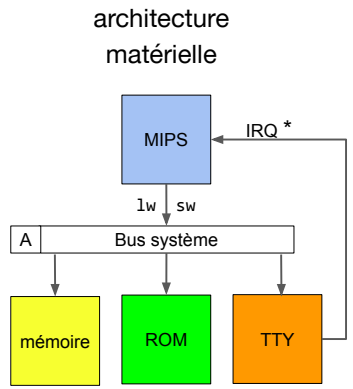
ALMO

Architecture matérielle de la machine ALMO
Présentation du GIET

Questions

- Comment contrôler un périphérique ?
- Comment l'application accède à la mémoire ?
- Comment l'application accède aux périphériques ?
- Quels sont les rôles du système d'exploitation ?
- Où est le système d'exploitation ?
- Qu'est-ce que le noyau du système d'exploitation ?
- Comment demande-t-on un service au système d'exploitation ?
- Comment le noyau traite un appel système ?
- Comment compile-t-on le noyau et l'application ?
- Comment le noyau sait-il où est la fonction main() de l'application ?
- Comment le système et l'application démarrent ?

Le matériel

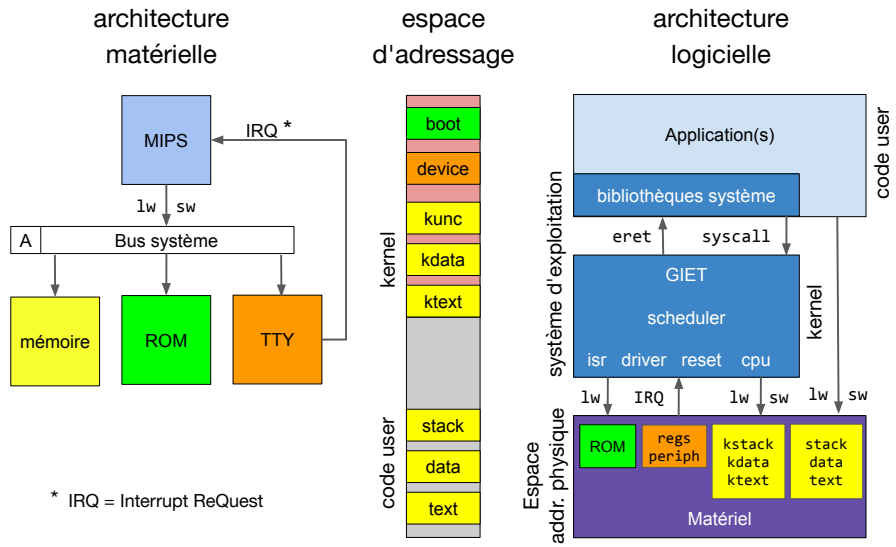


* IRQ = Interrupt ReQuest

Ce schéma représente l'architecture de principe de l'ordinateur.

- le cœur MIPS en haut qui interagit avec le reste par des instructions load/store
- ces requêtes load/store sont routées par le bus système vers les composants qui mappent les adresses concernées
- En bas, se trouve une mémoire normale, une mémoire en lecture seule et au moins un contrôleur d'entrée-sortie (ici le terminal)
- Le terminal peut envoyer un signal d'état IRQ (Interrupt ReQuest) pour demander au noyau de prendre en compte un caractère tapé au clavier

Une image pour résumer ce cours

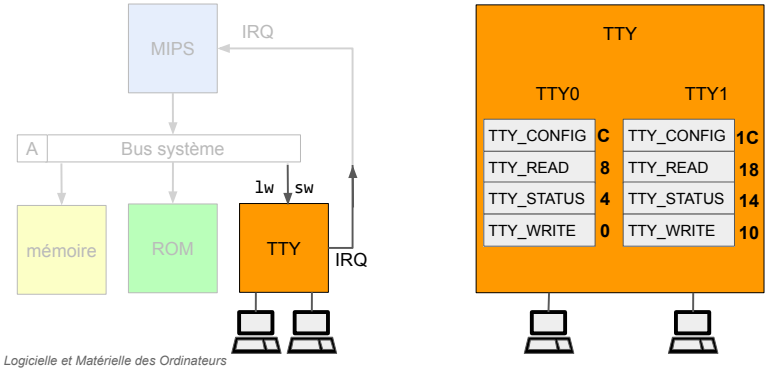


* IRQ = Interrupt ReQuest

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.

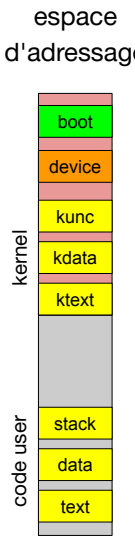
Le contrôleur de terminaux texte (TTY)
qui permet de gérer un ou plusieurs couples écran-clavier
est un contrôleur de périphériques cibles



L'espace d'adressage

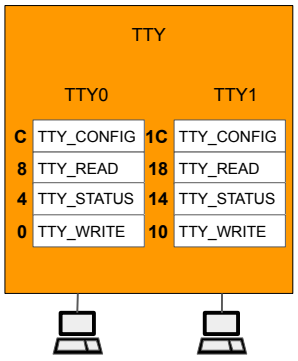
Ce schéma représente l'espace d'adressage du processeur.

- Les adresses basses sont utilisées par l'application pour son code (text), ses données globales (data) et sa pile (stack)
- Les adresses hautes à partir de `0x8000 0000` sont réservées au noyau du système d'exploitation pour son code (ktext), ses données globale (kdata).
- Le segment device rassemble les adresses où sont mappées les registres des périphériques (devices)



- Le segment boot contient les adresses de démarrage du processeur. Ce segment est dans la partie haute et donc exécutable en mode kernel
- Le segment kunc est utilisé pour la communication entre "tâches". Nous verrons son usage plus tard.
- Vous pouvez remarquer l'absence de segment kstack parce que le GIET utilise la pile de l'utilisateur pour exécuter le code noyau.

Contrôleur de terminaux TTY



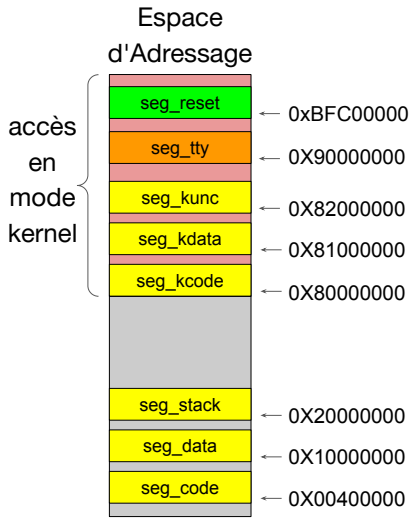
Tous les registres sont alignés sur des mots,
chaque terminal utilise un segment de 4 mots (16 octets).

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

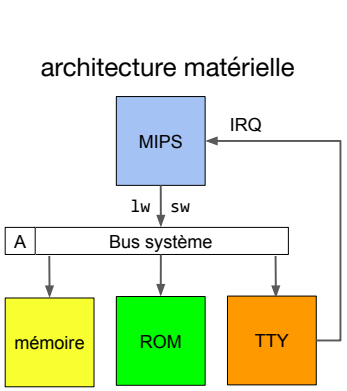


Le choix du mapping de la mémoire physique (cartographie) est fait par l'architecte de la machine.

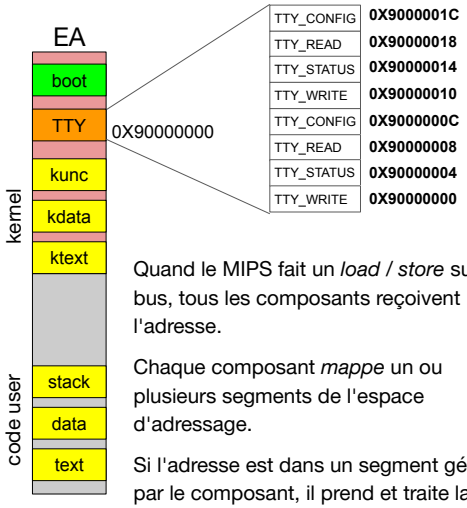
C'est un mapping statique puisque ce sont des segments de mémoire physique. Il ne peut pas y avoir de création de segment de mémoire physique.

Les segments au delà de 0x80000000 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.

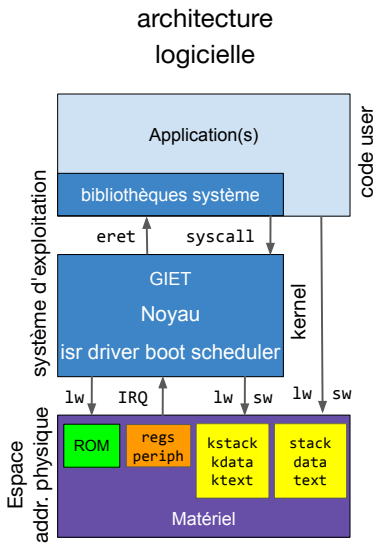
Chaque composant *mappe* un ou plusieurs segments de l'espace d'adressage.

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

La place du système d'exploitation

Ce schéma représente les interactions entre les applications, le système d'exploitation et le matériel

- En haut, les applications. Elles peuvent accéder directement à certains segments de l'espace d'adressage. Mais, elles doivent passer par des bibliothèques système pour accéder au matériel (autre que le processeur lui-même bien sûr).
- Les bibliothèques système sont des ensembles de fonctions définissant des API (Application Programming Interface) proposant des abstractions du matériel. Ces fonctions sont exécutées en mode utilisateur. Elles demandent des services au noyau par des syscalls (appel système). Le retour à l'utilisateur se fait par eret.
- Au centre, il y a le noyau de l'OS (kernel), il reçoit ses ordres de l'application (syscall) ou du matériel (interruption et exception).
- En bas, le matériel accessible par des lectures ou des écritures dans l'espace d'adressage



Rôles et devoirs du noyau d'un système d'exploitation

Rôles :

1. Gérer les ressources matérielles et les services pour les applications
   - commande aux périphériques : terminal, disque, réseau, etc.
   - allocation de mémoire
   - allocation du processeur
2. Gérer les événements émanant des périphériques
   - fin de commande aux périphériques
   - Tick d'horloge pour réaliser des opérations périodiques
3. Gérer les erreurs de programmation où les allocations à la volée
   - division par 0
   - violation de privilège
   - instruction illégale
   - faute de page, erreur de segmentation (bus error)

Devoirs : l'OS doit garantir

- la **qualité de service** (QoS) (équité, priorité ou respect des échéances),
- la **sécurité** des applications (confidentialité, intégrité, et accès aux services (DoS))
- la **sûreté** de fonctionnement du matériel (pour éviter la casse)

Attention : Sécurité est différent de Sûreté

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, elle va le faire en passant par une fonction de la bibliothèque système

```
$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'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.

Bibliothèque système : libc 1/2

S'il y a bien une bibliothèque système présente dans tous les OS, c'est la libc.
La libc du GIET est rudimentaire mais suffisante pour notre usage : `stdio.[ch]`
stdio.c devra être compilé avec le programme utilisateur.

```
/* TTY device related functions */
unsigned int tty_putc(char byte);
unsigned int tty_puts(char *buf);
unsigned int tty_putw(unsigned int val);
unsigned int tty_getc(char *byte);
unsigned int tty_getc_irq(char *byte);
unsigned int tty_gets_irq(char *buf, unsigned int bufsize);
unsigned int tty_getw_irq(unsigned int *val);
unsigned int tty_printf(char *format,...);

/* Timer device related functions */
unsigned int timer_set_mode(unsigned int mode);
unsigned int timer_set_period(unsigned int period);
unsigned int timer_reset_irq();
unsigned int timer_get_time(unsigned int *time);

/* système */
unsigned int procid();
unsigned int procnumber();
unsigned int proctime();
unsigned int ctx_switch();
```

Fonctions pour lire et écrire le terminal TTY
: Nous verrons qu'il y a plusieurs TTY dans cette architecture : un par thread, c'est un choix pédagogique imposé. Le numéro du TTY est imposé.

Fonctions pour contrôler le TIMER
: `mode` permet de démarrer le TIMER et de demander une IRQ à chaque période
`period` est défini en cycles

Fonctions pour contrôler le système
: permet de connaître le numéro de la tâche, (nous verrons plus tard), le numéro du cœur, la valeur de son compteur de cycles interne et de demander un changement de tâche

Bibliothèque système : libc 2/2

S'il y a bien une bibliothèque système présente dans tous les OS, c'est la libc.
La libc du GIET est rudimentaire mais suffisante pour notre usage : `stdio.[ch]`
stdio.c devra être compilé avec le programme utilisateur.

```
/* Block device related functions */
unsigned int ioc_read(unsigned int lba, void *buffer, unsigned int count);
unsigned int ioc_write(unsigned int lba, void *buffer, unsigned int count);
unsigned int ioc_completed();

/* Frame buffer device related functions */
unsigned int fb_sync_read(unsigned int offset, void *buffer, unsigned int length);
unsigned int fb_sync_write(unsigned int offset, void *buffer, unsigned int length);
unsigned int fb_read(unsigned int offset, void *buffer, unsigned int length);
unsigned int fb_write(unsigned int offset, void *buffer, unsigned int length);
unsigned int fb_completed();

/* Software barrier related functions */
unsigned int barrier_init(unsigned int index, unsigned int count);
unsigned int barrier_wait(unsigned int index);

/* Misc */
void exit();
unsigned int rand();
static inline void *memcpy(void *_dst, const void *_src, unsigned int size)
```

Fonctions pour le disque
: lire, écrire, attendre

Fonctions pour le FB
: lire, écrire synchrone ou asynchrone et dans ce cas, attendre

Fonctions de synchro
: synchronisation des tâches

Fonctions diverses
: sortie du programme
générateur de nombre
mouvement de données

libc, un exemple de fonction

Les fonctions système vont traiter les données de l'application
et faire un appel au noyau par une instruction syscall :

```
/*
 * tty_putw() : displays the value of a 32-bit word with decimal characters.
 * - The terminal index is implicitely defined by the processor identifier
 *   (and by the task ID in case of multi-tasking).
 * - It doesn't use the TTY_PUT_IRQ interrupt, and the associated kernel buffer.
 * - Returns the number of written characters (should be equal to ten).
 */
unsigned int tty_putw(unsigned int val)          // l'utilisateur demande d'afficher un entier
{
    char buf[10];                                // la fonction système fabrique une chaîne
    unsigned int i;                              // de caractère à partir du nombre
    for (i = 0; i < 10; i++)
    {
        buf[9 - i] = (val % 10) + 0x30;
        val = val / 10;
    }
    return sys_call(SYSCALL_TTY_WRITE,           // appelle de la fonction sys_call()
            (unsigned int) buf,                  // qui cache l'instruction syscall et qui rend $2
            10,                                  // $4 ← buf, $5 ← 10, $6 ← 0, $7 ← 0
            0,0);                                // syscall
}
```

Nous verrons plus tard le détail de la fonc. `sys_call()`, ici, il faut juste savoir ce qu'elle fait.

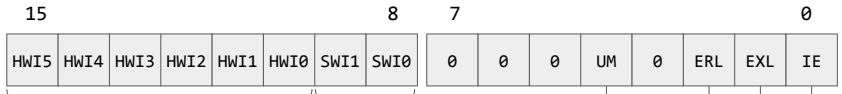
Entrée dans le noyau depuis l'application

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` : sauve l'adresse de l'instruction syscall (SR: `C0.$14`)
   - `SR[EXL]` ← `1` : mise à 1 du bit EXL du Status Register (SR: `C0.$12`)
   - `CR[XCODE]` ← `8` : la cause d'appel est mise dans le champ XCODE du registre de cause (CR: `C0.$13`)
   - `PC` ← `0x80000180` : adresse d'entrée du noyau
2. Une exception survient toujours à l'exécution d'une instruction en faute
   La majorité des exceptions sont fatales, mais certaines non...
   - `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 (C0.$12)

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



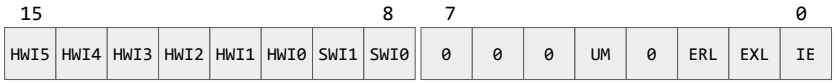
UM — User Mode = 1 en mode USER

ERL — Error Level = 1 au démarrage et quand le MIPS est dans un état incohérent

EXL — Exception Level = 1 à l'entrée dans le noyau après un syscall, une exception et une IRQ

IE — global IRQ enable = 1 quand les interruptions sont acceptées

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

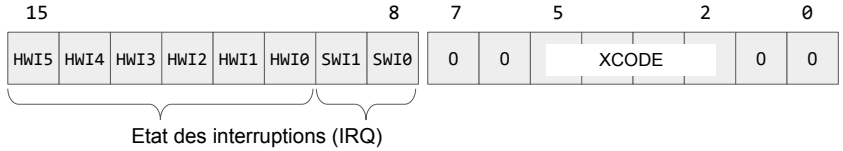
- Si l'un des bits ERL ou EXL est à 1 alors
  le MIPS est en mode KERNEL IRQ masquée quelque-soit l'état de UM et IE

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 (C0.$13)

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 7

`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 : code reset

- 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`

```
/* initializes SR reg. : UM EXL and IE */
li      $26,    0x00000013
mtc0    $26,    $12

/* initializes stack pointer */
la      $29,    seg_stack_base
addiu   $29,    $29,    0x4000  /* 16KB */

/* jumps in user mode function main () */
la      $26,    seg_data_base
lw      $26,    0($26)
mtc0    $26,    $14
eret
```

- 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

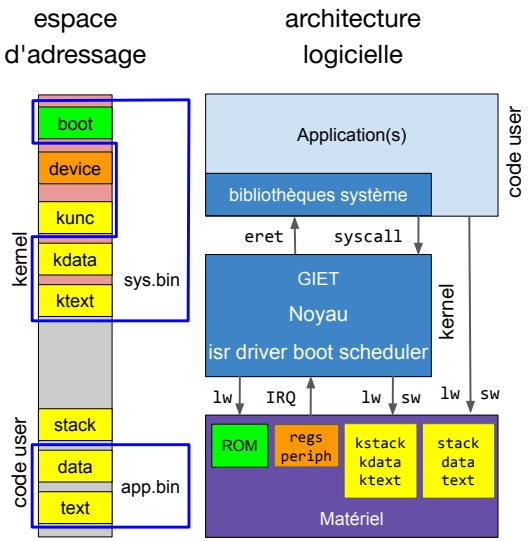
Analyse des fichiers du noyau

```
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
    ├── irq_handler.h
    ├── exc_handler.h
    ├── ctx_handler.h
    └── drivers.h
```

```
USER
├── Makefile
├── main.c     : code application
├── config.h: configuration hardware
├── 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
- S'il y a deux binaires, il y a deux éditions de liens, une pour `sys.bin` et une pour `app.bin`, il y a donc deux ldscript.



Comment le noyau retrouve la fonction main

```c
#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

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

app.ld

Grâce 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

```c
__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);
}
```

```c
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

```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

En résumé, nous avons vu :

- comment les registres de périphériques sont mappés dans l'espace d'adressage du MIPS32 sur la machine utilisée par ce module
- que le MIPS32 boot à l'adresse `0xBFC00000`
- que le système d'exploitation est composé de bibliothèques système compilées avec les applications utilisateurs et d'un noyau
- à quoi sert le système d'exploitation et les garanties qu'il offre
- comment l'application utilisateur communique avec le noyau
- les fonctions de l'unique bibliothèque système disponible dans ce module
- le détail des registres SR (Status), CR (Cause) et EPC (Exception PC)
- le principe de traitement des appels système (le détail sera vu plus tard)
- comment sortir du noyau pour aller dans l'application utilisateur
- que le noyau et l'application utilisateur sont deux binaires indépendants
- le code de boot et comment le noyau connaît l'adresse de la fonction `main()`
- `comment sont produits les deux binaires app.bin et sys.bin`

En TME, vous allez compiler et exécuter une application sur la plateforme

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 ...