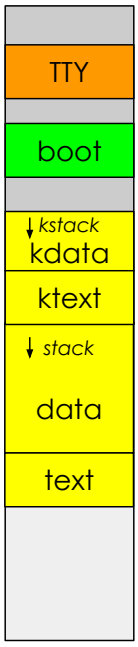
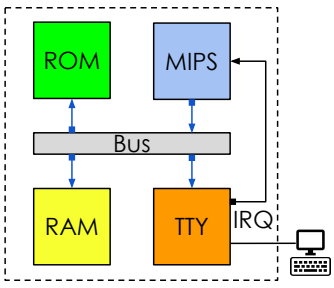
Application en mode user

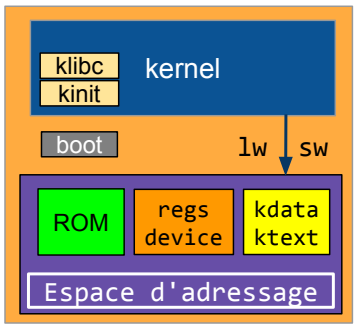
LU3IN029 Archi L3 S5

franck.wajsburt@lip6.fr

Ce que nous avons vu

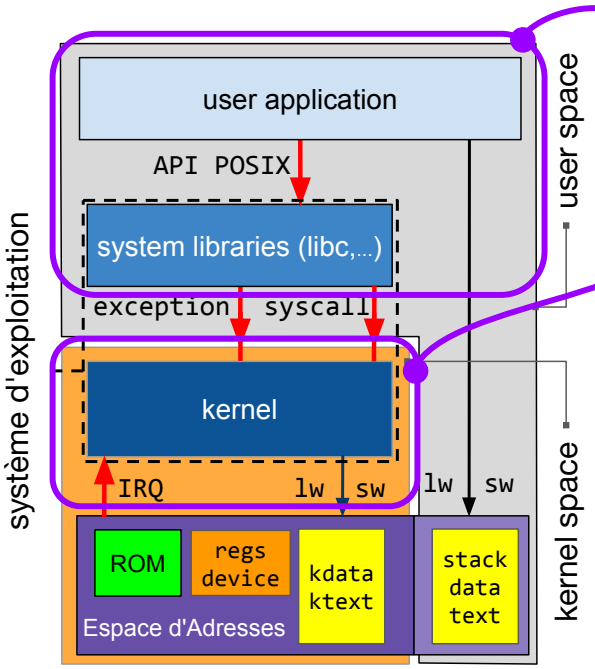






- Un SoC (System-on-Chip) contient au moins un processeur, une mémoire et un contrôleur de périphérique.
- Le processeur accède à la mémoire et aux périphériques par des requêtes de lecture et d'écriture (`lw/sw`) dans son espace d'adressage.
- Le **`kernel`** est la partie du système d'exploitation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

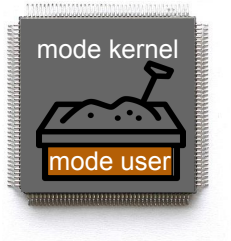
Questions

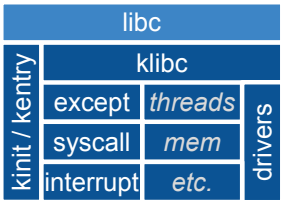


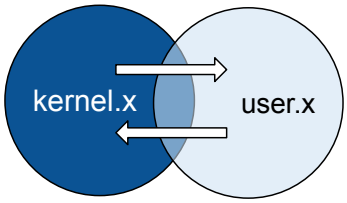
Une application s'exécute sur le système d'exploitation (*kernel* + *libraries*), mais

- Comment le noyau lance-t-il l'application ?
- Comment l'application appelle le noyau ?
- L'application (`user.x`) et le noyau (`kernel.x`) sont 2 exécutables qui communiquent, que doivent-ils avoir en commun ?

Plan







- Modes d'exécution du MIPS
- Composants du noyau et de la libc
- Communication entre kernel.x et user.x
- Visite guidée du code sur un exemple

Modes d'exécution du MIPS

- Mode kernel → tous les droits
- Mode user → droits restreints

Pourquoi deux modes...

On ne peut pas faire confiance à l'application !

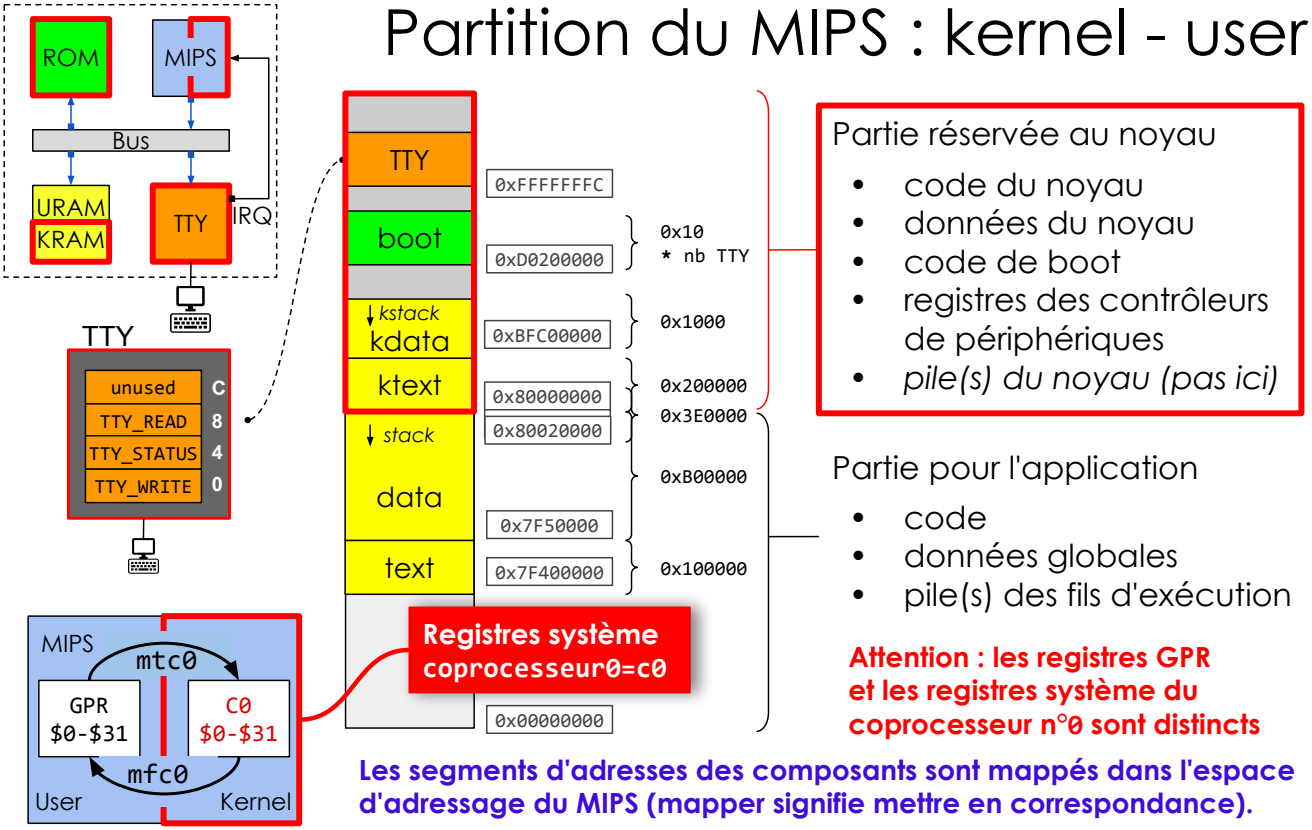
- Elle peut casser le matériel en le contrôlant mal
- Elle peut tenter d'accéder à des données ne lui appartenant pas
- Elle peut tenter de modifier le noyau système d'exploitation

⇒ Le MIPS propose un mode user d'exécution bridé pour l'application et un mode kernel avec tous les droits.

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 !

Sachez que 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 réservée au noyau

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

Partie pour l'application

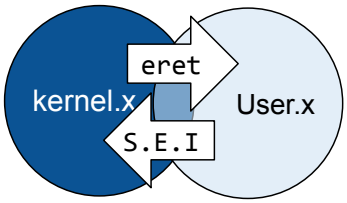
- code
- données globales
- pile(s) des fils d'exécution

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).

Passage de mode

- 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.
- 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 **`R`**e**`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
- les accès mémoire hors segment
  les accès mémoire non alignés
  la division par 0
  l'exécution de code erroné
  le dépassement de capacité
  provoque aussi l'entrée dans kernel
  ce sont les **exceptions**, la plupart
  sont fatales (on 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

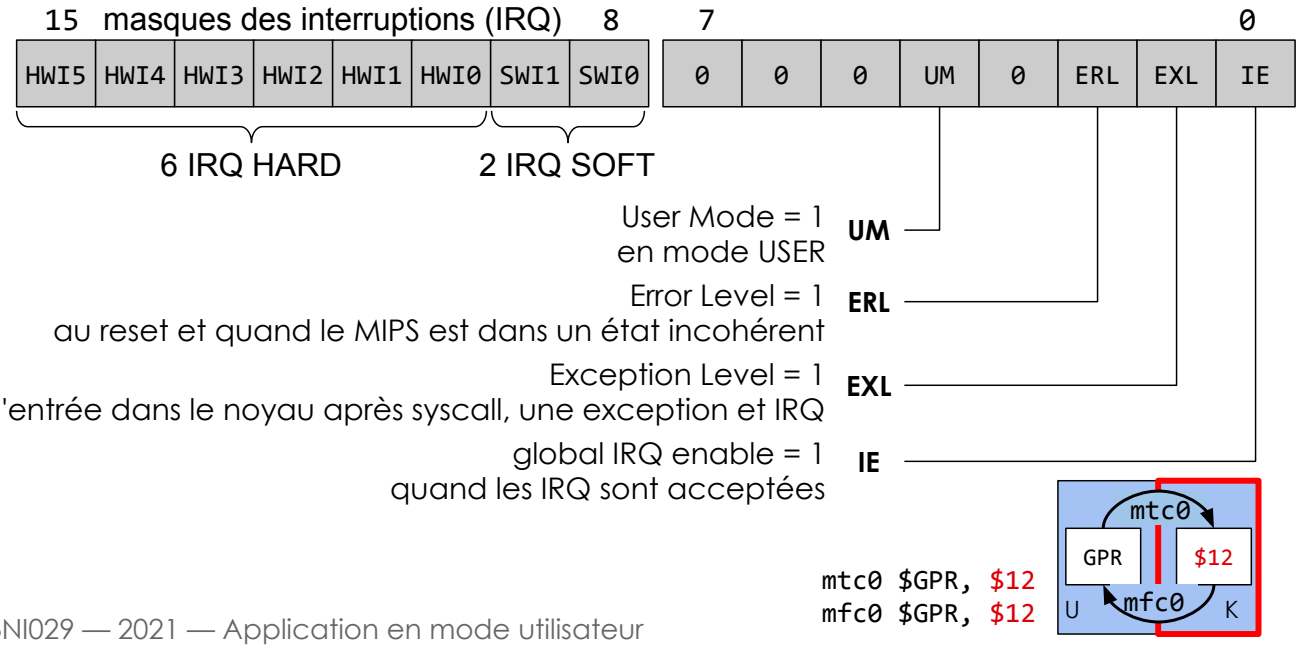
- instructions d'accès aux registres protégés

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

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

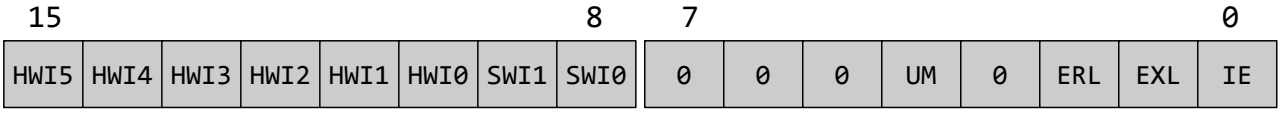
Status Register : `c0_sr` (`$12` du `c0`)

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




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

Comportement du registre c0_sr ($12)



Comportement du MIPS

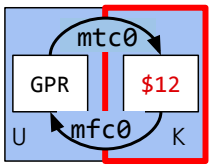
- Si UM est à 1: le MIPS est en mode USER
- Si IE est à 1: le MIPS autorise les IRQ à interrompre le programme courant

SAUF si 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**

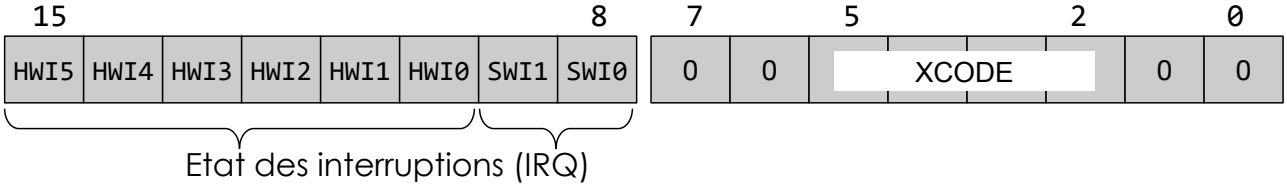
Valeurs typiques de SR

- 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 : c0_cause ($13 du c0)

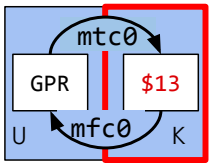
Le registre CR contient la cause d'entrée dans le noyau (après syscall, except ou irq)



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`, exception, 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

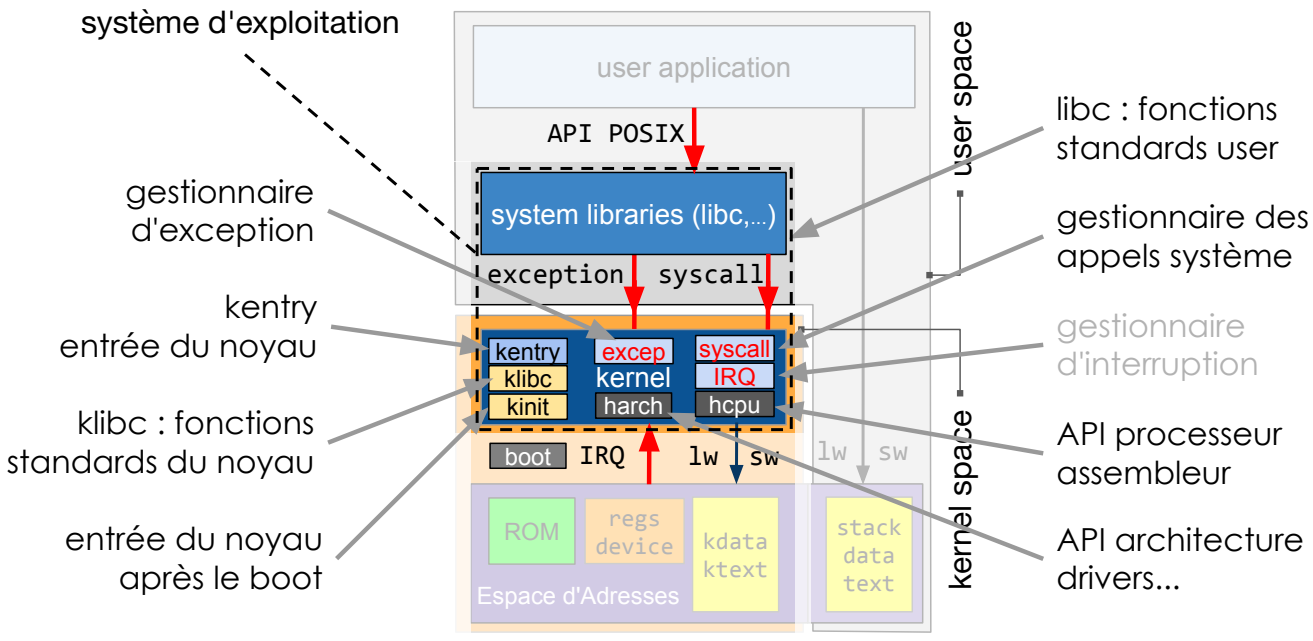
Ce qu'il faut retenir

- 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_cause` (`$12`) cause d'appel du noyau défini dans le champ `XCODE`
  - `c0_sr` (`$13`) mode d'exécution et les masques d'interruption
  - `c0_epc` (`$14`) adresse de l'instruction retour ou de l'instruction fautive

Composants du système d'exploitation

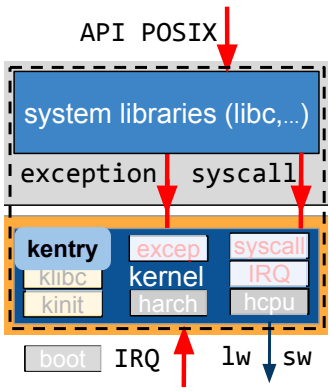
- Le point d'entrée : kentry
- Les gestionnaires d'évènements → syscall, exception et IRQ.
- Abstraction du matériel → cpu (MIPS) et architecture (almo1)
- Bibliothèque de fonctions standards noyau : klibc
- Bibliothèque de fonctions standards utilisateur : libc

Composants du système d'exploitation



Kentry

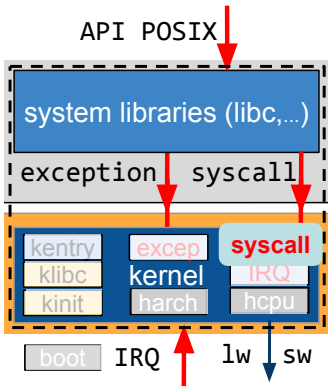
C'est l'unique porte d'entrée du noyau
→ *Sauf au démarrage où l'on entre dans le noyau par* `kinit()`



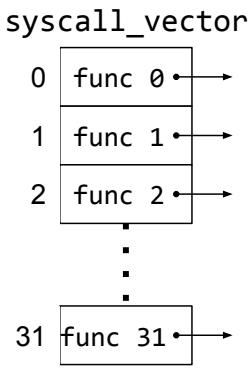
- Le code de `kentry` est à l'adresse `0x80000180`
- Il est nécessairement en assembleur
- Il ne modifie aucun registre sauf `$26` et `$27`
- Il analyse le registre `c0_cause`
  pour savoir quel gestionnaire appeler
  - gestionnaire de syscall
  - gestionnaire d'exception
  - gestionnaire d'interruption
- Le processeur est en mode kernel et les interruptions sont masquées (*elles resteront masquées quel que soit la cause, c'est un choix d'implémentation simplificateur*)

gestionnaire de syscall

Gère les appels système de l'utilisateur après le passage par kentry

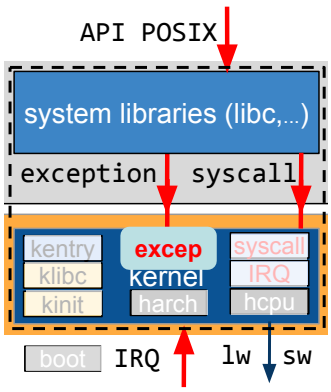


- C'est encore du code assembleur
- On entre dans le gestionnaire avec
  - `$2` contenant le numéro du service
  - `$4,$5,$6,$7` contenant les arguments
- Dans le noyau, Il existe un tableau de pointeurs de fonctions dont la case n°i contient le pointeur vers la fonction réalisant le service n°i
  Ce 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)`
- Le noyau ne fait évidemment jamais de syscall, il fait des appels de fonctions directs



gestionnaire d'exception

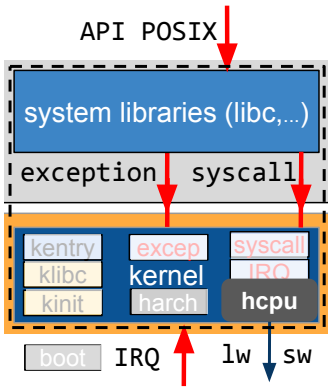
Gère les exceptions provoquées par des erreurs dans l'application,
Ici, elles sont toujours fatales. Il faut juste connaître la raison du problème.



- C'est encore du code assembleur
- Sauve l'état de tous les registres dans un tableau
- Appelle une fonction du noyau (fonction en C)
  qui demande l'affichage des valeurs du tableau
- Puis le programme est normalement déchargé,
  mais là, il se bloque simplement
- Les exceptions ne sont jamais provoquées par
  le noyau (sinon c'est un *kernel panic*)

API processeur : hcpu

Contient kentry et les gestionnaires d'événements vus précédemment
et des fonctions permettant d'accéder à des services spécifiques

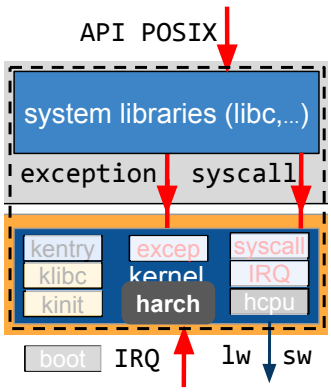


- le numéro du core, dans le cas où il y en a plusieurs
- la valeur du compteur de cycles
- mais aussi, des services que nous verrons plus tard
  - le masquage des interruptions
  - l'accès atomique à la mémoire (read-modif-write)
- Ce module est écrit en c et en assembleur

En cas de changement de processeur
c'est la seule partie du noyau à réécrire

API architecture : harch

Contient l'ensemble des fonctions permettant d'accéder aux contrôleurs de périphériques

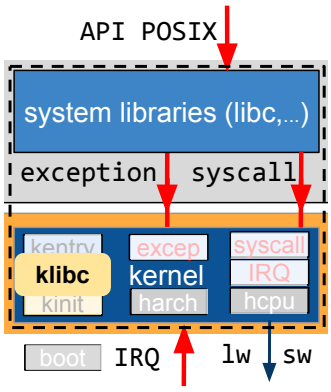


- ici, seulement le contrôleur de TTYs
- mais aussi, des contrôleurs de disques, des timers, des contrôleurs vidéo ou même des contrôleurs de la fréquence du SoC, il existe une multitude de périphériques !

Le noyau définit une API avec des services standards qui doivent être proposés par tous les périphériques

- Ce sont les fonctions : `open()`,`read()`,`write()`,`close()`, etc.
- Ces fonctions sont écrites différemment pour chaque périphérique, il y a un read pour TTY, un pour le disque, etc.
- L'ensemble de ces fonctions forment un pilote (driver)
- L'ajout d'un périphérique dans le SoC entraîne l'ajout d'un pilote dans le noyau.

Bibliothèque standard du noyau : klibc



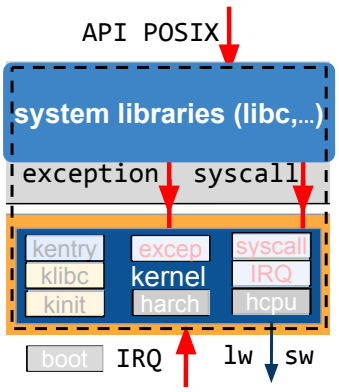
Le noyau rassemble ici toutes les fonctions utiles qui ne sont pas en rapport direct avec la gestion d'une ressource matérielle ou logicielle.

- des fonctions print comme k`printf()`
- déplacement de mémoire comme `memcpy()`
- de gestions de chaîne comme `strlen()`
- et d'autres comme `rand()`

Ces fonctions peuvent utiliser les autres modules
k`printf()` utilise directement `tty_write()` de harch

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 *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 le 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 en mode kernel et qui ressort.

Ce qu'il faut retenir

- L'adresse de démarrage du MIPS est `0xBFC00000`
- Au boot, le MIPS est en mode kernel et saute la fonction `kinit()` du noyau par `jr`
- Depuis le mode user, l'entrée dans le noyau est `kentry` à l'adresse `0x80000180` quelque soit la cause (syscall, exception et interruption)
- `kentry` analyse la cause (avec dans le registre système `c0_cause`) puis aiguille vers le bon gestionnaire pour son traitement (syscall, exception ou interruption)
- Le noyau définit une API vers le processeur et une API vers les périphériques afin de permettre le portage vers d'autres processeurs et d'autres architectures
- Le noyau contient une **k**libc avec quelques fonctions utiles (k`printf`, `memcpy`, etc.)
- L'application accède aux services du noyau via des bibliothèques système (libc) qui encapsule l'appel de `syscall`.

  Dans le cas général, le noyau contient d'autres sous-systèmes pour la gestion des fichiers, de la mémoire, des communications réseau, des threads (fils d'exécution dans un processus) et des processus.

Communication entre kernel.x et user.x

- passage user → kernel
- passage kernel → user

passage user → kernel

Nous avons vu qu'il y a 3 causes : syscall, exception et interruption.
Dans tous les cas, le MIPS saute implicitement à l'adresse `0x80000180`
(notez qu'un saut explicite à cette adresse provoque une exception)

Convention d'appel pour syscall

- `$2` contient un numéro de service (commun kernel / user)
- `$4, $5, $6, $7` contiennent les arguments
- `$2` contient le code de retour (en général 0 si tout va bien)
- seuls les registres GPR persistants (`$16` à `$23`) sont garantis inchangés
- 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
- instruction **`syscall`** → appel de fonction **`syscall_vector[$2]($4,$5,$6,$7,$2)`**
  `syscall_vector[]` est un tableau de pointeurs sur des fonctions dans le noyau

Convention d'appel pour les exceptions et les interruptions

- Dans notre cas, les exceptions sont fatales, mais parfois elles ne le sont pas et on revient dans l'application, nous n'avons pas vu les interruptions, mais disons qu'elles s'insèrent entre deux instructions. Dans les deux cas, tous les registres sont conservés intacts, l'exception ou l'interruption a juste « voler » du temps.

passage kernel → user

Il y a deux types de passage kernel → user

- 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`
- au démarrage de l'application et là Il y a deux problèmes
  1. Quelle est l'adresse de la première instruction de l'application ?
     → par convention ce sera la première adresse de .text
  2. Est-ce qu'on peut appeler directement la fonction `main()` ?
     → non ! `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 `_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` signifiant `"c runtime 0"`

Démarrage de l'application → 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
}
```

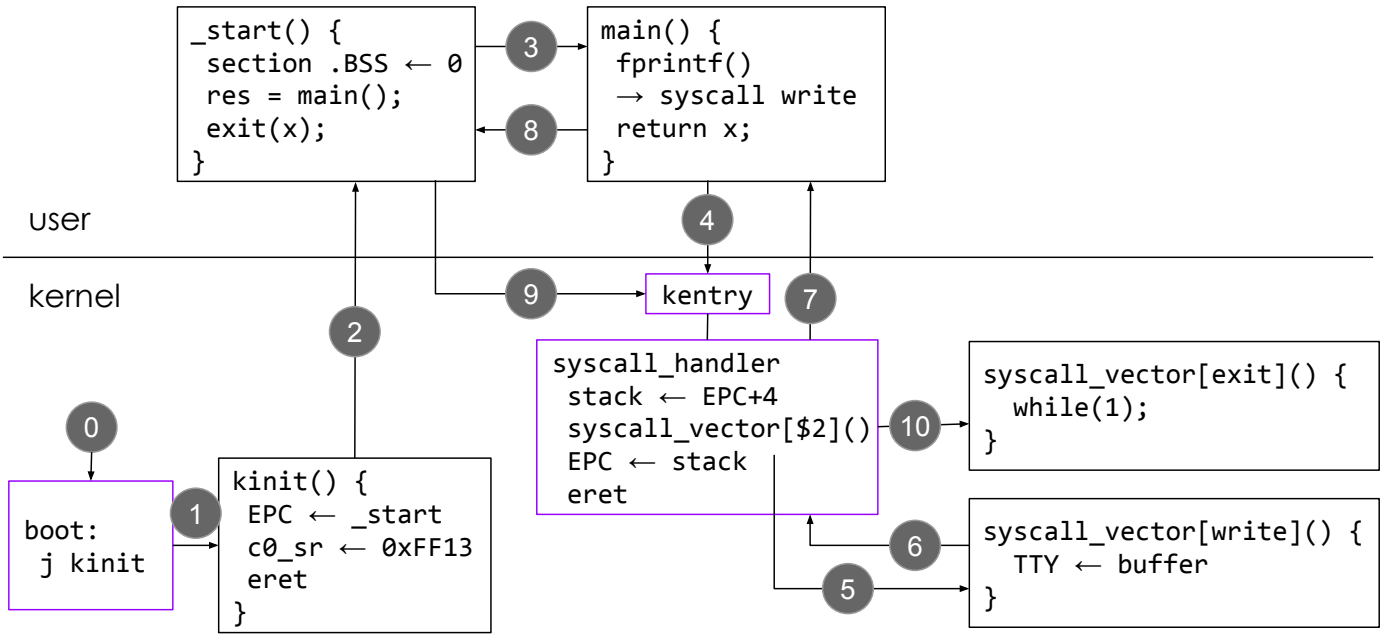
user.ld

crt0.c

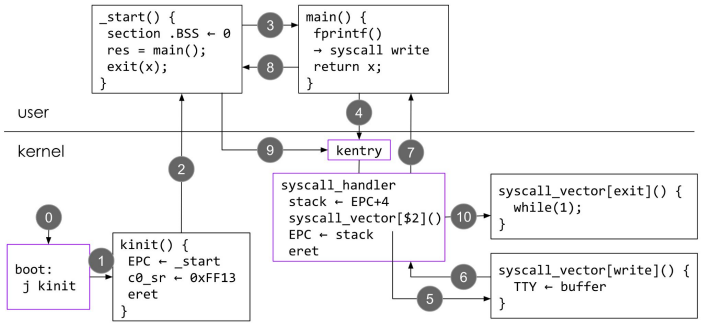
```
__attribute__ ((section (".start")))
void _start (void)
{
    int res;
    for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);
    res = main ();
    exit (res);
}
```

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

Parcours de boot à exit

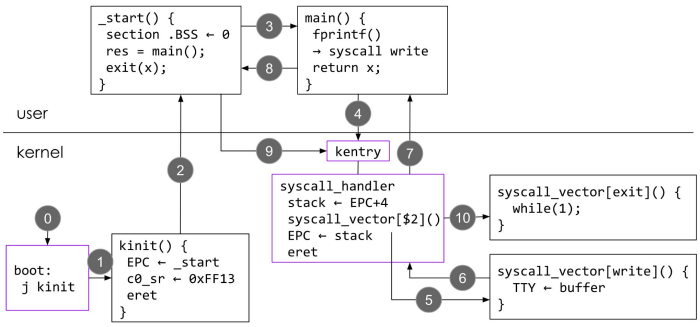


Parcours de boot à exit



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()`**).

Parcours de boot à exit



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[write]()` 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 rétabli 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[exit]()` qui, ici, se contente d'arrêter l'exécution.

Ce qu'il faut retenir

- 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 : syscall` → Kernel : **`syscall_vector[$2]($4,$5,$6,$7,$2)`**
- les interruptions et les exceptions (quand elles reviennent) sont des voleurs de temps du point de vue de l'application en cours.
- 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.

Visite guidée

- Visite du code de la dernière étape
- Les ldscripts
- Les étapes des TP

Ensemble des fichiers de la dernière étape

```
4_libc/
├── Makefile          produit tout par appel récursif (ke
├── kernel
│   ├── harch.h       API de l'archi du SoC donc les drivers
│   ├── harch.c       code des drivers (seulement TTY)
│   ├── hcpu.h        API propre au cpu
│   ├── hcpua.S       boot + kentry + gestionnaires + fct
│   ├── hcpuc.c       fonction en c du CPU comme kdump()
│   ├── kernel.ld           ldscript du kernel
│   ├── kinit.c       fonction d'initialisation du kernel
│   ├── klibc.h       API de le klibc
│   ├── klibc.c       implémentation de klibc
│   ├── ksyscalls.c   syscall_vector[] + qqs syscall
│   └── Makefile      produit le binaire kernel.x
├── common
│   └── syscalls.h    n°services communs K et U
├── ulib
│   ├── crt0.c        fct _start() et syscall
│   ├── libc.h        API de la libc user
│   ├── libc.c        fonctions de la libc user
│   ├── user.ld       ldscript de l'application user
│   └── Makefile      produit les fichiers libc.o, crt0.o
└── uapp
    ├── main.c        code de l'application
    └── Makefile      produit le binaire user.x
```

kernel.x

user.x

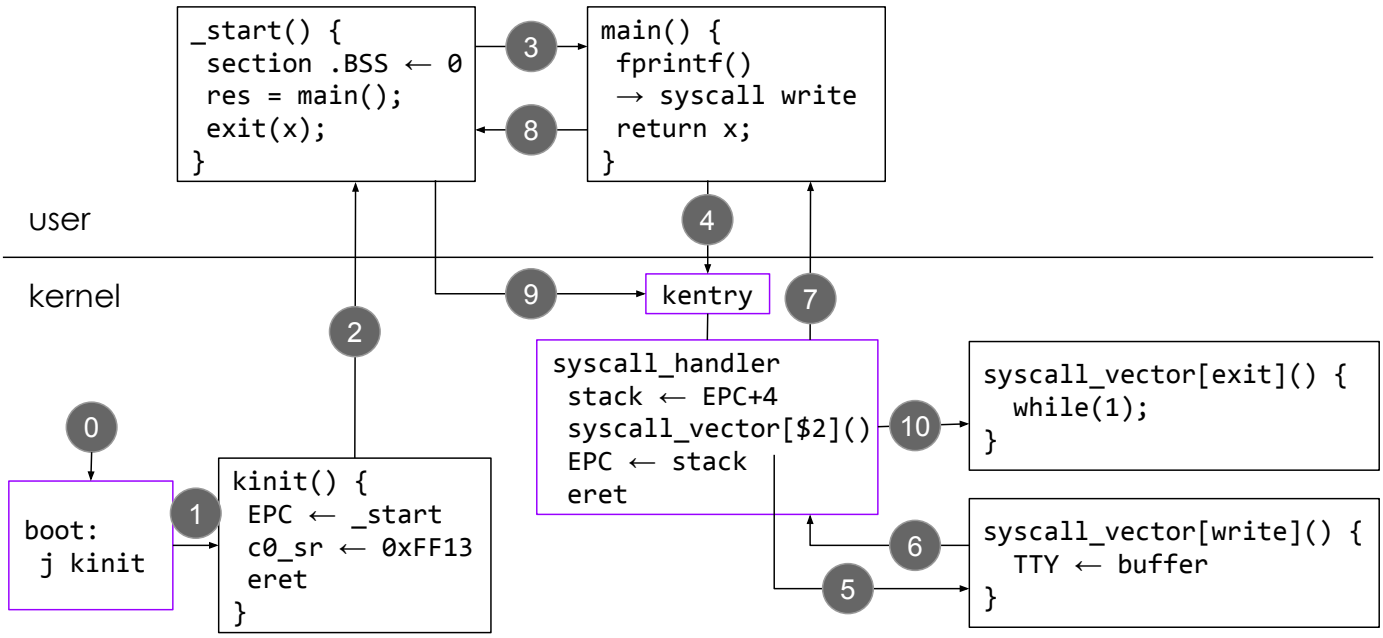
Un but :

réduire le nombre de fichiers en mettant toutes les fonctions d'un même module dans un même fichier

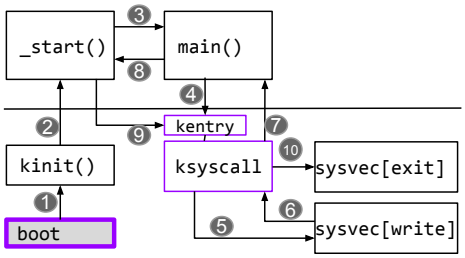
main.c

```c
#include <libc.h>

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

Parcours de boot à exit



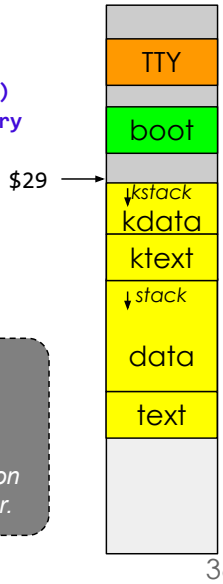
kernel/hcpua.S/boot



La section .boot sera placé par l'éditeur de lien dans le segment d'adresse boot

```
.section .boot,"ax"

boot:
    la      $29,    __kdata_end     // define stack pointer (first address after kdata region)
    la      $26,    kinit           // get address of kinit() function
    jr      $26                     // goto kinit()
```

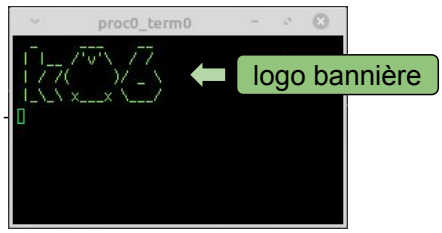
```
// def. of a new section: .boot (see https://bit.ly/3gzKWob)
// flags "ax":     a -> allocated means section put in memory
//                 x -> section contains instructions
```

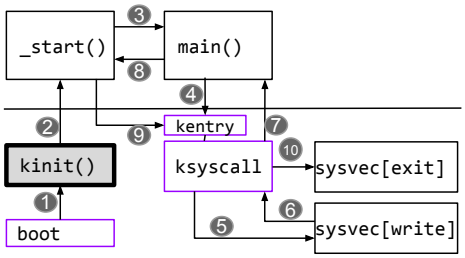


Au boot, ici, on se contente d'initialiser le pointeur de pile dans .kdata, puis d'entrer dans le noyau parce qu'il est déjà dans la mémoire (quelque part dans le segment d'adresse `.ktext`).

Dans un vrai système, le boot doit aller chercher le chargeur de système d'exploitation.
Ce chargeur se trouve souvent au tout début du disque dur, puis ce chargeur de système d'exploitation doit aller chercher le système d'exploitation sur le disque, puis doit l'écrire dans la mémoire et y entrer.

kernel/kinit.c/kinit()





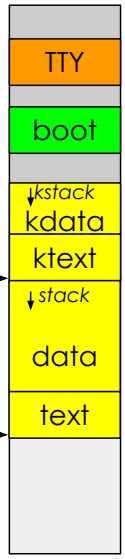
mise à 0 des variables globales du **kernel**

```
#include <klibc.h>

static char banner[] =  // banner's text defined on several lines
"  _       ___       __ \n"
"  | |__ /'v'\  / / \n"
"  | / /(     )/ _ \\n"
"  |_\_\ x___x \___/\n\n";

void kinit (void)
{
    kprintf (0, banner);
```

Démarre l'application qui, ici, est déjà en mémoire ! *En vrai, il faudrait la lire sur le disque*

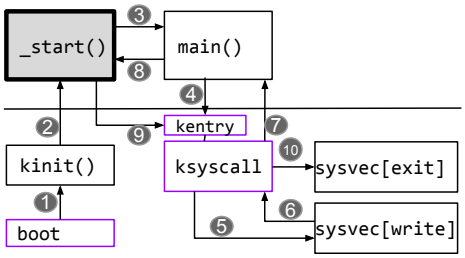
```
.globl app_load
app_load:
    mtc0    $4,     $14     // void app_load (void * fun)
    li      $26,    0x12    // go to user code
    mtc0    $26,    $12     // put _start address in c0_EPC
    la      $29,    __data_end // define next status reg. value
    eret                    // UM <- 1, IE <- 0, EXL <- 1
                            // define new user stack pointer
```

change le pointeur de pile en haut de `.data`




```
    // put bss sections to zero. bss contains uninitialised global variables
    extern int __bss_origin;    // first int of bss section (defined in ldscript kernel.ld)
    extern int __bss_end;       // first int of above bss section (defined in ldscript kernel.ld)
    for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);

    extern int _start;          // _start is the entry point of the app (defined in kernel.ld)
    app_load (&_start);         // function to start the user app (defined in hcpua.S)
}
```


ulib/crt0.c/_start()

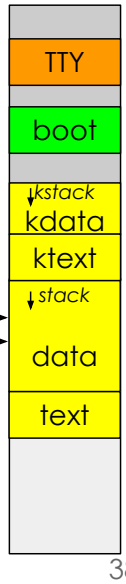



```
#include <libc.h>

extern int main (void);         // tell the compiler that main() exists

//int syscall (int a0, int a1, int a2, int a3, int syscall_code)
__asm__ (".globl syscall      \n"       // it is an external function, it must be declared globl
         "syscall:            \n"       // syscall function label
         "   lw  $2,16($29)   \n"       // since syscall has 5 parameters, the fifth is in the stack
         "   syscall          \n"       // EPC <- address of syscall, j 0x80000180, SR.EXL <- 1
         "   jr  $31          \n");     // $31 must not have changed

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

    extern int __bss_origin;        // first int of uninitialized global data
    extern int __bss_end;           // first int of above the uninitialized global data
    for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);

    res = main ();
    exit (res);
}
```

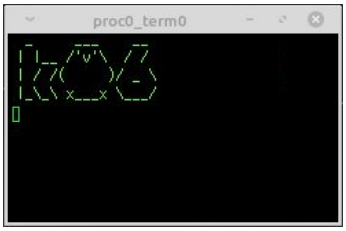
`_start()` est la fonction de démarrage de l'application, cette fonction n'est pas écrite par le programmeur et le noyau doit connaître son adresse pour y sauter !

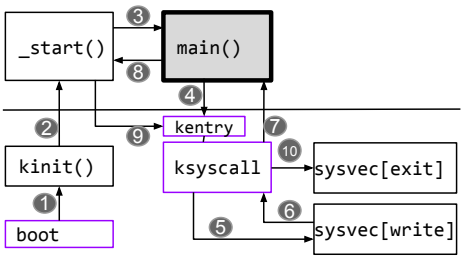
mise à 0 des variables globales non initialisées de **l'application**

fonction principale de l'application écrite par le programmeur. *Ici, elle n'a aucun argument, en vrai elle aurait les arguments du shell*



uapp/main.c/main()

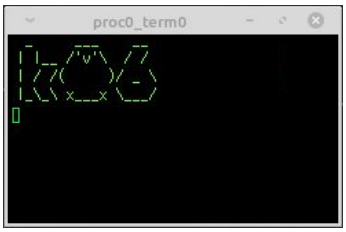


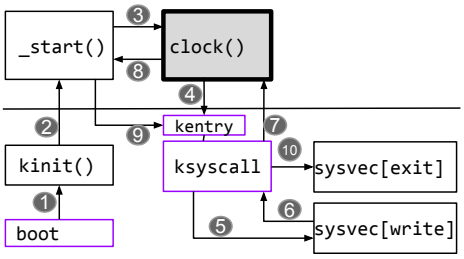



```c
#include <libc.h>

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

Ici, on veut juste afficher un message avec la date (en cycles).
Ce sera quand même 2 appels système : `clock()` et `fprintf()`
Le code de ces fonctions est dans la libc, d'où l'include au début

ulib/libc.c/clock()






```c
#include <syscalls.h>   // kernel services

unsigned clock (void)
{
    return syscall (0, 0, 0, 0, SYSCALL_CLOCK);
}
```

Le but de la fonction `clock()` est de préparer les arguments et d'appeler la fonction `syscall()` qui entre dans le noyau.

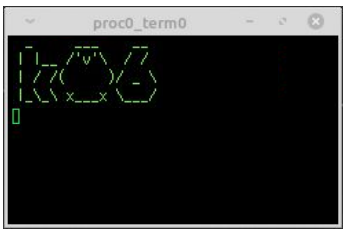
ici le code produit par gcc (un peu ré-ordonné pour la lisibilité et avec des commentaires)
la fonction `syscall()` est forcément écrite en assembleur mais elle ne fait pas grand-chose

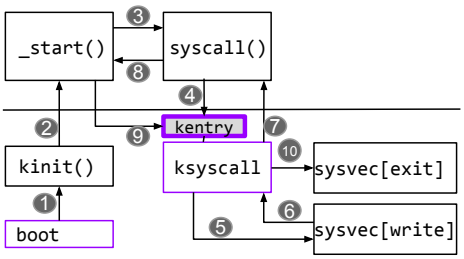
```
<clock>:
   27bdffe0    addiu   sp,sp,-32          # nr=1 nv=0 na=5 mais 32?
   afbf001c    sw      ra,28(sp)          # sauve ra = $31
   24020005    li      v0,5               # v0 = $2
   afa20010    sw      v0,16(sp)          # 5e arg dans la pile
   00003825    move    a3,zero            # a3 = $7
   00003025    move    a2,zero            # a2 = $6
   00002825    move    a1,zero            # a1 = $5
   00002025    move    a0,zero            # a0 = $4
   0fd00277    jal     7f4009dc <syscall> # appel de la fct syscall
   8fbf001c    lw      ra,28(sp)          # restore $31
   27bd0020    addiu   sp,sp,32           # restore le pt de pile
   03e00008    jr      ra                 # sort
```

```
<syscall>:
   8fa20010    lw  v0,16(sp)   # récupère $2 depuis la pile
   0000000c    syscall         # $4 à $7 inchangé -> syscall
   03e00008    jr  ra          # sort avec val de retour $2
```

kernel/hcpua.S/kentry

Entrée dans le noyau





Notez que le noyau peut utiliser les registres $26 et $27 sans les sauver avant car ces registres lui sont réservés. Ce sont des registres temporaires pour le noyau, gcc ne les utilise jamais, il ne contiennent rien pour l'application

```
kentry:                                 // kernel entry

    mfc0    $26,    $13                 // read CR (Cause Register)
    andi    $26,    $26,    0x3C        // apply cause mask (keep bits 2 to 5)
    li      $27,    0x20                // 0x20 is the syscall code
    bne     $26,    $27,    kpanic      // if not then that is not a syscall
```

```
ksyscall:
   [...]
```

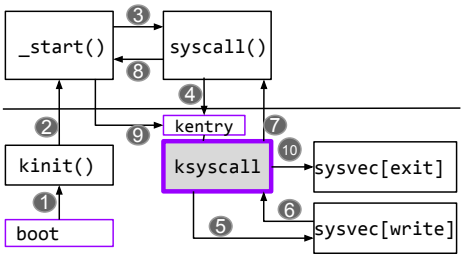
L'entrée dans le noyau, il faut analyser le champ XCODE du registre de cause `c0_cause` ($13 du coprocesseur 0) et, ici, si ce n'est pas le code de syscall, c'est la panique !

Le code du gestionnaire de syscall est juste après ...

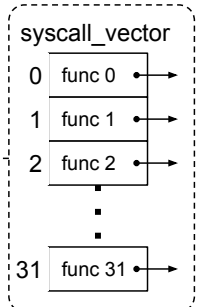
kernel/hcpua.S/ksyscall

Gestionnaire de syscalls → début

- alloue de la place dans la pile pour $31, EPC, SR et les 5 arguments des services : $4 à $7 + $2 (n° du service)
- trouve l'adresse du service dans `syscall_vector[$2]`
- change le mode d'exécution du MIPS → `UM=0 IE=0`
- appelle le service avec un `jal syscall_vector[$2]`




```
ksyscall:
    addiu   $29,    $29,    -8*4        // context for $31 + EPC + SR + syscall_code + 4 args
    mfc0    $27,    $14                 // $27 <- EPC (addr of syscall instruction)
    mfc0    $26,    $12                 // $26 <- SR (status register)
    addiu   $27,    $27,    4           // $27 <- EPC+4 (return address)
    sw      $31,    7*4($29)            // save $31 because it will be erased
    sw      $27,    6*4($29)            // save EPC+4 (return address of syscall)
    sw      $26,    5*4($29)            // save SR (status register)
    sw      $2,     4*4($29)            // save syscall code (useful for debug message)
    mtc0    $26,    $12                 // SR <- kernel-mode w/o INT (UM=0 ERL=0 EXL=0 IE=0)

    la      $26,    syscall_vector      // $26 <- table of syscall functions
    andi    $2,     $2,     SYSCALL_NR-1 // apply syscall mask
    sll     $2,     $2,     2           // compute syscall index (multiply by 4)
    addu    $2,     $26,    $2          // $2 <- & syscall_vector[$2]
    lw      $2,     ($2)                // $2 <- syscall_vector[$2]
    jalr    $2                          // call syscall function
```

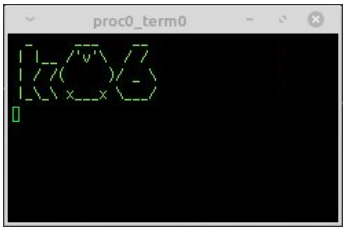


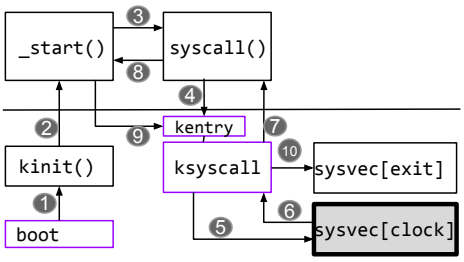

```
void *syscall_vector[] = {
    [0 ... SYSCALL_NR - 1] = unknown_syscall,
    [SYSCALL_EXIT]       = exit,
    [SYSCALL_TTY_READ]   = tty_read,
    [SYSCALL_TTY_WRITE]  = tty_write,
    [SYSCALL_CLOCK]      = clock,
};
```

initialisation du vecteur de syscalls = tableau de pointeur de fonctions ayant au plus 5 arguments : $4 à $7 et le 5e dans la pile (n°service)

kernel/hcpua.S/clock()

Gestionnaire de syscalls → service





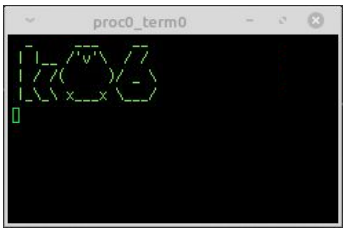

```
.globl clock
clock:
    mfc0    $2, $9
    jr      $31
```

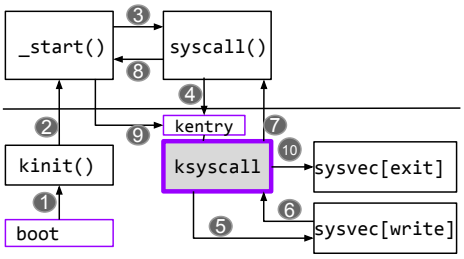
Le service `clock()` se contente de récupérer dans $2 la valeur du compteur de cycles présents par défaut dans le registre $9 du coprocesseur 0 du MIPS et sort !

`clock()` est nécessairement écrite en assembleur. `.globl` permet de la rendre visible hors de `hcpua.o`

kernel/hcpua.S/ksyscall

Gestionnaire de syscalls → fin

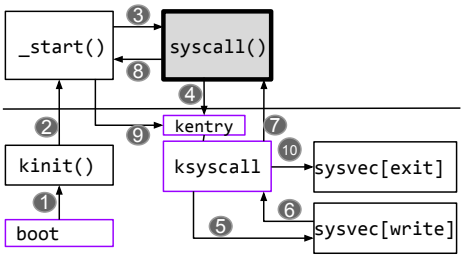





```
jalr    $2                      // call syscall function

lw      $26,    5*4($29)        // get old SR
lw      $27,    6*4($29)        // get return address of syscall
lw      $31,    7*4($29)        // restore $31 (return address of user syscall function)
mtc0    $26,    $12             // restore SR
mtc0    $27,    $14             // restore EPC
addiu   $29,    $29,    8*4     // restore stack pointer
eret                            // return : jr EPC with c0_sr.EXL <- 0
```

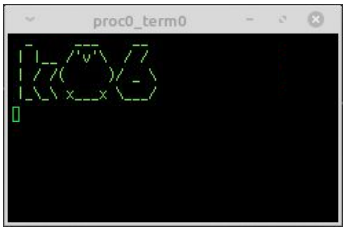
- La valeur de retour du service est dans $2
- on restore $31, EPC, SR
- on restore le pointeur de pile
- On sort avec `eret` ≈ `jr EPC` et effacer le bit `SR.EXL`

ulib/crt0.c/syscall()



À la sortie du noyau, on revient dans la fonction `syscall()` vue au slide 40

$2 contient la valeur de retour, il suffit de sortir pour revenir dans la fonction appelante ici on retourne dans `clock()`



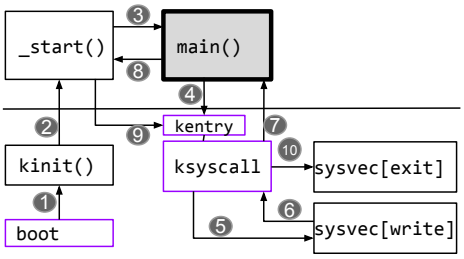

```c
#include <libc.h>
```

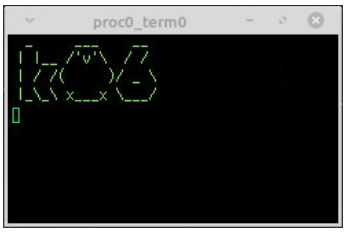
`syscall()` est forcément écrite en assembleur. Notez qu'on peut inclure du code assembleur directement dans un code C. Cette fonction est ici la seule fonction écrite en assembleur que nous aurons, c'est pourquoi, nous avons choisi de ne pas créer un fichier .S spécifique pour `syscall()` et la mettre dans le fichier `crt0.c` qui contient aussi la fonction `_start()`. Le fichier **`crt0.c`** contient en fait la **fonction d'entrée dans l'application** (`_start()`) et la **fonction de sortie** (`syscall()`)

Syntaxe : `__asm__("une string avec des \n entre instructions, ici, décomposée comme pour banner")`

```c
//int syscall (int a0, int a1, int a2, int a3, int syscall_code)
__asm__ (".globl syscall     \n"        // it is an external function, it must be declared globl
         "syscall:           \n"        // syscall function label
         "   lw  $2,16($29)  \n"        // since syscall has 5 parameters, the fifth is in the stack
         "   syscall         \n"        // EPC <- address of syscall, j 0x80000180, SR.EXL <- 1
         "   jr  $31         \n");      // $31 must not have changed
```


uapp/main.c/main()

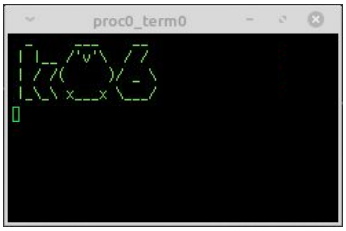


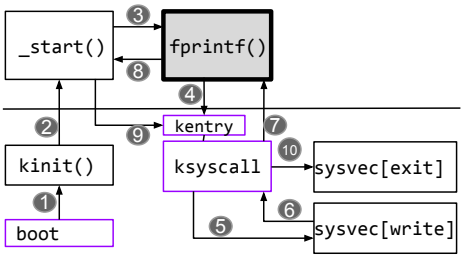



```c
#include <libc.h>

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

- Au retour de `clock()` qui ne contenait que l'appel à la fonction `syscall()`.
- On revient dans `main()` et on va entrer dans `fprintf()` définie dans `libc.c`

ulib/libc.c/fprintf()





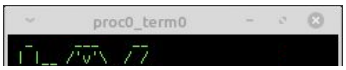

```
int fprintf(int tty, char *fmt, ...)
{
    int res;
    char buffer[PRINTF_MAX];
    va_list ap;
    va_start (ap, fmt);
    res = vsnprintf(buffer, sizeof(buffer), fmt, ap);
    va_end(ap);

    res = syscall( tty, (int)buffer, res, 0, SYSCALL_TTY_WRITE);
    return res;
}
```

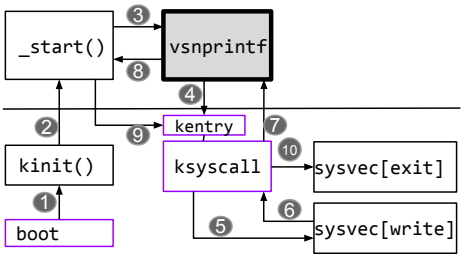
fprintf() est une fonction variadique
(avec un nombre variable d'arguments)

- Dans la déclaration, on met **...** à la fin
- On déclare une variable **ap** de type **va_list** (type **builtin** défini par gcc)
- **va_start()** est permet de dire à gcc quel est le dernier argument explicite et donc où commence la liste des arguments variables **ap**
- On pourrait directement utiliser **ap,** mais ici, on choisit de le passer en paramètre de la fonction **vsnprintf()** dont le dernier argument est de type **va_list**
- à la fin il faut fermer la liste ap avec **va_end()**

L'accès aux arguments de la liste **ap** est réalisé avec **va_arg()**, ici, dans **vsnprintf()**

ulib/libc.c/vsnprintf()





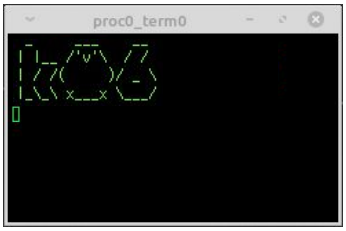

```
static int vsnprintf (char * buffer, unsigned size, char *fmt, va_list ap)
{
    char arg[16];       // buffer used to build the argument
    char *tmp;          // temporary pointer used to build arguments
    [...]

    while (*fmt) {                          // for all char in fmt
        [...]
        switch (*fmt) {                     // study the different cases
          [...]
          case 's':                         // case %s (string)
            tmp = va_arg (ap, char *);      // tmp points to this string argument
            tmp = (tmp) ? tmp : "(null)";   // replace "" by "(null)"
            goto copy_tmp;                  // go to copy tmp in buffer
          [...]
    }
  abort:
    *buf = '\0';                                // put the ending char 0
    res = (int)((unsigned)buf-(unsigned)buffer);// compute the nb of char to write
    return res;
}
```

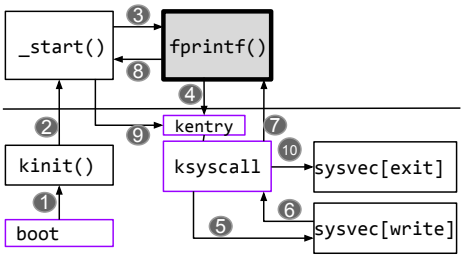
Dans ce slide, il n'y a pas le code de **vsnprinf()**, mais vous pouvez lire le code de la libc si cela vous intéresse :-)

Le principe consiste à parcourir la chaîne fmt et à chaque **%** on regarde le caractère suivant pour connaître le type de l'argument de la liste **ap** dont il faut prendre la valeur : **s:char*, d:int, x:int,** etc.

va_arg(ap,type) permet de prendre la variable suivante dans la liste **ap** et type informe gcc du type de cet argument.

ulib/libc.c/fprintf()






```
int fprintf(int tty, char *fmt, ...)
{
    int res;
    char buffer[PRINTF_MAX];
    va_list ap;
    va_start (ap, fmt);
    res = vsnprintf(buffer, sizeof(buffer), fmt, ap);
    va_end(ap);

    res = syscall( tty, (int)buffer, res, 0, SYSCALL_TTY_WRITE);
    return res;
}
```

à la fin il faut fermer la liste ap avec **va_end()**

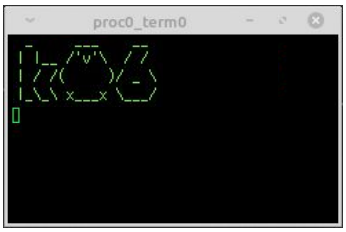
Au retour de **vsnprintf()**, le tableau **buffer[]** contient les caractères à envoyer au terminal n°**tty**, **res** contient le nombre de caractères dans la **buffer[]**

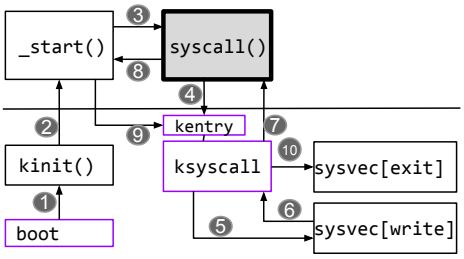
Pour écrire ces caractères, il faut demander au noyau par un appel système :
- **$4** : n°**tty**
- **$5** : adresse du **buffer[]**
- **$6** : nombre de caractères à afficher

En résumé :

Variadique : *type* `va_list ap` ; *ouverture* `va_start(ap,arg)` *puis N lectures* `va_arg(ap,type)` *et fermeture* `va_end(ap)`

ulib/crt0.c/syscall()

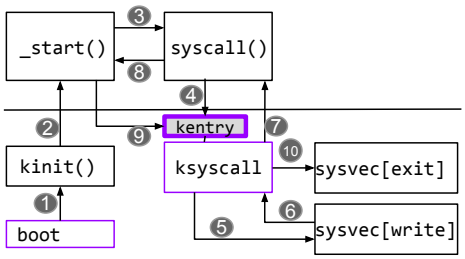




Nouvel appel de **syscall()**

```
#include <libc.h>

extern int __bss_origin;        // first int of uninitialized global data
extern int __bss_end;           // first int of above the uninitialized global data
extern int main (void);         // tell the compiler that main() exists

//int syscall (int a0, int a1, int a2, int a3, int syscall_code)
__asm__ (".globl syscall      \n"     // it is an external function, it must be declared globl
         "syscall:            \n"     // syscall function label
         "   lw  $2,16($29)   \n"     // since syscall has 5 parameters, the fifth is in the stack
         "   syscall          \n"     // EPC <- address of syscall, j 0x80000180, SR.EXL <- 1
         "   jr  $31          \n");   // $31 must not have changed
```

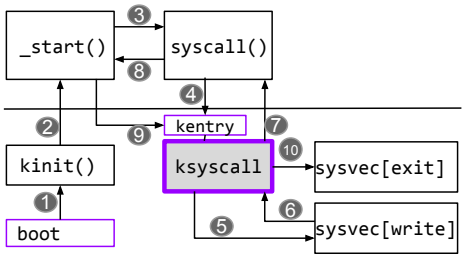
kernel/hcpua.S/kentry



Nouvelle entrée dans **kentry**

```
kentry:                                 // kernel entry

    mfc0    $26,    $13                 // read CR (Cause Register)
    andi    $26,    $26,    0x3C        // apply cause mask (keep bits 2 to 5)
    li      $27,    0x20                // 0x20 is the syscall code
    bne     $26,    $27,    not_syscall // if not then that is not a syscall
```

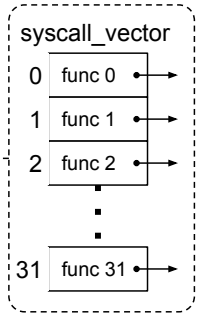

kernel/hcpua.S/ksyscall



Gestionnaire de syscalls → début

Cette fois l'appel système est **SYSCALL_TTY_WRITE**

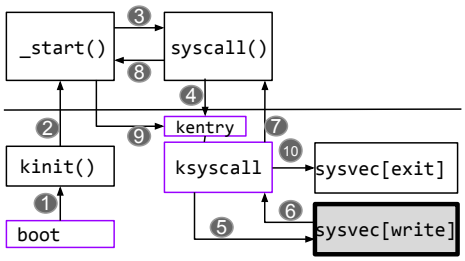
```
ksyscall:
    addiu   $29,    $29,    -8*4        // context for $31 + EPC + SR + syscall_code + 4 args
    mfc0    $27,    $14                 // $27 <- EPC (addr of syscall instruction)
    mfc0    $26,    $12                 // $26 <- SR (status register)
    addiu   $27,    $27,    4           // $27 <- EPC+4 (return address)
    sw      $31,    7*4($29)            // save $31 because it will be erased
    sw      $27,    6*4($29)            // save EPC+4 (return address of syscall)
    sw      $26,    5*4($29)            // save SR (status register)
    sw      $2,     4*4($29)            // save syscall code (useful for debug message)
    mtc0    $26,    $12                 // SR <- kernel-mode w/o INT (UM=0 ERL=0 EXL=0 IE=0)

    la      $26,    syscall_vector      // $26 <- table of syscall functions
    andi    $2,     $2,     SYSCALL_NR-1// apply syscall mask
    sll     $2,     $2,     2           // compute syscall index (multiply by 4)
    addu    $2,     $26,    $2          // $2 <- & syscall_vector[$2]
    lw      $2,     ($2)                // $2 <- syscall_vector[$2]
    jalr    $2                          // call syscall function
```



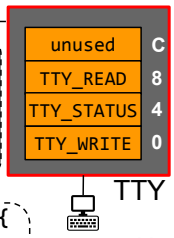

```
void *syscall_vector[] = {
    [0 ... SYSCALL_NR - 1] = unknown_syscall,
    [SYSCALL_EXIT]      = exit,
    [SYSCALL_TTY_READ]  = tty_read,
    [SYSCALL_TTY_WRITE] = tty_write,
    [SYSCALL_CLOCK]     = clock,
};
```

et donc : **jal syscall_vector[SYSCALL_TTY_WRITE] = tty_write()**

kernel/harch.c/tty_write()



Description de la composition des registres du TTY par une struct C




```
struct tty_s {
    int write;          // tty's output address
    int status;         // tty's status address something to read if not null)
    int read;           // tty's input address
    int unused;         // unused address
};

extern volatile struct tty_s __tty_regs_map[NTTYS]; // NTTYS is a #define
int tty_write (int tty, char *buf, int count)
{
    int res = count;
    tty = tty % NTTYS;  // to be sure that tty is an existing tty
    while ((count != 0) && (*buf != 0)) {
        __tty_regs_map[ tty ].write = *buf;
        count--;
        buf++;
    }
    return res - count;
}
```

L'accès à un registre du TTY est simplifié par cette déclaration de structure C

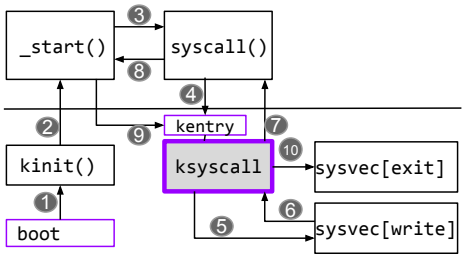
`_tty_regs_map` est un label défini dans le fichier ldscript `kernel.ld`, ce label est égal à l'adresse du premier registre du TTY dans l'espace d'adressage du MIPS (pour ce SoC).

Cette déclaration permet de dire au compilateur que cette adresse est de type `struct tty_s` et qu'elle est allouée ailleurs (`extern`)

Il y a autant de jeu de registres qu'il y a de TTY, d'où le tableau de struct

kernel/hcpua.S/ksyscall

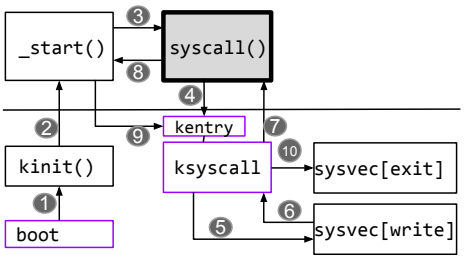
Gestionnaire de syscalls → fin




```
jalr    $2                      // call syscall function

lw      $26,    5*4($29)        // get old SR
lw      $27,    6*4($29)        // get return address of syscall
lw      $31,    7*4($29)        // restore $31 (return address of syscall function)
mtc0    $26,    $12             // restore SR
mtc0    $27,    $14             // restore EPC
addiu   $29,    $29,    8*4     // restore stack pointer
eret                            // return : jr EPC with EXL <- 0
```

ulib/crt0.c/syscall()

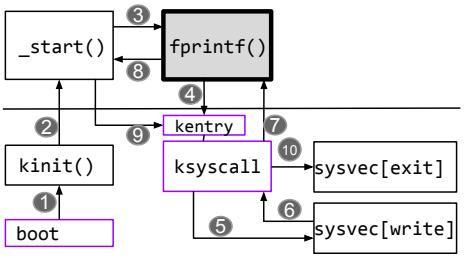
sortie de la fonction **syscall()**




```
#include <libc.h>

extern int __bss_origin;        // first int of uninitialized global data
extern int __bss_end;           // first int of above the uninitialized global data
extern int main (void);         // tell the compiler that main() exists

//int syscall (int a0, int a1, int a2, int a3, int syscall_code)
__asm__ (".globl syscall     \n"        // it is an external function, it must be declared globl
         "syscall:           \n"        // syscall function label
         "   lw  $2,16($29)  \n"        // since syscall has 5 parameters, the fifth is in the stack
         "   syscall         \n"        // EPC <- address of syscall, j 0x80000180, SR.EXL <- 1
         "   jr  $31         \n");      // $31 must not have changed
```


ulib/libc.c/fprintf()

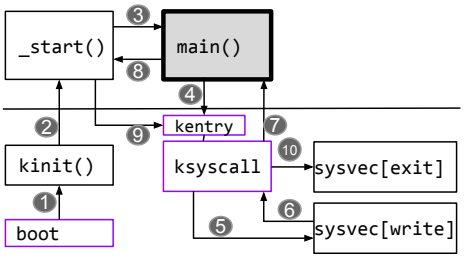
sortie de la fonction **fprintf()**




```
int fprintf(int tty, char *fmt, ...)
{
    int res;
    char buffer[PRINTF_MAX];
    va_list ap;
    va_start (ap, fmt);
    res = vsnprintf(buffer, sizeof(buffer), fmt, ap);
    va_end(ap);
    res = syscall( tty, (int)buffer, res, 0, SYSCALL_TTY_WRITE);
    return res;
}
```

uapp/main.c/main()

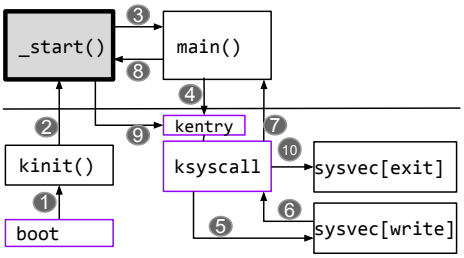
sortie de la fonction **main()**




```c
#include <libc.h>

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


ulib/crt0.c/_start()

retour dans la fonction **_start()**

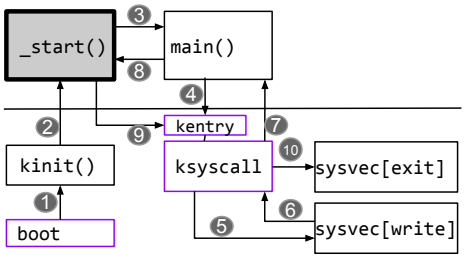



```c
#include <libc.h>

extern int __bss_origin;        // first int of uninitialized global data
extern int __bss_end;           // first int of above the uninitialized global data
extern int main (void);         // tell the compiler that main() exists

//int syscall (int a0, int a1, int a2, int a3, int syscall_code)
__asm__ (".globl syscall     \n"        // it is an external function, it must be declared globl
         "syscall:           \n"        // syscall function label
         "   lw  $2,16($29)  \n"        // since syscall has 5 parameters, the fifth is in the stack
         "   syscall         \n"        // EPC <- address of syscall, j 0x80000180, SR.EXL <- 1
         "   jr  $31         \n");      // $31 must not have changed

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

    for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);
    res = main ();
    exit (res);
}
```

Puisque la fonction **main()** n'a pas fait appel à **exit()** et que **exit()** est la seule manière de sortir d'un programme utilisateur, alors c'est la fonction **start()** qui doit le faire avec la valeur rendue par **main()**.
On ne revient jamais de **exit()** !

ulib/libc.c/exit()

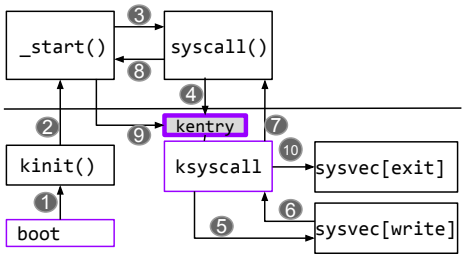
dernier appel système avec la valeur de retour de l'application




```c
void exit (int status)
{
    syscall( status, 0, 0, 0, SYSCALL_EXIT);        // never returns
}
```


kernel/hcpu.S/kentry

Dernière entrée dans **kentry**

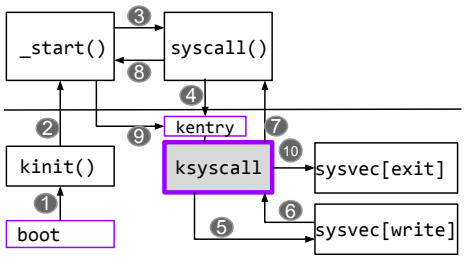



```
kentry:                                 // kernel entry

    mfc0    $26,    $13                 // read CR (Cause Register)
    andi    $26,    $26,    0x3C        // apply cause mask (keep bits 2 to 5)
    li      $27,    0x20                // 0x20 is the syscall code
    bne     $26,    $27,    not_syscall // if not then that is not a syscall
```

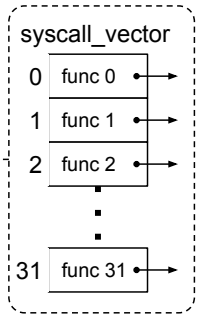
kernel/hcpua.S/ksyscall

Gestionnaire de syscalls → début

Cette fois l'appel système est **SYSCALL_EXIT**




```
ksyscall:
    addiu   $29,    $29,    -8*4            // context for $31 + EPC + SR + syscall_code + 4 args
    mfc0    $27,    $14                     // $27 <- EPC (addr of syscall instruction)
    mfc0    $26,    $12                     // $26 <- SR (status register)
    addiu   $27,    $27,    4               // $27 <- EPC+4 (return address)
    sw      $31,    7*4($29)                // save $31 because it will be erased
    sw      $27,    6*4($29)                // save EPC+4 (return address of syscall)
    sw      $26,    5*4($29)                // save SR (status register)
    sw      $2,     4*4($29)                // save syscall code (useful for debug message)
    mtc0    $26,    $12                     // SR <- kernel-mode w/o INT (UM=0 ERL=0 EXL=0 IE=0)

    la      $26,    syscall_vector          // $26 <- table of syscall functions
    andi    $2,     $2,     SYSCALL_NR-1    // apply syscall mask
    sll     $2,     $2,     2               // compute syscall index (multiply by 4)
    addu    $2,     $26,    $2              // $2 <- & syscall_vector[$2]
    lw      $2,     ($2)                    // $2 <- syscall_vector[$2]
    jalr    $2                              // call syscall function
```

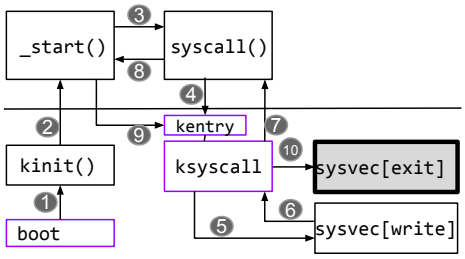



```
void *syscall_vector[] = {
    [0 ... SYSCALL_NR - 1] = unknown_syscall,
    [SYSCALL_EXIT]      = exit,
    [SYSCALL_TTY_READ]  = tty_read,
    [SYSCALL_TTY_WRITE] = tty_write,
    [SYSCALL_CLOCK]     = clock,
};
```

et donc : **jal syscall_vector[SYSCALL_EXIT] = exit()**

kernel/klibc.c/exit()

On affiche un message, on est dans le noyau : **kprintf()** et **clock()** sont appelées directement sans syscall




```
void exit (int status)
{
    kprintf (0, "\n\n[%d] EXIT status = %d\n", clock(), status);
    while (1);  // infinite loop
}
```

Là c'est la fin ! on fige le simulateur !!!

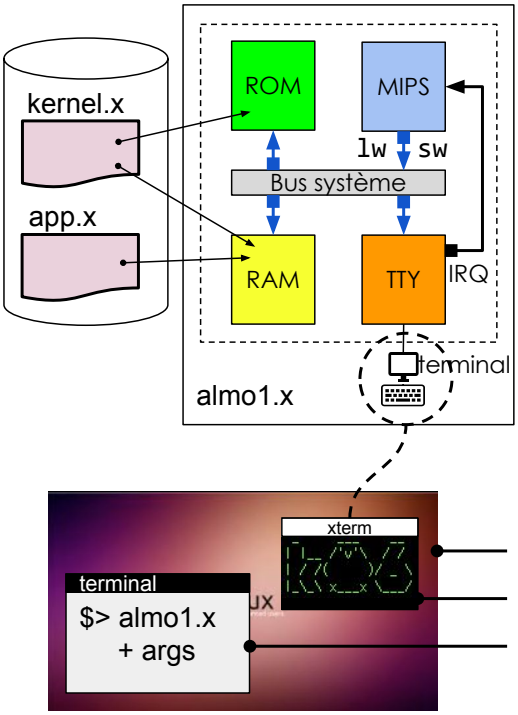
Le simulateur propose un mode debug pour de suivre la trace d'exécution détaillée des instructions **trace0.s** et la séquence d'appel des fonctions **label0.s**
le 0 c'est le n° du processeur, si on en a plusieurs, il y a **trace1.s**, **label1.s**, etc.

```
> cd k06/tp2
> cd 3_syscalls
> make debug
> less label0.s
```

label0.s

```
K    12:    <boot> ---------------- kernel/hcpua.S
K    37:    <kinit> --------------- kernel/kinit.c
K    60:    <kprintf>-------------- kernel/klibc.c
K   116:    <vsnprint-------------- kernel/klibc.c
K  1371:    <tty_puts-------------- kernel/harch.c
K  2189:    <app_load-------------- kernel/hcpua.S
U  2222:    <_start> --------------- user/crt0.c
U  2299:    <main> ----------------- user/main.c
U  2349:    <syscall>-------------- ulib/crt0.c
K  2370:    <kentry> --------------- kernel/hcpua.S
K  2384:    <ksyscall-------------- kernel/hcpua.S
K  2446:    <clock> ---------------- kernel/hcpua.S
U  2519:    <syscall>-------------- ulib/crt0.c
K  2521:    <kentry> --------------- kernel/hcpua.S
K  2526:    <ksyscall-------------- kernel/hcpua.S
K  2546:    <tty_puts-------------- kernel/harch.c
U  2811:    <syscall>-------------- ulib/crt0.c
K  2824:    <kentry> --------------- kernel/hcpua.S
K  2829:    <ksyscall-------------- kernel/hcpua.S
K  2849:    <tty_puts-------------- kernel/harch.c
U  2927:    <syscall>-------------- ulib/crt0.c
K  2929:    <kentry> --------------- kernel/hcpua.S
K  2943:    <ksyscall-------------- kernel/hcpua.S
K  2993:    <exit> ----------------- kernel/klibc.c
K  3007:    <clock> ---------------- kernel/hcpua.S
K  3034:    <kprintf>-------------- kernel/klibc.c
K  3050:    <vsnprint-------------- kernel/klibc.c
K  3730:    <tty_puts-------------- kernel/harch.c
mode  date  label                   file
```

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`

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    = 0x00B00000 ;
__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
}
```

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 l'espace d'adressage

Description de la manière de remplir les régions (ex: `kdata_region`) de l'espace d'adressage avec des sections de sorties (ex. .kdata) contenant les sections d'entrées produites par le compilateur (ex: .rodata, .sdata).

ulib/app.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 */

_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. Il y en a moins, l'application n'a pas à connaître les régions du kernel.

Description des régions de l'espace d'adressage

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

Conclusion

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

Nous avons vu

- que le MIPS a deux modes d'exécution : kernel et user
- que le mode user interdit les adresses supérieures à `0x80000000`
- 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 `0x80000180` quelque-soit 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'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 le noyau sait comment appeler l'application grâce à la convention `crt0`
- que la première fonction de l'application est `_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

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