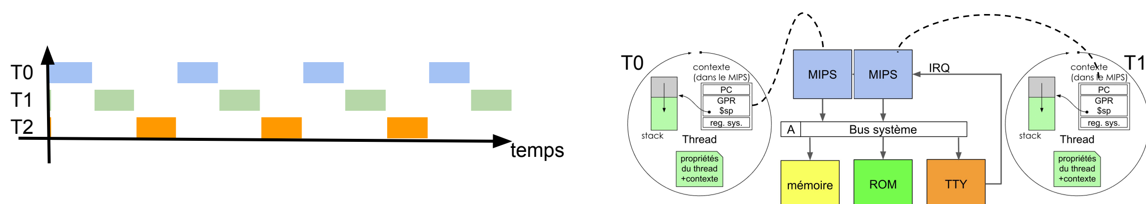
Mécanismes de synchronisation

LU3IN031 Architecture des ordinateurs - 2
Matériel et Logiciel
B7

franck.wajsburt@lip6.fr
V4

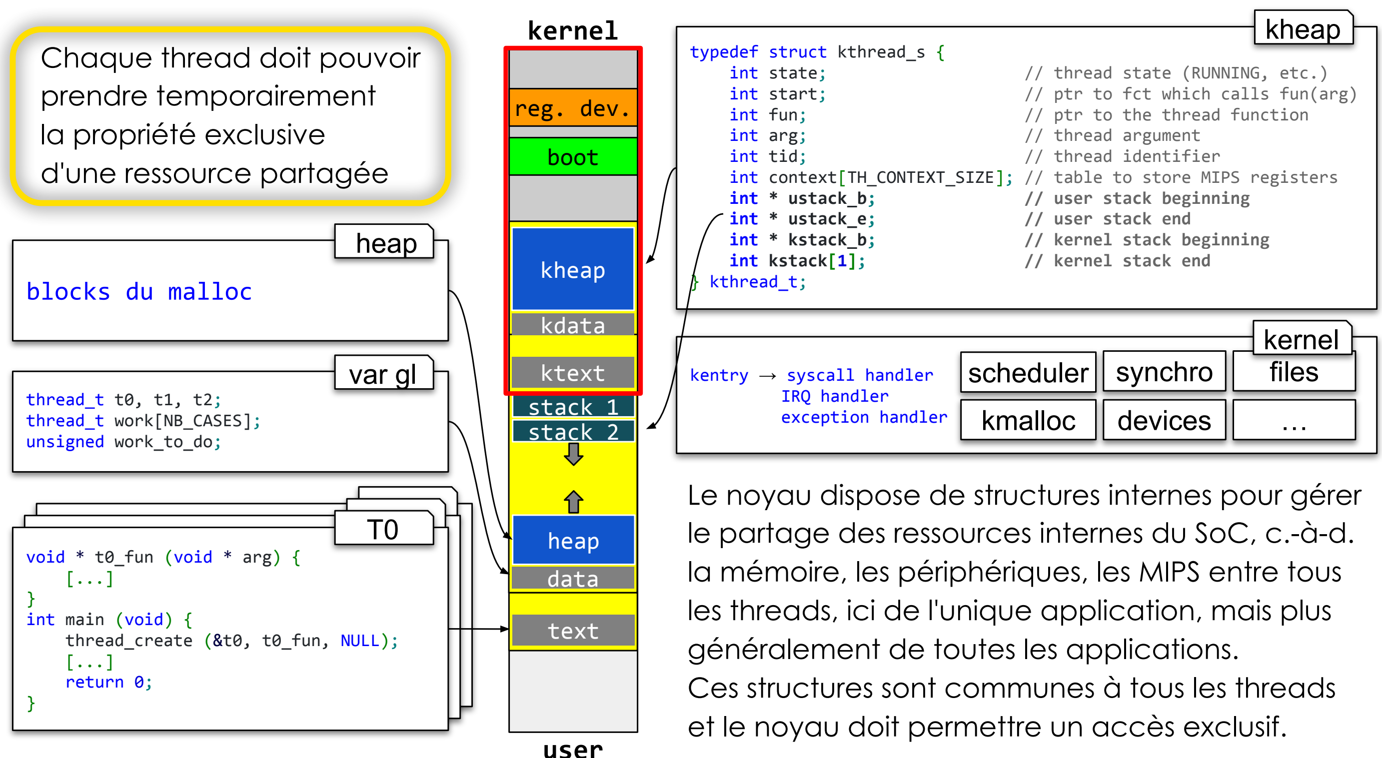
Le problème du partage…



T0, T1 et T2 se partagent les variables globales de l'application et du noyau

- Au niveau utilisateur dans l'application les 3 threads pourraient écrire des messages sur un terminal et comme il y a le temps partagé, un thread peut commencer à écrire un message et perdre le MIPS, le nouveau thread peut aussi écrire un message ainsi les messages vont être mélangés. Dans kO6, les syscalls ne sont pas interruptibles, mais c'est peu commun.
- Deux threads peuvent appeler un `malloc()` en même temps sur 2 cores (ou sur le même à cause du temps partagé), le `malloc()` est entièrement exécuté en mode user et il peut donc être interrompu par un `thread_yield()`, le `malloc()` vu précédemment modifie des structures et ne doit pas être interrompu dans son travail sous peine de corrompre ces structures.
- Au niveau du noyau, c'est le même problème pour `kmalloc()` et plus généralement pour toutes les structures partagées par les threads : l'ordonnanceur, les périphériques, etc.

…en fait tout est partagé



Le noyau dispose de structures internes pour gérer le partage des ressources internes du SoC, c.-à-d. la mémoire, les périphériques, les MIPS entre tous les threads, ici de l'unique application, mais plus généralement de toutes les applications.
Ces structures sont communes à tous les threads et le noyau doit permettre un accès exclusif.

Objectifs de la séance

On peut exécuter plusieurs threads en temps partagé mais ils partagent aussi les ressources comme l'allocateur de mémoire ou le terminal, et plus généralement tous les périphériques. Il est nécessaire de les synchroniser afin de garantir la propriété exclusive de chaque ressource et éviter que les threads les modifient en même temps

- Quel est le mécanisme de synchro de base sur lequel tous les autres sont construits ?
- Si les threads se synchronisent, certains doivent attendre, quels sont les nouveaux états ?
- Si une ressource est partagée, comment gérer une file d'attente pour les threads ?
- Comment peut-on récupérer le résultat d'un thread ?
- Comment récupérer les codes d'erreurs des appels systèmes en multi-threads ?
- En TME, comment synchroniser les threads à la fin d'une étape de calcul ?

Mécanismes de synchronisation basiques

Principe du verrou ... problème

Le mécanisme de base utilise une variable binaire nommé verrou ou **lock** indiquant la disponibilité d'une ressource : 1 = busy ; 0 = free (not busy)

Il y a deux opérations de base :

verrouillage : tant que (lock == busy) attendre; lock ← busy
libération : lock ← free

Le verrouillage est une attente active et donc bloquante, l'API proposée est :

```
typedef unsigned spinlock_t;              // ici le spinlock est un simple entier
void spin_lock (spinlock_t * lock);       // spin a cause de l'attente active
void spin_unlock (spinlock_t * lock);     // fonction non bloquante
```

C'est simple ... sauf qu'il faut se protéger contre les accès concurrents venant de deux ou de plusieurs threads s'exécutant sur le même MIPS ou sur plusieurs.

Problème **Si un lock est free et que 2 threads exécutent `spin_lock()` en même temps, chacun va voir que le verrou est libre et le prendre : ce n'est pas correct**

Solution **Il faut garantir l'atomicité du verrouillage**

Leslie Lamport a proposé l'algorithme de la boulangerie permettant de se passer d'opérations atomiques (1974), c'est très beau mais inutile ici... https://www.wikiwand.com/fr/Algorithme_de_la_boulangerie

Principe du verrou ... solutions

Tous les mécanismes de partage doivent garantir qu'une case mémoire n'a qu'un seul écrivain à chaque instant, au sinon le **matériel** doit proposer un mécanisme permettant la séquence atomique **lecture** d'un mot - **modification** de sa valeur - **écriture** de sa nouvelle valeur, une séquence read-modify-write

> *C'est un problème de **consensus** entre les threads qui doivent se mettre d'accord sur la valeur du verrou et en avoir la même vision. Dans le cas général, il existe des solutions permettant à des participants d'avoir un consensus sur des états d'un système alors même que ces participants sont non fiables et qu'ils s'échangent des messages sur un réseau non fiable ! (ex. https://www.wikiwand.com/en/Raft_(algorithm)) (inutile ici mais intéressant aussi :-)*

Dans un SoC, les conditions sont très simplifiées parce que le système est fiable (le MIPS, le bus système, les mémoires et le code du noyau sont fiables).

Nous allons voir 4 mécanismes possibles et nous utiliserons la dernière de cette liste :

1. Mémoire de verrous matériels
2. instruction **`tas`** test-and-set
3. Instruction **`cas`** compare-and-swap
4. Couple d'instructions **`ll / sc`** (load Link / store conditional)

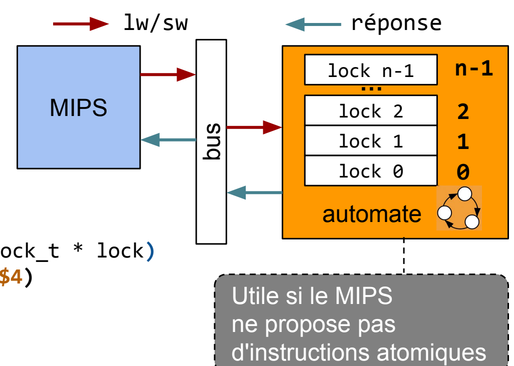
4 Solutions matérielles pour les verrous

1. Mémoire de verrous

Mémoire dont la lecture provoque une écriture

`lw $rt, imm($rs)` lit la valeur du mot à l'adresse `$rs+imm` si sa valeur est 0 alors le mot passe à 1

`sw $rt, imm($rs)` écrit la valeur de $r à l'adr. `$rs+imm`



Utile si le MIPS ne propose pas d'instructions atomiques

```
void spin_lock (spinlock_t * lock)
spin_lock:    lw   $8, ($4)
              beqz $8, spin_lock
              jr   $31
```

```
void spin_unlock (spinlock_t * lock)
spin_unlock: sw   $0, ($4)
             jr   $31
```

2. Test And Set   *(Attention ce n'est pas une instruction MIPS)*

`tas $rt, imm($rs)`

écrit la valeur de `$rt` dans le mot à l'adresse `$addr+imm` et rend l'ancienne valeur

```
void spin_lock (spinlock_t * lock)
spin_lock:    li   $8, 1
              tas  $8, ($4)
              bnez $8, spin_lock
              jr   $31
```

```
void spin_unlock (spinlock_t * lock)
spin_unlock: sw   $0, ($4)
             jr   $31
```

Pour prendre un verrou, on tente d'écrire 1 jusqu'à ce qu'on y parvienne. On le sait car la valeur rendue par **tas** est alors 0.

4 Solutions matérielles pour les verrous

3. Compare And Swap *(Attention ce n'est pas une instruction MIPS)*

`cas $old, $new, imm($addr)`

compare la valeur de `$old` avec celle du mot à l'adresse `$addr+imm`
si elles sont égales, alors écrit `$new`

```
void spin_lock (spinlock_t * lock)
spin_lock:   li   $8, 0
             li   $9, 1
             cas  $8, $9, ($4)
             bnez $8, spin_lock
             jr   $31
```

```
void spin_unlock (spinlock_t * lock)
spin_unlock: sw   $0, ($4)
             jr   $31
```

Pour prendre un verrou, on tente d'écrire 1 jusqu'à ce qu'on y parvienne. On le sait car la valeur rendue par **cas** est alors 0.

Compare And Swap permet de faire des compteurs atomiques

```
void atomic_add (int * counter, int val)
atomic_add:  lw   $8, ($4)
             addu $9, $8, $5
             cas  $8, $9, ($4)
             bnez $8, atomic_add
             jr   $31
```

Problème ABA

Si on a une variable **var** partagée par plusieurs threads peut prendre les valeurs **A** et **B**

1. Si un thread **T** lit la valeur **A**, puis
2. tente d'écrire la valeur **B** avec **CAS A,B,(var)**

S'il y parvient en renvoyant bien l'ancienne valeur **A**
il se peut que **var** soit passée par la valeur **B**
puis qu'elle soit revenue à la valeur **A**

Le thread **T** ne peut pas le savoir, cela peut être un problème, p. ex. pour les compteurs atomiques, si le nombre de valeurs du compteur est faible…

4 Solutions matérielles pour les verrous

4. Load Link / Store Conditional *(Solution présente dans le MIPS)*

`ll $rt, imm($rs)`

lit dans `$rt` la valeur du mot présent l'adresse `$rs+imm`
C'est comme un `lw` **mais** la mémoire note qu'un `ll` est en cours à cette adresse (`link`)
La mémoire ouvre une "réservation" à cette adresse (`ll` est aussi appelée Load Reserved)

`sc $rt, imm($rt)`

tente d'écrire `$rt` à l'adresse `$addr+imm`
L'écriture est acceptée uniquement s'il y a une réservation ouverte pour cette adresse
Mais la réservation est fermée par toutes les instructions mémoire autre que `ll`
Si le store a fonctionné `$rt` contient 1 sinon il contient 0

```
void spin_lock (spinlock_t * lock)
spin_lock:
    ll    $2, ($4)
    bnez  $2, spin_lock
spin_lock_sc:
    li    $2, 1
    sc    $2, ($4)
    beqz  $2, spin_lock
    jr    $31
```

```
void spin_unlock (spinlock_t * lock)
spin_unlock: sw   $0, ($4)
             jr   $31
```

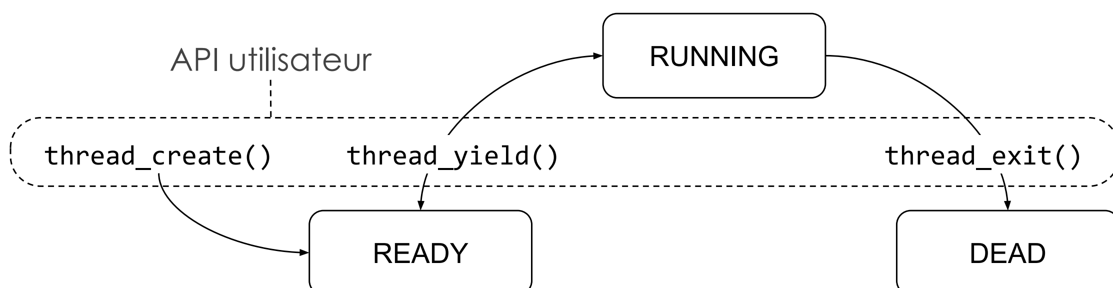
La solution LL / SC n'a pas le problème ABA parce la mémoire garantie qu'aucune opération n'a lieu entre un `ll` et un `sc` réussi, LL / SC peut simuler un CAS et pas l'inverse
MAIS ce n'est pas gratuit, la mémoire doit faire des réservations

Etats des threads

États de thread (avant)

Jusqu'à présent les threads n'avaient que 3 états

- `RUNNING` seul le thread qui possède le MIPS est dans cet état (il y a un seul MIPS)
- `READY` état de tous les threads vivants qui attendent le MIPS
- `DEAD` état dans lequel un thread se mettait quand il exécutait `thread_exit()`
  Le thread qui meurt doit toujours appeler l'ordonnanceur pour lancer un autre thread,
  En conséquence, l'effacement du thread `DEAD` est retardé, il fait par un autre thread.

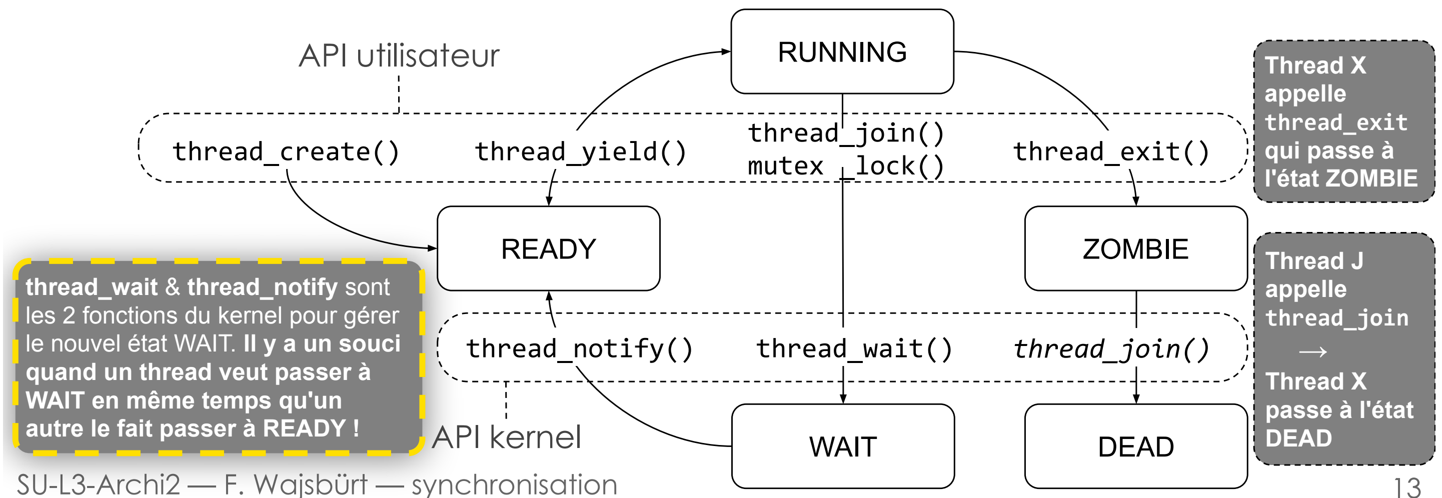


Il n'y avait pas d'état WAIT dans la version précédente du noyau et la conséquence était que lorsqu'un thread attendait une donnée d'un périphérique, il ne se mettait pas en attente, il rendait le MIPS, qu'il reprenait plus tard pour retenter jusqu'à réussir, c'était de la scrutation en attente active (polling)

États de thread (désormais)

Désormais, les threads ont 5 états

- `RUNNING` seul le thread qui possède le MIPS est dans cet état (il y a un seul MIPS)
- `READY` état de tous les threads vivants qui attendent le MIPS
- `ZOMBIE` état dans lequel un thread se met quand il exécute `thread_exit()`
- `WAIT` état dans lequel un thread se met quand il demande une ressource indisponible
- `DEAD` état dans lequel un thread **est mis par un autre thread** dès que sa valeur est récupérée



Structure thread_s invisible

Les fonctions manipulant les threads :
soit pointent le thread courant comme **thread_exit()**,
soit un identifiant sur un thread comme **thread_join()**

La structure thread est "opaque" (non visible)

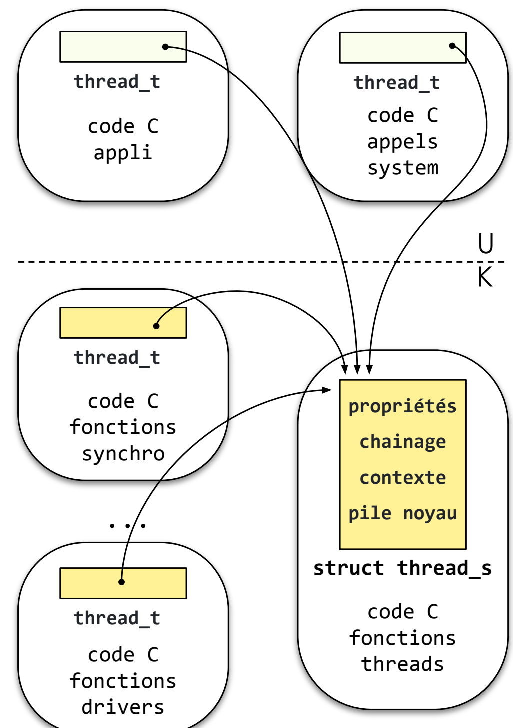
→ Le code de l'application et le code de tous les modules du noyau manipulent des **pointeurs** sur la structure `thread_s` sans possibilité d'accéder à ses champs.

→ Seul le fichier C qui implémente le code des fonctions de threads connait les champs de la structure.

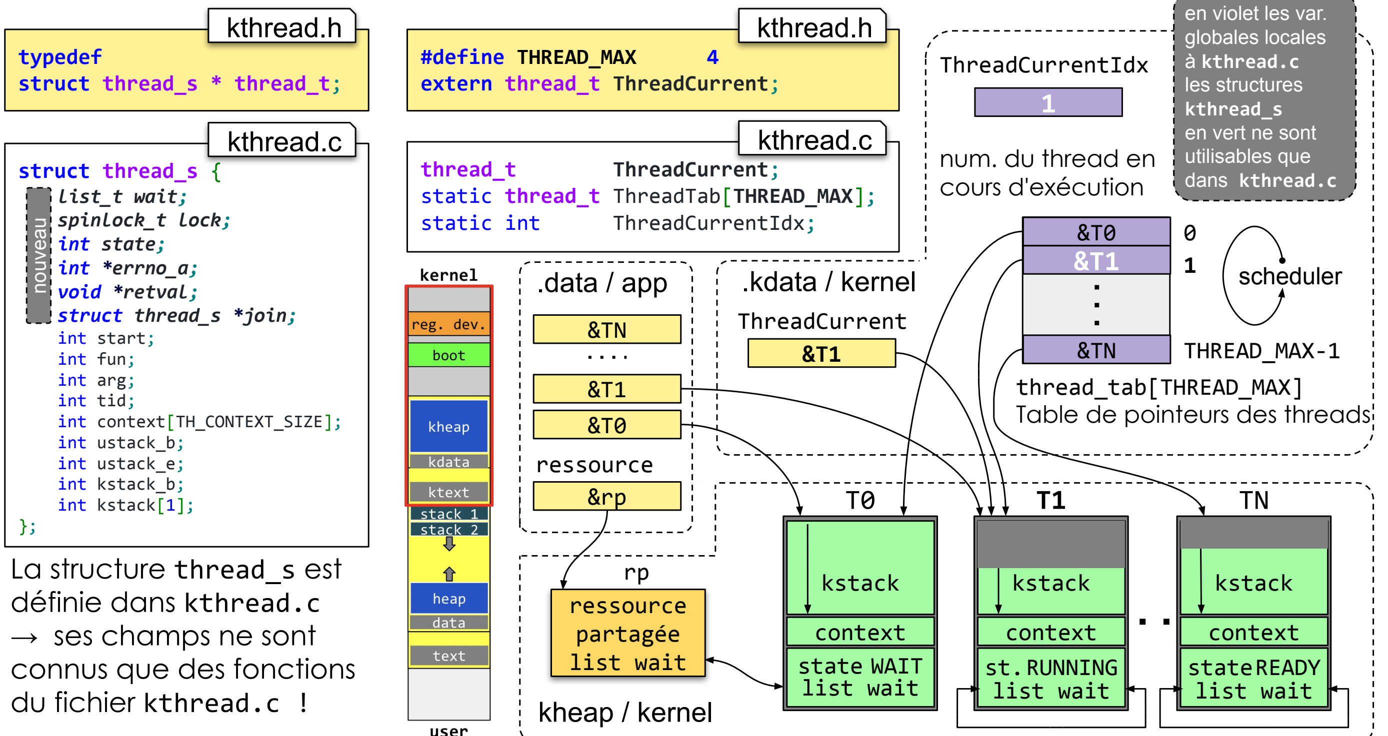
Pour la création, on donne un pointeur sur la variable pointeur qui contiendra le pointeur sur la structure et pour les autres fonctions se sert du pointeur comme un identifiant…

```
typedef struct thread_s * thread_t;
int thread_create (thread_t * threadp, …);
int thread_join (thread_t thread);
```

Ce sera le cas de toutes les structures du noyau



Struct thread_s et ordonnanceur (naïf)



thread_wait & thread_notify

Scénario

Une tâche **T0** veut prendre une ressource partagée **R** déjà prise par la tâche **T1**
T0 s'ajoute à la liste d'attente de **R** et **T0** appelle **thread_wait()** pour passer à l'état **WAIT**.
Quand **T1** rend **R**, **T1** voit que **T0** attend dans la liste d'attente de **R**, **T1** détache **T0** puis
T1 donne **R** à **T0** puis **T1** appelle **thread_notify(T0)** pour la faire passer à l'état **READY**

Séquence des étapes (vrai parallélisme)

*la compétition entre T0 et T1 se nomme **race condition** le résultat dépend de l'ordre*

T1	T0
a1. **T1** prend **R**	
.	b1. **T0** veut **R**
.	b2. **T0** s'ajoute à **R**
a2. **T1** rend **R**	.
a3. **T1** donne **R**	.
a4. **T1** notify(**T0**)	.
	b3. **T0** wait()

Problème

Si **T0** se met en **WAIT** en possession de **R** alors aucun thread ne viendra le réveiller, c'est fini ! L'application est bloquée !

Solution

T0 doit savoir que **T1** l'a déjà réveillé !

Quand **T0** se met en attente, alors elle doit passer de RUNNING à WAIT, mais si **T1** exécute notify(T0), elle passe **T0** à l'état **READY**, ce que **T0** peut tester

```
void thread_wait (void) {
    spin_lock (&ThreadCurrent->lock);
    if (ThreadCurrent->state == TH_STATE_RUNNING)
        ThreadCurrent->state = TH_STATE_WAIT;
    spin_unlock (&ThreadCurrent->lock);
    sched_switch ();
}
void thread_notify (thread_t thread) {
    spin_lock (&ThreadCurrent->lock);
    thread->state = TH_STATE_READY;
    spin_unlock (&ThreadCurrent->lock);
}
```

Le spin_lock protège le test de thread_wait
Soit T0 passe à READY avant le test et T0 reste READY
soit après et il passe aussi à READY

Mutex

Présentation du problème

Nous avons vu les fonctions de base de synchronisation

```
typedef struct spinlock_s * spin_lock_t;       // structure non visible
void spin_lock (spinlock_t * lock);            // fonction bloquante, on sort avec le lock
void spin_unlock (spinlock_t * lock);          // fonction pour relâcher le lock
```

Auxquels s'ajoutent les fonctions d'initialisation et de destruction

```
void spin_init (spinlock_t * lock);            // fonction d'init, en princ. avec attributs
void spin_destroy (spinlock_t * lock);         // fonction pour libérer la ressource
```

Ce sont des fonctions qui font des attentes actives, scrutant le spinlock jusqu'à le posséder, il n'y a pas de priorités entre les demandeurs et donc il y a des famines potentielles.
Notez que ces derniers points peuvent être résolus grâce à des spinlocks à tickets (fondés sur l'algorithme de la boulangerie de Lamport mais simplifié grâce aux instructions atomiques)

Les spinlocks sont aussi nommés busy lock pour indiquer que l'attente utilise le MIPS.

Ce n'est pas un problème tant que l'attente est bornée dans le temps, et c'est le cas dans le noyau quand il modifie une variable partagée, c'est court et les interruptions sont masquées.

Mais pour les threads les spinlocks sont chers. Si un thread possédant un spinlock perd le MIPS, il ne peut plus le rendre et les autres demandeurs devront attendre plusieurs quanta !

API Mutex

Le Noyau propose un mécanisme pour demander la propriété sans attente active

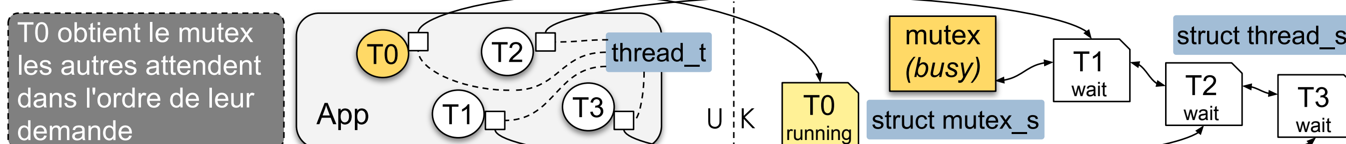
```
typedef struct thread_mutex_s * thread_mutex_t;   // on ne veut pas connaître le contenu
int thread_mutex_lock (thread_mutex_t * lock);    // fonction bloquante, on sort avec le mutex
int thread_mutex_unlock (thread_mutex_t * lock);  // fonction pour relâcher le mutex
```

Auxquels s'ajoutent les fonctions d'initialisation et de destruction

```
int thread_mutex_init (thread_mutex_t * lock);    // fonction d'init, en princ. avec attributs
void thread_mutex_destroy (thread_mutex_t * lock);// fonction pour libérer la ressource
```

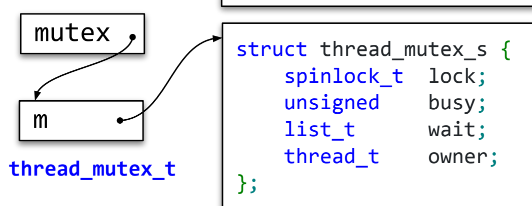
C'est la même API, la fonction `thread_mutex_lock()` est bloquante, mais elle ne fait pas d'attente active. Si le mutex n'est pas libre, le thread perd le MIPS et il ne le retrouvera qu'en possession du mutex, elle n'utilise pas `thread_yield()` mais `thread_wait()` qui est la fonction de l'API kernel des threads appelée après le chaînage dans une liste d'attente

Il y a une priorité entre les threads, définie par une politique du noyau (ici, premier arrivé premier servi) et donc il n'y a pas de famine, *un thread est toujours sûr d'obtenir un mutex**.



* Enfin, presque puisque les mutex utilisent les spinlocks

thread_mutex_lock




```
typedef struct thread_mutex_s * thread_mutex_t;
```

```
struct thread_mutex_s {
    spinlock_t   lock;
    unsigned     busy;
    list_t       wait;
    thread_t     owner;
};
```

Les mutex sont utilisés pour gérer la propriété exclusive ! Donc, en plus du **lock**, de la variable d'état **busy** et de la liste chainée des threads en attente du mutex **wait**, il y a un champ propriétaire **owner** pour tester que **lock** / **unlock** sont réalisées par le même thread, sinon c'est une erreur

```
int thread_mutex_lock (thread_mutex_t * mutex) {
    thread_mutex_t m = *mutex;
    spin_lock (&m->lock);
    if (m->busy) {
        thread_addlast (&m->wait, ThreadCurrent);
        spin_unlock (&m->lock);
        thread_wait ();
    } else {
        m->busy = 1;
        m->owner = ThreadCurrent;
        spin_unlock (&m->lock);
    }
    return SUCCESS;
}
```

Les tests sur les arguments ont été retirés *(sur le slide)*

L'argument mutex est un pointeur sur un mutex de type `thread_mutex_t` lui même un pointeur sur une structure de type `thread_mutex_s`

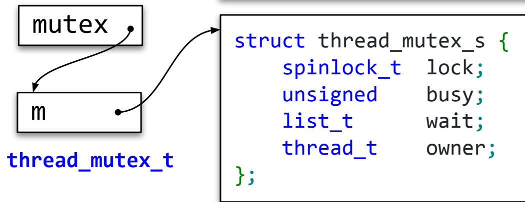
prendre le lock pour l'exclusivité sur struct *m
si le mutex est occupé
- ajouter le thread courant en fin de liste
- **lâcher le lock**
- changer l'état du thread → WAIT sauf si notify arrive avant wait

sinon
- prendre le mutex
- indiquer le propriétaire
- **lâcher le lock**

fsi

il faut absolument lâcher le lock avant d'entrer dans thread_wait() car on perd le MIPS

thread_mutex_unlock




```
typedef struct thread_mutex_s * thread_mutex_t;
```

```
struct thread_mutex_s {
    spinlock_t  lock;
    unsigned    busy;
    list_t      wait;
    thread_t    owner;
};
```

Les mutex sont utilisés pour gérer la propriété exclusive ! Donc, en plus du **`lock`**, de la variable d'état **`busy`** et de la liste chainée des threads en attente du mutex **`wait`**, il y a un champ propriétaire **`owner`** pour tester que **`lock`** / **`unlock`** sont réalisées par le même thread, sinon c'est une erreur

Les tests sur les arguments ont été retirés *(sur le slide)*

L'argument mutex est un pointeur sur un mutex de type `thread_mutex_t` lui même un pointeur sur une structure de type `thread_mutex_s`

prendre le lock pour l'exclusivité sur struct *m
prendre le premier thread en attente
s'il y en a un
    indiquer la propriété
    réveiller ce thread → READY dans tous les cas
sinon   lâcher le mutex
    plus de propriétaire
fsi
lâcher le lock

```
int thread_mutex_unlock (thread_mutex_t * mutex) {
    thread_mutex_t m = *mutex;
    spin_lock (&m->lock);
    list_t * thread_waiting = list_getfirst(&m->wait);
    if (thread_waiting) {
        m->owner = thread_item (thread_waiting);
        thread_notify (m->owner);
    } else {
        m->busy = 0;
        m->owner = NULL;
    }
    spin_unlock (&m->lock);
    return SUCCESS;
}
```

Ne pas lâcher le lock tant que le changement de propriétaire n'est pas fait, thread_notify ne fait pas perdre le MIPS

thread_mutex_lock & thread_mutex_unlock

Nous avons vu la "**race condition**" (compétition) entre `thread_notify()` et `thread_wait()`
Il y en a aussi une entre `thread_mutex_lock()` et `thread_mutex_unlock()`

Si le thread **`T0`** veut prendre le mutex, le voit **busy**, s'ajoute à la liste d'attente et s'endort, et qu'en même temps le thread **`T1`**, regarde s'il y a un thread, mais n'en voit pas encore, lache le mutex et sort, **`T0`** ne sera jamais réveillé ! **Ce cas est évité grâce au lock**

Si **`T0`** voit le mutex **busy**, il s'ajoute dans la liste sans être interrompu par un **`mutex_unlock`**
Si **`T1`** commence à rendre le mutex, **`T0`** devra attendre que ce soit fini avant de prendre

```
int thread_mutex_lock (thread_mutex_t * mutex) {
    thread_mutex_t m = *mutex;
    spin_lock (&m->lock);
    if (m->busy) {
        thread_addlast (&m->wait, ThreadCurrent);
        spin_unlock (&m->lock);
        thread_wait ();
    } else {
        m->busy = 1;
        m->owner = ThreadCurrent;
        spin_unlock (&m->lock);
    }
    return SUCCESS;
}
```

```
int thread_mutex_unlock (thread_mutex_t * mutex) {
    thread_mutex_t m = *mutex;
    spin_lock (&m->lock);
    list_t * thread_waiting = list_getfirst(&m->wait);
    if (thread_waiting) {
        m->owner = thread_item (thread_waiting);
        thread_notify (m->owner);
    } else {
        m->busy = 0;
        m->owner = NULL;
    }
    spin_unlock (&m->lock);
    return SUCCESS;
}
```

exit & join

Il faut récupérer l'héritage !

La fonction principale d'un thread a pour prototype : `void * fct (void *arg);`

`fct()` démarre en parallèle avec les threads existants un certain temps après sa création par :

```
int thread_create (thread_t *thread_p, void *(*fct)(void *), void *arg);
```

Un thread se termine par la fonction `pthread_exit(retval)` appelée explicitement par `fct()`
ou implicitement par `thread_start()`, le lanceur de la fonction `fct()` quand on revient de `fct()`

```
static void thread_start (void *(*fct) (void *), void *arg) {
    void *retval = fct (arg);
    thread_exit (retval);
}
```

La pointeur `retval` doit pouvoir être lue par un autre thread de l'application, c'est le but de la fonction `thread_join()` qui va permettre un **rendez-vous** le `pthread_exit()` du thread donné en argument :

```
void thread_join (thread_t thread, void **retval);
```

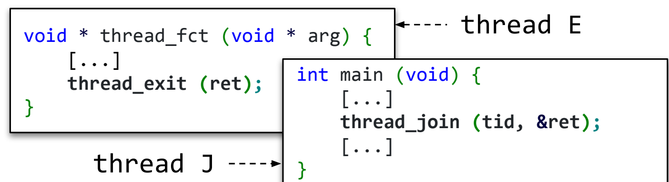
```
thread_t tid;
struct retval_s { int val; ... };
struct arg_s { int val; ... };


void * thread_fct (void * arg) {
    retval_t *retval = malloc (sizeof (struct retval_s));
    [... calcul sur arg et initialisation de retval ...]
    return (void *) retval; // thread_exit (retval)
}
```

```
int main (void) {
    struct retval_s * ret;
    struct arg_s * arg = malloc (sizeof (struct arg_s));
    [... calcul et initialisation de arg ...]
    thread_create ( &tid,  thread_fct, arg);
    [... calcul exécuté en parallèle avec thread_fct ...]
    thread_join ( tid, &ret);
    fprintf (0, "retval : %d ... \n", ret->val, ...);
    return 0;
}
```

thread_exit & thread_join

Dans l'exemple précédent, le `thread main()` attend grâce à la fonction `thread_join()` que le `thread_fct()` ait exécuté `thread_exit()` pour récupérer le résultat, c'est un rendez-vous !



Problème, Il y a deux possibilités :

1. Soit `thread_join()` est exécuté en 1er
2. Soit `thread_exit()` est exécuté en 1er

Solution

Ajouter 2 champs dans la structure thread qui vont permettre à chaque thread de savoir s'il est le premier ou le dernier au rendez-vous.

`void * retval`
pour que `thread_exit()` puisse stocker le résultat du thread

`thread_t join`
pour que `thread_join()` puisse s'enregistrer comme attendant

`void thread_exit (void *retval);`

- Écrit dans le champ `retval` du thread courant
- Passe le thread courant à l'état `ZOMBIE`
- teste si le champ join de sa structure est `NULL`
  - si oui, c'est qu'il est le premier, il ne fait rien.
  - si non c'est que `thread_join` est déjà exécuté et il passe le thread enregistré à l'état `READY`

`void thread_join (thread_t expected, void **retval);`

- Écrit dans le champ `join` du thread `expected`
- Teste si le thread expected est `ZOMBIE`
  - si oui, il récupère `retval` et il passe le thread expected à l'état `DEAD`
  - sinon, il passe à l'état `WAIT` et il appelle `sched_switch()` pour céder le MIPS

exit & join : race condition

Évidemment, il y a une race condition !

si le `thread J` (qui exécute `thread_join`)
`J1)` s'inscrit dans le champ `join` du thread `E` ,
`J2)` ne voit pas le `thread E` à l'état `ZOMBIE`
`J3)` décide de s'endormir

mais qu'entre `J2)` et `J3)`
le `thread E` (qui exécute `thread_exit`)
`E1)` passe à l'état `ZOMBIE`,
`E2)` voit que le `thread J` est déjà arrivé
`E3)` passe le `thread J` à `READY`

Donc, si les étapes s'exécutent dans l'ordre :
`J1) J2) E1) E2) E3) J3)` alors le `thread J` passe à l'état `WAIT` et le `thread E` restera à `ZOMBIE`,
→ le `thread J` ne sera jamais notifié ! **C'est fatal**

On utilise le lock du `thread J` accessible aux deux threads afin que les étapes `J2)` et `J3)` ne soient pas entrecoupées par les étapes `E1) E2) E3)`

```
int thread_join (thread_t thread_expected, void **retval)
{
    if (thread_expected == NULL) return ESRCH;
J1) thread_expected->join = ThreadCurrent;
    spin_lock (&thread_expected->lock);
J2) if (thread_expected->state != TH_STATE_ZOMBIE) {
J3)    ThreadCurrent->state = TH_STATE_WAIT;
        spin_unlock (&thread_expected->lock);
        sched_switch ();
    } else {
         spin_unlock (&thread_expected->lock);
    }
    *retval = thread_expected->retval;
    thread_expected->state = TH_STATE_DEAD;
    return SUCCESS;
}
```

```
void thread_exit (void *retval)
{
    ThreadCurrent->retval = retval;
E1) ThreadCurrent->state = TH_STATE_ZOMBIE;
    spin_lock (&ThreadCurrent->lock);
E2) if (ThreadCurrent->join != NULL)
E3)     ThreadCurrent->join->state = TH_STATE_READY;
    spin_unlock (&ThreadCurrent->lock);
    sched_switch ();
}
```

Errno

Les codes d'erreurs des appels système

Quand l'application fait un appel système, parfois ça ne fonctionne pas et le système rend un code d'erreur, mais pas seulement, le code d'erreur est aussi placé dans une variable globale nommée `errno` et cette variable est utilisée par la fonction de la libc (utilisateur) pour afficher des messages d'erreur. Par exemple, si on veut allouer un segment d'adresse trop grand (1To), alors le résultat est un pointeur `NULL`, la fonction `perror()` affiche le message "`not enough memory`", ici, **errno** est passé à exit() comme valeur de sortie du programme.

```
int main (void) {
    void char * buffer;
    buffer = malloc (100000000); // trop grand
    if (buffer == NULL) {
      perror ("main"); // affiche "main: not enough memory"
      fprintf (0, "errno = %d\n", errno);
      exit (errno);
    }
    [... calculs ...]
    return 0;
}
```

errno est changé, par presque tous les appels système, il devrait être testé juste après les appels, sinon il est perdu

`man errno` →
donne tous les codes de Linux

kO6 : `common/errno.h`

```
enum errno_code {
    FAILURE = -1,
    SUCCESS ,
    EPERM   ,
    EINTR   ,
    EIO     ,
    ENXIO   ,
    E2BIG   ,
    EAGAIN  ,
    ENOMEM  , // not … memory
    EACCES  ,
    EFAULT  ,
    EBUSY   ,
    ENODEV  ,
    EINVAL  ,
    ENOTTY  ,
    EFBIG   ,
    ENOSPC  ,
    ENOSYS  ,
    ERANGE  ,
    ESRCH   ,
    EDEADLK
};
```

errno

errno est une variable globale dans l'espace utilisateur, c.-à-d. accessible par toutes les fonctions de l'application, mais elle n'est pas définie dans la section `.data`, en effet, tous les threads font des appels systèmes mais chacun voit son propre `errno`. La variable `errno` est dîte « thread safe »

Du point de vue du programmeur,
c'est une variable comme les autres.
On doit pouvoir écrire

```
void * thread_fct (void * arg) {
   char * buffer = malloc (100000000);// trop grand
   if (errno == ENOMEM) {
      perror ("main"); // "main: not enough memory"
      errno = FAILURE;
      exit (errno);
   }
   [...]
}
```

L'idée, c'est que chaque thread calcule l'adresse de la variable errno grâce à une fonction : **`__errno_location()`**

`int * __errno_location (void);`

est une fonction qui rend une adresse d'entier différente pour chaque thread.

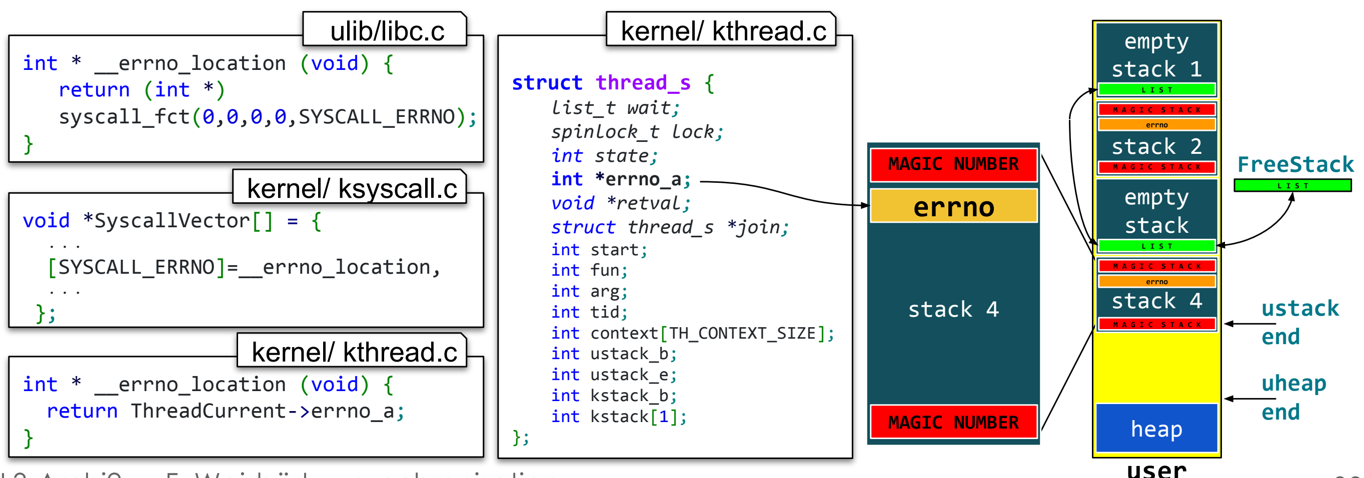
`errno` est alors un simple **`#define`**

`#define errno *__errno_location()`

Implémentation de errno pour kO6

errno est une variable prédéfinie placée dans le **thread local storage** (**TLS**) (la mémoire locale des threads), il existe plusieurs méthodes pour créer d'autres variables, soit à l'aide du compilateur gcc, soit par des API ad hoc dépendantes de l'OS. Dans l'état actuel de kO6, `errno` sera la seule. Si vous voulez en savoir plus, lisez (en français) https://www.wikiwand.com/fr/Thread_Local_Storage

Pour kO6, la variable `errno` est placée tout en haut de chaque pile user. L'adresse de cette variable est présente dans la structure `struct thread`. Elle est récupérée par un syscall.



Résumé

- Les mécanismes de synchronisation existent pour permettre aux threads de partager des ressources matérielles (les périphériques) ou logicielles (les buffers de communication) ou, nous allons le voir en TME, pour synchroniser des étapes de calcul.
- Nous avons vu 4 Solutions matérielles pour 'exécuter la séquence `read-modify-write` de manière atomique et que le MIPS utilise le couple d'instructions `LL/SC`
- Le mécanisme de base utilisé par le noyau est un verrou à attente active: spinlock
- Si les ressources sont partagées par plusieurs threads, alors quand un thread veut une ressource déjà utilisée, il est placé en état d'attente `WAIT`, dont il sort quand la ressource est lâchée par son propriétaire actuel, qui lui donne et le refait passer à l'état `READY`
- Les nouvelles fonctions de changement d'états sont `thread_wait` et `thread_notify` et qu'il a une "race condition" (compétition) qui peut provoquer de blocage du système.
- Les "race conditions" sont réglées par un usage judicieux des spinlocks
- L'utilisateur dispose de verrous à liste d'attente, les MUTEX, qui permettent d'éviter les attentes actives. Les fonctions de l'API sont `thread_mutex_lock` et `thread_mutex_unlock`
- Un thread peut récupérer la valeur de sortie d'un autre, rendu par `thread_exit,` grace à `thread_join,` et qu'il y a aussi une "race condition" (compétition)
- Chaque thread peut récupérer le code de retour des appels système grâce à une variable globale, mais locale à chaque thread, placée dans le `Thread Local Storage`

TME

- Quelques questions de cours
- Etudier un nouveau mécanisme de synchronisation, nommé barrières de synchronisation
- Implémenter une API de sémaphores