

Gestion des Threads

MI074 - 6

Notions Processus / Thread

Un processus est un conteneur de ressources permettant d'exécuter une application. Les ressources sont :

- un espace mémoire pour le code et les données
- un programme (listing)
- un ensemble de fichiers ouverts
- des sockets réseaux pour la communication inter-processus
- des fils d'exécution(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*)
- Des propriétés (ordonnancement, type de terminaison, etc)

Les threads peuvent être connus ou pas du noyau



Notion d'un Thread

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-coeurs.

Inconvénient

- Absence de protection inter-threads

Les états d'un thread

CREATE

Le thread vient d'être créé mais n'a pas encore été chargé sur le processeur

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 (attend un coeur)

WAIT

Le thread est en attente d'une ressource

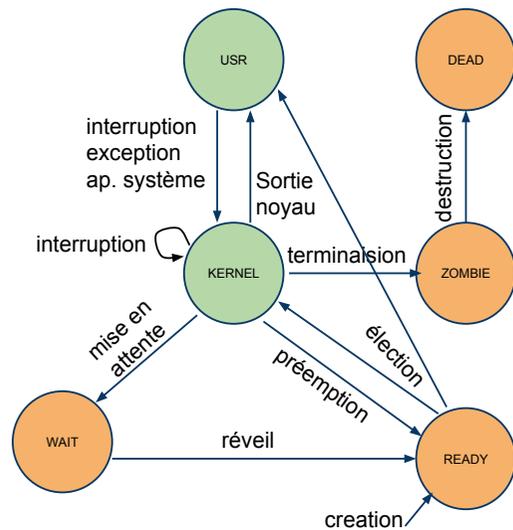
ZOMBIE

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

DEAD

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

Graphe de changement d'état



Les noeuds représentent les états dans lesquels un thread peut être au cours de son existence.

En vert le thread s'exécute
En orange il attend dans une liste

Les arcs représentent les événements qui imposent des changements d'état

Propriétés de l'ordonnanceur

- L'ordonnanceur se nomme scheduler en anglais.
- L'ordonnanceur est le sous-système en charge de la gestion des threads prêts à exécuter par un cœur ou par les cœurs.
- On a un ordonnanceur par cœur (c'est un choix, on peut en discuter).
Un thread est affecté à un cœur (donc à un ordonnanceur) à la création et dans notre cas, ce placement initial n'est pas modifiable, le noyau ne permet pas la migration des threads.
- L'ordonnanceur propose une API qui décrit des opérations de transition d'état des threads (cf graphe).
- L'implémentation de ces opérations, c'est à dire les algorithmes aboutissant au choix des threads lors des décisions d'ordonnement définissent la **politique** d'ordonnement.
- L'ordonnanceur peut supporter **simultanément** plusieurs politiques de gestion des threads, donc lorsqu'on change l'état d'un thread, on lui applique la fonction de transition correspondant à la politique dont il dépend.
- Chaque politique de gestion définit sa propre structure de données pour y ranger les threads en attente dans l'ordonnanceur (un tableau, une liste, un tableau de liste, un arbre, etc.).

Ordonnement des Threads

- L'ordonnement consiste à décider l'ordre d'exécution des threads.
- Les décisions d'ordonnement se font quand le système **décide** de la place d'un thread dans une liste de thread.
- Les **décisions** d'ordonnement sont classées en fonction du temps qui sépare : (1) l'instant de la décision **de** (2) l'instant d'exécution du thread.
- Il y a 3 types de décisions d'ordonnement:
 - Court terme (élection)
quand on décide quel thread s'exécute sur un cœur parmi les threads prêts
 - Long terme (placement initial à la création)
à la création d'un thread choix de l'affinité avec un cœur en fonction de critères globaux
 - Moyen terme (planification)
quand on organise les threads prêts ou en attente.
De run à wait, de wait à ready, de wait à swap.
Équilibrage de la charge des processeurs (migration, priorité, ...)
- Chaque thread peut dépendre d'une politique d'ordonnement différente
On peut créer des threads plus prioritaires que d'autres qui seront favorisés lors des décisions d'ordonnement.

Cloisonnement des sous-systèmes

On veut que l'ordonnanceur propose des mécanismes qui soient :

- indépendants de la structure détaillée des threads
- indépendants des politiques de gestion des threads

Pour garantir cette indépendance, l'ordonnanceur:

- ne connaît pas la structure d'un thread
- ne connaît pas l'implémentation des fonctions de son API, car il y a autant d'implémentation que de politique.
- On va avoir autant de structures de données pour les threads prêts et d'implémentation de l'API scheduler que de politique.

De même les threads ne connaissent pas les structures de données de l'ordonnanceur générique et des ordonnanceurs par politique.

⇒ On va utiliser des structures opaques

On va définir les API dans les fichiers scheduler.h et thread.h

API du scheduler.h (1)

```
#include <klist.h>

typedef struct thread_s thread_t; // hidden structure of thread
typedef struct sched_s sched_t; // hidden structure of scheduler

// -----
// scheduler function behavior
// -----

typedef thread_t * sched_op_t (sched_t *sched, thread_t *thread); // generic prototype

typedef struct sched_ops_s {
    sched_op_t *select; // chose a new thread
    sched_op_t *create; // add a new thread
    sched_op_t *yield; // try to yield the cpu
    sched_op_t *sleep; // put to sleep the current thread
    sched_op_t *clock; // call at periodic intervalle
    sched_op_t *exit; // end the current thread
    sched_op_t *wakeup; // wakeup a thread
    sched_op_t *destroy; // terminate a thread
} sched_ops_t;

enum sched_policy_e {
    SCHED_RR = 0, // round robin time shared
    SCHED_FIFO, // FIFO no preemption
    SCHED_OTHER, // user define
    SCHED_POLICY_NR // number of scheduler policy
};

extern int sched_init_rr (sched_t **sched, sched_ops_t *ops); // initialisation rr scheduler
extern int sched_init_ff (sched_t **sched, sched_ops_t *ops); // initialisation ff scheduler
extern int sched_init_ot (sched_t **sched, sched_ops_t *ops); // initialisation tt scheduler
```

tableau des fonction de changement d'état il y a une implémentation par politique

API du scheduler.h (2)

```
// -----
// scheduler accessors
// -----

extern list_t * sched_get_list_dead (void); // return the list dead

extern thread_t * sched_get_thread_run (void); // return the current thread
extern thread_t * sched_get_thread_idle (void); // return the thread idle

extern void sched_set_thread_run (thread_t *thread);
extern void sched_set_thread_idle (thread_t *thread);

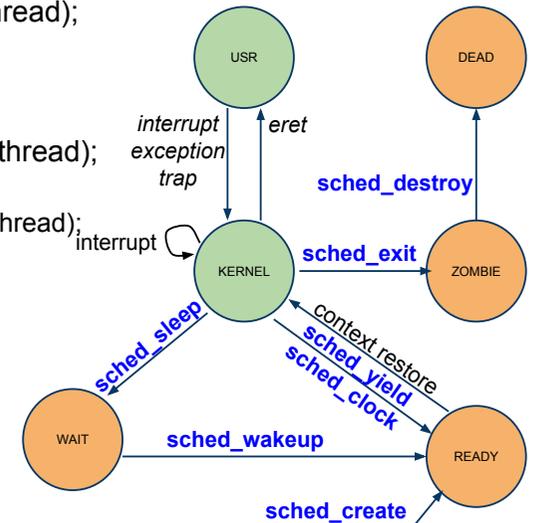
// -----
// scheduler operations : la fonction appelée dépend de la politique de gestion du thread
// -----

extern void sched_init (void); // initialise one scheduler
extern void sched_create (thread_t *thread); // add a new thread
extern void sched_wakeup (thread_t *thread); // wakeup a thread
extern void sched_destroy (thread_t *thread); // terminate a thread
extern void sched_clock (void); // periodic interruption
extern void sched_yield (void); // try to yield the cpu
extern void sched_sleep (void); // asleep the current thread
extern void sched_exit (void); // end the current thread

#endif
```

API du scheduler utilisé par les threads

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



Opération schedule

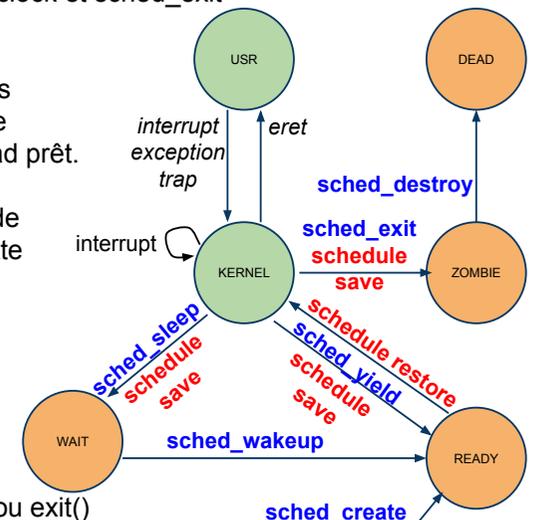
Le changement du thread en cours (run) est provoqué par les fonctions sched_sleep, sched_yield, sched_clock et sched_exit

L'ordonnanceur générique appelle dans un ordre fixe les électeurs des ordonnanceurs de chaque politique pour demander de donner un thread prêt.

Puis l'ordonnanceur fait l'opération de sauvegarde / restauration de contexte

S'il n'y a aucun thread prêt, on choisit le thread_idle()

L'opération `shedule()`; `elect - save - restore` n'est pas externe car elle est seulement appelée par `sleep()` `yield()` `clock()` ou `exit()`



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 débogage ou d'observation de l'état du système.

Le thread Idle ne doit jamais s'endormir

API de thread.h (1)

```
#ifndef _THREAD_H
#define _THREAD_H

#include <hal_arch.h>
#include <scheduler.h>
#include <klist.h>

// -----
// thread structure accessors
// -----

enum thread_type_e {
    ITHREAD = 0, // IDLE Thread
    KTHREAD, // KERNEL Thread
    UTHREAD, // USER Thread
    THREAD_TYPE_NR // number of thread types
};

enum thread_state_e {
    CREATE = 0, // Initial Thread at the first beginning
    READY, // Thread ready for a cpu
    USR, // Running Thread in user mode
    KERNEL, // Running Thread in kernel mode
    WAIT, // Thread waiting for anything but CPU
    ZOMBIE, // after exit but not yet joined
    DEAD, // waiting for garbage collection
    THREAD_STATE_NR // number of thread states
};

extern unsigned thread_get_cpuid (thread_t *thread);
extern unsigned thread_get_state (thread_t *thread);
extern unsigned thread_get_type (thread_t *thread);
extern unsigned thread_get_policy (thread_t *thread);
extern void thread_set_state (thread_t *thread, unsigned thread_state);
```

API de thread.h (2)

```
// -----
// List of all threads (only used for statistics and debug)
// -----

extern spinlock_t thread_ropo_lock;
extern list_t thread_ropo;

// -----
// Manage the ready list of thread, required since the thread structure is hident
// -----

extern thread_t * thread_list_item (list_t *item);
extern void thread_list_add_first (list_t *root, thread_t *thread);
extern void thread_list_add_last (list_t *root, thread_t *thread);

// -----
// thread context and co
// -----

typedef void * thread_fct_t (void *); // Thread function type

extern thread_t * thread_create (unsigned thread_type,
                                unsigned sched_policy,
                                unsigned cpuid,
                                thread_fct_t * start,
                                void * arg);
extern unsigned thread_save (thread_t * thread);
extern void thread_restore (thread_t * thread);
extern void thread_exit (void * exit_value) __attribute__((noreturn));

extern void * thread_idle (void * arg);
extern void * thread_start (void * arg);

#endif
```