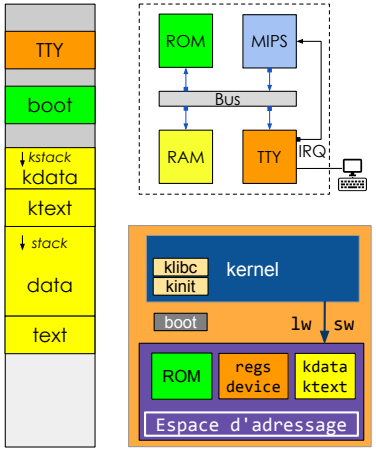
Application en mode user

LU3INx29 Architecture des ordinateurs 1

franck.wajsburt@lip6.fr

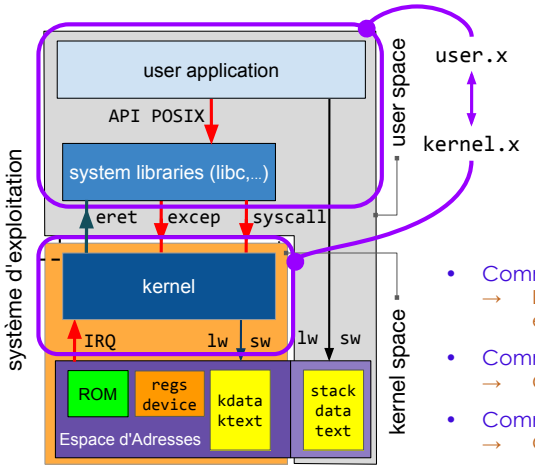
27 nov 2023

Ce que nous avons vu au dernier cours



- Un SoC (System-on-Chip) contient au moins un processeur, une mémoire et un contrôleur de périphérique (ici un terminal TTY)
- Le processeur accède à la mémoire et aux contrôleurs de périphériques par des requêtes de lecture et d'écriture (`lw/sw`) dans son espace d'adressage.
- Le **`kernel`** est la partie du système d'exploitation qui gère l'accès aux ressources matérielles (processeur, mémoire, périphériques) pour les applications.
- Le kernel est composé de plusieurs parties, par ex. **`kinit`** pour le démarrage et **`klibc`** pour les fonctions *standards*.
- Le MIPS démarre à l'adresse **`0xBFC00000`** *mappée en mémoire* où se trouve le **`boot`.**

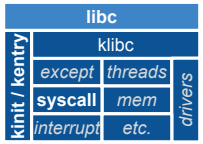
Questions pour cette séance

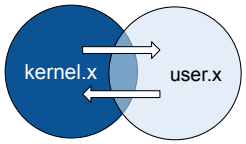


Une application s'exécute au dessus du système d'exploitation (*application + libraries → kernel*)

- Comment le kernel lance-t-il l'application ?
  → La fonction de démarrage de l'application est au début de la section .text …
- Comment l'application utilise-t-elle le kernel ?
  → Grâce aux appels syscall …
- Comment le kernel se protège de l'application ?
  → Grâce au mode d'exécution du MIPS …

Plan de la séance






Modes d'exécution du MIPS


Quelques composants du noyau et de la libc


Communication entre kernel.x et user.x


Visite guidée du code sur un exemple (en annexe)

Modes d'exécution du MIPS

Mode kernel → tous les droits
Mode user → droits restreints

Pourquoi deux modes...

On ne peut pas faire confiance à une application !

- Elle peut casser le matériel en l'utilisant mal
- Elle peut tenter d'accéder à des données ne lui appartenant pas
- Elle peut tenter de modifier le noyau du système d'exploitation

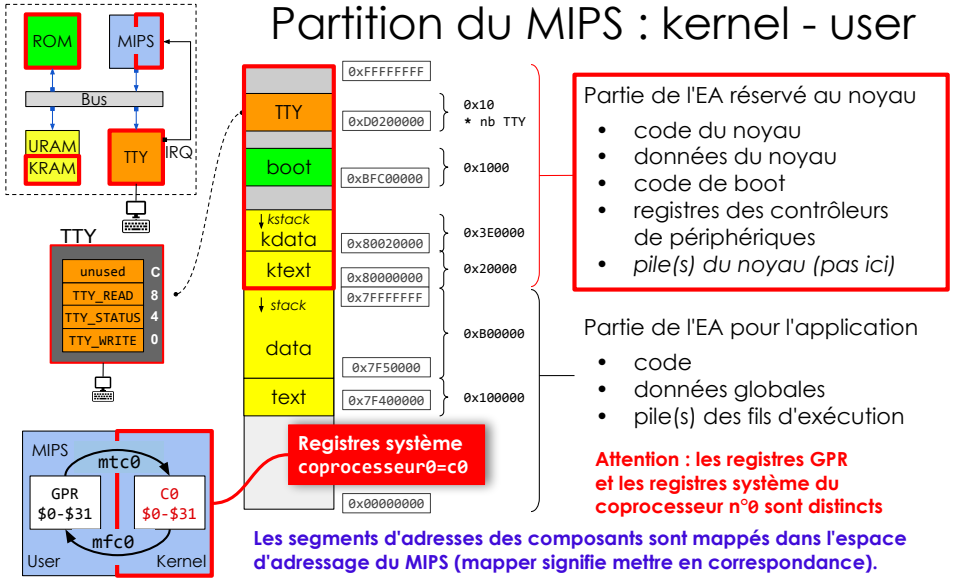
Le MIPS propose donc

- un mode *d'exécution* **user** bridé pour l'application
- un mode **`kernel`** avec tous les droits pour le noyau de l'OS.

Le mode **user** n'accède qu'à une partie de l'espace d'adressage et il ne peut pas exécuter les instructions permettant de changer de mode !

Certains processeurs proposent plus de modes pour restreindre les droits, par exemple un mode pour le code des pilotes de périphériques.

Partition du MIPS : kernel - user



Partie de l'EA réservé au noyau

- code du noyau
- données du noyau
- code de boot
- registres des contrôleurs de périphériques
- *pile(s) du noyau (pas ici)*

Partie de l'EA pour l'application

- code
- données globales
- pile(s) des fils d'exécution

Attention : les registres GPR et les registres système du coprocesseur n°0 sont distincts

Les segments d'adresses des composants sont mappés dans l'espace d'adressage du MIPS (mapper signifie mettre en correspondance).

Espace d'adressage pour l'application user

Pour produire l'application user, l'éditeur de lien a besoin
- d'avoir une description de l'espace d'adressage utilisable en mode user
- de savoir comment remplir les régions de mémoire avec le code et les données.

`ulib/app.ld`

Les régions des applications sont aux adresses < `0x80000000`

```
__text_origin   = 0x7F400000 ;    /* first byte address of user code region */
__text_length   = 0x00100000 ;
__data_origin   = 0x7F500000 ;    /* first byte address of user data region */
__data_length   = 0x00B00000 ;
__data_end      = __data_origin + __data_length  ; /* first addr after user data region */

_start          = __text_origin;  /* address where _start() function is expected */

MEMORY {
    text_region  : ORIGIN = __text_origin,  LENGTH = __text_length
    data_region  : ORIGIN = __data_origin,  LENGTH = __data_length
}

SECTIONS {
    .text : {
        *(.start)    /* with _start() which calls main() expected at beginning of .text */
        *(.text)     /* all others codes */
    } > text_region
    .data : {
        *(.*data*)          /* initialized global variables */
        . = ALIGN(4);       /* move the filling pointer to an word aligned address */
        __bss_origin = .;   /* first byte of uninitialized global variables */
        *(.*bss*)           /* uninitialized global variables */
        . = ALIGN(4);       /* move the filling pointer to an word aligned address */
        __bss_end = .;      /* first byte after the bss section */
    } > data_region
}
```

Déclaration des adresses et des tailles des segments dans l'espace d'adressage. L'application n'a pas besoin de connaître les régions du kernel.

Description des régions de mémoire dans l'espace d'adressage

Description de la manière de remplir les régions de mémoire avec des sections de sorties de l'éditeur de lien contenant les sections d'entrées produites par le compilateur

Notez que le code est dans 2 types de sections .start et .text

Espace d'adressage vu du noyau

kernel/kernel.ld

```
__tty_regs_map   = 0xd0200000 ;     /* tty's registers map */

__boot_origin    = 0xbfc00000 ;     /* first byte address of boot region */
__boot_length    = 0x00001000 ;     /* boot region size */
__ktext_origin   = 0x80000000 ;     /* first byte address of kernel code region */
__ktext_length   = 0x00020000 ;
__kdata_origin   = 0x80020000 ;     /* first byte address of kernel data region */
__kdata_length   = 0x003E0000 ;
__kdata_end      = __kdata_origin + __kdata_length ; /* first addr after kernel data region */
__text_origin    = 0x7F400000 ;     /* first byte address of user code region */
__text_length    = 0x00100000 ;
__data_origin    = 0x7F500000 ;     /* first byte address of user data region */
__data_length    = 0x00800000 ;
__data_end       = __data_origin  + __data_length  ; /* first addr after user data region */

_start           = __text_origin;   /* address where _start() function is expected */

MEMORY {
    boot_region  : ORIGIN = __boot_origin,  LENGTH = __boot_length
    ktext_region : ORIGIN = __ktext_origin, LENGTH = __ktext_length
    kdata_region : ORIGIN = __kdata_origin, LENGTH = __kdata_length
    text_region  : ORIGIN = __text_origin,  LENGTH = __text_length
    data_region  : ORIGIN = __data_origin,  LENGTH = __data_length
}

SECTIONS {
    .boot : {
        *(.boot)            /* boot code in boot region */
    } > boot_region
    .ktext : {
        *(.kentry)          /* kernel's entry code whatever the cause */
        *(.text)            /* code of any object file (except boot) in kernel code region */
    } > ktext_region
    .kdata : {
        *(.*data*)          /* initialized global variables */
        . = ALIGN(4);       /* move the filling pointer to an word aligned address */
        __bss_origin = .;   /* first byte of uninitialized global variables */
        *(.*bss*)           /* uninitialized global variables */
        . = ALIGN(4);       /* move the filling pointer to an word aligned address */
        __bss_end = .;      /* first byte after the bss section */
    } > kdata_region
}
```

Notez qu'on décrit toutes les régions, même si ld ne remplit que les régions >= 0x80000000

Déclaration des adresses et des tailles des segments dans l'espace d'adressage

Ces variables sont utilisées dans ce fichiers mais sont aussi accessibles par le code C

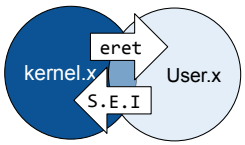
Description des régions de mémoire de l'espace d'adressage

Description de la manière de remplir les régions réservées au noyau (p. ex: `kdata_region`) avec des sections de sorties (p. ex. .kdata) contenant les sections d'entrées produites par le compilateur (p. ex.: `.rodata`, `.sdata`, `.sbss`).

Notez que le code est aussi dans 2 sections distinctes, .kentry et .text

Passage de mode du MIPS

- Le **MIPS démarre en mode kernel** pour initialiser le matériel et les structures de données du noyau et pour charger une application utilisateur.
  Dans notre cas, l'application est déjà en mémoire.
- Puis, le **MIPS passe en mode user en utilisant l'instruction `eret`**
  pour exécuter l'application utilisateur qui commence au début de `.text`
- Ensuite, le **MIPS retourne en mode kernel pour 3 raisons** :
  [S] un appel système après l'exécution de l'instruction **`syscall`**
  [E] une **exception** après l'exécution d'une instruction *erronée*
  [I] une requête d'**interruption** après qu'un composant ait levé une ligne d'interruption (ou **`IRQ`** comme **I**nterruption **Re**ceived **Q**uest)
- Ensuite, le MIPS retourne à l'application … ou pas.



Registres système impliqués
- EPC adresse de retour
- CR registre de cause
- SR registre status

Modes du MIPS

mode USER

- adresses de 0 à 0x7FFFFFFF uniquement les segment légaux
- instruction **`syscall`** pour une demande de service au noyau
- - accès mémoire hors segment ;
  - accès mémoire non alignés ;
  - instructions illégales en mode user;
  - exécution de code erroné ;
  - dépassement de capacité ; etc.

  provoquent l'entrée dans kernel ce sont les **exceptions**, la plupart sont fatales (on n'en revient pas !)
- **IRQ :** interruptions matérielles

mode KERNEL

- Tout l'espace d'adressage
- instruction **`eret`** pour sortir du noyau
- Banc de registres protégés (système)

n°	*nom*	*fonction*
$12	c0_sr	registre d'état
$13	c0_cause	cause d'appel du noyau
$14	c0_epc	adresse de retour ou de l'instruction fautive
$8	c0_bar	adresse malformée
$15	c0_procid	numéro du core
$9	c0_count	compteur de cycles

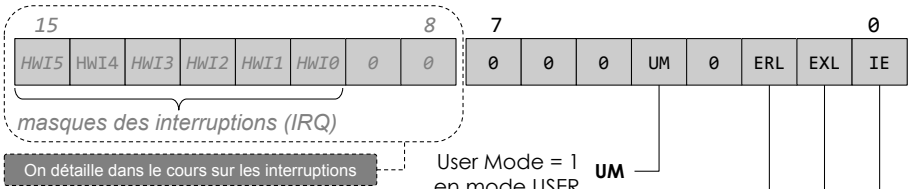
- instructions d'accès aux registres protégés

```
mtc0   $GPR, $c0      movetoc0
mfc0   $GPR, $c0      movefromc0
```

Attention : il y a 2 bancs de registres distincts
(**GPR** et Coprocessor 0 (**c0**))

Status Register : `c0_sr` (`$12` du copro 0)

`c0_sr` contient le mode d'exécution *et les masques des lignes d'interruption*



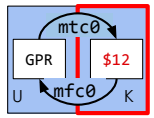
On détaille dans le cours sur les interruptions

User Mode = 1 en mode USER — **UM**

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

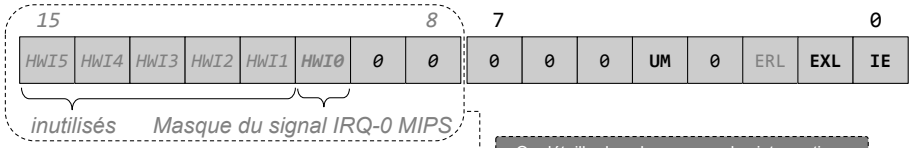
Exception Level = 1 à l'entrée dans le noyau après syscall, une exception et IRQ — **EXL**

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




```
mtc0 $GPR, $12
mfc0 $GPR, $12
```

Comportement du registre `c0_sr` (`$12`)



Comportement du MIPS

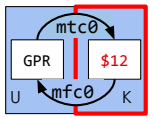
- Si `UM` est à `1`: le MIPS est en mode USER (donc si `UM` est à `0` c'est le mode kernel)
- Si `IE` est à `1`: le MIPS autorise les IRQ à interrompre le programme courant

SAUF si les bits `ERL` ou `EXL` sont à 1, en effet

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

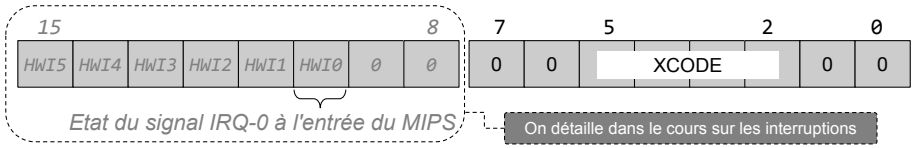
Valeurs typiques de `c0_sr` pour la plateforme

- Lors de l'exécution d'une application USER → `0x0411`
- À l'entrée dans le noyau → `0x0413`
- Pendant l'exécution d'un syscall → `0x0401`



Cause Register : `c0_cause` (`$13` du copro `0`)

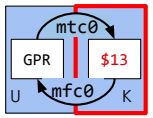
Le registre CR contient la cause d'entrée dans le noyau (après syscall, except ou irq)



Valeurs de XCODE effectivement utilisés dans cette version du MIPS

0_{10} = 0000_2	: INT	Interruption
4_{10} = 0100_2	: ADEL	Adresse illégale en lecture
5_{10} = 0101_2	: ADES	Adresse illégale en écriture
6_{10} = 0110_2	: IBE	Bus erreur sur accès instruction
7_{10} = 0111_2	: DBE	Bus erreur sur accès donnée
8_{10} = 1000_2	: SYS	Appel système (SYSCALL)
9_{10} = 1001_2	: BP	Point d'arrêt (BREAK)
10_{10} = 1010_2	: RI	Codop illégal
11_{10} = 1011_2	: CPU	Coprocesseur inaccessible
12_{10} = 1100_2	: OVF	Overflow arithmétique

```
mtc0 $GPR, $13
mfc0 $GPR, $13
```



Entrée et sortie du noyau

`syscall` ou exception ou interruption

`c0_sr.EXL` ← `1`
mise à 1 du bit EXL du registre Status Register donc passage en mode kernel sans interruption

`c0_cause.XCODE` ← numéro de cause
par exemple `8` si la cause est syscall

`EPC` ← `PC` ou `PC+4`
`PC` adresse de l'instruction courante pour syscall et exception
`PC+4` adresse suivante pour interruption

`PC` ← `0x80000180`
C'est là que se trouve l'entrée du noyau toute cause confondue [syscall, except, IRQ]

`eret`

`c0_sr.EXL` ← `0`
mise à 0 du bit EXL du registre Status Register donc passage en mode `c0_sr.UM` et avec interruption ou pas suivant `c0_sr.IE`
`c0_sr.UM = 1 ⇒ mode user`
`c0_sr.IE = 1 ⇒ int autorisées`

`PC` ← `EPC`
désigne l'adresse de la prochaine instruction à exécuter

Les registres du coprocesseur 0 (`c0`) (dits registres système) sont en rouge

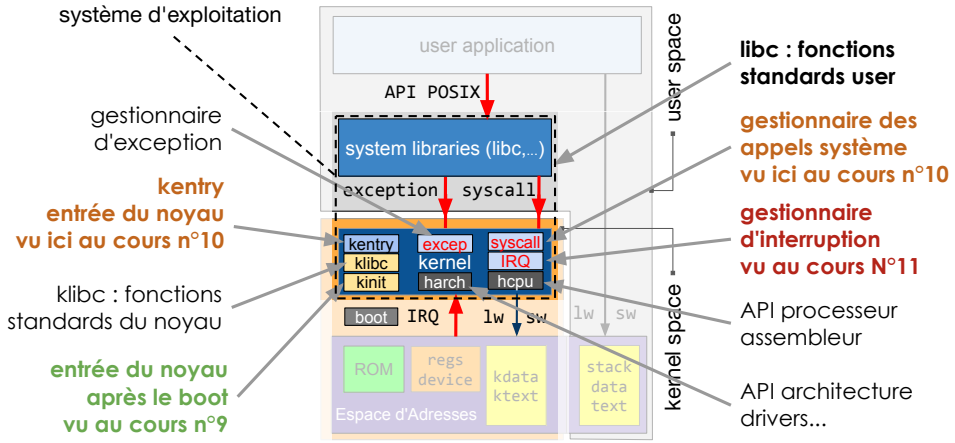
Ce qu'il faut retenir

- Le MIPS propose **2 modes d'exécution** :
  - un mode kernel avec tous les droits et
  - un mode user avec des droits restreints.
- Dans le mode **kernel**, le programme peut accéder
  - aux registres système (du **`Coprocessor 0`**) via les instructions **`mtc0`** et **`mfc0`**
  - à tout l'espace d'**adressage de `0` à `0xFFFFFFFF`**
- Dans le mode **user**, le programme ne peut accéder
  - qu'à la moitié de l'espace d'adressage **(adresses < `0x80000000`)**
  - ne peut pas utiliser les instructions **`mtc0` et `mfc0`**, une tentative produit une **exception**
- Le MIPS **démarre en mode kernel** et saute dans le mode user avec l'instruction **`eret`**
- Le noyau est appelé pour 3 raisons :
  - exécution de l'instruction **`syscall`**
  - une **exception** due à une erreur du programme (div par 0, violation, etc.)
  - une **interruption** demandée par un contrôleur de périphérique
- Les registres système du coprocesseur `0` pour la gestion des appels du noyau sont :
  - `c0_sr` (`$12`) mode d'exécution et masques d'interruption
  - `c0_cause` (`$13`) cause d'appel du noyau défini dans le champ `XCODE`
  - `c0_epc` (`$14`) adresse de l'instruction retour ou de l'instruction fautive

Composants du système d'exploitation

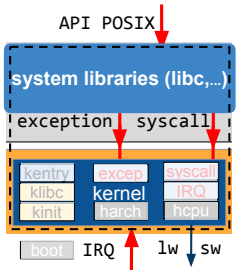
- Présentation des composants de ce système d'exploitation
- Bibliothèque de fonctions standards utilisateur : `libc`
- Le point d'entrée pour toutes les causes : `kentry`
- L'un des 3 gestionnaires : le gestionnaire de `syscall`

Composants du système d'exploitation



Bibliothèque standard de l'utilisateur : libc

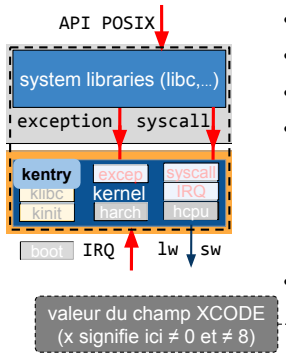
La libc est dans le système d'exploitation, mais elle n'est pas dans le noyau, elle implémente l'API POSIX (ici un *pseudo-POSIX*).



- Les fonctions de la libc font appel au noyau pour accéder à ses ressources en utilisant une fonction d'appel du noyau.
- Cette fonction d'appel est propre au processeur, dans le cas du MIPS, la fonction d'appel utilise l'instruction `syscall`
- Si on change de processeur, il faut réécrire cette fonction.
- Les fonctions de la libc sont liées avec l'application et sont donc présentes dans l'exécutable `user.x`
- Ces fonctions sont exécutées en mode user sauf au moment où elle exécute l'instruction `syscall`, à cet instant elles entrent dans le noyau (`kentry`), qui exécute le service demandé par `syscall` en mode kernel et qui revient dans l'application user (sauf pour le `syscall exit`) qui est définitif.

Kentry

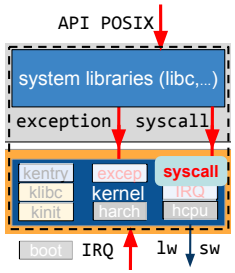
C'est l'unique porte d'entrée normale du noyau
→ *sauf au démarrage où on entre dans le noyau par* `kinit()`



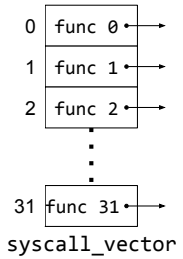
- Le code de `kentry` est à l'adresse `0x80000180`
- Il est nécessairement en assembleur
- Il ne modifie aucun registre GPR sauf `$26` et `$27`
- Il analyse le champ XCODE du registre de `c0_cause` pour savoir quel gestionnaire appeler :
  - (8) gestionnaire de syscall (service demandé par l'app.)
  - (0) gestionnaire d'interruption (service demandé par periph.).
  - (x) gestionnaire d'exception (bug matériel de l'app.)
- Le processeur passe en mode kernel et les interruptions sont masquées (*elles seront ré-autorisés pendant le traitement de certains syscalls*)

gestionnaire de syscall

Gère les appels système de l'utilisateur après le passage par kentry (`XC0DE=8`)



- C'est du code assembleur qui appelle des fonctions
- On entre dans le gestionnaire avec
  - `$2` contenant le numéro du service
  - `$4,$5,$6,$7` contenant les arguments
- Dans le noyau, Il existe un tableau de pointeurs de fonctions dont la case `n°i` contient le pointeur vers la fonction réalisant le service `n°i`
  Ce tableau est nommé **vecteur de syscall**
  - `SYSCALL_NR` : le nombre de services
  - `syscall_vector[SYSCALL_NR]`
- Le gestionnaire fait simplement un appel de fonction
  - `syscall_vector[$2]($4, $5, $6, $7, $2)`
    Ces fonctions ont au plus 5 arguments, mais possiblement moins
- Le noyau ne fait jamais de syscall, il fait de simples appels de fonctions.



Ce qu'il faut retenir

- L'application accède aux services du noyau via des bibliothèques système (libc) qui encapsulent les appels système (`syscall`).
- Depuis le mode user, l'entrée dans le noyau est `kentry` à l'adresse `0x80000180` quelque soit la cause d'appel (syscall, exception et interruption)
- `kentry` analyse la cause d'appel (avec le champ `c0_cause.XCODE`) puis aiguille vers le bon gestionnaire pour son traitement (syscall, exception ou interruption)
- Le gestionnaire de syscall utilise le numéro de service reçu dans le registre $2 comme index dans un tableau de pointeurs sur de fonctions (tableau nommé vecteur de syscall) puis appelle la fonction concernée avec les arguments reçu dans les registres $4 à $7.

Dans le cas général, le noyau contient d'autres sous-systèmes pour la gestion des fichiers, de la mémoire, des communications réseau, des threads, des processus, etc.

Passages entre kernel.x et user.x

- Démarrage app. passage initial — kernel → user
- Demande de service — user → kernel
- Retour à l'app. à la fin du service — kernel → user

Remarque : cette partie du cours décrit plusieurs fois les interactions entre kernel et user, dans l'espoir d'être plus clair, d'abord les principes puis les détails

passage kernel → user

Il y a **2 types** de passage `kernel` → user

1. Au démarrage de l'application et Il y a 2 problèmes à résoudre (1.1 et 1.2) :
   - 1.1 **Il faut connaître l'adresse de la première instruction de l'application**
     → par convention ce sera la première adresse de la section `.text`
   - 1.2 **On ne peut pas appeler la fonction `main()` directement**
     → `main()` ne peut **pas** être **la première fonction** appelée parce qu'**il y a des choses à faire avant**
     → par convention la première fonction d'une application est nommée `_start()`
     — `_start()` **initialise** toutes les **variables** globales **non initialisées** (segment BSS*)
     — `_start()` **appelle** la fonction `main()`
     — `_start()` **appelle** la fonction `exit()` si `main()` ne l'a pas fait
     `_start()` est placée dans une section nommée propre `.start` que le ldscript place en début de section `.text`
     Tout le code de démarrage d'une application dont la fonction `_start()` est placé par convention dans un fichier nommé `crt0.c`
     "crt" → "C runtime", 0 → "the very beginning"
2. Au retour d'un syscall (*ou d'une exception ou d'une interruption*)
   - Il n'y a pas de problème, l'adresse de saut est dans `EPC` (nous allons le voir après)

passage user → kernel

Depuis l'app. ; il y a **3 causes** d'appel du kernel : **syscall**, **exception** et **interruption**.

syscall : la convention d'appel vous est déjà connue (vue pour MARS)

- **$2** contient un numéro de service (numéro commun aux kernel et user)
- **$4, $5, $6, $7** contiennent les arguments
- au retour **$2** contient la valeur de retour (en général 0 si tout va bien)
- seuls les registres GPR persistants ($16 à $23) sont garantis inchangés
- syscall se comporte presque comme un appel de fonction, la différence est que l'appelant de `syscall` ne réserve pas de place dans la pile pour les arguments ($4 à $7) *
- instruction **`syscall`** fait un appel de fonction **`syscall_vector[$2]($4,$5,$6,$7,$2)`** on rappelle que **`syscall_vector[]`** est un tableau de pointeurs sur des fonctions du noyau

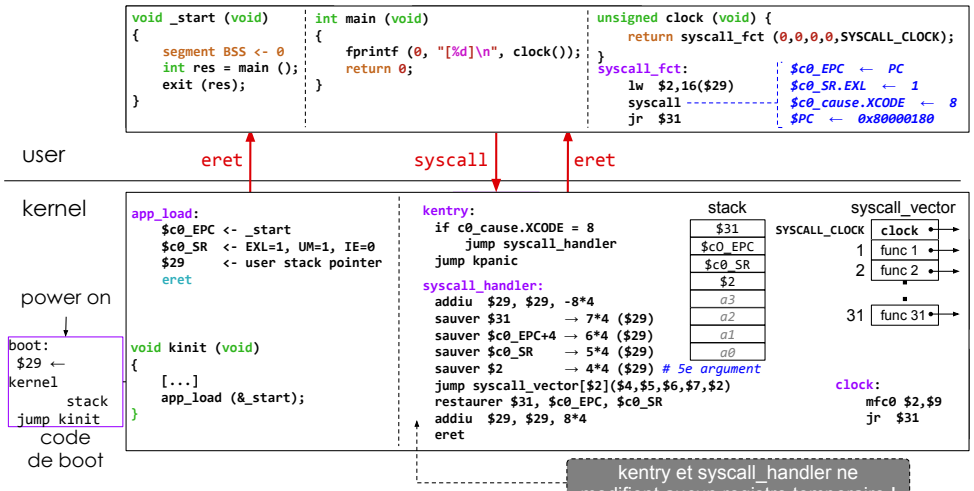
exceptions et **interruptions**

- Une exception est une faute du programme, dans notre cas, elles sont fatales, mais parfois elles ne le sont pas et on revient dans l'application. Ici, on affiche les registres et on se bloque.
- Les interruptions sont demandées par les périphériques, elles s'insèrent entre deux instructions. Dans les deux cas, tous les registres sont conservés intacts, l'interruption a juste « volé » du temps à l'application courante. Nous verrons ça en détail au prochain cours.

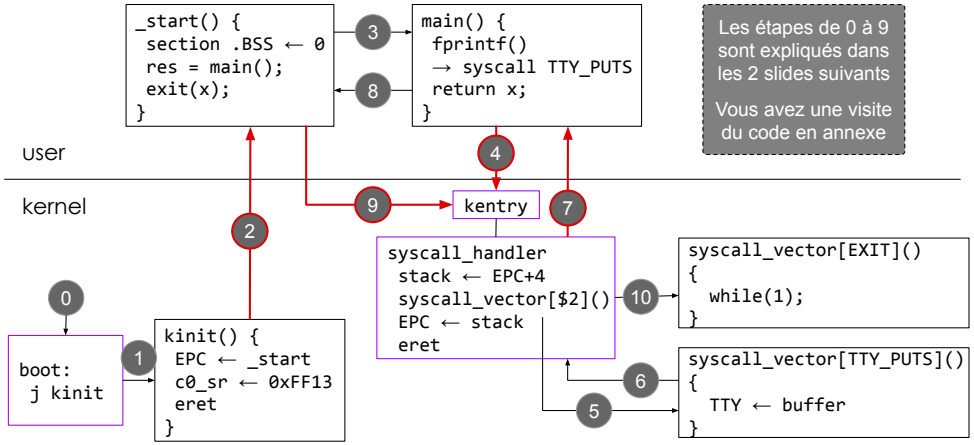
Dans tous les cas, le MIPS saute à l'adresse **`0x80000180`** avec la cause dans **`c0_cause`**

** Il y a une bonne raison mais on le verra en archi-2 :-))*

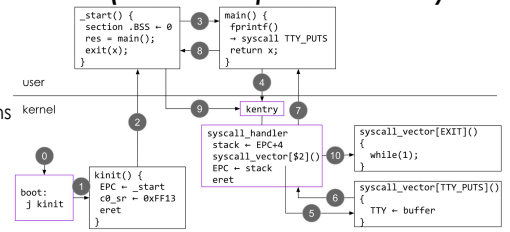
Un parcours de boot à syscall *(big picture)*



Un parcours de boot à exit *(en 10 étapes)*



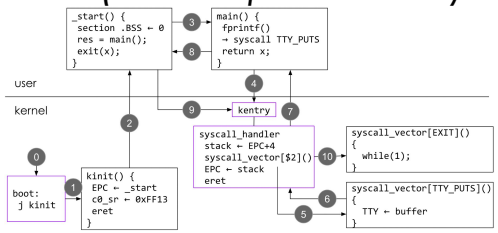
Un parcours de boot à exit *(les étapes 0 à 4)*



0. Après l'activation du signal reset, le MIPS saute à l'adresse de boot **`0xBFC00000`**, le MIPS est en mode kernel, les interruptions sont masquées (le bit **`c0_sr.ERL`** est à 1).
1. Le code de boot se contente d'initialiser le pointeur de pile en haut de la section **`.kdata`** puis il appelle la fonction **`kinit()`**
2. Démarrage de l'application avec la fonction **`_start()`**, cette fonction prépare la mémoire utilisateur en initialisant les variables globales non initialisées par le programme lui-même (elles sont dans la section .BSS).
3. Appel de la fonction **`main()`**, c'est la fonction principale de l'application (elle devrait recevoir des arguments de la ligne de commande, ici il n'y en a pas). La fonction **`main()`** peut demander l'arrêt de l'application par l'appel à la fonction **`exit()`** ou juste sortir par **`return x`**, et laisser **`_start()`** faire l'appel à **`exit()`**
4. L'exécution de **`fprintf()`** définie dans la libc provoque l'exécution d'une instruction **`syscall`** qui déroute l'exécution de l'application vers l'adresse **`kentry`**, le point d'entrée unique du noyau (hormis **`kinit()`**).

Un parcours de boot à exit *(les étapes 5 à 10)*

5. `kentry` a décodé le registre de cause et fait appel au gestionnaire de `syscall` (`syscall_handler`) qui sauvegarde dans la pile les valeurs de registres lui permettant de revenir de l'appel système (dont `EPC+4`) et elle appelle la fonction présente dans la table `syscall_vector[]` à la case du n° de service



6. La fonction `syscall_vector[SYSCALL_TTY_PUTS]()` envoie les octets du buffer dans le registre `WRITE` du TTY
7. Au retour de la fonction précédente, on revient dans le gestionnaire de `syscall` qui rétablit la valeur des registres sauvegardés dans la pile et qui prépare le registre EPC pour l'exécution de l'instruction `eret` qui revient dans la fonction `main()`
8. L'exécution de return permet de sortir de la fonction `main()` pour revenir dans la fonction `_start()`. L'application est terminée, il faut appeler `exit()`
9. La fonction `exit()` exécute l'instruction `syscall` qui saute dans `kentry` comme à l'étape 4.
10. Comme à l'étape 6, le gestionnaire de `syscall` appelle cette fois la fonction `syscall_vector[SYSCALL_EXIT]()` qui, ici, se contente d'arrêter l'exécution.

boot → kinit → app_load → _start → main

Kernel | User

kernel/hcpua.S
```
.section .boot,"ax"
boot:
    la      $29,    __kdata_end  # init kernel stack pointer
1   la      $26,    kinit        # goto kinit
    jr      $26
```

kernel/kinit.c
```
#include <klibc.h>
static char banner[] =      "    _      ___     __ \n"
                            "  | |__ /'v'\  / / \n"
                            "  | / /(     )/ _ \\n"
                            "  |_\_\ x___x \___/\n\n";
void kinit (void) {
    kprintf (0, banner);
    extern int __bss_origin;     // init kernel bss section
    extern int __bss_end;        // kernel global variables
    for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);
2   extern int _start;
    app_load (&_start);          // load & launch user app.
}
```

kernel/hcpua.S
```
.globl app_load
app_load:
    mtc0    $4,     $14          # c0_EPC ← _start address
    li      $26,    0x12         # c0_SR ←  0x12, thus
    mtc0    $26,    $12          # UM ← 1, EXL ← 1, IE ← 0
3   la      $29,    __data_end   # init user stack pointer
    eret                         # PC ← c0_EPC & EXL ← 0
```

ulib/crt0.c
```
#include <libc.h>

__attribute__ ((section (".start")))
void _start (void) {
    int res;

    extern int __bss_origin;     // init user bss section
    extern int __bss_end;        // user global variables
    for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);

    extern int main (void);
4   res = main ();               // call main then at return
    exit (res);                  // exit if main didn't do it
}
```

uapp/main.c
```
#include <libc.h>

int main (void) {
5   fprintf (0, "[%d] app is alive\n", clock());
    return 0;
}
```

Ces codes et ces données sont placés dans l'espace d'adressage du MIPS tel que c'est demandé par les fichiers ldscript : `kernel.ld` et `user.ld`

** Une annexe contient une visite détaillée du code*

clock → syscall_fct → kentry → syscall_handler → clock

User | Kernel

ulib/libc.c
```
#include <syscalls.h> // numéro de syscall
// la fonction user clock() appelle la fonct. syscall_fct()
// qui va utiliser l'instruction syscall du MIPS
unsigned clock (void) {
1   return syscall_fct (0, 0, 0, 0, SYSCALL_CLOCK);
}
```

ulib/crt0.c
```
#include <libc.h>

// int syscall_fct(int a0, int a1, int a2, int a3, int code)
__asm__ (".globl syscall_fct \n" // les arguments sont dans
         "syscall_fct:       \n" // les registres $4 à $7
2        "   lw  $2,16($29)  \n" // le code est dans la pile
         "   syscall         \n" // au retour $2 est le res.
         "   jr  $31         \n"); // sortir
```

kernel/hcpua.S
```
// c0_cause.XCODE contient 8 (car syscall)
// c0_EPC contient l'adresse de l'instruction syscall
// c0_SR.EXL est à 1 → mode kernel avec IRQ masquées
kentry:
    mfc0    $26,    $13          // $26 ← c0_CAUSE
3   andi    $26,    $26,    0x3C // $26 ← XCODE * 4
    li      $27,    0x20         // $27 ← 8 * 4 (syscall)
    bne     $26,    $27,    kpanic // Si pas syscall → kpanic
syscall_handler:
   [...]    // code du gestionnaire de syscall
```

kernel/hcpua.S
```
syscall_handler:
    addiu   $29,    $29,    -8*4 // alloc contexte
    mfc0    $27,    $14          // c0_EPC
    mfc0    $26,    $12          // c0_SR
    addiu   $27,    $27,    4    // c0_adr de retour
    sw      $31,    7*4($29)     // car jalr
    sw      $27,    6*4($29)     // sw c0_EPC
    sw      $26,    5*4($29)     // sw c0_SR
    sw      $2,     4*4($29)     // n° service
    mtc0    $0,     $12          // UM ← 0 IE ← 0
    la      $26,    syscall_vector // adr
    andi    $2,     $2,     SYSCALL_NR-1
    sll     $2,     $2,     2    // * 4
    addu    $2,     $26,    $2   // adr in
    lw      $2,     ($2)         // clock() adr
4   jalr    $2                   // call clock()
    lw      $26,    5*4($29)     // c0_SR
    lw      $27,    6*4($29)     // c0_EPC
    lw      $31,    7*4($29)     // $31
    mtc0    $26,    $12          // set c0_SR (EXL==1)
    mtc0    $27,    $14          // set c0_EPC adr retour
    addiu   $29,    $29,    8*4  // free contexte
    eret                         // PC←c0_EPC & c0_SR.EXL←0
```

stack: $31, $c0_EPC, $c0_SR, $2, $a3, $a2, $a1, $a0

syscall_vector: SYSCALL_CLOCK → clock (1), func 1 (2), func 2, ..., func 31

kernel/hcpua.S
```
.globl clock
clock:
5   mfc0    $2, $9  // c0_s9 contient un compteur de cycles
    jr      $31     // c'est déjà fini
```


placement code et data : `kernel/kernel.ld`

```
__tty_regs_map   = 0xd0200000 ;    /* tty's registers map */

__boot_origin    = 0xbfc00000 ;    /* first byte address of boot region */
__boot_length    = 0x00001000 ;    /* boot region size */
__ktext_origin   = 0x80000000 ;    /* first byte address of kernel code region */
__ktext_length   = 0x00020000 ;
__kdata_origin   = 0x80020000 ;    /* first byte address of kernel data region */
__kdata_length   = 0x0031E0000 ;
__kdata_end      = __kdata_origin + __kdata_length ; /* first addr after kernel data region */
__text_origin    = 0x7F400000 ;    /* first byte address of user code region */
__text_length    = 0x00100000 ;
__data_origin    = 0x7F500000 ;    /* first byte address of user data region */
__data_length    = 0x00800000 ;
__data_end       = __data_origin  + __data_length  ; /* first addr after user data region */

_start           = __text_origin;  /* address where _start() function is expected */

MEMORY {
    boot_region  : ORIGIN = __boot_origin,  LENGTH = __boot_length
    ktext_region : ORIGIN = __ktext_origin, LENGTH = __ktext_length
    kdata_region : ORIGIN = __kdata_origin, LENGTH = __kdata_length
    text_region  : ORIGIN = __text_origin,  LENGTH = __text_length
    data_region  : ORIGIN = __data_origin,  LENGTH = __data_length
}

SECTIONS {
    .boot : {
        *(.boot)              /* boot code in boot region */
    } > boot_region
    .ktext : {
        *(.kentry)            /* kernel's entry code whatever the cause */
        *(.text)              /* code of any object file (except boot) in kernel code region */
    } > ktext_region
    .kdata : {
        *(.*data*)            /* initialized global variables */
        . = ALIGN(4);         /* move the filling pointer to an word aligned address */
        __bss_origin = . ;    /* first byte of uninitialized global variables */
        *(.*bss*)             /* uninitialized global variables */
        . = ALIGN(4);         /* move the filling pointer to an word aligned address */
        __bss_end = . ;       /* first byte after the bss section */
    } > kdata_region
}
```

C'est dans **`kernel.ld`**, utilisé par l'éditeur de lien lors de la création du kernel, qu'est définie l'adresse de la fonction `_start`. On choisit l'adresse du début de la section `.text` (qui contiendra le code de l'app.)

c'est la convention d'appel de l'application.

Il faut absolument que l'application respecte ce choix, ce sera possible grâce au ldscript `user.ld` utilisé par l'éditeur de lien pour créer l'application.

ulib/crt0.c
```
void kinit (void) {
    [...]
    extern int __bss_origin;     // init kernel bss section
    extern int __bss_end;        // kernel global variables
    for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);
    extern int _start;
    app_load (&_start);          // load & launch user app.
}
```

Les adresses `__bss_origin` et `__bss_end` définissent respectivement la première et la dernière adresse dans le segment d'adresse `.kdata` où sont mises les variables globales non initialisées par le kernel.

placement code et data : `ulib/user.ld`

C'est par le fichier ldscript `user.ld` que le programmeur peut imposer l'adresse de la fonction `_start`

ulib/user.ld

```
__text_origin     = 0x7F400000 ; /* first byte address of user code region */
__text_length     = 0x00100000 ;
__data_origin     = 0x7F500000 ; /* first byte address of user data region */
__data_length     = 0x00B00000 ;
__data_end        = __data_origin  + __data_length  ; /* first addr after user data region */
MEMORY {
    text_region   : ORIGIN = __text_origin,
                    LENGTH = __text_length
    data_region   : ORIGIN = __data_origin,
                    LENGTH = __data_length
}
SECTIONS {
    .text    : {
        *(.start);          /* c runtime at the beginning thow to launch main() */
        *(.text)            /* all others codes */
    } > text_region
    .data : {
        *(.*data*)          /* initialized global variables */
        . = ALIGN(4);       /* move the filling pointer to an word aligned address */
        __bss_origin = .; /* first byte of uninitialized global variables */
        *(.*bss*)           /* uninitialized global variables */
        . = ALIGN(4);       /* move the filling pointer to an word aligned address */
        __bss_end = .;      /* first byte after the bss section */
    } > data_region
}
```

ulib/crt0.c

```
__attribute__ ((section (".start")))
void _start (void) { // appelée par kinit
    int res;
    for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);
    res = main ();
    exit (res);
}
```

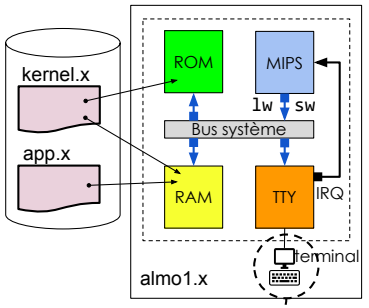
mise à 0 des variables globales non explicitement initialisées par le programme

`__bss_origin` et `__bss_end` sont déclarées `extern` dans le même fichier `ulib/crt0.c` mais on ne l'a pas fait apparaître ici

Ce qu'il faut retenir

- Le noyau doit connaître l'adresse du début de l'application nommée `_start()` placée au début du segment de `.text` (code user)
- C'est en plaçant cette fonction dans une section spécifique `.start` que l'éditeur de lien peut imposer le placement de `_start()`
- La fonction `_start()` initialise les variables globales non initialisées, lance `main()` et appelle `exit()` (pour le cas où `main()` ne l'a pas appelé).
- Les numéros de services `syscall` sont définis dans un fichier commun au noyau et à l'application, ils font partie de l'API du noyau.
- un `syscall` se comporte presque comme un appel de fonction :
  - au maximum 4 arguments dans `$4` à `$7` pour l'utilisateur
  - le n° de service et la valeur de retour dans `$2`,
  - seuls les registres persistants sont garantis inchangés.
  - `User` utilise **`syscall_fct(a0,a1,a2,a3,code)`** avec les bons arguments
    Kernel appelle la fonction **`syscall_vector[$2]($4,$5,$6,$7,$2)`**

Création des binaires

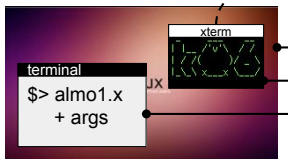


Nous avons deux codes binaires

`kernel.x` contenant le kernel

`app.x` contenant l'application et les fonctions de la libc utilisées par l'application

Chaque binaire est créé par l'éditeur de liens à partir des fichiers objet et d'un fichier `ldscript` décrivant l'espace d'adressage et comment remplir les segments



Écran du système hôte : Linux

terminal du simulateur `almo1.x`

terminal de commande Linux pour lancer `almo1.x`

Conclusion

- Ce que nous avons vu
- Quelles sont les étapes du TME
- Création des binaires

Nous avons vu

- que le MIPS a deux modes d'exécution : **kernel** et **user**
- que le mode user interdit les adresses supérieures à `0x80000000` que la première fonction de l'application est `_start()`
- que le noyau sait où est la fonction `_start()` grâce à une convention (`.start`)
- que `_start()` initialise les variables globales non initialisées avant d'appeler `main()`
- que `_start()` appelle la fonction `exit()` si `main()` ne l'a pas déjà fait
- qu'il y a trois causes d'appel du noyau : **syscall**, **exception** et **interruption**
- que le kernel n'a qu'un seul point d'entrée nommé **kentry** en `0x80000180` ∀ la cause
- que le MIPS dispose de registres système (`c0`) contenant, entre autre, le mode d'exécution dans `c0_sr/$12` et la cause d'appel du noyau dans `c0_cause/$13`
- qu'à l'entrée dans le noyau :
  `c0_EPC` ← PC ou PC+4; `c0_sr.EXL` ← 1 et `c0_cause.XCODE` ← la cause d'appel
- qu'un appel système est semblable à un appel de fonction avec privilèges, mais *l'adresse de retour* est stockée dans le registre système `c0_EPC/$14`
- que l'exécution de l'inst. `syscall` avec `$2` contenant le n°service et `$4` à `$7` les args. appelle la fonction du noyau `syscall_vector[$2]($4,$5,$6,$7,$2)`

Quelles sont les étapes du TME ?

1. Le Kernel seul avec une klibc
   → 1 seul exécutable `kernel.x` mais avec k`printf()`
   → exercice : ajout d'une fonction dans klibc
2. Le kernel et l'application mais tout en mode kernel, l'application a tous les droits
   → 2 exécutables `kernel.x` et `user.x`, appel de la fonction `main()` par le kernel
   → exercice : ajout d'une petite fonction appelée par `main()`
3. Ajout du kentry et des gestionnaires de syscall et d'exception
   → 2 exécutables `kernel.x` et `user.x` fonctionnant dans les bons modes
   `kernel.x` lance `user.x` et `user.x` appelle `kernel.x` pour ses services
   → exercice : ajout d'un nouveau service dans le gestionnaire de syscalls
4. Ajout d'une libc pseudo-POSIX
   → possibilité d'écrire des applications qui ressemblent à des vraies :)
   → exercice : ecriture d'un petit jeu simple