

TME

[?Séance 9] [Séance 10] [?Séance 11]

1. A. Travaux dirigés

1. A1. Les modes d'exécution du MIPS
2. A2. Langage C pour la programmation système
3. A3. Passage entre les modes kernel et user
4. A4. Génération du code exécutable

3. B. Travaux pratiques

1. B1. Ajout d'une bibliothèque de fonctions standards pour le kernel (klibc)
2. B2. Programme utilisateur mais exécuté en mode kernel
3. B3. Programme utilisateur utilisé en mode user mais sans libc
4. B4. Ajout de la librairie C pour l'utilisateur

Application simple en mode utilisateur

Cette page décrit la séance complète : partie TD et partie TP. Elle commence par la partie TD avec des questions ou des exercices à faire sur papier, réparties dans 4 sections. Certaines questions de sections différentes sont semblables, c'est normal, cela vous permet de réviser. Puis, dans la partie TP, il y a des questions sur le code avec quelques exercices de codage simples à écrire et à tester sur le prototype. La partie TP est découpée en 4 étapes. Pour chaque étape, nous donnons (1) une brève description avec une liste des objectifs principaux de l'étape, (2) une liste des fichiers avec un bref commentaire sur chaque fichier, (3) une liste de questions simples dont les réponses sont dans le code, le cours ou le TD et enfin (4) un petit exercice de codage.

IMPORTANT

Avant de faire cette séance, vous devez avoir lu les documents suivants :

- Séance de TME sur le démarrage du prototype? : *obligatoire*
- Cours sur l'exécution d'une application en mode user : *obligatoire*
- Document sur l'assembleur du MIPS et la convention d'appel des fonctions : *recommandé*
- Documentation sur le mode kernel du MIPS32 : *obligatoire*

Récupération du code du TP

- Téléchargez **l'archive code du tp2** et placez là dans le répertoire `$HOME/AS5`
- Assurez-vous que vous avez déjà sourcé le fichier `Source-me.sh` (sinon lisez [?Configuration de l'environnement des TP ? Étape 3](#))
- Ouvrez un terminal, allez dans le répertoire `AS5` (`cd ~/AS5`) et décompressez l'archive du tp2 avec **`tar xvzf tp2.tgz`**
Cette étape est peut-être inutile si vous avez déjà fait la décompression de l'archive au moment de son téléchargement.
- Dans le terminal, exécutez la commande `cd ; tree -L 2 AS5`. Vous devriez obtenir ceci (tp1 et tp2):

```
AS5
?? bin
?   ?? almol.x
?   ?? gcc
?   ?? Source-me.sh
```

```

?   ??? test
?   ??? tracelog
??? tp1
?   ??? 1_hello_boot
?   ??? 2_init_asm
?   ??? 3_init_c
?   ??? 4_nttys
?   ??? 5_driver
?   ??? Makefile
??? tp2
    ??? 1_klibc
    ??? 2_appk
    ??? 3_syscalls
    ??? 4_libc
    ??? Makefile

```

Objectif de la séance

Cette séance illustre le [cours2](#). Les applications de l'utilisateur s'exécutent en mode user. Dans la séance précédente, nous avons vu que les registres de commande des contrôleurs de périphériques sont placés dans l'espace d'adressage du processeur. Les adresses de ces registres ont été placées dans la partie de l'espace d'adressage interdite en mode user. Ainsi, une application n'a pas un accès direct aux périphériques, elle doit utiliser des appels système (avec l'instruction `syscall`) pour demander au noyau du système d'exploitation de faire l'accès. C'est ce que nous allons voir.

Le code est désormais découpé en 4 couches logicielles :

- **1_klibc**
? le code de boot (utilisé seulement au démarrage) mais avec une librairie de fonctions standard pour le noyau;
- **2_appk**
? le noyau du système d'exploitation, ici pour l'essentiel, la fonction d'initialisation `kinit()` et une application mais **sans** le gestionnaire des appels systèmes;
- **3_syscalls**
? le noyau **avec** le gestionnaire des appels système et une application **sans** la librairie de fonctions standards (libc);
- **4_libc**
? le noyau avec gestionnaire de `syscall`, une libc, et une application

A. Travaux dirigés

A1. Les modes d'exécution du MIPS

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. Dans la section **A3**, nous verrons le code de gestion des changements de mode dans le noyau.

Questions

1. Le MIPS propose deux modes d'exécution, rappelez quels sont ces deux modes et à quoi ils servent? (*Nous l'avons dit dans le descriptif de la séance*).
2. Commencez par rappeler ce qu'est l'espace d'adressage du MIPS et dites ce que signifie « une adresse X est mappée en mémoire ».

Dites si une adresse X mappée en mémoire est toujours accessible (en lecture ou en écriture) quelque soit le mode d'exécution du MIPS.
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, ce sont les registres système (dit du coprocesseur 0).

Comment sont-ils numérotés? Chaque registre porte un nom correspondant à son usage, quels sont ceux que vous connaissez: donner leur nom, leur numéro et leur rôle? Peut-on faire des calculs avec des registres? Quelles sont les instructions qui permettent de les manipuler?
4. Le registre status est composé de plusieurs champs de bits qui ont chacun une fonction spécifique.

Décrivez le contenu du registre status et le rôle des bits de l'octet 0 (seulement les bits vus en cours).
5. Le registre cause 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
6. Le registre `CO_EPC` est un registre 32 bits qui contient une adresse. Vous devriez l'avoir décrit dans la question 2.

Expliquez pourquoi ce doit être l'adresse de l'instruction qui provoque une exception qui doit être stockée dans `CO_EPC`?
7. 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?
8. Quelle est l'adresse d'entrée dans le noyau?
9. Que se passe-t-il quand le MIPS entre dans le noyau, après l'exécution de l'instruction `syscall`?
10. Quelle instruction utilise-t-on pour sortir du noyau et entrer dans l'application ? Dites précisément ce que fait cette instruction dans le MIPS.

A2. Langage C pour la programmation système

La programmation en C, vous connaissez, mais quand on programme pour le noyau, c'est un peu différent. Il y a des éléments de syntaxe ou des besoins spécifiques.

Questions

1. En assembleur, vous utilisez les sections prédéfinies `.data` et `.text` pour placer respectivement les data et le code ou alors vous pouvez créer vos propres sections avec la directive `.section` (nous avons utilisé cette possibilité pour la section `.boot`). Il est aussi possible d'imposer ou de créer des sections en langage C avec le mot clé `__attribute__`. Ce mot clé du C permet de demander certains comportements au compilateur. Il y a en a beaucoup (si cela vous intéresse vous pouvez regarder dans la [?doc de GCC sur les attributs](#)). En cours, nous avons vu un attribut permettant de désigner ou créer une section dans laquelle est mise la fonction concernée. Quelle était la syntaxe de cet attribut (regardez sur le slide 37).
2. En C, vous savez que les variables globales sont toujours initialisées, soit explicitement dans le programme lui-même, soit implicitement à la valeur 0. Les variables globales initialisées sont placées dans la section `.data` (ou plutôt dans l'une des sections `data: .data, .sdata, .rodata, etc.`) et elles sont présentes dans le fichier objet (`.o`) produit par le compilateur. En revanche, les variables globales non explicitement initialisées ne sont pas présentes dans le fichier objet. Ces dernières sont placées dans un segment de la famille `.bss`. Le fichier `ldscript` permet de mapper l'ensemble des segments en mémoire. Pour pouvoir initialiser à 0 les segments `bss` par programme, il nous faut connaître les adresses de début et de fin où ils

sont placés en mémoire.

Le code ci-dessous est le fichier `ldscript` du kernel `kernel.ld` (nous avons retiré les commentaires mais ils sont dans les fichiers).

Expliquez ce que font les lignes 11, 12 et 15.

```
1 SECTIONS
2 {
3     .boot : {
4         *(.boot)
5     } > boot_region
6     .ktext : {
7         *(.text*)
8     } > ktext_region
9     .kdata : {
10        *(.*data*)
11        . = ALIGN(4);
12        __bss_origin = .;
13        *(.*bss*)
14        . = ALIGN(4);
15        __bss_end = .;
16    } > kdata_region
17 }
```

3. Nous connaissons les adresses des registres de périphériques. Ces adresses sont déclarées dans le fichier `ldscript kernel.ld`. Ci-après, nous avons la déclaration de la variable de `ldscript` `__tty_regs_map`. Cette variable est aussi utilisable dans les programmes C, mais pour être utilisable par le compilateur C, il est nécessaire de lui dire quel type de variable c'est, par exemple une adresse d'entier ou une adresse de tableau d'entiers, Ou encore, une adresse de structure.

Dans le fichier `kernel.ld`:

```
__tty_regs_map = 0xd0200000 ; /* tty's registers map, described in devices.h */
```

Dans le fichier `harch.c` :

```
12 struct tty_s {
13     int write;           // tty's output address
14     int status;         // tty's status address something to read if not null)
15     int read;           // tty's input address
16     int unused;        // unused address
17 };
18
19 extern volatile struct tty_s __tty_regs_map[NTTYS];
```

À quoi servent les mots clés `extern` et `volatile` ?

Si `NTTYS` est une macro dont la valeur est 2, quelle est l'adresse en mémoire

`__tty_regs_map[1].read` ?

4. Certaines parties du noyau sont en assembleur. Il y a au moins les toutes premières instructions du code de boot (démarrage de l'ordinateur) et l'entrée dans le noyau après l'exécution d'un `syscall`. Le gestionnaire de `syscall` est écrit en assembleur et il a besoin d'appeler une fonction écrite en langage C. Ce que fait le gestionnaire de `syscall` est:
- ◆ trouver l'adresse de la fonction C qu'il doit appeler pour exécuter le service demandé;
 - ◆ placer cette adresse dans un registre, par exemple `$2`;
 - ◆ exécuter l'instruction `jal` (ici, `jal $2`) pour appeler la fonction.

Que doivent contenir les registres `$4` à `$7` et comment doit-être la pile?

1. Vous avez appris à écrire des programmes assembleur, mais parfois il est plus simple, voire nécessaire, de mélanger le code C et le code assembleur. Dans l'exemple ci-dessous, nous voyons comment la fonction `kinit()` procède pour entrer dans la fonction placée à l'adresse `__crt0` définie dans le fichier `kernel.ld`.

Remarquez la syntaxe, ici `volatile` permet de dire au compilateur d'insérer le code tel que sans le modifier. Notez aussi l'absence de `,` entre les chaînes de caractères. Le premier argument de `__asm__` est une chaîne de caractères **unique** dans laquelle les instructions sont séparées par de `\n`. Il peut y avoir d'autres arguments, nous n'en aurons pas besoin.

Dans quelle section se trouve l'adresse `__crt0`? Combien vaut-elle? Est-ce que cette valeur est imposée par le processeur MIPS comme l'adresse de boot ou d'entrée dans le kernel? Quelle fonction est à cette adresse? Pourquoi doit-on écrire ce code en assembleur?

```

9 void kinit (void)
10 {
11     kprintf (0, banner);
12
13     // put bss sections to zero. bss contains uninitialised global variables
14     extern int __bss_origin;    // first int of bss section
15     extern int __bss_end;      // first int of above bss section
16     for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);
17
18     // this code allows to exit the kernel to go to user code
19     __asm__ volatile ( "la    $26,    __text_origin \n"    // get first address
20                       "mtc0  $26,    $14 \n"              // put it in c0_EPC
21                       "li    $26,    0b00010010 \n"       // next status {UM,0,
22                       "mtc0  $26,    $12 \n"              // UM <- 1, IE <- 0,
23                       "la    $29,    __data_end \n"        // define new user st
24                       "eret \n"); // j EPC and EXL <- 0
25 }

```

2. Dans le code C de la question précédente, à quoi servent les lignes 14 à 16? Pourquoi faire des déclarations `extern`?

A3. Passage entre les modes kernel et user

Le noyau et l'application sont deux exécutables compilés indépendamment mais pas qui ne sont pas indépendants. 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?
2. Dans la question **A2.5**, nous avons vu comment la fonction `kinit()` appelle la fonction `__start()` grâce à un bout de code en assembleur placé au début de la section `.text`. Nous allons voir maintenant quelles sont les conditions de cet appel. Dans le code de la question **A2.5**, `$26` est un registre de travail pour le kernel. Quels sont les autres registres modifiés? Expliquez pour chacun la valeur affectée.
3. Que faire avant l'exécution de la fonction `main()` du point de vue de l'initialisation? Et au retour de la fonction `main()`?
4. Nous avons vu que le noyau est sollicité par des événements, quels sont-ils? Nous rappelons que l'instruction `syscall` initialise le registre `c0_cause`, comment le noyau fait-il pour connaître la cause de son appel?
5. `$26` et `$27` sont deux registres temporaires que le noyau se réserve pour faire des calculs sans qu'il ait besoin de les sauvegarder dans la pile. Ce ne sont pas des registres système 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 les fichiers du TP). La section `.kentry` est placée à l'adresse `0x80000000` par l'éditeur de lien. La directive `.org` (ligne 16) permet de déplacer le pointeur de remplissage de la section courante du nombre d'octets donnés en argument, ici `0x180`. Pouvez-vous dire pourquoi ? Expliquer les lignes 25 à 28.

kernel/hcpu.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,    not_syscall

```

6. Le gestionnaire de `syscall` est la partie du code qui gère le comportement du noyau lors de l'exécution de l'instruction `syscall`. C'est un code en assembleur présent dans le fichier `kernel/hcpu.S` que nous allons observer. Pour vous aider dans la compréhension de ce code, vous devez imaginer que l'instruction `syscall` est un peu comme un appel de fonction. Ce code utilise un tableau de pointeurs de fonctions nommé `syscall_vector` défini dans le fichier `kernel/ksyscalls.c`. Les lignes 36 à 43 sont chargées d'allouer de la place dans la pile.

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.

common/syscalls.h

```

1 #define SYSCALL_EXIT          0
2 #define SYSCALL_TTY_WRITE    1
3 #define SYSCALL_TTY_READ     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_TTY_READ]    = tty_read,
    [SYSCALL_TTY_WRITE]   = tty_write,
    [SYSCALL_CLOCK]       = clock,
};

```

kernel/hcpu.S

```

34 ksyscall:
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

```

A4. Génération du code exécutable

Pour simuler le logiciel, il faut produire deux exécutables. Nous utilisons, ici, un Makefile hiérarchique et des règles explicites. Cela sort du cadre de l'architecture, mais vous avez besoin de ce savoir-faire pour comprendre le code, alors allons-y.

Questions

1. Rappelez à quoi sert un Makefile?
2. Vous n'allez pas à avoir à écrire un Makefile complètement. Toutefois, si vous ajoutez des fichiers source, vous allez devoir les modifier en ajoutant des règles. Nous avons vu brièvement la syntaxe utilisée dans les Makefiles de ce TP au cours n°1. Les lignes qui suivent sont des extraits de `1_klibc/Makefile` (le Makefile de l'étape1). Dans cet extrait, quelles sont la cible finale, les cibles intermédiaires et les sources? A quoi servent les variables automatiques de make? Dans ces deux règles, donnez-en la valeur.

```

kernel.x : kernel.ld obj/hcpu.o obj/kinit.o obj/klibc.o obj/harch.o
    $(LD) -o $@ -T $^
    $(OD) -D $@ > $@.s

```

```

obj/hcpu.o : hcpu.S hcpu.h
    $(CC) -o $@ $(CFLAGS) $<
    $(OD) -D $@ > $@.s

```

3. Dans le TP, à partir de la deuxième étape, nous avons trois répertoires de sources `kernel`, `ulib` et `uapp`. Chaque répertoire contient un fichier Makefile différent destiné à produire une cible différente grâce à une règle nommée `compil`, c.-à-d. si vous tapez `make compil` dans un de ces répertoires, cela compile les sources locales.

Il y a aussi un Makefile dans le répertoire racine `4_libc`. Dans ce dernier Makefile, une des règles est destinée à la compilation de l'ensemble des sources dans les trois sous-répertoires. Cette règle appelle récursivement la commande `make` en donnant en argument le nom du sous-répertoire où descendre : `make -C <répertoire> [cible]` est équivalent à `cd <répertoire>; make [cible]` ; `cd ..`

Ecrivez la règle `compil` du fichier `4_libc/Makefile`.

```

4_libc/
??? Makefile      : Makefile racine qui invoque les Makefiles des sous-répertoires et q
??? common ?????????? répertoire des fichiers commun kernel / user
??? kernel ?????????? Répertoire des fichiers composant le kernel
?   ??? Makefile  : description des actions possibles sur le code kernel : compilation

```

```

??? uapp ?????????????? Répertoire des fichiers de l'application user seule
?   ??? Makefile      : description des actions possibles sur le code user : compilation et
??? ulib ?????????????? Répertoire des fichiers des bibliothèques système liés avec l'appli
??? Makefile      : description des actions possibles sur le code user : compilation et

```

B. Travaux pratiques

Pour les travaux pratiques, vous devez d'abord répondre aux questions, elles ont pour but de vous faire lire le code et revoir les points du cours et vous guidez, un peu, pour l'exercice. Les réponses sont dans le cours ou dans les fichiers sources. Certaines ont déjà été traitées en TD, c'est normal. Ensuite, vous passez aux exercices pratiques.

Le code se trouve dans `$AS5/tp2/`, ouvrez un terminal et allez-y. Dans ce répertoire, vous avez 4 sous-répertoires et un Makefile. Le fichier `$AS5/tp2/Makefile` permet de faire le ménage en appelant les Makefiles des sous-répertoires avec la cible `clean`.

B1. Ajout d'une bibliothèque de fonctions standards pour le kernel (klibc)

Objectifs de l'étape

Le noyau gère les ressources matérielles et logicielles utilisées par les applications. Il a besoin de fonctions standards pour réaliser des opérations de base, telles qu'une fonction `print` ou une fonction `rand`. Ces fonctions ne sont pas très originales, mais elles recèlent des subtilités que vous ne connaissez peut-être pas encore, vous pouvez les regarder par curiosité. En outre, nous allons utiliser un Makefile définissant un graphe de dépendance explicite entre les fichiers cibles et les fichiers sources avec des règles de construction.

Fichiers

```

1_klibc/
??? kinit.c      : fichier contenant la fonction de démarrage du noyau
??? harch.h     : API du code dépendant de l'architecture
??? harch.c     : code dépendant de l'architecture du SoC
??? hcpu.h      : prototype de la fonction clock()
??? hcpu.S      : code dépendant du cpu matériel en assembleur
??? kernel.ld   : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
??? klibc.h     : API de la klibc
??? klibc.c     : fonctions standards utilisées par les modules du noyau
??? Makefile    : description des actions possibles sur le code : compilation, exécution, ne

```

Questions

1. Ouvrez le fichier Makefile, En ouvrant tous les fichiers dessiner le graphe de dépendance de `kernel.x` vis-à-vis de ses sources?
2. Dans quel fichier se trouvent les codes dépendant du MIPS ?

Exercices

- Le numéro du processeur est dans les 12 bits de poids faible du registre `$15 (c0_cpuid)` du coprocesseur système (à côté des registres `c0_epc`, `c0_sr`, etc.). Ajoutez la fonction `int cpuid(void)` qui lit le registre `c0_cpuid` et qui rend un entier contenant juste les 12 bits de poids faible.

Vous pouvez vous inspirer fortement de la fonction `int clock(void)`. Comme il n'y a qu'un seul processeur dans cette architecture, `cpuid` rend toujours 0.
Ecrivez un programme de test (vous devrez modifier les fichiers `hcpu.h`, `hcpu.S` et `kinit.c`)

B2. Programme utilisateur mais exécuté en mode kernel

Objectifs de l'étape

Nous allons désormais avoir deux exécutables: le noyau et l'application. Dans cette étape, nous allons voir comment le noyau fait pour appeler l'application, alors même que celle-ci n'est pas compilée en même temps que le noyau. Nous allons passer du noyau à l'application à la fin de la fonction `kinit()`.

Nous allons donc entrer dans l'application, en revanche, dans cette étape, nous n'allons pas mettre en place la gestion des syscalls. C'est-à-dire qu'il ne sera pas possible de revenir dans le noyau depuis l'application. C'est bien entendu une étape intermédiaire, parce qu'il faut absolument pouvoir invoquer le noyau depuis l'application pour accéder aux périphériques. Pour pouvoir quand même accéder aux registres de périphériques, nous allons **exceptionnellement** exécuter l'application en mode kernel. Ainsi, l'application pourra accéder aux adresses de l'espace d'adressage réservées au mode `kernel`.

Nous avons deux exécutables à compiler et donc deux `Makefiles` de compilation. Nous avons aussi un `Makefile` qui invoque récursivement les `Makefiles` de compilation.

Fichiers

```
2_appk/
??? Makefile      : Makefile racine qui invoque les Makefiles des sous-répertoires et qui exécute
??? kernel ?????????? Répertoire des fichiers composant le kernel
?   ??? kinit.c   : fichier contenant la fonction de démarrage du noyau
?   ??? harch.h   : API du code dépendant de l'architecture
?   ??? harch.c   : code dépendant de l'architecture du SoC
?   ??? hcpu.h    : prototype de la fonction clock()
?   ??? hcpu.S    : code dépendant du cpu matériel en assembleur
?   ??? klibc.h   : API de la klibc
?   ??? klibc.c   : fonctions standards utilisées par les modules du noyau
?   ??? kernel.ld : ldscript décrivant l'espace d'adressage pour l'édition de liens du kernel
?   ??? Makefile  : description des actions possibles sur le code kernel : compilation et nettoyage
??? user ?????????? Répertoire des fichiers composant l'application user
??? crt0.c        : fonctions d'interface entre kernel et user, pour le moment : crt0()
??? main.c        : fonction principale de l'application
??? user.ld       : ldscript décrivant l'espace d'adressage pour l'édition de liens du user
??? Makefile      : description des actions possibles sur le code user : compilation et nettoyage
```

Questions

1. Combien de fichiers de type `ldscript` avons-nous ?
2. Dans quel fichier se trouve la première fonction de l'application et comment s'appelle-t-elle?
3. Quelle est la fonction du noyau qui appelle cette fonction et dans quel fichier?
4. Comment le noyau fait-il pour démarrer l'application en mode `kernel`? (la réponse est dans la fonction de la question précédente).

Exercice

- Vous n'allez pas faire grand-chose pour cette étape parce qu'elle n'est pas très utile du fait de l'impossibilité de revenir dans le noyau après l'entrée dans l'application. Affichez juste un second message depuis la fonction `main()`

B3. Programme utilisateur utilisé en mode user mais sans libc

Objectifs de l'étape

Le programme utilisateur doit absolument s'exécuter en mode user et il doit passer par des appels système pour accéder aux services du noyau. Les services, ici, sont limités (l'accès au TTY, exit et clock), il n'empêche que pour gérer ces appels, il faut l'analyseur des causes d'appels à l'entrée du noyau et un gestionnaire de `syscall`. Il faut aussi le gestionnaire d'exceptions, parce que s'il y a une erreur de programmation, le noyau doit afficher quelque chose pour aider le programmeur.

Le passage de l'application au noyau par le biais de l'instruction `syscall` impose que les numéros de services soient identiques pour le noyau et pour l'application. Ces numéros de service (comme `SYSCALL_TTY_WRITE`, `SYSCALL_EXIT` sont définis dans le fichier `syscall.h` communs au noyau et à l'application. Ce fichier est mis dans un répertoire à part nommé `common`. Il n'y a qu'un seul fichier ici, mais dans un système plus élaboré, il y en a d'autres.

Fichiers

```

3_syscalls/
??? Makefile      : Makefile racine qui invoque les Makefiles des sous-répertoires et qui exécute
??? common ?????????? répertoire des fichiers commun kernel / user
?   ??? syscall.h : API la fonction syscall et des codes de syscalls
??? kernel ?????????? Répertoire des fichiers composant le kernel
?   ??? kinit.c   : fichier contenant la fonction de démarrage du noyau
?   ??? harch.h   : API du code dépendant de l'architecture
?   ??? harch.c   : code dépendant de l'architecture du SoC
?   ??? hcpu.h    : prototype de la fonction clock()
?   ??? hcpu.S    : code dépendant du cpu matériel en assembleur
?   ??? klibc.h   : API de la klibc
?   ??? klibc.c   : fonctions standards utilisées par les modules du noyau
?   ??? kpanic.h  : déclaration du tableau de dump des registres en cas d'exception
?   ??? kpanic.c  : fonction d'affichage des registres avant l'arrêt du programme
?   ??? ksyscalls.c : Vecteurs des syscalls
?   ??? kernel.ld : ldscript décrivant l'espace d'adressage pour l'édition de liens du kernel
?   ??? Makefile  : description des actions possibles sur le code kernel : compilation et nettoyage
??? user ?????????????? Répertoire des fichiers composant l'application user
??? crt0.c        : fonctions d'interface entre kernel et user, pour le moment : crt0()
??? main.c        : fonction principale de l'application
??? user.ld       : ldscript décrivant l'espace d'adressage pour l'édition de liens du user
??? Makefile      : description des actions possibles sur le code user : compilation et nettoyage

```

Questions

1. Dans quel fichier se trouve la définition des numéros de services tels que `SYSCALL_EXIT` ?
2. Dans quel fichier se trouve le vecteur de `syscall`, c'est-à-dire le tableau `syscall_vector[]` contenant les pointeurs sur les fonctions qui réalisent les services correspondants aux `syscall` ?
3. Dans quel fichier se trouve le gestionnaire de `syscalls` ?

Exercice

- Vous allez ajouter un appel système nommé `SYSCALL_CPUID` qui rend le numéro du processeur. Nous allons lui attribuer le numéro 4 (notez que ces numéros de services n'ont rien à voir avec les numéros utilisés pour le simulateur MARS). Pour ajouter un appel système, vous devez modifier les fichiers : `common/syscalls.h`, `kernel/ksyscall.c`, `kernel/hcpu.S` et `kernel/hcpu.h.cpuid(void)`.

B4. Ajout de la librairie C pour l'utilisateur

Objectifs de l'étape

L'application utilisateur n'est pas censée utiliser directement les appels système. Elle utilise une librairie de fonctions standards (la `libc` POSIX, mais pas seulement) et ce sont ces fonctions qui réalisent les appels système. Toutes les fonctions de la `libc` n'utilisent pas les appels système. Par exemple, les fonctions `int rand(void)` ou `int strlen(char *)` (rendent, respectivement, un nombre pseudoaléatoire et la longueur d'une chaîne de caractères) n'ont pas besoin du noyau. Les librairies font partie du système d'exploitation mais elles ne sont pas dans le noyau.

Le terme « librairie » vient de l'anglais « library » qui signifie bibliothèque. On utilise souvent le mot librairie même si le sens en français n'est pas le même que celui en anglais. Disons que, dans notre contexte, les deux mots sont synonymes.

Normalement, les librairies système sont des « vraies » librairies au sens `gcc` du terme. C'est-à-dire des archives de fichiers objet (`.o`). Ici, nous allons simplifier et ne pas créer une *vraie* librairie, mais seulement un fichier objet `libc.o` contenant toutes les fonctions. Ce fichier objets doit être lié avec le code de l'application.

L'exécutable de l'application utilisateur est donc composé de deux parties : d'un côté, le code de l'application et, de l'autre, le code de la librairie `libc` (+ `crt0`). Nous allons répartir le code dans deux répertoires `uapp` pour les fichiers de l'application et `ulib` pour les fichiers qui ne sont pas l'application, c'est-à-dire la `libc`, le fichier `crt0.c` mais aussi le fichier `ldscript user.ld`.

Fichiers

```
4_libc/
??? Makefile      : Makefile racine qui invoque les Makefiles des sous-répertoires et qui exécute
??? common ?????????? répertoire des fichiers commun kernel / user
?   ??? syscalls.h : API la fonction syscall et des codes de syscalls
??? kernel ?????????? Répertoire des fichiers composant le kernel
?   ??? kinit.c    : fichier contenant la fonction de démarrage du noyau
?   ??? harch.h    : API du code dépendant de l'architecture
?   ??? harch.c    : code dépendant de l'architecture du SoC
?   ??? hcpu.h     : prototype de la fonction clock()
?   ??? hcpu.S     : code dépendant du cpu matériel en assembleur
?   ??? klibc.h    : API de la klibc
?   ??? klibc.c    : fonctions standards utilisées par les modules du noyau
?   ??? kpanic.h   : déclaration du tableau de dump des registres en cas d'exception
?   ??? kpanic.c   : fonction d'affichage des registres avant l'arrêt du programme
?   ??? ksyscalls.c : Vecteurs des syscalls
?   ??? kernel.ld  : ldscript décrivant l'espace d'adressage pour l'édition de liens du kernel
?   ??? Makefile   : description des actions possibles sur le code kernel : compilation et nettoyage
??? uapp ?????????????? Répertoire des fichiers de l'application user seule
?   ??? main.c     : fonction principale de l'application
?   ??? Makefile   : description des actions possibles sur le code user : compilation et nettoyage
??? ulib ?????????????? Répertoire des fichiers des bibliothèques système liés avec l'application
```

```
??? crt0.c      : fonctions d'interface entre kernel et user, pour le moment : crt0()
??? libc.h     : API pseudo-POSIX de la bibliothèque C
??? libc.c     : code source de la libc
??? user.ld    : ldscript décrivant l'espace d'adressage pour l'édition de liens du user
??? Makefile   : description des actions possibles sur le code user : compilation et nettoyage
```

Questions

1. Pour ce petit système, dans quel fichier sont placés tous les prototypes des fonctions de la libc? Est-ce ainsi pour POSIX sur LINUX?

Exercice

- Vous allez juste ajouter la fonction `int cpuid()` dans la librairie `libc`.
- Au premier TP, vous deviez créer un petit jeu 'guess', vous pouvez en faire une application utilisateur, en utilisant cette fois les fonction de la `libc`.