Ordonnancement et Synchronisation Avancée dans Linux

Algorithme d'Ordonancement CPU de Linux

- Le temps est divisé en *époques*
  - Allocation d'une nouvelle tranche de temps (quantum)
    - Au début de chaque époque
    - A chaque processus : Quantum[Pi]
  - Quantum[Pi]
    - maximum de temps CPU que Pi peut obtenir
      - durant cette époque
    - Interruption d'horloge
      - Quantum[Pcourant] --
    - Quand Quantum[Pi] = 0
      - Préemption
      - Processus remplacé par un autre processus dans l'état prêt
  - Fin de l'époque ?
    - Quand tous les processus ont épuisé leur tranche
      - Tous = ceux dans l'état prêt

Valeur des Quanta Attribuée par l'Ordonanceur

- Quantum de base
  - Hérité du parent
  - Reconduit à chaque époque
  - Modifiable par nice() et setpriority()
- Quantum de base du parent ?
  - Hérité du grand-parent ...
  - ...
  - Init (processus 1)
  - Swapper (processus 0)
    - Quantum initialisé à DEF_COUNTER

    #define DEF_COUNTER (10 * HZ / 100)
    - 100-105 ms

Ordonnancement Temps Reel dans Linux

- Tout est prévu ...
  - Mais le noyau n'est pas interruptible !
- Trois classes d'ordonancement
  - SCHED_FIFO
    - Priorité : statique + temps reel
    - FIFO : le premier arrivé est **toujours** le premier servi
      - Pas d'alternance en cas d'égalité de priorité
  - SCHED_RR
    - Priorité : statique + temps reel
    - Alternance en processus de même niveau
  - SCHED_OTHER
    - Classe normale
    - Temps partagé + priorité dynamique classique de Unix
- Appels systèmes spécifiques Linux

Implémentation : Structures de Données

- Champs de la struct task_struct
  - need_resched : mis à 1 quand le quantum est épuisé
    - Provoque changement de contexte (ou changement d'époque)
  - policy : SCHED_FIFO, SCHED_RR ou SCHED_OTHER
  - rt_priority : priorité statique temps reel
  - counter : valeur courante du quantum (Quantum[Pi])
  - nice : ajuste la longueur du quantum (entre -20 et +19)
  - cpu_allowed : bitmap des CPU possibles
  - cpus_runnable : bitmap indiquant le CPU actuel
    - 111....111 : aucun
    - 000...010 : CPU 1 (car bit1 = 1)
  - processor
    - processeur courant *ou du dernier processeur utilisé*

La fonction schedule()

- Déclenche le changement de contexte
  - Soit explicitement demandé par un processus
    - [interruptible_]sleep_on, ...
  - Soit implicitement lors du retour en mode utilisateur
    - quand need_resched = 1
- Schéma classique d'invocation explicite
  1. Insertion dans la bonne file (par ex: wait_queue)
  2. Changement état pour TASK_[UN]INTERRUPTIBLE
  3. Appel de schedule()
  4. (réveil) Vérifier que ressource est disponible
     => Si non dispo : retour en 2.
  5. Retrait du processus de la file

Sélection du Processus Elu

- La priorité ne fait pas tout...
  - Evaluation de la **qualité** de chaque processus
- La fonction goodness(Pi, this_cpu, prev_mm)
  - Appliquée à chaque processus Pi prêt
  - Retourne une mesure de qualité du processus Pi
    - Dépend du cpu
    - Dépend de la configuration mémoire du processus précédent (prev_mm)
  - Valeurs retournées
    - -1 : un processus qui a demandé à laisser la main (yield)
    - 0 : un processus qui a épuisé son quantum
    - >= 1000 : un processus temps reel
    - 2 à 77 :
      - Pi->counter +20 – Pi->nice
      - Bonus : +15 si p->processor = this_cpu et +1 si mm=prev_mm

Horloge Système

- Horloge Système
  - Horloge logicielle : compteur `jiffies (u. long int)`
    - Initialisée à 0
    - Incrémentée à chaque interruption d'horloge
  - Précision dépendant de la valeur de `HZ`
    - Macro définie dans <linux/param.h>
      - Nombre d'interruption d'horloge par seconde
      - Généralement fixée à 100 mais variable d'une plate-forme à l'autre
    - Assez fiable, mais pas totalement
      - Si l'interruption est masquée trop longtemps
    - Précision assez faible
      - Besoin de précision plus élevée ? Attente active
    - Raisonnable pour programmer des attentes passives
    - Risque de rebouclage faible (16 mois pour `HZ`=100)

Attente Passive Basée sur Horloge Système

- Mettre le processus en attente jusqu'à expiration d'un délai
  - Rappel : attente passive = sans consommation de CPU
    - Donner la main à un autre processus en attendant
    - Effets de bord positifs sur les performances globale, la priorité, ...
- Deux cas de figure :
  - Attente bornée d'un évènement
    - `[interruptible_]sleep_on_timeout(wq,delay)`
      - Attendre jusqu'à jiffies+delay
  - Attente de la seule expiration du délai
    - `set_current_state(TASK_INTERRUPTIBLE)`
    - `schedule_timeout(delay)`
    - Utiliser une *wait queue* fonctionne aussi !

Horloge de Précision

- Horloge basée sur registre spécifique
  - Registre incrémenté à chaque cycle
  - Fortement dépendant de l'architecture
    - Exemple x86 (<asm/msr.h>) :
      - `rdtsc(low,high)` : 64 bits (2 x32)
      - `rdtscl(low)` : 32 bits (32 bits poids faible)
    - Risques de rebouclages fréquent
  - Version plus portable
    - `cycle_t` **`get_cycles`**`(void)`
      - définie dans <linux/timex.h>
      - (idem `rdtscl` sur x86)

Renvoie 0 si non supporté

- Attente active (calibrée par calcul des BogoMips)
  - `void` **`udelay`**`(usec)`
  - `void` **`mdelay`**`(msec)` (msec X udelay(1000))

Exécution Différée

- Pourquoi l'exécution différée ?
  - Typiquement pour les tâches moins urgentes d'un traitant d'interruption
- Linux propose 4 mécanismes d'exécution différée
  - Les interruptions logicielles (*softirqs*)
    - Création statique (boot)
    - Exécution concurrente sur plusieurs CPU
  - Les *Tasklets*
    - Création dynamique
    - Exécution concurrentes si type différents
  - Les *BottomHalves*
    - Statiques
    - Non concurrentes
  - Les Task Queues

Architecture Globale Mécanismes d'Exécution Différée

- Hiérarchie
  - Certaines Task Queues sont construites à partir de BH
  - Les Bottom Halves sont construites à partir des Tasklets
  - Les Tasklets sont construits à partir des SoftsIRQs
- Mode opératoire
  - Initialisation
    - Généralement lors de l'init du noyau
  - Activation
    - La fonction différée est mise en attente de traitement
  - Masquage
    - Désactivation sélective d'une fonction différée
  - Exécution
    - Exécution de toutes les fonction d'un même type
    - A certains moment bien précis

Softs IRQs

- Mécanisme de bas niveau
- 4 types, classées par niveau de priorité (0 = Max)
  0. `HI_SOFTIRQ`
     - tasklets et bottom halves de priorité haute
  1. `NET_TX_SOFTIRQ`
     - Transmission des paquets vers les cartes réseau
  2. `NET_RX_SOFTIRQ`
     - Reception des paquets en provenance des cartes réseau
  3. `TASKLET_SOFTIRQ`
     - traitement des tasklets
- Distribution des SoftIRQs sur les différents processeurs
  - Une thread noyau par CPU

Tasklets

- Construites au dessus de deux SoftsIRQs
  - `HI_SOFTIRQ`
  - `TASKLET_SOFTIRQ`
- Chaque Tasklet contient sa propre fonction
- Mode d'utilisation
  - Init « manuelle »
    - Créer `tasklet_struct`
    - Invoquer `tasklet_init(&ts, &func, data(UL))`
  - Macros d'init
    - `DECLARE_TASKLET(name,function,data)`
    - Variante `DECLARE_TASKLET_DISABLED`

Tasklets (2)

- Programmer l'exécution d'une TL :
  - `tasklet_schedule()`
  - `tasklet_hi_schedule()`
- Désactivation :
  - Incrémenter compteur `count` du descripteur
    - `tasklet_disable()`
    - `tasklet_disable_nosync()`
- Réactivation
  - Décrémenter compteur : `tasklet_enable()`
- Suppression
  - Certaines taslets se reprogramment indéfiniment...
  - `tasklet_kill()`

Bottom Halves

- Tasklet de haute priorité
  - Ne peut pas être exécutée de façon concurrente avec une autre BH (qq soit type et nombre de CPUs)
  - Une 15aines existent, mais principalement 4 utilisées :
    - TIMER_BH :
      - activée par interruption horloge
      - Exécution dès retour du traitant
    - TQUEUE_BH :
      - Exécution de la file des Task Queues correspondante
      - Activée à chaque interruption d'horloge
    - SERIAL_BH : port série
    - IMMEDIATE_BH
      - au plus tôt
      - Exécute la file des Task Queues immédiates

Task Queues

- Prédéfinies
  - Certaines sont déclenchées par une BH
    - tq_immediate (IMMEDIATE_BH)
    - tq_timer (TQUEUE_BH)
    - Insertion par `queue_task(name,file_tq)`
  - Mais pas toutes :
    - tq_context :
      - exécutée par thread noyau keventd
        - Exécution dans le contexte d'un processus
        - Peut bloquer (alloc mémoire GFP_KERNEL ...)
      - Insertion de la tâche par `schedule_task()`

Timers du Noyau

- Même principe que *Task Queues* mais à une date programmée
  - Déclarés à l'aide d'une structure (`<linux/timer.h>`)

```
struct timer_list {
    struct list_head list;
    unsigned long expires;
    unsigned long data;
    void (*function)(unsigned long);
};
```

jiffies

  - API
    - init_timer(tl) : init struct
    - add_timer(tl) : activation
    - mod_timer(tl,expires)
    - del_timer(tl)
    - del_timer_sync(tl) : retourne avec garantie que le timer ne s'exécute plus sur aucun CPU

Synchronisation

- Opération atomique :
  - lecture/modification/écriture sur un compteur
- Barrière mémoire
  - Garantit que les instructions placées après la barrière ne seront pas exécutées avant (optimisation compilateur)
- Spin Locks
  - Verrou avec attente active
- Spins locks Read/Write
  - Lectures multiples ou écriture exclusive
- Sémaphores
- Completions
- Désarmement interruptions

Opérations Atomiques

- Type `atomic_t` (compteur 24bits)
- Accesseurs/Opérateurs
  - atomic_read(&x)
  - atomic_set(&x,i)
  - atomic_add(i,&x)
  - atomic_sub(i,&x)
  - atomic_sub_and_test(i,&x)
  - atomic_inc(&x)
  - atomic_dec(&x)
  - atomic_dec_and_test(&x)
  - atomic_inc_and_test(&x)
  - atomic_add_negative(i,&x)

Opération Atomiques sur Bits

- test_bit(nr,addr)
- set_bit(nr,addr)
- clear_bit(nr,addr)
- change_bit(nr,addr)
- test_and_set_bit(nr,addr)
- test_and_clear_bit(nr,addr)
- test_and_change_bit(nr,addr)
- atomic_clear_mask(mask,addr)
- atomic_set_mask(mask,addr)

Barrières Mémoire

- Barrière pour uni- et multi-pro
  - mb()
  - rmb() : barrière en lecture mémoire
  - wmb() : barrière en écriture mémoire
- Barrière pour multi-pro seulement
  - smp_mb()
  - smp_rmb()
  - smp_wmb()

Spinlocks

- Attente active des CPU concurrents
  - Ne font rien sur uni-pro
    - excepté spin_trylock() qui retrourne 1
  - Spin locks simples
    - type `spinlock_t`
    - spin_lock_init(s) : initialise à 1 (ouvert)
    - spin_lock(s) : attend et vérrouille
    - spin_unlock(s) : déverrouille
    - spin_unlock_wait(s) : attend que le verrou soit ouvert
    - spin_is_locked(s) : test sans attendre
    - spin_trylock(s) : essaie de verrouiller et retourne 1 si succès, 0 sinon

Spinlock de Lecture/Ecriture

- Autoriser de multiples lectures ou 1 (seule) écriture
  - Type `rwlock_t`
  - `rwlock_init(rw)`
  - `read_lock(rw)`
  - `read_unlock(rw)`
  - `write_lock(rw)`
  - `write_unlock(rw)`
- Version optimisée pour éviter les cache-miss en cas de nombreuses lectures (« big reader lock »)
  - type __brlock_array
  - br_read_lock()/br_read_unlock()
  - br_write_lock()/br_write_unlock()

Sémaphores Noyau

- Sémaphores normaux :
  - Type `struct semaphore`
  - init_MUTEX(s) : val=1 (libre)
  - init_MUTEX_LOCKED(s) : val=0 (pris)
  - up(s) : relâchement ou val++
  - down(s) : val-- ou dormir sur wq (UNINTERRUPTIBLE)
  - down_interruptible(s)
- Permettent lectures multiples ou lecture unique
  - Type struct rw_semaphore
  - init_rwsem(rws)
  - down_read()/down_write()
  - up_read()/up_write()

Completions

- Corrige une « subtle race condition » sur SMP
  - Concurrence entre up() et down() mal gérée sur SMP
  - Type struct completion
  - complete(&c) : libère
  - wait_for_completion(&c) : attend libération puis verrouille

Désarmement des Interruptions

- Interruptions hard sur Uni-pro
  - __cli() : désactive int
  - __sti() : réactive int
- Interruptions hard sur Multi-pro
  - cli() : désactive
  - sti() : réactive
- Désarmement local (CPU courant) des *SoftIrqs*
  - local_bh_disable()
  - local_bh_enable()