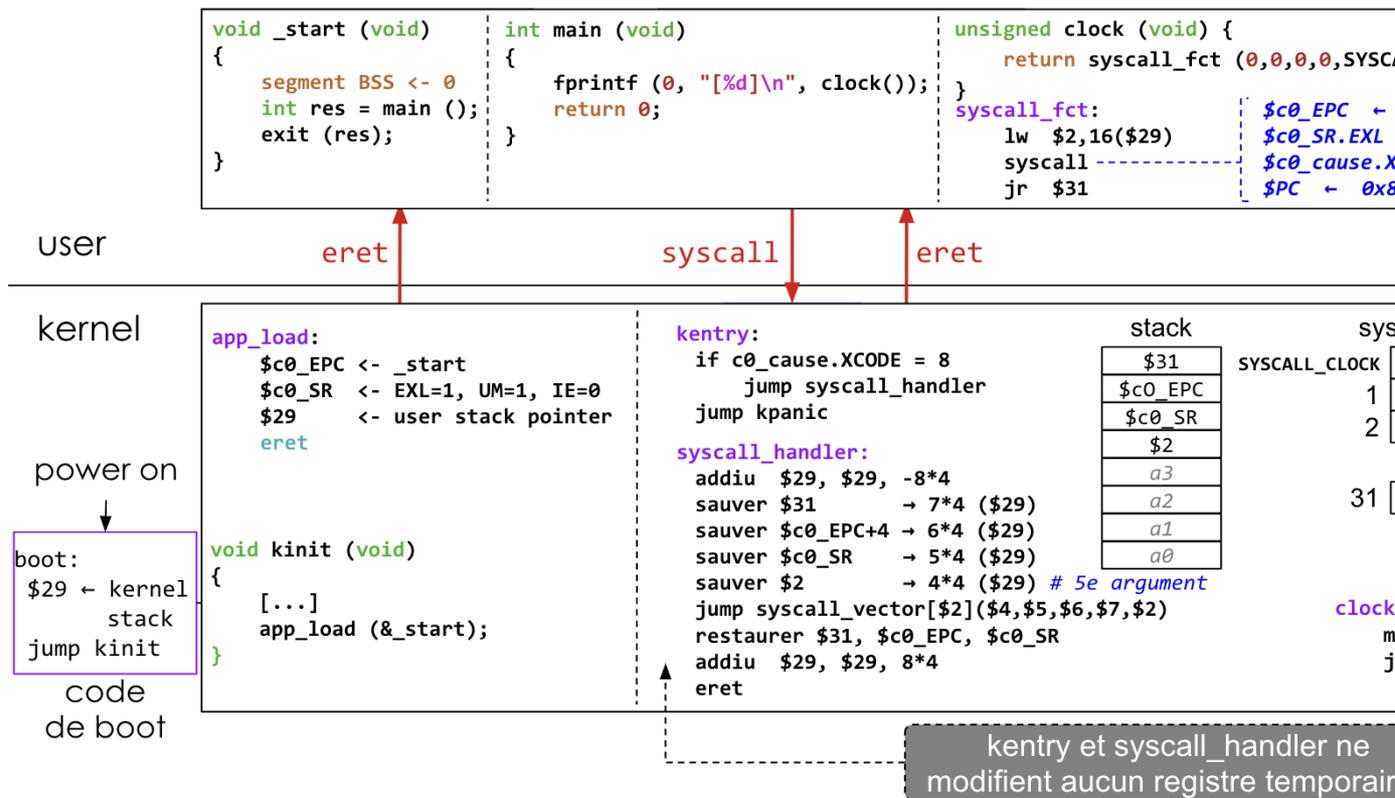


1. 1. Les modes d'exécution du MIPS
2. 2. Passage entre les modes kernel et user
3. 3. Langage C pour la programmation système
4. 4. Génération du code exécutable

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()`, depuis le démarrage du SoC jusqu'à l'exécution de la fonction `exit()`. Son but est de comprendre les interactions entre le code de boot, le noyau, l'application et les bibliothèques système. Le schéma 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 devrait charger le noyau depuis le disque, mais, ici, le noyau est déjà en mémoire.
- 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 nommée `_start()`. Dans le noyau, on voit aussi, la routine `kentry` qui est le point d'entrée du noyau pour la gestion des services, ici, il n'y a que le

gestionnaire d'appel système (`syscall`). Son comportement est succinctement résumé.

- En haut, c'est l'application, décomposée en trois parties. La première à gauche est la fonction `_start()` appelée par le noyau au début de l'application. Cette fonction initialise les variables globales, puis appelle la fonction `main()`. Si on sort de la fonction `main()`, elle fait l'appel système `exit()`. La seconde partie au centre contient le code de l'utilisateur. 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.

1. 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? (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». Dites si une adresse X mappée dans l'espace d'adressage 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, ce sont les registres système (dans le 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? (C10 S5+S11)
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). (C10 S12+S13+S15)
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 (C10 S14+S15)
6. Le registre `C0_EPC` 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)
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? (C10 S11)
8. Quelle est l'adresse d'entrée dans le noyau au démarrage et après ? (C10 S15 S20)
9. Que se passe-t-il quand le MIPS entre dans le noyau, lors de l'exécution de l'instruction `syscall`? (C10 S15)
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. (C10 S15)

2. 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? (C9 S18+S22+S23 C10 annexe S6+S8)
2. Regardons comment la fonction `kinit()` appelle la fonction `__start()`, il y a deux fichiers impliqués `kinit.c` et `hcpu.a.S`, les commentaires ont été retirés. Pour rappel, la fonction `__start()` est la fonction d'entrée de l'application utilisateur, 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.

```
kinit.c:
void kinit (void)
{
    [...]
    extern int __start;
    app_load (&__start);
}

hcpu.a.S:
.globl app_load
app_load:
    mtc0    $4,      $14
    li     $26,     0x12
    mtc0    $26,     $12
    la     $29,     __data_end
    eret
```

Où se trouve 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` ?

3. Que faire avant l'exécution de la fonction `main()` du point de vue de l'initialisation? Et 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 registre `c0_cause`, comment le noyau fait-il pour connaître la cause de son appel? (C10 S25)
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.

kernel/hcpu.a.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

```

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 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/hcpua.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 du code assembleur 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_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 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

```

3. 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. Pour répondre aux questions, vous devez avoir lu les transparents de l'annexe du cours 10, dans lesquels une séquence complète de code (du boot à exit) est détaillée.

Questions

1. En assembleur, vous utilisez les sections prédéfinies `.data` et `.text` pour placer respectivement les data et le code, mais 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 la directive `__attribute__((section("section-name")))`. La directive du C `__attribute__` permet de demander certains comportements au compilateur. Ici, c'est la création d'une section, mais il y a beaucoup d'attributs possibles (si cela vous intéresse vous pouvez regarder dans la [?doc de GCC sur les attributs](#)). Comment créer la section `.start` en C ?
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 (`kentry`) 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, nous utilisons le registre `$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 et le pointeur de pile?

5. 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 `syscall()` est écrite. Cette fonction utilise l'instruction `syscall`.

Deux exemples d'usage de la fonction `syscall()` pris dans le fichier `tp2/4_libc/ulib/libc.c`

```

1 int fprintf (int tty, char *fmt, ...)
2 {
3     int res;
4     char buffer[PRINTF_MAX];
5     va_list ap;
6     va_start (ap, fmt);
7     res = vsnprintf(buffer, sizeof(buffer), fmt, ap);
8     res = syscall (tty, (int)buffer, 0, 0, SYSCALL_TTY_PUTS);
9     va_end(ap);
10    return res;
11 }
12
13 void exit (int status)
14 {
15     syscall( status, 0, 0, 0, SYSCALL_EXIT);           // never returns

```

```
16 }
```

Le code de cette fonction est dans le fichier `tp2/4_libc/ulib/crt0.c`

```
1 //int syscall (int a0, int a1, int a2, int a3, int syscall_code)
2 __asm__ (
3 ".globl syscall      \n"
4 "syscall:           \n"
5 "    lw  $2,16($29)  \n"
6 "    syscall         \n"
7 "    jr  $31         \n"
8 );
```

Combien d'arguments a la fonction `syscall()` ? Comment la fonction `syscall()` reçoit-elle ses arguments ? A quoi sert la ligne 3 de la fonction `syscall()` et que se passe-t-il si on la retire ? Expliquer la ligne 5 de la fonction `syscall()`. Aurait-il été possible de mettre le code de la fonction `syscall()` dans un fichier `.S` ?

4. 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/hcpua.o obj/kinit.o obj/klibc.o obj/harch.o
$(LD) -o $@ -T $^
$(OD) -D $@ > $@.s
```

```
obj/hcpua.o : hcpua.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
```