250 Ordonnancement des processus

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Ordonnanceur

Qu'est-ce que c'est

- C'est une partie du SE
- Il sert à déterminer quels processus sont actifs (et lesquels ne le sont pas)

Ses règles de base

- Ne peut qu'élire un processus prêt
- L'élu est celui qui a la plus haute priorité compte tenu de la politique locale

Schéma algo ordonnanceur

tant que *pas de processus élu* **faire**
    consulter liste des processus prêts;
    sélectionner celui qui a la plus haute priorité;
    si *pas d'élu* **alors**
        attendre jusqu'à la prochaine interruption
        (processeur à l'état latent);
    fin
fin
marquer le processus élu actif;
basculer le contexte;

Quand intervenir ?

Aux changements d'état

- Création de processus
- Terminaison d'un processus
- Passage d'actif à bloqué (demande d'E-S)
- Passage de bloqué à prêt (ressource disponible)
- Passage d'actif à prêt (fin de quantum)

$\rightarrow$ Mais aussi si changement de priorité (ou d'ordonnanceur)

Concrètement ?

- Appel système
- Interruption matérielle, dont l'horloge programmable
- **Question.** Ces deux cas couvrent-ils toutes les possibilités ?

Ordonnanceurs non-préemptifs

Processus actif jusqu'à

- Une demande d'entrée-sortie bloquante
  (ou tout autre appel système bloquant)
- Une demande explicite de laisser la main (`sched_yield(2)`)

$\rightarrow$ Dans les deux cas, c'est à la demande du processus

- On parle aussi de « multitâche coopératif »
- Très rare dans les systèmes modernes

Question

- Et si on laissait la main à un processus bloqué par une E-S ?

Ordonnanceurs préemptifs

À n'importe quel moment, on peut suspendre un processus

Fin du tour

- Expiration d'un quantum de temps alloué au processus

$\rightarrow$ Interruption matérielle due a l'horloge programmable

Perte de priorité

- Nouveau processus prioritaire créé
- Processus prioritaire qui passe de bloqué à prêt
- Changement de priorité dans les processus

Question

- Quels sont les avantages du préemptif sur le non-préemptif ?

Objectifs d'ordonnancement

- Respect de la politique locale
  - Les processus plus prioritaires ont plus la main
- Équité
  - Tous les processus de même priorité ont autant la main l'un que l'autre
- Efficience
  - Utilisation efficace des différentes ressources (processeur)

Problème : on ne peut pas toujours avoir les 3

Exemple: 3 processus de même priorité sur 2 processeurs

- On met deux processus sur un CPU et le 3e sur l'autre
  $\rightarrow$ Inéquitable
- On alterne et déplace les processus entre CPU
  $\rightarrow$ Inefficient

$\rightarrow$ il faut faire des compromis!

Critères d'évaluation

- Maximiser le nombre de tâches terminées par unité de temps
- Minimiser le temps entre acceptation et terminaison (temps total)
- Maximiser le temps d'utilisation du CPU
- Minimiser le temps d'attente (latence)
- Maximiser le temps de réponse (interactivité)

$\rightarrow$ Ils ne sont pas indépendants

Pour chaque critère

- En moyenne ?
- Au pire ?
- Au mieux ?

Beaucoup de critères possibles et on n'a encore rien fait en pratique

Objectifs d'ordonnancement spécifiques

Pour les systèmes interactifs

- Minimiser le temps de réponse
- Proportionnaliser le temps de réponse à la complexité perçue de la tâche

$\rightarrow$ Donner l'impression à l'utilisateur que le système est réactif

Pour les systèmes temps réel

- Respecter les contraintes de temps (au pire cas)
- Prédiction de la qualité de service
- Les systèmes temps réel ont des besoins spéciaux et des ordonnanceurs spéciaux
- On y reviendra

CPU bound vs. *I/O bound*

- *CPU burst* : le **temps de calcul** avant prochaine E-S (ou prochain appel système bloquant)

Programme *CPU bound*

- Le processeur est le facteur limitant
- Surtout des calculs, peu d'entrées-sorties
- *CPU bursts* probablement longs

Programme *I/O bound*

- Les entrées-sorties sont le facteur limitant
- Surtout des entrées-sorties, peu de calculs
- *CPU bursts* probablement courts

Questions

- En quoi savoir la catégorie aide l'ordonnanceur ?
- Peut-on catégoriser plus finement ?

Ordonnancement sous Linux

Plusieurs politiques cohabitent

- 3 « normales » : `SCHED_OTHER`*, `SCHED_BATCH`, `SCHED_IDLE`
- 3 « temps réel » : `SCHED_FIFO`*, `SCHED_RR`*, `SCHED_DEADLINE` (classes de priorité strictes)
- Tous sont préemptifs

Page de man

- `sched(7)`, `chrt(1)`, `sched_setattr(2)`

*norme POSIX

Stratégies d'ordonnancement standard

File d'attente (non-préemptif)

- Premier arrivé, premier servi
- FIFO (*first in, first out*)

Avantages

- Facile à comprendre : file d'attente à la caisse
- Facile à implémenter
- Équitable ?

Implémentation

- Une file de processus prêts
- La tête de file est le prochain élu
- Les processus qui (re)deviennent prêts $\rightarrow$ en fin de file

Exercice et simulation (file)

processus	temps d'arrivée	temps de calcul
p1	0	9
p2	1	3
p3	2	3

Exercice et simulation (file)

processus	temps d'arrivée	temps de calcul
p1	0	9
p2	1	3
p3	2	3

processus	temps d'attente	temps total
p1	0	9
p2	8	11
p3	10	13
minimum	0	9
moyenne	6	11
maximum	10	13

Alternative

processus	temps d'arrivée	temps de calcul
p4	2	9
p5	1	3
p6	0	3

Alternative

processus	temps d'arrivée	temps de calcul
p4	2	9
p5	1	3
p6	0	3

Faites-le chez vous :)

Files d'attente + priorité + préemption

- Des niveaux distincts de priorité
  Par exemple de 1 (faible) à 99 (forte)
- Une file d'attente par niveau de priorité
- Priorité stricte : prioritaire = passer toujours devant

Avantages

- Facile à comprendre : file d'attente au parc d'attractions
- Facile à implémenter
- Permet un contrôle de l'utilisateur (politique)

SCHED_FIFO (Posix)

- `chrt --fifo 90 macommande`
- **Question** Et si `macommande` part en boucle infinie ?

Exercice et simulation (files prioritaires)

processus	temps d'arrivée	temps de calcul	priorité[†]
p1	0	9	2
p2	1	3	3
p3	2	3	1

[†]Grand = prioritaire, petit = pas prioritaire

Exercice et simulation (files prioritaires)

processus	temps d'arrivée	temps de calcul	priorité[†]
p1	0	9	2
p2	1	3	3
p3	2	3	1

processus	temps d'attente	temps total
p1	3	12
p2	0	3
p3	10	13
minimum	0	3
moyenne	4.3	9.3
maximum	10	13

[†]Grand = prioritaire, petit = pas prioritaire

Tourniquet

- File d'attente + quantum de temps
- RR (*Round-robin*)
- Quantum expiré $\rightarrow$ va à la fin de la file

Avantages

- Simple à comprendre : chacun son tour
  File d'attente au jeu gonflable
- Borne le temps que peut consommer un processus
- Équitable ?

Questions

- CPU-bound vs IO-bound, qui y gagne ?
- Et si un processus part en boucle infinie ?

Exercice et simulation (tourniquet)

processus	temps d'arrivée	temps de calcul	quantum
p1	0	9	4
p2	1	3	4
p3	2	3	4

Exercice et simulation (tourniquet)

processus	temps d'arrivée	temps de calcul	quantum
p1	0	9	4
p2	1	3	4
p3	2	3	4

processus	temps d'attente	temps total
p1	6	15
p2	3	6
p3	5	8
minimum	3	6
moyenne	4.7	9.7
maximum	6	15

Tourniquet + priorité

- Files d'attente + priorité + quantum de temps
- Quand le quantum est expiré, on va à la fin de sa file d'attente
- Mais on reste dans sa file d'attente

SCHED_RR (Posix)

- `chrt --rr 90 macommande`
- **Question.** Et si `macommande` part en boucle infinie ?
- Pages de man : `sched_rr_get_interval(2)` et `/proc/sys/kernel/sched_rr_timeslice_ms`

Problème des algos précédents

- Des décisions sont prises
- Mais indépendamment des caractéristiques des processus ou de leurs comportements

$\rightarrow$ Ce n'est qu'à postériori qu'on se désole (ou se félicite)

Solutions

- L'utilisateur choisit l'ordonnanceur en fonction de ce qui fonctionne bien pour son usage
- L'ordonnanceur prend en compte les caractéristiques et/ou le comportement

$\rightarrow$ Pourquoi pas les deux ?

Le plus court d'abord

- On choisit le plus court dans la file d'attente
- SJF (*shortest job first*)
- Hypothèse (forte) : on connait (estime) le temps de calcul
- Avantage : temps optimaux si arrivée en même temps

Version préemptive

- Temps **restant** plus court d'abord
- On perd la main si un plus court arrive
- Pas de quantum de temps
- **Question.** Pourquoi pas de quantum?

Exercice et simulation (plus court temps restant)

processus	temps d'arrivée	temps de calcul
p1	0	9
p2	1	3
p3	2	3

Exercice et simulation (plus court temps restant)

processus	temps d'arrivée	temps de calcul
p1	0	9
p2	1	3
p3	2	3

processus	temps d'attente	temps total
p1	6	15
p2	0	3
p3	2	5
minimum	0	3
moyenne	2.7	7.6
maximum	6	15

Problèmes du plus court

Connaître le temps

- Fourni par l'utilisateur
  - Estimation, maximum, catégorie de programme
- Analyse de l'historique
  - Qu'était le comportement du processus

Famine (*starvation*)

- Un gros processus n'a jamais la main
- Si de petits processus qui arrivent continuellement
- **Question** comment éliminer la famine ?

Linux CFS (*completely fair scheduler*)

- Depuis Linux 2.6.23 (2007)
- Objectifs : utilisation CPU et interactivité
- Pas de file d'attente
- Compte le temps réellement consommé
- Le temps d'attente (E-S) pris en compte (améliore l'interactivité)

Quantum non fixe

- On répartit le prochain bloc de temps
- Partage entre tous les processus
- La part de chacun dépend du temps CPU déjà consommé

Politiques de CFS

- `SCHED_OTHER` (appelé aussi `SCHED_NORMAL`) : le défaut
- `SCHED_BATCH` : comme `SCHED_OTHER` mais moins de préemptions
- `SCHED_IDLE` : plus faible que nice 19

Gentillesse (Posix)

Nice value

- Attribut par processus (ou thread)
- de -20 à +19 (sous Linux)
- Voir `nice(1)`, `renice(1)` et `nice(2)`

Principe

- Plus on est gentil plus on laisse sa place
- Privilèges nécessaires pour être pas gentil
- Priorité non stricte : c'est juste du bonus

Sous Linux (CFS)

nice affecte le calcul du « temps consommé » donc change la portion de CPU attribuée

Temps réel : différentes utilisations du terme

En direct

- Ça se passe maintenant
- L'horloge temps réel donne l'heure courante
- `top(1)` affiche en temps réel les processus

Soumis à des **contraintes** temporelles

- Le respect des échéances fait partie du cahier des charges
- Rater des échéances est un problème
- Exemple : un système de vidéo-conférence

Soumis à des contraintes temporelles **strictes** (dur)

- Rater **une** échéance est une **catastrophe**
- Exemple : un système de freins dans une voiture
- Principe de base : le **temps au pire** doit être contrôlé
- Quitte à dégrader les performances moyennes

Ordonnancement et temps réel

Préalablement connues

n processus, avec

- Période d'arrivée P_i
- Échéance
- Durée d'exécution au pire (coût) C_i

$\rightarrow$ du plus grand au plus petit

Garantie

Le système doit rejeter les processus qu'il ne peut pas ordonnancer sans respect des échéances

Exemples

Taux monotone (*rate-monotonic*)

- Nécessite des périodes connues
- Est élu celui qui a la période la plus courte (priorité constante)
- Test d'admissibilité: $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(\sqrt[n]{2} - 1)$

Prochaine échéance d'abord (*earliest deadline first*)

- Plus l'échéance est proche, plus sa priorité augmente
- Test d'admissibilité: $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$
- Sous Linux: `SCHED_DEADLINE`

Multi-processeur

- La même chose qu'en mono-processeur
- Mais en plus complexe

Problèmes

- Caches CPU et prédicteurs CPU
- Mémoire non uniforme (NUMA, *Non-Uniform Memory Access*)
- Hétérogénéité processeur (HMP, *heteregeneous multiprocessing*)

Quelques solutions

Affinité CPU naturelle

- L'ordonnanceur maintient le processus sur un même processeur
- Problème : déséquilibrage ; solution : rééquilibrer

Affinité CPU explicite

- Laisser l'utilisateur assigner des processeurs
- `taskset(1)`, `sched_setaffinity(2)`

Autres ordonnancements

Entrées-sorties disques

- Décider quelle donnée doit être écrite (ou lue) en premier
- `ionice(1)` et `ioprio_set(2)`

Paquets réseau

- Décider quels paquets sont routés en priorité
- `tc(1)` (*trafic control*)

Gestion de la performance

- Le SE contrôle voltage et niveau de veille des processeurs
- Décider quelle politique adopter (en fonction des processus)