

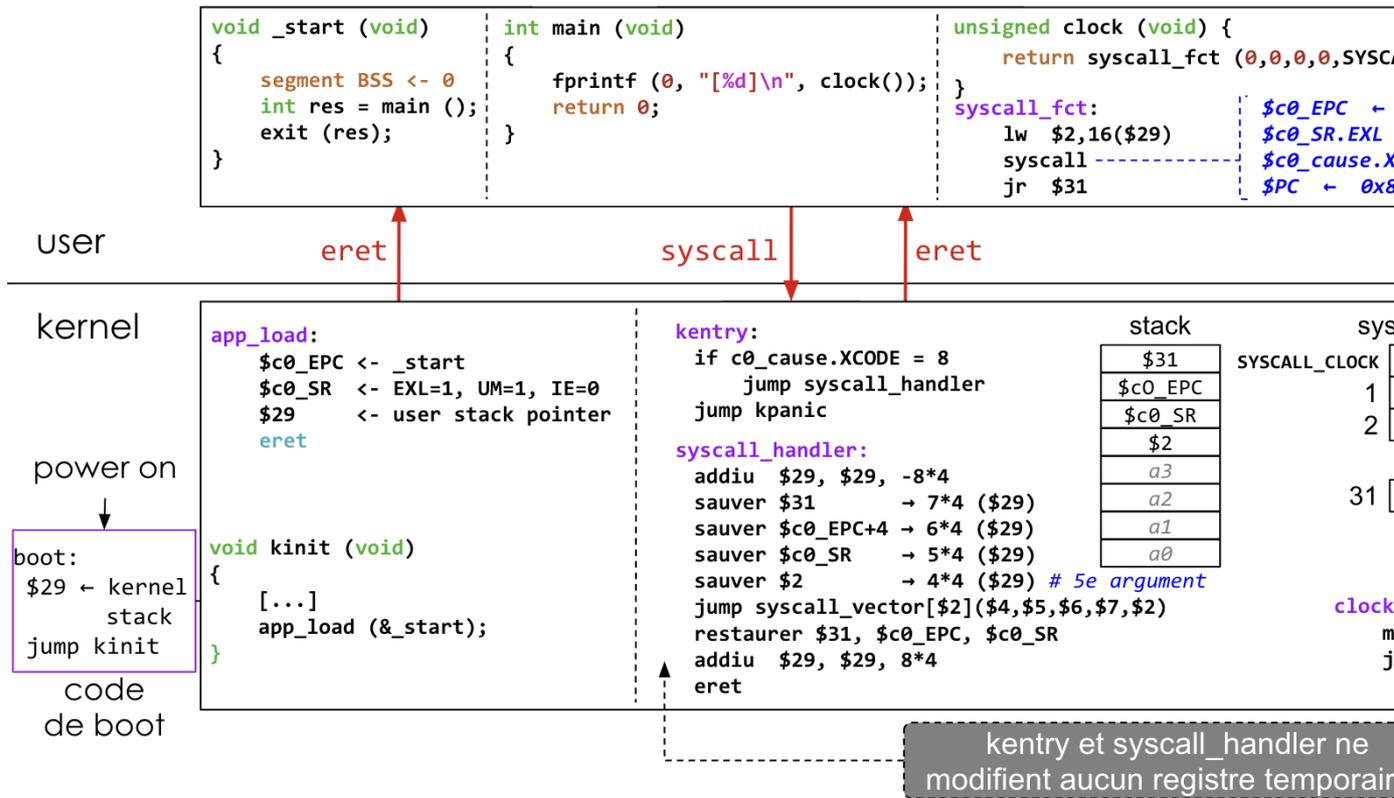
1. Les modes d'exécution du MIPS et les instructions *systeme*
2. Passage entre les modes kernel et user

Application simple en mode utilisateur

Le schéma présenté rapidement au cours 10 (slides 26 à 31) et en détail dans l'annexe du cours 10 (slides 1 à 32) représente l'exécution d'une application utilisateur très simple dont le comportement est défini par la fonction `main()`.

L'exécution part du démarrage du SoC et va jusqu'à l'exécution de la fonction `exit()` qui stoppe l'avancée du programme.

L'objectif de ce schéma est de comprendre les interactions entre le code de boot, le noyau, l'application et les bibliothèques système. Le schéma ci-dessous ne contient pas l'intégralité du code pour des raisons évidentes de lisibilité, mais ce qui reste devrait suffire.



- En bas à gauche, c'est le code de boot qui, ici, se contente d'initialiser la pile d'exécution du noyau et d'entrer dans le noyau par la fonction `kinit()` (kernel init). Ce code s'exécute en mode kernel, mais il ne fait pas partie du noyau car, dans un vrai système, il doit charger le noyau depuis le disque dur, mais, ici, le noyau est déjà en mémoire alors c'est plus simple.
- En bas, c'est le noyau avec la fonction `kinit()` qui initialise les structures de données internes du noyau. Ici, il s'agit juste de mettre les variables globales non initialisées à 0, puis d'appeler la routine `app_load` qui va entrer dans la première fonction de l'application utilisateur nommée `_start()`. Dans le noyau, sur la figure, on voit aussi la routine `kentry` qui est le point d'entrée du noyau pour la gestion des services.

Actuellement, il n'y a que le gestionnaire d'appel système (`syscall`). Son comportement est succinctement résumé.

- En haut, c'est l'application utilisateur, décomposée en trois parties. La première à gauche est la fonction `_start()` appelée par le noyau au tout début de l'application. Cette fonction initialise à 0 les variables globales non initialisées dans le programme, puis elle appelle la fonction `main()`. Si on sort de la fonction `main()` avec un `return`, la fonction `_start` fait l'appel système `exit()`. La seconde partie au centre contient le code de l'utilisateur (*notez que la fonction `main()` ou l'une des fonctions appelées par la fonction `main()` peut demander une sortie anticipée de l'application en appelant directement `exit()`*). Enfin, la troisième partie, à droite, c'est le code des bibliothèques système utilisées par l'application, ce sont elles qui font les appels système, ici, seule la fonction `clock()` est représentée.

Le but de cette séance est de s'intéresser à des points particulier de ce schéma :

- D'abord, nous abordons les 2 modes d'exécution du MIPS, kernel et user, utilisés respectivement pour le noyau et l'application utilisateur.
- Puis, nous voyons les passages du noyau à l'application et de l'application au noyau.
- Ensuite, nous nous intéressons à comment écrire le code C et assembleur pour contrôler le placement en mémoire.
- Enfin, il y a quelques quelques questions sur comment compiler pour faciliter la compréhension des TPs.

1. Les modes d'exécution du MIPS et les instructions système

Dans cette section, nous allons nous intéresser à ce que propose le processeur MIPS concernant les modes d'exécution. Ce sont des questions portant sur l'usage des modes en général et le comportement du MIPS vis-à-vis de ces modes en particulier.

Questions

1. Le MIPS propose deux modes d'exécution, rappelez quels sont ces deux modes, quel est le mode utilisé par le noyau et quel est le mode utilisé par l'application ? (C10 S6+S7)
2. Commencez par rappeler ce qu'est l'espace d'adressage du MIPS et dites ce que signifie «une adresse X est mappée dans l'espace d'adressage du MIPS». Est-ce qu'une adresse X mappée dans l'espace d'adressage du MIPS est toujours accessible (en lecture ou en écriture) quelque soit le mode d'exécution du MIPS. (C10 S7)
3. Le MIPS propose des registres à usage général (GPR *General Purpose Register*) pour les calculs (\$0 à \$31). Le MIPS propose un deuxième banc de registres à l'**usage du système d'exploitation** dans le coprocesseur 0. Chaque registre du coprocesseur 0 porte un nom correspondant à son usage, nous en avons vu 3 en cours (C10 S7+S10 à S14) : `c0_sr`, `c0_cause` et `c0_epc`. Donner leur numéro et leur rôle en une phrase ?
4. Les deux instructions qui permettent de manipuler les registres du coprocesseur 0 sont `mtc0` et `mfc0` (C10 S11). Quelle est leur syntaxe ? réponse dans Documentation MIPS Architecture et assembleur (4.) Est-ce qu'on peut manipuler ces registres de coprocesseur avec d'autres instructions ? Écrivez les instructions permettant de faire `c0_epc = c0_epc + 4` (vous utiliserez le registre GPR §8)

5. Le registre status (`c0_sr` ou $\$12$ du coprocesseur 0) est composé de plusieurs champs de bits qui ont chacun une fonction spécifique.
Décrivez le contenu du registre status et le rôle des bits 0, 1 et 4 de l'octet 0. (C10 S12+S13+S15)
réponse dans Documentation MIPS Architecture et assembleur (6.)
6. Le registre cause (`c0_cause` ou $\$13$ du coprocesseur 0) est contient la *cause d'appel* du kernel.
Dites à quel endroit est stockée cette *cause* et donnez la signification des codes 0, 4 et 8 (C10 S14+S15)
réponse dans Documentation MIPS Architecture et assembleur (7.)
7. Le registre `c0_epc` ($\$14$ du coprocesseur 0) est un registre 32 bits qui contient une adresse. Vous devriez l'avoir décrit dans la question 2.
Expliquez pourquoi, dans le cas d'une exception, ce doit être l'adresse de l'instruction qui provoque une exception qui doit être stockée dans `c0_epc`? (C10 S15)
8. Nous avons vu trois instructions utilisables **seulement** lorsque le MIPS est en mode kernel, lesquelles? Que font-elles?
Est-ce que l'instruction `syscall` peut-être utilisée en mode user? (C10 S11)
9. Quelle est l'adresse d'entrée dans le noyau au démarrage (à la sortie du code de boot) et après (depuis l'application) ? (C10 S15 S20)
10. Que se passe-t-il lorsqu'on entre dans le noyau après de l'exécution de l'instruction `syscall`? (C10 S15)
11. Quelle instruction utilise-t-on pour sortir du noyau afin d'entrer dans l'application ?
Dîtes précisément ce que fait cette instruction dans le MIPS. (C10 S15)

2. Passage entre les modes kernel et user

Le noyau et l'application sont deux exécutables compilés indépendamment mais qui ne sont pas indépendants puisqu'on doit passer du noyau à l'application et inversement. Vous savez déjà que l'application appelle les services du noyau avec l'instruction `syscall`, voyons comment cela se passe vraiment depuis le code C. Certaines questions sont proches de celles déjà posées, c'est volontaire.

Questions

1. Comment imposer le placement d'adresse d'une fonction ou d'une variable en mémoire lorsqu'on produit un programme binaire exécutable, c'est-à-dire quel outil de la chaîne de compilation réalise ce placement en mémoire et avec quel fichier de configuration ? (C9 S18+S22+S23 C10 annexe S6+S8)
2. La première fonction d'un programme utilisateur est la fonction `_start()`, c'est elle qui appelle la fonction `main()`. La fonction `_start()` est donc dans le code de l'application, et non pas dans le noyau. Cependant le noyau doit connaître son adresse afin de pouvoir y sauter et ainsi entrer dans l'application. Dans le code ci-après, nous voyons comment la fonction `kinit()` appelle cette fonction `_start()`. Deux fichiers sont impliqués : `kinit.c` dans lequel se trouve la fonction `void kinit(void)` et `hcpu.S` dans lequel se trouve la fonction `void app_load(void *)` en charge d'appeler la fonction `_start()`.

```

kinit.c:
void kinit (void)
{
    [...]
    extern int _start;           # declaree ailleurs a une adresse connue de l'editeur
    app_load (&_start);        # appel de la fonction app_load definie dans hcpu.S
}

hcpu.S:
.globl app_load
app_load:
    mtc0    $4,          $14      # $4 contient l'argument

```

```

li    $26,    0x12    # $26 <-- 0x12 == 0b0001010
mtc0  $26,    $12    # c0_sr <-- 0x12
la    $29,    __data_end # initialisation du pointeur de pile
eret

```

Comme le noyau et l'application sont deux exécutables compilés indépendamment, il doit y avoir une convention permettant au noyau de savoir quelle est l'adresse de `start()`.

Où se trouve donc la fonction `_start()` et comment le kernel connaît-il son adresse ? (C10 S30+S32)

À quoi sert `.globl app_load` ? (C9 S18 C10 S20)

Quels sont les registres utilisés dans le code de `app_load` ?

Que savez-vous de l'usage de `$26` ? Quels sont les registres modifiés ? Expliquez pour chacun la valeur affectée.

Que fait l'instruction `eret` ? (C10 S15)

3. Que doit-on faire dans la fonction `_start()` avant l'exécution de la fonction `main()` du point de vue de l'initialisation ? Et que doit-on faire dans la fonction `_start()` au retour de la fonction `main()` ? (C10 S24)
4. Nous avons vu que le noyau est sollicité par des demandes de service, quels sont-ils ? Nous rappelons que l'instruction `syscall` initialise le champs `xcode` du registre `c0_cause`, ainsi donc comment le noyau fait-il pour connaître la cause de son appel ? (C10 S25)
5. On rappelle que `$26` et `$27` sont deux registres GPR temporaires *réservés* pour le noyau pour faire des calculs sans qu'il ait besoin de les sauvegarder dans la pile. **Ce ne sont pas des registres du coprocesseur 0** comme `c0_sr` ou `c0_epc`. En effet, l'usage de ces registres (`$26` et `$27`) par l'utilisateur ne provoque pas d'exception du MIPS. Toutefois, si le noyau est appelé alors il modifie ces registres et donc l'utilisateur perd leur valeur.

Le code assembleur ci-après contient les instructions exécutées à l'entrée dans le noyau, quelle que soit la cause. Les commentaires présents dans le code ont été volontairement retirés (ils sont dans le cours et dans les fichiers du TP). La section `.kentry` est placée à l'adresse `0x80000000` par l'éditeur de lien, conformément à ce qui est demandé dans son fichier `ldscript kernel.ld`.

kernel/hcpua.S

```

15 .section    .kentry,"ax"
16 .org      0x180
22
23 kentry:
24
25     mfc0    $26,    $13
26     andi    $26,    $26,    0x3C
27     li     $27,    0x20
28     bne    $26,    $27,    kpanic

```

Ligne 16, la directive `.org DEP` (`.org` pour *origine*, `DEP` pour *déplacement*) permet de placer le pointeur de remplissage de la section courante à `DEP` octets du début de la section, ici `DEP = 0x180`.

Pourquoi faire ça ? Aurait-on pu remplacer le `.org 0x180` par `.space 0x180` ? (C10 S5 et connaissance de l'assembleur) Expliquer les lignes 25 à 28. (C10 S20+S26+S31)

6. Le gestionnaire de `syscall` est la partie du code noyau qui gère l'exécution des services demandés par l'instruction `syscall`.

Pour ce noyau, c'est un code en assembleur présent dans le fichier `kernel/hcpua.S` que nous allons détailler.

Pour vous aider dans la compréhension du code, vous devez vous souvenir que l'instruction `syscall` réalise un peu un appel de fonction:

- la fonction est définie par un *numéro de syscall* contenu dans le registre GPR `$2`;
- les arguments (4 au maximum) sont dans les registres `$4` à `$7`;

- la fonction appelante de syscall n'a pas réservé d'espace dans la pile pour les arguments (il faudra le faire);
- le registre \$2 contient la valeur de retour du syscall.

Le numéro contenu dans le registre \$2 est utilisé par le noyau pour indexer un tableau de pointeurs de fonctions de *syscall* nommé `syscall_vector[]`, ou vecteur de syscalls en français. Ce vecteur de syscalls est défini dans le fichier `kernel/ksyscalls.c`.

Les lignes 36 à 43 du code assembleur (`kernel/hcpua.S`) sont chargées d'allouer de la place dans la pile, nous allons voir pourquoi...

common/syscalls.h

```
1 #define SYSCALL_EXIT          0
2 #define SYSCALL_READ         1
3 #define SYSCALL_WRITE        2
4 #define SYSCALL_CLOCK        3
5 #define SYSCALL_NR           32
```

kernel/ksyscalls.c

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

kernel/hcpua.S

```
34 syscall_handler:
35
36     addiu    $29,    $29,    -8*4
37     mfc0    $27,    $14
38     mfc0    $26,    $12
39     addiu    $27,    $27,    4
40     sw      $31,    7*4($29)
41     sw      $27,    6*4($29)
42     sw      $26,    5*4($29)
43     sw      $2,     4*4($29)
44     mtc0    $0,     $12
45
46     la      $26,    syscall_vector
47     andi    $2,     $2,     SYSCALL_NR-1
48     sll    $2,     $2,     2
49     addu    $2,     $26,    $2
50     lw      $2,     0($2)
51     jalr   $2
52
53     lw      $26,    5*4($29)
54     lw      $27,    6*4($29)
55     lw      $31,    7*4($29)
56     mtc0    $26,    $12
57     mtc0    $27,    $14
58     addiu    $29,    $29,    8*4
59     eret
```

Dessinez l'état de la pile après l'exécution de ces instructions. Que fait l'instruction ligne 44 et quelle conséquence cela a-t-il? Que font les lignes 46 à 51? Et enfin que font les lignes 53 à 59 sans détailler

ligne à ligne. (C10 S26+S31+S34)