Ordonnancement

Menu

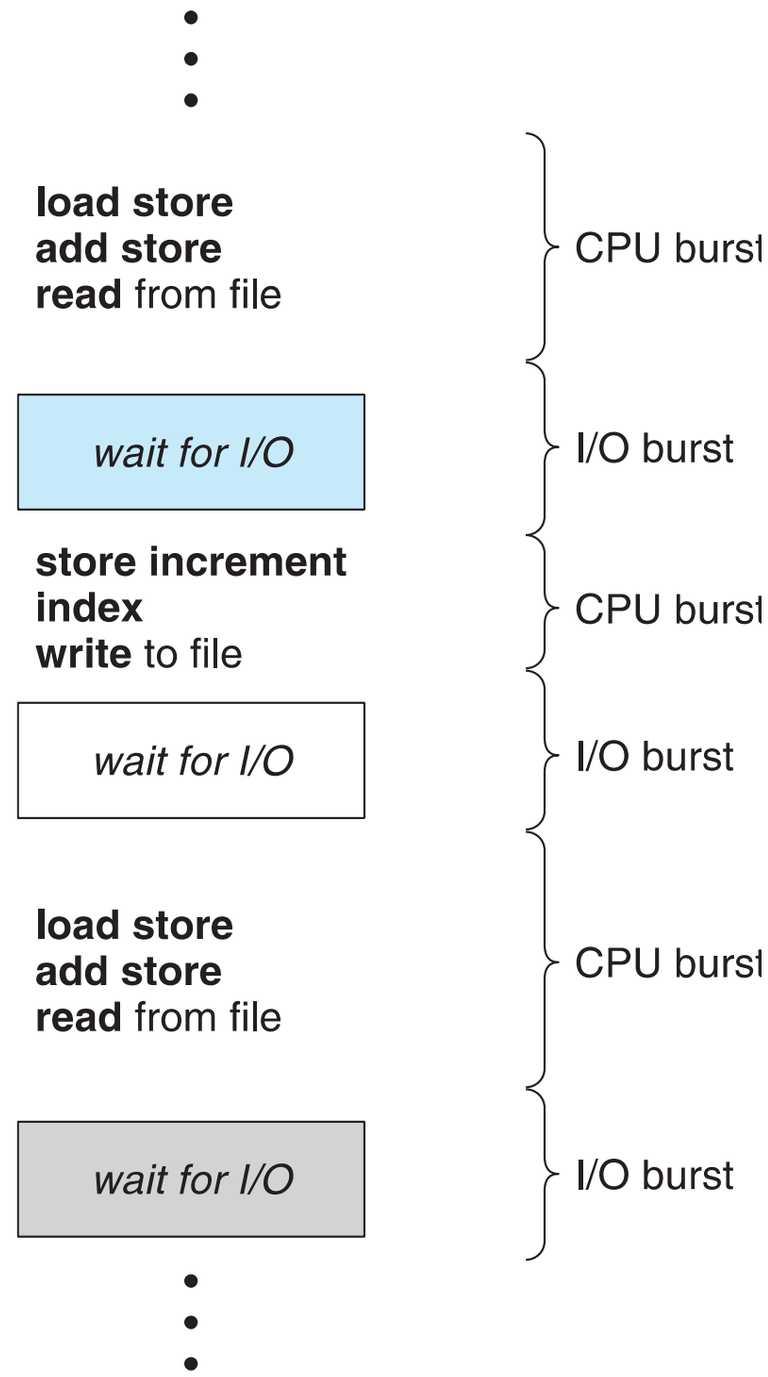
- Introduction
- Algorithmes d'ordonnancement
- Ordonnancement de threads
- Ordonnancement multi-processor

Menu

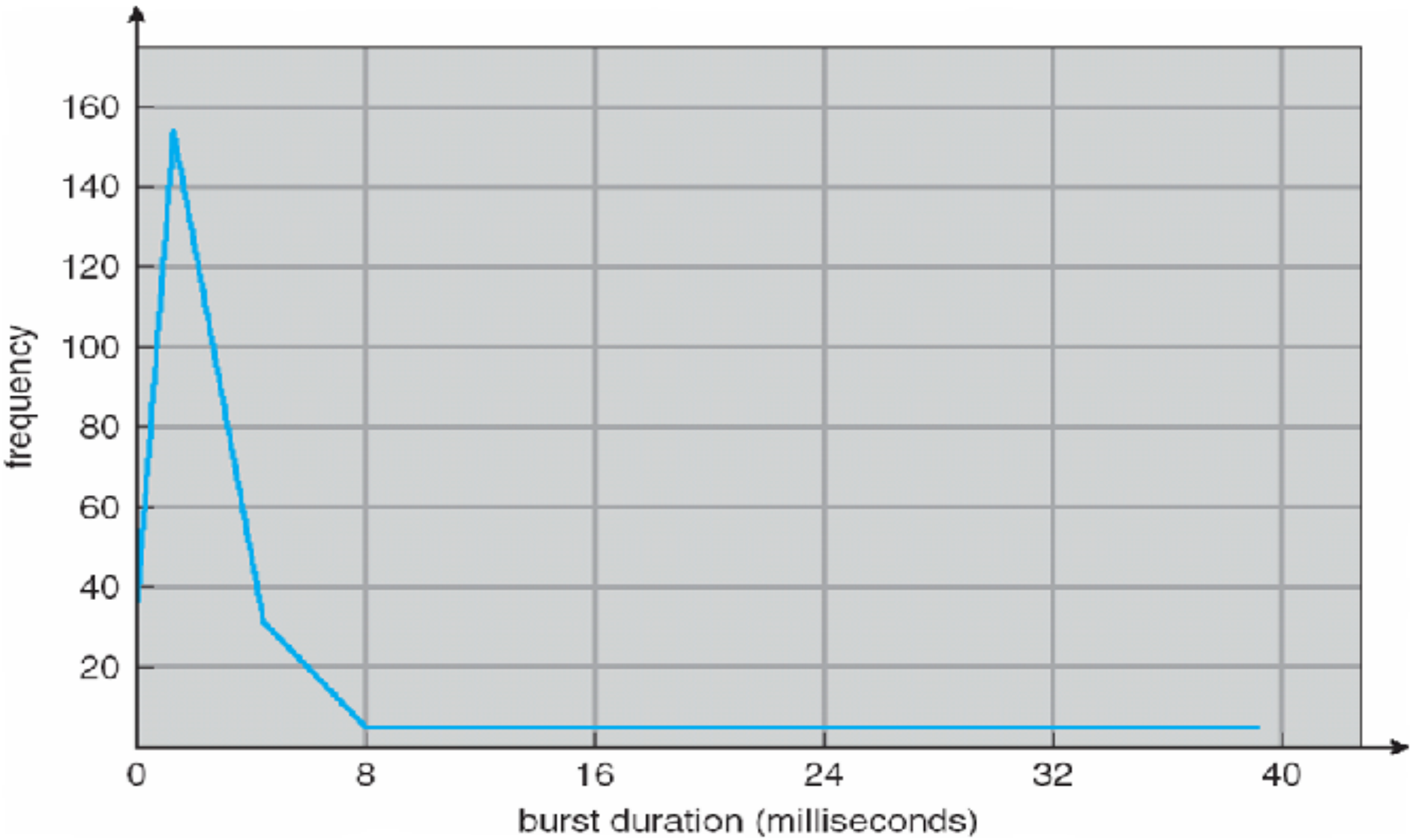
- **Introduction**
- Algorithmes d'ordonnancement
- Ordonnancement de threads
- Ordonnancement multi-processor

Concepts de base

- L'ordonnancement de processus permet de mieux utiliser le CPU avec la multiprogrammation
- Exécution découpée en *CPU burst* et *I/O burst*
- Préoccupation principale: distributions des *CPU bursts*
- Ordonnancement vise à profiter du parallélisme
  - Maintenir le CPU occupé pendant l'attente d'un périphérique
  - Maintenir les périphérique occupés pendant l'attente du CPU



Histogram des Temps pour CPU-burst



Ordonnanceur du CPU

- Rappel: l'ordonnanceur à **court-terme** choisi parmi les processus dans le **ready queue**
- Décision d'ordonnancement se fait:
  1. Lorsqu'un processus passe de *running* à *waiting* (bloquer)
  2. Lorsqu'un processus passe de *running* à *ready* (interrupt)
  3. Lorsqu'un processus passe de *waiting* à *ready* (E/S termine)
  4. Lorsqu'un processus termine
- Ordonnancement non-préemptif: sous contrôle du processus (#1,#4)
  - Coopératif: pas de conditions de course
- Ordonnancement préemptif: hors de contrôle du processus (#2,#3)

Dispatcher

- Le module de **dispatch** transfère le contrôle au processus sélectionné:
  - Changer le contenu des registres (context switch)
  - Passer en mode *utilisateur*
  - Sauter au bon endroit dans le programme (en utilisant le program counter dans le PCB)

- Latence du dispatcher: temps pour passer d'un processus à un autre

- Cette latence doit être minimisée
  - Impact significatif si fréquence élevée

Critères d'Ordonnancement

- **Utilisation du CPU**
  - À maximiser
- **Débit** – quantité de travail effectif par unité de temps (e.g. #processes/s)
  - À maximiser
- **Délai d'execution** - intervalle entre le moment de la soumission et le moment de l'achèvement
  - À minimiser
- **Temps d'attente** – temps totale qu'un processus passe dans *ready*
  - À minimiser
- **Temps de réponse** – délai entre une requète et le début de sa réponse
  - À minimiser

Menu

- Introduction
- **Algorithmes d'ordonnancement**
- Ordonnancement de threads
- Ordonnancement multi-processor

Ordonnancement FCFS (FIFO queue)

- Exécution dans l'order d'arrivée -> non-préemptif

Processus	CPU burst time
P_1	24
P_2	3
P_3	3

- Diagramme de Gantt

FCFS Scheduling (cont.)

- Supposons que les processus arrivent dans l'ordre: P2 , P3 , P1
- Diagramme de Gantt:

Processus	CPU burst time
P_1	24
P_2	3
P_3	3

- Temps d'attente: P1 = 6; P2 = 0; P3 = 3
- Temps moyen d'attente: (6 + 0 + 3) / 3 = 3 << 17
- L'ordre est très important
- **Convoy effect** - processus court derrière un long processus
  - Ex: Un processus *CPU-bound* et beaucoup de processus *I/O-bound*

Shortest-Job-First (SJF)

- Exécute dans l'ordre de durée, du plus court au plus long

- SJF est optimal – donne le temps d'attente moyen minimum
  - Mais comment peut l'ordonnanceur savoir le durée d'exécution?
    - ✓ Peut demander à l'utilisateur: l'utilisateur doit essayer d'estimer le temps le plus court pour être mieux planifié, mais sans le dépasser, car son processus peut alors être pénalisé, ou reporté
    - ✓ On peut utiliser une **estimation**

Prédiction de la longueur de la prochaine CPU burst

- Estimer la durée d'exécution sur la base du comportement passé

t_n durée *effective* du CPU burst n

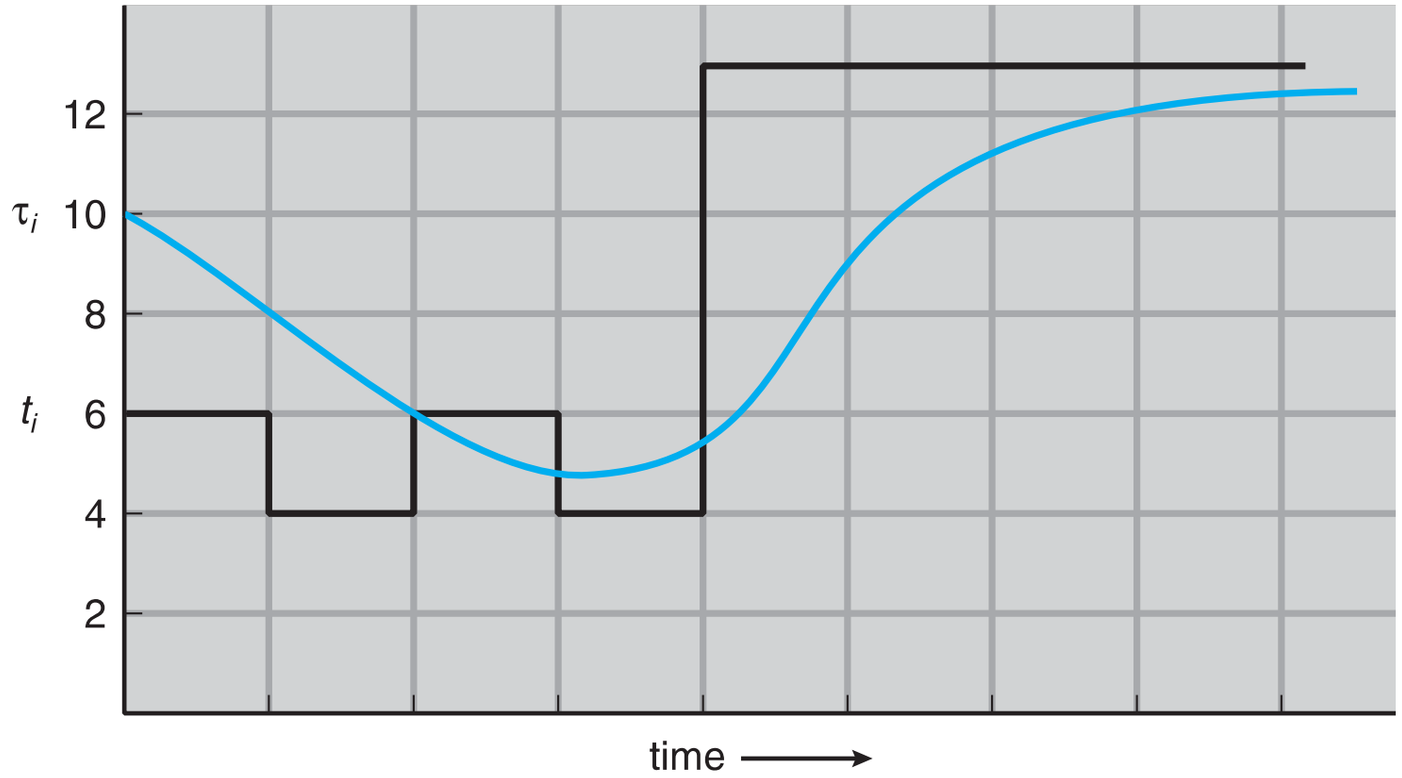
τ_n durée *prévue* du CPU burst n

α facteur d'amortissement

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

- Un processus avec une "burst" de CPU suivante prédite la plus courte que le processus en cours d'exécution peut arrêter le processus en cours
  - Version préemptive appelée **shortest-remaining-time-first**

Prédiction de la longueur de la prochaine CPU burst



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Shortest-remaining-time-first

Processus	*Arrivée*	CPU burst time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Temps moyen d'attente:

- Version Non-preemptif:

Ordonnancement par Priorité

- Une priorité numérique est associée à chaque processus
- L'ordonnanceur choisi le processus de la plus haute priorité
- SJF et SRTF correspondent à une priorité de 1/CPU_burst
- Utilisation d'une priorité combinant plusieurs facteurs:
  - Priorité indiqué par l'utilisateur:
    - Préemptive (termine l'exécution d'unstops executing lower priority) or
    - Non-préemptive (can still switch order in such ready queue)
  - Longueur prévu du prochaine *CPU-burst*
  - **Âge** – pour éviter les *famines*

Exemple d'ordonnancement par priorité

Processus	CPU burst time	Priorité
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Diagramme de Gantt:

- Temps d'attente moyen =

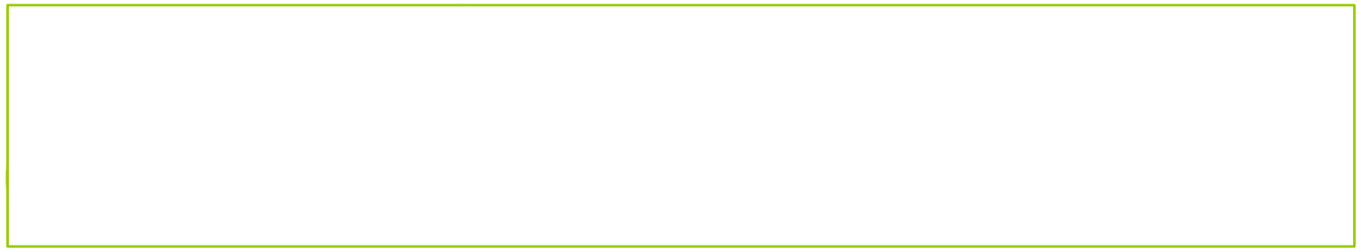
Algorithme du tourniquet (Round Robin)

- Préemption de l'exécution après écoulement d'un **quantum** de temps
  - habituellement, de l'ordre de 10ms-100ms
- S'il y a **n** processus dans la file d'attente prête et que le quantum est **q**, chaque processus reçoit **1/n** du temps CPU en paquets de **q** unités de temps au plus. Aucun processus n'attend plus de **(n-1)q** unités de temps
- Le timer interrompt chaque quantum pour donner le CPU a le processus suivant
- Performance
  - **q** très grand ⇒ FIFO
  - **q** petit ⇒ overhead de context switch est trop haut
- **q** normalement 10msec to 100msec, context switch < 10 usec

Exemple de Round-Robin

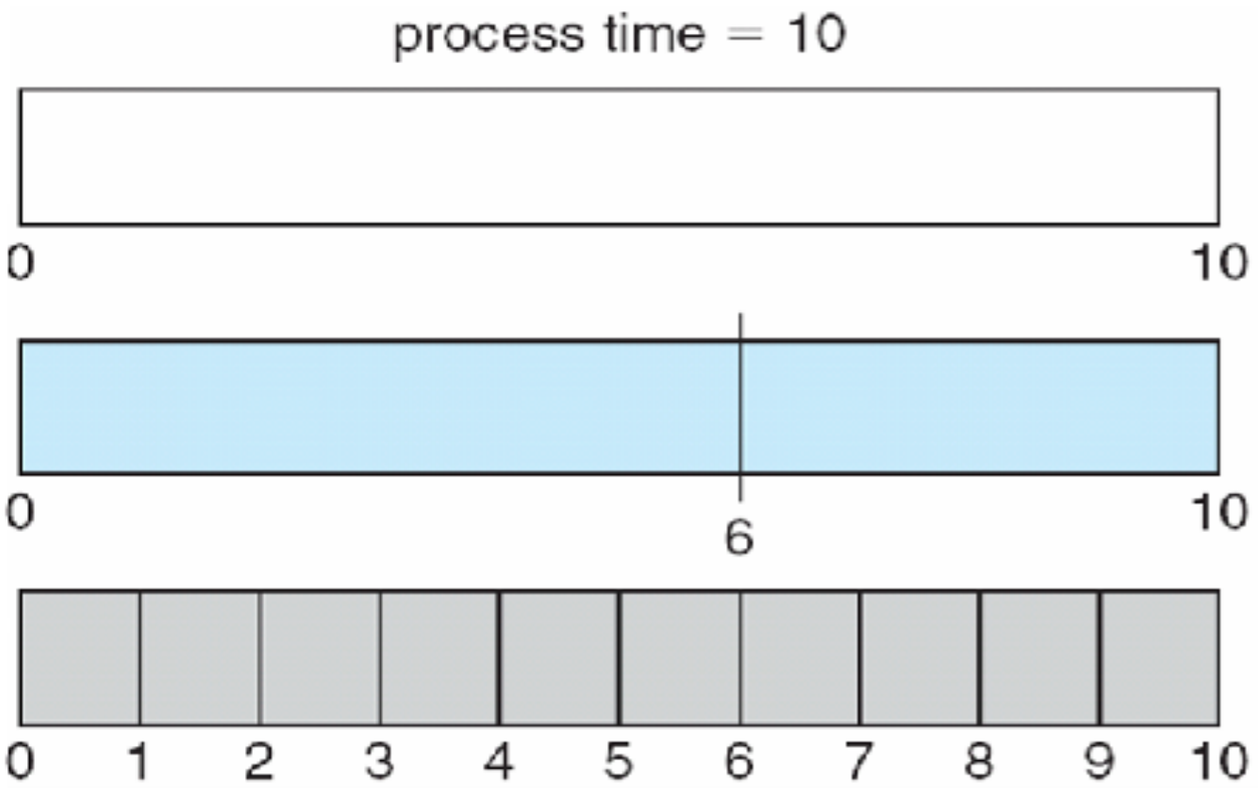
Processus	CPU burst time
P_1	24
P_2	3
P_3	3

Diagramme de Gantt, avec quantum de 4:



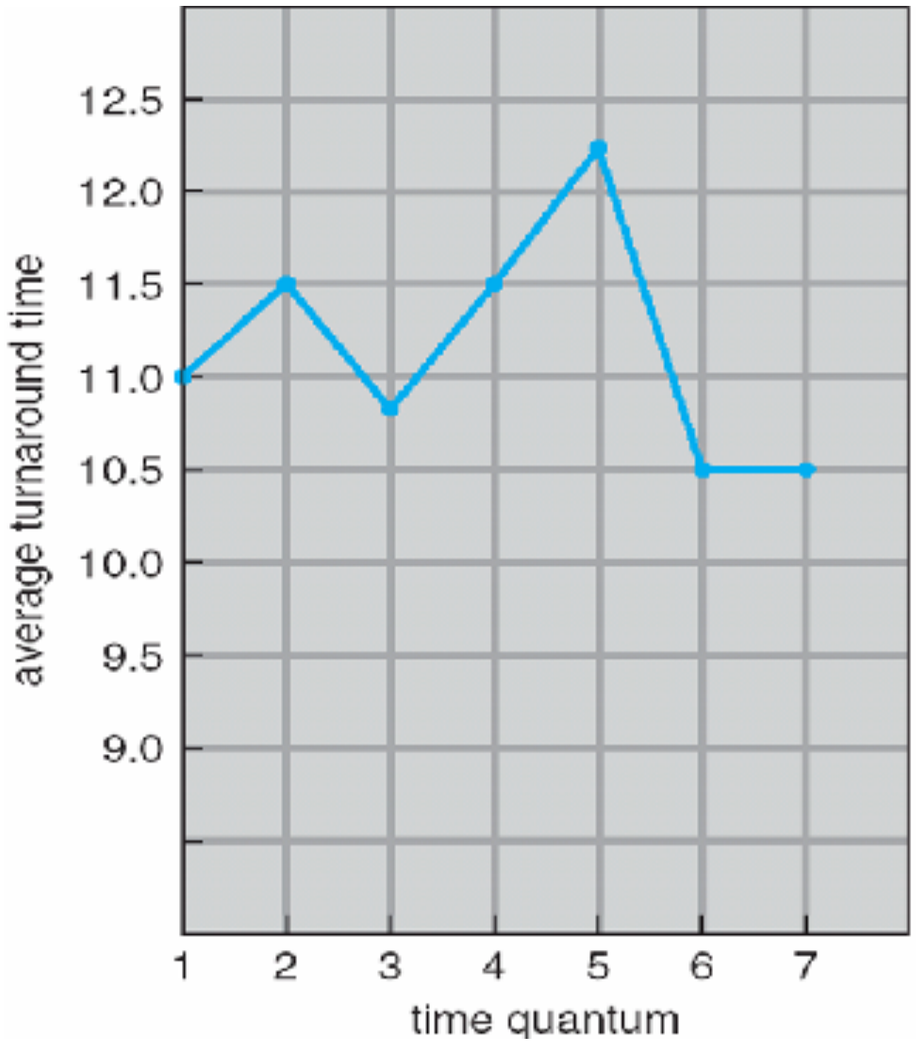
- Temps d'attente moyen:
- Délai d'execution moyen
- Temps réponse moyen:

Temps Quantum vs. Context Switch



quantum	context switches
12	0
6	1
1	9

Delai d'execution vs. Quantum



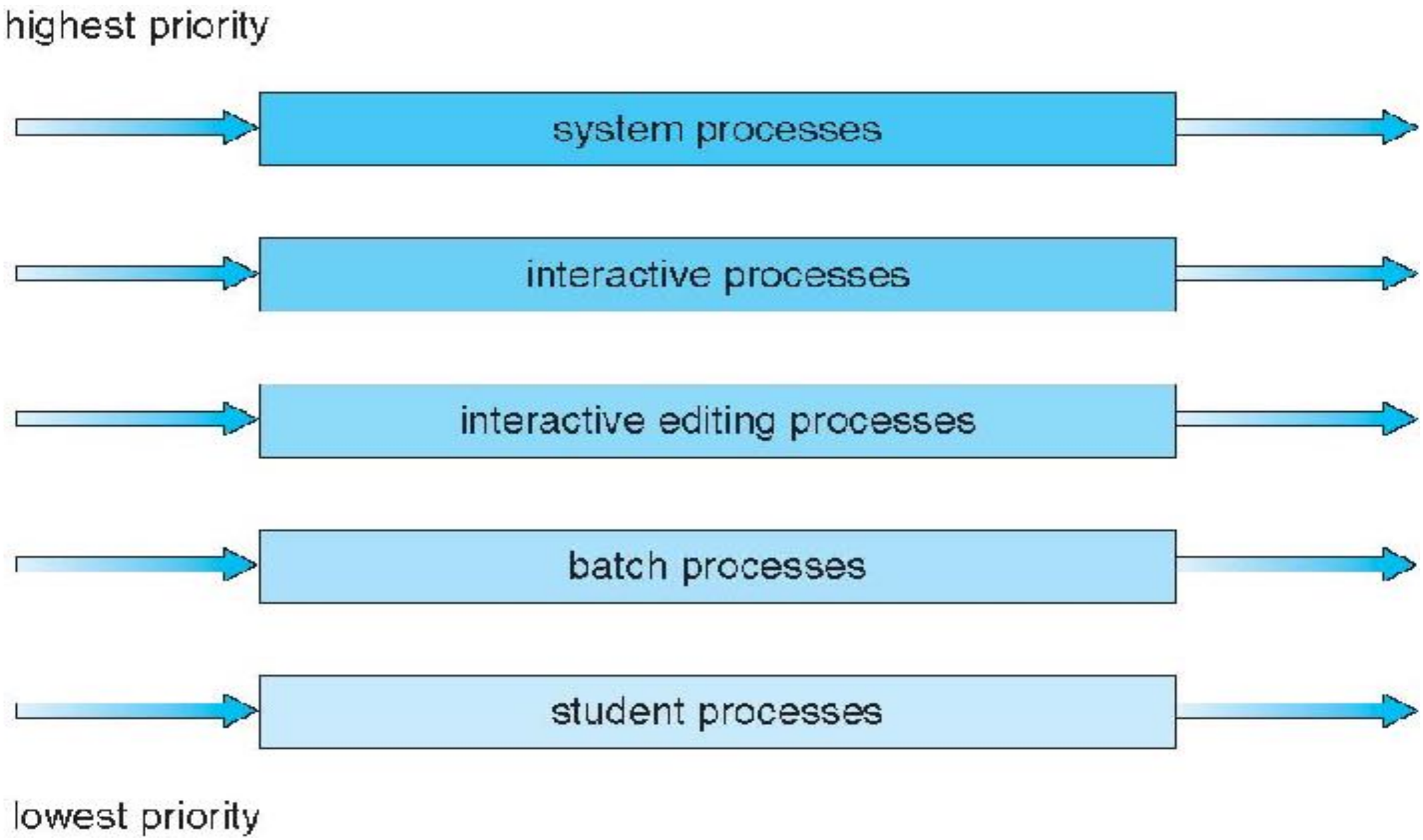
process	time
P_1	6
P_2	3
P_3	1
P_4	7

règle générale:
80% *CPU-bursts* devraient être plus courtes que **q**

Queues multi-niveaux

- La queue *ready* est partitionnée en plusieurs queue, par exemple:
  - **foreground** (interactif)
  - **background** (tâches de fonds)
- Processus toujours dans même file d'attente donnée
- Chaque file d'attente a son propre algorithme d'ordonnancement:
  - foreground - RR
  - background - FCFS
- l'ordonnancement doit être effectuée entre les queues:
  - Ordonnancement à priorité fixe
    - i.e., servir tous à dans foreground avant le background
    - Risques de famine
  - Tranche de temps
    - chaque file d'attente reçoit une certaine quantité de temps CPU qu'elle peut programmer parmi ses processus
    - par exemple, 80% à foreground en RR, 20% à background en FCFS

Ordonnancement en Multilevel Queue



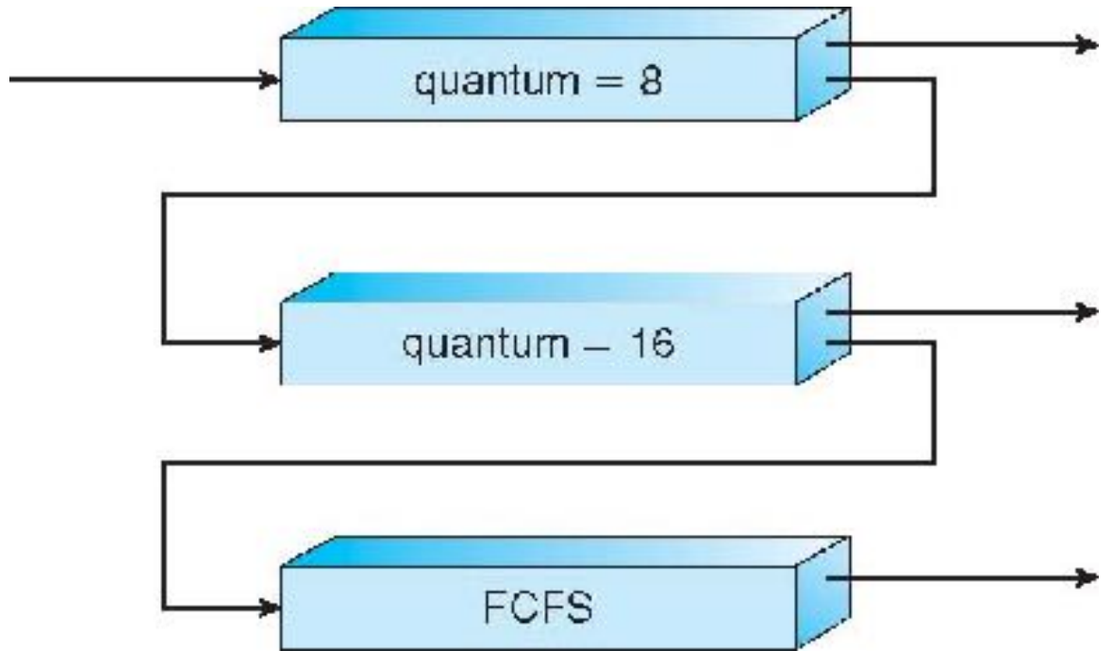
Queue multi-niveaux à rétroaction

- Les processus peuvent changer de queue
  - Typiquement, de manière automatique, par âge ou priorité

- Paramétrée par:
  - Nombres de queues
  - Ordonnancement de chaque queue
  - Ordonnancement entre les queues
  - Critère de promotion de processus
  - Critère de démotion de processus

Exemple de multi-level feedback queue

- Trois queues:
  - Q0 – RR avec quantum de 8 ms
  - Q1 – RR avec quantum de 16 ms
  - Q2 – FCFS



- Ordonnancement:
  - Nouveau processus entrent a **Q0** et sont servi FCFS
    - ✓ Quand il gagne CPU, le travail reçoit 8ms
    - ✓ S'il ne se termine pas en 8ms, le travail est déplacé vers **Q1**
  - A Q1, le travail est à nouveau servi par FCFS et reçoit 16ms supplémentaires
    - ✓ S'il ne se termine toujours pas, il est préempté et déplacé vers la file d'attente **Q2**

Menu

- Introduction
- Algorithmes d'ordonnancement
- **Ordonnancement de threads**
- Ordonnancement multi-processor

Ordonnancement des threads

- Rappel: Il y a des threads noyaux et les threads utilisateurs
- Rappel: Modèles many-to-one et many-to-many, thread library schedules user-level threads to run on LWP
  - P**rocess-contention scope** (PCS) car competition est entres les threads dans un processus
- Thread de noyau programmé sur CPU est **system-contention scope** (SCS) – competition entre tous les threads du système

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling 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");
   }
```

Pthread Scheduling API

```
/* 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(&id[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);
}
```

Menu

- Introduction
- Algorithmes d’ordonnancement
- Ordonnancement de threads
- **Ordonnancement multi-processor**

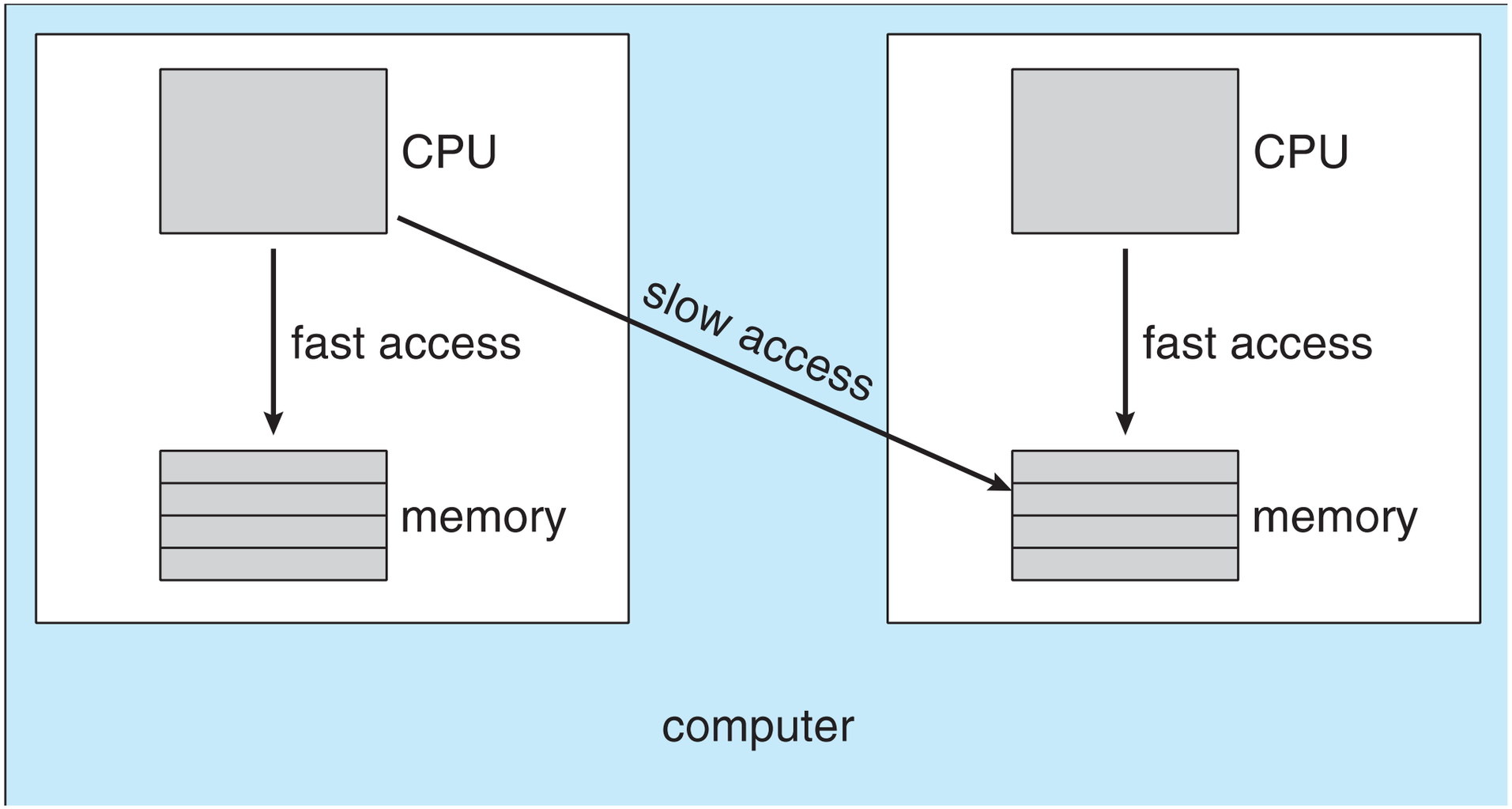
Ordonnancement multi-processeurs

- Ordonnancement plus complexe

- **Multiprocesseur asymétrique** – un processeur s'ordonnance
- **Multiprocesseur symétrique** (SMP) – chaque processeur s'ordonnance

- **Affinité à un processeur** – processus a une affinité pour le processeur sur lequel il est actuellement en cours d'exécution (pourquoi?)
  - Affinité "soft" – un processus peut migrer occasionnellement
  - Affinité "hard" - un processus reste dans son processeur

"NUMA"



il pourrait être que le processeur a un accès plus rapide à certaines parties de la mémoire

Équilibrage de charge

- “load balancing”: tenter de maintenir les processeurs également occupés
  - Migration “Push”: Un tâche vérifient périodiquement la charge des processeur
  - Migration “Pull”: Processeur inactif tire un tâche qui attend un processeur occupé
- La migration s’oppose à l’affinité

Sommaire

- Les critères d'evaluation: utilisation du CPU, débit, délai d'execution, temps d'attente, temps de réponse
- Les algorithms: First-come-first-server (FCFS) = FIFO, Shortest-Job-First (comment on peut savoir le durée d'exécution?), Shortest-remaining-time-first (preemptif vs. non-preemptif), priorité, round robin
- Queues multi-niveaux
  - à retroaction
- Ordonnancement de threads (PCS vs SCS)
- Ordonnancement multi-processor: Asymétrique vs symétrique, l'affinité à un processeur, équilibrage de charge