ALMO

GIET
Gestionnaire Interruptions Exceptions Trap

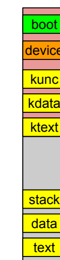
Plan

- Architecture de la plateforme
  - TTY : terminaux texte
  - TIMER : compteur de temps
  - ICU : contrôleur d'interruptions
- Interfaces du noyau
  - avec les applications utilisateur
  - avec les contrôleurs de périphériques
- Fonctionnement des services
  - Appels système (trap ou syscall)
  - Interruptions
  - *Exceptions*

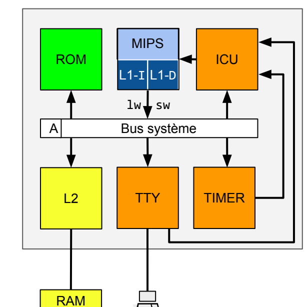
Architecture de la plateforme

Architecture de la plateforme

- Cette plateforme contient un seul coeur MIPS.
- Ce MIPS peut accéder à tous ces périphériques en passant par le bus système.
- Tous les périphériques sont des cibles, c.-à-d. qu'ils répondent seulement à des commandes de lecture ou d'écriture du cœur MIPS.
  - ROM : mémoire avec le code de boot
  - L2 : cache de niveau 2
  - TTY : gestionnaire de terminaux
  - TIMER : compteur de temps
  - ICU : concentrateur d'IRQ
- Les périphériques sont contrôlés par des registres mappés dans l'espace d'adressage.
- Les périphériques peuvent lever des signaux d'interruption pour informer d'un évènement.



L'espace d'adressage permet d'accéder à la mémoire pour le code et les données du programme et du noyau du système d'exploitation mais aussi aux registres des périphériques.

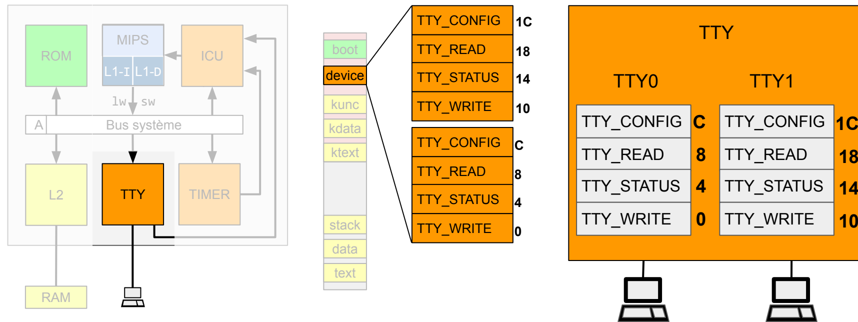


Architecture de la plateforme / TTY

http://www.soclib.fr/trac/dev/wiki/Component/VciMultiTty

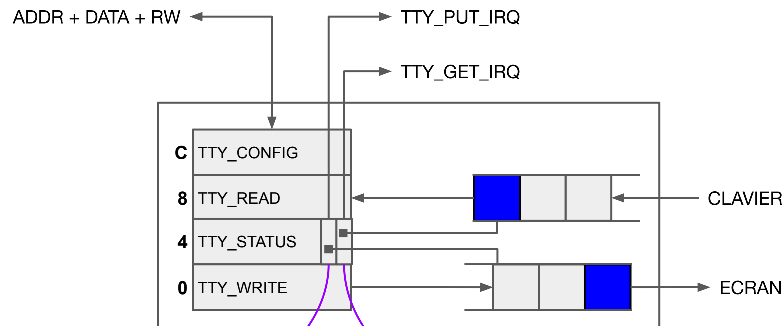
Le contrôleur de terminaux texte (TTY) permet de gérer un ou plusieurs écran-clavier.
C'est un périphérique cible contrôlé par des accès en lecture / écriture dans ses registres.

- TTY_WRITE (écriture seule) port de sortie vers le terminal
- TTY_STATUS (lecture seule) informe qu'un caractère est à lire ou qu'on peut envoyer un caractère
- TTY_READ (lecture seule) port d'entrée depuis le clavier
- TTY_CONFIG (écriture seule) configuration du protocole d'envoi (non utilisé)



Architecture de la plateforme / TTY

http://www.soclib.fr/trac/dev/wiki/Component/VciMultiTty



à 1 s'il y a une place libre dans la FIFO de sortie

TTY_PUT_IRQ est actif si TTY_STATUS [1] == 1

Pour acquitter TTY_PUT_IRQ il faut écrire dans TTY_WRITE

à 1 s'il y a un caractère présent dans la FIFO d'entrée

TTY_GET_IRQ est actif si TTY_STATUS [0] == 1

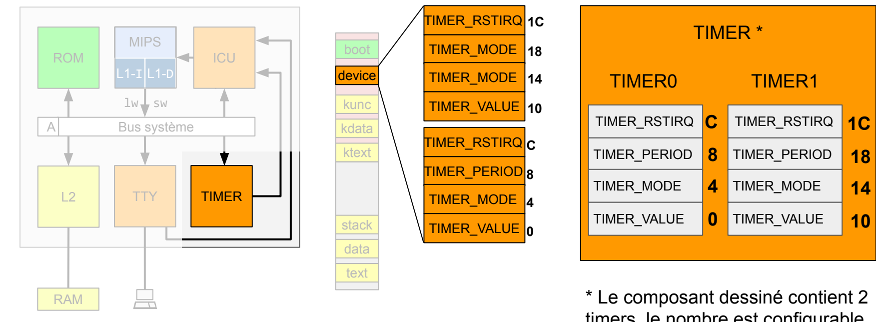
Pour acquitter TTY_GET_IRQ il faut lire dans TTY_READ

Architecture de la plateforme / TIMER

http://www.soclib.fr/trac/dev/wiki/Component/VciMultiTimer

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.

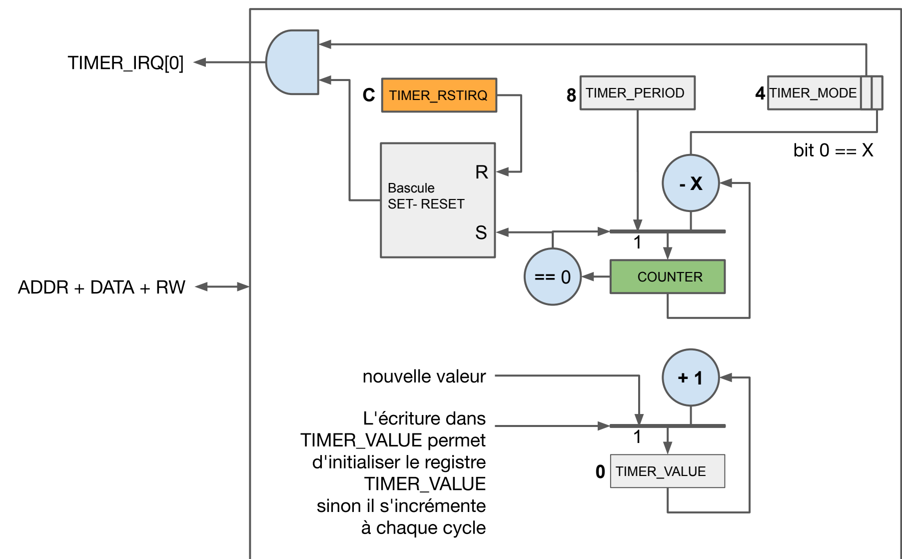
- TIMER_VALUE (lecture/écriture) +1 à chaque cycle
- TIMER_MODE (écriture seule) configure le mode de fonctionnement
  - Bit 0: 1 = timer en marche (décompte) ; 0 = timer arrêté
  - Bit 1: 0 = pas d'IRQ quand le compteur atteint 0
- TIMER_PERIOD (écriture seule) période entre 2 IRQ
- TIMER_RESETIRQ (écriture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



* Le composant dessiné contient 2 timers, le nombre est configurable.

Architecture de la plateforme / TIMER

http://www.soclib.fr/trac/dev/wiki/Component/VciMultiTimer

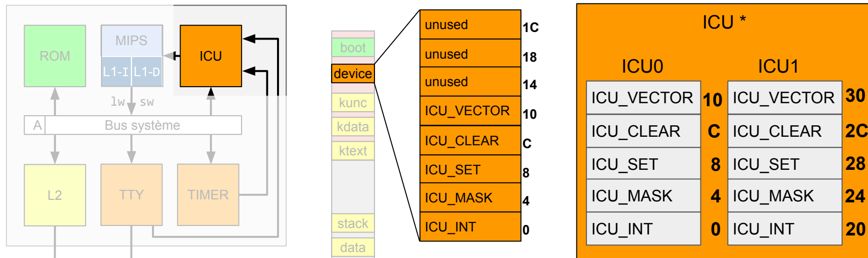


Architecture de la plateforme / ICU

http://www.soclib.fr/trac/dev/wiki/Component/VciMultiIcu

L'ICU est un multiconcentrateur de signaux d'IRQ. Chaque IRQ peut être masquée.
Dans le cas d'une architecture multicores, il permet de router les IRQ vers n'importe quel core.
C'est un périphérique cible contrôlé par des accès en lecture / écriture dans ses registres.

- ICU_INT (lecture seule) état des lignes IRQ
- ICU_MASK (lecture seule) masques des lignes IRQ
- 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_VECTOR (lecture seule) numéro de la ligne IRQ active d'index la plus petite

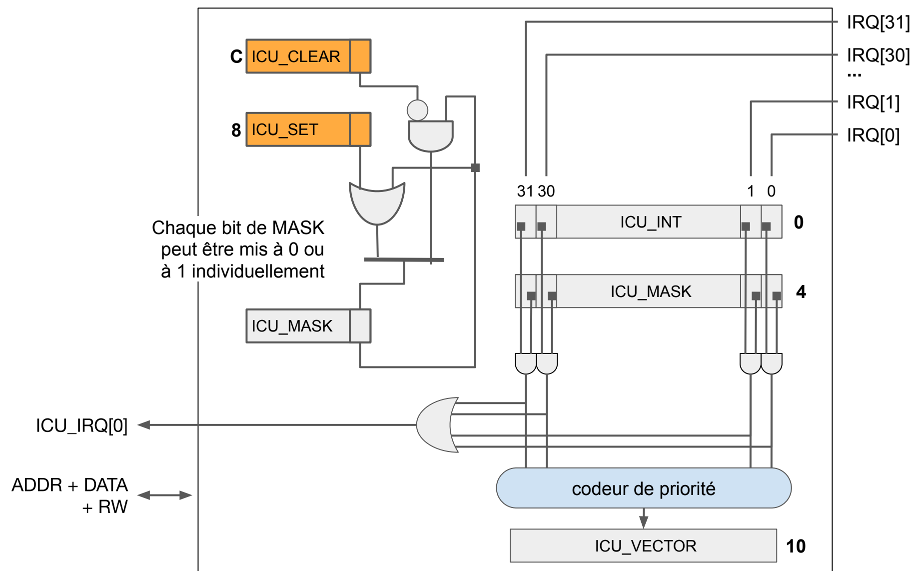


ICU *			
ICU0		ICU1	
ICU_VECTOR	10	ICU_VECTOR	30
ICU_CLEAR	C	ICU_CLEAR	2C
ICU_SET	8	ICU_SET	28
ICU_MASK	4	ICU_MASK	24
ICU_INT	0	ICU_INT	20

* Le composant dessiné contient 2 ICU, le nombre est configurable. Dans la plateforme à gauche, 1 seul ICU est nécessaire.

Architecture de la plateforme / ICU

http://www.soclib.fr/trac/dev/wiki/Component/VciMultiIcu

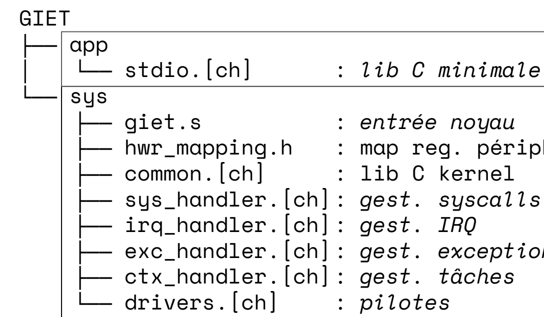


Système GIET

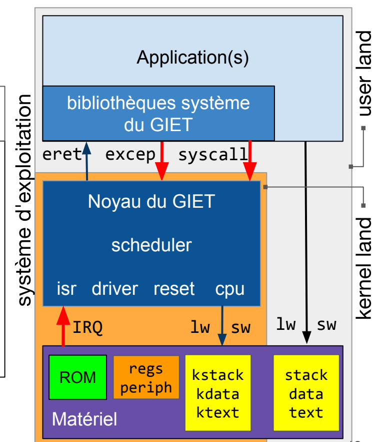
Système GIET

Le système GIET est un système d'exploitation statique permettant à plusieurs applications coopératives de se partager la plateforme matérielle.

- Toutes les applications sont compilées ensemble pour former un seul code binaire.
- Le système GIET est composé
  - d'une bibliothèque système avec un libc minimale
  - d'un noyau qui gère les ressources matérielles et logicielles proposées aux applications



```
GIET
├── app
│   └── stdio.[ch]        : lib C minimale
└── sys
    ├── giet.s            : entrée noyau
    ├── hwr_mapping.h     : map reg. périph
    ├── common.[ch]       : lib C kernel
    ├── sys_handler.[ch]  : gest. syscalls
    ├── irq_handler.[ch]  : gest. IRQ
    ├── exc_handler.[ch]  : gest. exception
    ├── ctx_handler.[ch]  : gest. tâches
    └── drivers.[ch]      : pilotes
```



Noyau de système d'exploitation

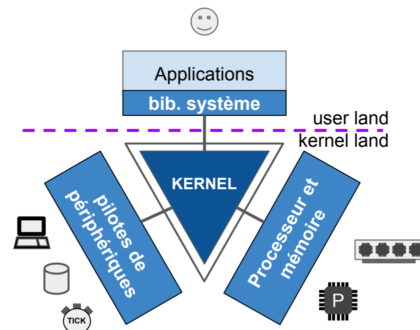
Le noyau est la partie du code du système d'exploitation qui gère les ressources matérielles et logicielles pour le compte des applications.

- Le noyau propose une abstraction de services pour les applications utilisateur grâce à l'API des appels système (syscall). Ces services sont :
  - L'accès au matériel: entrées-sorties, disque, mémoire, processeur, cache,...
  - *Le démarrage de nouveaux processus* (programme en exécution) *
  - *La communication entre les processus* *
  - *Le système de fichiers et les sockets réseau* *
- *Le noyau propose aussi une abstraction de services pour les pilotes de périphériques afin de pouvoir facilement ajouter de nouveaux périphériques* *
- *Le noyau propose enfin une abstraction de services pour l'architecture de l'ordinateur (processeur, mémoire, opérations atomiques) pour faciliter le portage sur les ordinateurs* *
- Le noyau est un code sûr (*sans bug*), il garantit :
  - Le bon usage du matériel (pas de configuration dangereuse)
  - *La sécurité des processus (confidentialité et intégrité des données)* *
  - *L'équité ou la priorité d'accès aux services par les processus* *

⇨ *En gris italique, ce sont les services et les garanties que n'offre pas ou pas bien le GIET*

Interfaces (API) du noyau

- Un noyau a toujours 3 interfaces :
  - avec les utilisateurs
  - avec les périphériques
  - avec le processeur et la mémoire
- Chaque interface est définie par une API (Application Programming Interface)
  - Utilisateur : syscall
  - Périphérique : open, read, write, etc.
  - Proc et Mem : context, IRQ, cache flush



Ce découpage est important parce qu'en général ce ne sont pas les mêmes personnes qui développent les différents codes.

> Attention dans le cas du GIET, l'interface avec les périphériques, le processeur et la mémoire n'est pas bien formalisée parce qu'il n'y a pas d'ambition d'ajouter facilement de nouveaux périphériques ni de faire de multiples portages.

Entrée dans le noyau

- Le code est dans le fichier giet.s (p. 2)
- L'entrée dans le noyau se fait toujours à l'adresse 0x80000180 quelque soit la cause d'appel : interruption, exception ou syscall
- Quelque soit la cause
  - SR [EXL] ← 1 ⇒ mode kernel, interruption masquée
  - CR [XCODE] ← CAUSE ⇒ 16 causes possibles
  - EPC ← PC si c'est un syscall ou une exception
    ← PC+4 si c'est une interruption
    ⇒ on sait où revenir
- Le travail initial du GIET est d'analyser la cause pour aller vers le bon gestionnaire
  - _int_handler pour une interruption (p. 3)
  - _sys_handler pour un syscall (p. 2)
  - _cause_xxx pour les exception (p. 21, p. ex. _cause_ovf)

Interface du noyau / utilisateur / bibliothèque système

- L'application utilisateur peut demander un service par l'instruction syscall.
- Ces appels se trouvent dans les fonctions de la bibliothèque système `stdio.`
- Le prototype des fonctions sont dans `stdio.h` (p. 39), par exemple :
  - `unsigned int tty_putc(char byte);`
  - `unsigned int tty_getc(char *byte);`
  - `unsigned int tty_getc_irq(char *byte);`
- Le code de ces fonctions est dans `stdio.c`

```
unsigned int tty_putc(char byte) {
    return sys_call(   SYSCALL_TTY_WRITE ,     // numéro d'appel système → $2
                       (unsigned int) &byte ,  // arg0  → $4
                       1,                      // arg1  → $5
                       0,                      // arg2  → $6
                       0);                     // arg3  → $7
}
```

- La fonction `sys_call()` (p. 41) est contient le code assembleur permettant d'appeler l'instruction `syscall` avec les arguments `arg0` à `arg3` dans les bons registres puis de récupérer le résultat dans $2 (code d'erreur)

Interface du noyau / utilisateur / sys_call()

```
static inline unsigned int sys_call( unsigned int call_no ,
                                     unsigned int arg_0 , unsigned int arg_1 ,
                                     unsigned int arg_2 , unsigned int arg_3)
{
     register unsigned int reg_no_and_output    asm("v0") = call_no; // $2 ← call_no
     register unsigned int reg_a0               asm("a0") = arg_0;   // $4 ← arg_0
     register unsigned int reg_a1               asm("a1") = arg_1;   // $5 ← arg_1
     register unsigned int reg_a2               asm("a2") = arg_2;   // $6 ← arg_2
     register unsigned int reg_a3               asm("a3") = arg_3;   // $7 ← arg_3

     asm volatile(   "syscall"  // code assembleur (volatile bloque les optimisations de gcc
                  : "=r" (reg_no_and_output) // output argument
                  : "r" (reg_a0),            // input arguments
                    "r" (reg_a1),
                    "r" (reg_a2),
                    "r" (reg_a3),
                    "r" (reg_no_and_output)
                  : "memory",      // informe gcc que le code assembleur accède à la mémoire, donc
                                   // gcc ne doit pas utiliser les registres comme des caches
                    // ces registres (temporaires) seront sauvés dans la pile par gcc uniquement
                    // s'ils contiennent des valeurs importantes
                    "at", "v1", "ra", "t0", "t1", "t2", "t3", "t4", "t5", "t6", "t7", "t8", "t9" );

     return reg_no_and_output;
}
```


Interface du noyau / utilisateur / sys_call()

Extrait de app.bin produit par le compilateur

```
00400000 <sys_call>:
  // Prologue
  400000:     27bdfff8     addiu   sp,sp,-8      // réserve 2 cases dans la pile
  400004:     afbf0004     sw      ra,4(sp)      // sauve $31
  400008:     afa40008     sw      a0,8(sp)      // place les arguments de sys_call dans la pile
  40000c:     afa5000c     sw      a1,12(sp)
  400010:     afa60010     sw      a2,16(sp)
  400014:     afa70014     sw      a3,20(sp)

  // corps
  400018:     8fa20008     lw      v0,8(sp)      // initialise les registres pour le syscall
  40001c:     8fa4000c     lw      a0,12(sp)
  400020:     8fa50010     lw      a1,16(sp)
  400024:     8fa60014     lw      a2,20(sp)
  400028:     8fa70018     lw      a3,24(sp)
  40002c:     0000000c     syscall               // syscall

  // epilogue
  400030:     8fbf0004     lw      ra,4(sp)      // restaure $31
  400034:     03e00008     jr      ra            // retour à la fonction appelante
  400038:     27bd0008     addiu   sp,sp,8       // exécuté à cause du saut retardé du MIPS
```


Interface du noyau / utilisateur

Liste de tous les appels système proposés par le GIET

```
#define  SYSCALL_PROCID         0x00 // lecture du numéro de processeur
#define  SYSCALL_PROCTIME       0x01 // lecture du nombre de cycle depuis reset
#define  SYSCALL_TTY_WRITE      0x02 // écriture d'un buffer sur le TTY associé au core
#define  SYSCALL_TTY_READ       0x03 // lecture d'un caractère depuis le TTY associé au core
#define  SYSCALL_TIMER_WRITE    0x04 // écriture dans un registre de contrôle du TIMER
#define  SYSCALL_TIMER_READ     0x05 // lecture d'un registre d'état du TIMER
#define  SYSCALL_GCD_WRITE      0x06 //
#define  SYSCALL_GCD_READ       0x07 //
#define  SYSCALL_TTY_READ_IRQ   0x0A // lecture d'un car. dans le FIFO d'entrée du TTY du core
#define  SYSCALL_TTY_WRITE_IRQ  0x0B // écriture d'un car. dans la FIFO de sortie du TTY du core
#define  SYSCALL_CTX_SWITCH     0x0D //
#define  SYSCALL_EXIT           0x0E //
#define  SYSCALL_PROCNUMBER     0x0F //
#define  SYSCALL_FB_SYNC_WRITE  0x10 //
#define  SYSCALL_FB_SYNC_READ   0x11 //
#define  SYSCALL_FB_WRITE       0x12 //
#define  SYSCALL_FB_READ        0x13 //
#define  SYSCALL_FB_COMPLETED   0x14 //
#define  SYSCALL_IOC_WRITE      0x15 //
#define  SYSCALL_IOC_READ       0x16 //
#define  SYSCALL_IOC_COMPLETED  0x17 //
#define  SYSCALL_BARRIER_INIT   0x18 //
#define  SYSCALL_BARRIER_WAIT   0x19 //
```


Interface du noyau / périphériques

- Il y a plusieurs fonctions d'accès qui dépendent du périphérique.
- Le noyau connaît le mapping des registres de périphériques et il y écrit comme s'il s'agissait de variables globales.
- Attention, comme ces registres peuvent changer de valeur à tout instant
  - Elles ne doivent pas être écrites dans le cache
  - Elles ne doivent pas non plus être conservées dans les registres.
    C'est pourtant ce qu'essaie de faire GCC pour optimiser l'usage de la mémoire. En effet, quand GCC travaille sur une variable globale, il lui assigne un registre GPR et évite de faire des accès en mémoire.
    C'est un problème pour les registres de périphérique.
    C'est pourquoi on place le mot clé volatile devant les déclarations des adresses de périphériques.

Interface du noyau / périphériques

- Pour le TTY (p. 10 et p .26)
  - `unsigned int _tty_config(unsigned int tty_id , unsigned int proc_id , unsigned int task_id)`
  - `unsigned int _tty_write(const char * buffer , unsigned int length)`
  - `unsigned int _tty_read(char * buffer , unsigned int length)`
  - `unsigned int _tty_read_irq(char * buffer , unsigned int length)`
  - `void _isr_tty_get_task0()`
  - `void _isr_tty_get_task1()`
  - `void _isr_tty_get_task2()`
  - `void _isr_tty_get_task3()`
- Pour le TIMER (p. 9 et p. 25)
  - `unsigned int _timer_write(unsigned int register_index , unsigned int value)`
  - `unsigned int _timer_read(unsigned int register_index , unsigned int * buffer)`
  - `void _isr_timer()`
- Pour l'ICU (p. 10)
  - `unsigned int _icu_write(unsigned int register_index , unsigned int value)`
  - `unsigned int _icu_read(unsigned int register_index , unsigned int * buffer)`

fonction write du pilote de TTY

```c
unsigned int _tty_write(const char *buffer, unsigned int length)
{
    volatile unsigned int *tty_address;                     // adresse volatile du tty

    unsigned int proc_id;
    unsigned int task_id;
    unsigned int tty_id;
    unsigned int nwritten;

    proc_id = _procid();                                    // numéro du processeur
    task_id = _current_task_array[proc_id];                 // numéro du thread
    tty_id  = proc_id*NB_MAXTASKS + task_id;                // numéro du TTY attribué par défaut

    tty_address = (unsigned int*)&seg_tty_base + tty_id*TTY_SPAN;   // adresse des regs de ce TTY

    for (nwritten = 0; nwritten < length; nwritten++)       // pour tous les caractères
    {
        if ((tty_address[TTY_STATUS] & 0x2) == 0x2)         // test s'il y a un bourrage
            break;                                          // si oui sortir
        else
            tty_address[TTY_WRITE] = (unsigned int)buffer[nwritten]; // sinon écrire
    }

    return nwritten;                                        // rend le nb de caractères écrits
}
```


Interface du noyau / processeur

- Cette API permet de demander au processeur
  - de sauver son état (l'ensemble de ces registres) dans un tableau, ou de restaurer un état depuis un tableau.
  - de masquer ou démasquer les interruptions.
  - de faire des opérations atomiques : incrémentation, prise de verrou
- Nous verrons cette API lorsque nous présenterons l'ordonnanceur de tâches.

code GIET

Entrée du 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 */ };
```


Appel système

- Au début de _sys_handler,
  - $2 contient le numéro du service demandé
  - $4 à $7 contiennent les arguments
- Le noyau va simplement appeler une fonction pour le service demandé.
  La valeur de retour de cette fonction sera la valeur de retour du syscall

Ce qu'il faut faire, c'est :

1. Allouer dans la pile de la place pour le registre SR, l'adresse de retour et les arguments
2. Sauver le registre SR et l'adresse de retour (EPC + 4)
3. Aller chercher l'adresse de la fonction dans le tableau `_syscall_vector[]` `_syscall_vector[$2 * 4]`
4. Passer en mode kernel avec IRQ : `SR[EXL]←0 SR[UM]←0 SR[INT]←1`
5. Appeler la fonction
6. Passer en mode kernel sans IRQ : `SR←0`
7. Restaurer le registre SR (EXL est à 1), le registre EPC et le pointeur de pile
8. Sortir

Appel système et sortie du noyau

```
_sys_handler:
     addiu   $29 , $29 , -24       /* 2 slots for SR&EPC , 4 slots for args passing */
     mfc0    $26 , $12             /* load SR */
     sw      $26 , 16($29)         /* save it in the stack */
     mfc0    $27 , $14             /* load EPC */
     addiu   $27 , $27 , 4         /* increment EPC for return address */
     sw      $27 , 20($29)         /* save it in the stack */

     andi    $26 , $2 , 0x1F       /* $26 <= syscall index (i < 32) */
     sll     $26 , $26 , 2         /* $26 <= index * 4 */
     la      $27 , _syscall_vector /* $27 <= & _syscall_vector [0] */
     addu    $27 , $27 , $26       /* $27 <= & _syscall_vector [i] */
     lw      $3, 0($27)            /* $3 <= syscall address */
     li      $27 , 0xFFFFFFED      /* Mask for UM & EXL bits */
     mfc0    $26 , $12             /* $26 <= SR */
     and     $26 , $26 , $27       /* UM = 0 / EXL = 0 */
     mtc0    $26 , $12             /* interrupt enabled */

     jalr    $3                    /* jump to the proper syscall */

     mtc0    $0 , $12              /* interrupt disbled */
     lw      $26 , 16($29)         /* load SR from stack */
     mtc0    $26 , $12             /* restore SR */
     lw      $26 , 20($29)         /* load EPC from stack */
     mtc0    $26 , $14             /* restore EPC */
     addiu   $29 , $29 , 24        /* restore stack pointer */
     eret                          /* exit GIET */
```


vecteur des appels système : _syscall_vector

Dans le fichier sys_handler.c (p. 27) :
_syscall_vector : tableau de pointeurs des fonctions réalisant les appels système

```
const void *_syscall_vector[32] = {
    &_procid,           /* 0x00 */
    &_proctime,         /* 0x01 */
    &_tty_write,        /* 0x02 */
    &_tty_read,         /* 0x03 */
    &_timer_write,      /* 0x04 */
    &_timer_read,       /* 0x05 */
    &_gcd_write,        /* 0x06 */
    &_gcd_read,         /* 0x07 */
    [...]
    &_ioc_write,        /* 0x15 */
    &_ioc_read,         /* 0x16 */
    &_ioc_completed,    /* 0x17 */
    &_barrier_init,     /* 0x18 */
    &_barrier_wait,     /* 0x19 */
    &_sys_ukn,          /* 0x1A */
    &_sys_ukn,          /* 0x1B */
    &_sys_ukn,          /* 0x1C */
    &_sys_ukn,          /* 0x1D */
    &_sys_ukn,          /* 0x1E */
    &_sys_ukn,          /* 0x1F */
};
```

Interruption

- L'interruption arrive n'importe quand si elles sont autorisées.
- Le noyau doit "voler" des cycles au programme interrompu pour exécuter l' ISR.
- Le noyau va appeler une fonction _int_demux qui va déterminer quelle est l'IRQ et appeler la bonne ISR.
- _int_démux est une fonction C qui va donc protéger les registres persistants, mais pas les registres temporaires

Ce qu'il faut faire, c'est :

1. Allouer dans la pile de la place pour tous les registres temporaires et pour le registre EPC.
2. appeler la fonction _int_demux()
3. Restaurer les registres temporaires, le registre EPC et le pointeur de pile.
4. Sortir

Interruption

_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 _interrup_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];   // vecteur des int
        isr();
    }
}
```


En résumé, nous avons vu :

- les périphériques TTY (rappel), TIMER et ICU (nouveau)
- comment sont gérés les syscall et les interruptions par le GIET
- comment sortir du noyau

En TD et en TME, vous allez entrer dans le code et l'analyser.
Le but est de vous l'approprier … en partie.

Prochaine séance

Nous allons voir les périphériques initiateur.
Ils réalisent des opérations sur la mémoire en parallèle avec le processeur et ce n'est pas sans conséquence.

- Nous allons voir les problèmes de cohérence et de synchronisation.

- Nous verrons également comment exploiter ces périphériques pour accélérer les applications.