Gestion des Threads

MI074 -5

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-processeurs.

Inconvénient

- Absence de protection inter-threads

Notions Processus / Thread

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

- un espace mémoire
- un programme (code)
- des descripteurs de fichiers
- des sockets réseaux
- des fils d'exécution ou 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 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

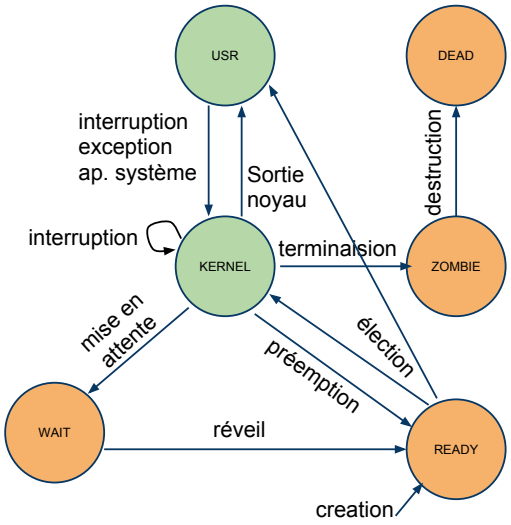
READY
: Le thread est prêt à être exécuté

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

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 évènements 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 core ou par les cores.
- On a un ordonnanceur par core (c'est un choix, on peut en discuter).
  Un thread est affecté à un core (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'ordonnancement définissent la **politique** d'ordonnancement.
- L'ordonnanceur peut supporter **simultanément** plusieurs politiques de gestion des threads, donc lorsqu'on change d'état un thread on lui applique la fonction de transition correspondant à la politque 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.).

Ordonnancement des Threads

- L'ordonnancement consiste à décider l'ordre d'exécution des threads.
- Les décisions d'ordonnancement se font quand le système **décide** de la place d'un thread dans une liste de thread.
- Les **décisions** d'ordonnancement 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'ordonnancement:
  - Court terme (élection)
    quand on décide quel thread s'exécute sur un core parmi les threads prêts
  - Long terme (placément initial à la création)
    à la création d'un thread choix de l'affinité avec un core 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 polique d'ordonnancement différente
  On peut creer des threads plus prioritaires que d'autres qui seront favorisés lors des décisions d'ordonnancement.

Cloisonnement des sous-systèmes

On veut que l'ordonnanceur propose des mécanismes qui soient :
- indépendants de la structure détaillée des threads
- indépendants des poliques de gestion des threads

Pour garantir cette indépendance, l'ordonnanceur:
- ne connait pas la structure d'un thread
- ne connait pas l'implémentation des fonctions de son API, car il y a autant d'implémentation que de politique.
- On va avoir autant de structure de donnée 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

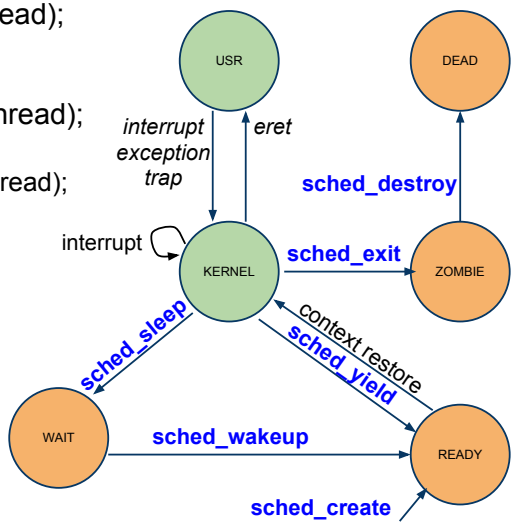
On va définir les API dans les fichiers scheduler.h et thread.h

API du scheduler.h (1)

```
#include <klist.h>

typedef struct thread_s thread_t;                   // hiden structure of thread
typedef struct sched_s sched_t;                     // hiden 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 *elect;                              // 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
```

API du scheduler utilisé par les threads

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);



API du scheduler.h (2)

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

extern list_t   * sched_get_list_dead    (void);    // return the ready 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
// ---------------------------------------------------------------------------------------------------

extern void sched_init      (void);                 // create all schedulers
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
```

Opération schedule

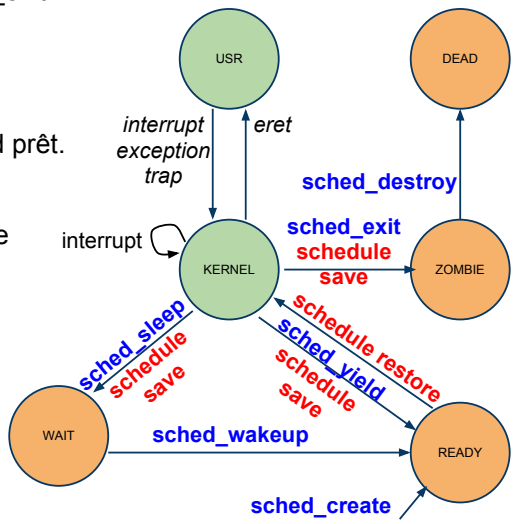
Le changement du thread en cours (run) est provoqué par les fonctions sched_sleep, sched_yield et sched_exit

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

Puis l'ordanceur fait l'opération de sauvegarde / restoration 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() 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 débugage 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 begining
    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_rope_lock;
extern list_t       thread_rope;

// ---------------------------------------------------------------------------------------------
// 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
```

Scheduler générique (extrait 1)

```
// scheduler structure
// Only one structure an array of array but it could be be possible to do an other way
// ---------------------------------------------------------------------------------------------

typedef struct schedroot_s {
    spinlock_t    lock;                     // protect for parallel access
    thread_t *    run;                      // current thread for the core
    thread_t *    idle;                     // idle thread for the core
    list_t        dead;                     // list of dead threads on this core
    sched_ops_t   op[SCHED_POLICY_NR];      // pointers to specific operation
    sched_t     * sched[SCHED_POLICY_NR];   // pointers to specific structure
} schedroot_t;

static schedroot_t schedroot[CONFIG_CPU_NR];    // one scheduler per core

// scheduler public accessors ------------------------------------------------------------------

list_t *sched_get_list_dead(void)               {return &(schedroot[CPUID].dead);}

thread_t *sched_get_thread_run(void)            {return schedroot[CPUID].run;}
thread_t *sched_get_thread_idle(void)           {return schedroot[CPUID].idle;}

void sched_set_thread_run(thread_t *thread)     {schedroot[CPUID].run = thread;}
void sched_set_thread_idle(thread_t *thread)    {schedroot[CPUID].idle = thread;}

// scheduler private accessors -----------------------------------------------------------------

static inline schedroot_t * sched_get_root(thread_t *thread) {
    return &(schedroot[thread_get_cpuid(thread)]);}

static inline sched_t * sched_get_sched(thread_t *thread) {
    return sched_get_root(thread)->sched[thread_get_policy(thread)];}

static inline sched_ops_t * sched_get_op(thread_t *thread) {
    return &(sched_get_root(thread)->op[thread_get_policy(thread)]);}
```

Scheduler générique (extrait 2)

```
// scheduler operations
// ------------------------------------------------------------------------------------------------

void sched_init(void) // chaque processeur va lancer son sched_init
{
    spin_init(&(schedroot[CPUID].lock));
    schedroot[CPUID].run = NULL;
    schedroot[CPUID].idle = NULL;
    list_root_init(&(schedroot[CPUID].dead));
    sched_init_rr(&(schedroot[CPUID].sched[SCHED_RR])   ,&(schedroot[CPUID].op[SCHED_RR])   );
  //sched_init_ff(&(schedroot[CPUID].sched[SCHED_FIFO]) ,&(schedroot[CPUID].op[SCHED_FIFO]) );
  //sched_init_ot(&(schedroot[CPUID].sched[SCHED_OTHER]),&(schedroot[CPUID].op[SCHED_OTHER]));
}

static thread_t * elector(schedroot_t *schedcpu, thread_t *this)
{
      La fonction elector prend en parametre un pointeur sur l'ordonnanceur du CPU courant et sur le thread courant.
      Notez que le thread this est n'est jamais dans l'état KERNEL lorsqu'on appelle l'electeur. Il peut être dans l'état
      SLEEP, READY ou ZOMBIE.
      La fonction appelle les électeurs de chaque politique qui peuvent rendre un pointeur vers un thread READY ou
      CREATE ou un pointeur NULL s'ils n'ont pas de thread dans leur 'liste' ready.
      Si aucun électeur n'a de thread, la fonction rend le thread idle.
}

static void commute(thread_t *this, thread_t *next)
{
      La fonction commute effectue la commutation de thread de this à next.
      Si this = next, il n'est pas utile de faire la commutation, il faut juste changer l'état de this pour KERNEL
      Sinon il faut faire le couple save-restore ATTENTION il y a une subtilité pour les threads next dans l'état CREATE.
      En effet, leur état n'a jamais été sauvé par save et il ne sortiront donc pas par save. Il faut faire attention au lock.
}
```

Scheduler spécifique (extrait 1)

```
#include <thread.h>
#include <libk.h>

struct sched_s {
    list_t ready;
};

sched_t sched_rr[CONFIG_CPU_NR];

static thread_t *elect_rr(sched_t *sched, thread_t *this)
{
    thread_t * elected;
    if (list_isempty(&(sched->ready))) {
        MESSG(VERBOSE,"ELECT_RR return NULL");
        return NULL;
    }
    // rendre le pointeur sur thread elue le premier extrait de la liste ready
    MESSG(VERBOSE,"ELECT_RR return %p", elected);
    return elected;
}

static thread_t *create_rr(sched_t *sched, thread_t *thread)
{
    MESSG(STANDARD,"CREATE_RR %p on cpu %d", thread, thread_get_cpuid(thread));
    ASSERT(thread_get_state(thread) == CREATE,);
    thread_list_add_last(&(sched->ready),thread);
    return thread;
}
```

Scheduler générique (extrait 3)

```
static void schedule(sched_op_t *oper)
{
    unsigned    status;
    thread_t    *this = sched_get_thread_run();
    schedroot_t *schedcpu = sched_get_root(this);
    spin_lock_noirq(    &(schedcpu->lock), &status);
    oper(               sched_get_sched(this), this);
    commute(            this, elector(schedcpu, this));
    spin_unlock_noirq(  &(schedcpu->lock), status);
}

static void call(sched_op_t *oper, thread_t *thread)
{
    unsigned     status;
    schedroot_t *schedcpu = sched_get_root(thread);
    spin_lock_noirq(    &(schedcpu->lock), &status);
    oper(               sched_get_sched(thread), thread);
    spin_unlock_noirq(  &(schedcpu->lock), status);
}

void sched_create(thread_t *thread) {call(     sched_get_op(thread)->create,  thread);}
void sched_wakeup(thread_t *thread) {call(     sched_get_op(thread)->wakeup,  thread);}
void sched_destroy(thread_t *thread){call(     sched_get_op(thread)->destroy, thread);}
void sched_clock(void)              {schedule( sched_get_op(sched_get_thread_run())->clock);}
void sched_yield(void)              {schedule( sched_get_op(sched_get_thread_run())->yield);}
void sched_sleep(void)              {schedule( sched_get_op(sched_get_thread_run())->sleep);}
void sched_exit(void)               {schedule( sched_get_op(sched_get_thread_run())->exit);}
```

Scheduler spécifique (extrait 2)

```
static thread_t * yield_rr(sched_t *sched, thread_t *this)
{
    ASSERT(thread_get_state(this) == KERNEL,);
    if (this != sched_get_thread_idle()) {
        MESSG(STANDARD,"YIELD_RR %p", this);
        thread_list_add_last(&(sched->ready), this);
    }
    thread_set_state(this, READY);
    return this;
}

static thread_t * clock_rr(sched_t *sched, thread_t *this)
{
     return NULL;
}

static thread_t * sleep_rr(sched_t *sched, thread_t *this)
{
    MESSG(STANDARD,"SLEEP_RR %p", this);
    ASSERT(thread_get_state(this) == KERNEL,);
    thread_set_state(this, WAIT);
    return this;
}
```

Scheduler spécifique (extrait 3)

```
static thread_t * wakeup_rr(sched_t *sched, thread_t *thread)
{
    MESSG(STANDARD,"WAKEUP_RR %p on cpu %d", thread, thread_get_cpuid(thread));
    return NULL;
}

static thread_t * exit_rr(sched_t *sched, thread_t *this)
{
    if (thread_get_type(this) == ITHREAD) return NULL; // idle never die
    MESSG(STANDARD,"EXIT_RR %p", this);
    return this;
}

static thread_t * destroy_rr(sched_t *sched, thread_t *thread)
{
    ERROR("Not yet implemented");
    return NULL;
}

int sched_init_rr(sched_t **sched, sched_ops_t *ops)
{
    *sched       = &(sched_rr[CPUID]);
    list_root_init(&((*sched)->ready));
    ops->elect   = elect_rr  ;                    // chose a new thread
    ops->create  = create_rr ;                    // add a new thread
    ops->yield   = yield_rr  ;                    // try to yield the cpu
    ops->clock   = clock_rr  ;                    // put to sleep the current thread
    ops->sleep   = sleep_rr  ;                    // put to sleep the current thread
    ops->wakeup  = wakeup_rr ;                    // wakeup the thread
    ops->exit    = exit_rr   ;                    // end a thread
    ops->destroy = destroy_rr;                    // terminate a thread
    return 0;
}
```

Thread (extrait 1)

```
// thread structure
// -----------------------------------------------------------------------------------------------
struct thread_s {
    spinlock_t  lock;
    int         type;
    int         state;
    int         policy;
    int         cpuid;
    list_t      list;
    list_t      rope;
    void *      exit_value;
    unsigned    pws[CONTEXT_SIZE];
};

spinlock_t thread_rope_lock;
list_t thread_rope;

// Manage the ready list of thread
// -----------------------------------------------------------------------------------------------

thread_t * thread_list_item      (list_t *p)                 {return list_item(p,thread_t,list);}
void       thread_list_add_first (list_t *root, thread_t *th) {list_add_first(root,&th->list);}
void       thread_list_add_last  (list_t *root, thread_t *th) {list_add_last(root,&th->list);}

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

unsigned   thread_get_cpuid  (thread_t *thread)     {return thread->cpuid;}
unsigned   thread_get_type   (thread_t *thread)     {return thread->type; }
unsigned   thread_get_state  (thread_t *thread)     {return thread->state;}
unsigned   thread_get_policy (thread_t *thread)     {return thread->policy;}

void thread_set_state (thread_t *thread, unsigned state) {
    if (state<THREAD_STATE_NR) thread->state=state;}
```

Thread (extrait 2)

```
// thread operations
// -----------------------------------------------------------------------------------------------

static void thread_exit_wrap(void)
{
    thread_exit(NULL);
}

thread_t *thread_create(unsigned        type,
                        unsigned        sched_policy,
                        unsigned        cpuid,
                        thread_fct_t *  start,
                        void *          arg)
{
    thread_t *thread = kmalloc(sizeof(thread_t));
    MESSG(STANDARD,"THREAD_CREATE %p ", thread);

    static int cpuid_default = 0;

    cpuid = ((cpuid < 0) ? cpuid_default++
                         : cpuid) % CONFIG_CPU_NR;
    thread->cpuid = cpuid;

    spin_init(&(thread->lock));
    void * stack_ptr = kmalloc(CONFIG_STACK_SIZE)
                     + CONFIG_STACK_SIZE - 4;

    cpu_context_init (
        thread->pws,                // struct cpu_context_s *ctx,
        (unsigned) 0,               // unsigned mode_usr,
        (unsigned) stack_ptr,       // unsigned stack_ptr,
        (unsigned) start,           // unsigned entry_func,
        (unsigned) thread_exit_wrap,// unsigned exit_func,
        (unsigned) arg              // unsigned arg1);
        );
```

Thread (extrait 3)

```
    MESSG(VERBOSE, "stack=%p start=%p", stack_ptr, start);

    thread->policy = sched_policy;

    switch (type) {

    case ITHREAD :
        thread->type = ITHREAD;
        thread->state = CREATE;

        spin_init(&thread_rope_lock);
        list_root_init(&thread_rope);
        break;

    case KTHREAD:
        ...
        break;

    case UTHREAD:
        ERROR("UTHREAD not yet implemented");
    }
    return thread;
}

unsigned thread_save(thread_t *thread)
{
    return cpu_context_save(thread->pws);
}
```

```
void thread_restore(thread_t *thread)
{
    MESSG(STANDARD,"THREAD_RESTORE %p lock %d",
                   thread);
    sched_set_thread_run(thread);
    thread_set_state(thread, KERNEL);
    cpu_context_restore(thread->pws, 1);
    ERROR("thread_restore : never return");
}

void thread_exit(void *exit_value)
{
    MESSG(STANDARD,"THREAD_EXIT %p",
                   sched_get_thread_run());
    sched_get_thread_run()->exit_value = exit_value;
    sched_exit();
    ERROR("never return !");
}
```