famine des processus peu prioritaires.

- On fait évoluer la priorité en fonction du temps passé hors du processeur.
- Par exemple (BSD), tous les n top d'horloge, une nouvelle priorité est recalculée :

$$P_{\text{réelle}} = P_{min} + \frac{T_{cpu}}{n} + 2 \times P_{nice}$$

Avec :

Évolution de la priorité - I

Les algorithmes utilisant une priorité fixe sont susceptibles de conduire à la famine des processus peu prioritaires.

- On fait évoluer la priorité en fonction du temps passé hors du processeur.
- Par exemple (BSD), tous les n top d'horloge, une nouvelle priorité est recalculée :

$$P_{\text{réelle}} = P_{min} + \frac{T_{cpu}}{n} + 2 \times P_{nice}$$

Avec :

- $P_{\text{réelle}}$ la priorité calculée (elle est rammenée à une valeur entre P_{min} et P_{max}.

Évolution de la priorité - I

Les algorithmes utilisant une priorité fixe sont susceptibles de conduire à la famine des processus peu prioritaires.

- On fait évoluer la priorité en fonction du temps passé hors du processeur.
- Par exemple (BSD), tous les n top d'horloge, une nouvelle priorité est recalculée :

$$P_{\text{réelle}} = P_{min} + \frac{T_{cpu}}{n} + 2 \times P_{nice}$$

Avec :

- $P_{\text{réelle}}$ la priorité calculée (elle est rammenée à une valeur entre P_{min} et P_{max}.
- P_{min} la priorité minimum (la meilleur).

Évolution de la priorité - I

Les algorithmes utilisant une priorité fixe sont susceptibles de conduire à la famine des processus peu prioritaires.

- On fait évoluer la priorité en fonction du temps passé hors du processeur.
- Par exemple (BSD), tous les n top d'horloge, une nouvelle priorité est recalculée :

$$P_{\text{réelle}} = P_{min} + \frac{T_{cpu}}{n} + 2 \times P_{nice}$$

Avec :

- $P_{\text{réelle}}$ la priorité calculée (elle est rammenée à une valeur entre P_{min} et P_{max}.
- P_{min} la priorité minimum (la meilleur).
- P_{max} la priorité maximum (la plus mauvaise).

Évolution de la priorité - I

Les algorithmes utilisant une priorité fixe sont susceptibles de conduire à la famine des processus peu prioritaires.

- On fait évoluer la priorité en fonction du temps passé hors du processeur.
- Par exemple (BSD), tous les n top d'horloge, une nouvelle priorité est recalculée :

$$P_{\text{réelle}} = P_{min} + \frac{T_{cpu}}{n} + 2 \times P_{nice}$$

Avec :

- $P_{\text{réelle}}$ la priorité calculée (elle est rammenée à une valeur entre P_{min} et P_{max}.
- P_{min} la priorité minimum (la meilleur).
- P_{max} la priorité maximum (la plus mauvaise).
- T_{cpu} une estimation du temps passé dans le processeur.

Évolution de la priorité - I

Les algorithmes utilisant une priorité fixe sont susceptibles de conduire à la famine des processus peu prioritaires.

- On fait évoluer la priorité en fonction du temps passé hors du processeur.
- Par exemple (BSD), tous les n top d'horloge, une nouvelle priorité est recalculée :

$$P_{\text{réelle}} = P_{min} + \frac{T_{cpu}}{n} + 2 \times P_{nice}$$

Avec :

- $P_{\text{réelle}}$ la priorité calculée (elle est rammenée à une valeur entre P_{min} et P_{max}.
- P_{min} la priorité minimum (la meilleur).
- P_{max} la priorité maximum (la plus mauvaise).
- T_{cpu} une estimation du temps passé dans le processeur.
- P_{nice} une valeur de « Bonne conduite ».

Évolution de la priorité - II

- T_{cpu} est calculée :

Évolution de la priorité - II

- T_{cpu} est calculée :
  - En augmentant de 1 à chaque top d'horloge la valeur du processus exécuté

Évolution de la priorité - II

- T_{cpu} est calculée :
  - En augmentant de 1 à chaque top d'horloge la valeur du processus exécuté
  - En diminuant la valeur au cours du temps :

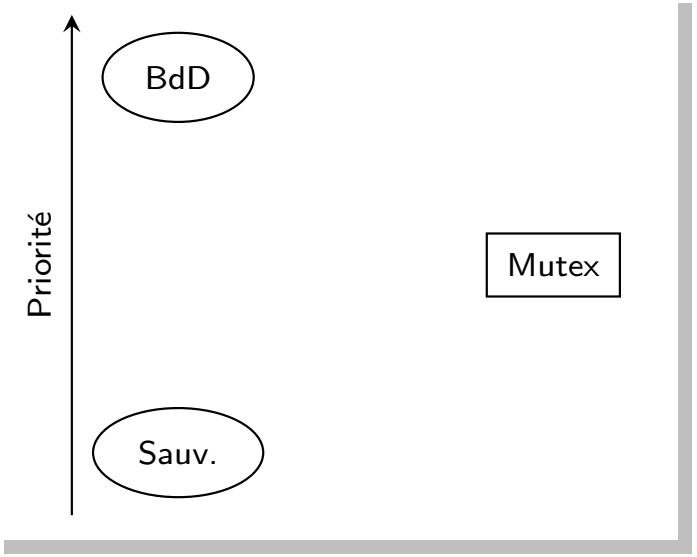
$$\text{Toutes les secondes } T_{cpu} \quad = \quad T_{cpu} \frac{2load}{2load + 1}$$

où *load* est la moyenne du nombre de threads en attente.

Inversion de priorité - I

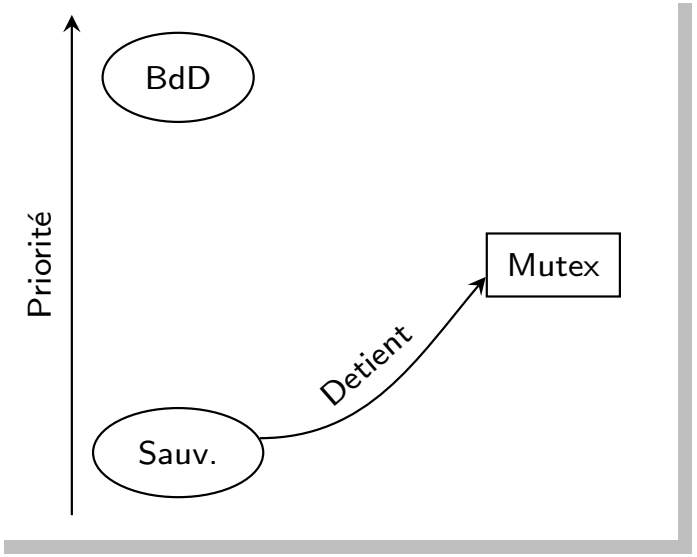
Supposons 2 processus, une base de donnée et sa sauvegarde, qui partagent une zone protégée par un *mutex*.

- Comme la base a un rôle important, sa priorité est élevée.
- La sauvegarde a une priorité faible.



Inversion de priorité - I

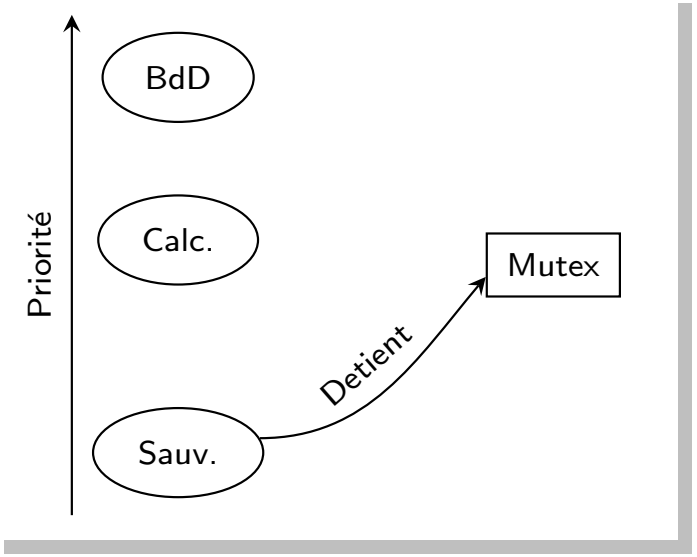
Supposons 2 processus, une base de donnée et sa sauvegarde, qui partagent une zone protégée par un *mutex*.



- Comme la base a un rôle important, sa priorité est élevée.
- La sauvegarde a une priorité faible.
- À un instant donné la sauvegarde acquiert le *mutex*.

Inversion de priorité - I

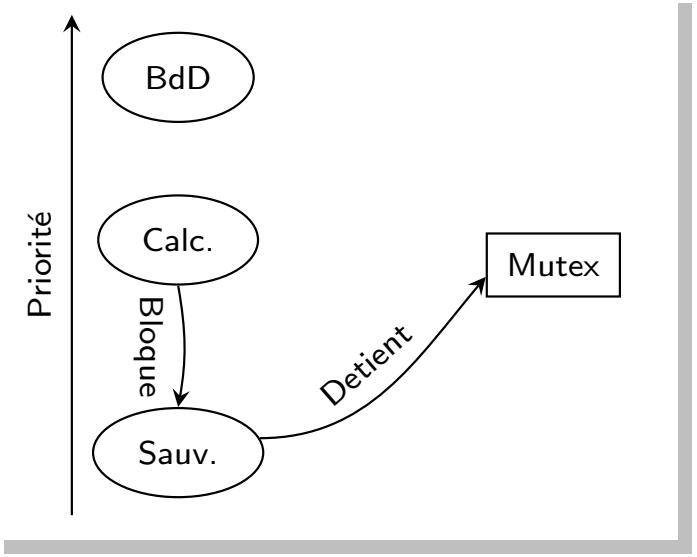
Supposons 2 processus, une base de donnée et sa sauvegarde, qui partagent une zone protégée par un *mutex*.



- Comme la base a un rôle important, sa priorité est élevée.
- La sauvegarde a une priorité faible.
- À un instant donné la sauvegarde acquiert le *mutex*.
- Si une tâche de calcul de priorité moyenne arrive à ce moment.

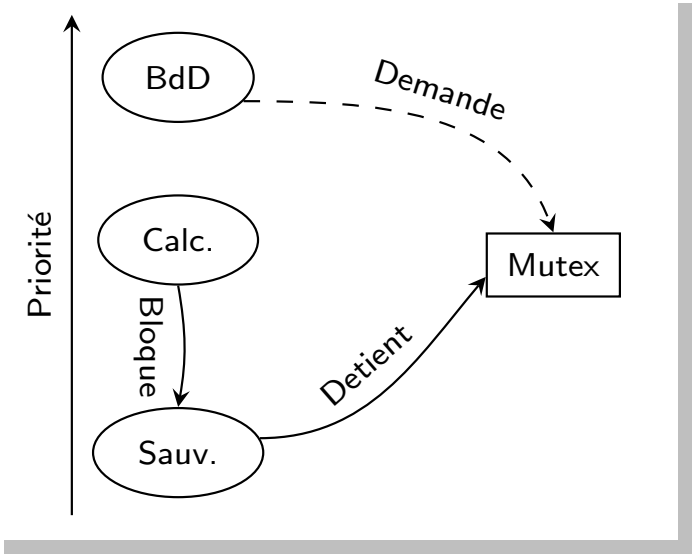
Inversion de priorité - I

Supposons 2 processus, une base de donnée et sa sauvegarde, qui partagent une zone protégée par un *mutex*.



- Comme la base a un rôle important, sa priorité est élevée.
- La sauvegarde a une priorité faible.
- À un instant donné la sauvegarde acquiert le *mutex*.
- Si une tâche de calcul de priorité moyenne arrive à ce moment.
- Le calcul bloque la sauvegarde (car elle est plus prioritaire).

Inversion de priorité - I

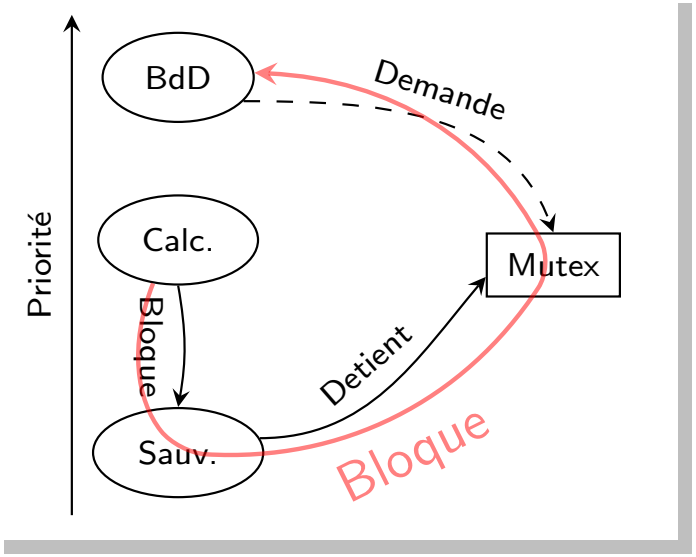


Supposons 2 processus, une base de donnée et sa sauvegarde, qui partagent une zone protégée par un *mutex*.

- Comme la base a un rôle important, sa priorité est élevée.
- La sauvegarde a une priorité faible.
- À un instant donné la sauvegarde acquiert le *mutex*.
- Si une tâche de calcul de priorité moyenne arrive à ce moment.
- Le calcul bloque la sauvegarde (car elle est plus prioritaire).
- La sauvegarde bloque la base de donnée car elle détient le mutex.

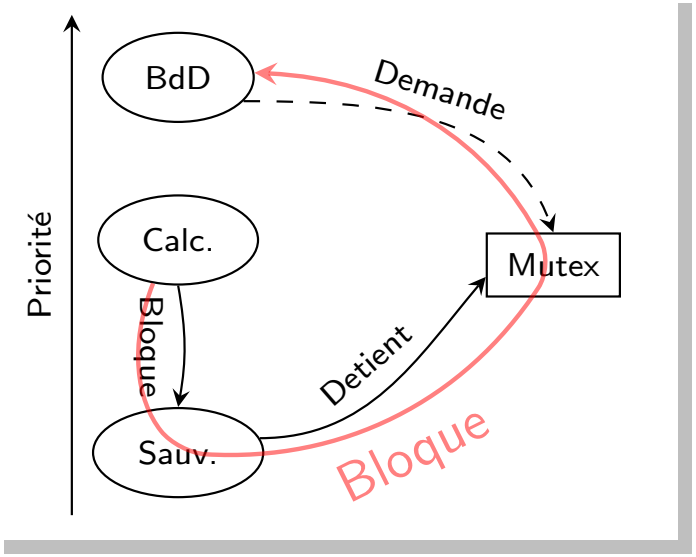
Inversion de priorité - I

Supposons 2 processus, une base de donnée et sa sauvegarde, qui partagent une zone protégée par un *mutex*.



- Comme la base a un rôle important, sa priorité est élevée.
- La sauvegarde a une priorité faible.
- À un instant donné la sauvegarde acquiert le *mutex*.
- Si une tâche de calcul de priorité moyenne arrive à ce moment.
- Le calcul bloque la sauvegarde (car elle est plus prioritaire).
- La sauvegarde bloque la base de donnée car elle détient le mutex.

Inversion de priorité - I



Supposons 2 processus, une base de donnée et sa sauvegarde, qui partagent une zone protégée par un *mutex*.

- Comme la base a un rôle important, sa priorité est élevée.
- La sauvegarde a une priorité faible.
- À un instant donné la sauvegarde acquiert le *mutex*.
- Si une tâche de calcul de priorité moyenne arrive à ce moment.
- Le calcul bloque la sauvegarde (car elle est plus prioritaire).
- La sauvegarde bloque la base de donnée car elle détient le mutex.

En pratique le calcul bloque la base de donnée qui est une tâche plus importante que lui.

Inversion de priorité - II

- Pour résoudre ce problème, certains systèmes utilisent *l'héritage de priorité*
  - Un processus qui bloque un processus plus prioritaire obtient sa priorité.
  - Ce n'est utilisé que dans des systèmes spécifiques (par ex. temps réel).
- Les autres utilisent des méthodes moins sûres :
  - Pour linux, l'évolution de la priorité fait en sorte que la tâche bloquante finira par être élue.
  - Pour windows, une tâche qui n'est pas élue pendant ≪ longtemps ≫ obtient la priorité maximale pendant 2 quantum de temps.

Linux

Le système linux ordonnance directement les threads (pas les processus). Il définit plusieurs classes de threads :

- 2 classes dites « temps réel »
  - Les FIFO (SCHED_FIFO)
  - Les Round Robin (SCHED_RR)

  qui ne sont pas réellement temps réel, car elles ne permettent pas d'obtenir de garanties sur les échéances.
- Les threads en temps partagé :
  - Les threads standards (SCHED_OTHER)
  - Les threads de calculs (SCHED_BATCH)
  - Les threads de tâches de fond (SCHED_IDLE)

Threads Temps Réel

- Ils sont toujours plus prioritaires que des threads en temps partagé.
- Ils ont une priorité statique comprise entre 0 et 99.
- Ils sont réservés au système.
- La différence entre les deux est :
  - Les threads FIFO ne peuvent être interrompus que par un thread plus prioritaire.
  - Les threads RR peuvent être interrompus par un thread plus prioritaire ou lorsque leur quantum de temps est atteint.

Threads en Temps Partagé - I

- Les threads normaux SCHED_OTHER :
  - Leur priorité statique est nulle (ils sont toujours plus lents que les threads temps réel)
  - Leur quantum de temps est plus faible.
  - Ils ont une priorité dynamique basée sur une valeur de *courtoisie* (*nice*) et qui évolue en fonction de leur utilisation du processeur
  - Le système applique un bonus aux threads qui bloquent (sur des entrées sortie) et un malus aux processus qui utilisent le processeur.
  - $\Rightarrow$ les processus interactifs sont favorisés.
  - En tout il y a 40 priorités pour les threads en temps partagé (donc 140 en tout)

Threads en Temps Partagé - II

- Les threads de traitments par lot SCHED_BATCH :
  - identiques aux précédents,
  - mais considérés forcément comme threads utilisant le processeur, ils sont pénalisés.
- Les tâches de fond
  - sans priorité statique ni courtoisie ;
  - ordonnancées en dernier/

Windows

- Le système propose 32 priorités
  - de 0 à 15 pour les threads utilisateurs
  - de 16 à 31 pour le système (Temps Réel)
- Entre les threads de même priorité, le système utilise *Round Robin*
- La priorité se calcule en 2 parties :
  - Priorité de base du processus : IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, REALTIME_PRIORITY_CLASS.
  - La priorité du thread dans le processus : THREAD_PRIORITY_IDLE, THREAD_PRIORITY_LOWEST, THREAD_PRIORITY_BELOW_NORMAL, THREAD_PRIORITY_NORMAL, THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_HIGHEST, THREAD_PRIORITY_TIME_CRITICAL,

Résumé

		Processus					
		Idle	Below n.	Normal	Above n.	High	R. T.
Threads	Idle	1	1	1	1	1	16
	Lowest	2	4	6	8	11	22
	Below n.	3	5	7	9	12	23
	Normal	4	6	8	10	13	24
	Above n.	5	7	9	11	14	25
	Highest	6	8	10	12	15	26
	Time Crit.	15	15	15	15	15	31

En plus

- Des bonus/malus
  - bonus à tout processus dont la fenêtre arrive au premier plan,
  - bonus pour ceux qui se réveillent suite à une entrée-sortie,
  - malus pour un thread qui finit sont quantum de temps,
  - bonus pour un thread qui n'a pas eu la main pendant une période ≪ trop longue ≫.
- Des priorités limites
  - THREAD_PRIORITY_IDLE : 16 pour les temps réels et 1 pour les normaux
  - THREAD_PRIORITY_TIME_CRITICAL : 31 pour les temps réels et 15 pour les normaux
- Des threads particuliers de priorité 0 pour remplir de 0 la mémoire (sécurité).

Cas des multiprocesseurs

Dorénavant, les ordinateurs ont plusieurs cœurs de calculs. Cela permet d'exécuter plusieurs tâches à la fois mais pose de nouveaux problèmes d'ordonnancement.

- Les caches font que l'accès aux données n'est pas identique pour tous les processeurs.
- Deux tâches échangeant des données sur des processeurs différents vont générer des défauts de caches.
- Dans les modèles avec plusieurs processeurs, il y a des cas où chaque processeur à un banc de mémoire dédié (NUMA).
- Un processeur ayant besoin d'avoir accès à une mémoire sur un autre banc doit faire appel à la MMU de l'autre processeur.

Si l'ordonnanceur ne tient pas compte de cela, le resultat ne sera pas efficace.

Par exemple, pour un hyperviseur, il est préférable que les tâches d'une même machine virtuelle soit ordonnancées sur le même processeur : celui qui dispose de sa mémoire.

Isolations/jails

Les systèmes permettent maintenant de regrouper des processus pour les gérer ensembles et les isoler. Par exemple, sous linux les *cgroups* permettent de :

- limiter leur utiliation de ressources, arbitrer entre les différents groupes ;
- stopper, sauvegarder et redémarrer des groupes ;
- isoler les processus via des *espaces de nommage* :
  - identifiant de processus (PID), ces processus ne voient que ceux du même groupe ;
  - interface réseaux, IPC : ils ne voient que les outils de communications affectés au groupes ;
  - systèmes de fichiers : ils peuvent avoir leur propre point de montage ;
  - utilisateurs ;
  - ...

Virtualisation au niveau du système

L'isolation autorise à créer un système proche de la virtualisation :

- des systèmes quasi séparés ;
- mais qui partagent le même noyau ;
- demandent beaucoup moins de ressources ;
- sous linux se sont les conteneurs, par exemple LXC, DOCKER.

Cela permet :

- d'isoler des applications ;
- de faire cohabiter plusieurs versions d'une même librairie ;
- d'implémenter des microservices.

Conclusion

- Le but de l'ordonnancement est de choisir la tâche qui va s'exécuter.
- Un mauvais ordonnancement peut être catastrophique.
- Sur les systèmes simples :
  - Souvent basé sur une priorité qui évolue au cours du temps.
  - 2 classes de processus : système (RT) et utilisateur.
  - Utilisation du *Round Robin*
- Système plus complexes :
  - Temps réel (besoin de garantie)
  - Automatisme (contraintes)

À retenir

- Différence entre préemptif et colaboratif
- Algorithme d'ordonnancement
  - FIFO
  - Round Robin
  - Priorité
- Évolution de la priorité dans le temps
- Inversion de priorité