Threads

LU3IN031 Architecture des ordinateurs - 2
Matériel et Logiciel
B5

franck.wajsburt@lip6.fr
V3

Ce que nous avons vu

- Le MIPS propose deux modes d'exécution : kernel et user
  - Le mode kernel est utilisé par le kernel pour gérer les ressources de la machine (le/s processeur/s, la mémoire et les périphériques)
  - Le mode user est utilisé par les applications, ce mode interdit l'accès à une partie de l'espace d'adressage et à certaines instructions
- Le kernel démarre une application en sautant (avec `eret`) à la fonction `_start()` placée, par convention, au début de la section `.text`
- L'application revient dans le kernel pour 3 raisons :
  - Les appels système avec `syscall` pour demander un service
  - Les exceptions quand l'application exécute une instruction incorrecte
  - Les interruptions quand un périphérique réclame le processeur pour exécuter un traitement urgent

Objectifs de la séance

Une fonction d'un programme en cours d'exécution se nomme thread ou « fil d'exécution » en français.

Si dans le SoC, il n'y a qu'un seul MIPS, il est impossible d'exécuter plusieurs threads en même temps !

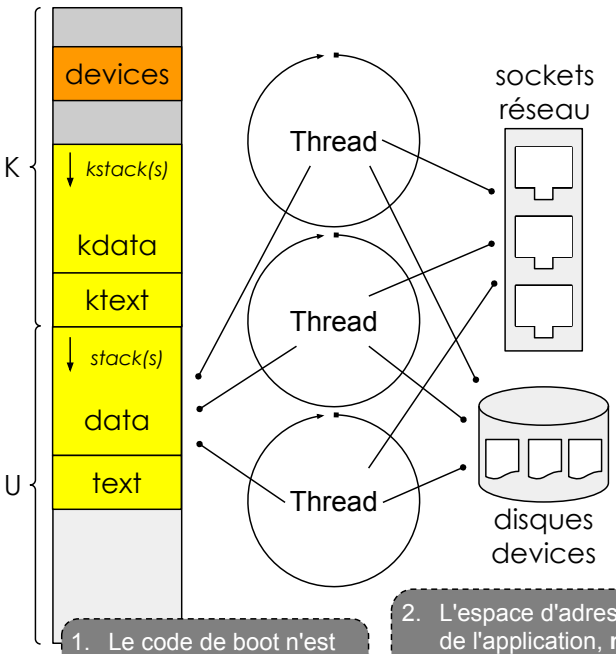
Comment peut-on en donner l'illusion?

- Comment nomme-t-on un programme en cours d'exécution ?
- De quoi a besoin un thread pour s'exécuter ?
- Qu'est-ce que l'exécution en temps partagé ?
- Comment Implémenter les threads au plus simple ?

Processus

Un processus est un **programme en cours d'exécution**

(https://www.wikiwand.com/fr/Processus_(informatique))

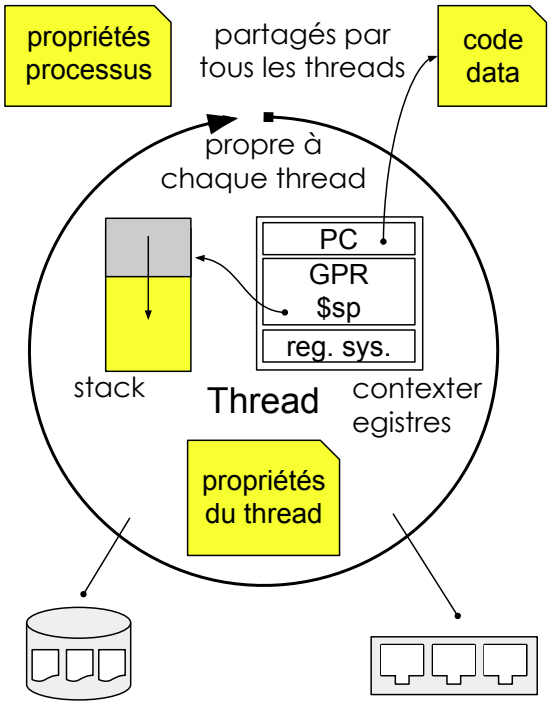


Processus = **Conteneur des ressources** nécessaires pour exécuter un programme.
Un processus contient :

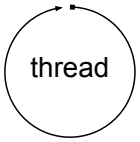
- Un **espace d'adressage** pour y mettre le code du programme, les données globales et les piles d'exécution.
- Des ressources d'**entrées-sorties** : fichiers (disque et devices) et sockets (réseau)
- Des **propriétés** (état, parenté, droits, ...)
- Une console (terminal)
- Au moins un **fil d'exécution (threads)**

1. Le code de boot n'est pas dessiné car il n'est pas dans le kernel
2. L'espace d'adressage utilisé par l'application contient le code, les données et les piles de l'application, **mais aussi** le code, les données et les piles du kernel qui sont utilisées par l'application lorsqu'elle fait des appels systèmes ou qu'elle traite les interruptions.
3. Les fichiers du disque et les périphériques (devices) sont gérés par la même API

Thread



Un thread est donc un fil d'exécution du processus, c'est ce que le processeur est en train de faire. C'est donc « *vivant* » 😊



(d'où la représentation comme une roue qui tourne)

Un Thread est défini par :

- Une **pile d'exécution** des fonctions
- Un **contexte de thread**, c.-à-d. un état des registres du processeur, dont le PC, et le pointeur dans la pile d'exécution.
- Des **propriétés** propres au thread
  - la fonction principale du thread
  - un état (prêt à être exécuté, en attente d'une ressource, …)
  - des compteurs de temps
  - etc.

Assertions sur les threads

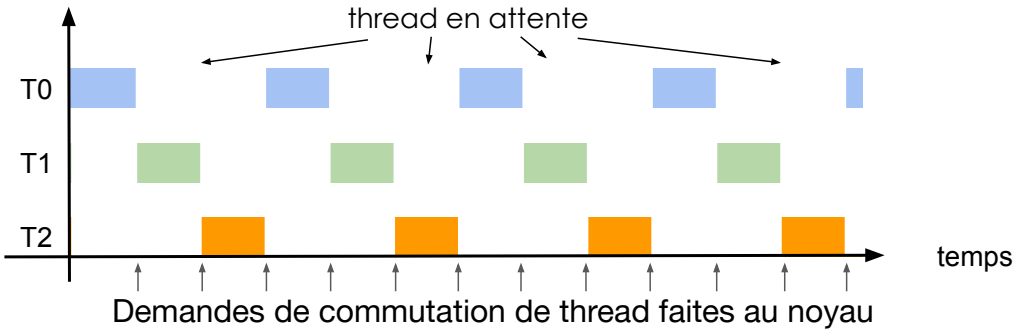
- Un processus possède au moins 1 thread, mais il peut en avoir plusieurs.
- Tous les threads partagent le même espace d'adressage
  - le même programme et chaque thread peut utiliser les fonctions qu'il veut
  - les mêmes données globales (mais ils ne doivent pas partager les piles)
  - *les mêmes propriétés du processus : état, droit, parenté, etc.*
  - *les mêmes fichiers ouverts, un thread peut ouvrir un fichier et un autre l'utiliser*
  - *les mêmes sockets réseaux pour communiquer vers l'extérieur*
  - *les threads disposent de mécanismes de synchronisation et de communication entre eux : mutex, sémaphore, fifos, etc.*

on ne va pas voir ça maintenant

- Chaque thread dispose
  - d'une fonction principale qui est l'une des fonctions du programme
  - de sa propre pile pour ses contextes d'exécution de fonctions
  - d'un état des registres du processeur en exécution et en sauvegarde
  - de propriétés : taille de pile, état d'exécution, compteurs de temps, etc.

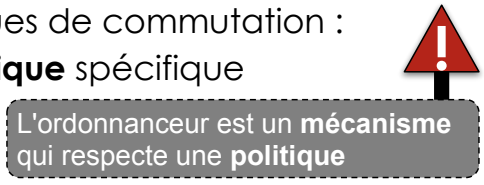
Exécution en temps partagé

- Dans une machine ne contenant qu'un seul cœur de processeur, l'exécution de plusieurs threads peut se faire en temps partagé, c.-à-d. un thread après l'autre, à tour de rôle périodiquement.
- Si un processus contient 3 threads (T0, T1 et T2) alors à un instant `t`, 1 seul des 3 threads s'exécute, les autres sont en attente du processeur.
- L'exécution en temps partagé signifie plus généralement le partage équitable du processeur entre tous les utilisateurs

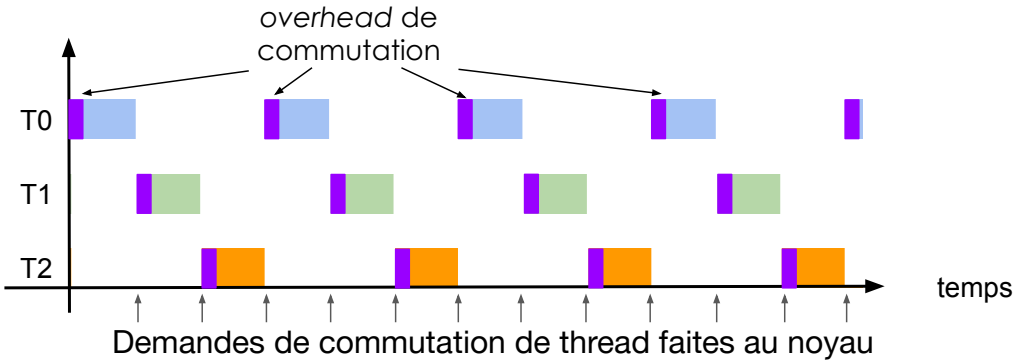


Ordonnanceur

- La commutation de threads dans le noyau est faite par l'**ordonnanceur**
- L'**ordonnanceur** reçoit des demandes périodiques de commutation :
  il **choisit** un nouveau thread selon une **politique** spécifique
  il **sauve** l'état du thread courant
  il **charge** l'état du thread élu.

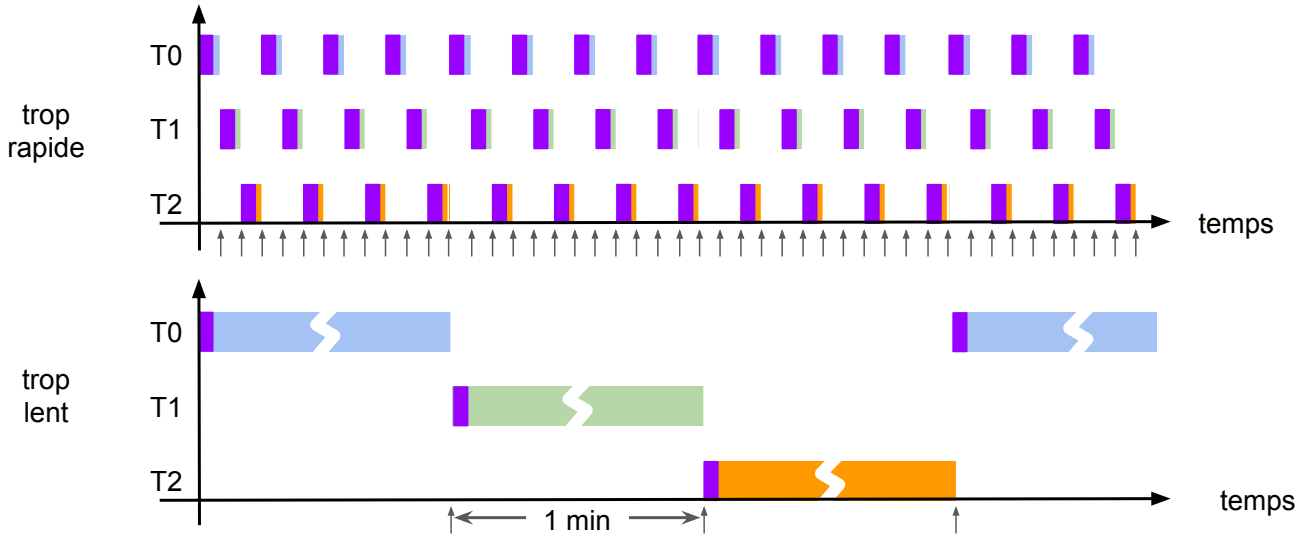


- Ce n'est pas gratuit, il y a un « ***overhead*** *cost* » (c.-à-d. des frais généraux) car le temps de commutation est un temps perdu pour l'application.



Fréquence d'ordonnancement

- La fréquence de commutation doit être assez grande pour donner l'illusion du parallélisme, mais pas trop à cause de l'overhead
- La fréquence de commutation dépend de la machine, entre 10 et 100Hz



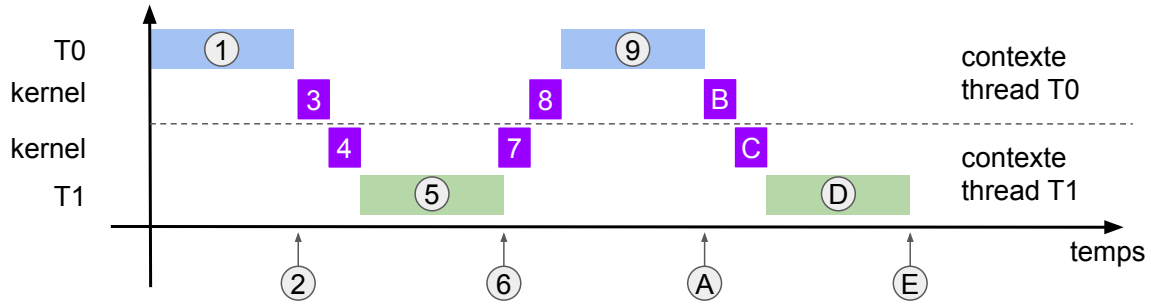
Quand faire l'ordonnancement ?

- Commutation périodique (*ou exécution en temps partagé*)
  - Une **IRQ périodique du TIMER** interrompt le thread en cours.
  - L'opération de **commutation de thread** est demandée par **l'ISR du TIMER.**
    cette opération est nommée **yield** (céder). Le thread cède le processeur et un nouveau thread, choisi par l'ordonnanceur de thread, *gagne* le processeur.
  - La **durée entre deux IRQ** du timer est nommée **tick**.
  - La **commutation est à chaque tick** ou à chaque multiple de ticks nommé **quantum**
- Commutations à l'initiative du thread (*ce n'est pas du temps partagé*)
  1. Le thread en cours peut demander lui-même la commutation de thread (avec yield),
  2. Quand un thread demande un service au noyau, mais que ce service ne peut pas être rendu immédiatement car dépendant de la disponibilité d'une ressource (p. ex. un périphérique ou un objet mémoire partagé), alors le thread peut demander au noyau de provoquer une commutation de thread (avec yield).
     Le thread cède le processeur parce qu'il ne peut plus avancer,
     il pourra à nouveau avancer lorsque la ressource attendue sera disponible
     et qu'il sera à nouveau élu par l'ordonnanceur

Comment faire la commutation ?

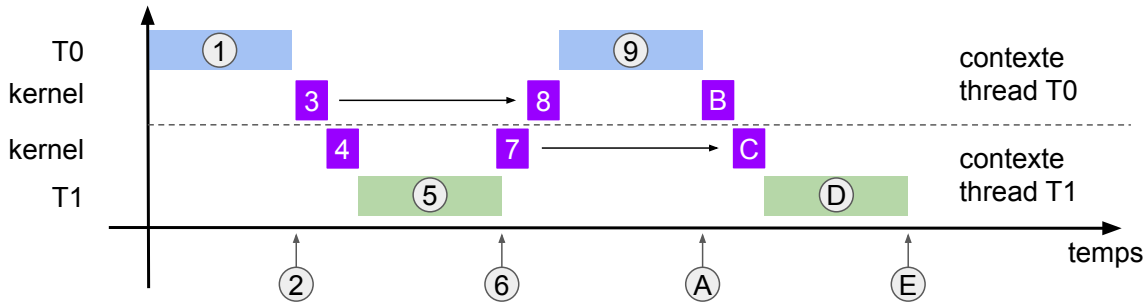
- Dans une commutation, il y a
  - le **thread sortant** qui perd le processeur,
  - le **thread entrant** qui gagne le processeur.
- Pour définir la commutation de thread, il faut :
  - déterminer le **contexte d'un thread** (l'ensemble des registres à sauver)
  - définir un **mécanisme de sauvegarde et restauration** de ces contextes,
  - définir une **politique d'ordonnancement** des threads,
    c'est-à-dire définir l'ordre dans lequel les threads doivent s'exécuter.
- La commutation entre deux threads se déroule en trois temps :
  - **élection** d'un thread entrant selon la politique d'ordonnancement,
  - **sauvegarde** du contexte (les registres du processeur) du thread sortant,
  - **chargement** du contexte du thread entrant.

Principe de la commutation de thread



- Régime stationnaire où 2 threads se partagent le processeur
  1. Le thread T0 s'exécute ①
  2. Le processeur reçoit une IRQ du timer ② qui interrompt le thread T0 (→ `kentry`)
  3. Le kernel analyse en 3 la cause et exécute l'`ISR` du TIMER (→ appelle `yield()`)
     - élection du thread entrant
     - sauvegarde des registres du thread sortant,
     - chargement des registres du thread entrant
  4. En 4, c'est la sortie de la fonction `yield()`, sortie de l'`ISR`, sortie du `kentry`
  5. Le thread T1 s'exécute en ⑤
- On continue ainsi, les étapes se répètent ⑥≡② 7≡3 8≡4 ⑨≡① etc.

Démarrage d'un thread 1/2



Dans ce chronogramme, on est en régime stationnaire.

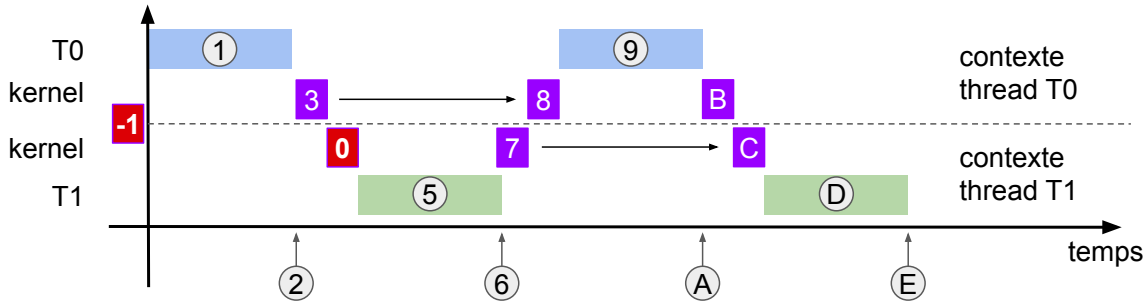
Le thread T0 qui est interrompu par l'IRQ ② entre dans `kenty` en 3 dont il sortira en 8

Ça veut dire que le thread **T1,** qui est élu puis restauré, **avait déjà été élu** et que 4 est la sortie d'une commutation ayant eu lieu dans le passé comme 8 est la sortie de 3

Quand on passe de 3 (dans T0) à 4 (dans T1), on reste dans le kernel pour sortir d'une ISR.

Mais, si T1 n'a jamais été élu dans le passé, si c'est la première fois qu'il gagne le processeur, il faut agir différemment, car on ne peut pas revenir dans une ISR !

Démarrage d'un thread 2/2



Dans ce chronogramme, on a le cas du démarrage de T0 et le démarrage T1

Le thread T0 est la fonction `main()`, il est lancée par **le lanceur du thread main** -1
ici, c'est dans `kinit()` qu'on lance l'application, c'est-à-dire le thread main

Le thread T0 qui est interrompue par l'IRQ ② entre dans kentry en 3 mais en sortant on doit appeler un **lanceur de thread** et non pas 4 parce T1 n'a jamais été commuté

Ensuite, c'est un régime stationnaire, on rentre dans 7 il y aura 8 puisqu'il y a eu un 3

Implémentation

Contraintes et choix

A ce niveau de construction du noyau, nous avons plusieurs contraintes
qui vont imposer les choix d'implémentation qui changeront à la prochaine séance !

Les Contraintes

- **Il faudrait créer dynamiquement deux structures pour chaque thread,**
  - une dans le kernel (dans `.kdata`) pour les propriétés, le contexte de registres et la pile d'exécution des services du kernel (et des interruptions)
  - une dans l'application (dans `.data`) pour la pile d'exécution de ses fonctions.
- **Le noyau devrait traiter un ensemble variable de threads,** il ne connaît pas le nombre de threads à la compilation. Or l''ordonnanceur doit parcourir l'ensemble des threads lors des commutations.

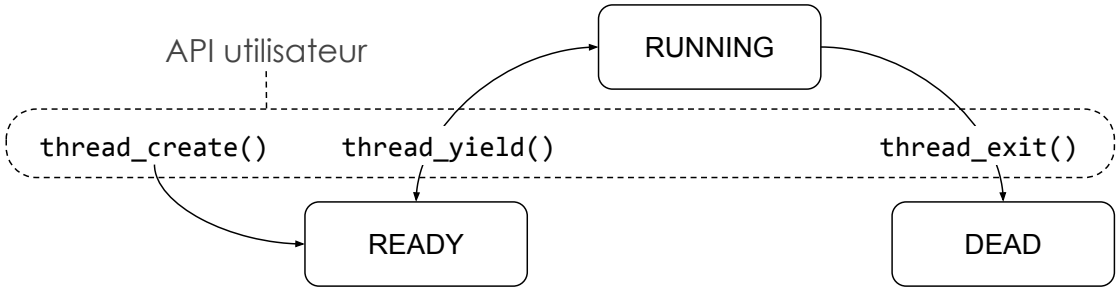
Les Choix (simplificateurs à cette étape de construction du noyau)

- Dans le code actuel, il n'y a pas encore d'allocateurs dynamiques de mémoire.
  → **On choisit de rassembler les deux structures du thread en une seule (`thread_t`) et de la mettre dans la section `.data` des variables globales de l'application (donc côté user)**
- Dans le code actuel, il n'y a pas d'API pour la gestion des listes chaînées et **on ne veut pas** faire des manipulations délicates de pointeurs (insertion, extraction, parcours, etc.)
  → **On choisit de ne pas chaîner les threads entre eux avec des pointeurs, donc le noyau a un simple tableau de taille fixe dans la section `.kdata` pointant sur les threads de l'application**

États de thread

A ce niveau de construction du noyau, les threads n'ont que 3 états

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



Il n'y a pas d'état WAIT, c'est impossible* dans cette version du noyau et la conséquence est que lorsqu'un thread attend une donnée d'un périphérique, il ne se met pas vraiment en attente, il rend le processeur qu'il reprendra plus tard pour pour retenter jusqu'à réussir, c'est de la scrutation (polling)

* Pour sortir de l'état WAIT, il faut une gestion de files d'attente pour les ressources partagées, ce n'est pas encore disponible

API utilisateur avec un exemple

Les fonctions de gestion sont réduites au minimum (3 fonctions)

1. Création
```
int thread_create (
            thread_t * thread,
            void *(*fct) (void *),
            void *arg);
```
2. Cession
```
int thread_yield (void);
```
3. Terminaison
```
int thread_exit (void *retval);
```

Dans cet exemple, `thread_yield()` est demandé par les threads, mais il est aussi imposé par le noyau à chaque **tick** dans l'ISR du timer (ici, 1 quantum = 1 tick)

```c
#include <libc.h>
#include <thread.h>

thread_t t1;

void * t1_fct (void * arg) {
    for (int i = 0; i < 5; i++) {
        fprintf (0, "[%d] t1 is alive (%d) : %d\n",
                clock(), i, (char *)arg);
        thread_yield();
    }
    thread_exit(NULL);
}

int main (void) {
    thread_create (&t1, t1_fct, "bonjour");
    for (int i = 0; i < 10; i++) {
        fprintf (0, "[%d] app is alive (%d)\n",
                    clock(), i);
        thread_yield();
    }
    return 0;
}
```

variable globale contenant la pile, les propriétés et une table pour sauver les registres..

fonction principale du thread `t1`

ici, quand on sort du thread `t1` il s'arrête et il doit être supprimé (*pas de `thread_join`*)

`thread_yield()` demande une commutation à l'ordonnanceur

quand on sort du thread `main` l'application s'arrête et la valeur de retour est celle de `main()`

API utilisateur

`thread_create` demande au noyau de créer un nouveau thread dans l'application. Elle reçoit un pointeur sur le nouveau thread (**`thread`**), un pointeur sur sa fonction (**`fct`**) et son argument (**`arg`**). Elle donne ses arguments au noyau et un pointeur sur la fonction utilisateur (**`thread_start`**)

```
int thread_create (thread_t * thread, void *(*fct) (void *), void *arg) {
    return syscall_fct ((int) thread, (int) fct, (int) arg, (int) thread_start, SYSCALL_THREAD_CREATE);
}
```

`thread_yield` demande au noyau de céder le processeur à un autre thread dans l'état READY, Le thread qui perd le processeur va passer de l'état RUNNING à l'état READY

```
int thread_yield (void) {
    return syscall_fct (0, 0, 0, 0, SYSCALL_THREAD_YIELD);
}
```

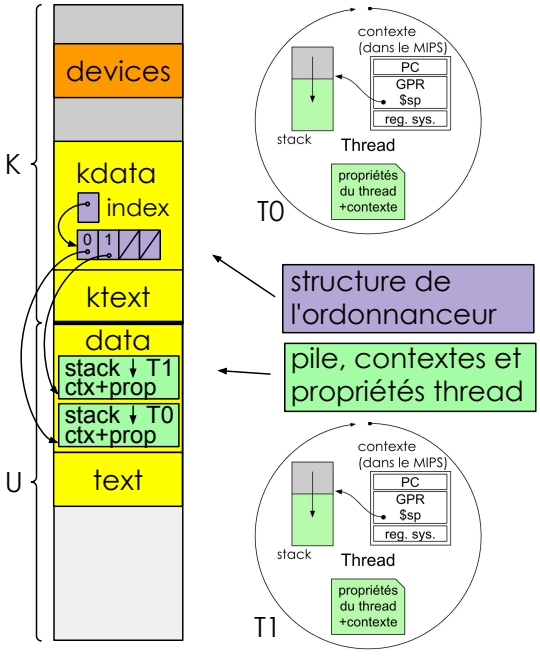
`thread_exit` demande au noyau d'arrêter le thread avec **`retval`** comme valeur de retour.

```
void thread_exit (void *retval) {
    syscall_fct ((int) retval, 0, 0, 0, SYSCALL_THREAD_EXIT);
}
```

`thread_start` est la fonction **utilisateur** qui appelle la fonction du thread **`fct(arg)`**, puis **`thread_exit()`** En fait, elle joue le rôle que la fonction **`_start()`** joue pour l'application et sa fonction **`main()`**

```
static void thread_start (void *(*fct) (void *), void *arg) {
    fct (arg);      // call the function with its arg (maybe, it will call thread_exit())
    thread_exit(0); // otherwise, if the function fct ends, then ask the kernel to exit the thread
}
```


Structures de données pour les threads



- Pour s'exécuter un thread a besoin d'au moins :
  - 3 segments d'adresses en mémoire
    - **`stack`** pile d'exécution des fonctions
    - **`data`** données globales de l'application
    - **`text`** code de l'application
  - 1 contexte pour sauver les registres
  - des propriétés (ptr fonction principale, état, …)
- Si on a plusieurs threads, ils partagent les mêmes segments **`text`** et **`data`**, mais chaque thread a son propre segment **`stack`** et ses propres propriétés.
- Pour chaque thread, une structure dans les données globales de l'application contient la pile (dans la struct) la sauvegarde du contexte et les propriétés (état, etc.)
- Un tableau de pointeurs de threads dans les données globales du kernel lui permet de les retrouver et une variable index contient le numéro du thread en cours

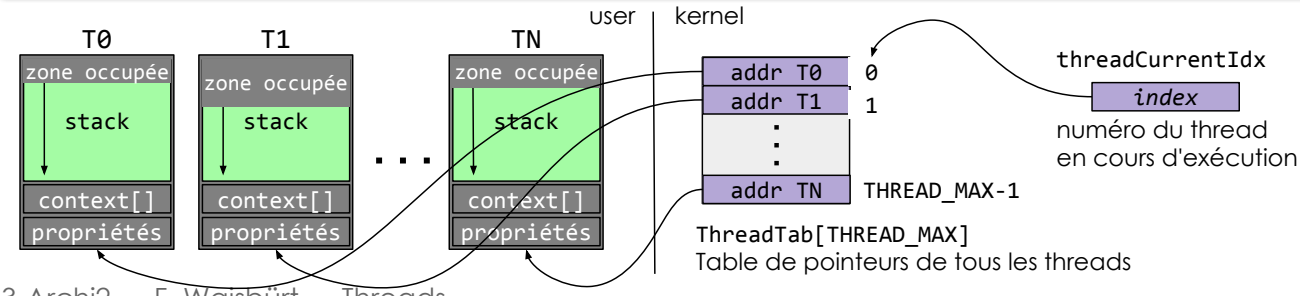
Structure thread_t et tableau ThreadTab

Thread → les structures de threads sont placées dans le segment des données user `.data`

```
typedef struct thread_s {
    int state;                  // état du thread du point de vue de l'ordonnanceur (READY-RUNNING-DEAD)
    int start;                  // ptr sur la fonction de démarrage du thread (non-écrite par le programmeur)
    int fct;      } propriétés  // ptr sur la fonction principale du thread (écrite par le programmeur)
    int arg;                    // argument du thread (un seul argument void * casté à int)
    int tid;                    // identifiant du thread
    int context[TH_CONTEXT_SIZE]; // table pour sauver les registres quand le thread perd le processeur
    int stack[THREAD_STACK_SIZE]; // pile du thread, utilisée par l'application en mode user et en mode kernel
} thread_t;
```

Ordonnanceur → variables globales dans le segment des données kernel `.kdata`

```
thread_t *ThreadTab[THREAD_MAX]; // simple table pour tous les threads de l'application (cases NULL si vide)
int      ThreadCurrentIdx;       // index, dans la table ThreadTab, du thread en cours d'exécution
```



API thread dans le noyau

Extrait de la table **SYSCALL_VECTOR[]** :

SYSCALL_THREAD_CREATE	&thread_create_kernel()
SYSCALL_THREAD_YIELD	&thread_yield()
SYSCALL_THREAD_EXIT	&thread_exit()

On rappelle que l'appel à la fonction utilisateur
syscall_fct(a0, a1, a2, a3, syscall_code) entraîne l'exécution de la fonction noyau
SYSCALL_VECTOR[syscall_code](a0, a1, a2, a3, syscall_code)

```
int thread_create_kernel (thread_t * thread, int fun, int arg, int start) {
  thread->state = TH_STATE_READY; // état initial : READY, il peut être choisir par l'électeur de l'ordonnanceur
  thread->start = start;          // fonction utilisateur de démarrage du thread : start() appelle fun(arg)
  thread->fun = fun;              // fonction principale du thread dans l'application
  thread->arg =  arg;             // argument de la fonction principale du thread
  thread->context[TH_CONTEXT_SR] = 0x413;                                    // reg. status: HWI0=1 UM=1 EXL=1 IE=1
  thread->context[TH_CONTEXT_RA] = (int) thread_bootstrap;                   // fonction d'amorçage pour la 1e fois
  thread->context[TH_CONTEXT_SP] = (int)&thread->stack[THREAD_STACK_SIZE];// ptr de pile (unique pour le moment)
  thread->stack[0] = MAGIC_TH_STACKEND;                                      // nombre magique en fin pile (debug)
  sched_insert (thread);          // insertion du thread dans l'ordonnanceur (ici c'est la table ThreadTab[])
  return 0;
}

int thread_yield (void) {
  ThreadTab[ThreadCurrentIdx]->state = TH_STATE_READY;   // L'état passe de RUNNING à READY (le thread cède juste le MIPS)
  sched_switch ();                                       // Demande à l'ordonnanceur de trouver un autre thread READY
  return 0;
}

void thread_exit (void *value_ptr) {
  ThreadTab[ThreadCurrentIdx]->state = TH_STATE_DEAD;    // L'état passe à DEAD (pour le moment, c'est tout, ça se compliquera)
  sched_switch ();                                       // Demande à l'ordonnanceur de trouver un autre thread READY
}
```

Commutation de contexte de thread

La commutation entre deux threads se déroule en 3 temps :

1. l'élection d'un thread entrant selon la politique d'ordonnancement,
2. la sauvegarde du contexte du thread sortant,
3. la chargement du contexte du thread entrant (→ état RUNNING)

L'ordonnanceur fait un parcours circulaire de la table **ThreadTab** en commençant par le numéro du thread sortant + 1, c'est une politique round-robin

La commutation est réalisée par la fonction `sched_switch()`

```
void sched_switch (void) {  //
    int th_curr = ThreadCurrentIdx;                    // n° du thread courant dans ThreadTab
1.  int th_next = sched_elect ();                      // demande le numéro du prochain thread
    if (th_next != th_curr) {                          // Si c'est le même thread, ne rien faire !
2.      if (thread_save (ThreadTab[th_curr]->context)) {   // sauve le ctx du thread sortant et rend 1
            ThreadCurrentIdx = th_next;                // mise à jour de ThreadCurrentIdx
3.          thread_load (ThreadTab[th_next]->context);     // chargement de contexte & sortie par jr $31
        }                                              // donc de thread_save() mais qui rend 0
    }
    ThreadTab[ThreadCurrentIdx]->state= TH_STATE_RUNNING;  // Le thread choisi passe dans l'état
}                                                      // TH_STATE_RUNNNIG
```

La subtilité c'est que la sortie de la fonction `thread_load()` par le `jr $31` habituel fait sortir de la fonction dont les registres ont été restaurés, DONC de `thread_save()` !
`...` sauf la première fois qu'un thread est élu puisqu'il n'a jamais été sauvé avant `...`

thread_save() & thread_load()

```
int thread_save (int context[])
int thread_load (int context[])
```

- Ce sont des fonctions écrites en assembleur dans le fichier `hcpua.S` parce qu'elles sont spécifiques au processeur.
- L'argument de ces fonctions est le pointeur sur le tableau du contexte du thread concerné

`thead_save()`

- Quand on entre dans `thread_save()`, elle sauve les registres **et la valeur de retour est 1**
- Quand on sort de `thread_save()`, on est toujours dans le même thread qu'en entrant avec la même pile, et ensuite il faut poursuivre la commutation (chargement du nouveau)

`thead_load()`

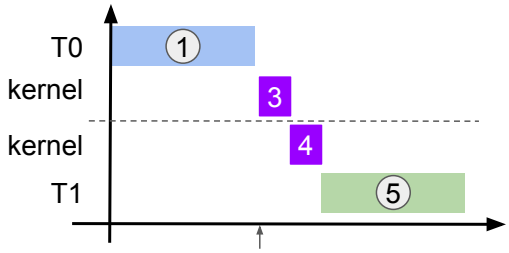
- Quand on entre dans `thead_load()`, elle charge les registres **et la valeur de retour est 0**
- Quand on sort de `thread_load()`, on est dans dans le thread élu dont les registres viennent d'être chargés, et donc, quand on est **en régime stationnaire**, on sort bien de `thread_load()` mais **on revient dans l'appelant du `thread_save()`, mais cette fois avec la valeur de retour 0**, et dans ce cas, on ne chargera évidemment pas le nouveau thread.

Contexte de thread

Le contexte d'un thread contient :

- `$16` … `$23`, `$30` : les 9 registres persistants
- `$31` : l'adresse de retour de la fonction `thread_load()`
- `c0_epc` : l'adresse de retour du service kernel courant
- `c0_sr` : le mode dans lequel on doit revenir
- `$29` : le pointeur de pile

Les registres temporaires ne sont pas sauvés parce qu'ils sont sauvés dans la pile du thread qui a perdu le processeur et ils seront restaurés quand il le regagnera !



3 Le thread T0 est dans le kernel pour un changement de contexte suite à un tick ou une demande explicite `thread_yield()`. Il appelle la fonction `sched_switch()` en sachant bien que les registres temporaires seront perdus, `sched_switch()` sauve donc seulement les registres persistants.

4 Les registres persistants de T1 sont restaurés à la fin de 3 et on revient dans 4 c.-à-d. le retour de `sched_switch()` dans le thread T1 comme si c'était une fonction normale.

Lancement d'un thread 1/3

Assertion

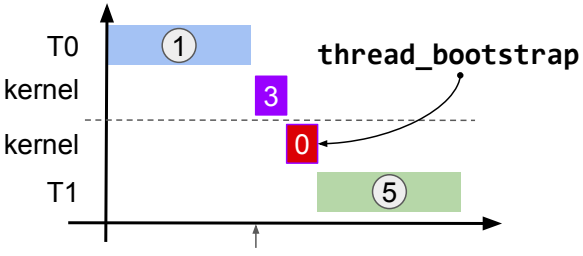
- Le chargement d'un contexte et donc l'entrée dans un thread se fait uniquement par la fonction `thread_load()`.

En régime stationnaire

- Nous venons de voir le régime stationnaire, quand on revient dans un thread qui avait vu le processeur et qui l'avait perdu dans la fonction `sched_switch()`.
  Mais comment cela se passe-t-il au départ ?

Démarrage d'un thread normal (autre que main)

- La fin de `thread_load()` est un `jr $31` et `$31` est lu dans le contexte du thread entrant. La solution est donc de mettre dans $31 l'adresse d'une fonction qui va prendre en charge le démarrage du thread.
- Cette fonction de démarrage s'appelle `thread_bootstrap()`, donc à la 1re élection d'un thread, quand on sort de `thread_load()` on entre dans `thread_bootstrap()`



Lancement d'un thread 2/3

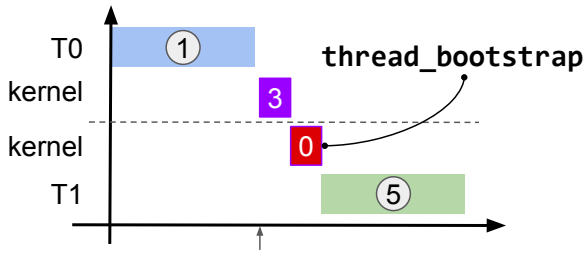
`void thread_bootstrap(void)`

- La fonction `thread_bootstrap()` est la fonction du noyau laquelle on saute à la sortie d'un `thread_load()` à la première élection d'un thread
- C'est une fonction nécessairement sans argument parce qu'on ne peut rien mettre dans les registres $4 à $7, en effet **on sort** d'un `thread_load()` qui ignore que le contexte chargé est un nouveau thread.

```
typedef struct thread_s {
    int state;
    int start;
    int fct;
    int arg;
    int tid;
    int context[TH_CONTEXT_SIZE];
    int stack[THREAD_STACK_SIZE];
} thread_t;
```

A l'initialisation de la structure thread
on met l'adresse de **`thread_bootstrap(void)`** dans **`$31`**

```
$16 … $23, $30
$31
c0_epc
c0_sr
$29
```



Lancement d'un thread 3/3

```
void thread_bootstrap (void)  {            // exécutée en sortie de thread_load() la 1ère fois
    thread_t * thread = ThreadTab [ThreadCurrentTdx]; // récupérer le ptr sur le thread courant
    thread->state = TH_STATE_RUNNING;                 // mettre à jour son état
    thread_launch (   thread->fct,         // $4 ← adresse de la fonction principale du thread
                      thread->arg,         // $5 ← argument de la fonction principale
                      thread->start);      // $6 ← adresse de la fonction user qui lance fct
}
```

```
thread_launch:        // lance la fonction utilisateur qui lance la fonction principale du thread
    mtc0  $6, $14     // EPC ← $6 qui contient l'adresse de la fonction où on veut aller
    eret              // c0_sr.EXL ← 0  &  j EPC
```

Pour le thread `main()` ,
la fonction de démarrage est :

```
void _start (void) {
    int res;
    for ( int *a = &__bss_origin;
          a != &__bss_end;
          *a++ = 0);
    res = main ();
    exit (res);
}
```

Quand on sort du main() alors on sort du processus main() peut appeler exit()

Pour un thread standard,
la fonction de démarrage est :

Type pointeur de fonction: fct est un pointeur sur une fonction qui prend un argument de type **void*** et qui rend un **void***

```
void thread_start (void *(*fct)(void *), void *arg) {
    fct (arg);
    thread_exit(0);
}
```

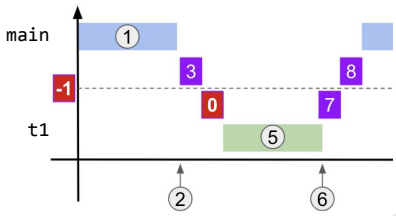
Quand on sort d'un thread alors on doit en informer le noyau. Le thread peut appeler thread_exit()

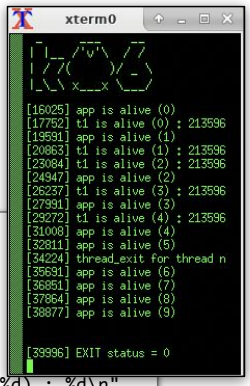
Séquence de la commutation

label0.s

```
K    12:     <boot> ---------------------- ./kernel/hcpua.S
K    37:     <kinit> --------------------- ./kernel/kinit.c
K    2184:   <arch_init> ----------------- ./kernel/harch.c
K    2299:   <thread_create_kernel> ------- ./kernel/kthread.c
K    2449:   <thread_load> ---------------- ./kernel/hcpua.S
-----------------------------------------------------------------
THREAD: 0
-----------------------------------------------------------------
K 0  2548:   <thread_bootstrap> ----------- ./kernel/kthread.c
K 0  2611:   <thread_launch> -------------- ./kernel/hcpua.S
U 0  2622:   <_start> --------------------- ./ulib/crt0.c
U 0  11957:  <main> ----------------------- ./uapp/main.c
U 0  12012:  <thread_create> -------------- ./ulib/thread.c
U 0  12040:  <syscall_fct> ---------------- ./ulib/crt0.c
K 0  12061:  <kentry> --------------------- ./kernel/hcpua.S
K 0  12084:  <syscall_handler> ------------ ./kernel/hcpua.S
K 0  12150:  <thread_create_kernel> ------- ./kernel/kthread.c
U 0  13620:  <thread_yield> --------------- ./ulib/thread.c
U 0  13647:  <syscall_fct> ---------------- ./ulib/crt0.c
K 0  13649:  <kentry> --------------------- ./kernel/hcpua.S
K 0  13654:  <syscall_handler> ------------ ./kernel/hcpua.S
K 0  13689:  <thread_yield> --------------- ./kernel/kthread.c
K 0  13738:  <sched_switch> --------------- ./kernel/kthread.c
K 0  13885:  <thread_save> ---------------- ./kernel/hcpua.S
K 0  14001:  <thread_load> ---------------- ./kernel/hcpua.S
-----------------------------------------------------------------
THREAD: 1
-----------------------------------------------------------------
K 1  14092:          <thread_bootstrap> --- ./kernel/kthread.c
K 1  14119:          <thread_launch> ------ ./kernel/hcpua.S
U 1  14130:          <thread_start> ------- ./ulib/thread.c
U 1  14153:          <t1_fct> ------------- ./uapp/main.c
U 1  15654:          <thread_yield> ------- ./ulib/thread.c
U 1  15663:          <syscall_fct> -------- ./ulib/crt0.c
K 1  15678:          <kentry> ------------- ./kernel/hcpua.S
K 1  15683:          <syscall_handler> ---- ./kernel/hcpua.S
K 1  15711:          <thread_yield> ------- ./kernel/kthread.c
K 1  15742:          <sched_switch> ------- ./kernel/kthread.c
K 1  15861:          <thread_save> -------- ./kernel/hcpua.S
K 1  15957:          <thread_load> -------- ./kernel/hcpua.S
-----------------------------------------------------------------
THREAD: 0
-----------------------------------------------------------------
U 0  17021:  <thread_yield> --------------- ./ulib/thread.c
U 0  17030:  <syscall_fct> ---------------- ./ulib/crt0.c
K 0  17035:  <kentry> --------------------- ./kernel/hcpua.S
```

-1 1 2 3 0 5 6 7 8 9






```c
#include <libc.h>
#include <thread.h>

thread_t t1;

void * t1_fct (void * arg) {
    for (int i = 0; i < 5; i++) {
        fprintf (0, "[%d] t1 is alive (%d) : %d\n",
                 clock(), i, (char *)arg);
        thread_yield();
    }
    return NULL;
}
int main (void) {
    thread_create (&t1, t1_fct, "bonjour");
    for (int i = 0; i < 10; i++) {
        fprintf (0, "[%d] app is alive (%d)\n",
                    clock(), i);
        thread_yield();
    }
    return 0;
}
```

Dans la séquence à gauche, des appels aux fonctions fprintf et clock ont été retiré pour plus de clarté

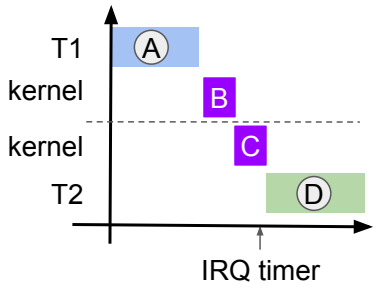
Exécution en temps partagé

- Si on veut faire un partage équitable du processeur pour tous les threads, on ne peut pas compter sur l'exécution explicite de **`thread_yield()`** par les threads eux-mêmes, il faut être plus strict.
- C'est le TIMER qui doit imposer le changement périodique de threads dans sa routine d'interruption (**ISR**) exécutée après sa requête d'interruption (**IRQ**)

```c
struct timer_s {        // Registres du TIMER
    int value;          // timer's counter : +1 each cycle, can be written
    int mode;           // timer's mode : bit 0 = ON/OFF ; bit 1 = IRQ enable
    int period;         // timer's period between two IRQ
    int resetirq;       // address to acknowledge the timer's IRQ
};
extern volatile struct timer_s __timer_regs_map[NCPUS];
void timer_isr (int timer) {
    __timer_regs_map[timer].resetirq = 1;   // Acquittement de l'IRQ
    thread_yield ();
}
```

Les étapes de traitement d'une IRQ TIMER

Un thread `T1` est en cours et l'`IRQ` du TIMER est levée (`IRQ n°0`)



B
- **déroutement** vers le noyau à l'adr. `0x80000180`
- analyse du champs `XCODE` du registre `c0_cause`
- appel du gestionnaire d'interruption
- sauvegarde des registres temporaires de `T1`
- lecture du numéro de l'`IRQ` dans l'ICU_HIGHEST (ici `0`)
- appel de l'ISR → ici c'est `timer_isr(0)`
  `IRQ_VECTOR_ISR[ICU_HIGHEST](IRQ_VECTOR_DEV[ICU_HIGHEST])`
- **execution de l'ISR**
  - acquittement de l'IRQ (c.-à-d. baisser la ligne d'interruption)
  - appel de `thread_yield()`
  - appel de `thread_switch()` **de T1**
    → élection T2, sauvegarde registres de T1, chargement registres T2
    → **fin de `thread_switch()`,fin de `thread_yield()`, fin de l'ISR de T2**

C
- restauration des registres temporaires de `T2`
- **retour** au programme interrompu dans `T2`

Commutation imposée

- Dans certains cas, un thread demande un service au noyau qui ne peut pas être rendu immédiatement, c'est le cas, par exemple, des lectures du TTY.
- La fonction utilisateur `int fgets (char *s, int size, int tty)` appelle `int tty_read (int tty, char *buf, int count)` dans le noyau qui doit remplir le buffer avec des caractères lus depuis le TTY n°`tty` jusqu'à `count` caractères.

  – `tty_read()` doit lire le clavier, mais s'il n'y a de caractère à lire, il n'est pas possible d'attendre et de garder le processeur, il faut rendre le processeur et retenter plus tard.

  – Ici c'est une lecture directe, nous allons voir une version utilisant les interruptions.

```
int tty_read (int tty, char *buf, unsigned count) {
    int res = 0;
    while (count--) {
        while (!__tty_regs_map[tty].status [tty]) {
            thread_yield(); // si rien alors partir
        }
        int c = __tty_regs_map[tty].read;// lecture
        *buf++ = c;
        res++
    }
    return res:
}
```

En résumé

- Un processus est un conteneur de ressources contenant l'espace d'adressage avec le code et les données, les entrées-sorties, des propriétés et des threads
- Un thread est une exécution du programme, il a besoin d'une pile d'exécution, d'un état de registres (dans le processeur quand il est RUNNING ou dans un tableau quand il n'est pas RUNNING) et de propriétés.
- Tous les threads partagent les ressources globales du processus
- Le noyau exécute les threads en temps partagé avec une fréquence de l'ordre de 10 à 100 Hz pour donner l'illusion du parallélisme, sans trop d'*overhead*
- C'est l'ISR du timer qui demande la commutation (yield) de thread au noyau
- Le tick est la durée entre deux interruptions du TIMER
- Pour commuter 2 threads, l'ordonnanceur dans le noyau doit élire un thread, sauver le contexte du thread sortant et charger le contexte du thread entrant
- La fonction de démarrage d'un thread appelle la fonction principale du thread puis appelle `thread_exit(0)` si ce n'est pas déjà fait par le thread lui-même

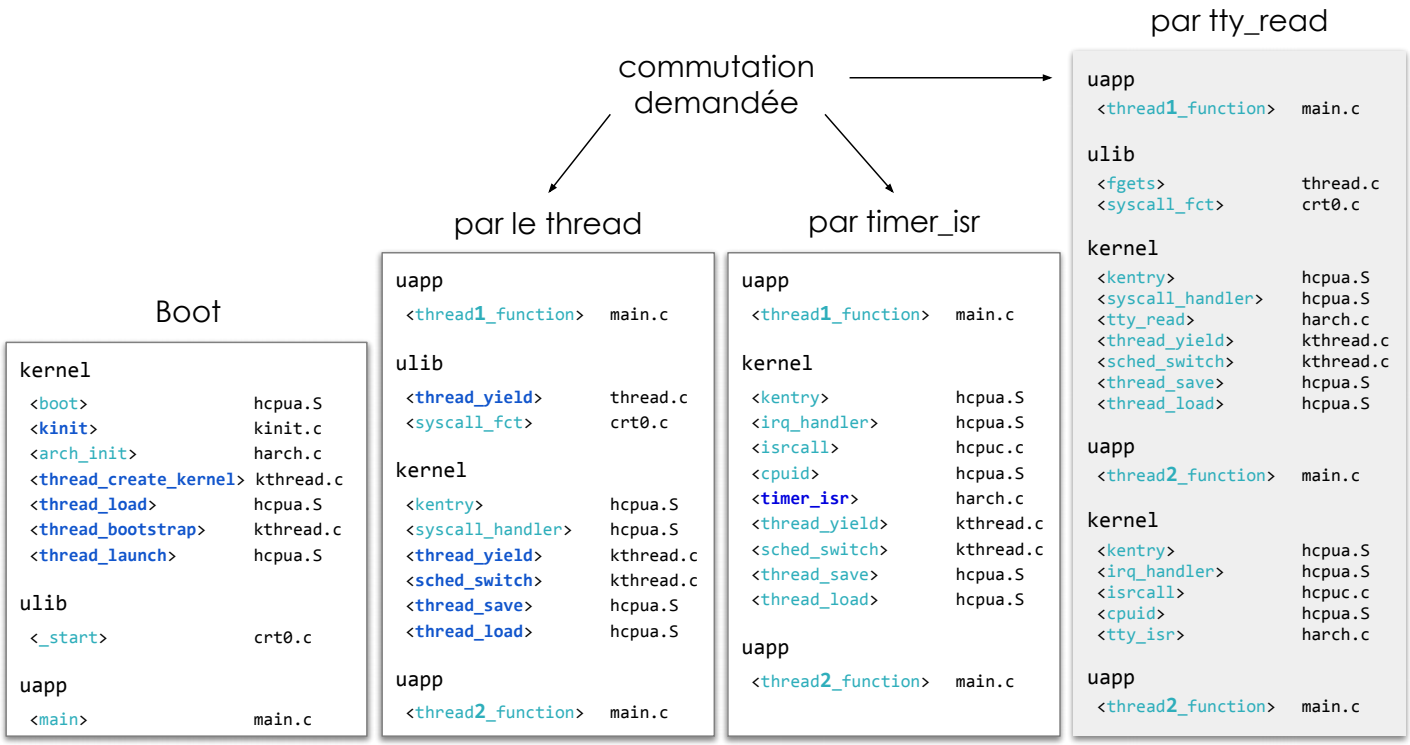
Questions

1. De quoi est composé l'espace d'adressage de l'application ?
2. Est-ce qu'un processus peut ne pas avoir de threads ?
3. Qu'est-ce que ne partagent pas deux threads du même processus ?
4. Que signifie "Exécution en temps Partagé" et Tick ?
5. Qu'est-ce que l'*overhead cost* de commutation ?
6. Pourrait-on imaginer une fréquence de commutation à 1GHz ?
7. Quel composant matériel permet la mise en œuvre du temps partagé ?
8. Que signifie thread_yield() et à quoi ça sert ?
9. Quels sont les 3 temps de la commutation de thread ?

1. L'espace d'adressage de l'application est composé de segments d'adresses dans lesquels se trouvent le code et les données.
2. Un processus est un programme en cours d'exécution, il faut au moins un thread avec sa pile d'exécution.
3. Deux threads du même processus ont des piles différentes, mais aussi un état des registres et un état d'exécution différents.
4. Exécution en temps Partagé = partage équitable du temps de processeur entre threads, Tick = période d'horloge.
5. l'overhead cost, c'est le temps perdu par le noyau pour commuter deux threads, c'est du temps perdu vu de l'application.
6. Avec un processeur cadencé à 1GHz, on demanderait la commutation toutes les instructions, c'est beaucoup trop rapide.
7. Il faut un timer qui lève une interruption périodiquement, dont l'ISR impose la commutation.
8. thread_yield(), c'est la cession du processeur par le thread en cours, c'est utile si le thread est bloqué sur une ressource.
9. Les 3 temps de la commutation de thread sont l'élection du prochain thread, la sauvegarde et la restauration de contexte

Séquences de codes

3 Séquences de code liées aux threads



boot

kinit

kinit est lancé par le boot après avoir initialisé le pointeur de pile

kernel/kinit.c

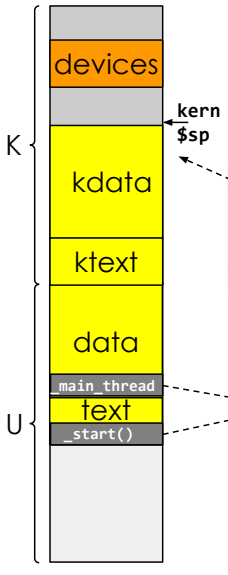
```
void kinit (void)
{
    kprintf (banner);

    extern int __bss_origin, __bss_end;
    for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);

    arch_init(20000);              // init architecture ; arg=tick

    extern thread_t _main_thread; // thread struct pour main()
    extern int _start;            // _start() point d'entrée app.

    thread_create_kernel (&_main_thread, 0, 0, (int)&_start);

    thread_load (_main_thread.context);

    kpanic();
}
```

Le kernel s'attend à trouver la fonction `_start()` au début de `.text` et la structure `thread_t` du main, au début de `.data` Il utilise le ldscript pour connaître ces deux adresses

kernel/kernel.ld

```
_start       = __text_origin;   /* expected _start() function */
_main_thread = __data_origin;   /* expected thread structure */
```

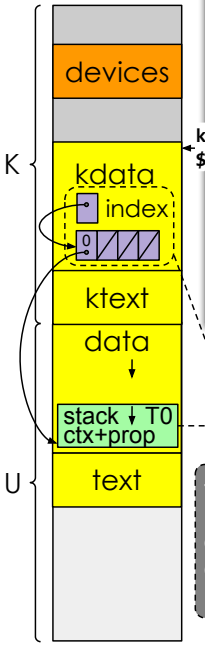
```
kernel
  <boot>                  hcpua.S
  <kinit>                 kinit.c
  <arch_init>             harch.c
  <thread_create_kernel>  kthread.c
  <thread_load>           hcpua.S
  <thread_bootstrap>      kthread.c
  <thread_launch>         hcpua.S

ulib
  <_start>                crt0.c

uapp
  <main>                  main.c
```



cette pile n'est utilisée que par kinit, elle est abandonnée dès le lancement du premier thread, définitivement

C'est lors de la création du code binaire de l'application qu'il faut placer la fonction `start()` et la structure `thread_t` du main au début des section `.text` et `.data` respectivement

thread_create_kernel



structure du scheduler volontairement simple

thread_create_kernel() initialise la structure thread et **sched_insert()** insert cette structure dans un place libre du scheduler

kernel/kinit.c

```
int thread_create_kernel (thread_t *thread, int fct, int arg, int start)
{
    thread->context[TH_CONTEXT_SR] = 0x413;                    // UM=1 EXL=1 IE=1
    thread->context[TH_CONTEXT_RA] = (int) thread_bootstrap;   // $31=thread_bootstrap
    thread->context[TH_CONTEXT_SP] = (int)&thread->stack[THREAD_STACK_SIZE]; // pile
    thread->stack[0] = MAGIC_TH_STACKEND;   // utile pour savoir si la pile a débordée

    thread->state = TH_STATE_READY;          // En attente du processeur
    thread->start = start;                   // start() appelle fct(arg)
    thread->fct = fct;                       // fonction du thread
    thread->arg =  arg;                      // argument du thread
    sched_insert (thread);                   // insert le thread dans l'ordonnanceur
    return 0;
}
```

kernel/kthread.c

```
static void sched_insert (thread_t * thread_new)
{
    int tid = 0;                                                    // thread identifier
    while ((tid < THREAD_MAX) && (ThreadTab[tid])) tid++;           // cherche une place
    if (tid == THREAD_MAX) {                                        // si plein → exit
        kprintf("[%d] to many thread created (thread.h/THREAD_MAX\n)",clock());
        exit(1);
    }
    thread_new->tid = tid;                                          // set thread ident
    ThreadTab[tid] = thread_new;                                    // store the thread
}
```

```
kernel
 <boot>                 hcpua.S
 <kinit>                kinit.c
 <arch_init>            harch.c
 <thread_create_kernel> kthread.c
 <thread_load>          hcpua.S
 <thread_bootstrap>     kthread.c
 <thread_launch>        hcpua.S
ulib
 <_start>               crt0.c
uapp
 <main>                 main.c
```

Notez qu'on n'initialise pas les registres persistants parce qu'ils ne contiennent rien et pas le registre **EPC** qui sera initialisé par **thread_launch**

thread_load → thread_bootstrap → thread_launc

kernel/hcpua.S

```
// int thread_load (int context[])
thread_load:                                // load all regs and returns 0
    lw      $16,    TH_CONTEXT_S0*4($4)     // restore all persistant registers
    lw      $17,    TH_CONTEXT_S1*4($4)
    lw      $18,    TH_CONTEXT_S2*4($4)
    lw      $19,    TH_CONTEXT_S3*4($4)
    lw      $20,    TH_CONTEXT_S4*4($4)
    lw      $21,    TH_CONTEXT_S5*4($4)
    lw      $22,    TH_CONTEXT_S6*4($4)
    lw      $23,    TH_CONTEXT_S7*4($4)
    lw      $30,    TH_CONTEXT_S8*4($4)
    lw      $27,    TH_CONTEXT_EPC*4($4)    // get next EPC addr of the new thread
    lw      $26,    TH_CONTEXT_SR*4($4)     // get next STATUS register value
    lw      $31,    TH_CONTEXT_RA*4($4)     // return addr of thread_load
    lw      $29,    TH_CONTEXT_SP*4($4)     // define the next stack pointer
    mtc0    $26,    $12                     // set STATUS register (U/K ; IRQ mode)
    mtc0    $27,    $14                     // set EPC
    li      $2,     0                       // return 0
    jr      $31                             // return to thread_save or thread_bootstrap
```

kernel/kthread.c

```
static void thread_bootstrap (void)
{
    thread_t * thread = ThreadTab [ThreadCurrentIdx];           // gets the current thread
    thread->state = TH_STATE_RUNNING;                           // the thread is now RUNNING
    thread_launch (thread->fct, thread->arg, thread->start);    // calls : start(fct,arg)
}
```

kernel/hcpua.S

```
thread_launch:          // calls _start() or thread_start(), with args in $4 and $5
    mtc0    $6,     $14 // $6 contains the pointer of the right start function --> EPC
    eret                // PC <- EPC  and SR.EXL <- 0
```

```
kernel
 <boot>                 hcpua.S
 <kinit>                kinit.c
 <arch_init>            harch.c
 <thread_create_kernel> kthread.c
 <thread_load>          hcpua.S
 <thread_bootstrap>     kthread.c
 <thread_launch>        hcpua.S
ulib
 <_start>               crt0.c
uapp
 <main>                 main.c
```

Comme c'est la première fois que le contexte est chargé **$31** contient l'adresse de **thread_boot_strap()** qui va démarrer ce nouveau thread

thread_launch() reçoit en argument la fonction user à démarrer qu'elle met dans **EPC** et cette fonction trouveras dans **$4** et **$5** la fonction principale du thread et son argument

commutation demandée
par `thread_yield()`
par `timer_isr()`

thread_yield → sched_switch

ulib/thread.c

```
int thread_yield (void)      // exécutée par l'application
{
    return syscall (0, 0, 0, 0, SYSCALL_THREAD_YIELD);
}
```

kernel/hcpua.S

```
syscall ⇒ kentry → syscall_handler → SYSCALL_VECTOR[SYSCALL_THREAD_YIELD]()
```

kernel/kthread.c

```
int thread_yield (void)      // exécutée par le kernel
{
    ThreadTab[ThreadCurrentIdx]->state = TH_STATE_READY;    // yield the CPU but always READY
    sched_switch ();                                        // Try to change thread
    return 0;                                               // 0 to tell OK
}
```

kernel/kthread.c

```
static void sched_switch (void)
{
    int th_curr = ThreadCurrentIdx;
    int th_next = sched_elect ();                           // get the current thread index
    if (th_next != th_curr) {                               // get a next ready thread
        if (thread_save (ThreadTab[th_curr]->context)){     // if it is not the same
            ThreadCurrentIdx = th_next;                     // Save current context, return 1
            thread_load (ThreadTab[th_next]->context);      // update ThreadCurrentIdx
        }                                                   // load context, goto new thread
    }                                                       // but with 0 as return value
    ThreadTab[ThreadCurrentIdx]->state= TH_STATE_RUNNING;   // the chosen one is RUNNNIG
}
```

trouve **toujours** un thread ready eventuellement c'est lui-même !

```
uapp
 <thread1_function>      main.c

ulib
 <thread_yield>          thread.c
 <syscall_fct>           crt0.c

kernel
 <kentry>                hcpua.S
 <syscall_handler>       hcpua.S
 <thread_yield>          kthread.c
 <sched_switch>          kthread.c
 <thread_save>           hcpua.S
 <thread_load>           hcpua.S

uapp
 <thread2_function>      main.c
```

Ce qu'il faut bien comprendre, c'est qu'on entre dans **`thread_save()`** pour sauver le contexte du thread courant et donc on ressort dans le même thread ! Et pour le dire, **`thread_save()` rend 1**

En revanche, on entre dans **`thread_load()`** pour charger le contexte du prochain thread et donc quand on sort, on est plus dans le même thread ! Dans le cas général, on sort … d'un **`thread_save()`** puisqu'on charger un état sauvegardé, mais cette fois **elle rend 0** (sauf au premier chargement d'un thread, on sort dans **`thread_bootstrap()`**)

On teste le retour de **`thread_save()`** pour savoir s'il doit appeler **`thread_load()`.**

thread_save → thread_load

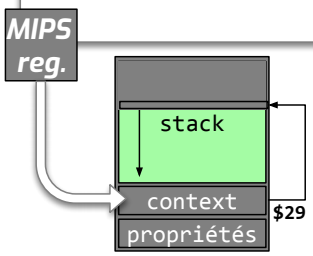
```
uapp
 <thread1_function>      main.c

ulib
 <thread_yield>          thread.c
 <syscall_fct>           crt0.c

kernel
 <kentry>                hcpua.S
 <syscall_handler>       hcpua.S
 <thread_yield>          kthread.c
 <sched_switch>          kthread.c
 <thread_save>           hcpua.S
 <thread_load>           hcpua.S

uapp
 <thread2_function>      main.c
```

kernel/hcpua.S

```
// int thread_save (int context[])
thread_save:
    mfc0    $26,    $12  // SR:  status
    mfc0    $27,    $14  // EPC: return addr
    sw      $16,    TH_CONTEXT_S0*4($4)
    sw      $17,    TH_CONTEXT_S1*4($4)
    sw      $18,    TH_CONTEXT_S2*4($4)
    sw      $19,    TH_CONTEXT_S3*4($4)
    sw      $20,    TH_CONTEXT_S4*4($4)
    sw      $21,    TH_CONTEXT_S5*4($4)
    sw      $22,    TH_CONTEXT_S6*4($4)
    sw      $23,    TH_CONTEXT_S7*4($4)
    sw      $30,    TH_CONTEXT_S8*4($4)
    sw      $26,    TH_CONTEXT_SR*4($4)
    sw      $27,    TH_CONTEXT_EPC*4($4)
    sw      $31,    TH_CONTEXT_RA*4($4)
    sw      $29,    TH_CONTEXT_SP*4($4)
    li      $2,     1   // return 1
    jr      $31         // goto caller
```

kernel/hcpua.S

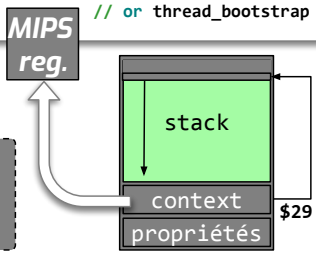
```
// int thread_load (int context[])
thread_load:
    lw      $16,    TH_CONTEXT_S0*4($4)
    lw      $17,    TH_CONTEXT_S1*4($4)
    lw      $18,    TH_CONTEXT_S2*4($4)
    lw      $19,    TH_CONTEXT_S3*4($4)
    lw      $20,    TH_CONTEXT_S4*4($4)
    lw      $21,    TH_CONTEXT_S5*4($4)
    lw      $22,    TH_CONTEXT_S6*4($4)
    lw      $23,    TH_CONTEXT_S7*4($4)
    lw      $30,    TH_CONTEXT_S8*4($4)
    lw      $26,    TH_CONTEXT_SR*4($4)
    lw      $27,    TH_CONTEXT_EPC*4($4)
    lw      $31,    TH_CONTEXT_RA*4($4)
    lw      $29,    TH_CONTEXT_SP*4($4)
    mtc0    $26,    $12 // SR:  status
    mtc0    $27,    $14 // EPC: return addr
    li      $2,     0   // return 0
    jr      $31         // goto thread_save
                        // or thread_bootstrap
```



thread sortant

Notez bien, que **thread_save()** rend **1**
thread_load() rend **0**

Quand on change de thread, et qu'on commute les contextes on change de pile puisque **$29** ne pointe plus au même endroit



thread entrant

On sort de **thread_load()** avec **jr $31** pris dans le contexte restauré.

S'il a été rempli par **thread_save()**
alors
retour dans **sched_switch()**
sinon
on va dans **thread_bootstrap()** puisque c'est la valeur initiale de **$31** donnée par **thread_create()**

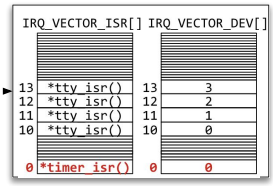
* → kentry → irq_handler → isrcall → timer_isr

U/K | Kernel

kernel/hcpua.S

```
// c0_cause.XCODE contient 0 (car IRQ)
// c0_EPC contient l'adresse de la prochaine instruction
// c0_SR.EXL est à 1 → mode kernel avec IRQ masquées
kentry:
    mfc0    $26,    $13             // $26 ← c0_CAUSE
    andi    $26,    $26,    0x3C    // $26 ← XCODE * 4
    li      $27,    0x20            // $27 ← 8 * 4 (syscall)
    beq     $26,    $27,  syscall_handler // si syscall
    beq     $26,    $20,  irq_handler     // si IRQ
    j       kpanic                        // exception
syscall_handler:
    // code du gestionnaire de syscall
irq_handler:
    // 20 regs to save (17 tmp regs+HI+LO+$31)
    addiu   $29,    $29,    -20*4
    sw      $31,    19*4($29)       // $31 lost by jal
    // save 17 tmp reg: $1 à $15, $24, $25
    sw      $1,     1*4($29)
    [...]
    sw      $25,    17*4($29)
    mflo    $2                      // get LOW
    mfhi    $3                      // get HIGH
    sw      $2,     19*4($29)       // save LOW
    sw      $3,     0*4($29)        // save HIGH
    jal     isrcall                 // call the right ISR
```

Vecteur d'interruption utilisé par **isrcall()** pour appeler la bonne ISR sur le bon périph.



kernel/harch.c

```
void isrcall (void) {
    int irq = icu_get_highest (cpuid());
    irq_vector_isr[irq] (irq_vector_dev[irq]);
}
```

kernel/harch.c

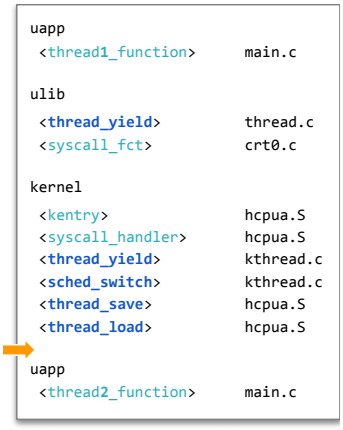
```
void timer_isr (int timer) {
    __timer_regs_map[timer].resetirq = 1; // acknowledge
    thread_yield ();
}
```

Vous pouvez revenir 2 slides en arrière pour voir le comportement de **thread_yield()** du kernel, c'est le même !

- Le thread **sortant** passe de RUNNING à **READY**
- On appelle **sched_switch()**
  - **sched_elect()**
  - **thread_save()**
  - **thread_load()**
- Le thread **entrant** passe de READY à **RUNNING**

Mais que se passe-t-il après **thread_load() ?**

en sortie de `thread_load` → `eret`



À la sortie de `thread_load()`, il y a 2 possibilités :

1. Le thread entrant est nouveau, il n'a jamais vu le processeur
   → On entre dans `thread_bootstrap()`
   → puis dans `thread_launch()` comme c'est expliqué slide 40
2. Le thread entrant avait exécuté `sched_switch()`
   et donc la sauvegarde du contexte avec `thread_save()`,
   → on revient donc dans `sched_switch()`, en sortant de `thread_save()` avec `0`
   → puis on revient dans l'appelant de `sched_switch()`, ici, juste `thread_yield()`
   → puis on revient dans l'appelant de `thread_yield()` qui peut être :
   - un service syscall (p. ex. `thread_yield()` kernel)
     → on retourne dans `syscall_handler` qui rétablie la pile et exécute `eret`
   - l'ISR du timer `timer_isr()`
     → on retourne dans `syscall_handler()` qui rétablie la pile et exécute `eret`

Conclusion

Vous avez vu en détail comment les threads utilisent le processeur à tour de rôle à une fréquence suffisamment élevé pour donner l'illusion d'une exécution parallèle, mais pas trop pour éviter de perdre du temps pendant la commutation.

Mais, le code est encore simple…

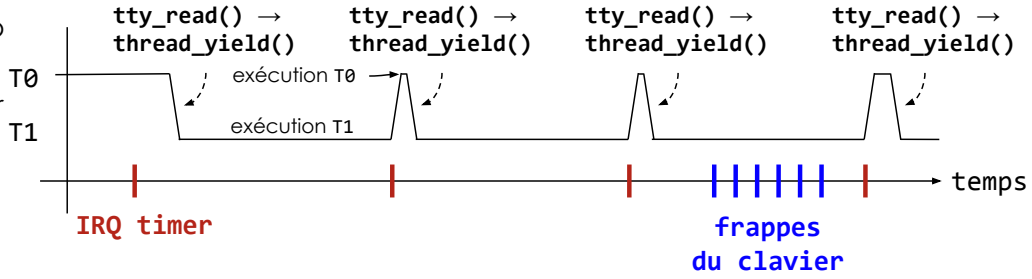
- **La structure `threads` est ici entièrement dans l'espace utilisateur parce que le noyau n'a pas de mémoire dynamique pour créer une structure dans son espace.**
  → Il faut ajouter des allocateurs de mémoire dynamique dans le noyau.
- **Les threads sont toujours READY, s'ils attendent une ressource (un caractère du clavier), ils doivent reprendre le processeur et la redemander jusqu'à satisfaction, c'est du *polling*.**
  → Il faut ajouter une gestion de listes dans le noyau pour qu'un thread puissent passer dans un état WAIT après s'être enregistré dans une liste d'attente
- **Les threads ne partagent pas de données et ne peuvent pas se synchroniser pour réaliser chacun une partie d'une application.**
  → Il faut ajouter des mécanismes de communication et de synchronisation en s'appuyant sur les mécanismes offerts par le processeur et l'architecture.

TME

vous allez programmer une gestion plus propre de la fonction `tty_read()`, en effet dans la 1ère version proposée, `tty_read()` tente de lire le registre `status` pour savoir s'il y a un caractère en attente, sinon elle cède le processeur avec `thread_yield()`, le problème est que l'on peut perdre des frappes du clavier.
Ici `T0` appelle `tty_read()` qui cède le processeur à `T1` en l'absence de frappes.

```
int tty_read (int tty, char *buf, unsigned count) {
    int res = 0;
    while (count--) {
        while (!__tty_regs_map[tty].status [tty]) {
            thread_yield(); // si rien alors partir
        }
        int c = __tty_regs_map[tty].read;// lecture
        *buf++ = c;
        res++
    }
    return res:
}
```

Mais, s'il y a beaucoup de frappes pendant que `T1` a le processeur alors on pourrait perdre des frappes.



Questions

Lorsqu'on démarre un thread, il faut passer par une fonction `thread_start()`

1. Que fait la fonction `thread_start()` ?
2. Est-ce que la fonction `thread_start()` est exécutée en mode user ?

La fonction `thread_save()` sauvegarde les registres du processeur du thread sortant

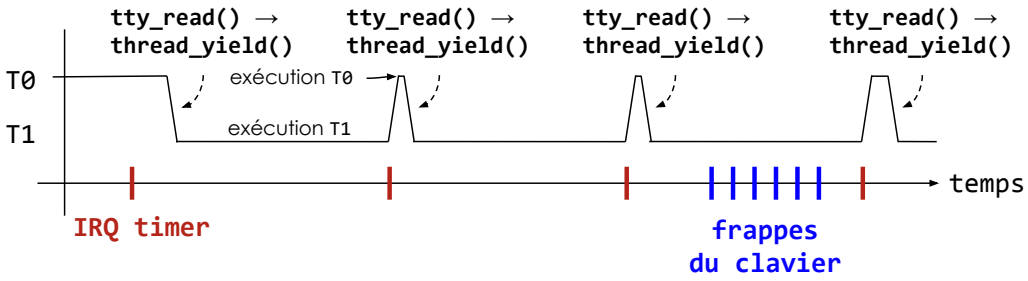
3. Faut-il sauvegarder tous les registres GPR ?
4. Pourquoi sauvegarder le registre status (`c0_sr`)

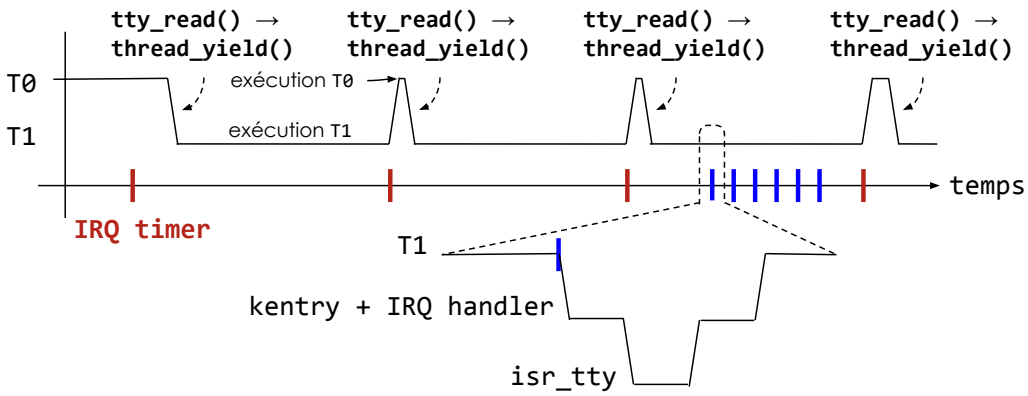
La fonction `thread_load()` charge les registres du processeur du thread entrant

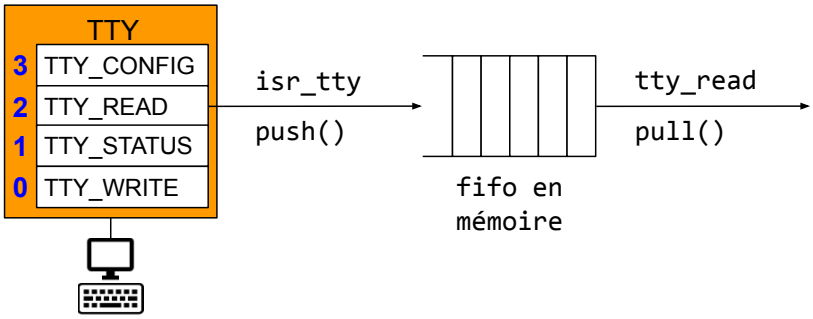
5. Quand on sort de thread_load(), où va-ton ?

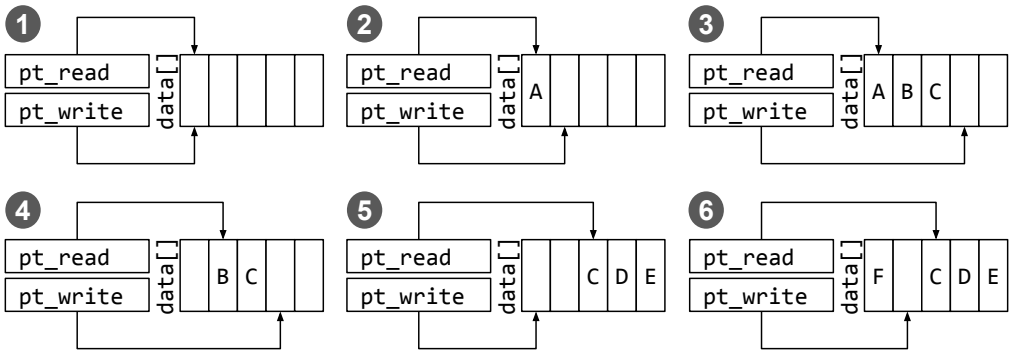
1. `thread_start()` appelle la fonction principale du thread et si on sort de cette fonction, `thread_start()` appelle `thread_exit()`
2. Oui, c'est une fonction utilisateur. Si c'était une fonction kernel, elle ne pourrait pas appeler la fonction principale du thread avec un simple appel de fonction en C.
3. Non, on ne sauve pas les registres temporaires, Pour le langage C, `thread_save()` est une fonction comme les autres qui n'a pas à sauver les registres temporaires mais seulement les registres persistants. Quand on exécute thread_save(), on a un certain état dans la pile du thread et quand on revient après un `thread_load()`, c'est comme si on n'avait pas perdu le processeur.
4. Il faut sauver le status car, c'est lui qui dit dans quel mode se trouve le MIPS, (user ou kernel), il faut revenir dans le même mode.
5. Quand on revient de `thread_load()`, on est dans le thread entrant, soit dans `thread_bootstrap()` si c'est la première fois que le thread est chargé, soit dans `thread_switch()` mais on sort de `thread_save()`, en réalité.

Slides en plus…






TTY
3
TTY_CONFIG
2
TTY_READ
1
TTY_STATUS
0
TTY_WRITE
isr_tty
push()
fifo en
mémoire
tty_read
pull()


1
pt_read
pt_write
data[]
2
pt_read
pt_write
data[]
A
3
pt_read
pt_write
data[]
A B C
4
pt_read
pt_write
data[]
B C
5
pt_read
pt_write
data[]
C D E
6
pt_read
pt_write
data[]
F C D E