HAL & co
Couches basses du noyau

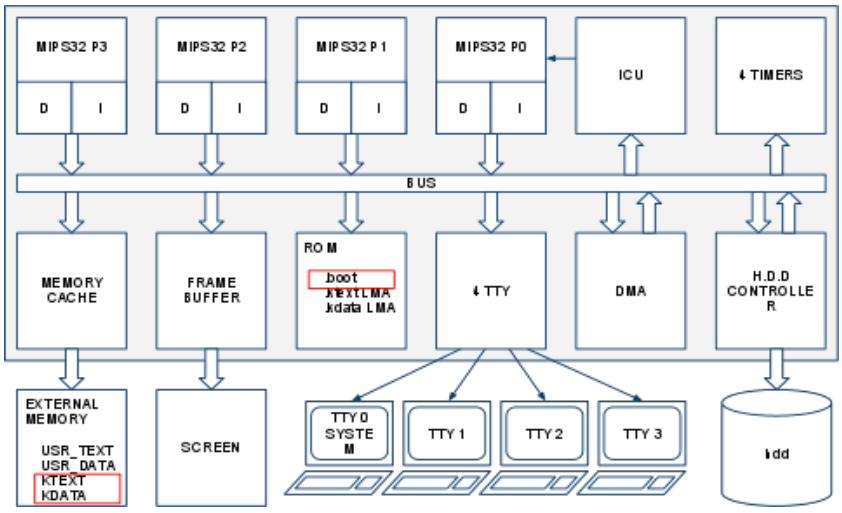
MI074 - 4

plan

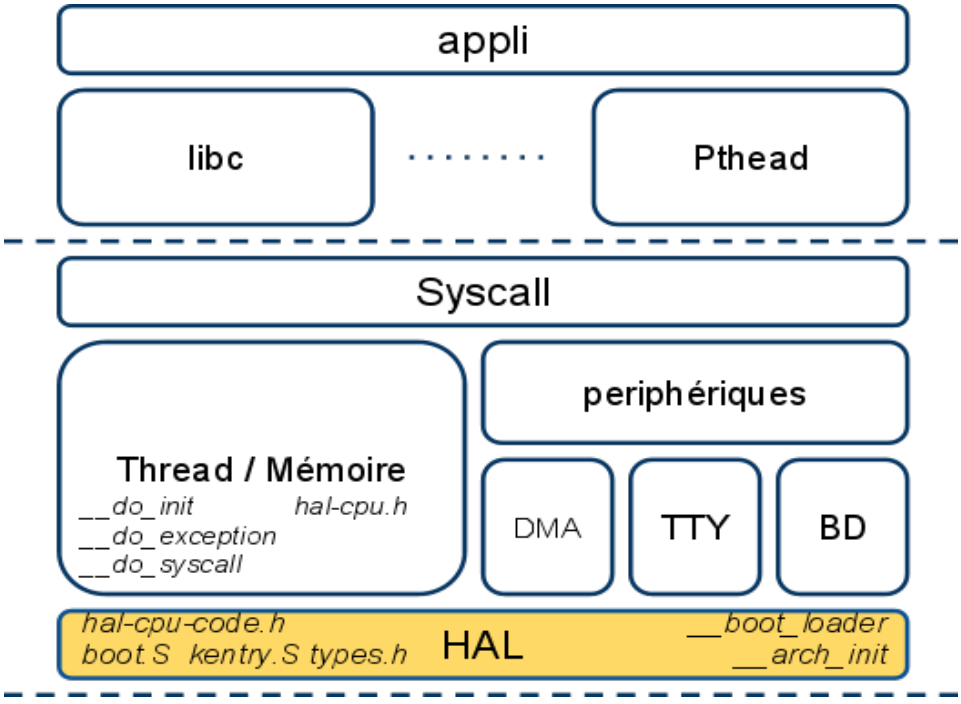
- Organisation du code
- Couche d'abstraction du CPU
- Le kentry et les trois points d'entrée du système
- Travaux à venir

Retour sur le bootloader

Le travail du bootloader permet d'exécuter le code du noyau dans ktext et d'avoir ses données dans kdata.



Architecture de l'OS final



Organisation du code

|-- **arch** HAL architecture
| `-- **soclib** spécifique à la plateforme (p.ex. segmention.h)
|-- **cpu** HAL processeur
| `-- **mipsel** spécifique au mips (p.ex. accès registres)
|-- **drivers** HAL périphérique (p.ex. tty, dma, ...)
|-- **kernel** code du noyau
|-- **libk** bibliothèque C pour le noyau
`-- **mm** allocateur mémoire

Usage actuel des répertoires

```
.
|-- Makefile
|-- apps
|-- build
|-- hdd-img.bin
`-- src_sys
    |-- arch
    |   `-- soclib
    |       |-- devices.h
    |       |-- kldscript.h
    |       `-- segmentation.h
    |-- cpu
    |   `-- mipsel
    |       `-- boot.S
    |-- kernel
    |   |-- __boot_loader.c
    |   |-- __do_init.c
    |   `-- config.h
    |-- libk
    `-- mm
```

Les fichiers actuels doivent se distribuer dans les répertoires de la manière suivante.

Il va falloir modifier le Makefile

HAL : Hardware Abstraction Layer

Comme son nom l'indique, une HAL est définie par un ensemble de structures de données et de fonctions qui permettent d'accéder au matériel de manière uniforme indépendamment du matériel réel.

La HAL est défini à travers plusieurs interfaces

- Interface Kernel / CPU
- Interface Kernel / Arch
- Interface Kernel / Devices

Nous allons voir d'abord l'interface CPU puis les interfaces Arch et Devices dans les prochains cours.

HAL-CPU : redéfinition des types entier

- Tous les processeurs n'ont pas la même largeur de mot, donc la largeur type int dépend du CPU.
- Il faut rédéfinir les types standards dans cpu/mipsel :

  types.h
  - long : mot machine
  - int : 4 octets

```
#ifndef _TYPES_H_
#define _TYPES_H_

#ifndef NULL
#define NULL                (void*)0
#endif

typedef unsigned long       uint_t;
typedef signed long         sint_t;

typedef unsigned char       uint8_t;
typedef signed char         sint8_t;

typedef unsigned short
int  uint16_t;
typedef signed short
int    sint16_t;

typedef unsigned
int        uint32_t;
typedef signed
int          sint32_t;

typedef uint32_t            size_t;
typedef sint32_t            ssize_t;

typedef sint32_t            error_t;
```

Code de la HAL

- L'interface de la HAL est présente dans un .h dans le répertoire kernel.
- Puisque la plupart des services sont courts, ils sont décrits dans des fonctions inlinées, donc dans un .h et pas dans .c
- dans kernel, on a :
  hal-cpu.h
  qui inclue le fichier présent dans cpu/mipsel
  hal-cpu-code.h
- C'est dans le Makefile en fonction d'un paramètre CPU que l'on indique quel hal-cpu-code.h inclure.

HAL : Interface Kernel / CPU

Les services : fonctions inlinées

registres spéciaux
numéro de proc, timestamp, numéro thread.
irq du CPU
masquage, démasquage.
spinlock
trylock, lock, unlock, ...
opérations atomiques
incrément, décrément, addition, ...
caches
invalidation de ligne de cache data
context
création, destruction, chargement, sauvegarde.

HAL : Time, identity & special registers

dans hal-cpu.h

- static inline uint_t cpu_get_id(void);
- static inline uint_t cpu_time_stamp(void);
- static inline struct thread_s *cpu_current_thread(void);
- static inline void cpu_set_current_thread(struct thread_s * thread);

dans hal-cpu-code.h

```
static inline uint_t cpu_get_id(void)
{
    register unsigned int proc_id;
    asm volatile ("mfc0  %0, $0":"=r" (proc_id));
    return proc_id;
}
static inline uint_t cpu_time_stamp(void)
{
    register uint_t cycles;
    asm volatile ("mfc0  %0, $9":"=r" (cycles));
    return cycles;
}
static inline struct thread_s *cpu_current_thread(void)
{
    register void *thread;
    asm volatile ("mfc0 %0, $4, 2":"=r" (thread));
    return thread;
}
static inline void cpu_set_current_thread(struct thread_s *thread)
{
    asm volatile ("mtc0 %0, $4, 2"::"r" (thread));
}
```

HAL : IRQ register/enable/disable/restore

dans hal-cpu.h

- static inline void cpu_disable_single_irq(uint_t irq_num, uint_t * old);
- static inline void cpu_enable_single_irq(uint_t irq_num, uint_t * old);
- static inline void cpu_disable_all_irq(uint_t * old);
- static inline void cpu_enable_all_irq(uint_t * old);
- static inline void cpu_restore_irq(uint_t old);

Notez que seules les interruptions du CPU nous interresse ici. Les interruptions qui passe par l'ICU seront abstraites dans l'architecture.

HAL : Spinlock trylock/unlock/init

Attente actives dans hal-cpu.h

- static inline bool_t cpu_spinlock_trylock(uint_t *lock);
- static inline void cpu_spinlock_lock(uint_t *lock);
- static inline void cpu_spinlock_unlock(uint_t *lock);
- static inline void cpu_spinlock_init(uint_t *lock);
- static inline void cpu_spinlock_destroy(uint_t *lock);

Dans notre cas seule la première fonction est délicate puisqu'elle utilise du code assembleur et les instruction ll et sc.

kentry : l'entrée du système

- On entre dans le système à la suite de 3 événements:
    1. une interruption provenant d'un périphérique.
    2. une exception détectée par le matériel
    3. une demande de service par l'utilisateur

- Pour le moment, on ne va traiter que les deux premiers

- L'entrée du système est à l'adresse kentry
  Le code est spécifique au CPU

kentry : comportement

Dans les cas 1. et 2. l'entrée est imprévisible,
Dans le cas 3. il est prévu

kentry:

test de la cause d'appel
si c'est un **appel système alors**
on change de pile et on appelle la fonction **__do_syscall**
sinon si c'est une **interruption alors**
si on est en mode user **alors** on change de pile **fsi**
on sauve les registres "temporaires" et on appelle la fonction **__do_interrupt**
sinon
on sauve tous les registres et on appelle **__do_exception**
fsi

Résumé

- Après le boot tous les processeurs démarrent __do_init().
- Un processeur va être chargé de demander l'initialisation de l'architecture avec la fonction __arch_init() puis d'initialiser les structures du noyau et de créer les threads de départ.
- Nous allons nous intéresser à ce processeur de démarrage. Les autres vont rester en attente.
- Pour le TME à venir nous allons donc
  - organiser le code dans les répertoires
  - changer le Makefile en conséquence
  - ajouter la HAL incomplete
  - ajouter la fonction __arch_init()
  - ajouter le kentry
  - traiter la première interruption