Interruptions & threads

LU3IN029 Archi L3 S5

franck.wajsburt@lip6.fr
6 décembre 2021

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 → nous allons voir comment !

Questions

- Comment les interruptions sont-elles provoquées
  et comment le kernel fait-il pour les traiter ?
- Une fonction en cours d'exécution se nomme thread, « *fil d'exécution* ».
  Dans notre plateforme, il n'y a qu'un seul processeur et pourtant nous souhaitons exécuter plusieurs fonctions de l'application en parallèle, c'est impossible, mais on peut en donner l'illusion, comment faire ?

Plan

Interruptions

Qu'est-ce qu'une interruption ?

- Une interruption, c'est la suspension de l'exécution de l'application en cours sur le processeur pour accomplir une tâche de plus haute priorité ou pour changer de ***thread**** ou même pour changer d'application.
- Ce sont les périphériques qui font des requêtes d'interruption au processeur.
- Le terme requête signifie demande, ordre, appel, etc. mais on utilise requête.
- Une requête d'interruption (**IRQ** pour Interrupt ReQuest) est transmise par un signal électrique à 2 états : un état inactif (baissé) et un état actif (levé).
- On dit qu'on lève une **IRQ** ou qu'on active une **IRQ**, c'est un état (non transitoire)
- Une **IRQ** doit **rester levée**/active **tant qu'elle n'est pas traitée** par le kernel.
- La suspension du programme en cours permet d'exécuter une **ISR** (**I**nterrupt **S**ervice **R**outine) dans le noyau pour traiter l'**IRQ**
- L'**ISR** communique avec le périphérique qui a levé/activé l'**IRQ** au moins pour lui demander de la baisser/désactiver → On dit que l'**ISR acquitte l'IRQ**

** Nous allons voir le principe des threads un peu plus loin*

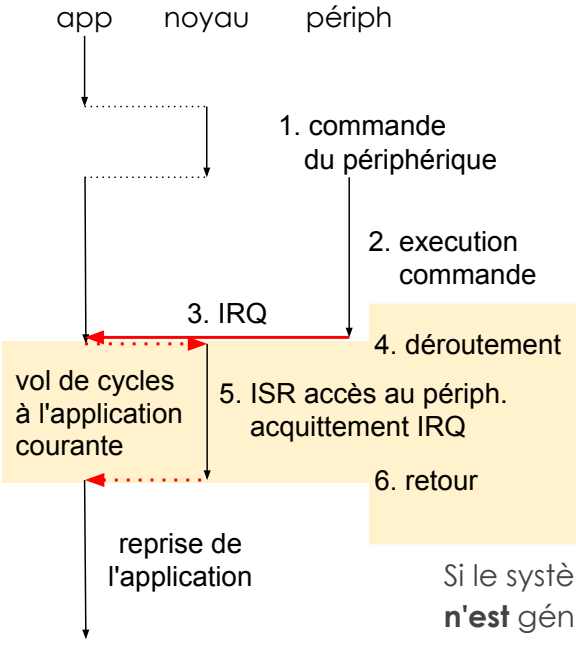
Qu'est-ce qu'une interruption ?

Une interruption est la conséquence d'un événement matériel.

D'où vient cet événement ?

- Il est causé par un composant (périphérique) qui signale ainsi
  - que ce qu'on lui a demandé (une commande) est terminé
  - ou qu'une donnée est prête à être lue
  - ou encore qu'il est prêt à recevoir une commande ou une donnée
- Un composant reçoit d'abord une commande du noyau et le composant signale par une IRQ que cette commande est traitée.
- Une IRQ est donc toujours attendue par le noyau puisque c'est la conséquence d'une commande → une IRQ non attendue est donc une erreur.
- *Notez que dans les architectures à plusieurs processeurs (multicores), un processeur peut envoyer une IRQ vers un autre processeur en écrivant dans un composant dédié → Ces sont des **IPI** (**I**nter **P**rocessor **I**nterrupt).*

Qu'est-ce qu'une interruption ?



Commande (cause) et interruption (conséquence)

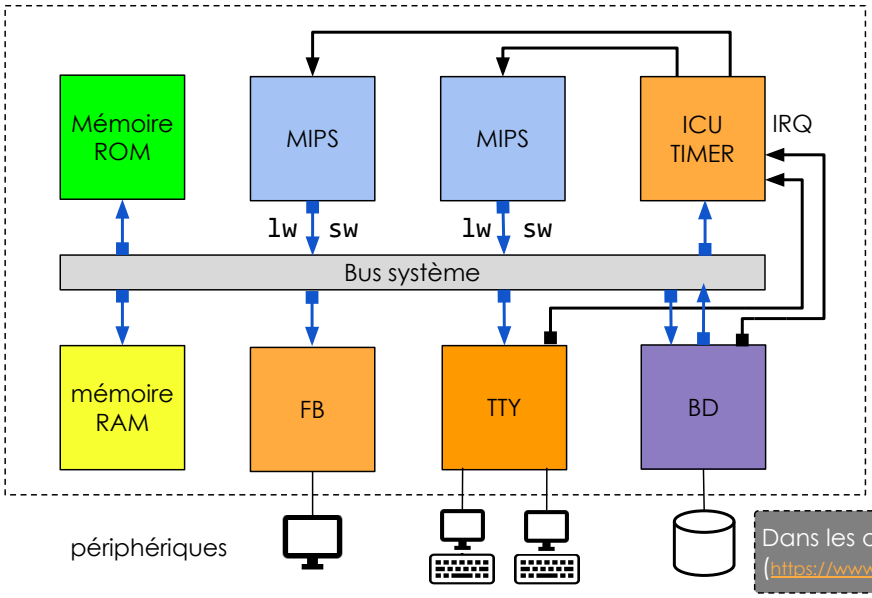
1. **Commande** du noyau
   - écriture/lecture dans les registres du contrôleur de périphérique (avec des instructions `sw`/`lw`)
2. **Exécution** de la commande
   - opération du périphérique
   - attente de réception d'une donnée
   - attente de l'acceptation d'une donnée
3. **Activation IRQ** Interrupt ReQuest
4. **Déroutement** vers le noyau
5. **Exécution de l'ISR** c.-à-d. la routine d'interruption
   - traitement de la requête (read ou write)
   - **Acquittement de l'IRQ** (baisse/désactivation)
6. **Retour** à l'application

Si le système exécute plusieurs applications, celle interrompue **n'est** généralement **pas** celle qui a fait la commande
⇒ c'est un **vol de cycle** pour l'application en cours

Interruptions vue du matériel

Routage matériel des IRQ en général

Les IRQ transitent par un un composant spécial :
l'**ICU I**nterrupt **C**ontroler **U**nit



Dans le cas général, il y a plusieurs MIPS, en TP, il n'y en a qu'1 !

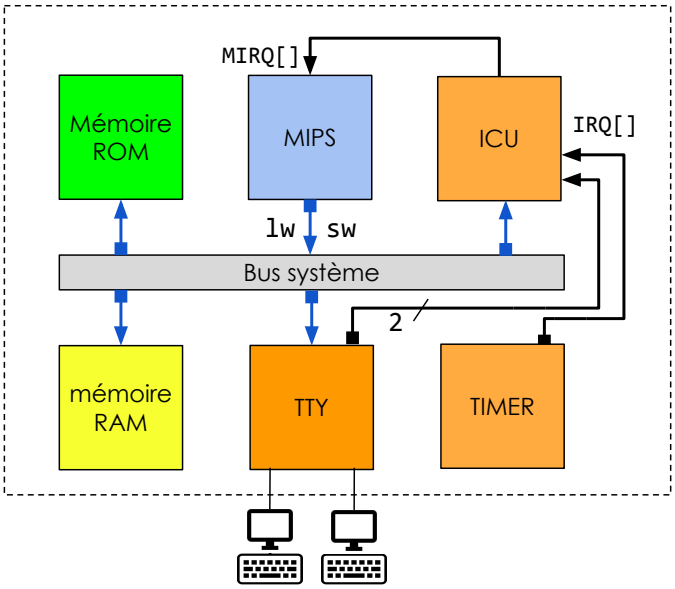
S'il y a plusieurs MIPS, chaque MIPS gère un sous-ensemble d'IRQ

Le rôle de l'ICU est de faire le routage des IRQ vers le MIPS concerné.

Dans les ordinateurs PC, l'ICU est nommé PIC (https://www.wikiwand.com/en/Programmable_interrupt_controller)

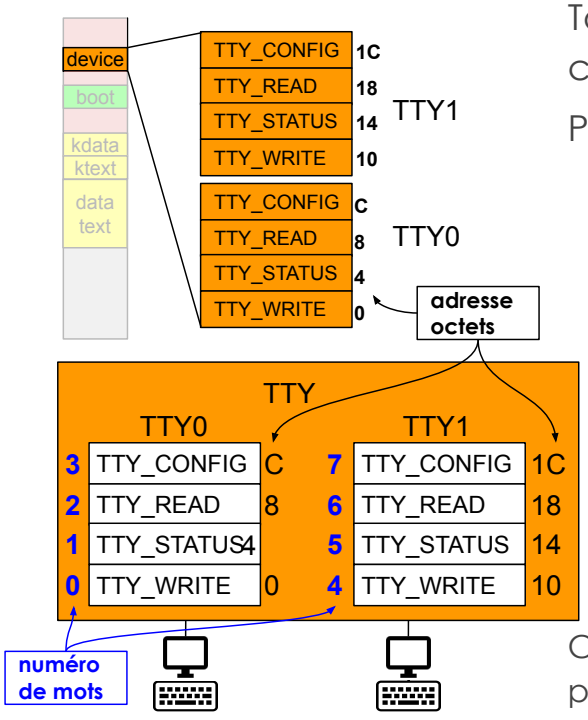
Routage matériel des IRQ pour les TPs

il y a 1 seul MIPS et seuls 2 composants peuvent lever des IRQ : TTY et Timer



- Le contrôleur de terminaux TTY a autant de signaux d'IRQ qu'il y a de TTY.
  En TP, jusqu'à 4 TTY et 2 sur le schéma
- Le composant TIMER peut lever plusieurs signaux d'IRQ périodiquement.
  En TP, il n'y a qu'1 seul TIMER et 1 seule IRQ
- L'ICU peut router jusqu'à 32 signaux d'IRQ (pins `IRQ[]`) venant des contrôleurs de périphériques (TTY, TIMER, etc.) vers le MIPS
  Elle peut *masquer* chacun des signaux d'IRQ et indiquer le numéro de l'IRQ active la plus prioritaire.
  En TP, il y a 5 IRQ max (4 TTY et 1 TIMER)
- L'IRQ en sortie de l'ICU entre sur l'une des 6 entrées d'IRQ du MIPS (pins `MIRQ[]`).
  En TP, on n'utilise que l'entrée (`MIRQ[0]`)

Contrôleur de terminaux TTY



Tous les registres sont alignés sur des mots, chaque terminal utilise un segment de 4 mots.

Pour chaque terminal

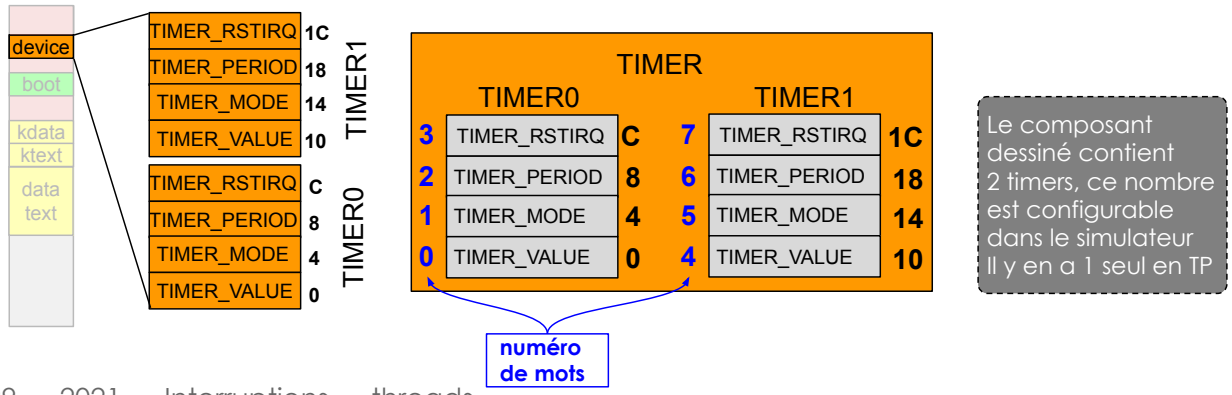
- TTY_WRITE 1 mot en écriture seule, le caractère ascii est mis dans l'octet de poids faible → sortie vers l'écran
- TTY_STATUS 1 mot en lecture seule, ≠ 0 s'il y a un caractère en attente dans TTY_READ
- TTY_READ 1 mot en lecture seule, le caractère tapé est dans l'octet de poids faible
- TTY_CONFIG inutilisé dans cette version, mais permet la configuration p. ex. du débit d'échange avec le terminal

Chaque TTY lève une IRQ si un caractère est reçu par le TTY donc si son STATUS est ≠ de 0

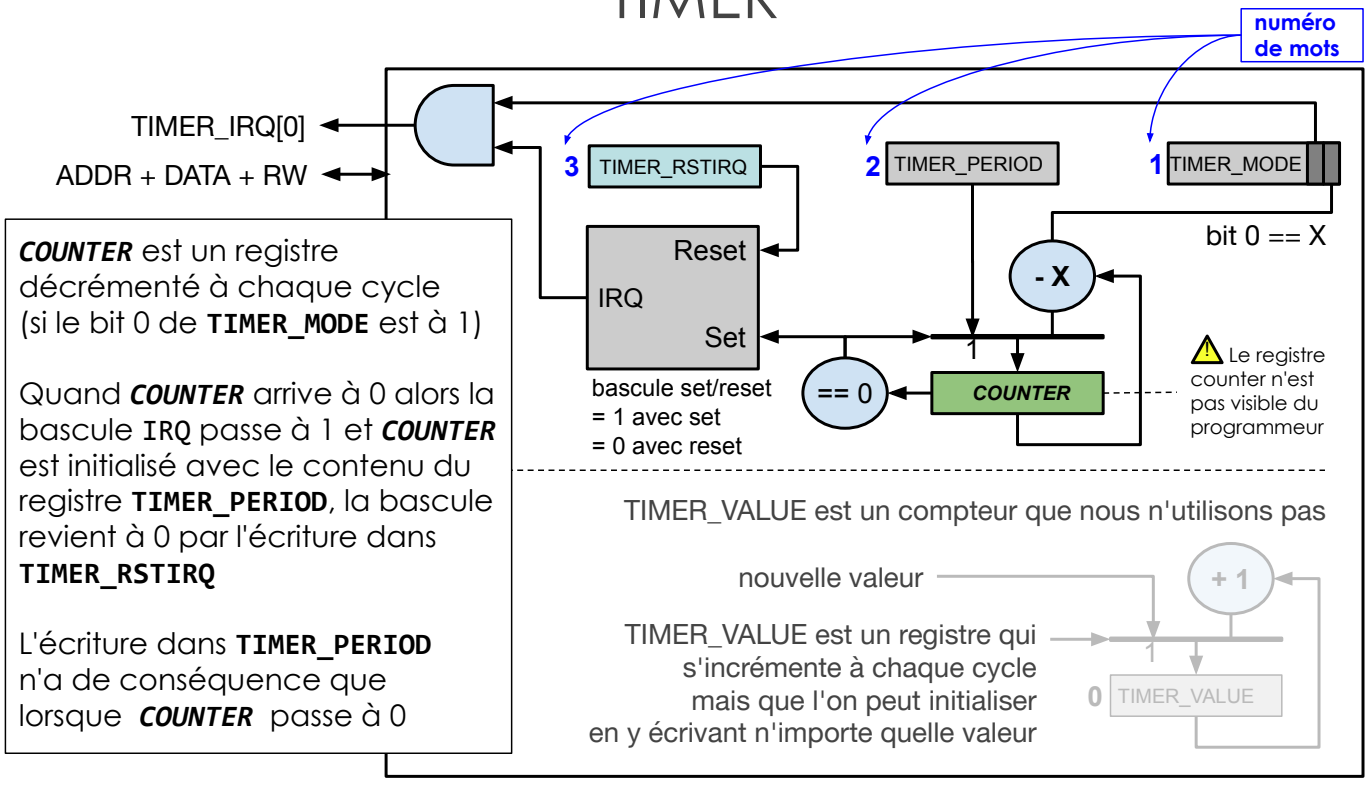
TIMER

Le TIMER contient des compteurs de temps qui peuvent lever des interruptions périodiques.
C'est un périphérique cible contrôlé par des accès en lecture / écriture dans ses registres.

- TIMER_VALUE (lecture/écriture) +1 à chaque cycle
- TIMER_MODE (écriture seule) configure le mode de fonctionnement
  Bit 0 : 1 → timer en marche (décompte) ; 0 → timer arrêté
  Bit 1 : 0 → pas d'IRQ quand le compteur atteint 0
- TIMER_PERIOD (écriture seule) période entre 2 IRQ
- TIMER_RESETIRQ (écriture seule) **écrire à cette adresse acquitte l'IRQ**



TIMER



COUNTER est un registre décrémenté à chaque cycle (si le bit 0 de **TIMER_MODE** est à 1)

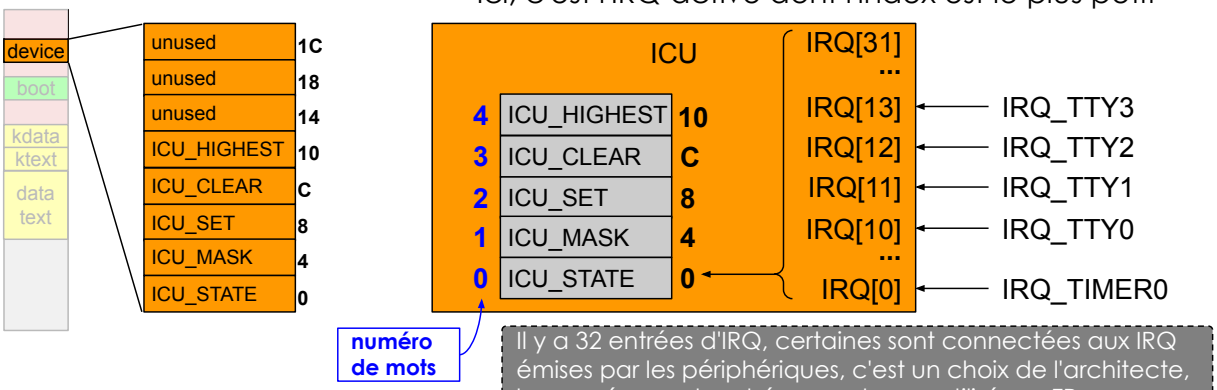
Quand ***COUNTER*** arrive à 0 alors la bascule **IRQ** passe à 1 et ***COUNTER*** est initialisé avec le contenu du registre **TIMER_PERIOD**, la bascule revient à 0 par l'écriture dans **TIMER_RSTIRQ**

L'écriture dans **TIMER_PERIOD** n'a de conséquence que lorsque ***COUNTER*** passe à 0

Interrupt Controller Unit ICU

L'ICU est un concentrateur de signaux d'IRQ. Chaque IRQ peut être masquée.
C'est un périphérique cible contrôlé par des lectures / écritures dans ses registres.

- ICU_STATE (lecture seule) état des lignes IRQ
- ICU_MASK (lecture seule) masques des lignes IRQ (sélection des IRQ désirées)
- ICU_CLEAR (écriture seule) commande de mise à 0 des masques d'IRQ
- ICU_SET (écriture seule) commande de mise à 1 des masques d'IRQ
- ICU_HIGHEST (lecture seule) **numéro de la ligne IRQ active la plus prioritaire**
  Ici, c'est l'IRQ active dont l'index est le plus petit

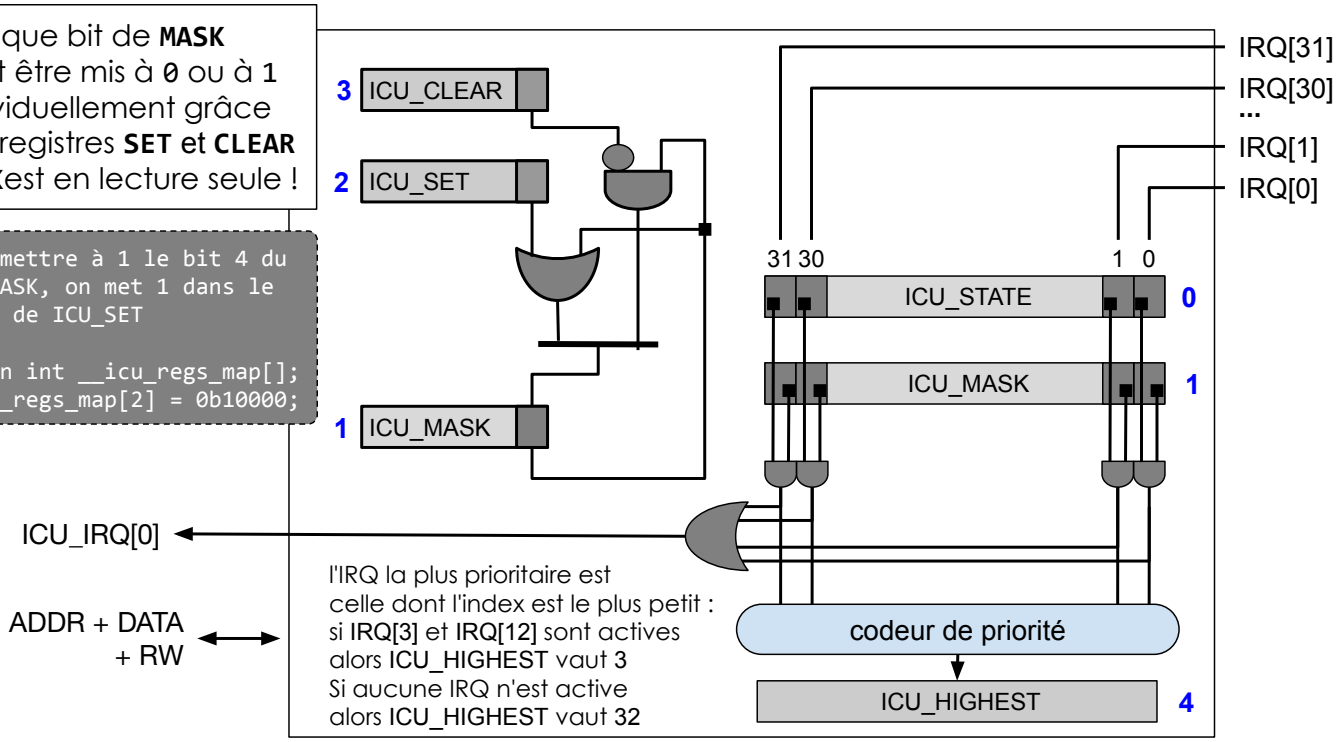


Interrupt Controller Unit ICU

Chaque bit de **MASK** peut être mis à **0** ou à **1** individuellement grâce aux registres **SET** et **CLEAR**
MASKest en lecture seule !

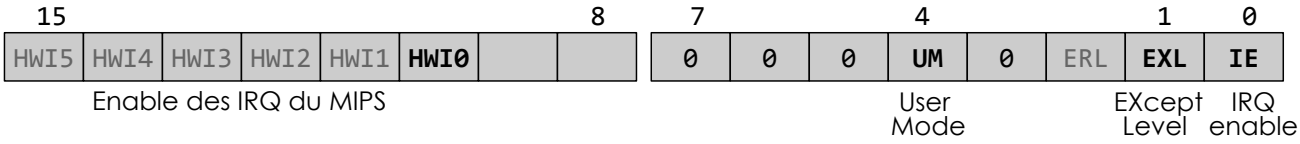
```
Pour mettre à 1 le bit 4 du
ICU_MASK, on met 1 dans le
bit 4 de ICU_SET

extern int __icu_regs_map[];
__icu_regs_map[2] = 0b10000;
```

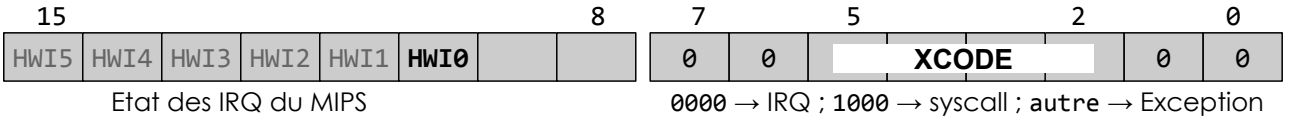


Rappel registres système : Status, Cause, EPC

Le registre `c0_sr ($12)` contient le mode d'exécution du MIPS et les autorisations d'IRQ

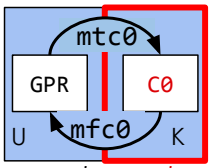


Le registre `c0_cause ($13)` contient la cause d'entrée dans le noyau (si IRQ, syscall ou except)



Le registre `c0_epc ($14)`

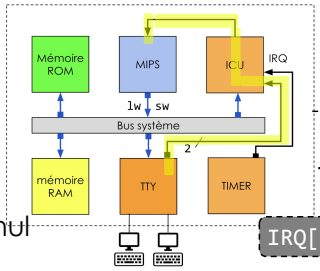
l'adresse de retour si c'est une IRQ ou sinon pour syscall et toutes les exceptions c'est l'adresse de l'instruction courante




```
mtc0 $GPR, $C0
mfc0 $GPR, $C0
```


Niveau de masquage des IRQ

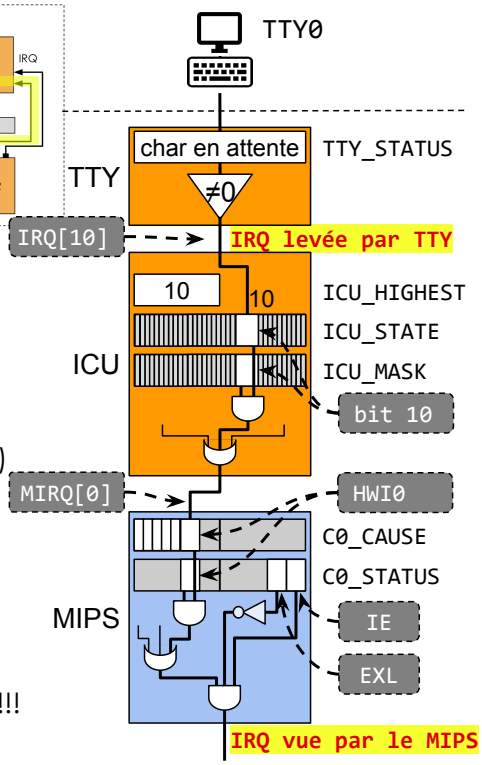
Quand une IRQ est émise par un contrôleur de périphérique, elle peut être masquée par le noyau lorsqu'il exécute du code *critique*



En TP, lors d'une frappe du clavier `TTY0` :

→ Le registre `TTY_STATUS` de `TTY0` devient non nul
→ Le contrôleur de `TTY` lève son `IRQ`
→ Le signal entre par la pin `IRQ[10]` de l'`ICU` (pour le SoC des TP)
→ Le bit `10` de `ICU_STATE` passe à `1`
→ Si c'est la seule `IRQ`, le registre `ICU_HIGHTEST` reçoit `10`
→ l'`ICU` fait un `AND` entre les bits `10` de `ICU_STATE` et `ICU_MASK`
→ puis un `OU` avec toutes les autres sorties de `AND` (ici non dessinés)
→ L'`IRQ` en sortie de l'`ICU` entre dans le MIPS (`MIRQ[0]`)
→ Le bit `HWI0` du registre de cause passe à `1`
→ Le MIPS fait un `AND` entre les bits `HWI0` de `C0_CAUSE` et `C0_STATUS`
→ puis un `OU` avec toutes les autres sorties de `AND` (ici non dessiné)
→ enfin on fait un `AND` avec les bits `IE` et `not EXL` de `C0_STATUS`.

Pour voir une `IRQ`, il faut que toutes les « vannes » soient ouvertes !!!



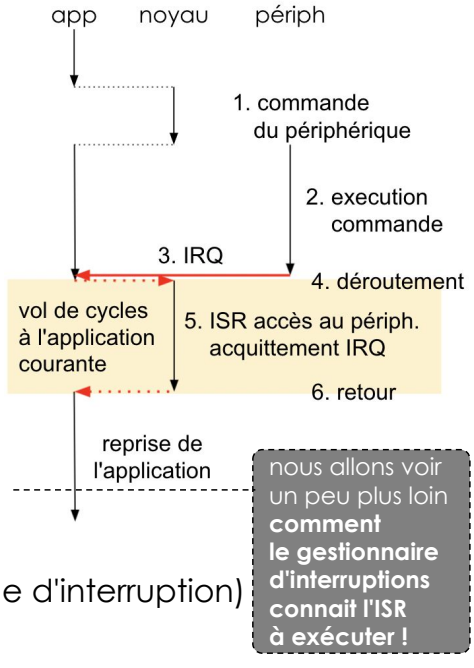
Interruptions vue du logiciel

Gestionnaire d'interruptions du noyau

Le gestionnaire d'interruption est invoqué lorsqu'**une IRQ s'active** (et elle n'est pas masquée).

Étapes de traitement :

- **déroutement** (4.) vers le noyau
  `c0_EPC` ← `PC+4` ; `c0_sr.EXL` ← `1` ; `c0_cause.XCODE` ← `0`
  `PC` ← `0x80000180`
- analyse du champ `XCODE` du registre `c0_cause`
- appel du gestionnaire d'interruption
- sauvegarde des registres temporaires
- lecture du numéro de l'IRQ dans ICU_HIGHEST
- **Exécution de l'ISR associée à ce numéro d'IRQ** (5.)
  - accès aux registres du périphérique
  - acquittement de l'IRQ (c.-à-d. baisser la ligne d'interruption)
- restauration des registres temporaires
- **retour** (6.) au programme interrompu



ISR : Interrupt Service Routine

Un pilote de périphérique contient donc :

- Des fonctions de commandes (p. ex. ici : `tty_puts()`, `tty_gets()`)
- et **une ISR pour gérer la terminaison des commandes** (p. ex: `tty_isr()`)

Les ISR (ou routines d'interruption) sont donc les fonctions qui traitent les IRQ

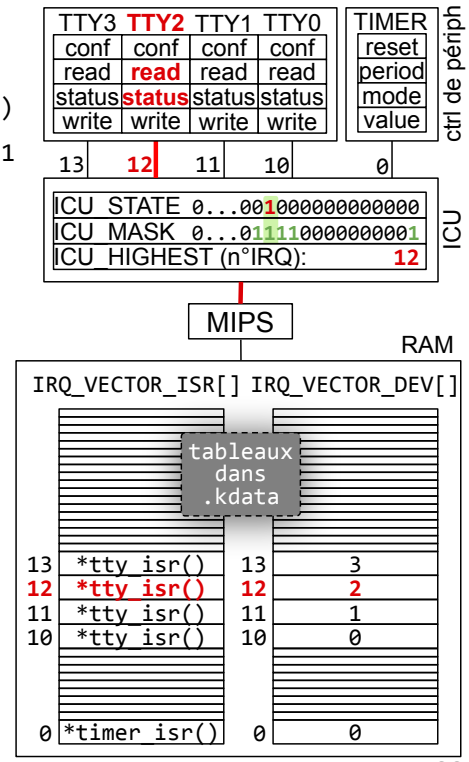
- Elles accèdent aux registres du contrôleur de périphérique ayant levé l'IRQ
  Cette étape est spécifique à chaque périphérique
- *Elles peuvent aussi programmer une nouvelle commande dans le cas où il y a une file d'attente de commandes envoyées par les applications et qui n'ont pu être démarrées parce que le périphérique était occupé.*
- *Elles peuvent demander au noyau de changer l'état de l'application qui était en attente de la terminaison de la commande*

pas dans cette version de l'OS

- Elles acquittent l'IRQ en accédant aux registres du contrôleur de périphériques. Cette étape est spécifique à chaque périphérique
- Pour l'OS des TP, les ISR ne sont pas interruptibles → c'est un choix simplificateur

Sélection et appel de la bonne ISR

- Lorsqu'une IRQ se lève et qu'elle n'est pas masquée, son numéro s'inscrit dans le registre `ICU_HIGHEST` et le MIPS est dérouté vers kentry : `PC←0x80000180` (et `c0_epc← PC+4`, `c0_sr.EXL←1` et `c0_cause.XCODE←0`)
- Par exemple, si l'IRQ vient du TTY2 alors le bit 12 de `ICU_STATE.12` reçoit `1` et puisque le bit 12 de `ICU_MASK.12` est à `1` alors `ICU_HIGHEST` prend `12`
- kentry appelle le **gestionnaire d'interruption** qui sauve les registres les registres temporaires avant d'**appeler la bonne ISR → comment fait-il ?**
- Le gestionnaire utilise un tableau indexé par le numéro d'IRQ, `IRQ_VECTOR_ISR[]`, dont les cases contiennent les pointeurs vers les `ISR`
- Il n'y a qu'une fonction `ISR()` par type de périphérique, c.-à-d. qu'il n'y a qu'une fonction `tty_isr()` utilisée quel que soit le numéro de TTY.
- Les fonctions `ISR()` ont besoin de savoir quelle instance a levé son IRQ, le gestionnaire utilise un autre tableau indexé par le numéro d'IRQ, `IRQ_VECTOR_DEV[]`, dont les cases contiennent le numéro d'instance, ce numéro est passé en argument aux fonctions `ISR()`
- Le gestionnaire d'interruption appelle donc la fonction : `IRQ_VECTOR_ISR[ICU_HIGHEST](IRQ_VECTOR_DEV[ICU_HIGHEST])`
- Dans l'exemple à droite, le gestionnaire appelle : **`tty_isr(2)`**



Configuration des IRQ

La configuration des IRQ est faite par **`arch_init()`** appelée par **`kinit()`**

1. Configuration du matériel
   - Configuration de chaque composant pouvant lever des IRQ (ici: TTY et TIMER)
   - Configuration du registre MASK de l'ICU pour choisir les IRQ que l'OS veut « voir »
   - Configuration du registre **`c0_sr`** du MIPS pour autoriser les interruptions
2. Configuration du noyau
   - Liaison (appelé **binding**) des couples (n° IRQ → ISR) et (n° IRQ → n°instance) en écrivant dans les tableaux `IRQ_VECTOR_ISR[]` et `IRQ_VECTOR_DEV[]`

     Notez que dans un OS plus avancé `IRQ_VECTOR_DEV[]` contiendrait un pointeur sur une structure de donnée propre au périphérique (structure « `device` »)

harch.c

```
void arch_init (int tick) {
    timer_init (0, tick);
    icu_set_mask (0, 0);
    irq_vector_isr [0] = timer_isr;
    irq_vector_dev [0] = 0;

    for (int tty = 1; tty < NTTYS; tty++) {
        icu_set_mask (0, 10+tty);
        irq_vector_isr [10+tty] = tty_isr;
        irq_vector_dev [10+tty] = tty;
    }
}

static void timer_init (int timer, int tick) {
    timer = timer % NCPUS;
    __timer_regs_map[timer].resetirq = 1;
    __timer_regs_map[timer].period =  tick;
    __timer_regs_map[timer].mode = (tick)?3:0;
}

static void icu_set_mask (int icu, int irq){
    icu = icu % NCPUS;
    __icu_regs_map[icu].set = 1 << irq;
}
```


Ce qu'il faut retenir

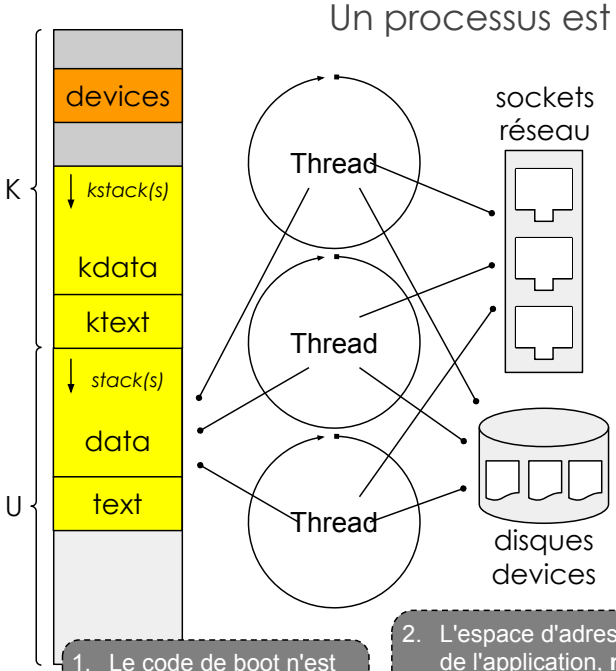
- Les IRQ sont des signaux électriques à 2 états produits par les périphériques pour prévenir d'un événement (fin de commande ou arrivée de donnée).
- Les IRQ passent par un concentrateur nommé ici ICU qui les numérote et qui peut les masquer individuellement.
- Ici, le TTY lève une IRQ à la réception d'un caractère du clavier et le timer lève une IRQ selon une période configurable.
- Quand une IRQ non masquée est levée, elle provoque le déroutement du programme en cours vers le noyau qui se rend compte que c'est une IRQ et qui exécute le gestionnaire d'interruption.
- Ce gestionnaire sait grâce à l'ICU quel est le numéro de l'RQ, et il utilise ce numéro comme index pour lire le vecteur d'interruption contenant les adresses des ISR (routine d'interruption) et un vecteur de device pour connaître l'instance.
- Une ISR commande un périphérique et acquitte l'IRQ pour baisser le signal
- Les ISR s'exécutent en "volant" des cycles aux applications.

Threads

Processus

Un processus est une programme en cours d'exécution

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



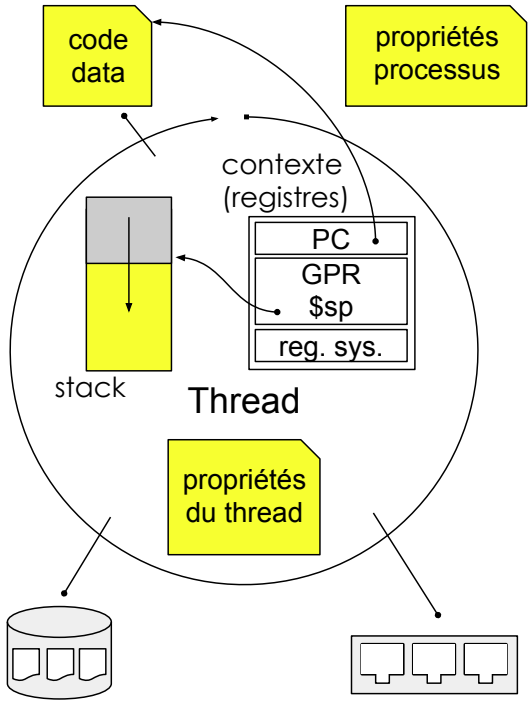
Pour l'OS, c'est un **conteneur de ressources** permettant d'exécuter un programme (application), il contient :

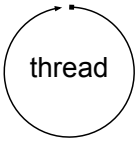
- un **espace d'adressage** pour le code du programme, les données globales et les piles d'exécution.
- des ressources d'**entrées-sorties** : fichiers (disque et devices) et sockets (réseau)
- des **propriétés** (état, parenté, droits, ...)
- et des **fils d'exécution** nommés **threads**

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





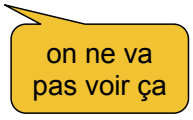
Un thread est donc un fil d'exécution du processus, c'est ce que le processeur est en train de faire. C'est donc « *vivant* » 😊
(d'où la représentation comme une roue qui tourne)

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

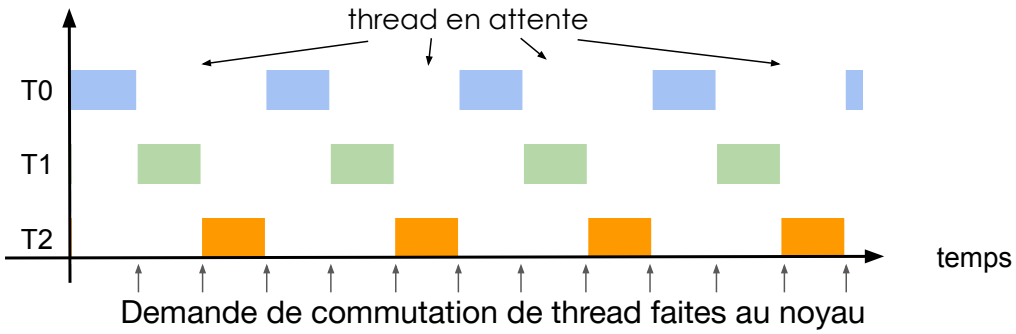
- Un processus a au moins un thread, mais il peut en avoir plusieurs.
- Tous les threads partagent le même espace d'adressage
  - le même programme et chaque thread peut utiliser les fonctions qu'il veut
  - les mêmes données globales (mais ils ne devraient pas partager les piles)
  - *les mêmes propriétés 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.*



- 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.

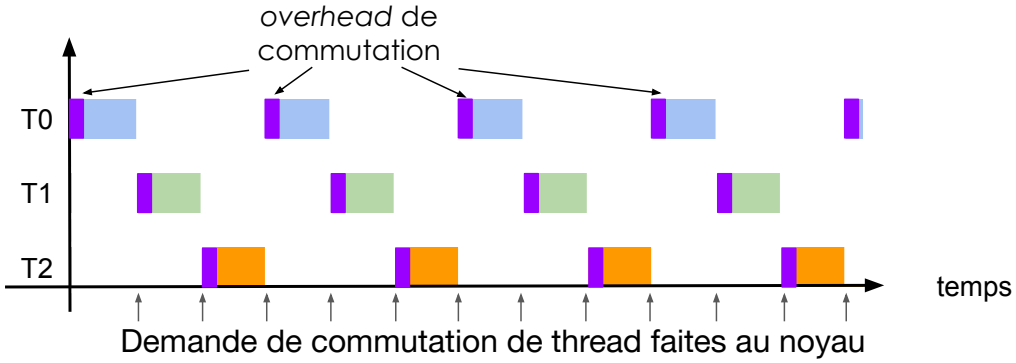
Exécution en temps partagé

- 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 trois threads (T0, T1 et T2) alors à un instant `t`, un seul des threads s'exécute, les autres sont en attente du processeur.



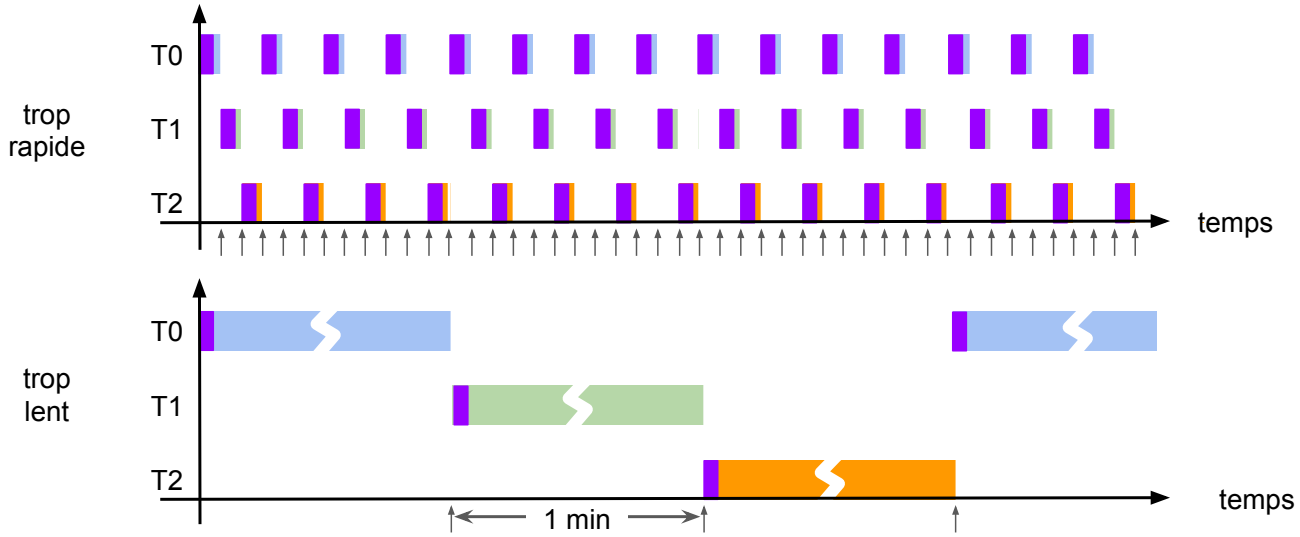
Ordonnanceur

- 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, **sauve** l'état du thread courant
  et **restaure** 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 dépend de la machine, c'est entre 10 et 100Hz



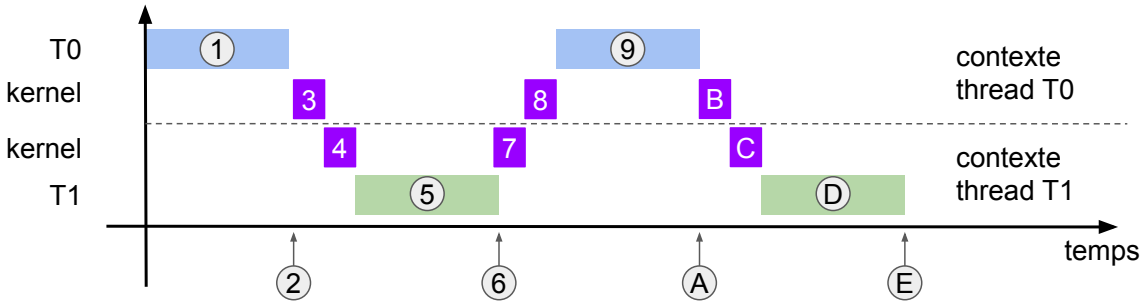
Quand faire l'ordonnancement ?

- 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** (cession). Le thread cède le processeur et un nouveau thread, choisi par le l'ordonnanceur de thread, gagne le processeur.
  - La **durée entre deux IRQ** du timer est nommée **tick**.
  - La **commutation est à chaque tick ou** chaque **quantum** (multiple de tick).
- 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 (un 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 noyau peut décider de provoquer une commutation de thread (un 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 élu par l'ordonnanceur

Comment faire la commutation ?

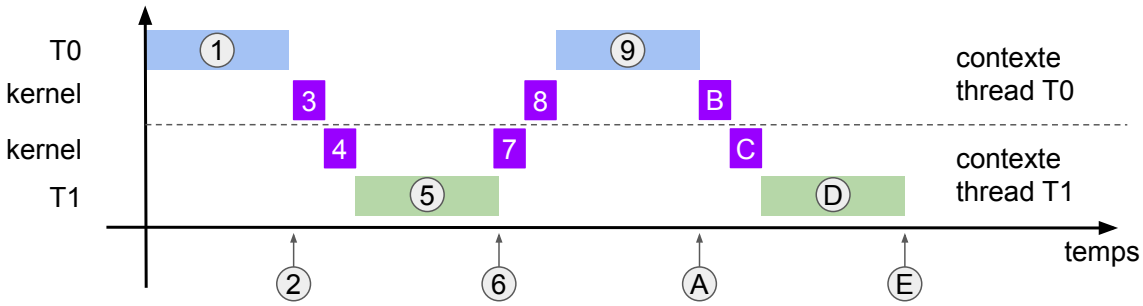
- Dans une commutation, il y a
  - le **thread sortant** qui perd le processeur,
  - le **thread entrant** qui gagne le processeur.
- Pour définir la commutation de thread, il faut :
  - déterminer le **contexte d'un thread** (l'ensemble des registres à sauver)
  - définir un **mécanisme de sauvegarde 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 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.

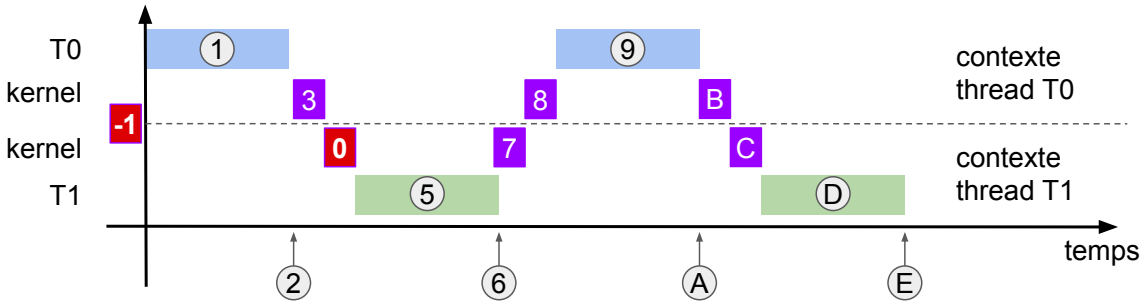
Le thread T0 qui est interrompu par l'IRQ (2) entre dans `kenty` en 3 dont il sortira en 8

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.

Contraintes

- Pour chaque thread, il faudrait créer dynamiquement deux groupes de structures
  - 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 kernel doit traiter un ensemble de threads dont il ne connaît pas le nombre au moment de sa compilation. L'ordonnanceur doit, par exemple, parcourir l'ensemble des threads. Toutefois, 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.)

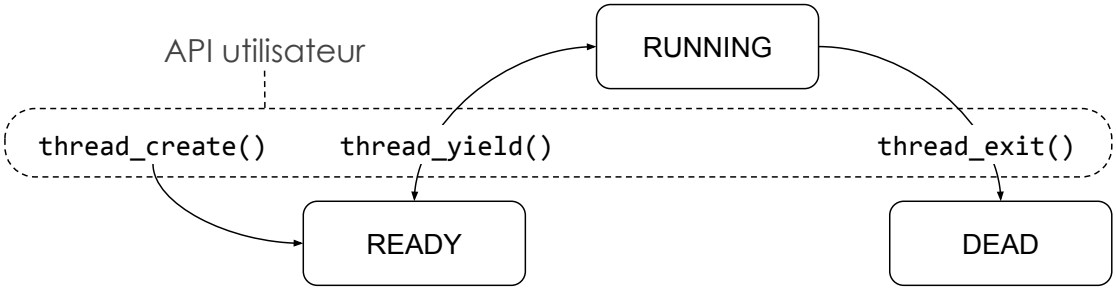
Choix (simplificateurs à cette étape de construction du noyau)

- 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 user)
- On choisit de ne pas chaîner les threads entre eux avec des pointeurs, donc le noyau a un tableau de taille fixe dans la section `.kdata` pointant sur les threads de l'application

États de thread

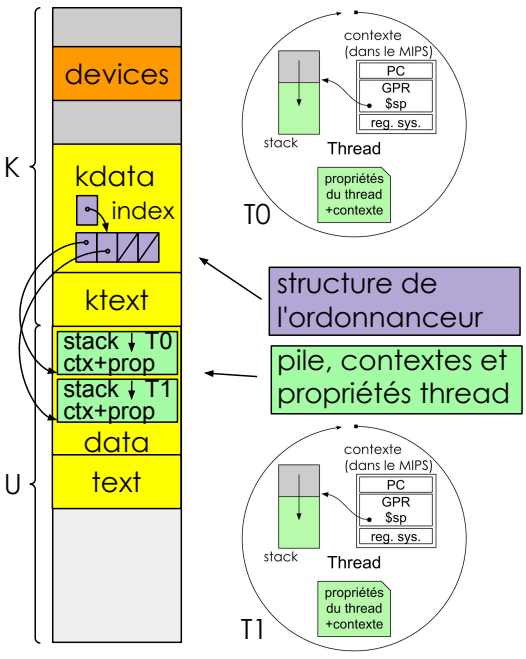
A ce niveau de construction du noyau, les threads n'ont que 3 états

- `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 qu'il retente jusqu'à réussir, c'est de la scrutation (polling)

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
    - `text` code
  - 1 contexte de registres
  - des propriétés (fonction principale, état, …)
- Si on a plusieurs threads, ils partagent les mêmes segments `text` et `data`, mais chaque thread a son segment `stack`.
- Pour chaque thread, une structure dans les données globales de l'application contient la pile, 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

API utilisateur et un exemple

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

1. Création
```
int thread_create (
            thread_t * thread,
            void *(*fun) (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.

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

thread_t t1;

void * t1_fun (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_fun, "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()`

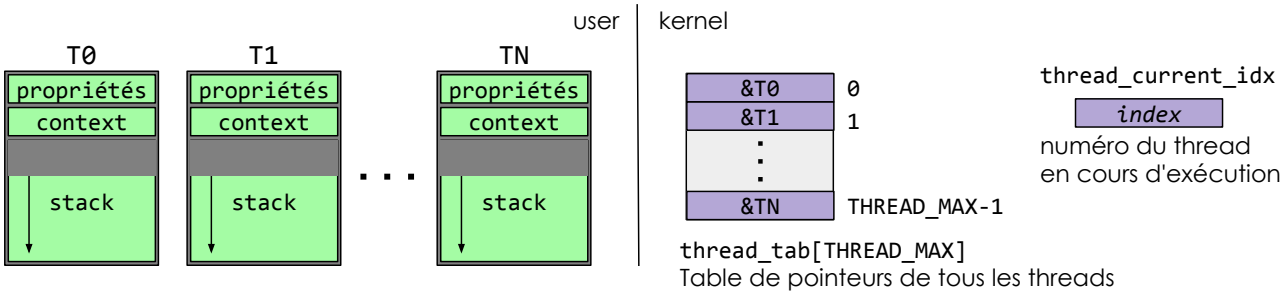
Structure `thread_t` et tableau `thread_tab`

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

```c
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 fun;        } 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 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`

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



Commutation de contexte de thread

La commutation entre deux threads se déroule en trois temps :

- l'élection d'un thread entrant selon la politique d'ordonnancement,
- la sauvegarde du contexte du thread sortant,
- la chargement du contexte du thread entrant.

L'ordonnanceur fait un parcours circulaire de la table **thread_tab** 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 = thread_current_idx;                       // n° du thread courant dans thread_tab
    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 !
        if (thread_save (thread_tab[th_curr]->context)) {   // sauve le ctx du thread sortant et rend 1
            thread_current_idx = th_next;                   // mise à jour de thread_current_idx
            thread_load (thread_tab[th_next]->context);     // chargement de contexte & sortie par jr $31
        }                                                   // donc de thread_save() mais qui rend 0
    }
    thread_tab[thread_current_idx]->state= TH_STATE_RUNNING;// the thread choisi est dans l'état
}                                                           // TH_STATE_RUNNNIG
```

La subtilité c'est que la sortie de la fonction `thread_load()` par le jr $31 habituel nous 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 (élection et chargement)

`thead_load()`

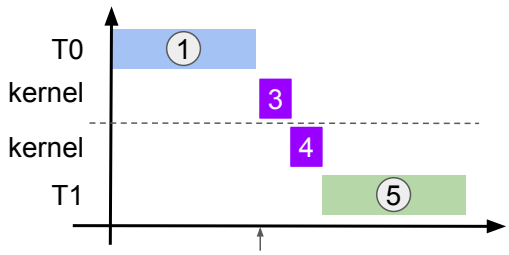
- Quand on entre dans `thead_load()`, elle charge les registres et la valeur de retour est 0
- Quand on sort de `thread_load()`, on est dans dans le thread élu dont les registres viennent d'être chargés, et donc, quand on est **en régime stationnaire**, on sort bien de `thread_load()` mais **on revient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

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

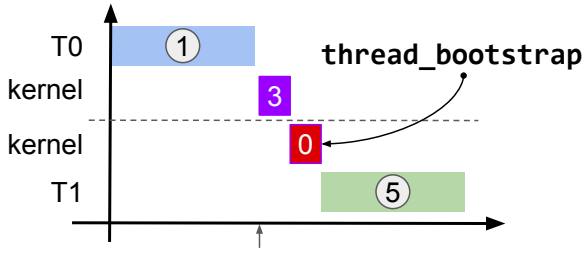
- Le chargement d'un contexte et donc l'entrée dans un thread se fait uniquement par la fonction `thread_load()`.

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()`. 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, quand on sort de `thread_load()` on entre dans `thread_bootstrap()`



Lancement d'un thread 2/3

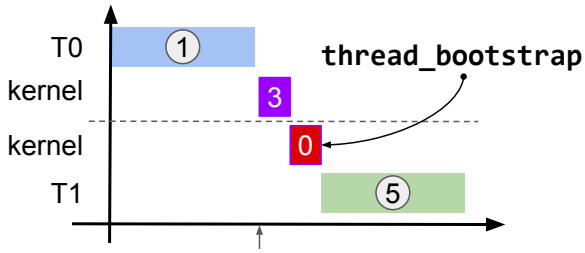
`void thread_bootstrap(void)`

- La fonction `thread_bootstrap()` est la fonction du noyau laquelle on saute à la sortie d'un `thread_load()` pour la première fois
- 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 fun;
    int arg;
    int context[TH_CONTEXT_SIZE];
    int stack[THREAD_STACK_SIZE];
} thread_t;
```

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

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



Lancement d'un thread 3/3

```
void thread_bootstrap (void)  {
    thread_t * thread = thread_tab [thread_current_idx]; // récupérer le ptr sur le thread courant
    thread->state = TH_STATE_RUNNING;                    // mettre à jour son état
    thread_launch (   thread->fun,          // $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 fun
}
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 :

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

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

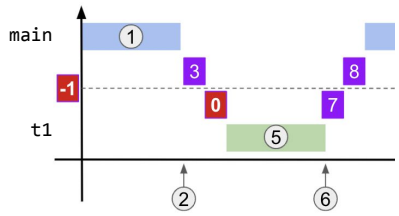
Quand on sort d'un thread alors on doit en informer le noyau. Le thread peut appeler thread_exit()

Séquence de la commutation

label0.s

```
K    12:    <boot> ------------------------------ ./kernel/hcpua.S
K    37:    <kinit> ----------------------------- ./kernel/kinit.c
K    62:    <kprintf> --------------------------- ./kernel/klibc.c
K  2184:    <arch_init> ------------------------- ./kernel/harch.c
K  2282:    <thread_create_kernel> -------------- ./kernel/kthread.c
K  2435:    <thread_load> ----------------------- ./kernel/hcpua.S
K  2541:    <thread_bootstrap> ------------------ ./kernel/kthread.c
K  2599:    <thread_launch> --------------------- ./kernel/hcpua.S
-------------------------------------------------------------------
THREAD: 0  main
-------------------------------------------------------------------
U 0  2611:    <_start> ---------------------------- ./ulib/crt0.c
U 0  15018:   <main> ------------------------------ ./uapp/main.c
U 0  15082:   <thread_create> ------------------- ./ulib/thread.c
U 0  16693:   <thread_yield> -------------------- ./ulib/thread.c
U 0  16720:   <syscall> ------------------------- ./ulib/crt0.c
K 0  16722:   <kentry> ------------------------- ./kernel/hcpua.S
K 0  16727:   <ksyscall> ----------------------- ./kernel/hcpua.S
K 0  16761:   <thread_yield> ------------------- ./kernel/kthread.c
K 0  16810:   <sched_switch> ------------------- ./kernel/kthread.c
K 0  16946:   <thread_save> -------------------- ./kernel/hcpua.S
K 0  17036:   <thread_load> -------------------- ./kernel/hcpua.S
K 0  17090:   <thread_bootstrap> --------------- ./kernel/kthread.c
K 0  17107:   <thread_launch> ------------------ ./kernel/hcpua.S
-------------------------------------------------------------------
THREAD: 1  t1
-------------------------------------------------------------------
U 1  17119:          <thread_start> ------------- ./ulib/thread.c
U 1  17145:          <t1_fun> ------------------- ./uapp/main.c
U 1  17307:          <fprintf> ------------------ ./ulib/libc.c
K 1  18272:          <kentry> ------------------- ./kernel/hcpua.S
K 1  18277:          <ksyscall> ----------------- ./kernel/hcpua.S
K 1  18299:          <tty_puts> ----------------- ./kernel/harch.c
U 1  18604:          <thread_yield> ------------- ./ulib/thread.c
U 1  18613:          <syscall> ------------------ ./ulib/crt0.c
K 1  18618:          <kentry> ------------------- ./kernel/hcpua.S
K 1  18623:          <ksyscall> ----------------- ./kernel/hcpua.S
K 1  18644:          <thread_yield> ------------- ./kernel/kthread.c
K 1  18666:          <sched_switch> ------------- ./kernel/kthread.c
K 1  18755:          <thread_save> -------------- ./kernel/hcpua.S
K 1  18814:          <thread_load> -------------- ./kernel/hcpua.S
K 1  18914:          <eret_syscall> ------------- ./kernel/hcpua.S
-------------------------------------------------------------------
THREAD: 0  main
-------------------------------------------------------------------
U 0  19019:   <fprintf> ------------------------- ./ulib/libc.c
U 0  19818:   <thread_yield> -------------------- ./ulib/thread.c
U 0  19827:   <syscall> ------------------------- ./ulib/crt0.c
K 0  19832:   <kentry> -------------------------- ./kernel/hcpua.S
```

-1 1 2 3 0 5 6 7 8 9

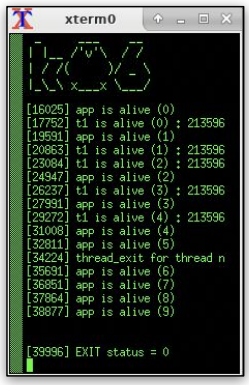



```
#include <libc.h>
#include <thread.h>

thread_t t1;

void * t1_fun (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_fun, "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

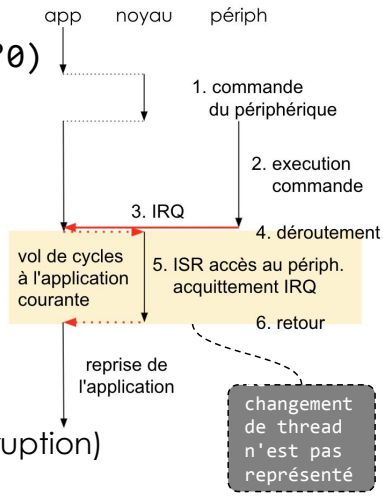
Exécution en temps partagé

- 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`)

```
struct timer_s {
    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`)



- **déroutement** (4.) 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** (5.)
  - acquittement de l'IRQ (c.-à-d. baisser la ligne d'interruption)
  - appel de `thread_yield()`

    → on sort du thread `T1` pour aller dans un autre thread, disons `T2`
    on reviendra dans le thread T1 lors d'un autre `thread_yield()` et
    on reviendra exactement dans cet état mais du temps aura passé
- restauration des registres temporaires de `T1` (quand T2 aura lui aussi été commuté)
- **retour** (6.) au programme interrompu dans `T1`

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_gets (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 un `'\n'` (enter ou linefeed) ou au plus `count` caractères. Elle appelle en boucle `int tty_getc (int tty)`
  - `int tty_getc (int tty)` doit lire le clavier et rendre le caractère lu
    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, mais dans le TME, vous verrez une version utilisant les interruptions.
  - *Cette version n'est pas correcte par le loop back est retardé par rapport à la frappe du clavier, voyez-vous pourquoi ?*

```
int tty_getc (int tty) {
    while (__tty_regs_map[tty].status [tty] == 0) {
        thread_yield(); // s'il n'y a rien alors partir
    }
    int c = __tty_regs_map[tty].read; // lecture
    __tty_regs_map[tty].write = c;    // loop back
    return c:
}
```

Ce qu'il faut retenir

- 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 ou à l'état sauvegardé) 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 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 restaurer 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

Conclusion

- Quelles sont les étapes du TME
- et la suite du module

Etapes du TME

Au prochain TME, vous allez manipuler le gestionnaire d'interruption et la gestion de threads en particulier faire des analyses temporelles des services du noyau.

Encore à venir ...

Il y a une suite…

Si cette entrée dans le fonctionnement d'une architecture de SoC et d'un système d'exploitation vous a intéressé, il y a la suite au second semestre dans l'UE `LU3IN031`.

Concernant l'architecture, vous verrez

- L'architecture interne d'un MIPS, en particulier le séquencement des instructions
- la micro-architecture des opérateurs en vue de leur performance
- L'architecture d'un SoC avec plusieurs MIPS se partageant le même espace d'adressage
- L'architecture d'un cache de premier niveau et les problèmes de cohérence en multicores
- Le fonctionnement du contrôleur de disque

Concernant le système d'exploitation, vous verrez

- Une gestion de la mémoire dynamique, nécessaire pour créer des variables et les détruire
- Une API de gestion de listes chaînées pour construire des structures de données plus complexes
- Une gestion des états d'attente de threads et des listes d'attentes sur les ressources partagées
- Une gestion plus propre des pilotes de périphériques
- Les mécanismes de communications et de synchronisation des threads en mono et multi-cores
- Un système de fichiers dans lequel pourra être ranger le noyau et l'application
- et pleins d'autres petites choses pour la programmation système....