Gestion des Threads

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 *(thread idle, thread kernel, thread user)*
- état du thread *(ready, wait, ...)*
- numéro de CPU *(affinité)*
- valeur de retour du thread *(après sa mort)*
- chainon de la liste des threads dans le même état
- chainon de la liste des threads vivants
- contexte *(état du processeur quand il attend)*

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
{
  UTHREAD,
  KTHREAD,
  ITHREAD
} thread_type_t;

typedef enum
{
   CREATE,
   READY,
   USR,
   KERNEL,
   WAIT,
   ZOMBIE,
   DEAD
} thread_state_t;
```

Les états d'un thread

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

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

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

READY
: Le thread est prêt à être exécuté

WAIT
: Le thread est en attente de quelque-chose

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

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 peut classer les **décisions** d'ordonnancement en fonction du temps qui sépare **:** (1) l'instant de la décision de placement du thread **et** (2) l'instant d'exécution du thread.
- 3 décisions d'ordonnancement:
  - long terme (placement initial)
    à la création d'un thread choix de l'affinité,
    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 pour ce processeur.
  - 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 (migration, priorité, ...)

Scheduler (ordonnanceur) : 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 processeur. Dans notre cas le choix initial n'est pas modifiable, 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 extension du scheduler

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.

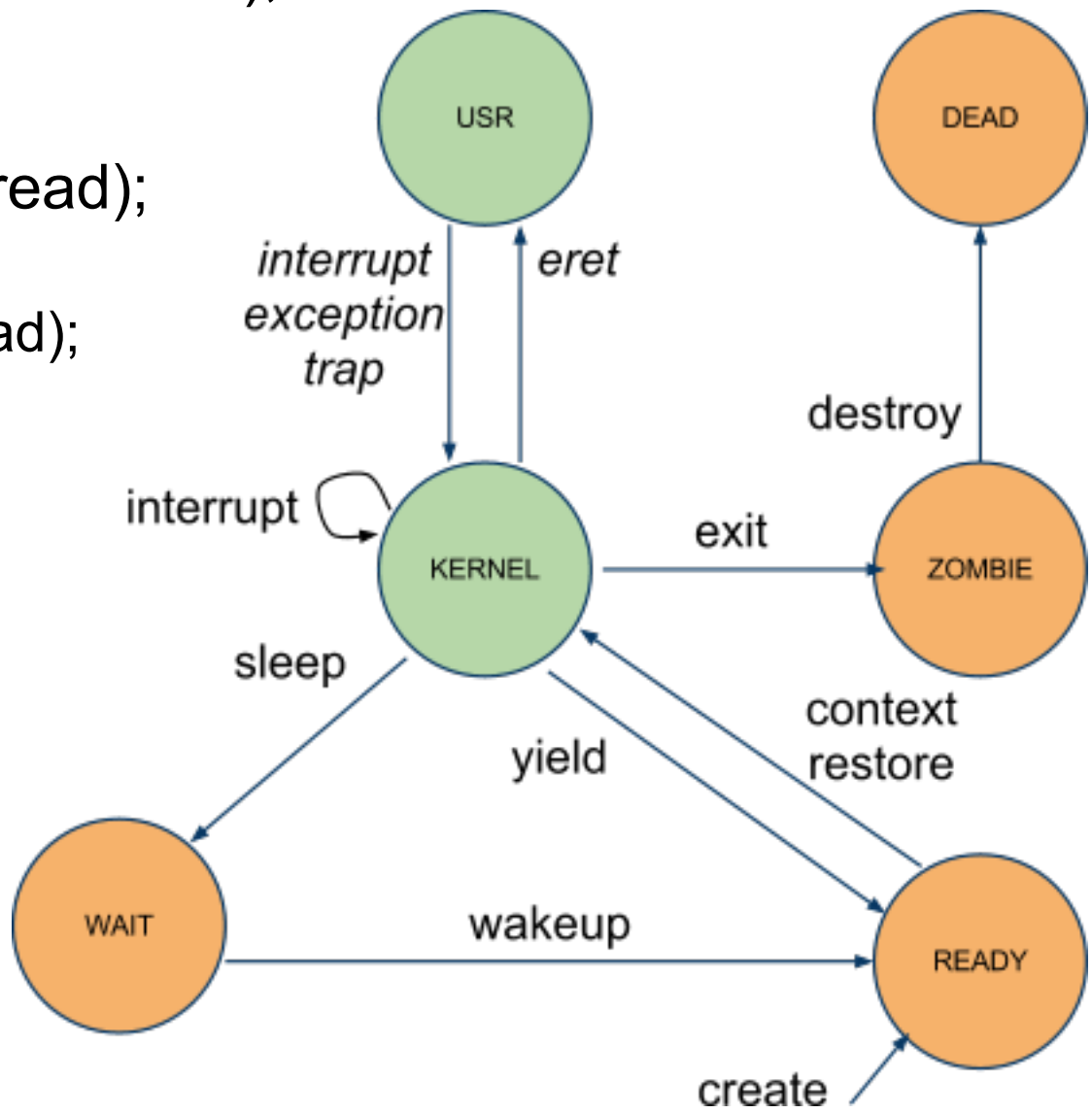
Scheduler : détails structures

```
typedef void (cpu_sched_t) (struct sched_s *sched, struct thread_s *thread);

struct sched_op {
   cpu_sched_t *create; // add a new thread
   cpu_sched_t *yield; // try to yield the cpu
   cpu_sched_t *sleep; // put to sleep the current thread
   cpu_sched_t *wakeup; // wakeup the thread
   cpu_sched_t *exit; // end a thread
   cpu_sched_t *destroy; // terminate a thread
};

struct sched_s {
   spinlock_t lock; // protect for parallel access
   struct thread_s *run; // current thread
   struct thread_s *idle; // idle thread
   list_t ready; // list of ready threads
   list_t dead; // list of dead threads
   struct sched_op sched; // scheduler operations for the cpu
   void *data; // for extension purpose
};

extern uint_t sched_init(struct sched_s *sched, struct thread_s *idle);
struct sched_s sched_tbl[CONFIG_CPU_NR];
```

Scheduler : API

```
uint_t sched_init(struct sched_s *sched);
void   add_created(struct thread_s *thread);
uint_t yield();
void   sleep();
void   wakeup(struct thread_s *thread);
void   exit();
void   destroy(struct tread_s *thread);
```

changement provoqué par
    void shedule();
qui appelle
    struct thread_s *elect();



Schedule : commutation de threads

Appelée dans yield(), sleep(), exit()

Fonction schedule()

- Sauvegarde le contexte du Thread courant
- Élit 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.
- Restaure 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

create

```
struct thread_s *thread_create(thread_type_t type, struct thread_t *start, void *arg, int cpuid)
{
   struct thread_s *thread = kmalloc(sizeof(struct thread_s));
   static int cpuid_default = 0;
   if (cpuid < 0) {
      cpuid = cpuid_default;
      cpuid_default = (cpuid_default+1)% CONFIG_CPU_NR;
   }
   spin_init(&(thread->lock));
   thread->cpuid = cpuid;
   switch (type) {
   case ITHREAD :
      thread->type = TH_IDLE;
      thread->state = KERNEL;
      break;
   case KTHREAD:
      thread->type = KTHREAD;
      thread->state = CREATE;
      break;
   }
   cpu_context_init (
      &(thread->pws),            // struct cpu_context_s *ctx,
      (uint_t) 0,              // uint_t mode_usr,
      (uint_t) kmalloc(CONFIG_STACK_SIZE) + CONFIG_STACK_SIZE - 4,    // uint_t stack_ptr,
      (uint_t) start,           // uint_t entry_func,
      (uint_t) kthread_exit,     // uint_t exit_func,
      (uint_t) thread,          // uint_t thread_ptr,
      (uint_t) arg              // uint_t arg1);
      );
   return thread;
}
```

exit

```
void exit(uint_t exit_value)
{
   struct sched_s *cpu = &cpu_tbl[cpu_get_id()];
   struct thread_s *this = cpu->run;
   kprint("thread_exit %p\n", this);
   cpu->sched.exit(cpu,this);
   ASSERT(1,"never return !\n");
}
```

sched_init

error_t sched_init(struct sched_s *sched, struct thread_s *idle)

initialise tous les schedulers
appelée une fois par __do_init
initialise le thread idle

create

void create(struct thread_s *thread)

ajoute le thread dans la bonne liste de thread en attente

sched_exit

void sched_exit(struct thread_s * thread)

passe le thread courant à l'état DEAD
schedule

yield

void yield(struct thread_s * thread)

appelle scheduler

sleep

void sched_sleep()

sched_wakeup

void sched_wakeup(struct thread_s *thread)

sched_destroy

void sched_destroy()

le prochain TME

- commutation de contexte "à la main"
- commutation de thread avec le scheduler

La fonction __do_init

Pour le processeur 0:

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

Pour tous les autres processeurs:

- chargation du thread idle