Chapitre 11

Scheduling Linux 2.6

Politiques de scheduling

- Le scheduling de Linux fonctionne avec des quanta dynamiques
- Les processus sont ordonnés par priorité
  - La valeur indique au scheduleur quel processus executer en priorité
- Les priorités sont dynamiques
  - Les processus qui n'ont pas eu le CPU ont leur priorité augmentée, et réciproquement
- Linux reconnaît les processus temps réel
- Mais aucune notion de batch vs interactif
  - Décision prise avec une heuristique basée sur le comportement antérieur du processus

Appels systèmes

- *nice()* : change la priorité statique d'un processus
- *getpriority()* : obtient la propriété statique max d'un groupe de processus
- *setpriority()* :
- *sched_getscheduler()* : obtient la politique de scheduling
- *sched_setscheduler()*
- *sched_getparam()* : obtient la priorité temps réel
- *sched_setparam()*
- *sched_yield()* : Rend le processeur volontairement
- *sched_get_priority_min()* : Obtient la priorité temps réel min
- *sched_get_priority_max()*
- *sched_rr_get_interval()* : valeur du quantum
- *sched_getaffinity()* : masque d'affinité CPU
- *sched_setaffinity()*

Préemption

- Les processus peuvent être préemptés
- Quand un processus devient TASK_RUNNING
  - Le kernel vérifie si ça priorité est plus grande que le processus en exécution
  - Si oui, l'exécution du processus courant est interrompue et le scheduler est invoqué
- Quand un processus prend tout son quanta
  - TIF_NEED_RESCHED mis dans sa structure *thread_info*
  - Quand interruption horloge, le kernel invoque le scheduler si flag présent
- Avant 2.6
  - Un processus en mode kernel ne pouvait pas être préempté

Algorithme de scheduling

- Version simple (<2.6)
  - A chaque process switch le kernel parcourt la liste des processus prêts
  - Calcule leurs priorités
  - Et choisit le gagnant
  - Mais algorithme très coûteux si beaucoup de processus
- Version 2.6
  - Sélectionne les processus en temps constant
  - Une queue par CPU
  - Distingue mieux les interactifs des batch

Classes de scheduling

- Les processus sont schedulés suivant 3 classes
- SCHED_FIFO :
  - Processus temps réels en fifo
  - Quand le scheduler assigne le CPU à un processus, il ne change pas sa position dans la queue des processus prêts.
  - Si un autre processus de même classe et même priorité prêt, tant pis…
- SCHED_RR
  - Processus temps réel en round robin
  - Le descripteur de processus est mis en fin de liste une fois le cpu assigné
  - Permet à tous les processus temps réel de même priorité de partager le cpu
- SCHED_NORM
  - Processus conventionnel

Priorité statique

- Chaque processus conventionnel a une priorité statique (PS)
  - Utilisée par le scheduler pour ordonner les processus conventionnels entre eux
  - Valeur de 100 (haute priorité) à 139 (basse priorité)
- Un nouveau processus hérite de la priorité de son parent
- Quantum de base (QB)
  - Déterminé à partir de la priorité statique
  - Assigné à un processus qui a épuisé son quanta précédent
  - Si PS < 120, QB = (140-PS)*20
  - Si PS >=102, QB = (140-PS)*5
- Un processus de basse priorité aura un quantum faible

Priorité dynamique

- Chaque processus a en plus une priorité dynamique (PD) de 100 (plus haute) à 139 (plus basse)
- La priorité dynamique sert au scheduler à choisir le processus
  - PD=max(100,min(PS-bonus +5, 139))
- Bonus est une valeur entre 0 et 10
  - Une valeur < 5 est une pénalité qui baissera la PD
  - Valeur >=5 augmente la priorité dynamique
- La valeur du bonus dépend de l'histoire passée du processus
  - Son temps moyen de sommeil
- Temps moyen de sommeil
  - Mesuré en nanosecondes, jamais plus grand que 1 seconde
  - Dépend de l'état (TASK_INTERRUPTIBLE vs TASK_UNINTERRUPTIBLE)
  - Diminue quand le processus s'exécute
- Le temps moyen de sommeil sert à déterminer si le processus est batch ou interactif
  - Si PD =< 3xPS/4 + 28 alors interactif
  - Ce qui est équivalent à bonus – 5 >= PS/4 – 28
  - PS/4-28 est le delta interactif

Équivalence sommeil - bonus

Temps de sommeil moyen	Bonus
Entre 0 et 100ms	0
Entre 100 et 200ms	1
Entre 200 et 300ms	2
Entre 300 et 400ms	3
Entre 400 et 500ms	4
Entre 500 et 600ms	5
Entre 600 et 700ms	6
Entre 700 et 800ms	7
Entre 800 et 900ms	8
Entre 900 et 1000ms	9
1 seconde	10

Exemple de valeurs

Description	PS	Valeur Nice	Quantum de base	Delta Interactif	Limite sommeil
Highest Static Priority	100	-20	800ms	-3	299ms
High Static Priority	110	-10	600ms	-1	499ms
Default Static Priority	120	0	100ms	+2	799ms
Low Static Priority	130	10	50ms	+4	999ms
Lowest Static Priority	139	19	5ms	+6	1199ms

Empêcher un processus de monopoliser le CPU

- Dans certaines circonstances, un processus aura un quanta très long
  - Risque de monopoliser le CPU
  - Mais seulement pour les processus de même priorité
- On veut donc
  - Donner un long quanta à certains processus
  - Mais les interrompre sous certaines conditions pour laisser d'autres de même priorité
- Comment?
  - Ne les laisser qu'une fraction de leur quantum
- Toutes les 1ms une interruption est levée
  - Le kernel regarde le processus courant
  - Si temps restant > Granularité, alors on lui enlève le CPU
    - Granularité exprimée en ticks (1ms)

Équivalence sommeil - granularité

Temps de sommeil moyen	Granularité
Entre 0 et 100ms	5120
Entre 100 et 200ms	2560
Entre 200 et 300ms	1280
Entre 300 et 400ms	640
Entre 400 et 500ms	320
Entre 500 et 600ms	160
Entre 600 et 700ms	80
Entre 700 et 800ms	40
Entre 800 et 900ms	20
Entre 900 et 1000ms	10
1 seconde	10

Processus actifs et expirés

- Un processus conventionnel de haute priorité ne devrait pas empêcher ceux de basse priorité de tourner
- Quand un processus fini son quantum, il peut être remplacé par un processus de plus basse priorité qui n'a pas fini le siens
- Le scheduler maintient 2 ensembles de processus
  - Processus Actifs : ils n'ont pas encore fini leur quantum
  - Processus expirés : ont épuisé leur quantum et ne peuvent pas s'exécuter tant qu'il reste des processus actifs
- Mais compliqué en pratique
  - Un processus batch qui finit son quantum devient expiré
  - Un processus interactif qui finit son quantum reste souvent actif
    - Sauf si un processus expiré attend depuis très longtemps
    - Ou un processus expiré a une priorité statique plus élevée

Scheduling temps réel

- Chaque processus temps réel a une priorité temps réel de 1 (plus élevée) à 99
- Le scheduler favorise toujours le processus temps réel de plus haute priorité
  - Aucun autre ne peut s'exécuter
- Les processus temps réel sont toujours considérés actifs
- Un processus temps réel est remplacé par un autre processus si
  - Il est préempté par un autre de plus haute priorité temps réel
  - Il effectue une opération bloquante
  - Il est stoppé ou tué
  - Il rend volontairement le CPU
  - Il est SCHED_RR et a fini son quantum

La structure *runqueue*

- La structure principale pour le scheduling en 2.6 est la *runqueue*
- Chaque CPU en possède une
- Chaque processus dans le système appartient à une seule queue
  - Il ne peut donc être exécuté que sur un CPU particulier
  - Mais migration entre queues possibles
- La *runqueue* contient 2 tableaux
  - Chacun contient 140 listes doublement chaînées de processus (1 par priorité)
  - 1 tableau contient les processus actifs, un autre les expirés
    - La *runqueue* a un pointeur *active* sur la queue des actifs, et un *expired* sur l'autre
  - Le rôle des tableaux change par une permutation des pointeurs

Équilibrage de *runqueues* en environnement multiprocesseurs

- 3 types de machines multiprocesseurs
  - Architecture classique
    - tous les CPUs partagent la même mémoire
  - Hyper-threading
    - Un CPU HT exécute plusieurs threads en même temps
    - Possède plusieurs copies des registres
    - Permet de continuer à exécuter un thread quand un autre est en attente mémoire
    - Vue par Linux comme plusieurs CPUs logiques
  - NUMA
    - Non-Uniform Memory Access
    - Des CPUs et de la RAM sont groupés en nœuds
    - L'accès à la mémoire du même nœud est très rapide, contrairement à la distante

Équilibrage de *runqueues* en environnement multiprocesseurs

- Un processus prêt est dans la *runqueue* d'un processeur
  - Le choix est fait lors du passage bloqué à prêt
- Si 2 processus CPU-bound se trouvent dans la même *runqueue*, alors risque de mauvaise utilisation de l'autre CPU
- Le kernel vérifie périodiquement que les *runqueues* sont équilibrées
  - Déplacement de processus si nécessaire
  - Mais doit tenir compte de la topologie
- Depuis Linux 2.6.7 introduction des *scheduling domains*

Scheduling Domains

- Un *scheduling domain* est un ensemble de CPUs dont la charge doit être équilibrée
  - Approche hiérarchique
  - Un domaine englobe tous les CPUs, puis des domaines fils englobent des groupes plus petits
- Chaque domaine est partitionné en groupes
  - L'équilibrage se fait entres groupes du même domaine