Threads

(incomplet)

MI074 -7

Plan

- Définition d'un thread
- Définition de l'ordonnanceur de threads
- API thread
- API ordonnanceur
- thread IDLE et thread MAIN
- Creation des threads initiaux
- Mise en oeuvre en TME

Notion d'un Thread

Un Thread est un fil d’exécution d’un programme

Chaque Thread possède principalement :

- Un contexte d’exécution
  - état des registres du processeur
- deux piles.
  - pile système
  - *pile utilisateur (pas pour le moment)*

Principaux intérêts :

- Création et gestion rapide (vs processus).
- Partage des ressources par défaut.
- Communication entre les threads simple via la mémoire (les variables globales).
- Déploiement efficace de l’application sur des architectures multi-processeurs.

Thread

La structure thread_s qui le définit :

- verrou pour l'accès exclusif
- type de thread
- état du thread
- numéro de CPU
- valeur de retour du thread
- chainon de la liste des threads dans le même état
- chainon de la liste des threads vivants
- contexte

Dans un premier temps on ne gère pas :

- le mode user
- la synchronisation sur la terminaison d'un thread
- la gestion du temps

Thread

```
struct thread_s {
  spinlock_t lock;
  thread_type_t type;
  thread_state_t state;
  uint_t cpuid;
  struct sched_s *sched;
  struct list_entry list;
  struct list_entry rope;
  void * exit_value;
  struct cpu_context_s pws;
};
```

```
typedef enum
{
  PTHREAD,
  KTHREAD,
  TH_IDLE
} thread_type_t;

typedef enum
{
   S_CREATE,
   S_READY,
   S_USR,
   S_KERNEL,
   S_WAIT,
   S_ZOMBIE,
   S_DEAD
} thread_state_t;
```

Les états d'un thread

S_CREATE
: Le thread vient d'être créé mais n'a pas encore été chargé sur le processeur (pas de contexte)

S_USR
: Le thread est en train d'être exécuté en mode user

S_KERNEL
: Le thread est en train d'être exécuté en mode kernel

S_READY
: Le thread est prêt à être exécuté

S_WAIT
: Le thread est en attente de quelque-chose

S_ZOMBIE
: Le thread est mort et un autre thread attend cette info.

S_DEAD
: Le thread est vraiment mort, on peut récupérer l'espace

Ordonnancement des Threads

- L'ordonnancement c'est quand le système **décide** de la place d'un thread dans une liste de thread.
- On classe les **décisions** en fonction du temps qui sépare l'instant de la décision de placement du thread et l'instant d'exécution du thread.
- 3 décisions d'ordonnancement:
  - long terme (placement initial)
  - à la création d'un thread, le système choisit en fonction de critères globaux
  - court terme (élection)
  - quand on décide quel thread s'exécute sur un processeur parmi les threads prêts.
  - moyen terme (planification)
  - quand on organise les tâches prêtes ou en attente.
    De run à wait, de wait à ready, de wait à swap.
    équilibrage de la charge des processeurs

Scheduler : objectifs

- Le système partitionne l'ensemble des Threads de l' application en sous-ensembles dans le but de les ordonnancer (les trier)

- Il y a autant de sous-ensembles que de processeurs.
  Un thread est affecté à un seul processeur tout au long de sa vie, le noyau ne permet pas la migration des tâches, ou la répartition dynamique de la charge du système.

- Il existe une structure d'ordonnancement pour chaque sous-ensemble, responsable de l'ordonnancement de ses Threads selon sa propre politique d'ordonnancement.

Scheduler : structures

La structure sched_s (par processeur)

- verrou pour l'accès exclusif
- pointeur sur le thread courant
- pointeur sur le thread idle
- racine de la liste des threads prêts
- racine de la liste des threads morts
- pointeur void vers une structure dépendante du matériel

Le système définit une structure qui regroupe l'ensemble des structures sched_s.
Ici c'est d'un tableau global indexé par le numéro de processeur.

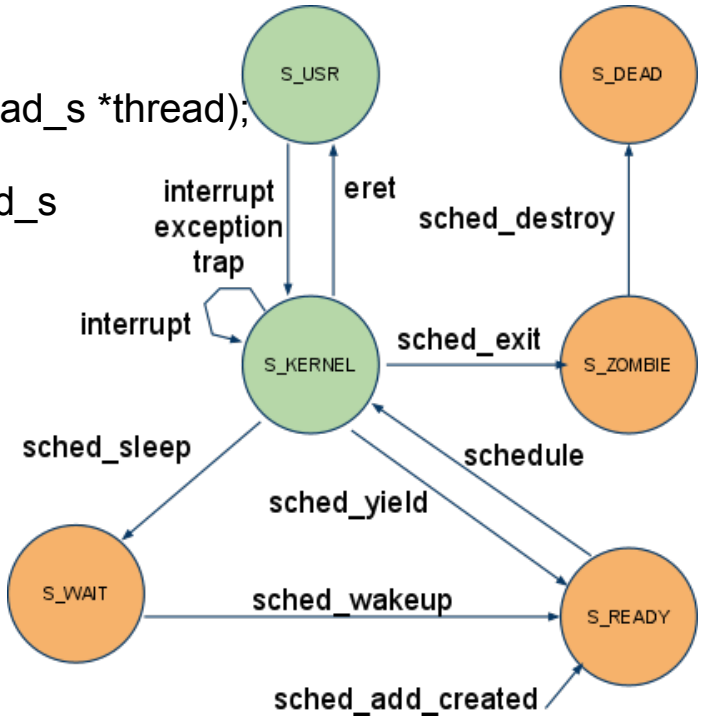
Scheduler : détails structures

```
struct sched_s
{
  spinlock_t lock;
  struct thread_s *run;
  struct thread_s *idle;
  struct list_entry ready;
  struct list_entry dead;
  void *data;
};

struct sched_s sched_tbl[CONFIG_CPU_NR];
```

Scheduler : API

```
uint_t sched_init(struct sched_s *sched);
void   sched_add_created(struct thread_s *thread);
uint_t sched_yield();
void   sched_sleep();
void   sched_wakeup(struct thread_s *thread);
void   sched_exit();
void   sched_destroy(struct tread_s
                     *thread);
```




```
void shedule();
struct thread_s *elect();
```

Schedule : commutation de threads

Appelée dans sched_yield(), sched_sleepsched_exit()

Fonction schedule()

- Sauvegarder le contexte du Thread courant
- Élire un nouveau Thread à partir de la liste des Threads à l'état prêt (READY) du processeur courant, selon la politique d'ordonnancement de ce processeur.
- Restaurer le contexte du Thread élu

schedule : commutation de threads

Algorithme de schedule()

SI la liste des Thread à l'état READY est vide ALORS sortir

SI (cpu_save_context() == 0)
{
- th_élu = élire un thread /* elect() */
- SI th_élu est vient d'être crée ALORS
  - Mettre th_élu à l'état RUN ou KERNEL
  - Charger le thread th_élu /* cpu_load_thread() */
- Mettre th_élu à l'état KERNEL
- Restaurer le contexte du th_élu /* cpu_restore_context() */

}

Thread Idle

Si un ordonnanceur ne trouve aucun Thread à l'état Ready.
Il charge un Thread particulier nommé Thread Idle.

- Au démarrage du système, aucun thread n'est disponible pour être chargé sur un processeur (exception du thread main).
- Lorsque tous les Threads d'un processeur sont en attente sur des ressources non disponibles.

L'utilité de ce Thread Idle est double :

- Pour ne pas bloquer le processeur vis-à-vis des interruptions et de pouvoir ainsi d'exécuter leurs ISR.
- Peut être programmé pour exécuter un code spécial pour un ramasse-miette, l'anticipation d'accès au disques, le débugage ou d'observation de l'état du système.

Le thread Idle ne doit jamais s'endormir

kthread_create

```
struct thread_s * kthread_create (
                        kthread_t *kfunc,
                        void *arg,
                        uint_t cpuid)
```

alloue la structure thread_s et son context
on ne donne que les arguments obligatoires
les autres sont des valeurs par défaut

kthread_exit

```
void kthread_exit(uint_t val)
```

affecte la valeur de retour
appel de sched_exit()

sched_init

error_t sched_init(struct sched_s *sched)

initialise tous les schedulers
appelée une fois par __do_init

sched_add_create

void sched_add_created(struct thread_s *thread)

ajoute le thread dans la bonne liste de thread en attente

sched_exit

void sched_exit()

sched_yield

void sched_yield()

sched_sleep

void sched_sleep()

sched_wakeup

void sched_wakeup(struct thread_s *thread)

sched_destroy

void sched_destroy()

Guide pour le prochain TME

- commutation de contexte avec de la mémoire statique
- commutation de contexte avec de la mémoire dynamique
- commutation de thread

Dans les trois cas les threads/contextes appellent explicitement schedule ou sched_yield

La fonction __do_init

Pour le processeur 0:

- initialiser la mémoire
- Pour chaque processeur
  - initialiser la structure de l'ordonnanceur
  - création du thread idle
- création du thread main défini par la fonction main()
- initialisation de la plateforme
- lever la barrière pour les processeurs en attente
- charger le thread main

Pour tous les autres processeurs:

- charger le thread idle