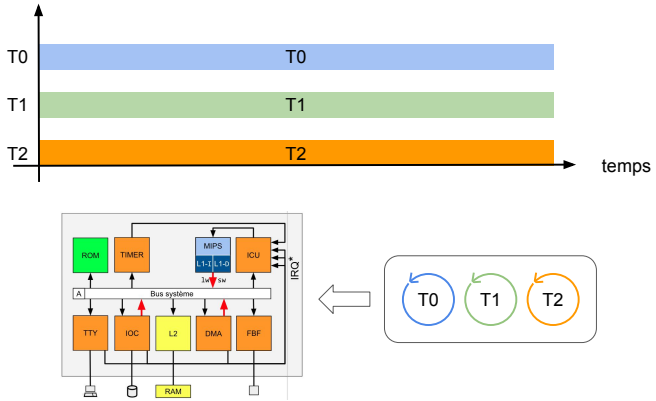
ALMO

GIET

Commutation de tâches

Question

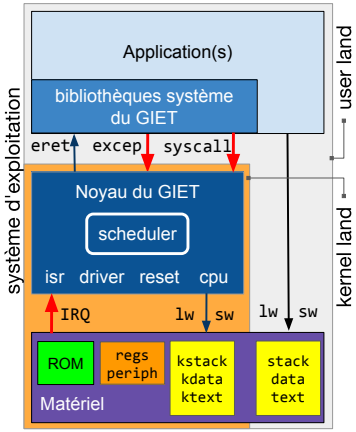
Comment un ordinateur à un seul cœur de calcul peut-il faire pour exécuter plusieurs applications (plusieurs tâches) simultanément ?



Plan

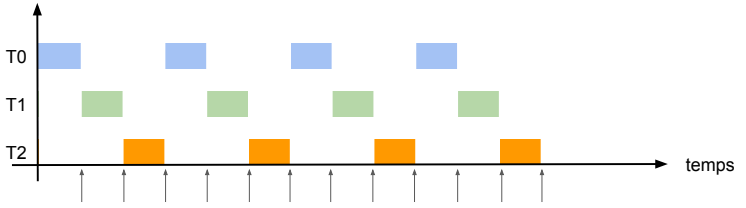
Dans le noyau, le service en charge de la commutation de tâche se nomme scheduler ou ordonnanceur

1. Principe de la commutation de tâche
2. Séquence des opérations de commutation pour le GIET
3. Analyse de code



Principe de solution

Il suffit d'exécuter les applications à tour de rôle à une fréquence suffisamment élevée pour que l'utilisateur ait l'impression que toutes les applications s'exécutent en même temps.

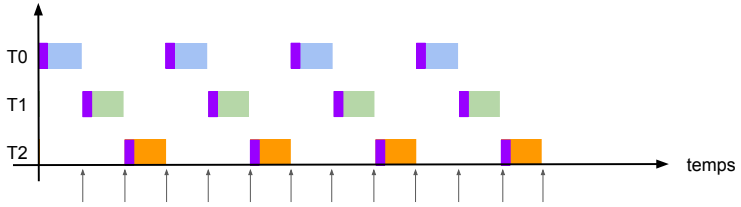


Demande de commutation de tâches faites au noyau

On nomme ce mode de fonctionnement : exécution en temps partagé.

Overhead

Ce n'est pas gratuit : il y a un "overhead", plus exactement overhead cost qui signifie "frais généraux". Dans un OS, c'est le coût d'un service du noyau.
Ici, c'est la durée d'une commutation de tâche, c'est du temps perdu car pendant ce temps le processeur n'exécute pas les tâches.

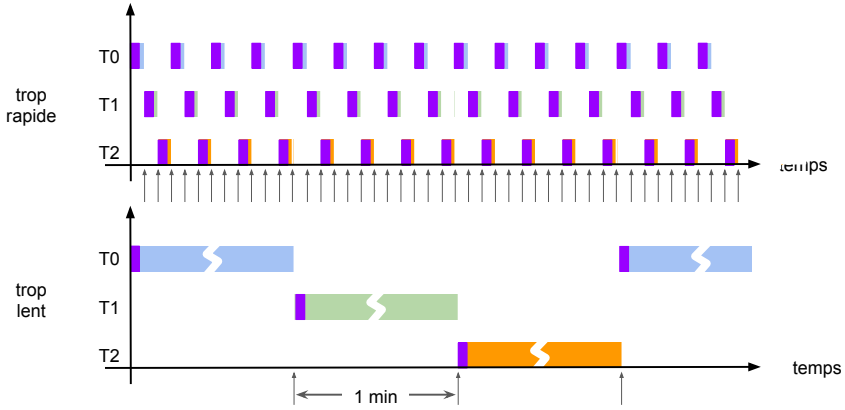


Demande de commutation de tâches faites au noyau

Sur le schéma, en violet, c'est le temps nécessaire à la commutation de tâches.

Quelle est la fréquence de commutation ?

La fréquence de commutation doit être assez élevée pour avoir l'illusion du parallélisme, mais pas trop à cause de l'overhead, le coût de la commutation.
La bonne fréquence dépend de la durée de l'overhead et de la fréquence du processeur, l'ordre de grandeur est entre 10 et 100 Hz.



Quand faire le changement de tâche ?

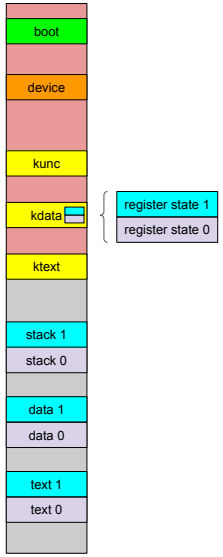
- Commutation périodique
  - Une IRQ périodique du TIMER interrompt la tâche en cours.
    L'opération de commutation de tâche sera réalisée par l'ISR du TIMER.
    La durée entre deux IRQ du timer est nommée **tick**
    On peut faire une commutation à chaque tick ou multiple de tick.
- Commutation à l'initiative de la tâche
  - La tâche en cours peut demander elle-même la commutation de tâche, cette opération est nommée **yield**. Elle perd le processeur et une nouvelle tâche choisie par le noyau gagne le processeur.
  - Quand une tâche demande un service au noyau, mais que ce service ne peut être rendu immédiatement car dépendant d'une ressource (p. ex. un périphérique ou un verrou logiciel), alors le noyau peut décider de provoquer une commutation de tâche.
    La tâche perd le processeur parce qu'elle ne peut plus avancer, lorsque la ressource attendue arrivera, elle pourra reprendre son travail.
- Avec le GIET, nous ne voyons que le commutation périodique.

Comment faire la commutation de tâche ?

- Dans une commutation, il y a
  - la **tâche sortante** qui est la tâche qui perd le processeur,
  - la **tâche entrante** qui est la tâche qui gagne le processeur.
- Pour définir la commutation de tâche, il faut :
  - déterminer le **contexte** d'une tâche (son état d'exécution)
  - définir **un mécanisme** de sauvegarde et de restauration de ces contextes,
  - définir **une politique** d'ordonnancement des tâches,
    c'est-à-dire définir l'ordre dans lequel les tâches doivent s'exécuter.
- La commutation entre deux tâches se déroule en trois parties, il faut faire :
  - **l'élection** une tâche entrante selon la politique d'ordonnancement,
  - **la sauvegarde** du contexte de la tâche sortante,
  - **la restauration** du contexte de la tâche entrante.

Contexte d'une tâche

- Contexte d'une tâche
  Pour s'exécuter une tâche a besoin de :
  - 3 segments d'adresses en mémoire :
    - stack
    - data
    - text
  - 1 état des registres du coeur
- Si on a plusieurs tâches, elles peuvent partager les mêmes segments d'adresses text et data, en revanche chaque tâche a un segment stack propre.
- Ici, on suppose que les segments d'adresses restent en mémoire. Sur ce dessin, l'espace d'adressage est partagé par toutes les tâches, il n'y a donc pas de sécurité entres elles.

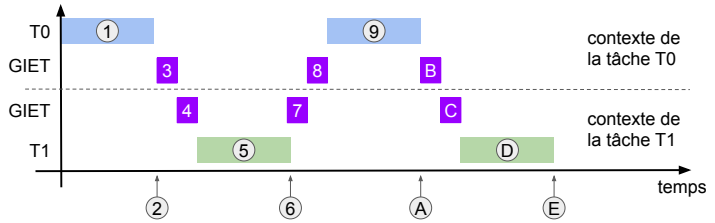


Contexte à sauver d'une tâche

- Il faut sauver les registres du cœur puisque le cœur va devoir être donné puis repris aux tâches périodiquement.
- Les registres du contexte pour le MIPS32 :
  - 32 registres GPR — sauf $0, $26, $27
    remarquez que $29 contient le pointeur dans stack
  - HI / LO — sorties des multiplieurs et diviseurs entiers
  - EPC — adresse de retour dans la tâche
  - SR — état du cœur, la commutation peut survenir alors que le cœur est en mode USER ou KERNEL

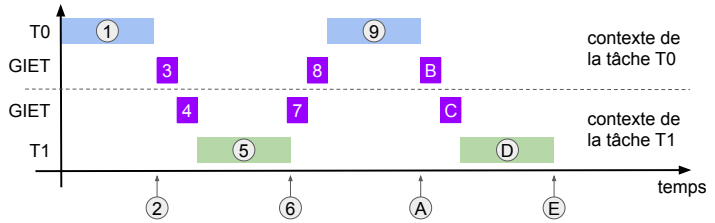
  36 registres mais pour la sauvegarde, on utilise un tableau aligné sur 64 (2^6)
- Il est inutile de sauver les segments d'adresses, puisque les segments restent en mémoire.

Principe de la commutation de tâche



- Régime stationnaire
  1. La tâche T0 s'exécute
  2. Le cœur reçoit une IRQ du timer qui interrompt la tâche T0
  3. Le noyau analyse la cause et exécute l'ISR du TIMER :
     Entrée dans la fonction de commutation de tâche
     a. élection d'une tâche entrante
     b. sauvegarde des registres de la tâche sortante, restauration des registres de la tâche entrante
  4. Sortie de la fonction de commutation de tâche, sortie de l'ISR, sortie du GIET
  5. La tâche T1 s'exécute…
- On continue ainsi : 6 = 2, 7 = 3, 8 = 4, 9 = 1, etc……

Cas particulier du démarrage 1/2



Dans ce chronogramme, on est en régime stationnaire.

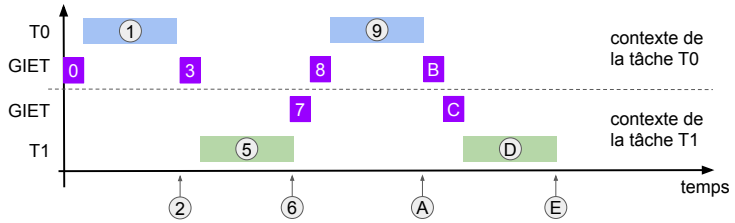
La tâche T0 qui est interrompue par l'IRQ ② entre dans le GIET en 3 dont il sortira en 8

Ça veut dire que la tâche T1 qui est élue puis restauré avait déjà été élue 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 GIET pour sortir d'une ISR.

Si T1 n'avait jamais été élu dans le passé, si c'était la première fois,
il aurait fallu **agir différemment, car on ne peut pas revenir dans une ISR !**

Cas particulier du démarrage 2/2



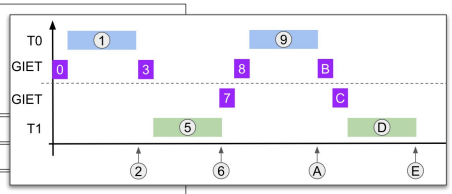
Dans ce chronogramme, on a le cas du démarrage de T0 et le démarrage T1

La tâche T0 est lancée par un lanceur d'application 0
ici c'est dans le reset par un simple : `jal main0  # si main0 est le main() de T0`

La tâche T0 qui est interrompue par l'IRQ ② entre dans le GIET en 3
mais on sort en commença directement dans T1, **il n'y a pas eu** 4

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

Séquence détaillée de la commutation de tâche 1/2



0 • **reset** [cf. TME]
  ○ configuration ICU
  ○ configuration TIMER
  ○ appel **T0** (Jump simple sans retour)

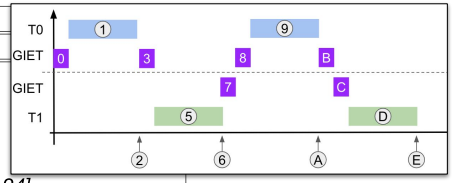
1 • **T0**

2 → IRQ TIMER reçu par T0

3 • **giet** *[p. 2]*
  ○ analyse du cause register CR
  ○ appel de **_int_handler**() *[p. 3]*
    ■ sauvegarde registres temporaires de T0
    ■ appel de **_int_demux**() de T0 *[p. 24]*
      • lecture de ICU_IT_VECTOR
      • appel de **_isr_switch**() de T0 *[p. 26]*
        ○ acquittement TIMER
        ○ appel de **_ctx_switch**() de T0 *[p. 30]*
          ■ élection de T1
          ■ appel de **_task_switch**(T0, T1) *[p. 5]*
            • sauvegarde contexte de T0
            • restauration contexte de T1
              eret saute dans T1

5 • **T1**

Séquence détaillée de la commutation de tâche 2/2



5 • **T1**

6 → IRQ TIMER

7 • **giet** *[p. 2]*
  ○ analyse du cause register CR
  ○ appel de **_int_handler**() *[p. 3]*
    ■ sauvegarde reg. temp. de T1
    ■ appel de **_int_demux**() de T1*[p. 24]*
      • lecture de ICU_IT_VECTOR
      • appel de **_isr_switch**() de T1 *[p. 26]*
        ○ acquittement TIMER
        ○ appel de **_ctx_switch**() de T1 *[p. 30]*
          ■ élection de T0
          ■ appel de **_task_switch**(T1, T0) *[p. 5]*
            • sauvegarde contexte de T1
            • restauration contexte de T0

8
        ○ sortie de _ctx_switch() de T0
      • sortie de _isr_switch() de T0
    ■ restauration registres temporaires de T0
    ■ eret retour dans T0

9 • **T0**

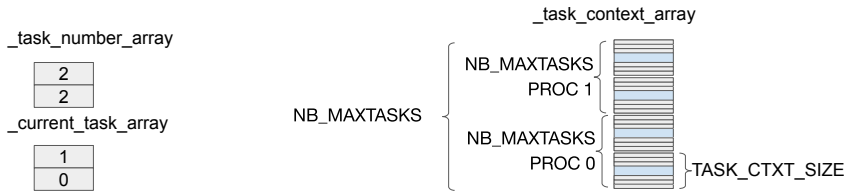
Variables globales du scheduler

Le système contient plusieurs variables globales pour la gestion des contextes

- NB_PROCS : nombre maximal de cœur dans la machine
- NB_MAXTASKS : nombre maximal de tâches par cœur
- TASK_CTXT_SIZE : nombre de registres d'un contexte (aligné sur une puis. 2)
- int _task_number_array[NB_PROCS] : tableau contenant le nombre de tâche pour un cœur
- int _current_task_array[NB_PROCS] : tableau contenant le numéro de la tâche en cours
- int _task_context_array[NB_PROCS * NB_MAXTASKS * TASK_CTXT_SIZE] : table des contextes

Par exemple :

NB_PROCS	: 2
NB_MAXTASKS	: 2
TASK_CTXT_SIZE	: 64



Etat initial du contexte

Quand une tâche n'a jamais été exécuté, il faut quand même initialiser son contexte avec un contexte initial parce que ce contexte sera utilisé par la fonction _task_switch().

Quels sont les registres à définir :

- le pointeur de pile $29 : qui doit pointer en haut de la pile moins la place pour les arguments de main().
- le registre SR : avec 0xFF13 (UM=1, IE=1 et EXL=1) pour qu'au moment du eret on passe en mode USER, interruptions autorisées
- le registre EPC : avec l'adresse de la fonction main()
- les registres $4 à $7 : avec les arguments de la fonction main attention, il n'y en a pas dans le cas du GIET
- le registre $31 : qui contient l'adresse de retour de la fonction _task_switch() qui en régime stationnaire retourne dans _ctx_switch() mais qui au départ va vers une instruction eret pour sauter directement dans la tâche

Analyse de code

Entrée dans le noyau

P. 2

```
.section .giet , "ax", @progbits
.space 0x180

_giet:
     mfc0  $27 , $13            /* $27 <= Cause register */
     la    $26 , _cause_vector  /* $26 <= _cause_vector */
     andi  $27 , $27 , 0x3c     /* $27 <= XCODE *4 */
     addu  $26 , $26 , $27      /* $26 <= & _cause_vector [ XCODE ] */
     lw    $26 , ($26)          /* $26 <= _cause_vector [ XCODE ] */
     jr    $26                  /* Jump indexed by XCODE */
```

P. 21

```
const _exc_func_t _cause_vector[16] = {
     &_int_handler , /* 0000 : external interrupt */
     &_cause_ukn , /* 0001 : undefined exception */
     &_cause_ukn , /* 0010 : undefined exception */
     &_cause_ukn , /* 0011 : undefined exception */
     &_cause_adel , /* 0100 : illegal address read exception */
     &_cause_ades , /* 0101 : illegal address write exception */
     &_cause_ibe , /* 0110 : instruction bus error exception */
     &_cause_dbe , /* 0111 : data bus error exception */
     &_sys_handler , /* 1000 : system call */
     &_cause_bp , /* 1001 : breakpoint exception */
     &_cause_ri , /* 1010 : illegal codop exception */
     &_cause_cpu , /* 1011 : illegal coprocessor access */
     &_cause_ovf , /* 1100 : arithmetic overflow exception */
     &_cause_ukn , /* 1101 : undefined exception */
     &_cause_ukn , /* 1110 : undefined exception */
     &_cause_ukn , /* 1111 : undefined exception */ };
```


_int_handler : Gestionnaire d'IRQ - début

```
_int_handler:
    addiu   $29,    $29,    -25*4   /* stack space reservation (19 registers to
                                       save and 4 free words to call function) */
    .set noat
    sw      $1,     4*4($29)        /* save $1 */
    .set at
    sw      $2,     5*4($29)        /* save $2 */
    sw      $3,     6*4($29)        /* save $3 */
    sw      $4,     7*4($29)        /* save $4 */
    sw      $5,     8*4($29)        /* save $5 */
    sw      $6,     9*4($29)        /* save $6 */
    sw      $7,     10*4($29)       /* save $7 */
    sw      $8,     11*4($29)       /* save $8 */
    sw      $9,     12*4($29)       /* save $9 */
    sw      $10,    13*4($29)       /* save $10 */
    sw      $11,    14*4($29)       /* save $11 */
    sw      $12,    15*4($29)       /* save $12 */
    sw      $13,    16*4($29)       /* save $13 */
    sw      $14,    17*4($29)       /* save $14 */
    sw      $15.    18*4($29)       /* save $15 */
    sw      $24,    19*4($29)       /* save $24 */
    sw      $25,    20*4($29)       /* save $25 */
    sw      $31,    21*4($29)       /* save $31 */
    mflo    $26
    sw      $26,    22*4($29)       /* save LO */
    mfhi    $26
    sw      $26,    23*4($29)       /* save HI */
    mfc0    $27,    $14
    sw      $27,    24*4($29)       /* save EPC */

    la      $26,    _int_demux
    jalr    $26                     /* jump to a C function to find the proper ISR
```

_int_demux : choix et exécution de l'ISR

_int_demux() va lire le registre _ICU_IT_VECTOR et l'utiliser comme index d'un tableau de pointeur de fonction, c'est le vecteur d'interruption.

Le tableau _interrupt_vector est initialisé dans le code du reset

```
void _int_demux(void)
{
    int interrupt_index;
    _isr_func_t isr;
    /* retrieves the highest priority active interrupt index */
    if (!_icu_read(ICU_IT_VECTOR , (unsigned int *) &interrupt_index))
    {
        /* no interrupt is active */
        if (interrupt_index > 31)
        return;
        /* call the ISR corresponding to this index */
        isr = _interrupt_vector[interrupt_index];
        isr();
    }
}
```


_ctx_switch : commutation de tâche

```
void _ctx_switch()
{
    unsigned char curr_task_index;
    unsigned char next_task_index;

    unsigned int * curr_task_context;
    unsigned int * next_task_context;

    unsigned int proc_id;

    proc_id = _procid();

    /* first, test if there is more than one task to schedule on the processor.
     * otherwise, let's just return. */
    if (_task_number_array[proc_id] <= 1)
        return;

    /* find the task context of the currently running task */
    curr_task_index = _current_task_array[proc_id];
    curr_task_context = &_task_context_array[(proc_id * NB_MAXTASKS + curr_task_index)
        * TASK_CTXT_SIZE];

    /* find the task context of the next running task (using a round-robin
     * policy) */
    next_task_index = (curr_task_index + 1) % _task_number_array[proc_id];
    next_task_context = &_task_context_array[(proc_id * NB_MAXTASKS + next_task_index)
        * TASK_CTXT_SIZE];

    /* before doing the task switch, update the _current_task_array with the
     * new task index */
    _current_task_array[proc_id] = next_task_index;

    /* now, let's do the task switch */
    _task_switch(curr_task_context, next_task_context);
}
```


_isr_switch : demande de commutation

```
void _isr_switch()
{
    volatile unsigned int *timer_address;
    unsigned int proc_id;

    proc_id = _procid();
    timer_address = (unsigned int *) &seg_timer_base + (proc_id * TIMER_SPAN);

    timer_address[TIMER_RESETIRQ] = 0; /* reset IRQ */
    _ctx_switch();
}
```


_task_switch : commutation de tâche

```
_task_switch:

    /* save _current task context */

    add     $27,    $4,     $0  /* $27 <= @_task_context_array[current_task_index] */

    mfc0    $26,    $12         /* $26 <= SR */
    sw      $26,    0*4($27)    /* ctx[0] <= SR */
    .set noat
    sw      $1,     1*4($27)    /* ctx[1] <= $1 */
    .set at
    sw      $2,     2*4($27)    /* ctx[2] <= $2 */
    sw      $3,     3*4($27)    /* ctx[3] <= $3 */
    sw      $4,     4*4($27)    /* ctx[4] <= $4 */
    sw      $5,     5*4($27)    /* ctx[5] <= $5 */
    sw      $6,     6*4($27)    /* ctx[6] <= $6 */
    sw      $7,     7*4($27)    /* ctx[7] <= $7 */
    sw      $8,     8*4($27)    /* ctx[8] <= $8 */
    sw      $9,     9*4($27)    /* ctx[9] <= $9 */
    sw      $10,    10*4($27)   /* ctx[10] <= $10 */
    sw      $11,    11*4($27)   /* ctx[11] <= $11 */
    sw      $12,    12*4($27)   /* ctx[12] <= $12 */
    sw      $13,    13*4($27)   /* ctx[13] <= $13 */
    sw      $14,    14*4($27)   /* ctx[14] <= $14 */
    sw      $15,    15*4($27)   /* ctx[15] <= $15 */
    sw      $16,    16*4($27)   /* ctx[16] <= $16 */
    sw      $17,    17*4($27)   /* ctx[17] <= $17 */
    sw      $18,    18*4($27)   /* ctx[18] <= $18 */
    sw      $19,    19*4($27)   /* ctx[19] <= $19 */
    sw      $20,    20*4($27)   /* ctx[20] <= $20 */
    sw      $21,    21*4($27)   /* ctx[21] <= $21 */
    sw      $22,    22*4($27)   /* ctx[22] <= $22 */
    sw      $23,    23*4($27)   /* ctx[23] <= $23 */
    sw      $24,    24*4($27)   /* ctx[24] <= $24 */
    sw      $25,    25*4($27)   /* ctx[25] <= $25 */
    sw      $28,    28*4($27)   /* ctx[28] <= $28 */
    sw      $29,    29*4($27)   /* ctx[29] <= $29 */
    sw      $30,    30*4($27)   /* ctx[30] <= $30 */
    sw      $31,    31*4($27)   /* ctx[31] <= $31 */
    mfc0    $26,    $14
    sw      $26,    32*4($27)   /* ctx[32] <= EPC */
    mfc0    $26,    $13
    sw      $26,    33*4($27)   /* ctx[33] <= CR */
```

```
    /* restore next task context */

    add     $27,    $5,     $0  /* $27<= @_task_context_array[next_task_index]
    lw      $26,    0*4($27)
    mtc0    $26,    $12         /* restore SR */
    .set noat
    lw      $1,     1*4($27)    /* restore $1 */
    .set at
    lw      $2,     2*4($27)    /* restore $2 */
    lw      $3,     3*4($27)    /* restore $3 */
    lw      $4,     4*4($27)    /* restore $4 */
    lw      $5,     5*4($27)    /* restore $5 */
    lw      $6,     6*4($27)    /* restore $6 */
    lw      $7,     7*4($27)    /* restore $7 */
    lw      $8,     8*4($27)    /* restore $8 */
    lw      $9,     9*4($27)    /* restore $9 */
    lw      $10,    10*4($27)   /* restore $10 */
    lw      $11,    11*4($27)   /* restore $11 */
    lw      $12,    12*4($27)   /* restore $12 */
    lw      $13,    13*4($27)   /* restore $13 */
    lw      $14,    14*4($27)   /* restore $14 */
    lw      $15,    15*4($27)   /* restore $15 */
    lw      $16,    16*4($27)   /* restore $16 */
    lw      $17,    17*4($27)   /* restore $17 */
    lw      $18,    18*4($27)   /* restore $18 */
    lw      $19,    19*4($27)   /* restore $19 */
    lw      $20,    20*4($27)   /* restore $20 */
    lw      $21,    21*4($27)   /* restore $21 */
    lw      $22,    22*4($27)   /* restore $22 */
    lw      $23,    23*4($27)   /* restore $23 */
    lw      $24,    24*4($27)   /* restore $24 */
    lw      $25,    25*4($27)   /* restore $25 */
    lw      $28,    28*4($27)   /* restore $28 */
    lw      $29,    29*4($27)   /* restore $29 */
    lw      $30,    30*4($27)   /* restore $30 */
    lw      $31,    31*4($27)   /* restore $31 */
    lw      $26,    32*4($27)
    mtc0    $26,    $14         /* restore EPC */
    lw      $26,    33*4($27)
    mtc0    $26,    $13         /* restore CR */

    jr      $31                 /* returns to caller */
```

_int_handler : Gestionnaire d'IRQ - fin

```
restore:
    .set noat
    lw      $1,     4*4($29)        /* restore $1 */
    .set at
    lw      $2,     4*5($29)        /* restore $2 */
    lw      $3,     4*6($29)        /* restore $3 */
    lw      $4,     4*7($29)        /* restore $4 */
    lw      $5,     4*8($29)        /* restore $5 */
    lw      $6,     4*9($29)        /* restore $6 */
    lw      $7,     4*10($29)       /* restore $7 */
    lw      $8,     4*11($29)       /* restore $8 */
    lw      $9,     4*12($29)       /* restore $9 */
    lw      $10,    4*13($29)       /* restore $10 */
    lw      $11,    4*14($29)       /* restore $11 */
    lw      $12,    4*15($29)       /* restore $12 */
    lw      $13,    4*16($29)       /* restore $13 */
    lw      $14,    4*17($29)       /* restore $14 */
    lw      $15,    4*18($29)       /* restore $15 */
    lw      $24,    4*19($29)       /* restore $24 */
    lw      $25,    4*20($29)       /* restore $25 */
    lw      $31,    4*21($29)       /* restore $31 */
    lw      $26,    4*22($29)
    mtlo    $26                     /* restore LO */
    lw      $26,    4*23($29)
    mthi    $26                     /* restore HI */
    lw      $27,    4*24($29)       /* return address (EPC) */
    addiu   $29,    $29,    25*4    /* restore stack pointer */
    mtc0    $27,    $14             /* restore EPC */
    eret                            /* exit GIET */
```


Conclusion

Le GIET permet d'exécuter plusieurs tâches par cœur.
C'est un système statique car il n'est pas possible

- le nombre de tâche par cœur est connu à l'avance
- il n'est pas possible de créer / détruire des tâches dynamiquement
- les tâches n'ont pas d'état (elles sont toujours prêtes)
- l'ordonnancement est fixe (à tour de rôle : round robin)

Dans un système dynamique

- le nombre de tâche par cœur n'est pas connu à l'avance
- il est possible de créer / détruire des tâches dynamiquement
- les tâches ont un état (elles ne sont pas toujours prêtes, elles peuvent être en attente d'un périphérique ou autre)
- l'ordonnancement dépend de leur état. Elles ne sont élues si elles sont en attente d'un périphériques, elles peuvent avoir une priorité, etc.

En résumé, nous avons vu :

- le principe d'exécution en temps partagé
- l'impact de l'overhead sur la fréquence de commutation de tâche
- le principe d'une commutation de tâche
- la définition du contexte d'une tâche
- le détail du mécanisme de commutation de tâche
- le problème de la première commutation pour une tâche
- le fonctionnement de l'ordonnanceur du GIET
- l'analyse du code du GIET pour la commutation

Prochaine séance

Comprendre les conséquences sur le matériel et sur le logiciel de la présence de plusieurs cœurs, puis étudier l'impact sur les performances.