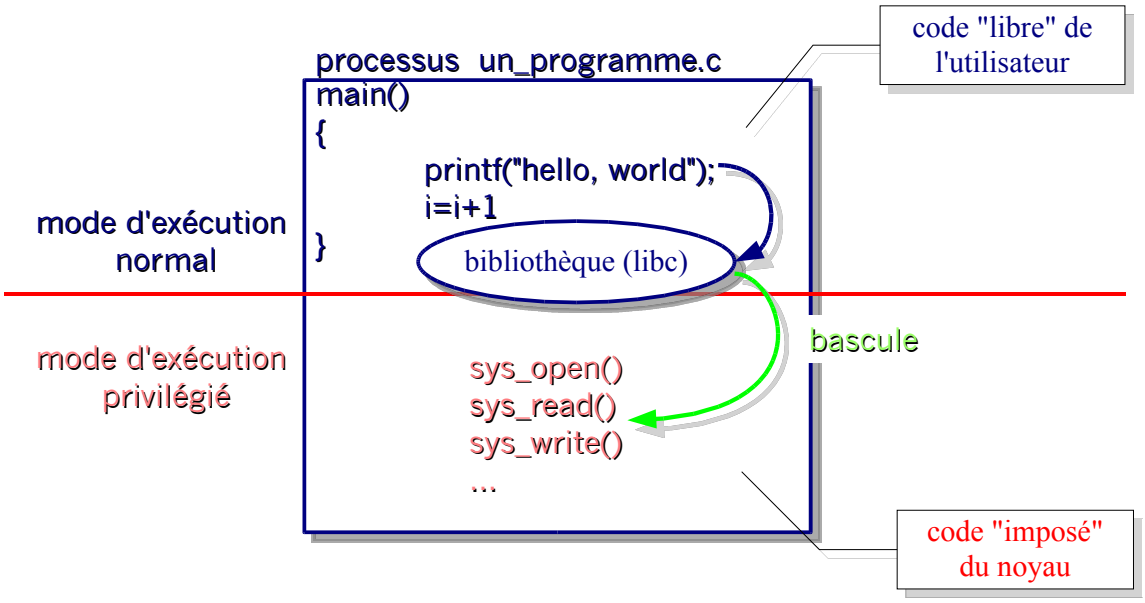
Un cas particulier de programmation concurrente:
Le noyau Linux (v. 2.4)

Modes d'Exécution (du processeur)

- 2 modes d'exécution sont indispensables pour Unix
  - **Mode Utilisateur** (normal)
    - Mode d'exécution pour les programmes utilisateurs
    - Protection forte contre les erreurs de programmation
  - **Mode Noyau** (privilégié)
    - Mode d'exécution du programme noyau
    - Instructions machine supplémentaires
    - Registres supplémentaires
      - Gestion de la mémoire virtuelle
    - Accès à l'***espace mémoire utilisateur***
      - Mémoire (virtuelle) du processus courant
    - Accès à l'***espace mémoire système***

Illustration : Passage en Mode Noyau



La Pile

- Utilisée par la plupart des langages
  - En particulier le langage C
    - Donc le noyau
  - Quelques exceptions : Fortran77, ...
- Chaque processus en possède (au moins) une
  - Elle permet de mémoriser les paramètres d'appel des fonctions
    - Autorise les appels récursifs
  - Lorsque l'exécution de la fonction se termine
    - Retrait des paramètres d'appel
    - Empilement de la valeur de retour

La Pile Système

- Où se trouve la pile ?
  - A l'adresse indiquée par un registre
- Quand un processus « bascule » en mode noyau
  - Le mot d'état du processeur est remplacé
    - Sauvegarde registres (instruction, pile, ...)
    - Chargement registres pour le mode système **du processus**
    - Une pile système **différente** pour chaque processus
  - MAIS
    - En mode noyau tous les processus **partagent le même espace mémoire**
      - Certaines zones (la plus grande partie) sont vues identiques
      - Certaines sont vues différemment (la pile, le contexte propre)

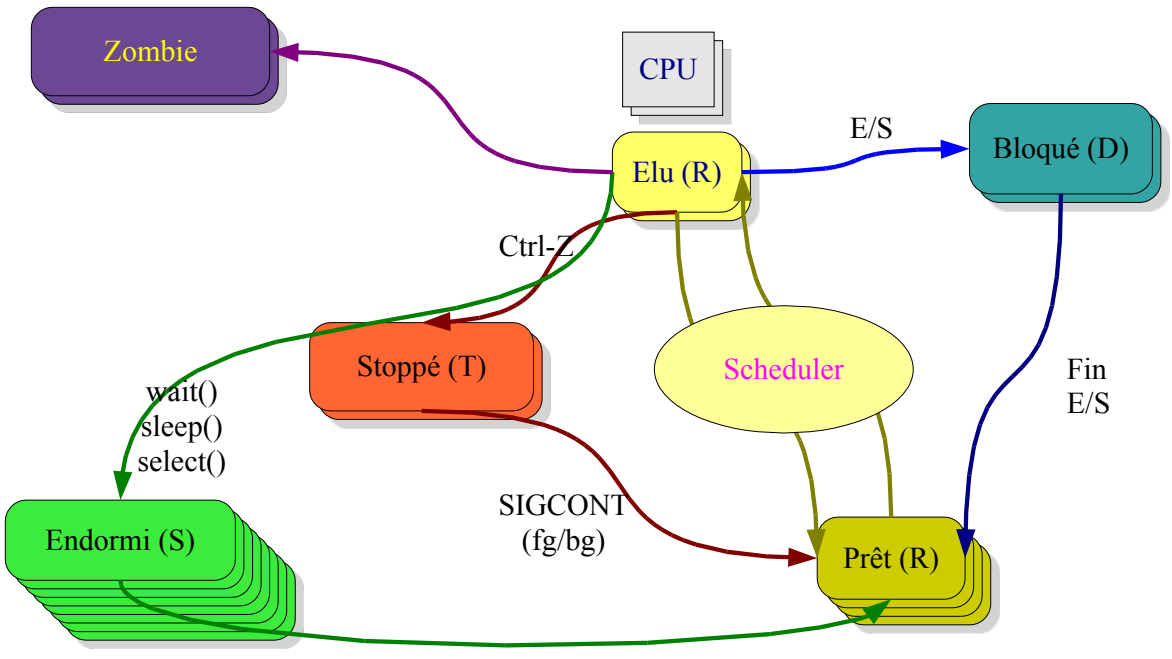
Zoologie des Processus

- Processus = Instance d'un programme en cours d'exec
  - Pour le système, un processus c'est
    - Un état d'exécution
    - Un contexte
    - Une consommation (détention) de ressources
  - Variantes autour de la notion de processus
    - « Lourd » : notion classique de processus Unix
      - POSIX : Création par fork() = duplication du contexte
      - Séparation (et protection) de l'espace d'adressage propre
    - « Leger » : notion plus récente de « thread »
      - POSIX : Création par pthread_create() = partage du contexte
      - Hébergés dans le contexte d'un processus lourd (thread main)
      - Création plus rapide
      - Partage des données
      - MAIS : problèmes de concurrence...

Etats d'un Processus : Vue Utilisateur

- Champ STAT de la commande `ps -x (interne noyau)`
  - `R : TASK_RUNNING`
    - Prêt à s'exécuter ou en cours d'exécution
  - `S : TASK_INTERRUPTIBLE`
    - Endormi en attente interruptible (non exclusive) d'un événement
  - `D : TASK_UNINTERRUPTIBLE`
    - Endormi en attente non interruptible (exclusive) d'un evènement (typiquement une E/S)
  - `Z : TASK_ZOMBIE`
  - `T : TASK_STOPPED`
    - Arrêté ou Tracé
  - `W` : Pas de page résidente
  - `N` : Gentillesse positive
  - + Un état supplémentaire implicite : processus élu

Graphe des Transitions entre Etats



Quels Type de Processus Supportés par Linux ?

- Threads de niveau noyau
  - Création par l'A.S. `clone(fn,arg,flags)`
    - `fork() et vfork() = dérivés de clone()`
    - flags = Choix de ce qui doit être partagé (ou cloné)
      - CLONE_VM : espace mémoire
      - CLONE_FS : root, cwd, umask
      - CLONE_FILES : fichiers ouverts
      - CLONE_PARENT : même parent
      - CLONE_PID : même pid (uniquement si parent = 0, lors du boot)
      - CLONE_PTRACE : « ptracé » comme parent
      - CLONE_SIGHAND : même traitants interrupt.
      - CLONE_THREAD : même groupe que parent (= thread POSIX)
      - CLONE_SIGNAL : CLONE_THREAD + CLOSE_SIGHAND
      - CLONE_VFORK : bloque parent jusqu'à more ou exec du fils
    - flags = 0 : processus lourd

Threads Internes du Noyau

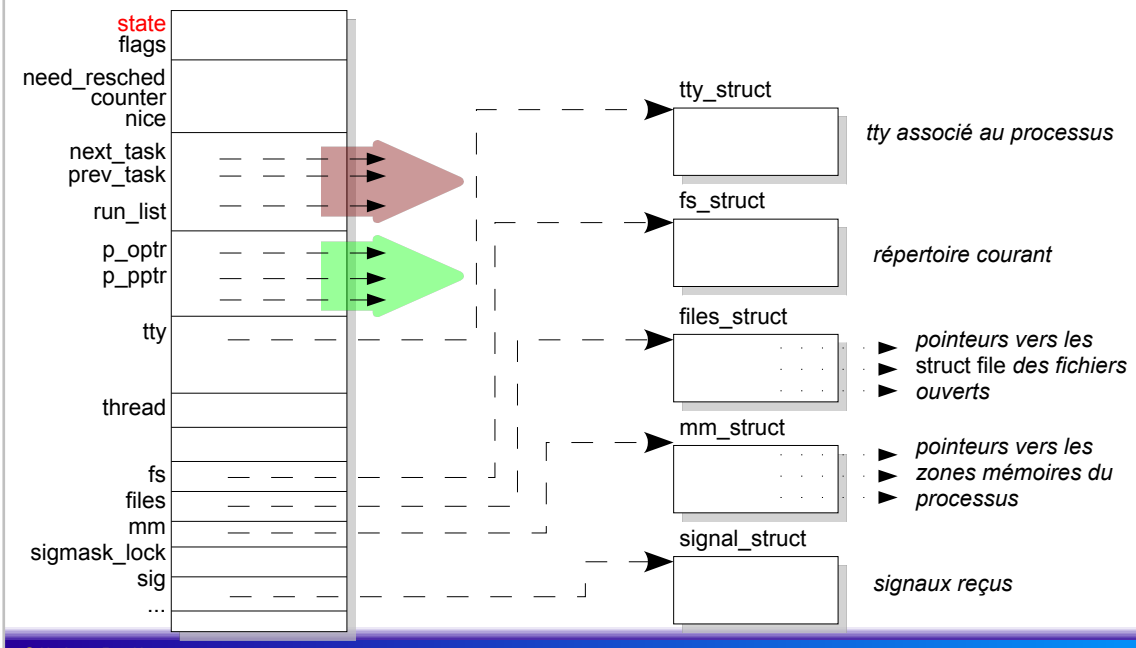
- Threads particulières :
  - Ne basculent jamais en mode utilisateur
  - Plus légères
    - Pas besoin de les encombrer de ce qui concerne le mode utilisateur
  - Utiles pour traîter certaines travaux en tâche de fond
    - `keventd` : execute ***task queue*** qt_context
    - `kapm` : événement liés à APM
    - `kswapd` : collecte mémoire
    - `kflushd/bdflush` : évacue les pages « sales » du cache disque pour récupérer de la mémoire
    - `kupdated` : evacue aussi page sales, mais pour éviter trop de pertes en cas de crash
    - `ksoftirqd` : exécute les ***tasklets*** (un par CPU)
- Création : `int` **`kernel_thread`**`(fn, arg,flags)`

Deux Thread Noyau Particulières : 0 et 1

- Thread de pid 0 :
  - Seule thread créée spontanément
    - Dans `start_kernel()`
  - Son rôle :
    - Lancer thread 1 (en utilisant `kernel_thread()`)
    - Exécuter `cpu_idle()` forever ...
      - Attente d'une interruption
      - Choisie par scheduler quand aucune autre thread dans l'état TASK_RUNNING
- Thread de pid 1 :
  - Exécute `init()`
    - Fin init noyau
  - Charge exécutable init par `execve()`
    - Devient processus normal

Contexte d'un Processus

- Descripteur : `struct task_struct` (<linux/sched.h>)

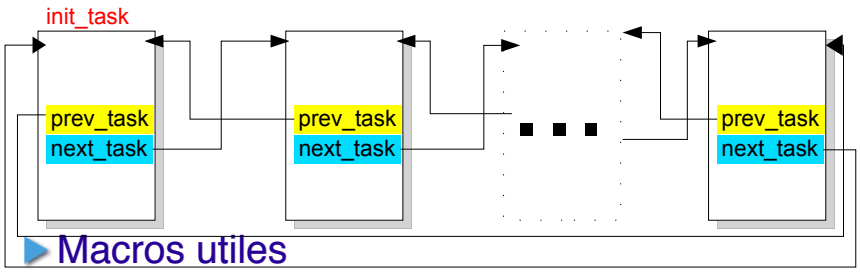


Identification des Processus

- Processus courant = `current (task_struct)`
  - Macro : masque 13 bits du pointeur de pile (reg `%esp`)
- Tous les processus (lourds ou légers) ont un descripteur
  - Identification non ambiguë à partir de l'adresse linéaire (32 bits) du descripteur (adresse `struct task_struct`)
  - Identification classique Unix : champ `pid`
    - cas général : getpid() <=> `current->pid`
    - cas particulier : threads type POSIX
      - Toutes les threads qui partagent le même contexte doivent répondre au même pid
      - Introduction d'un identifiant de groupe : `tgid`
        - `= pid` du premier processus (thread) du groupe
        - Identique pour toutes les threads du groupe
        - getpid() retourne `current->tgid`

Liste Chaînée des Processus

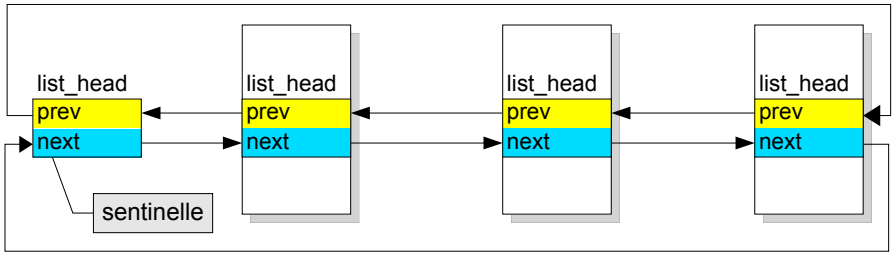
- Liste circulaire doubelement chaînée
  - Pointeurs `prev_task` et `next_task` du descripteur
  - Tête de liste : `init_task` (processus init, pid=1)



  - Macros utiles
    - `SET_LINKS/REMOVE_LINKS`
    - `for_each_task(p)`
      - `for (p=&init_task ;(p=p->next_task)!=&init_task ;)`

A Propos de Listes Doublement Chaînées...

- Structure récurrente dans le noyau
  - Solution générique : la `struct list_head`



  - Attention : ne pointe pas directement au début de la structure chaînée
    - Macros classiques : list_add(new,prev), list_add_tail(new,head), list_del(entry), list_empty(head)
    - Récupération adr struct : list_entry(ptr,type,field)
    - Boucle : list_for_each(ptr,head)

Liste des Processus Prêts (TASK_RUNNING)

- Champ `run_list` du descripteur de processus
  - type `list_head`
- Macros spécifiques
  - `add_to_runqueue(&task_struct)`
    - Insertion en début de liste
  - `del_from_runqueue(&task_struct)`
  - `move_first_runqueue(&task_struct)`
  - `move_last_runqueue(&task_struct)`
  - `bool task_on_runqueue(&task_struct)`
  - `wake_up_process(&task struct)`
    - Place le processus dans l'état `TASK_RUNNING`
    - invoque `add_to_runqueue()`

Relations entre Processus

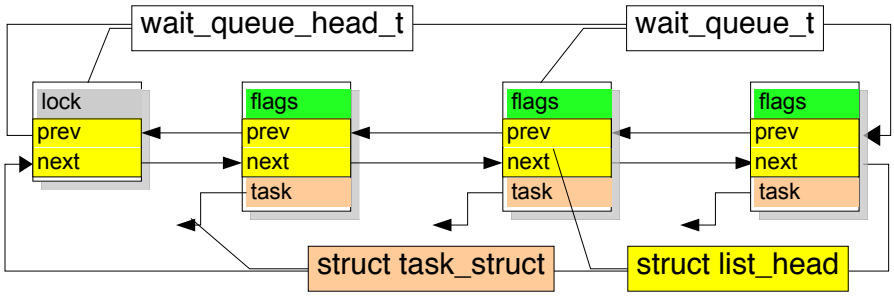
- Le descripteur du processus pointe vers plusieurs processus
  - `p_opptr` : parent original
    - Celui qui a créé le processus
  - `p_pptr` : parent courant
    - Différent si le processus est tracé
  - `p_cptr` : Dernier fils créé
  - `p_ysptr` : frère cadet suivant
    - Créé juste après par le même père
  - `p_osptr` : frère aîné précédent
    - Créé juste avant par le même père

Organisation des Files de Processus

- Les processus peuvent se trouver dans différentes files en fonction de leur état
  - ***process list*** : tous les processus
    - champs `prev_task` et `next_task` de `task_struct`
  - `TASK_RUNNING` : champ run_list
  - `TASK_STOPPED` et `TASK_ZOMBIE` : pas d'autre chaînage
  - `TASK_INTERRUPTIBLE` et `TASK_UNINTERRUPTIBLE`
    - Files trop nombreuses pour apparaître directement dans la `struct task_struct`
    - Utilisation de ***wait queues***
      - Correspondent chacune à l'attente d'un événement particulier

Wait Queues

- Structures basées sur la `struct list_head`



- flags = 1 : processus « exclusif »
  - Réveil = un seul à la fois
  - A défaut réveil des processus qui ont flags = 0
- flags = 0 : processus non « exclusif »
  - Tous les processus sont réveillés ensemble
    - Compétition possible ...

Utilisation des *Wait Queues*

- Déclaration
  - Statique : `DECLARE_WAIT_QUEUE_HEAD(wq_head)`
  - Dynamique : `init_waitqueue_head(wq_head)`
- Insertion
  - `add_wait_queue(wq_head,wq_elem)`
  - `add_wait_queue_exclusive(wq_head,wq_elem)`
- File vide ?
  - `waitqueue_active(wq_head)`
- Insertion du processus avec mise en sommeil
  - `[interruptible_]sleep_on[_timeout](wq_head[,to])`
  - `wait_event[_interruptible](wq_head,condition)`
- Extraction et réveil d'un processus
  - `wake_up[_interruptible][_sync][_nr|_all](wq_head[,nr])`

regarder priorité pour élection éventuelle

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 compiltateur)
- 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) : relâcment 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 « subttile 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()