Systèmes d'Exploitation II

Ordonnancement

Pablo de Oliveira

Caractéristiques des Processus

Alternance E/S et Calcul

- La plupart des programmes alternent entre:
- des phases de calcul (utilisation du CPU)
- des phases d'Entrée/Sortie (attente des données)
- Quelle est la longueur moyenne d'une phase de calcul ?

Distribution des phases de calcul (CPU Burst)

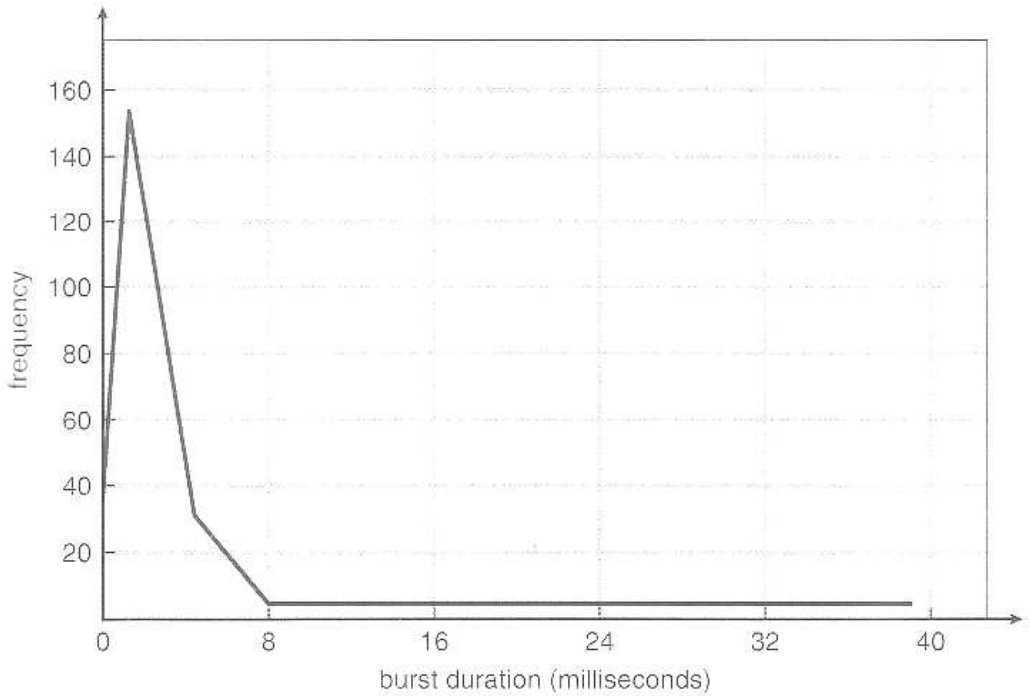


Figure 5.2 Histogram of CPU-burst durations.

- Loi exponentielle
- Phases de CPU longues sont rares
- Correspondent à du calcul sans aucune sortie sur disque, ou d'entrée utilisateur.

Classification des processus

- **CPU-bound** (Limité par le Calcul)
  - processus gourmand en CPU, phases CPU longues
- **IO-bound** (Limité par les E/S)
  - le processus passe son temps à attendre les E/S
- **Intéractif**
  - phases CPU courtes, puis phases E/S très longues (attente de l'utilisateur)

Exemple CPU-Bound vs IO-Bound

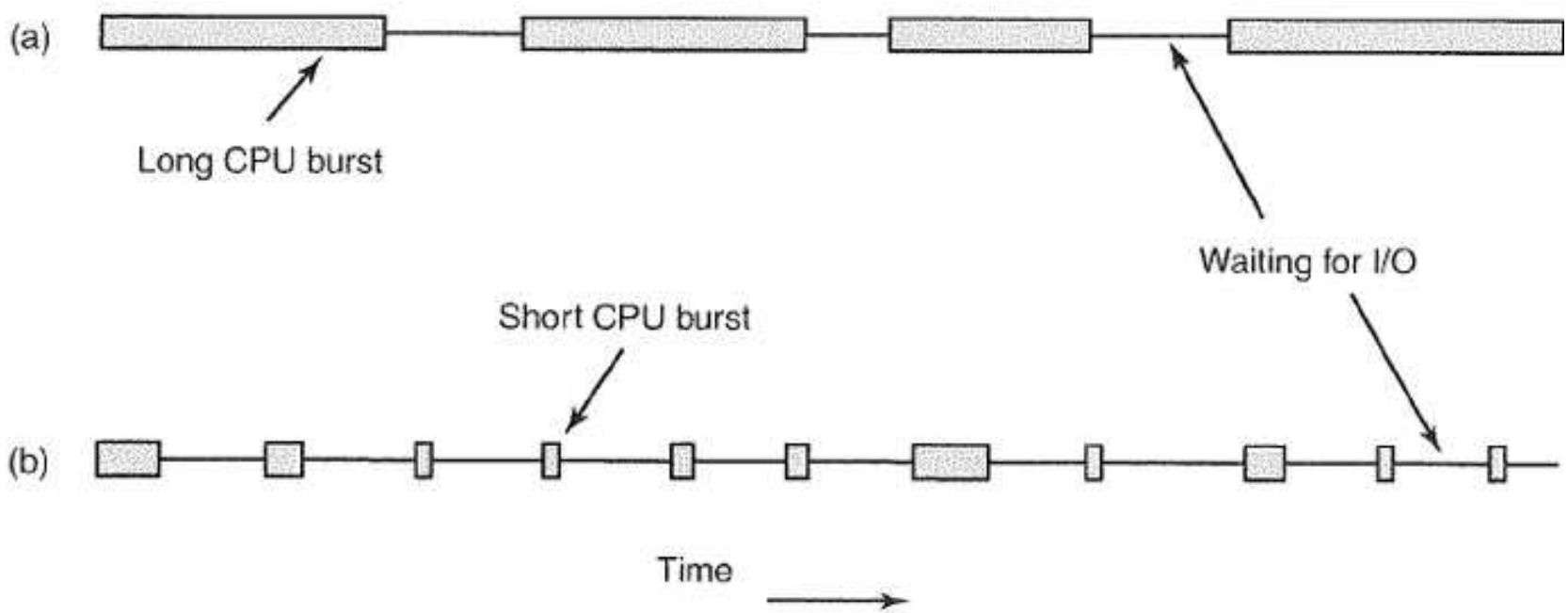


Figure 2-38. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

Ordonnancement

Architecture mono-cœur ou multi-cœur

- Ordonnancement sur un processeur:
  - Plusieurs tâches s'exécutent sur un seul processeur. Un ordonnanceur, passe d'une tâche à l'autre pour simuler une execution concurrente.
- Ordonnancement sur multi-processeur:
  - Plusieurs tâches s'exécutent sur plusieurs processeurs, en parallèle.

Problème

- Quel est le meilleur ordonnanceur ?

Objectifs

- Point de vue Système:
  - Maximiser utilisation CPU
  - Maximiser le débit (nombre de tâches complétées par unité de temps)
- Point de vue Utilisateur:
  - Minimiser la latence:
    - Temps de complétion d'une tâche (durée entre l'arrivée d'une tâche et sa complétion)
    - Temps de réponse (durée entre l'arrivée d'une tâche et le début de son exécution)
  - Minimiser l'attente (durée passée à attendre)

Ordonnanceur Préemptif

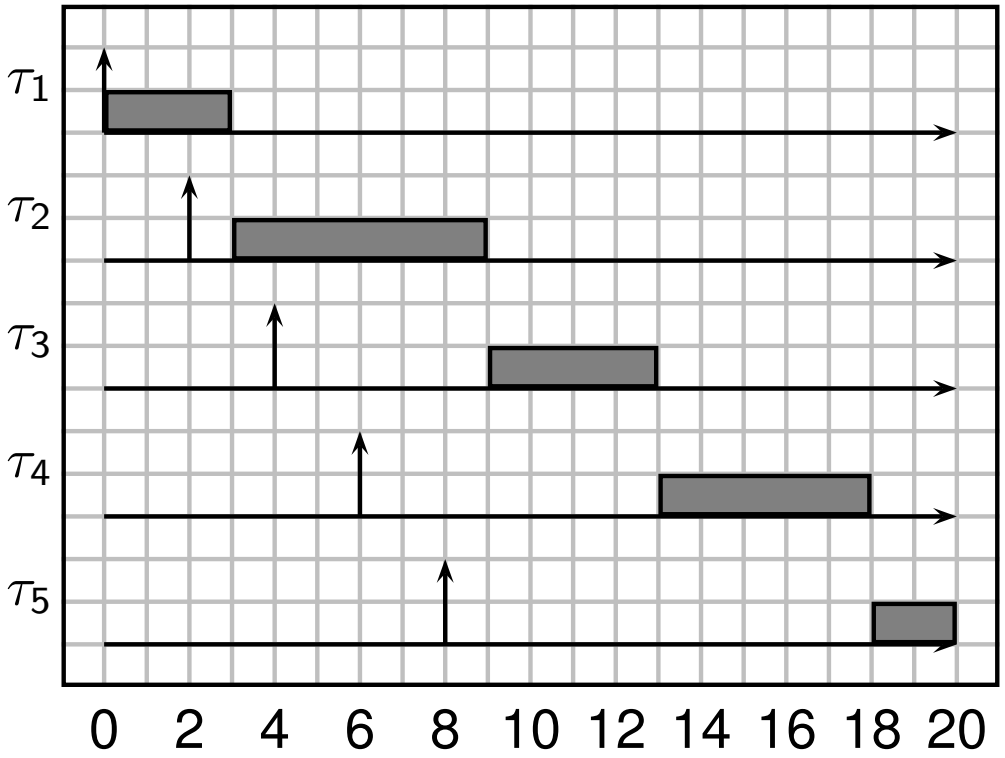
- Un ordonnanceur **Non-Préemtif** n'interrompt jamais une tâche qu'il a commencé.
- Un ordonnanceur **Préemtif** peut interrompre une tâche trop longue, ou bloquée, pour exécuter une autre tâche.

Ordonnancement FCFS

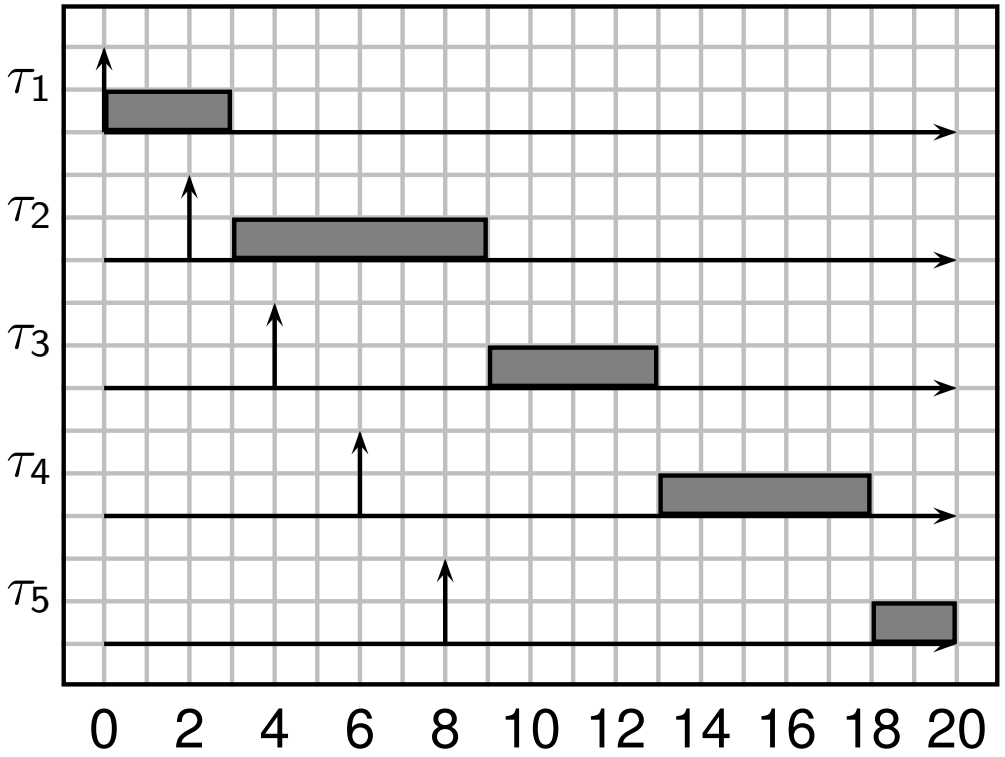
- Premier arrivé premier servi
- Non Préemptif

FCFS: Exemple d'ordonnancement

- Tâches τ_1 à τ_5
- Activation A_i: 0, 2, 4, 6, 8
- Durée D_i: 3, 6, 4, 5, 2



FCFS: Métriques

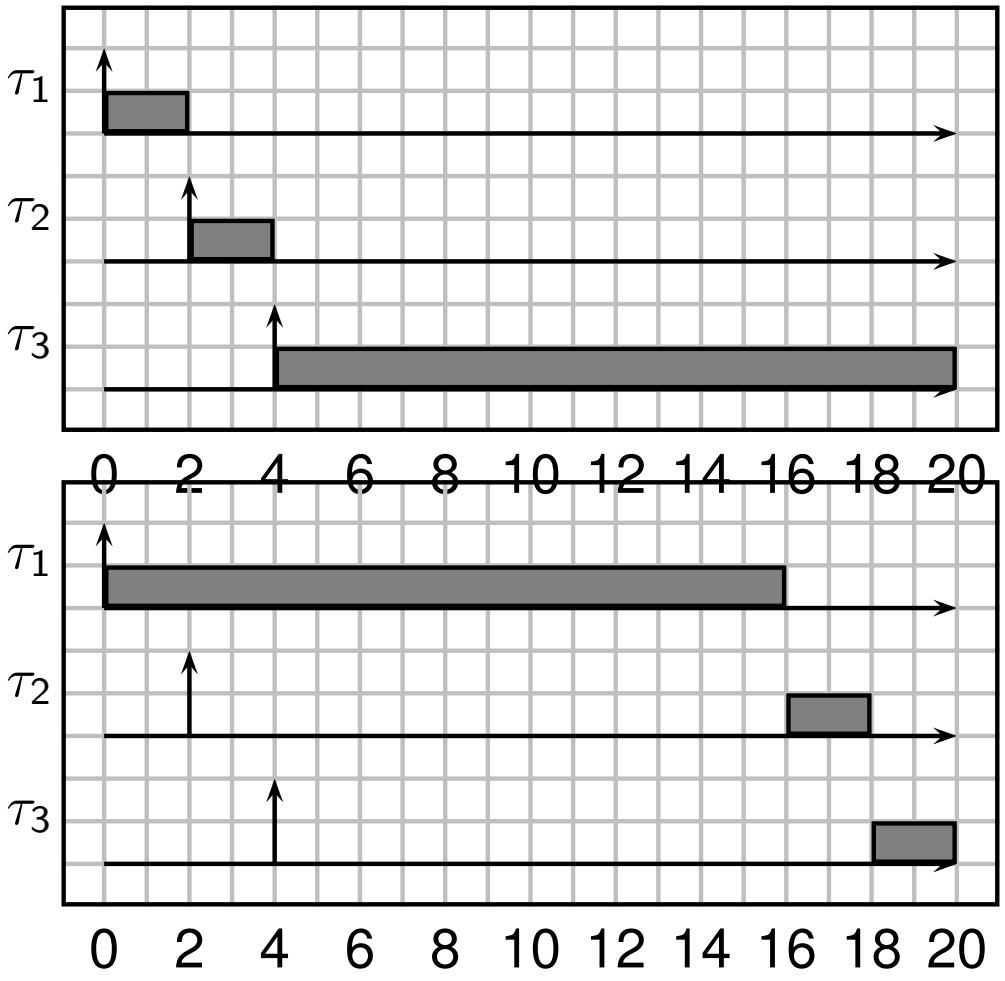


- Utilisation CPU: 100% et Débit: $\frac{5}{20} = 0.25$
- Complétion (total):
  $(3-0)+(9-2)+(13-4)+(18-6)+(20-8) = 43$
- Réponse/Attente (total): 23 (non-préemtif réponse = attente)

FCFS: Analyse

- Avantages:
  - Simple à implémenter
- Désavantages:
  - Effet de “convoi”: les jobs courts attendent beaucoup derrière un gros job
  - Performance est très variable selon l'ordre des tâches

Effet de convoi: illustration



SJF Shortest Job First (Plus court job d'abord)

SJF est optimal pour réduire le temps d'attente

- On suppose que l'on exécute les processus dans l'ordre $1, 2, \ldots, k$
- Soient les durées d'exécution $p_1, p_2, \ldots, p_k$
- Alors le temps d'attente total est:

$$0+p1+(p_1+p_2)+(p_1+p_2+p_3)+\cdots = (k-1).p_1+(k-2).p_2+\cdots+p_k$$

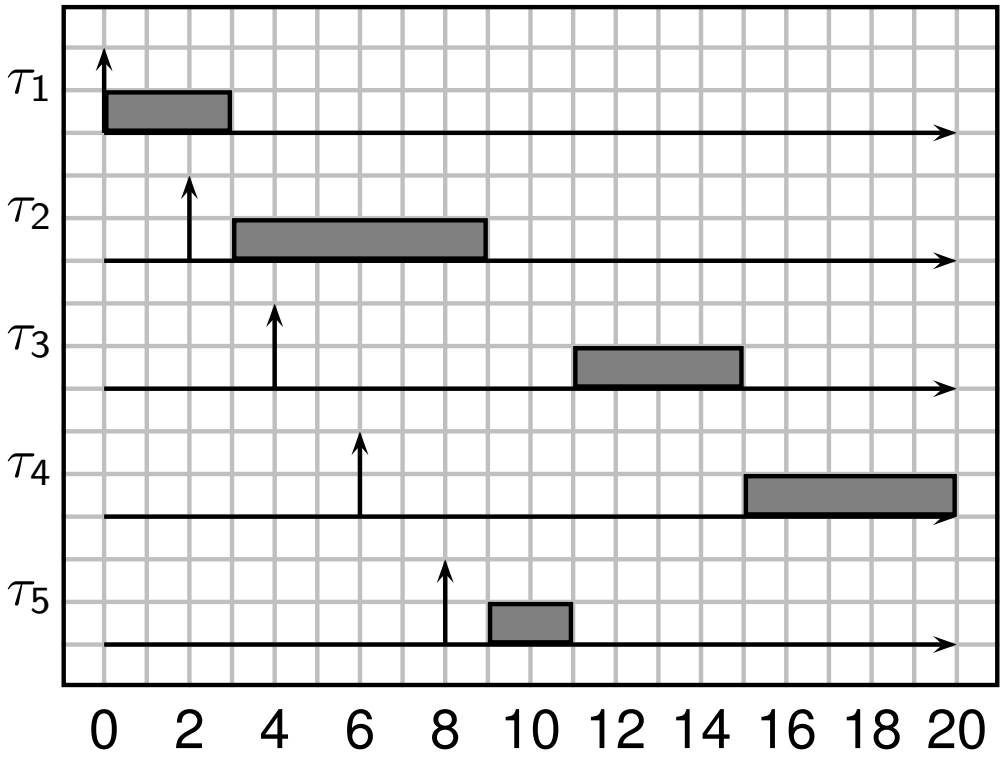
- Pour diminuer le temps d'attente, il faut exécuter d'abord les jobs les plus courts

SJF: deux versions

- Non préemptive: un job n'est pas interrompu
- Préemptive: Shortest-Remaining-Time-First (SRTF)
  - si un job arrive qui est plus court que le temps restant du processus courant, on préempte.

SJF: Exemple d'ordonnancement

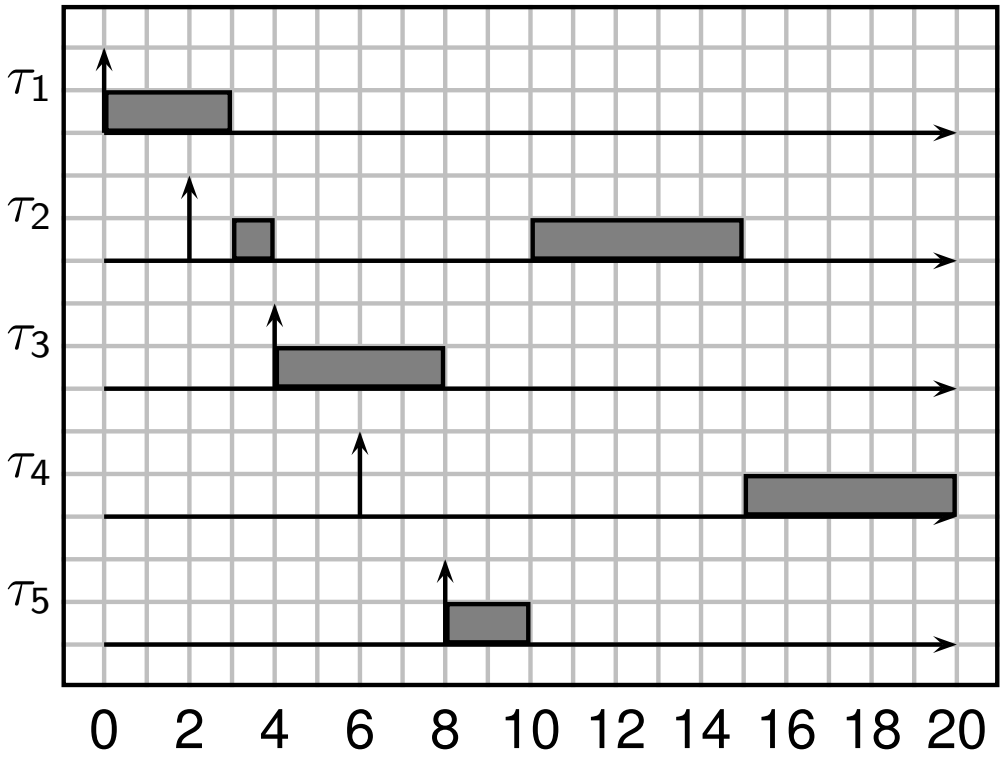
- Version non-préemptive



- Complétion (total): 38
- Attente (total): 18
- Réponse (total): 18

SRTF: Exemple d'ordonnancement

- Version préemptive



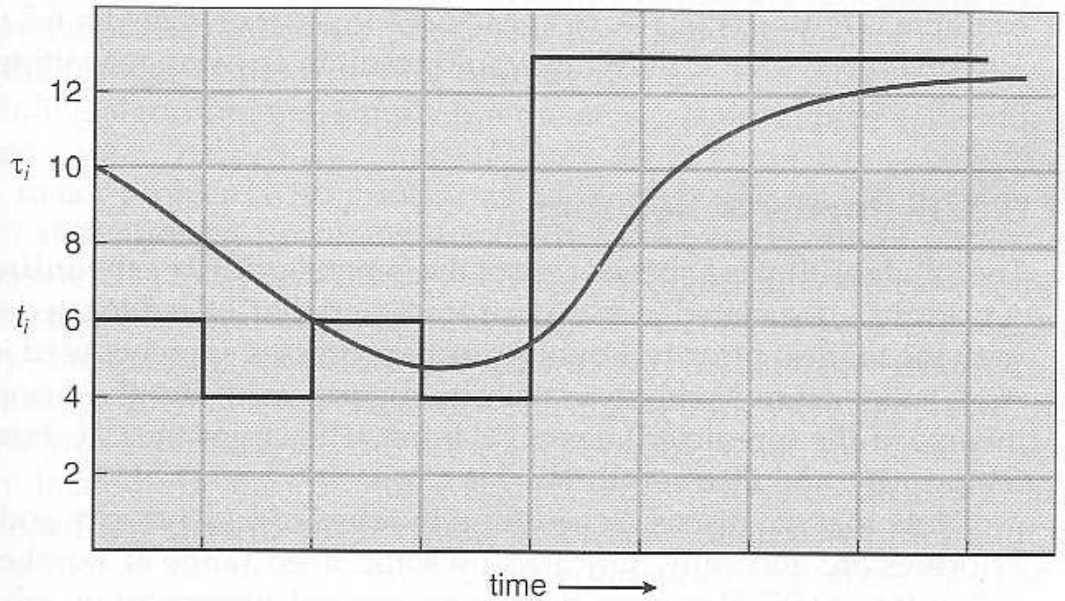
- Complétion (total): 36
- Attente (total): 16
- Réponse (total): 10

SJF/SRTF: Analyse

- Avantages:
  - Optimal pour le temps d'attente moyen
- Désavantages:
  - Difficile à estimer la durée d'un processus
  - Famine: si beaucoup de jobs courts, les jobs longs ne s'exécutent jamais

SJF: Comment estimer le temps d'un processus ?

- Hypothèse: sur une courte fenêtre de temps, la distribution des durées est proche. Utiliser le passé pour prédire l'avenir.
- Moyenne exponentielle: $prediction_{i+1} = \alpha.duree_i + (1 - \alpha).prediction_i$



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

Figure 5.3 Prediction of the length of the next CPU burst.

Ordonnancement Round-Robin

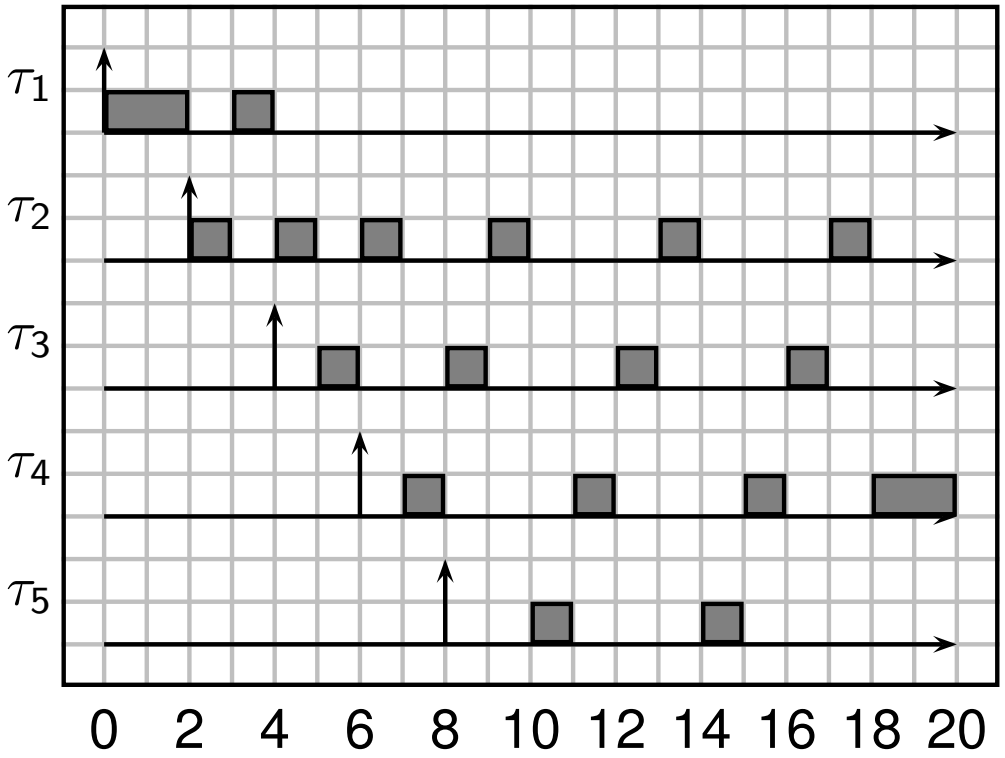
- Ordonnancement en Tourniquet
- On définit un quantum q
- Changement de processus:
  - Si un processus est plus long que q il est préempté
  - Si un processus attend une entrée il est préempté

RR: Intéressant pour contrôler le temps de réponse

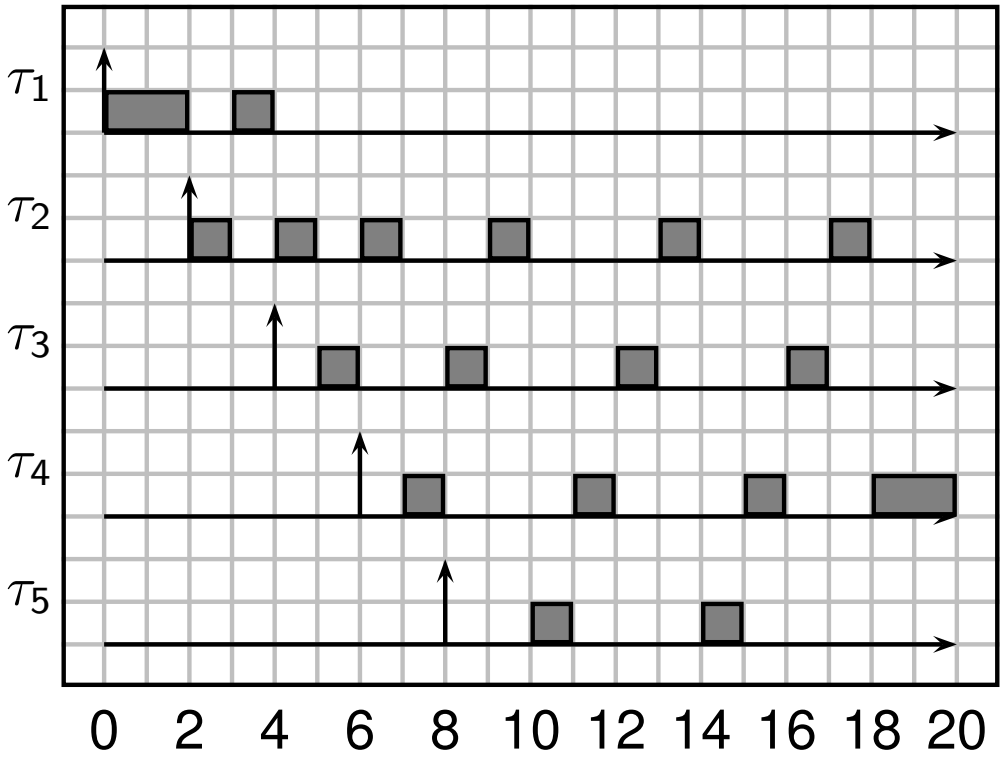
- Avec n processus
  - Chaque processus utilise (environ) $\frac{1}{n}$ du temps CPU
  - Chaque exécution dure au maximum q unités de temps
  - Aucun processus n'attends sans réponse plus de $(n-1)q$ unités de temps

Exemple d'ordonnancement (q=1)

- Tâches τ_1 à τ_5
- Activation A_i: 0, 2, 4, 6, 8
- Durée D_i: 3, 6, 4, 5, 2



RR: Métriques



- Complétion (total): 54
- Réponse (total): 4
- Attente (total): 34

RR: Analyse

- Avantages:
  - Intéressant pour les jobs courts
  - Équitable
  - Moins variable que FCFS
- Désavantages:
  - Pour les jobs longs, le coût de changement de contexte est problématique

RR: Comment choisir q?

- Trop grand: RR dégénère en FCFS, mauvais pour les jobs courts
- Trop petit: coût de changement de contexte pénalise les jobs longs
- À chaque préemption, on paye le coût de changement de contexte c.
  - Il faut choisir q de 10 à 100 fois plus grand que c
  - En pratique aujourd'hui:
    - q est entre 10ms et 100ms
    - c est entre 0.1ms et 1ms

Ordonnancement Multi-niveaux

- Difficile d'estimer le temps d'un processus.
  - Ordonnanceur adaptatif
- On considère plusieurs priorités.
  - Le processus prêt de plus haute priorité est toujours choisi.
  - Au début tous les processus ont la priorité max.
  - Chaque processus à un quota d'exécution q

Multi-niveaux: Analyse

- Stratégie:
  - Si un processus dépasse un quota q, il est préempté et sa priorité est réduite
  - Si un processus bloque avant q, sa priorité est augmentée
- Résultat:
  - Les processus très courts (IO-bound) se retrouvent avec des priorités élevées
  - Les processus longs (CPU-bound) se retrouvent avec des priorités faibles
- Famine:
  - Pour éviter la famine, les processus “vieux” reçoivent un bonus de priorité

Et dans Linux ?

- Linux 1.2: Round-Robin
- Linux 2.2: Introduit des priorités
- Linux 2.4: Semblable à un ordonnancement Multi-Niveaux en $O(n)$
- Linux 2.6: Multi-Niveaux en $O(1)$
- Linux 2.6.21: CFS (Completely Fair Scheduler), équilibre le temps alloué à chaque tâche en utilisant un RB-Tree (http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/)

Et le multi-coeur ?

- Problème complexe, recherche en cours.
- Quelques problèmes intéressants:
  - Affinité: mettre les tâches qui communiquent sur le même processeur
  - Équilibrage de charge: répartir les tâches équitablement entre les processeurs
  - Migration: coûteuse entre deux processeurs différents

Résumé

- SJF et SRTF optimaux pour le temps d'attente
  - Mais difficile de prédire le temps d'un processus
- RR équitable
  - Intéressant pour les processus interactifs
  - Peu de variabilité entre processus
- FCFS simple a implémenter
  - Performance très variable, effet de convoi
- Multi-Niveaux Efficace pour gérer aussi bien des jobs CPU et IO-bound