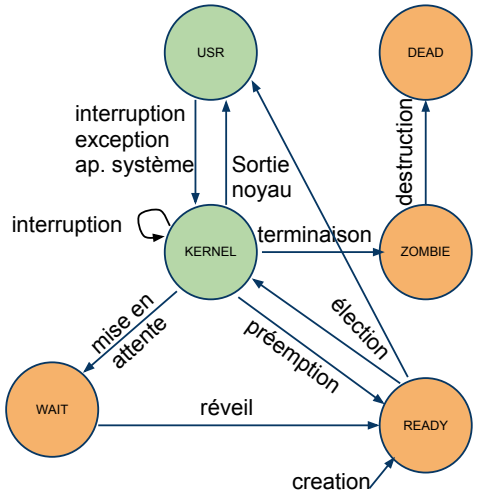
Synchronisation

MI074 - 6

Changement d'état des threads



- Dans les états USR et KERNEL :
  le thread est pointé par le champ run du scheduler générique.
- Dans l'état WAIT :
  le thread est dans la liste d'attente d'une ressource.
- Dans l'état READY :
  le thread dans la liste d'attente d'un scheduler spécifique en attente du processeur.
- Dans l'état ZOMBIE et DEAD :
  le thread est dans la liste d'attente dead du scheduler générique.

Comment se font les changements d'état ?

Comportement des opérations

```
void   sched_create(thread_t *thread);
void   sched_yield();
void   sched_sleep();
void   sched_wakeup(thread_t *thread);
void   sched_exit();
void   sched_destroy(thread_t *thread);
```

thread_create
fabrique un thread dans l'état create mais ne l'ajoute pas dans un scheduler.
on dit sur quel CPU et politique on veut.

Toutes les opérations sched_xxxx sur un scheduler sont protégées contre les accès parallèles et concurents. On commence par prendre le verrou du scheduler et on bloque les IRQ (une interruption peut provoquer un accès au scheduler donc un étreinte fatale)

sched_create thread
Le thread dans le scheduler du CPU demandé (suivant la politique demandée).
sched_create ne demande pas de commutation.

sched_yield
toujours demandé par un thread dans l'état KERNEL.
le thread courant est mis dans l'état READY, placé dans la liste d'attente de son scheduler sauf si c'est le tread_idle.
un nouveau thread est choisi, il y en a toujours un (lui-même ou le thread_idle)
le thread courant est commuté si ce n'est pas lui même.
Attention il y a un cas particulier si le thread_choisi est dans l'état CREATE parce que la commutation sort directement à la première instruction du nouveau thread et il faut avoir laché le verrou du scheduler avant le restore

Comportement des opérations

```
void   sched_create(thread_t *thread);
void   sched_yield();
void   sched_sleep();
void   sched_wakeup(thread_t *thread);
void   sched_exit();
void   sched_destroy(thread_t *thread);
```

sched_sleep
toujours demandé par le thread courant, mais contrairement à sched_yield,
le thread doit déjà être dans l'état WAIT et déjà dans une liste d'attente
Il peut arriver que le thread soit revenu de l'état WAIT et présent dans une liste READY
dans le cas ou un sched_wakeup survient entre juste avant le sleep.
Après sched_sleep: on demande une élection et une commutation.

sched_wakeup thread
Toujours demandé sur un thread dans l'état WAIT déjà détaché de sa liste d'attente.
Met le thread à l'état READY et l'attache à la liste d'attente du scheduler

sched_exit
toujours demandé par le thread courant déjà dans l'état ZOMBIE et dans une liste DEAD
Après sched_sleep: on demande une élection et une commutation.
A noter que la sauvegarde de l'état du thread ne sert à rien...

sched_destroy thread
toujours demandé sur un thread DEAD détaché de sa liste DEAD.
free du thread et des piles.

thread_join & thread_exit

`int thread_join(thread_t *thread, void **value_ptr)`
permet au thread courant de se mettre en attente de **`thread`** et de récuperer sa valeur

`void thread_exit(void *exit_value)`
permet d'arreter un thread et de réveiller le thread en attente s'il y en a un

```
void thread_exit(void *exit_value)

    affecter le champs exit_value de this
    PRENDRE LE VERROU de this
    état de this <- ZOMBIE
    attaché this dans la liste DEAD du scheduler
    S'il n'y a pas encore de thread join en attente
        LACHER LE VERROU de this
    sinon
        état de join <- READY  (il était WAIT)
        LACHER LE VERROU de this
        sched_wakeup de join
    finsi
    sched_exit de this
```

```
int thread_join(thread_t *thread, void **value_ptr)

    PRENDRE LE VERROU de thread
    Si l'état de thread est ZOMBIE
        LACHER LE VERROU de thread
    sinon
        mettre this dans en attente de thread
        état de this <- WAIT
        LACHER LE VERROU de thread
        sched_sleep de this
    finsi
    récupérer la valeur de thread
```

On remarque que l'on a jamais simultanément le verrou du thread que l'on joint (celui dont on attend la mort) et le verrou du scheduler.

Etudier les différents cas si exit et join sont quasi-simultanés

Partage de ressource

Soit T1 et T2 : 2 threads
Soit P une ressource

T1 PREND P
prendre le verrou de P
si P est libre
    mettre P à l'état PRIS
    lacher le verrou de P
    sortir
si P est PRIS
    mettre T1 à l'état WAIT
    mettre T1 dans la liste d'attente de P
    lacher le verrou de P
    SLEEP T1

T2 LACHE P
prendre le verrou de P
si P est attendu par un thread
    extraire un thread T1 en attente
    lacher le verrou de P
    WAKEUP T1

On voit encore que le verrou de P et du scheduler ne sont pas pris simultanément

Si T2 LACHE P s'insère avant SLEEP T1 alors T1 PREND P

T1 s'est endormi mais il va être réveillé par WAKEUP T1 qui le met à l'état READY et le place dans la liste ready

puis T1 lance SLEEP T1 qui agit comme un yield.

thread_join & thread_exit

`int thread_join(thread_t *thread, void **value_ptr)`
permet au thread courant de se mettre en attente de **`thread`** et de récuperer sa valeur

`void thread_exit(void *exit_value)`
permet d'arreter un thread et de réveiller le thread en attente s'il y en a un

```
void thread_exit(void *exit_value)
{
    thread_t *this = sched_get_thread_run();
    MESSG(STANDARD,"THREAD_EXIT %p", this);
    this->exit_value = exit_value;
    spin_lock(&(this->lock));
    thread_set_state(this, ZOMBIE);
    thread_list_add_last(sched_get_list_dead(),this);
    if (this->join == NULL) {
        spin_unlock(&(this->lock));
    } else {
        thread_set_state(this->join, READY);
        spin_unlock(&(this->lock));
        sched_wakeup(this->join);
    }
    sched_exit();
    ERROR("THREAD_EXIT : never return !");
}
```

```
int thread_join(thread_t *thread, void **value_ptr)
{
    thread_t *this = sched_get_thread_run();
    MESSG(STANDARD,"THREAD_JOIN %p wait for %p",
                    this, thread);
    spin_lock(&(thread->lock));
    if (thread_get_state(thread) == ZOMBIE) {
        spin_unlock(&(thread->lock));
    } else {
        thread->join = this;
        thread_set_state(this, WAIT);
        spin_unlock(&(thread->lock));
        sched_sleep();
    }
    if (value_ptr)
        *value_ptr = thread->exit_value;
    thread_set_state(thread, DEAD);
    return 0;
}
```

Le mutex

mutex.h

```
typedef struct mutex_s mutex_t;

extern mutex_t * mutex_create (void);
extern int mutex_init (mutex_t *mutex);
extern int mutex_trylock (mutex_t *mutex);
extern int mutex_lock (mutex_t *mutex);
extern int mutex_unlock (mutex_t *mutex); }
extern int mutex_destroy (mutex_t *mutex);
```

mutex.c

```
struct mutex_s
{
    spinlock_t  lock;
    int         value;
    list_t      wait;
}

int mutex_init(mutex_t *mutex)
{
    spin_init(&mutex->lock);
    mutex->value = 0;
    list_root_init(&mutex->wait);
    return 0;
}
```

```
int mutex_lock(mutex_t *mutex)
{
    spin_lock(&(mutex->lock));
    if (mutex->value == 0) {
        mutex->value = 1;
        spin_unlock(&(mutex->lock));
    } else {
        thread_t *this = sched_get_thread_run();
        thread_set_state(this, WAIT);
        thread_list_add_last(&mutex->wait,
                             sched_thread_run());
        spin_unlock(&(mutex->lock));
        sched_sleep();
    }
    return 0;
```

```
int mutex_unlock(mutex_t *mutex)
{
    spin_lock(&(mutex->lock));
    if (list_isempty(&mutex->wait)) {
        mutex->value = 0;
        spin_unlock(&mutex->lock);
    } else {
        thread_t *thread = thread_list_item(
                           list_unlink(list_first(
                           &mutex->wait)));
        spin_unlock(&(mutex->lock));
        sched_wakeup(thread);
    }
    return 0;
}
```