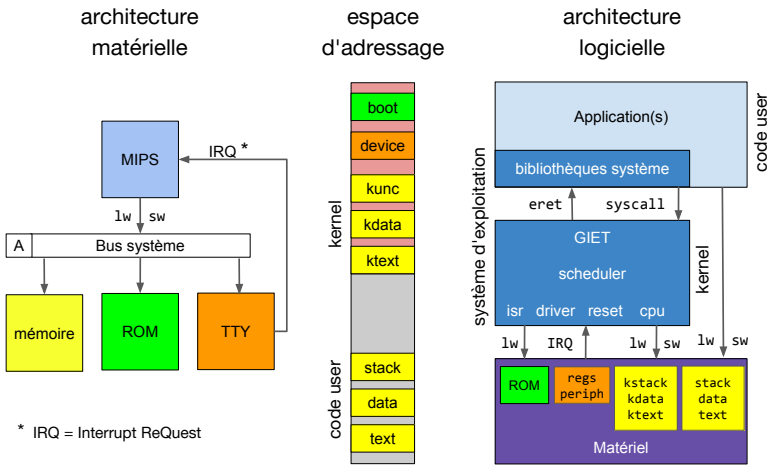
ALMO

Bus système et périphériques / GIET

Questions

- Comment contrôler un périphérique ?
- Où est le système d'exploitation ?
- Comment l'application accède à la mémoire
- Comment l'application accède aux périphériques ?
- Comment le noyau analyse la demande de service ?
- Quels sont la structure du systèmes sur le disque ?
- Comment compile-t-on le noyau et l'application ?
- Comment le noyau sait-il où est la fonction main() ?
- Comment le système puis l'application démarre ?

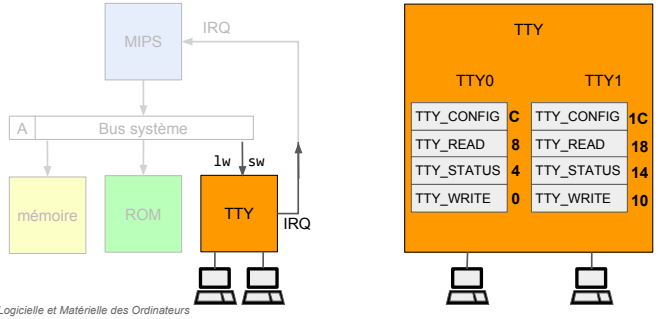
Une image pour résumer ce cours



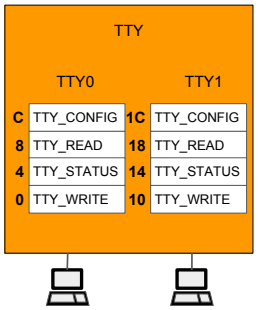
Périphérique cible

Un périphérique (device) cible est un composant réalisant un service spécifique, il est contrôlé par des accès en lecture ou en écriture dans ses registres.

Le contrôleur de terminaux texte (TTY)
qui permet de gérer un ou plusieurs couples écran-clavier
est un contrôleur de périphériques cibles



Contrôleur de terminaux TTY



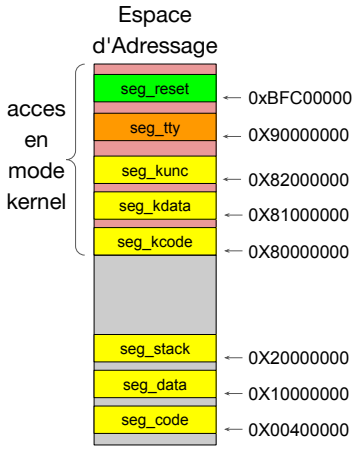
Tous les registres sont alignés sur des mots, chaque terminal utilise un segment de 4 mots (16 octets).

Pour chaque terminal contrôlé

- TTY_WRITE 1 octet en écriture seule
  sortie vers l'écran
- TTY_STATUS 1 octet en lecture seule
  != 0 s'il y a un caractère en attente dans TTY_READ
- TTY_READ 1 octet en lecture seule
  caractère tapé au clavier
- TTY_CONFIG inutilisé dans cette version
  permet la configuration p. ex. le débit d'échange avec le terminal

Nous verrons d'autres contrôleurs de périphériques. Leurs registres de contrôle, nom et adresse, sont donnés page 37 et 38 du document. sur le code du GIET.

Mapping de l'espace d'adressage

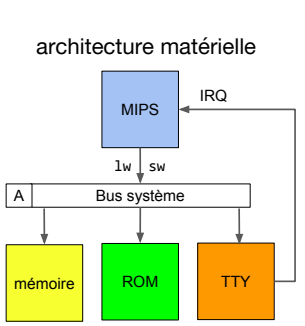


Le choix du mapping de la mémoire physique (cartographie) est fait par l'architecte de la machine.

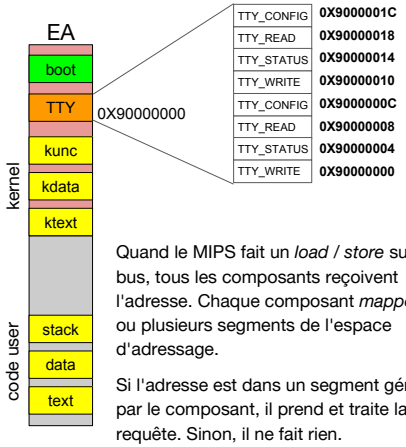
C'est un mapping statique puisque ce sont des segments de mémoire physiques. Il ne peut pas y avoir de création de segment.

Les segments ne peuvent être accédés que si le MIPS est en mode kernel.

Accès aux registres de périphériques



Les registres de périphériques sont placés dans l'espace d'adressage afin qu'ils puissent être lus et écrits par le MIPS.



Quand le MIPS fait un *load* / *store* sur le bus, tous les composants reçoivent l'adresse. Chaque composant *mappe* un ou plusieurs segments de l'espace d'adressage.

Si l'adresse est dans un segment géré par le composant, il prend et traite la requête. Sinon, il ne fait rien.

Application utilisateur

- L'application utilisateur est un programme avec d'au moins une fonction `main()`.
- Le code de l'application, ses données et sa pile sont directement accessibles en mémoire à condition qu'ils soient dans la partie autorisée au processeur en mode utilisateur (pour ce MIPS les adresses < 0x80000000)
- Lorsque l'application veut accéder à un périphérique, elle doit demander ce service au noyau par l'exécution de l'instruction syscall

```
$4 à $7     ⇐ arguments du service
$2          ⇐ numéro du service
syscall     = jal kernel
$2          ⇒ code d'erreur et 0 si succès
```

- `syscall` est une instruction non-privilégiée (exécutable en mode User) c'est l'unique instruction permettant d'entrer dans le noyau depuis l'application.
- L'application peut provoquer l'entrée dans le noyau lors des exceptions telles que l'exécution d'instructions illégales, les violations de privilège ou les divisions par 0.

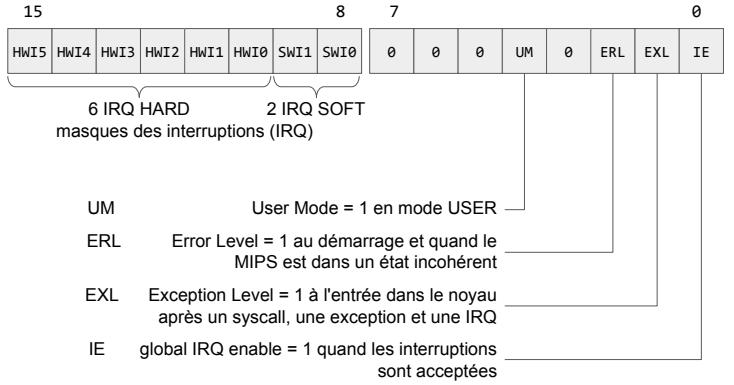
Entrée dans le noyau depuis l'application

Il y a deux types de raison d'appeler le noyau : syscall ou exception
Dans les deux cas, on saute à l'adresse d'entrée du noyau : `0x80000180`

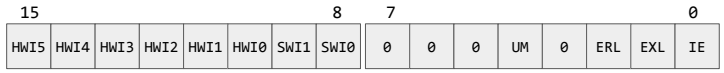
1. L'exécution de `syscall` provoque :
   - `EPC` ← `PC` : c'est à dire l'adresse de l'instruction syscall
   - `SR[EXL]` ← `1` : mise à 1 du bit EXL du registre Status Register
   - `CR[XCODE]` ← `8` : la cause d'appel est mise dans le champ XCODE du registre de cause
   - `PC` ← `0x80000180` : adresse d'entrée du noyau
2. Une exception survient toujours à l'exécution d'une instruction en faute
   La majorité des exception sont mortelles, nous ne serons pas détaillées aujourd'hui
   - `EPC` ← `PC` : c'est à dire l'adresse de l'instruction fautive
   - `SR[EXL]` ← `1` : mise à 1 du bit EXL du registre Status Register
   - `CR[XCODE]` ← *cause* : la cause de l'exception est mise dans le champ XCODE du registre de cause
   - `PC` ← `0x80000180` : adresse d'entrée du noyau

Contenu du Status Register SR

Le registre SR contient les masques des lignes d'interruption et le mode d'exécution.



Comportement du registre SR



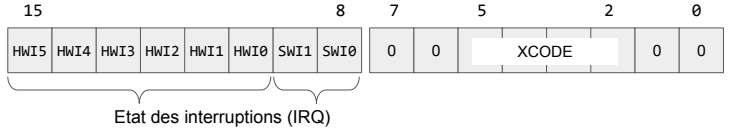
Comportement du MIPS
- Si `UM` est à 1: le MIPS est en mode USER
- Si `IE` est à 1 : le MIPS autorise les IRQ à interrompre le programme courant

SAUF SI `ERL` ou `EXL` sont à 1
- Si l'un des bits `ERL` ou `EXL` est à 1 alors
  le MIPS est en mode KERNEL IRQ masquée quelque-soit l'état de `UM et IE`

Valeurs typiques de SR
- Lors de l'exécution d'une application USER → `0xFF11`
- À `l'entrée dans le noyau` → `0xFF13`
- Pendant l'exécution d'un syscall → `0xFF01`
- Pendant l'exécution d'une section critique → `0xFF00`

Cause Register CR

Le registre CR contient la cause d'entrée dans le noyau



Valeurs de XCODE effectivement utilisés dans cette version du MIPS

0000	INT	Interruption
0100	ADEL	Adresse illégale en lecture
0101	ADES	Adresse illégale en écriture
0110	IBE	Bus erreur sur accès instruction
0111	DBE	Bus erreur sur accès donnée
1000	SYS	Appel système (SYSCALL)
1001	BP	Point d'arrêt (BREAK)
1010	RI	Codop illégal
1011	CPU	Coprocesseur inaccessible
1100	OVF	Overflow arithmétique

Traitement d'un appel système

- Un appel système doit être considéré comme un appel de fonction avec les privilèges du noyau.
  - Les registre $4 à $7 contiennent les args et $2 le numéro de service
  - au retour $2 contient le résultat
  - Les registres temporaires sont perdus et les persistants sont conservés
- Traitement *
  1. Analyse du registre CR
     - si `CR[XCODE]==8` alors saut à la routine de gestion du syscall
  2. Gestion du syscall
     - Allocation dans la pile de l'espace pour
       sauver l'adresse de retour (EPC+4)
       réserver de l'espace pour les arguments actuellement dans $4 à $7
     - Appel de la fonction dont le numéro est dans $2
       cette fonction exécute le service et met son résultat dans $2
     - Restauration de l'adresse de retour dans EPC
     - Restauration du pointeur de pile
     - retour vers le programme utilisateur

* Nous regarderons en détail le code du GIET plus tard dans ce cours

`eret` : Retour du noyau vers l'application

- L'exécution de eret provoque de manière atomique
  - `SR[EXL]` ← `0` : mise à 0 du bit EXL du registre Status Register
  - `PC` ← `EPC` : EPC contient l'adr. de la prochaine instruction

C'est cette instruction qui permet de sortir du noyau

Comment démarre le système

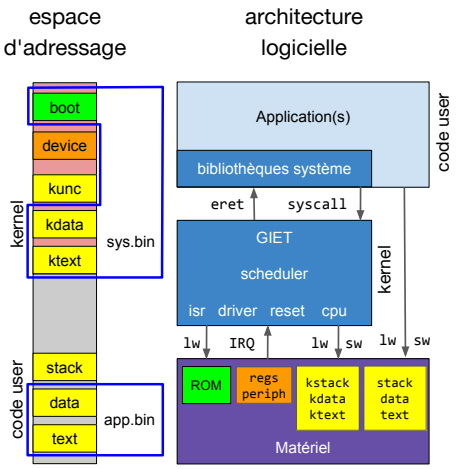
- Le processeur démarre en mode KERNEL
- Le code de démarrage (boot) se trouve à l'adresse `0xBFC00000`
- Il est chargé d'initialiser les périphériques et les structures du noyau.
  Nous verrons ça en détail plus tard.
  Pour l'instant, il se contente d'initialiser SR et SP
  - `SR` ← `0xFF13`
  - `SP` ← `seg_stack + stack_size`
- A la fin, il initialise EPC avec l'adresse de la fonction `main()`
  Nous allons voir comment le noyau peut trouver l'adresse de `main()`
  - `EPC` ← `&main`
- Enfin, il exécute `eret` pour entrer dans l'application utilisateur

Analyse des fichiers du noyau

```
GIET
├── app
│   ├── stdio.c      : lib C minimaliste
│   └── stdio.h
└── sys
    ├── giet.s           : entrée dans le noyau
    ├── sys_handler.c    : gestionnaire syscalls
    ├── irq_handler.c    : gestionnaire IRQ
    ├── exc_handler.c    : gestionnaire exception
    ├── ctx_handler.c    : gestionnaire des tâches
    ├── common.c         : lib C pour le kernel
    ├── drivers.c        : pilote de périphériques
    ├── common.h
    ├── hwr_mapping.h
    ├── sys_handler.h        USER
    ├── irq_handler.h        ├── Makefile
    ├── exc_handler.h        ├── main.c     : code application
    ├── ctx_handler.h        ├── config.h   : configuration hardware
    └── drivers.h            ├── app.ld     : ldscript application
                             ├── reset.s    : code du reset
                             ├── seg.ld     : mapping memoire
                             └── sys.ld     : ldscript systeme
```

Compilation du noyau et de l'application

- Deux binaires :
  - sys.bin : code kernel
  - app.bin : code user
- Chaque fichier source est compilé séparément puis on les rassemble lors de l'étape d'édition de liens



Comment le noyau retrouve la fonction main

```
#include <stdio.h>
__attribute__((constructor)) void main(void)
{
    char byte;
    char str[] = "\nHello World!\n";

    while (1) {
        tty_puts(str);
        tty_getc(&byte);

        if (byte == 'q') {
            exit(0);
        }
    }
    exit(0);
}
```

main.c

→ créé une section **.ctors** dans le .o contenant juste l'adresse de main

app.ld

```
INCLUDE seg.ld
SECTIONS
{
    . = seg_code_base;
    seg_code : {
        *(.text)
    }
    . = seg_data_base;
    seg_data : {
        *(.ctors)
        *(.rodata)
        *(.rodata.*)
        *(.data)
        *(.lit8)
        *(.lit4)
        *(.sdata)
        *(.bss)
        *(.sbss)
    }
}
```

Grace à l'attribut "constructor" et app.ld
On assure que l'adresse du main se trouve dans le premier mot de seg_data dont l'adresse est connue du noyau, le reset peut la lire et y sauter

Appel d'un service système via la libc

```
__attribute__((constructor)) void main(void)
{
    char byte;
    char str[] = "\nHello World!\n";

    while (1) {
        tty_puts(str);
        tty_getc(&byte);

        if (byte == 'q') {
            exit(0);
        }
    }
    exit(0);
}
```

```
unsigned int tty_puts(char * buf)
{
    unsigned int length = 0;
    while (buf[length] != 0) {
        length++;
    }
    return sys_call(SYSCALL_TTY_WRITE ,
                    (unsigned int) buf ,
                    length ,
                    0, 0);
}
```

syscall | eret app/stdio.c

```
unsigned int _tty_write(const char * buffer , unsigned int length)
{
    volatile unsigned int * tty_address;
    [ ... ]
    tty_address = (unsigned int *) &seg_tty_base + tty_id*TTY_SPAN;
    for (nwritten = 0; nwritten < length; nwritten++)
    {
        /* check tty 's status */
        if ((tty_address[TTY_STATUS] & 0x2) == 0x2)
            break;
        else
            /* write character */
            tty_address[TTY_WRITE] = (unsigned int) buffer[nwritten];
    }
    return nwritten;
}
```

sys/drivers.c

GIET

technique pour trouver le premier registre du TTY concerné

Prochaine séance, les caches….

Le MIPS lit jusqu'à une instruction par cycle mais l'accès à la mémoire peut prendre plusieurs dizaines de cycles, c'est incompatible.
La solution est d'utiliser des caches ...