Ordonnancement

Concepts de base

Critères d'ordonnancement

Algorithmes d'ordonnancement

Ordonnancement de threads

Ordonnancement dans les systèmes multiprocesseurs

Concepts de base

L'ordonnancement de processus permet de mieux utiliser le CPU

Exécution découpée en *CPU burst* et *I/O burst*

Préoccupation principale: distribution des *CPU bursts*

Ordonnancement vise à profiter du parallélisme

- Maintenir le CPU occupé pendant l'attente d'un périphérique
- Maintenir les périphériques occupés pendant l'attente du CPU

Ordonnanceur du CPU

l'*ordonnanceur à court-terme* choisi parmi les processus *ready*

Décision d'ordonnancement se fait:

1. Lorsqu'un processus passe de *running* à *waiting*
2. Lorsqu'un processus passe de *running* à *ready*
3. Lorsqu'un processus passe de *waiting* à *ready*
4. Lorsqu'un processus termine

Ordonnancement *non-préemptif*: sous contrôle du processus

- *coopératif*: pas de conditions de course

Ordonnancement *préemptif*: hors de contrôle du processus

Dispatcher

Le module de *dispatch* transfère le contrôle au processus sélectionné

- Changer le contenu des registres
- Passer en mode *utilisateur*
- Sauter au bon endroit dans le programme

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

- Le temps perdu dans un *context-switch* inclus des cache-miss

Cette latence doit être minimisée

- Impact significatif si fréquence élevée

Université de Montréal

Critères d'ordonnancement

Utilisation du CPU

- À maximiser, traditionnellement, minimiser plus récemment

Débit: quantité de travail effectif par unité de temps

- À maximiser

Temps d'attente: temps qu'un processus passe dans *ready*

- À minimiser

Temps de réponse: délai entre une requète et le début de sa réponse

- À minimiser

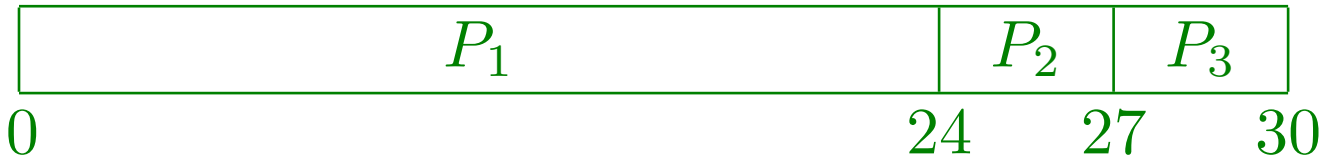
Université de Montréal

Ordonnancement FCFS (FIFO)

Exécution naïve dans l'ordre d'arrivée; *non-préemptif*

Processus	CPU burst time
P_1	24
P_2	3
P_3	3

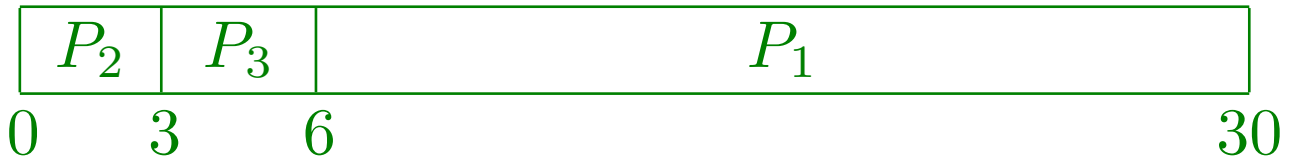
Diagramme de Gantt



Temps moyen d'attente: $(0 + 24 + 27)/3 = 17$

Shortest-Job First (SJF)

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



Temps moyen d'attente: $(0 + 3 + 6)/3 = 3$

SJF est *optimal*: donne le temps d'attente moyen minimum

- Bien sûr, en général, la durée d'exécution est inconnue!
- Le principe est quand même fréquemment utilisé
- On peut utiliser une *estimation*

Deviner la durée d'exécution

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

- Le passé est un bon prédicteur du futur

t_n durée *effective* du CPU burst n

τ_n durée *prévue* du CPU burst n

α facteur d'amortissement

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

Valeur courante de α: 0.5

Durée exacte peu importante: ordre de grandeur (10ms vs 10s)

$\alpha = 1$: pas de mémoire

$\alpha = 0$: ignore exécution effective

Université de Montréal

Shortest Remaining Time First

Comme SJF, mais préemptif

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

Ordonnancement par priorité

Une priorité numérique est associée à chaque processus

L'ordonnanceur choisi le processus de la plus haute priorité

En version *préemptive* ou non

SJF et SRTF correspondent à une priorité de $1/\text{burst}$

Utilisation d'une priorité combinant plusieurs facteurs

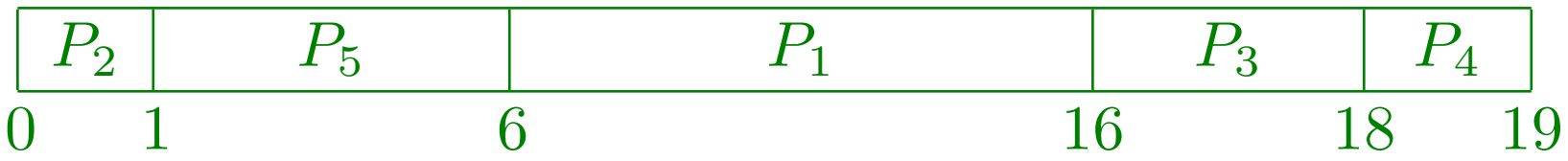
- Priorité indiquée par l'utilisateur
- $1/\tau_n$, pour avantager les processus courts
- Âge, pour éviter les *famines*

Université de Montréal

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:



Algorithme du tourniquet (Round Robin)

Préemption de l'exécution après écoulement d'un *quantum* de temps

- habituellement, de l'ordre de 10ms-100ms

Accorde un quantum à chaque processus avant de recommencer

Un long processus ne peut pas retarder excessivement un autre

Offre l'illusion de l'exécution simultanée

- N processus à la fois, mais N fois plus lentement

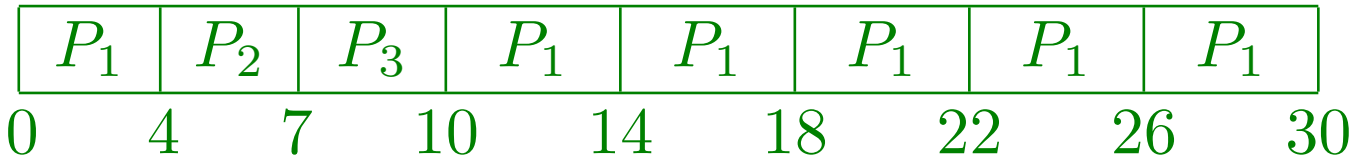
Choix du quantum important:

- trop grand: dégénère en FIFO
- trop court: cause perte de temps en context-switch

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: $(6(!) + 4 + 7)/3 = 5.66$ (dépend du quantum)

Généralement meilleurs temps de réponse que SJF

Queues multi-niveaux

Plusieurs queues *ready*

- Tâches de fonds, tâches interactives, tâches de système, ...

Chaque queue peut avoir son propre algorithme d'ordonnancement

- tâches interactives = RR, tâches de fonds = FIFO

Ordonnancement entre les queues:

- Par priorité des queues
- RR, possiblement pondéré (80% interactif, 20% tâches de fonds)

Efficace en resources, mais peu ou trop flexible

Université de Montréal

Queues multi-niveaux à rétroaction

Multi-level feedback queues

Les processus peuvent changer de queue

- Typiquement, de manière automatique, par âge ou τ_n

Grande catégorie paramétrée par:

- Nombre queues
- Ordonnancement de chaque queue
- Ordonnancement entre les queues (habituellement, priorité)
- Critère de promotion de processus
- Critère de démotion de processus

Exemple de multi-level feedback queues

N queues

Queue i : priorité $N - i$; RR avec un quantum de 2^i

Promotion quand le CPU burst se termine avant 50% du quantum

- Processus courts et interactifs augmente de priorité

Démotion quand le quantum se termine avant le CPU burst

- Processus longs diminuent de priorité

Promotion régulière de tous les processus

- Évite la famine: temps maximum garanti

Université de Montréal

Ordonnancement de threads

Distinction entre threads *user-level* et *kernel-level*

Si possible, ordonnancement par thread plutôt que par processus

Proportion du CPU peut être par thread ou par processus

- SCS (system contention scope): compétition entre tous les threads
- PCS (process contention scope): compétition entre thread siblings
- Plus généralement, une hiérarchie

Université de Montréal

Ordonnancement multi-processeurs

Ordonnancement plus complexe

Systèmes *homogènes*

- Multiprocesseur symétrique: chaque processeur s'ordonnance
- Multiprocesseur asymétrique: un processeur se charge des autres

Affinité à un processeur: préférence pour bénéficier de la localité

- Affinité *hard*: un processus reste dans son processeur
- Affinité *soft*: un processus peut migrer occasionnellement
- Processeurs (et affinités) regroupés hiérarchiquement

Équilibrage de charge

load balancing: tenter de maintenir les processeurs également occupés

- Migration *push*: tâche périodique de rééquilibrage
- Migration *pull*: processeur *idle* va chercher du travail ailleurs

On peut utiliser les deux à la fois

La migration s'oppose à l'affinité

- Exemple classique: 3 processus sur 2 CPUs
- Préférable de maintenir un processeur occasionnellement *idle*