reboot

LU3IN031 Architecture des ordinateurs - 2
Matériel et Logiciel
B1

franck.wajsburt@lip6.fr
V4

Objectifs du Module Archi-1

Concernant le matériel

- Circuiterie numérique simple (multiplexeur, additionneur, shifter, etc.)
- Principe d'exécution des instructions d'un processeur RISC
- Architecture d'un SoC mono-cœur et périphériques non-initiateurs (passifs)
- Concept d'espace d'adressage du MIPS
- Modes d'exécution du MIPS
- Gestion des requêtes d'interruption

Concernant le logiciel

- Programmation assembleur MIPS
- Convention des appels de fonctions pour le MIPS
- Partitionnement du système d'exploitation : noyau et bibliothèque système
- Démarrage du SoC jusqu'à l'exécution d'une application déjà en mémoire
- Gestionnaire des appels système
- Gestionnaire des interruptions

Objectifs du Module Archi-2

Concernant le matériel

- Micro-architecture des opérateurs en vue de leur performance
- Architecture interne d'un MIPS, en particulier le concept des micro-instructions
- Architecture d'un SoC avec plusieurs MIPS se partageant le même espace d'adressage
- Architecture d'un cache de premier niveau et les problèmes de cohérence en multi-MIPS
- Contrôleur de périphériques initiateurs (capable d'accéder à l'espace d'adressage)
- *Fonctionnement du contrôleur vidéo ou du contrôleur de disque (prévu)*

Concernant le logiciel

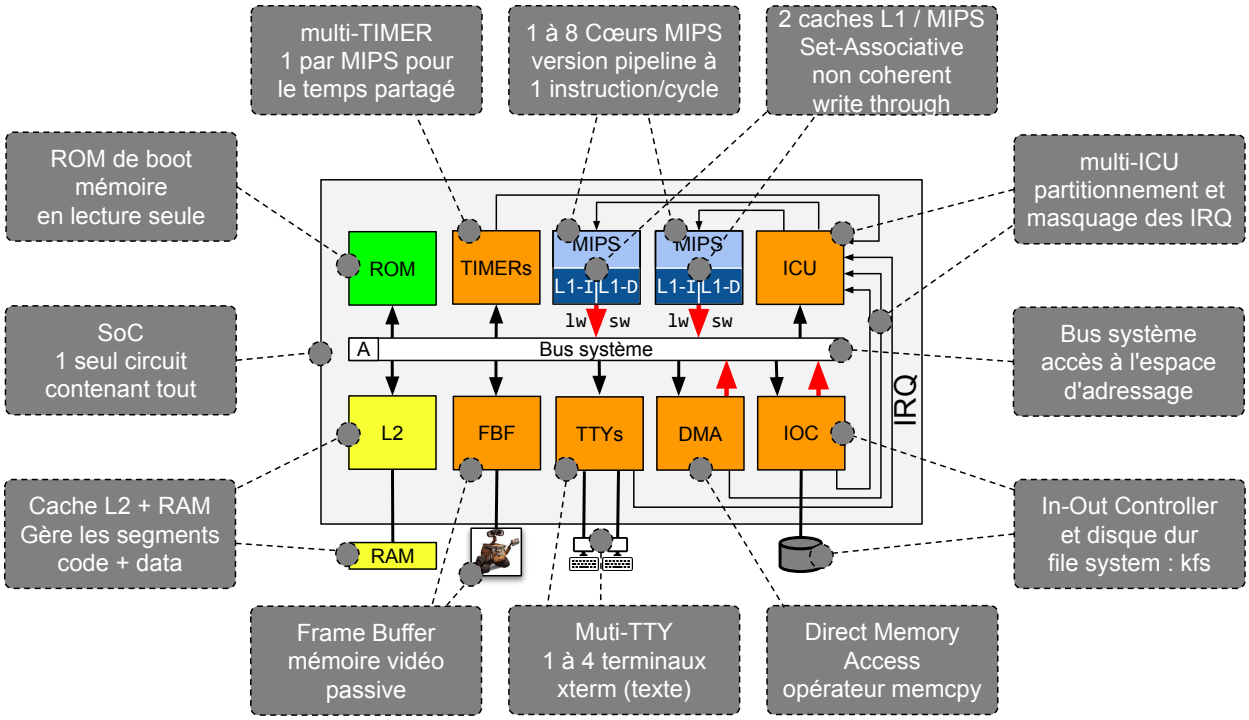
- Gestion de la mémoire dynamique, nécessaire pour créer des variables et les détruire
- API de gestion simplifiée des listes doublement chaînées pour les structures du noyau
- Gestion des états de threads et des listes d'attentes sur les ressources partagées
- Gestion propre des pilotes de périphériques
- Mécanismes de communications / synchronisation des threads en mono et multi-cores
- *API graphique (type SDL) ou File System (type FAT) (prévu)*

Plan de la séance

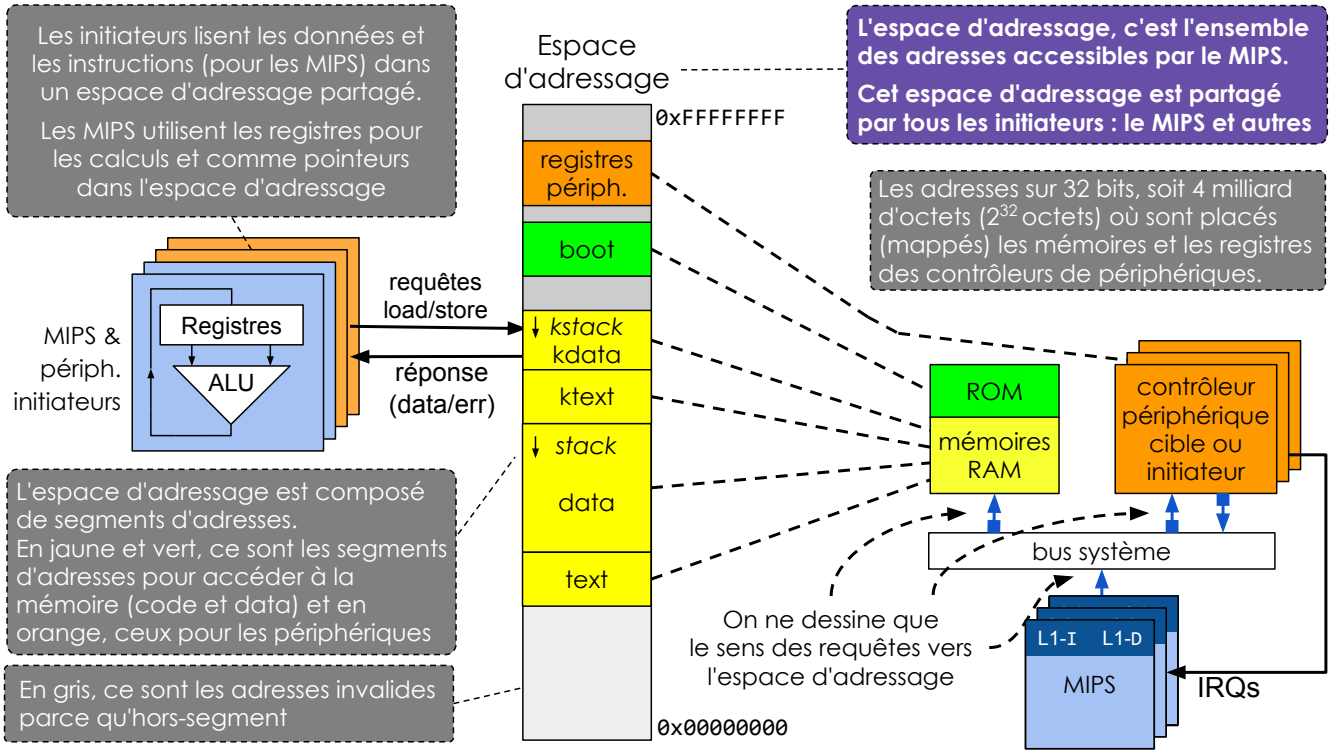
- Présentation d'un System-on-Chip (SoC) et espace d'adressage
- Rôle et constitution d'un système d'exploitation
- Compilation et placement du code dans l'espace d'adressage
- Exécution du simulateur du prototype virtuel du SoC
- Les modes d'exécution du MIPS du point de vue du matériel
- Passage entre noyau et application du point de vue du logiciel

SoC

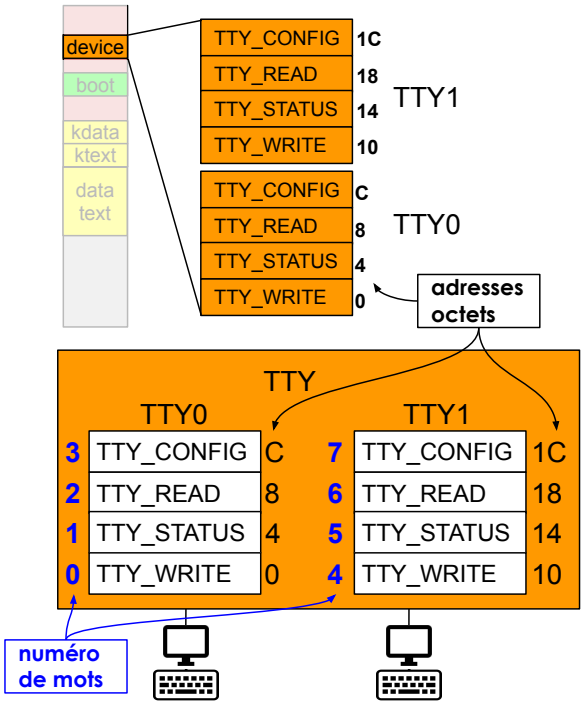
SoC - System-on-Chip - almo1



Autre vue d'un SoC



Cible : Contrôleur de terminaux TTY



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

Pour chaque terminal

- `TTY_WRITE` 1 mot en écriture seule, le caractère ascii est mis dans l'octet de poids faible
  → sortie vers l'écran
- `TTY_STATUS` 1 mot en lecture seule, ≠ 0 s'il y a un caractère ascii en attente dans TTY_READ
- `TTY_READ` 1 mot en lecture seule, le caractère ascii tapé est dans l'octet de poids faible
  lire TTY_READ acquitte l'IRQ du TTY concerné
- `TTY_CONFIG` inutilisé dans cette version, mais permet la configuration p. ex. du débit d'échange avec le terminal externe.

Chaque TTY lève une IRQ si un caractère est reçu par le TTY donc si son STATUS est ≠ de 0 (n TTY → n IRQ)

Le composant dessiné gère 2 TTYs, ce nombre est configurable de 1 à 4

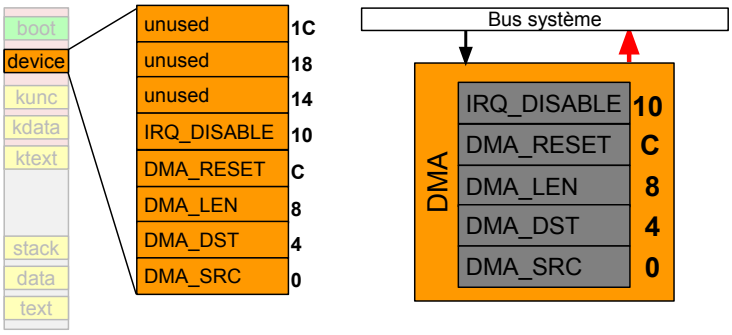
Initiateur : DMA (Direct Memory Access)

Le DMA réalise un copie de mémoire à partir de l'adresse DMA_SRC
vers l'adresse DMA_DST de DMA_LEN octets.
Dans l'ordre, on commence par écrire les adresses SRC, DST et IRQ_DISABLE (si besoin),
puis on écrit LEN, ce qui provoque le démarrage de la copie par le DMA

- DMA_IRQ_DISABLE (lecture/écriture) masquage de la ligne IRQ
- DMA_RESET (écriture seule) acquittement de la ligne IRQ
- DMA_LEN (écriture/lecture) taille en octets à déplacer
- DMA_DST (écriture seule) adresse de destination
- DMA_SRC (écriture seule) adresse source

A la fin de l'opération,
le DMA lève un interruption
et LEN contient le nombre
d'octet non écrits.

S'il est différent de 0,
c'est qu'il y a une erreur.



En résumé

Un SoC (System-on-Chip) est un ordinateur entier sur un circuit intégré, il est composé de :

- un ou plusieurs processeur(s) pour exécuter les programmes, ici ce sont des MIPS, avec des caches de premier niveau pour réduire le nombre d'accès mémoire (gain en performance et en énergie)
- une mémoire RAM ou d'un cache de niveau 2 vers une RAM externe contenant un ou plusieurs segment(s) d'adresses *mappés** dans l'espace d'adressage physique partagé du SoC.
  - pour le code du programme utilisateur et pour le système d'exploitation ;
  - pour les données du programme utilisateur et du système d'exploitation ;
  - et pour les piles d'exécution des fils d'exécution des programmes.
- une mémoire ROM avec le code de démarrage du processeur *mappé* dans l'espace d'adressage.
- des contrôleurs d'entrées-sorties configurables par des registres *mappés* dans l'espace d'adressage, donc accessible par des lectures / écritures (ces registres ne sont ni dans le MIPS, ni dans la RAM).
- des lignes d'interruption permettant aux contrôleurs d'entrées-sorties de prévenir de la terminaison des commandes demandées, ces lignes passent par un composant ICU qui permet de les masquer et de les router vers le bon processeur.
- un bus système routant (acheminant) les requêtes de lecture / écriture du ou des processeur(s) et des contrôleurs de périphériques initiateurs (comme DMA et IOC) vers les composants gérant les adresses concernées de l'espace d'adressage partagé.

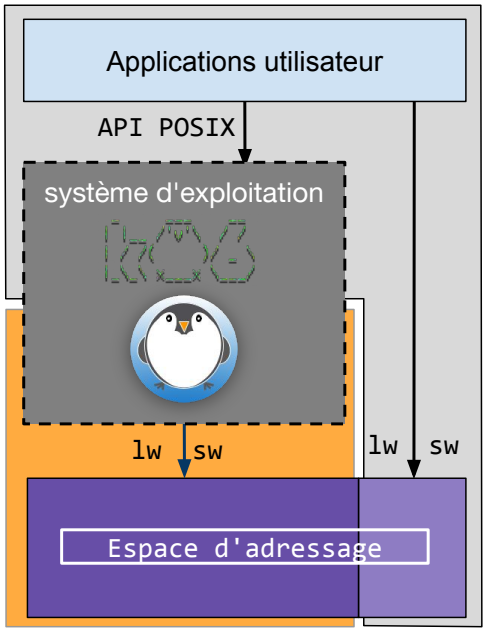
* *Mapper signifie « associer ou mettre en correspondance » deux éléments. Ici, les segments d'adresses des bancs de mémoires ainsi que les registres des contrôleurs de périphérique sont mappés dans l'espace d'adressage du MIPS*

Système d'exploitation

L'idée, ici, est de comprendre les mécanismes de base au coeur d'un système d'exploitation et pour ce faire, vous utiliserez un système construit spécialement pour ce module : **kO6** (prononcé « kit O 6 » (c'est la lettre O et non le chiffre 0)

Système d'exploitation et Applications

Le système d'exploitation permet aux applications de s'exécuter sur le Soc



Les applications utilisent les services d'un système d'exploitation au travers d'une API standard qui **virtualise** le matériel pour la rendre portable sur d'autres machines (POSIX Portable Operating Système Interface uniX https://www.wikiwand.com/fr/POSIX).

Services du système d'exploitation :

- bibliothèque de **fonctions standards** pour le calcul ou pour l'interface IHM (Interface-Homme-Machine)
- **création** et **destruction** des applications
- **allocation** équitable des ressources matérielles : mémoire, périphériques, fichiers, ports réseau, processeur(s), etc.
- **communication**, **synchronisation** et **gestion de signaux** entre applications

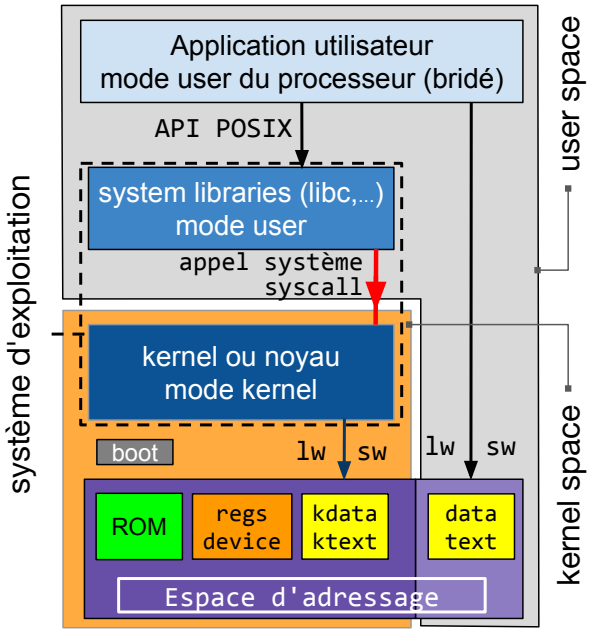
Le système d'exploitation garantit :

- la **sécurité** des applications, c'est-à-dire l'**intégrité** et la **confidentialité** des données, ainsi que la **disponibilité** des ressources ;
- la **sûreté** de fonctionnement du matériel.

Le système d'exploitation et les applications accède au matériel grâce à l'espace d'adressage et aux instruction load / store

Bibliothèques système - Noyau - Boot

Le système d'exploitation est composé des bibliothèques système et du Noyau (kernel)



La sécurité s'appuie sur les modes d'exécution du processeur

- user : le processeur est bridé, une partie de l'espace d'adressage et une partie des instructions sont interdites
- kernel : le processeur peut utiliser tout l'espace d'adressage et toutes les instructions (pour le changement de mode)

L'application utilise les fonctions des bibliothèques système, lesquelles s'exécutent en mode user et encapsulent les appels système implémentés exécutés en mode kernel dans le noyau

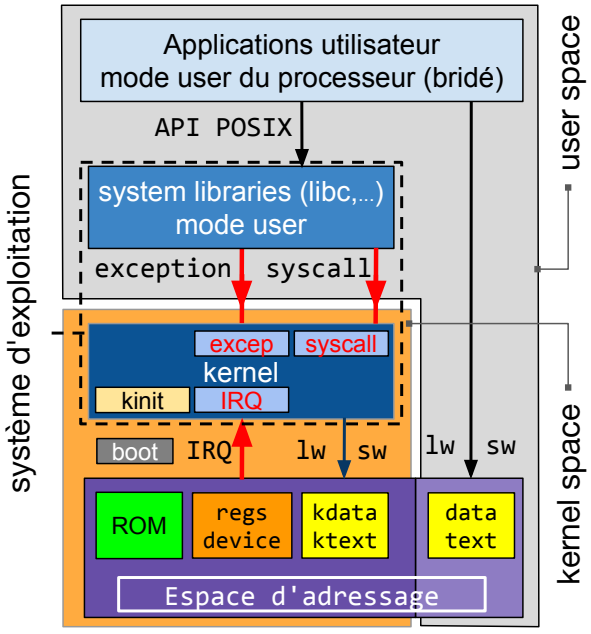
- il y a les bibliothèques POSIX : libc (printf(), read(), etc.), Pthreads, etc. et d'autres (window manager, maths, etc.)
- mais l'application utilise aussi directement une partie de l'espace d'adressage sans passer par le noyau.

Le code de boot démarre la machine

- il est dans une mémoire persistante (ROM) à l'adresse `0xBFC00000`, il ne fait pas partie du système d'exploitation
- En principe, il scanne le matériel présent, puis il place en mémoire un chargeur de noyau, lequel place le noyau en mémoire qui démarre et charge la 1re application utilisateur
- Sur le SoC des TP, les mémoires sont préchargées avec le noyau et une application, mais cela pourrait changer...

Les appels faits au noyau

Le noyau du système d'exploitation est appelé pour 4 raisons distinctes



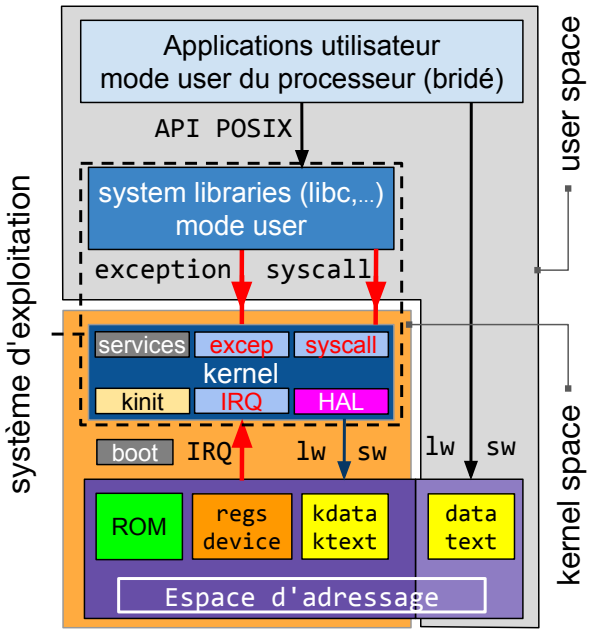
Après le boot, on doit entrer dans le noyau (une seule fois) et pendant l'exécution des applications, pour les **syscalls**, pour les **exceptions** et pour les **IRQ** des périphériques.

1. La **fonction kinit()**, la fonction d'entrée du kernel, initialise matériel, les structures de données et lance l'application `0`
2. Le **gestionnaire syscall** offre les services des sys. lib. p. ex :
   - les commandes de périphériques
   - l'allocation de mémoire
   - le lancement et l'arrêt d'application
3. Le **gestionnaire d'exception** gère les erreurs d'exécution du programme (le processeur ne sait pas quoi faire), p. ex. :
   - la division par 0 ou les instructions inconnues
   - la violation de privilège (accès interdit en mode **user**)
   - les erreurs d'accès mémoire (segmentation fault)
4. Le **gestionnaire d'interruption** gère les requêtes d'interruption (IRQ) (toujours attendues) venant des périphériques, p. ex. :
   - les fins de commande (lorsque le travail est fait)
   - les **ticks** d'horloge (pour que l'OS compte le temps)

Pour les 3 gestionnaires, c'est le même point d'entrée, nommé **kentry**, et placé à l'adresse `0x80000180` de l'espace d'adressage

Le code du noyau doit être portable

Le noyau définit des API pour l'accès à l'architecture matérielle



Le noyau implémente les **services** proposés aux applications, p. ex:

- Gestion des applications
- Gestion des fils d'exécution des applications
- Gestion de l'allocation mémoire
- Gestion des files d'attente des ressources partagées
- Gestion de la mémoire virtuelle
- Gestion des caches du système de fichiers
- Gestion des ports de communication réseau

Le code des services est doit être indépendant des détails de l'architecture. Il faut, bien sûr, un disque dur pour avoir le service de gestion des caches de fichiers mais le code du noyau ne doit pas dépendre du système de fichiers présent.

Pour ce faire, le noyau définit une interface d'abstraction du matériel (API **HAL** Hardware Abstraction Layer) pour abstraire :

- l'accès au processeur : masquage des interruptions, sauvegarde de contextes, flush des caches, etc.
- L'accès aux périphériques (**drivers**) : open, read, write, etc. (dans unix, les périphériques sont gérés comme des fichiers).
- L'accès au réseau client-server : select, connect, etc.

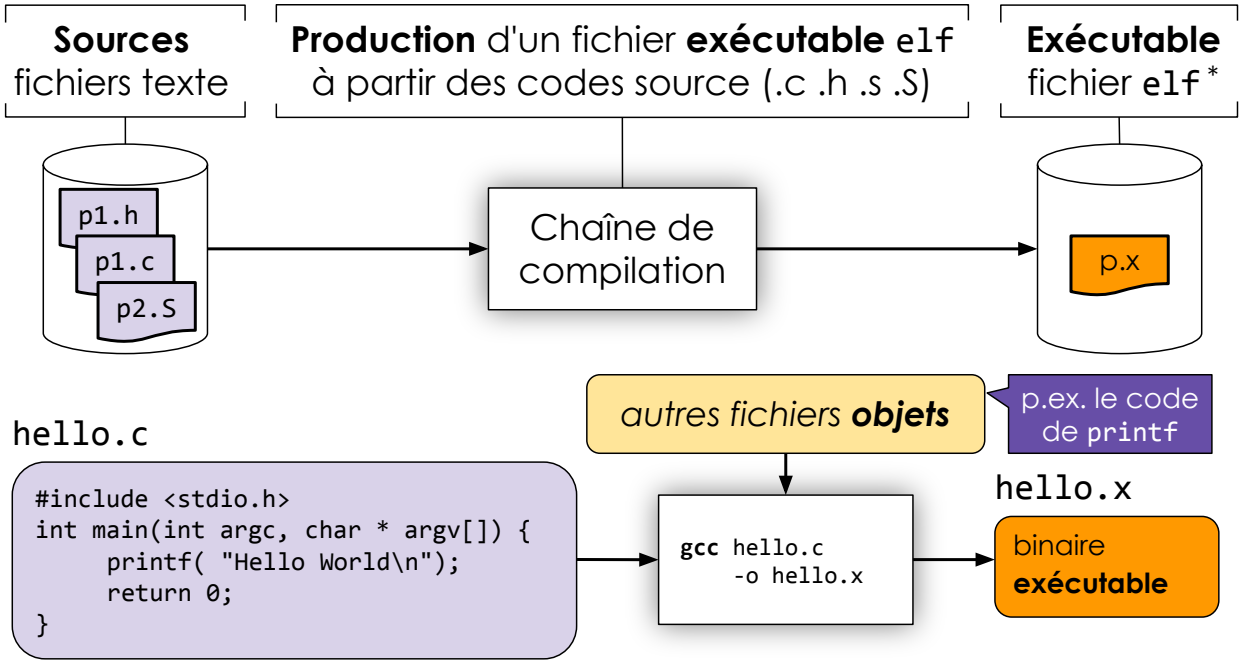
Cette API **HAL** n'est pas standard mais elle existe car elle permet le portage d'un système d'exploitation sur plusieurs machines.

En résumé

- Une API définit une interface standard de services. C'est ensemble de déclarations de fonctions, de types, de variables, etc. (en langage C, elle est définie dans des fichiers `.h`)
- Les applications utilisent un système d'exploitation (OS Operating System) pour accéder aux ressources matérielles grâce à des API utilisateur comme POSIX.
- Un système d'exploitation est composé de 2 parties : les bibliothèques système qui implémentent l'API utilisateur (POSIX) et le noyau (kernel) qui gère les ressources matérielles.
- Les applications et les bibliothèques système s'exécutent en mode user (mode sans privilège : c'est-à-dire sans pouvoir accéder aux ressources protégées)
- Le noyau utilise le mode kernel du processeur pour s'exécuter (mode privilégié). *Notez que nous parlons des noyaux monolithiques qui gère seul tous les services en mode kernel, il existe d'autres architectures de noyau (micro-noyau, exo-noyau, noyau hybride)*
- Le noyau rend ses services grâce à 3 gestionnaires : gestionnaire d'appel système (syscall), gestionnaire d'exception (erreur) et gestionnaire d'interruptions (IRQ)
- Le processeur démarre en mode kernel pour exécuter le code de boot qui entre dans le noyau pour initialiser le matériel (ce code est dans la HAL) et ses propres structures de données avant de démarrer la première application (et la seule pour cette UE).
- Pour cette UE, nous allons utiliser un petit OS ad hoc nommé kO6, POSIX-like.

Compilation

Chaîne de compilation

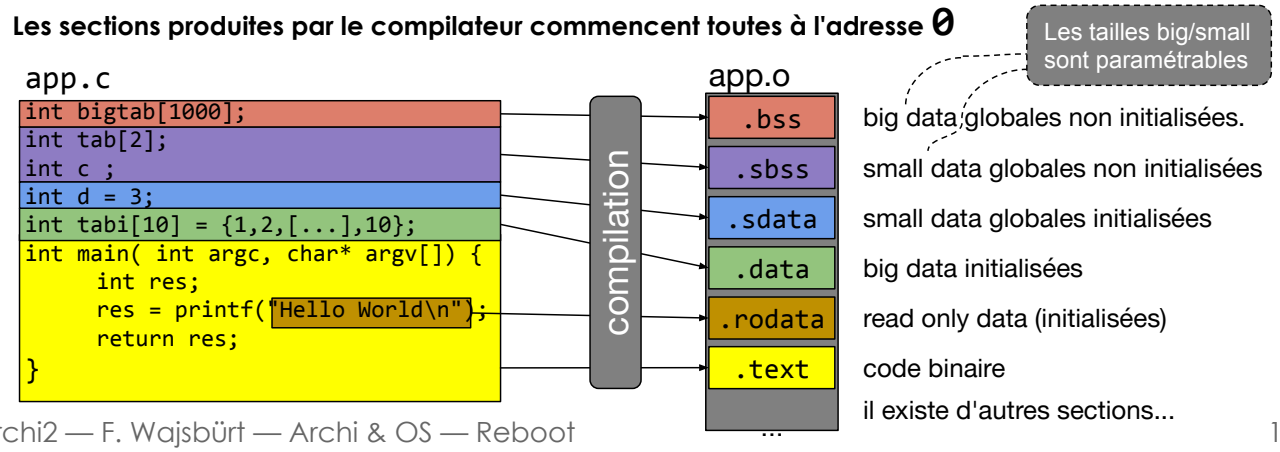


* Executable and Linking Format

Fichier objet

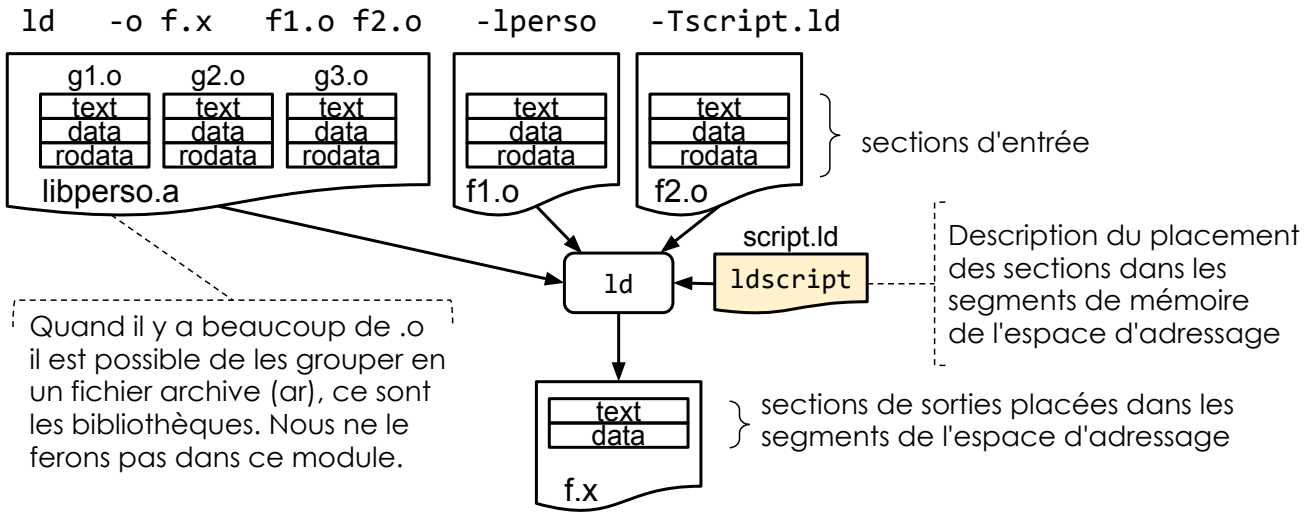
Le compilateur met le code et les données globales dans des sections typées du fichier objet

- Une section est un segment d'adresses destinée à contenir une catégorie d'information
  - Le code est dans une section `.text`
  - Les données globales sont placées dans différentes sections (`.*data*`, `.*bss*`, etc.) en fonction de leur taille et du fait qu'elles sont initialisées ou pas.
  - Les données globales non explicitement initialisées dans le code de l'application, sont initialisées à 0 au lancement de l'application.
- Il n'y a pas de sections pour les variables locales (tel que `int res`) car ce sont des données qui n'existent qu'à l'exécution du programme, et qui seront placées dans la pile d'exécution à une position en mémoire choisie par le noyau du système d'exploitation.
- **Les sections produites par le compilateur commencent toutes à l'adresse 0**



Edition de liens

Le compilateur produit des fichiers objets (`.o`) avec du code binaire incomplet, les sections ne sont pas placées dans l'espace d'adressage par le compilateur lui-même et donc les adresses de saut ou de variables globales dans le `.o` ne sont pas connues.
⇒ **Il faut lier les `.o` (les unir) pour produire un fichier exécutable complet (au format elf)**.



Edition de lien : fichier ldscript

```
__tty_regs_map    = 0xd0200000 ;
__boot_origin     = 0xbfc00000 ;
__boot_length     = 0x00001000 ;
__ktext_origin    = 0x80000000 ;
__ktext_length    = 0x00020000 ;
__kdata_origin    = 0x80020000 ;
__kdata_length    = 0x003E0000 ;
__kdata_end       = __kdata_origin + __kdata_length;
```

déclaration des variables du ldscript

Les noms choisis pour ces variables sont quelconques mais ils sont préfixés par convention par un double underscore « _ _ » pour éviter les conflits de noms avec les variables ou fonctions du programmes.
(https://www.gnu.org/software/libc/manual/html_node/Reserved-Names.html)

```
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
}
```

```
SECTIONS {
    .boot : {
        *(.boot)
    } > boot_region
    .ktext : {
        *(.text*)
    } > ktext_region
    .kdata :
        *(.*data*)
        . = ALIGN(4);
        __bss_origin = .;
        *(.*bss*)
        . = ALIGN(4);
        __bss_end = .;
    } > kdata_region
}
```

syntaxe : fichier-objet(section) * est un caractère *joker* qui prend tous les fichiers

définition : concaténation dans la section de sortie `.boot` de l'ensemble des sections .boot trouvées dans tous les fichiers objets (puisqu'on a mis une *) donnés en argument à l'éditeur de liens et placement dans la région boot_region

section de sortie (en rouge)

. représente le pointeur d'adresse de remplissage dans la région courante

Les sections d'entrée (en bleue) remplissent la section de sortie (en rouge)

Le pointeur d'adresse de remplissage est déplacé pour être multiple de 2^4

région (nous mettons ici une seule section de sortie par région)

Espace d'adressage de l'application user

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

`ulib/app.ld`

Les régions 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
}
```

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

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

Espace d'adressage 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    = 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
}
```

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 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).

Notez que le code est aussi dans 2 types de section, .kentry et .text

Compilation des codes source OS et app

```
.
├── Makefile
├── common
│   └── syscalls.h
├── kernel
│   ├── Makefile
│   ├── harch.c
│   ├── harch.h
│   ├── hcpu.h
│   ├── hcpua.S
│   ├── hcpuc.c
│   ├── kernel.ld
│   ├── kinit.c
│   ├── klibc.c
│   ├── klibc.h
│   └── ksyscalls.c
├── uapp
│   ├── Makefile
│   └── main.c
└── ulib
    ├── Makefile
    ├── crt0.c
    ├── libc.c
    ├── libc.h
    └── user.ld
```

Nous n'allons pas détailler le fonctionnement des outils de compilation et des makefiles, mais vous aurez le code source commenté que vous pourrez lire.

- A gauche se trouve la version la plus complexe du code du tp1.
- En dessous, un extrait de la séquence des commandes invoquées par le Makefile hiérarchique placé à la racine (*en haut de la liste à gauche*)

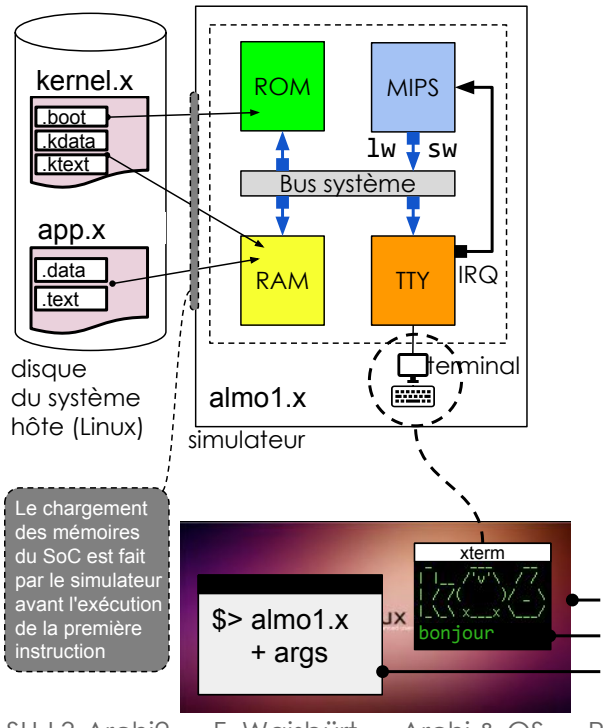
```
make -C kernel compil NTTYS=1 NCPUS=1
    makedepend [...] sources
    mipsel-unknown-elf-gcc -o obj/hcpua.o -c [...] hcpua.S
    mipsel-unknown-elf-gcc -o obj/ksyscalls.o -c [...] ksyscalls.c
    mipsel-unknown-elf-gcc -o obj/harch.o -c [...] harch.c
    mipsel-unknown-elf-gcc -o obj/hcpuc.o -c [...] hcpuc.c
    mipsel-unknown-elf-gcc -o obj/klibc.o -c [...] klibc.c
    mipsel-unknown-elf-gcc -o obj/kinit.o -c [...] kinit.c
    mipsel-unknown-elf-ld -o ../kernel.x -T kernel.ld kernel_objets
make -C ulib compil NTTYS=1 NCPUS=1
    makedepend [...] sources
    mipsel-unknown-elf-gcc -o obj/libc.o -c [...] libc.c
    mipsel-unknown-elf-gcc -o obj/crt0.o -c [...] crt0.c
make -C uapp   compil NTTYS=1 NCPUS=1
    makedepend [...] sources
    mipsel-unknown-elf-gcc -o obj/main.o -c [...] main.c
    mipsel-unknown-elf-ld -o ../user.x -T ../ulib/user.ld user_objets

almo1.x -KERNEL kernel.x -APP user.x -NTTYS 1 -NCPUS 1
```

compilation du **noyau** de l'OS (système d'exploitation)

compilation des bibliothèques système et de l'application

Simulateur du SoC

Simulateur du SoC `almo1.x`



Le simulateur du SoC utilisé en TP se nomme **`almo1.x`**.
Ce simulateur est configurable, on peut choisir le nombre de périphériques et même de MIPS

```
$> almo1.x -KERNEL kernel.x -APP app.x -NTTYS 2
```

`almo1.x` prend en paramètre :

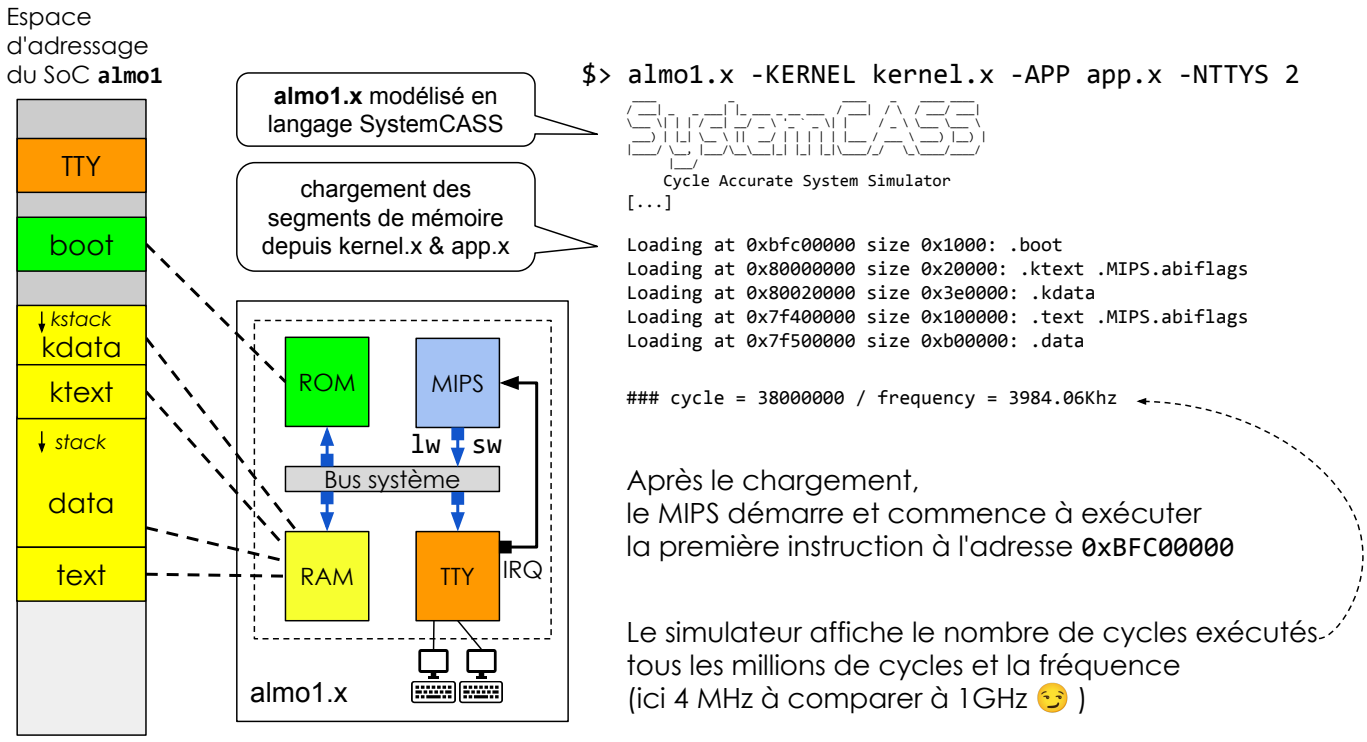
- les fichiers binaires MIPS du noyau et de l'application
- les paramètres du matériel (p. ex. nombre de TTY)
- le nombre de cycles à simuler (p. ex. 1000000)
- la demande de trace d'exécution des instructions

`almo1.x`

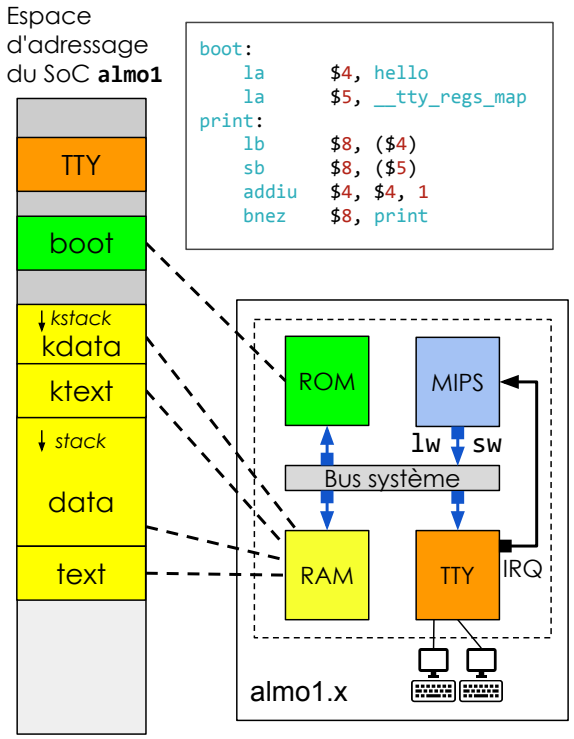
- **remplit les segments** de mémoire avec ce qu'il trouve dans les fichiers binaires (**`.text`** et **`.data`**)
- **active le reset** : le signal de démarrage du MIPS
- **démarre** l'exécution de la première instruction

Écran du système hôte (Linux)
TTY du simulateur **`almo1.x`**
terminal de commande Linux pour lancer **`almo1.x`**

Exécution des applications



Trace d'exécution du simulateur



Le simulateur permet de voir la trace d'exécution
des instructions assembleur cycle par cycle

```
$> make debug
almo1.x -KERNEL kernel.x -DEBUG 0 -NCYCLES 10000 -NTTY 1 [...]
Cycle Accurate System Simulator
[...]
generate trace[cpu].s, may take a while...
```

Chaque MIPS a son fichier de trace l'extension .s c'est pour la colorisation de l'éditeur de texte

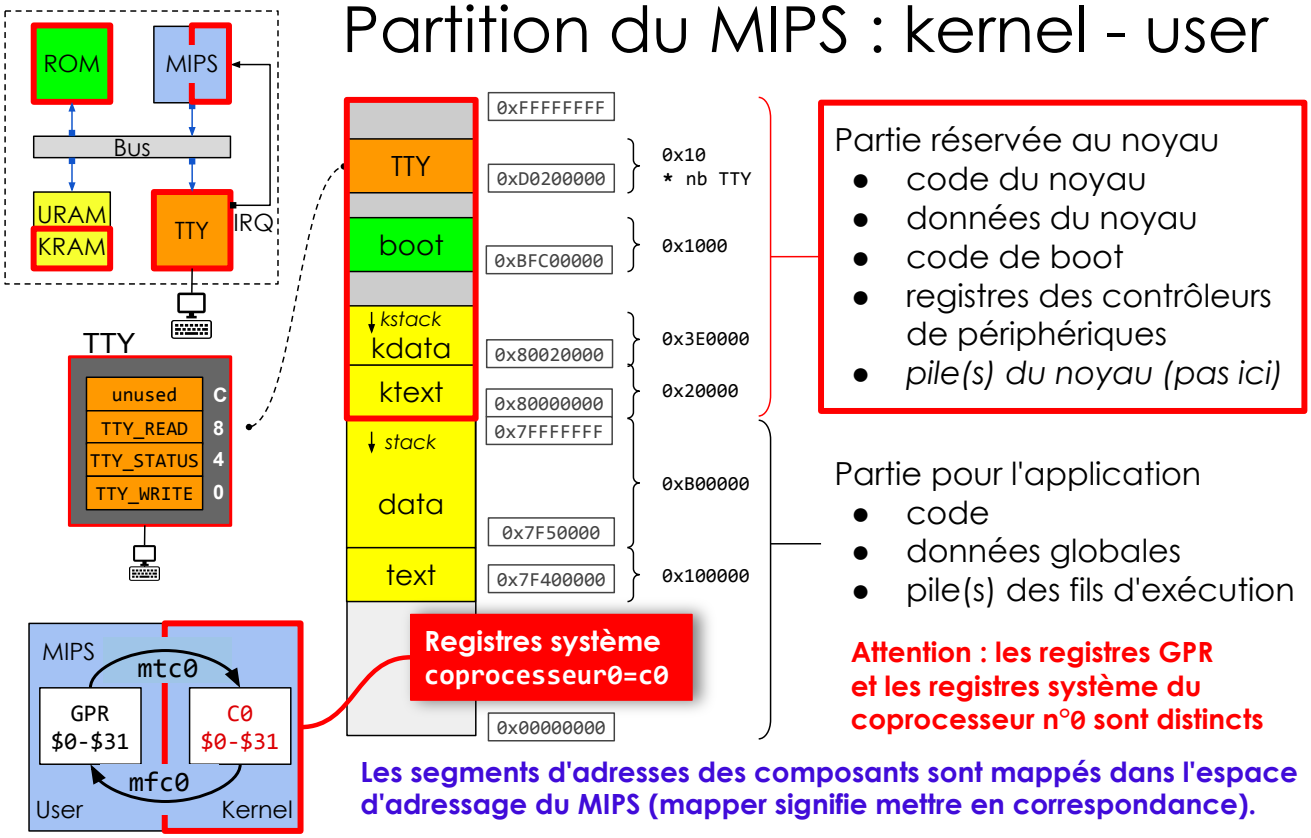
```
$> head trace0.s
```

mode	cycle	adresse	instruction	code assembleur
K	12:	<boot>:----------		
K	12:	0xbfc00000	0x3c04bfc0	lui a0,0xbfc0
K	13:	0xbfc00004	0x24840028	addiu a0,a0,40
K	14:	0xbfc00008	0x3c05d020	lui a1,0xd020
K	15:	0xbfc0000c	0x24a50000	addiu a1,a1,0
K	26:	<print>:----------		
K	26:	0xbfc00010	0x80880000	lb t0,0(a0)
K	27:	0xbfc00014	0xa0a80000	sb t0,0(a1)
K	37:	--> READ MEMORY @ 0xbfc00028 BE=---1 --> 0x6c6c6548		
K	39:	<-- WRITE MEMORY @ 0xd0200000 BE=---1 <-- 0x48		

et accès à la mémoire

Modes d'exécution du MIPS

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)*

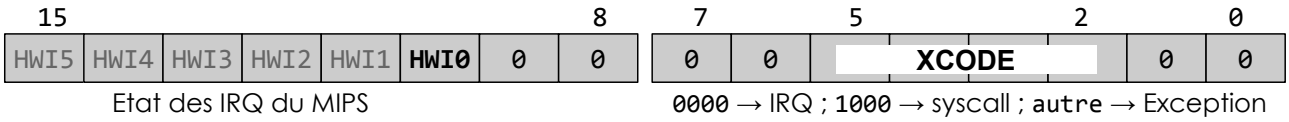
Partie pour l'application
- code
- données globales
- pile(s) des fils d'exécution

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

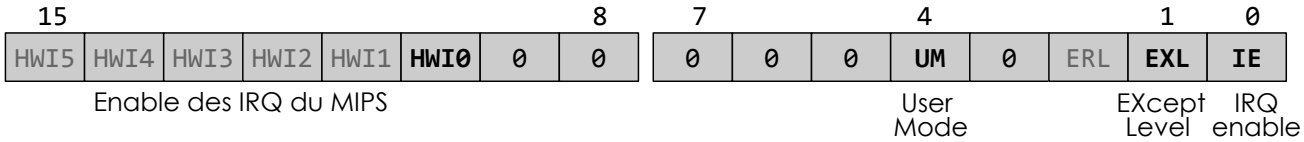
Les segments d'adresses des composants sont mappés dans l'espace d'adressage du MIPS (mapper signifie mettre en correspondance).

Registres système : Status, Cause, EPC

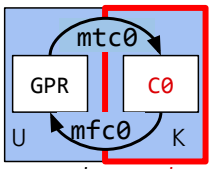
Le registre `c0_cause ($13)` contient la cause d'entrée dans le noyau (si IRQ, syscall ou except)



Le registre `c0_sr ($12)` contient le mode d'exécution du MIPS et les autorisations d'IRQ



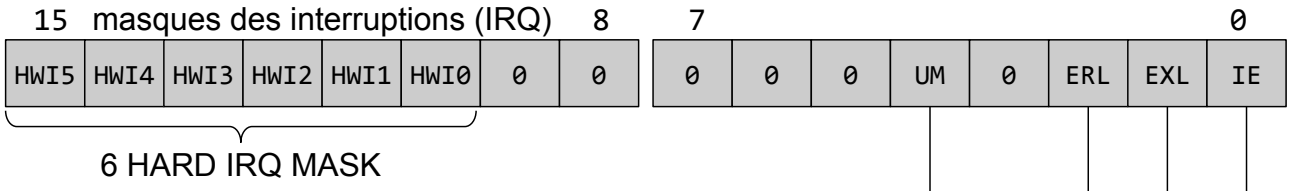
Le registre `c0_epc ($14)` contient l'adresse de retour si c'est une IRQ
ou l'adresse de l'instruction courante pour syscall et toutes les exceptions




```
mtc0 $GPR, $C0
mfc0 $GPR, $C0
```


Status Register : c0_sr ($12 du c0)

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

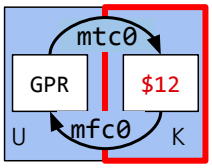


User Mode = 1
en mode USER — **UM**

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

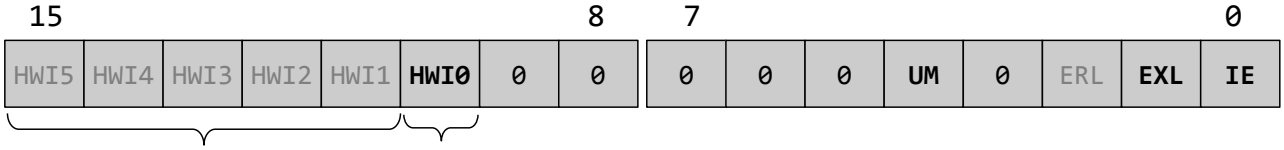
Exception Level = 1
à l'entrée dans le noyau après syscall, une exception et IRQ — **EXL**

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

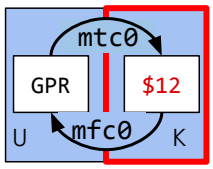
- Si UM est à 1: le MIPS est en mode USER
- Si IE est à 1: le MIPS autorise les IRQ à interrompre le programme courant

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**

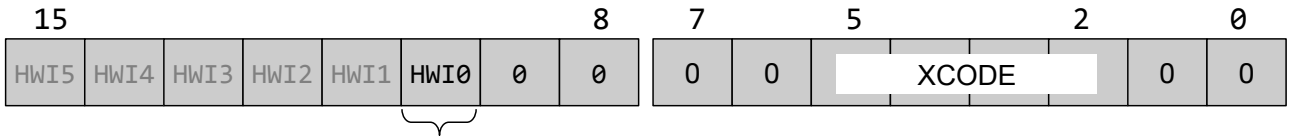
Valeurs typiques de c0_sr pour la plateforme

- 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 c0)

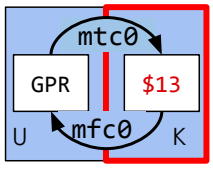
Le registre CR contient la cause d'entrée dans le noyau (après syscall, except ou irq)



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` ou **exception** ou **interruption**

`c0_sr.EXL` ← `1`

mise à 1 du bit EXL du registre Status Register donc passage en mode kernel interruptions masquées

`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 `kentry` toute cause confondue [syscall, exception, interruption]

`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

En résumé

- 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 pas utiliser
  - l'espace d'adressage au delà de **`0x7FFFFFFF`**
  - les instructions **`mtc0`**, **`mfc0`** et **`eret`** 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 (en plus du démarrage)
  - 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

Passages entre le noyau et l'application

passage kernel → application

Il y a 2 types de passage kernel → application

1. Au démarrage de l'application et là Il y a deux problèmes à résoudre :
   - **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 la fonction `_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 au 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"`
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)

* https://www.wikiwand.com/fr/Segment_BSS

boot → kinit → app_load → _start → main

Kernel | User

kernel/hcpua.S

```
.section .boot,"ax"
boot:
    la      $29,    __kdata_end  # kernel stack ptr
    la      $26,    kinit        # goto kinit
1   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, __bss_end; // kernel bss
    for (int *a = &__bss_origin;
             a != &__bss_end; *a++ = 0);
    extern int _start;
2   app_load (&_start);    // load & launch user app.
}
```

kernel/hcpua.S

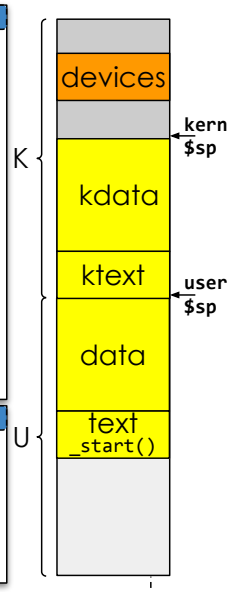
```
.globl app_load
app_load:
    mtc0    $4,    $14             # c0_EPC ← _start
    li      $26,   0x12            # c0_SR ←  0x12,
    mtc0    $26,   $12             # UM←1 EXL←1 IE←0
    la      $29,   __data_end      # user stack ptr
3   eret                           # PC ← EPC & EXL←0
```

ulib/crt0.c

```
#include <libc.h>

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

    extern int __bss_origin;     // init
    extern int __bss_end;        // user bss
    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;
}
```



La pile du kernel est à la fin de la section `.kdata`, celle de l'application est à la fin de la section `.data`
L'application est au début de la section `.text`

** La fin de ce cours contient une visite détaillée du code*

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

passage application → kernel

Il y a **3 gestionnaires** d'appel du kernel : **syscall**, **exception** et **interruption**
Dans tous les cas, le MIPS saute à **kentry** en `0x80000180` avec la cause dans `c0_cause`

Le **gestionnaire des syscalls**

syscall : service demandé au kernel avec la convention d'appel ci-dessous

- **`$2`** contient un numéro de service (numéros communs kernel / user)
- **`$4, $5, $6, $7`** contiennent les arguments du service (jamais plus de 4 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
- l'instruction `syscall` se comporte presque comme un appel de fonction, sauf que la fonction appelante de `syscall` ne réserve pas de place dans la pile pour les arguments (`$4` à `$7`)

l'instruction **`syscall`** provoque l'appel de fonction **`SYSCALL_VECTOR[$2]($4,$5,$6,$7,$2)`**
`SYSCALL_VECTOR[]` est un tableau de pointeurs sur des fonctions dans le noyau

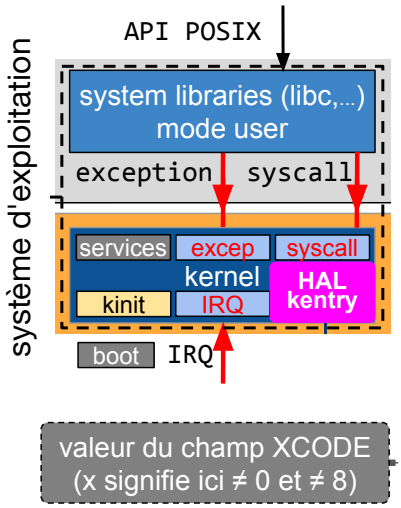
Le **gestionnaire des interruptions**

exceptions et **interruptions**

- Une exception est une faute du programme, dans notre cas, elles sont fatales, mais parfois 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 2 instructions. Dans les deux cas, tous les registres sont conservés intacts, l'interruption a juste « volé » du temps à l'application courante.

passage kernel → application

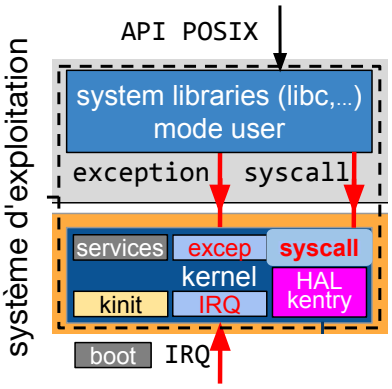
kentry est l'unique porte d'entrée du noyau
→ *Sauf au démarrage où l'on entre dans le noyau par* `kinit()`



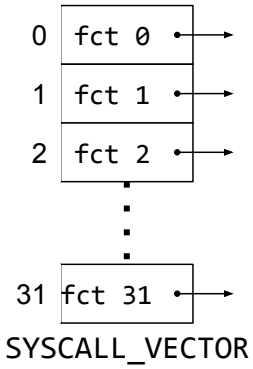
- Le code de **`kentry`** est à l'adresse `0x80000180`
- Il est nécessairement en assembleur et donc dans **HAL**
- Il ne modifie aucun registre sauf `$26` et `$27`
- Il analyse le champ `XCODE` du registre de `c0_cause` pour savoir quel gestionnaire appeler :
  - (8) gestionnaire de **syscall** (service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

gestionnaire de syscall

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



- C'est du code assembleur qui appelle des fonctions
  On aurait pu écrire une partie en C mais avec plus de lignes de code, alors ici, tout est en 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**, nommé **vecteur de syscall**, de pointeurs de fonctions dont la **case n°i** contient le pointeur vers la **fonction réalisant le service n°i**
  - `SYSCALL_NR` : le nombre de services
  - `void * 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 juste des appels de fonctions.



clock → syscall_fct → kentry → syscall_handler → clock

User | Kernel

ulib/libc.c

```
#include <syscalls.h> // numéro de syscall
// la fonction user clock() appelle la fonction 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

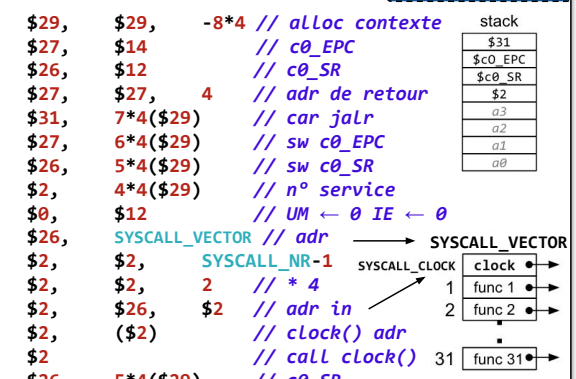
```
#include <libc.h>

// 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

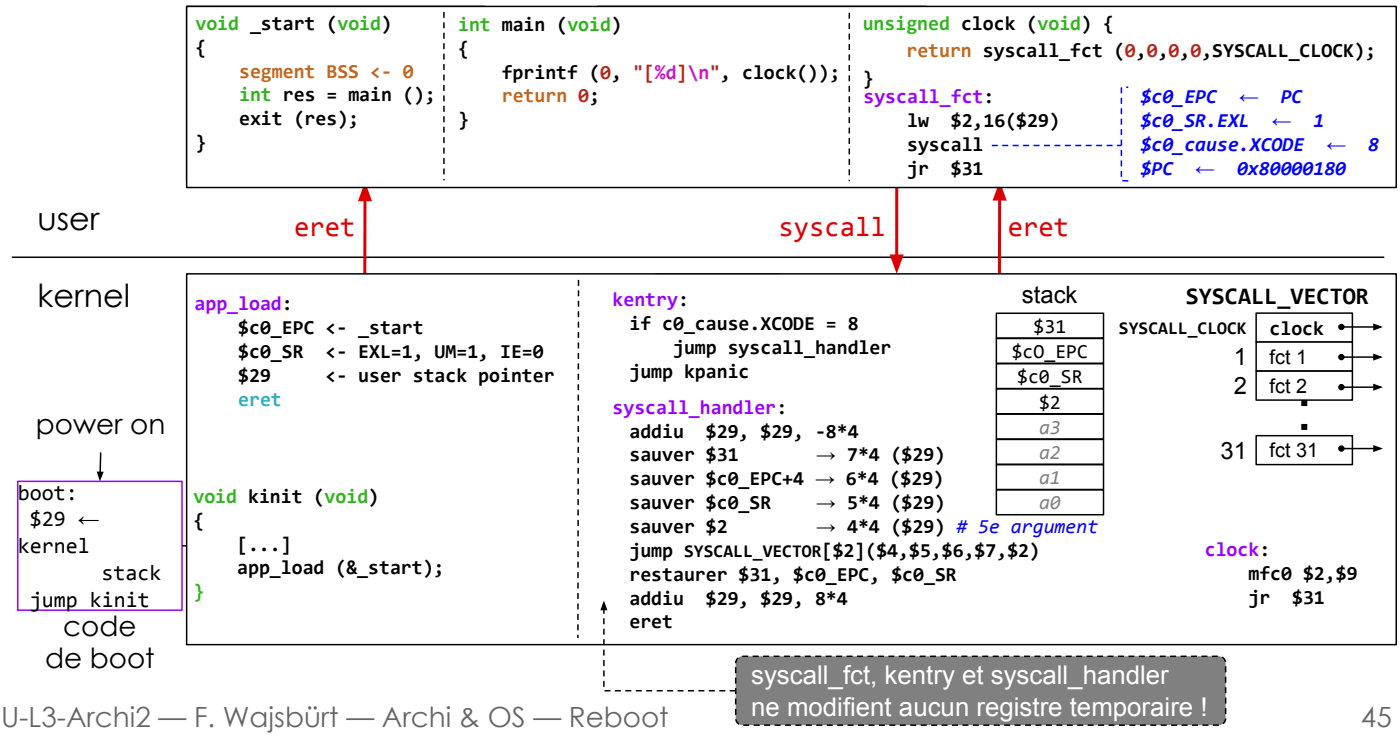
```
syscall_handler:
    addiu   $29,    $29,    -8*4 // alloc contexte
    mfc0    $27,    $14          // c0_EPC
    mfc0    $26,    $12          // c0_SR
    addiu   $27,    $27,    4    // adr de retour
    sw      $31,    7*4($29)     // sauv jalr
    sw      $27,    6*4($29)     // sauv c0_EPC
    sw      $26,    5*4($29)     // sauv 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
```



kernel/hcpua.S

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

Un parcours de boot à syscall *(at a glance)*



En résumé

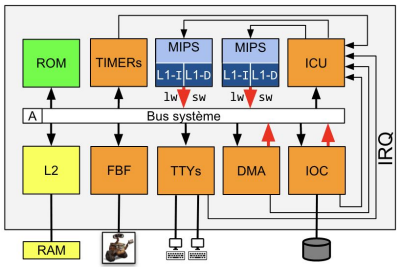
- Le noyau connaît l'adresse du début de l'application nommée `_start()` placée au début du segment `.text` (code user). **C'est une convention imposée par le noyau à l'application.**
- Cette fonction est mise dans une section `.start` pour pouvoir la placer avec le ldscript
- `_start()` initialise les variables globales non initialisées, lance `main()` et appelle `exit()`
- L'application accède aux services du noyau via des bibliothèques système (libc) qui encapsulent les appels système (`syscall`).
- L'entrée dans le noyau est `kentry` à l'adresse `0x80000180` quelque soit la cause d'appel (`syscall`, `exception`, `IRQ`), `kentry` analyse `c0_cause.XCODE` puis appelle le bon gestionnaire
- l'instruction `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.
    l'application appelle une fonction **`syscall_fct(a0,a1,a2,a3,code)`** avec les bons arguments
    le gestionnaire de syscall appelle la fonction **`SYSCALL_VECTOR[$2]($4,$5,$6,$7,$2)`**

Conclusion

- Nous avons abordé
  - l'architecture interne d'un petit SoC (`almo1`) de type micro-contrôleur
  - quelques contrôleurs de périphériques (TTY et DMA)
  - les modes d'exécution du MIPS et les raisons du passage de mode
  - le rôle et le partitionnement d'un OS (kO6) (noyau et bibliothèques système)
  - le démarrage du SoC
  - l'un des 3 gestionnaires du noyau (syscall, *interruption*, *exception*)
  - la chaîne de compilation C
  - le simulateur du prototype virtuel du SoC `almo1`
  - Nous reviendrons plusieurs fois sur ces parties en les complexifiant un peu :-)
- Dans le prochain cours, nous reviendrons sur le gestionnaire d'interruptions

Travaux Pratiques

Le but des TME, c'est que vous compreniez bien ce qu'il y a dans un SoC (almo1) et comment fonctionne un petit système d'exploitation (kO6).

ALMO1 

+ kO6 

- Pour le 1er TME, nous allons entrer dans le code de kO6 pour répondre à un grand nombre de petites questions dans le but de vous l'approprier.
- Nous allons vous proposer (imposer) un environnement de programmation.
- Vous allez ensuite écrire un peu de code.