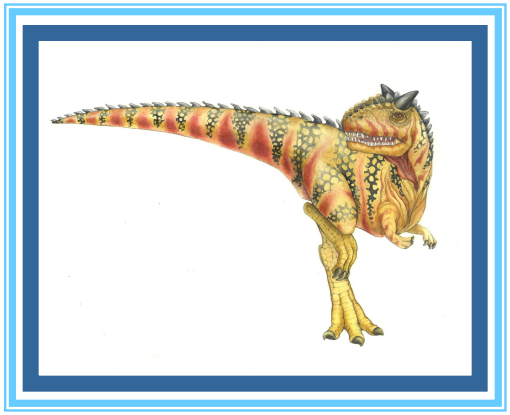
Chapter 5: Ordonnancement - Scheduling



Plan

- Concepts de base
- Critères d'ordonnancement
- Algorithmes d'ordonnancement
- Ordonnancement des threads
- Ordonnancement multiprocesseur
- Ordonnancement CPU temps réel

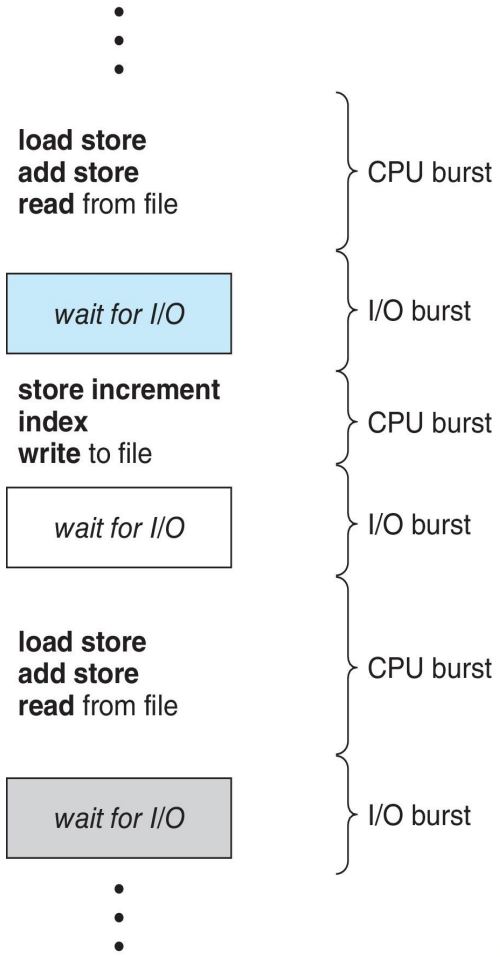
Objectives

- Les différentes stratégies d'ordonnancement
- Comparaison : équité, attente, efficacité
- Problèmes avec multiprocesseur et multicœur
- Ordonnancement temps réel

Concepts de base

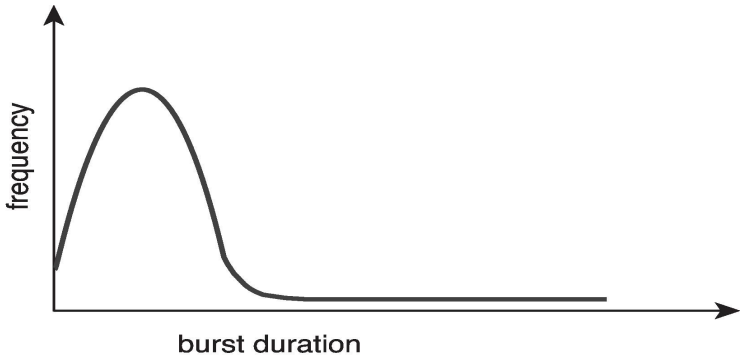
- Chaque fois, que le processeur devient inactif, le système d'exploitation **doit sélectionner** un processus de **la file d'attente des processus prêts**, et lui passe le contrôle.
  - L'ordonnancement (schedule) est le contrôle de l'ordre d'exécution de plusieurs processus par le système d'exploitation. Il est réalisé par un composant du noyau appelé ordonnanceur

- Utilisation optimale du CPU et de cycle d'éxécution :
  - Maximiser l'utilisation du CPU grâce à la multiprogrammation.
    - Cycle CPU–I/O : un processus alterne entre exécution sur le CPU et attente d'I/O.
    - Chaque burst CPU (période d'exécution) est suivi d'un burst I/O (attente d'entrée/sortie).
  - La distribution des bursts CPU est un facteur clé pour optimiser la planification du processeur.



Concepts de base

- **Distribution des Bursts CPU**
  - Beaucoup de courtes exécutions
  - Peu de longues exécutions



- **L'allocation du processeur**

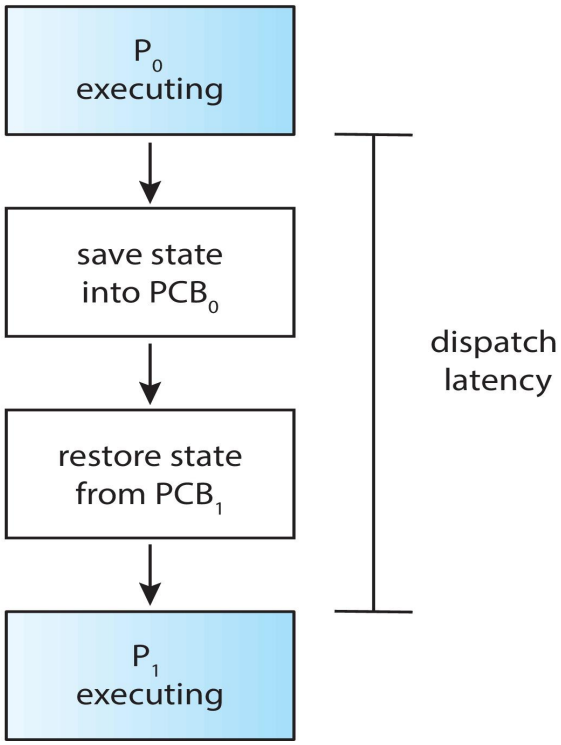
  La tâche d'ordonnancement et d'allocation du processeur est prise en charge par deux routines système:
  - **L'Ordonnanceur (Scheduleur)**
  - **Le Dispatcheur**

L'Ordonnanceur des Processus (scheduleur)

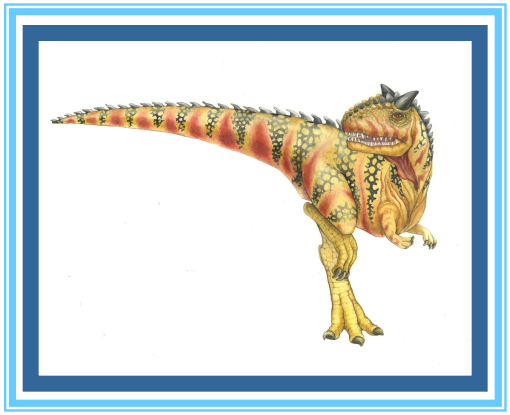
- Les processus sont placés dans une file d'attente avant d'être exécutés, tour à tour, selon la gestion spécifique de **l'ordonnanceur**.
- L'ordonnanceur gère la file d'attente des processus, l'arrivée des demandes et les place dans la file.
- **L'ordonnanceur** choisit un processus dans la **file d'attente des prêts** et lui attribue à un cœur de CPU.
- La file peut être ordonnée de différentes manières.
- Moments où l'**ordonnanceur** intervient :
  1. Passage de l'état d'exécution à l'état d'attente.
  2. Passage de l'état d'exécution à l'état prêt.
  3. Passage de l'état d'attente à l'état prêt.
  4. Fin du processus.
- Pour les cas 1 et 4, il n'y a pas de choix, un nouveau processus doit être sélectionné.
- Pour les cas 2 et 3, le système peut choisir quel processus exécuter ensuite.

Le Dispatcheur

- Entre en jeu après l'ordonnanceur,
- Le **Dispatcheur** est le module qui s'occupe de l'allocation du processeur à un processus sélectionné par l'ordonnanceur du processeur.
- Cela implique :
  - **Changement de contexte** : (sauvegarde/restauration de l'état du processus).
  - **Passage en mode utilisateur**.
  - **Reprise de l'exécution** au bon endroit dans le programme utilisateur.
- ***Latence de dispatch*** *: temps nécessaire pour arrêter un processus et en démarrer un autre.*



Chapter 5 : Les Algorithmes d'ordonnancement



Algorithmes d'ordonnancement

- L'ordonnancement se produit chaque fois que **deux processus ou plus sont en état prêt** en même temps. Il faut choisir quel processus va passer en exécution.

- La partie du système d'exploitation qui effectue ce choix se nomme **l'ordonnanceur** (**scheduler**). L'algorithme qu'il emploie s'appelle **algorithme d'ordonnancement**.

- Un ordonnanceur applique des **critères de sélection** pour sélectionner un processus pour l'exécuter.
  - **Temps d'arrivée** (temps de création du processus) ;
  - **Durée de traitement** (le temps que va prendre le processus pour être traité) ;
  - **La priorité du processus.**

Algorithmes d'ordonnancement

- Il existe différentes politiques d'ordonnancement :
  - **Sans priorité**
  - **Avec priorité**: à chaque processus est attribuée une priorité
  - **Non préemptif**: un processus élu à être exécuté **ne libère le processeur que s'il termine toute son exécution** ou bien est **bloqué** pour manque de ressource.
  - **Préemptif**: un processus en cours d'exécution peut être retiré avant d'avoir terminé son exécution. Utilisée par Windows, macOS, Linux et UNIX.

Ordonnancement Préemptif et Conditions de Course

- L'ordonnancement préemptif peut entraîner des **conditions de course (Race conditions)** lorsque des données sont partagées entre plusieurs processus.

- Prenons le cas de deux processus partageant des données. Pendant qu'un processus **met à jour les données**, il est **préempté** pour permettre au second processus de s'exécuter. Ce dernier tente alors de lire les données, qui se trouvent dans un état **incomplètes ou incohérentes.**

Note *: "Race conditions" est souvent traduit par "conditions de course", bien que certains préfèrent "situations de compétition".*

Critères d'ordonnancement

- Utilisation du CPU : Garder le processeur actif le plus possible. (Max CPU utilization)
- Débit (Throughput) : Nombre de processus terminés par unité de temps. (Max throughput)
- Temps de retour (Turnaround Time) : durée totale pour finir un processus (Min turnaround time)
- Temps d'attente : temps passé en file avant l'exécution (Min waiting time)
- Temps de réponse : Temps écoulé entre la soumission d'une demande et la première réponse. (Min response time)

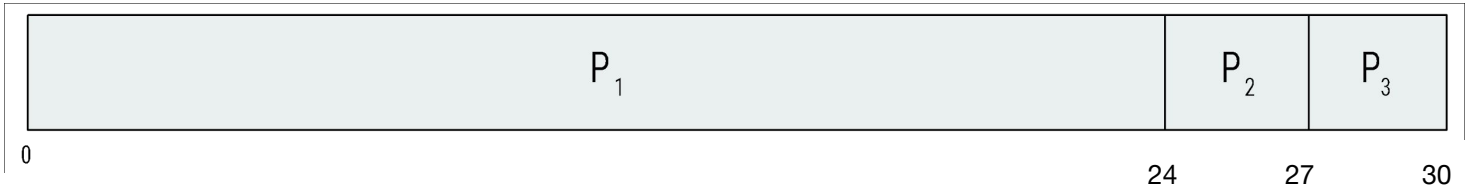
- Critères d'Optimisation des Algorithmes d'ordonnancement
  - **Maximiser** l'utilisation du CPU et le débit (nombre de processus terminés).
  - **Minimiser** le temps de retour (Délai d'exécution), d'attente et de réponse.
  - **Trouver un équilibre** entre **efficacité** et **équité**.

First- Come, First-Served (FCFS) Scheduling
premier arrivé, premier servi

FCFS (First Come First Served) ou **FIFO (First In First Out)**

Processus	Burst Time (Durée d'exécution)
P_1	24
P_2	3
P_3	3

- Supposons que les processus arrivent dans l'ordre : P_1 , P_2 , P_3
- Le diagramme de Gantt pour l'ordonnanceur est le suivant :



- Temps d'attente pour $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Temps d'attente moyen : $(0 + 24 + 27)/3 = 17$

***NB**. Le diagramme de GANTT illustre l'évolution du traitement des différents processus au cours du temps.*

First- Come, First-Served (FCFS) Scheduling
premier arrivé, premier servi (Cont.)

- Supposons que les processus arrivent dans l'ordre suivant : P_2 , P_3 , P_1
- Le diagramme de Gantt devient :

- Temps d'attente pour $P_1 = 6; P_2 = 0, P_3 = 3$
- Temps d'attente moyen : $(6 + 0 + 3)/3 = 3$ (**Bien meilleur que dans le cas précédent**)

Effet de Convoi : Processus court derrière un processus long

Un processus lié au CPU (**CPU-bound**) peut être ralenti par plusieurs processus lié au E/S (**I/O-bound**) qui prennent du temps pour les opérations d'entrée/sortie.

➔ Cela conduit à une **mauvaise utilisation du CPU** et une baisse de l'efficacité globale.

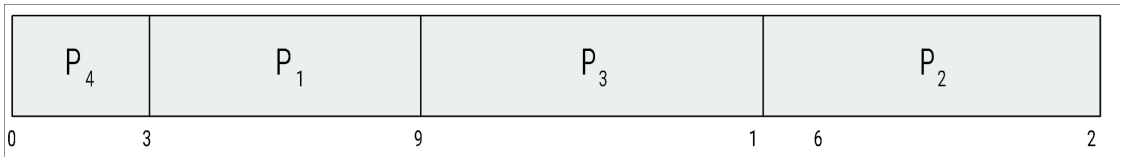
Shortest-Job-First (SJF) Scheduling
Le travail le plus court d'abord

- Associer à chaque processus la durée de son burst CPU
  - Utiliser ces durées pour planifier le processus ayant le temps d'exécution le plus court.
  - SJF est optimal – il minimise le temps d'attente moyen pour un ensemble donné de processus.
  - La version préemptive s'appelle shortest-remaining-time-first.

- **Exemple**

Process	*Burst Time*
P_1	6
P_2	8
P_3	7
P_4	3

  - **Diagramme de SJF :**



  - **Temps d'attente pour P1 = 3; P2 = 16; P3 = 9; P4 = 0**
  - **Temps d'attente moyen = (3 + 16 + 9 + 0) / 4 = 7**

- **NB.** Comment déterminer la durée du prochain burst CPU **?**
  - On pourrait demander à l'utilisateur
  - On peut estimer la durée à partir des bursts précédents.

Détermination de la durée du prochain burst CPU

- On peut estimer la durée : elle devrait être proche de la précédente.
- Cela se fait en utilisant les durées précédentes, avec une moyenne exponentielle.
  - t_n = est la durée réelle du *n-ième* burst CPU
  - τ_{n+1} = est la valeur prédite pour le prochain burst CPU
  - α , $0 \leq \alpha \leq 1$
  - Définir $t_{n+1} = \alpha\, t_n + (1-\alpha)\, \tau_n$
- En général, α est fixé à ½.

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - L'historique récent n'est pas pris en compte.
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Seul le dernier burst CPU réel est pris en compte.

- Si l'on développe la formule, on obtient :
  - $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\alpha\, t_{n-1} + \ldots + (1-\alpha)^j \alpha\, t_{n-j} + \ldots + (1-\alpha)^{n+1} \tau_0$
  - Comme α et $(1-\alpha)$ sont inférieurs ou égaux à 1, chaque terme suivant a moins de poids que le précédent.

Shortest Remaining Time First Scheduling (SRTF)
Ordonnancement par Temps Restant le Plus Court

- Version **préemptive du SJN**
- Ordonnancement SJF **Préemptif**
- Chaque fois qu'un nouveau processus arrive dans la file d'attente prête, la décision sur quel processus planifier ensuite est recalculée en utilisant l'algorithme **SJN (Shortest Job Next)**.

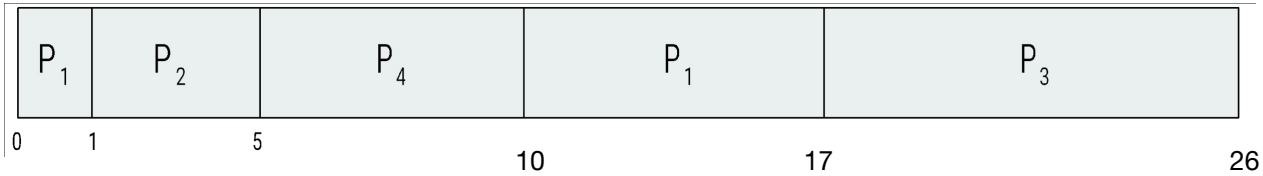
- Le **SRT** (Shortest Remaining Time) **est-il plus « optimal** » que le **SJN** en termes de temps d'attente moyen minimal pour un ensemble donné de processus .

Exemple d'ordonnancement par SRTF

- Nous ajoutons maintenant les concepts d'heures d'arrivée variables et de préemption à l'analyse

Process	*Arrival Time*	*Burst Time*
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

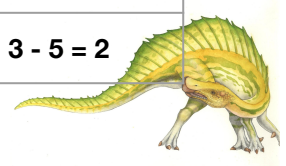
- Diagramme de Gantt SJF préventif



- **Principe du SRTF :**
  - Priorité au processus avec le temps restant le plus court (préemption si un nouveau processus plus court arrive).
  - **Calcul du temps d'attente = Temps d'achèvement – Temps d'arrivée – Durée d'exécution (WT = CT – AT – BT)**

Process	Arrival Time	Burst Time	Completion Time	**Waiting Time (WT)**
P1	0	8	17	**17 - 0 - 8 = 9**
P2	1	4	5	**5 - 1 - 4 = 0**
P3	2	9	26	**26 - 2 - 9 = 15**
P4	3	5	10	**10 - 3 - 5 = 2**

- **Temps d'attente moyen = [9+0+15+2]/4 = 26/4 = 6.5**



Principe de l'ordonnancement par quantum Round Robin - RR

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large ⇒ FIFO (FCFS)
  - *q* small ⇒ RR
- Note that q must be large with respect to context switch, otherwise overhead is too high

Principe de l'ordonnancement par quantum Round Robin - RR

- Chaque processus reçoit une petite unité de temps CPU (quantum, noté ***q***), généralement entre 10 et 100 millisecondes. Une fois ce temps écoulé, le processus est préempté et replacé en fin de file d'attente prête.

- Règles clés :
  - Si la file d'attente contient ***n*** processus et que le quantum est ***q***, chaque processus reçoit ***1/n*** du temps CPU, par blocs d'au plus ***q* unités**.
  - Aucun processus n'attend plus de **(n–1) × q** unités de temps avant de s'exécuter à nouveau.
  - Une interruption timer survient à chaque quantum pour déclencher l'ordonnancement du processus suivant.

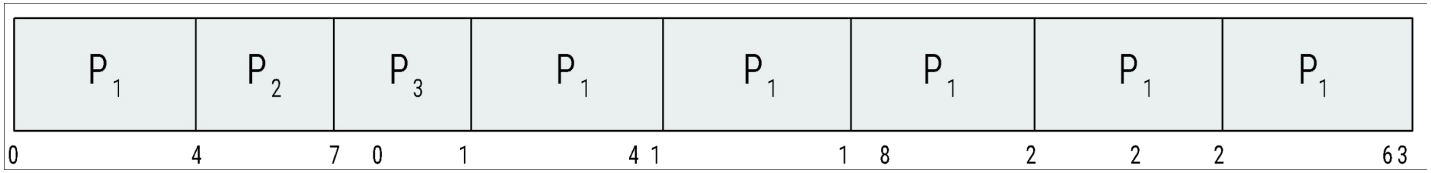
- Performance :
  - ***q* grand** ⇒ Comportement similaire à **FIFO** (First-Come, First-Served).
  - ***q* petit** ⇒ Approche pure de Round Robin (**RR**).

- Remarque : ***q*** doit être **largement supérieur** au **temps de changement de contexte**, sous peine de surcharge excessive (overhead).
- **Préemption** : Mécanisme qui interrompt un processus après ***q*** unités de temps pour assurer l'équité.
- **Overhead** : Si ***q*** est trop petit, le système passe plus de temps à gérer les changements de contexte qu'à exécuter les processus.
- **FIFO vs RR :**
  - $q \to \infty$ ⇒ FCFS (aucune préemption).
  - $q \to 0$ ⇒ RR idéal (mais irréaliste en pratique).

Exemple de RR avec Quantum de Temps = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Le diagramme de Gantt est :



- **Temps moyen de rotation (turnaround) généralement plus élevé** qu'avec SJF (Shortest Job First), mais **meilleure réactivité** (temps de réponse plus court).
- Le **quantum q doit être nettement supérieur** au temps de changement de contexte (context switch).
  - **q** est typiquement **entre 10 et 100 millisecondes**.
  - Le **temps de changement de contexte** est généralement **inférieur à 10 microsecondes.**

Ordonnancement avec priorités

- Un nombre entier (priorité) est associé à chaque processus.
- Le CPU est alloué au processus ayant la priorité la plus élevée (le plus petit entier ≡ priorité la plus haute).
  - **Préemptif** : Le processus en cours peut être interrompu si un processus de priorité supérieure arrive.
  - **Non préemptif** : Le processus conserve le CPU jusqu'à la fin de son exécution, même si un processus plus prioritaire arrive.

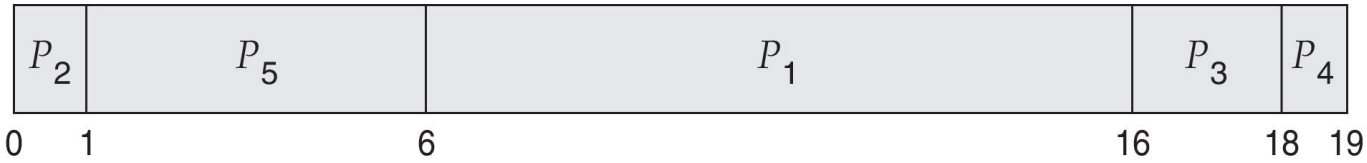
- **SJF est un cas particulier d'ordonnancement par priorité**, où la priorité est l'inverse du temps CPU restant estimé (plus le burst time est petit, plus la priorité est élevée).

- **Problème ≡ Famine** (**Starvation**) – Les processus de faible priorité peuvent ne jamais s'exécuter si des processus prioritaires arrivent en continu.
- **Solution ≡ Vieillissement** (**Aging**) – Augmenter progressivement la priorité des processus qui attendent depuis longtemps (ex : incrémenter leur priorité après chaque quantum d'attente).

Exemple d'ordonnancement avec priorités

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Le diagramme de Gantt est :



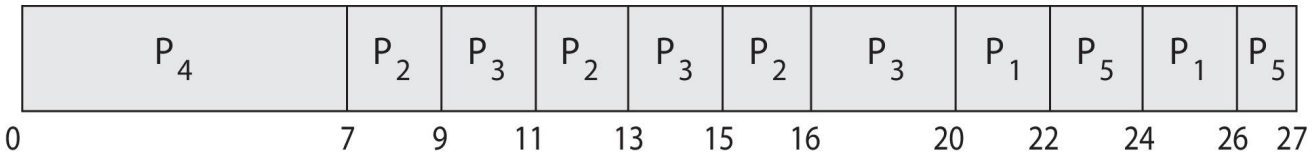
- Temps d'attente moyen = (6+0+16+18+1) / 5 = 8.2

Ordonnancement par Priorité avec Round-Robin

- Exécuter le processus avec la priorité la plus élevée. Les processus de même priorité s'exécutent en Round-Robin (tourniquet)
- Example:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Le diagramme de Gantt est, quantum = 2 :



Ordonnanceur à files multiniveaux
Multilevel Queue

- La file de processus prêts (**ready queue**) est divisée en **plusieurs sous-files distinctes**, chacune ayant :
  - Sa propre priorité (ex : système > interactif > batch).
  - Son algorithme d'ordonnancement dédié (ex : RR pour les tâches interactives, FCFS pour les batchs)

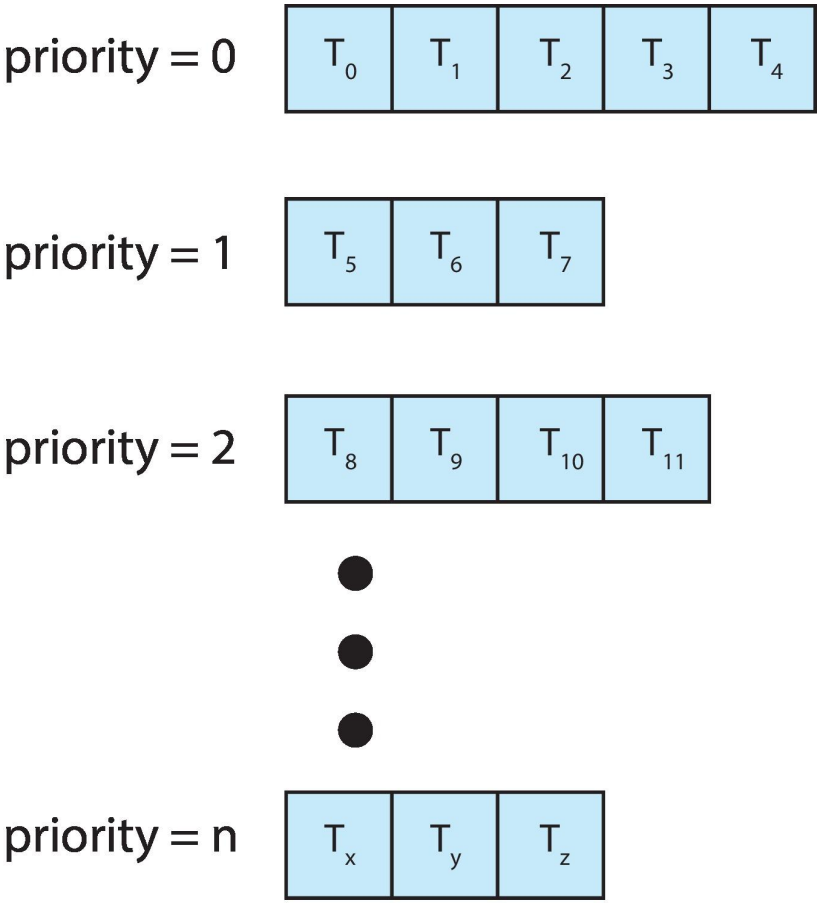
- **Exemple typique :**

Niveau	Type de processus	Ordonnancement utilisé
0	**Temps réel** (urgent)	Priorité (Préemptif)
1	**Interactif** (utilisateur)	Round-Robin (q=10 ms)
2	**Batch** (arrière-plan)	FCFS (non préemptif)

- **Paramètres de configuration :**
  - **Nombre de files** : Définit les niveaux de priorité (ex : temps réel, interactif, batch).
  - **Algorithme par file** : Chaque file peut avoir son propre algorithme :
    - RR pour les tâches interactives.
    - FCFS pour les traitements batch.
    - Priorité stricte pour les processus critiques.
  - **Méthode d'assignation :** Détermine la file d'un processus selon :
    - Son type (utilisateur, système…).
    - Sa priorité (fixe ou dynamique via aging).
  - **Ordonnancement entre files** :
    - Priorité stricte : une file doit se vider avant de passer à la suivante.
    - Partage de temps : répartition du CPU (ex : 70% vs 30%)

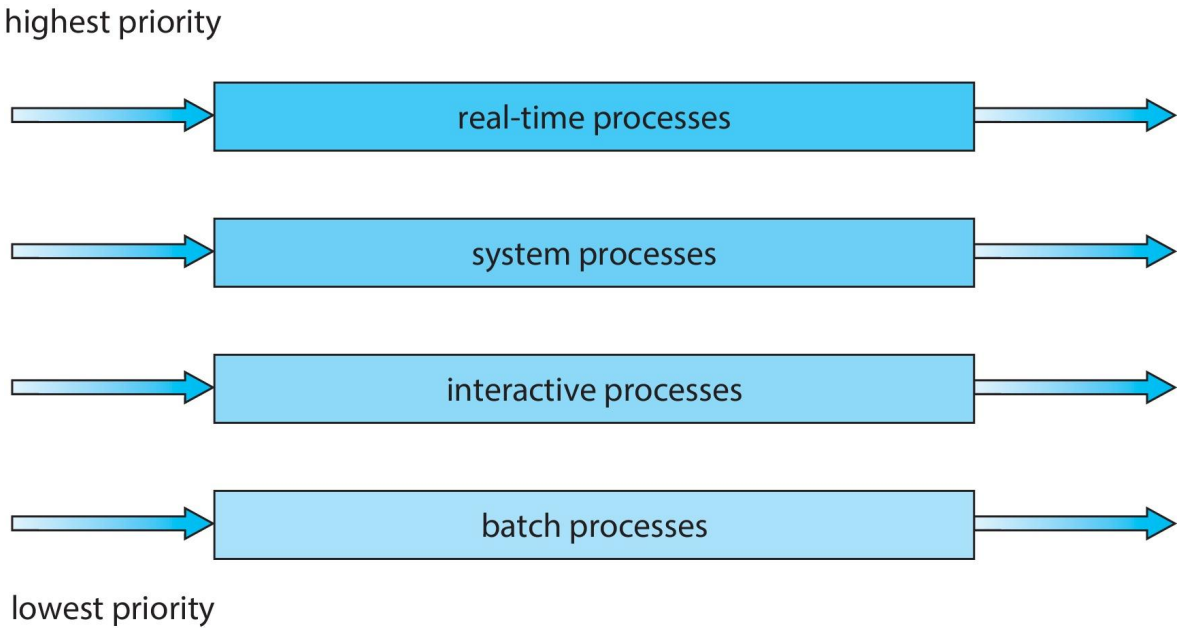
Multilevel Queue

- Pour l'ordonnancement par priorité, on utilise des files séparées pour chaque niveau de priorité : **on planifie toujours le processus dans la file ayant la plus haute priorité**



Multilevel Queue

- Priorisation basée sur le type de processus.

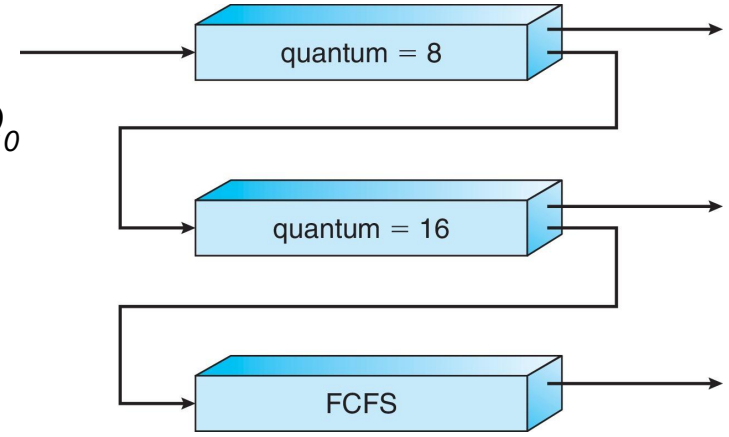


File à plusieurs niveaux avec rétroaction
Multilevel Feedback Queue

- Dans ce système - **File à plusieurs niveaux avec rétroaction (Multilevel Feedback Queue)**, un processus peut changer de file selon son comportement.
- Voici les éléments qui définissent ce type d'ordonnanceur :
  - Le nombre de files (niveaux de priorité).
  - L'algorithme utilisé dans chaque file (ex : Round-Robin, FCFS…).
  - Quand un processus est promu vers une file plus prioritaire.
  - Quand un processus est rétrogradé vers une file moins prioritaire.
  - Comment on choisit la file d'un nouveau processus qui demande du service.
- Ce type de file permet aussi de gérer le vieillissement (aging) pour éviter que certains processus n'attendent trop longtemps.

Exemple de file à plusieurs niveaux avec rétroaction

- Trois fils :
  - Q_0 – RR avec quantum 8 milliseconds
  - Q_1 – RR avec quantum 16 milliseconds
  - Q_2 – FCFS
- Ordonnancement
  - Un nouveau processus entre dans la file Q_0 où il est exécuté selon le Round-Robin.
  - Lorsqu'il obtient le CPU, il reçoit 8 millisecondes de temps.
  - S'il ne termine pas dans ce délai, il est déplacé vers la file Q_1.
  - Dans Q_1, il est à nouveau exécuté en Round-Robin et reçoit 16 millisecondes.
  - S'il n'est toujours pas terminé, il est préempté et déplacé vers la file Q_2, où il sera exécuté selon le principe FCFS jusqu'à la fin.



Ordonnancement des threads

- Il existe une différence entre les threads au niveau utilisateur et ceux au niveau noyau.
- Lorsque les threads sont pris en charge, ce sont les threads, et non les processus, qui sont ordonnancés.
- Modèles d'exécution :
  - Dans les modèles many-to-one et many-to-many, c'est la bibliothèque de threads qui décide quel thread utilisateur s'exécute sur un LWP (Lightweight Process).
  - Ce mécanisme est appelé portée de contention au niveau du processus (PCS - Process-Contention Scope), car les threads se font concurrence à l'intérieur d'un même processus.
  - L'ordonnancement se fait souvent selon une priorité définie par le programmeur.
- Ordonnancement au niveau système :
  - Les threads noyau sont, eux, ordonnancés directement par le système sur un CPU disponible.
  - Cela s'appelle la portée de contention au niveau du système (SCS - System-Contention Scope) : ici, tous les threads du système sont en concurrence pour accéder aux processeurs.

API Pthread

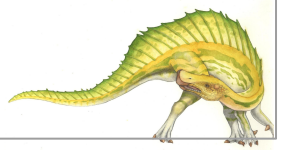
- L'API permet de spécifier **PCS** ou **SCS** lors de la création d'un thread.
  - **PTHREAD_SCOPE_PROCESS** : utilise l'ordonnancement **PCS** (portée de contention au niveau du processus).
  - **PTHREAD_SCOPE_SYSTEM** : utilise l'ordonnancement **SCS** (portée de contention au niveau du système).
- Cependant, cela , peut être limité par le système d'exploitation – uniquement Linux et macOS, permet PTHREAD_SCOPE_SYSTEM

Pthread API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling
scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

```
   /* set the scheduling algorithm to
PCS or SCS */
   pthread_attr_setscope(&attr,
PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)

pthread_create(&tid[i],&attr,runner,NULL
);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in
this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

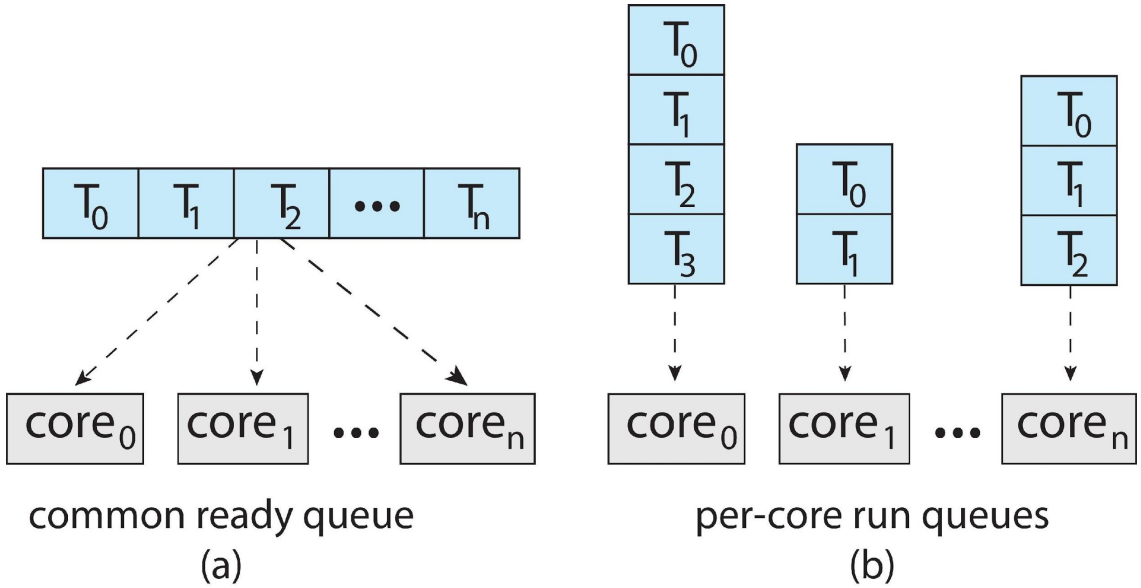


Ordonnancement avec plusieurs processeurs Multiple-Processor

- L'ordonnancement devient plus complexe quand il y a plusieurs CPU.
- On peut avoir différents types d'architectures :
  - Processeurs multicœurs (plusieurs cœurs dans un seul CPU)
  - Cœurs multithreadés (chaque cœur peut exécuter plusieurs threads)
  - Systèmes NUMA (Non-Uniform Memory Access) accès mémoire non uniforme
  - Multiprocesseurs hétérogènes (CPU de types différents dans un même système)

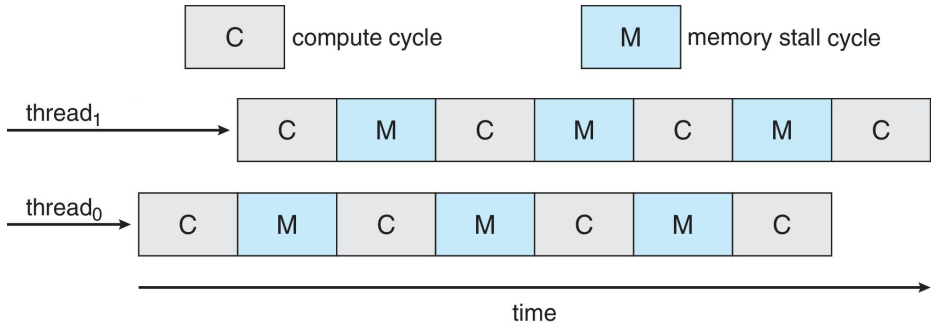
Ordonnancement avec plusieurs processeurs
Multiple-Processor

- Dans un système SMP (Symmetric multiprocessing - symétrique multi-processeur), chaque processeur s'auto-planifie.
- Deux options sont possibles :
  - (a) Tous les threads partagent une même file d'attente (file globale).
  - (b) Chaque processeur a sa propre file privée de threads.



Processeurs Multicœurs

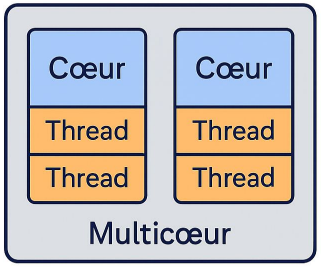
- Les processeurs multicœurs intègrent plusieurs unités de calcul (cœurs) sur une même puce, cela permet d'exécuter plusieurs tâches en parallèle avec une meilleure efficacité énergétique.
  - Un cœur peut gérer plusieurs threads à la fois.
  - Si un thread est bloqué (ex : en attente de données), un autre peut continuer à s'exécuter 👉 Cela évite les temps morts



- **Exemples d'architectures :**
  - Dual-core, Quad-core… (2, 4, 8+ cœurs).
  - Processeurs modernes (Intel Core i9, AMD Ryzen).

Processeurs multicœurs

- Plus rapide
  - Multiple táches peuven exécutée en paralèl
- Moins d'énergie
  - Comparaison a mùiples processeurs séparés
- Plusieurs threads par cœur
  - Profiter des temps d'attente (ex : en mmoire)

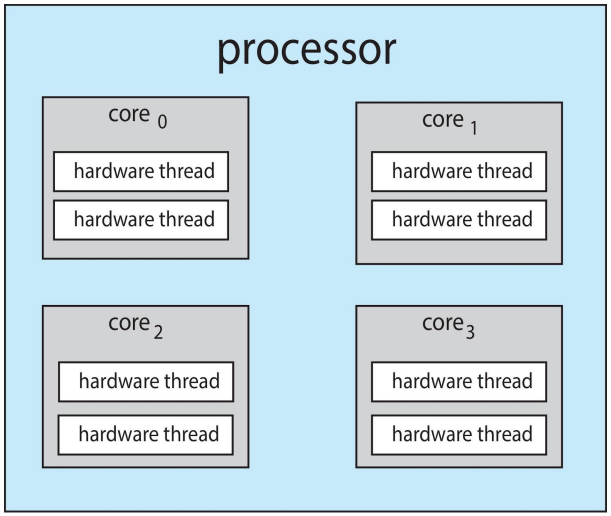


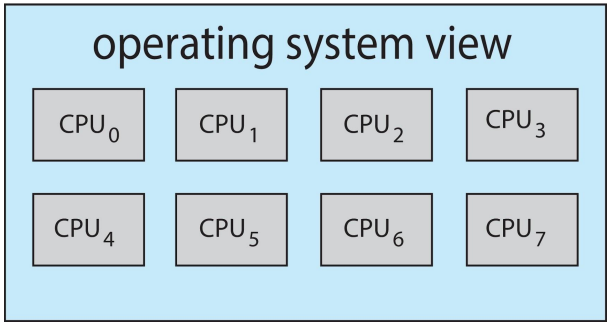
Plusieurs threads par cœur

Système multicœur multithreadé

- Le **Chip-Multithreading** (CMT) attribue à chaque cœur plusieurs threads matériels. (Intel appelle cela **Hyperthreading**.)

- ➡️ Par exemple, dans un système quadricœur avec 2 threads matériels par cœur, le système d'exploitation voit 8 processeurs logiques.



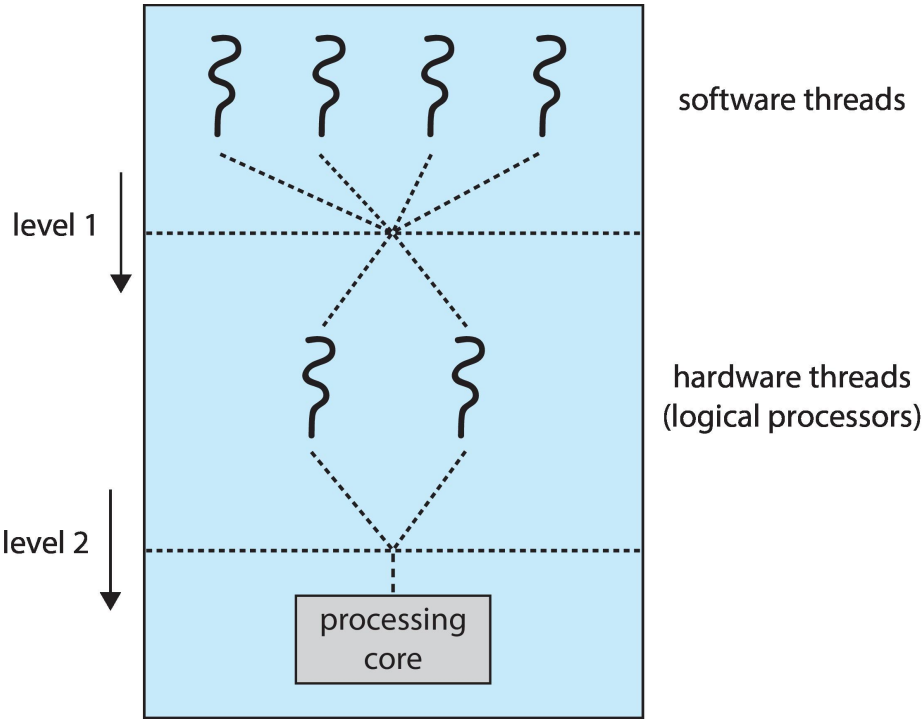


Système multicœur multithreadé

- Deux niveaux d'ordonnancement :

  1. Le système d'exploitation détermine quel thread logiciel exécuté sur un CPU logique.

  2. Chaque cœur physique décide quel thread matériel activer sur le cœur réel.



Ordonnancement Multiprocesseur
Équilibrage de Charge – Load Balancing

- En environnement SMP, il est crucial de maintenir tous les CPU actifs pour une efficacité optimale.
- L'équilibrage de charge Load balancing vise à répartir uniformément la charge de travail.
- Deux stratégies principales :
  - Push Migration (Migration Poussée) : Une tâche périodique vérifie la charge de chaque processeur. Si un CPU est surchargé, elle déplace une tâche vers un autre CPU moins occupé.
  - Pull Migration (Migration Tirée) : Un processeur inactif "**attire**" une tâche en attente depuis un processeur occupé.

Affinité Processeur et Équilibrage de Charge

- Lorsqu'un thread s'exécute sur un processeur, son accès à la mémoire est stocké dans le cache de ce processeur. On parle alors d'affinité du thread avec ce processeur ("processor affinity").
- **Problématique :**
  - L'équilibrage de charge peut perturber cette affinité en déplaçant un thread vers un autre processeur, ce qui entraîne une perte des données cache et impacte les performances.
- **Types d'Affinité :**
  - Affinité Douce (**Soft Affinity**) : L'OS tente de garder le thread sur le même processeur, sans garantie.
  - Affinité Forte (**Hard Affinity**) : Le processus peut définir explicitement les processeurs autorisés pour son exécution.

NUMA et Ordonnancement

Dans une architecture **NUMA** (Non-Uniform Memory Access), la mémoire est divisée en zones locales et distantes par rapport aux CPU.

Accès local = rapide, **accès distant** = lent (dû à la latence inter-nœuds).

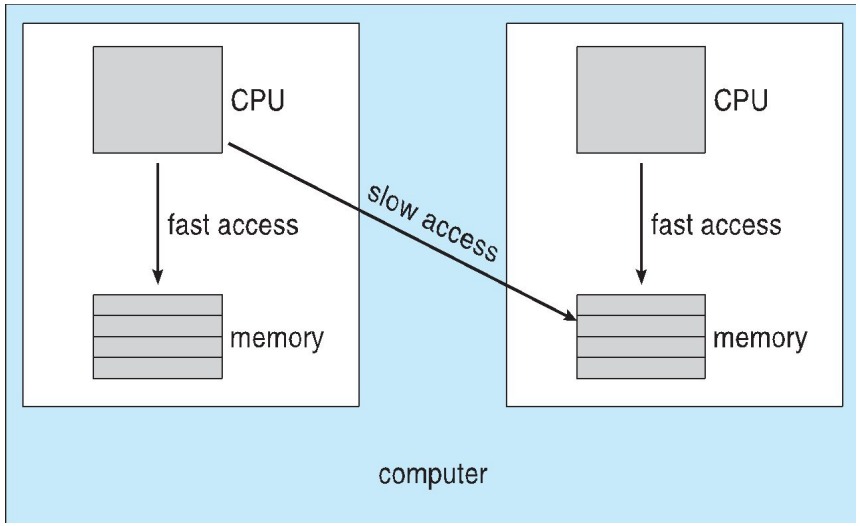
Rôle de l'OS :

Un OS NUMA-aware (comme Linux/Windows Server) optimise les allocations mémoire pour :

- ➔ Réduire les accès distants.
- ➔ Améliorer les performances en liant la mémoire au nœud NUMA du CPU actif.

Exemple concret :

- ➔ Sous Linux, la commande `numactl --preferred=node` force une application à utiliser la mémoire locale d'un nœud spécifique.



POSIX Real-Time

- Norme POSIX.1b
- L'API POSIX.1b fournit des fonctions pour gérer les threads temps réel.
- Elle définit deux classes d'ordonnancement pour ces threads :
  1. SCHED_FIFO (First In, First Out)
     - Ordonnancement par file FIFO (premier arrivé, premier servi).
     - Pas de partage de temps (time-slicing) pour les threads de même priorité.
  2. SCHED_RR (Round Robin)
     - Similaire à SCHED_FIFO, mais avec partage de temps (time-slicing) pour les threads de priorité égale.
- Elle définit deux fonctions pour obtenir et définir la politique de planification :
  1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
  2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

POSIX Real-Time API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr,
&policy) != 0)
      fprintf(stderr, "Unable to get
policy.\n");
   else {
      if (policy == SCHED_OTHER)
printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR)
printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO)
printf("SCHED_FIFO\n");
   }
   /* set the scheduling policy - FIFO, RR,
or OTHER */
   if (pthread_attr_setschedpolicy(&attr,
SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set
policy.\n");
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)

pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this
function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

End of Chapter 5