A. Travaux dirigés
A1. Les modes d'exécution du MIPS
A2. Langage C pour la programmation système
A3. Passage entre les modes kernel et user
A4. Génération du code exécutable

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, mais déjà lu*
- 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/kO6`
- Ouvrez un `terminal`
- Allez dans le répertoire `kO6` : **`cd ~/kO6`**
- Décompressez l'archive du tp2 : **`tar xvzf tp2.tgz`**
- Exécutez la commande : **`cd ; tree -L 1 kO6/tp2`**.
  Vous devriez obtenir ceci :

```
kO6/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 étapes :

- **`1_klibc`**
  → Le code de boot et `kinit()` avec une librairie de fonctions standard pour le noyau;
- **`2_appk`**
  → La fonction d'initialisation `kinit()` appelle une application mais le noyau n'a pas encore le gestionnaire des appels systèmes;
- **`3_syscalls`**
  → Ajout du gestionnaire des appels système et une application **sans** la librairie de fonctions standards utilisateur (libc);
- **`4_libc`**
  → Ajout de la libc (rudimentaire) et d'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*).

   > *Cours 10 / slides 6 et 7*
   >
   > - Il y a le mode kernel et le mode user.
   > - Le mode kernel est utilisé par le noyau alors que le mode user est utilisé par l'application
   > - Le mode kernel permet d'accéder à tout l'espace d'adressage et donc aux périphériques dont les registres sont *mappés* à des adresses accessibles uniquement lorsque le processeur est en mode kernel.

2. Commencez par rappeler ce qu'est l'espace d'adressage du MIPS et dîtes ce que signifie «une adresse X est mappée dans l'espace d'adressage». Dîtes 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.

   > *Cours 10 / slide 7*
   >
   > - L'espace d'adressage du MIPS, c'est l'ensemble des adresses que peut produire le MIPS, il y a 2^{32} adresses d'octets.
   > - On dit qu'une adresse `X` est mappée dans l'espace d'adressage, si cette adresse 'X' est bien dans un segment d'adresses utilisables de l'espace d'adressage `. Autrement dit, le MIPS peut faire des lectures et des écritures à cette adresse, ou encore qu'il y a bien une case mémoire pour cette adresse X`.
   > - Non `X` n'est pas toujours accessible, si `X < 0x80000000` elle est bien accessible quelque-soit le mode d'exécution du MIPS, mais si `X >= 0x80000000` alors `X` n'est accessible que si le MIPS est en mode kernel.

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?

   > *Cours 10 / slides 7, 8 et 9*

- Les registres système sont numérotés de $0 à $31, comme les registres GPR, ce qui peut induire une certaine confusion, parce qu'avec cette syntaxe, si on demande que trouve-t-on dans le registres `$14` ? Si on ne précise pas qu'il s'agit du registre `$14` du coprocesseur 0, alors on ne peut pas répondre. C'est pour cette raison qu'il est préférable d'utiliser leur nom (`EPC` ou `c0_epc` pour `$14` par exemple ou alors `c0_$14`)
- Nous avons vu 6

`c0_sr`	`$12`	contient essentiellement le mode d'exécution du MIPS et le bit d'autorisation des interruptions
`c0_cause`	`$13`	contient la cause d'appel du noyau
`c0_epc`	`$14`	contient l'adresse de l'instruction ayant provoqué l'appel du noyau ou l'adresse de l'instruction suivante
`c0_bar`	`$8`	contient l'adresse mal formée si la cause est une exception due à un accès non aligné (p.ex. lw a une adresse non multiple de 4)
`c0_count`	`$9`	contient le nombre de cycles depuis le démarrage du MIPS
`c0_procid`	`$15`	contient le numéro du processeur (utile pour les architectures multicores)

- non, il n'est pas possible de faire des calculs sur ces registres.
- On peut juste les lire et les écrire en utilisant les instructions `mtc0` et `mfc0`

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).

Cours 10 / slides 10 et 11

0	IE	Interrupt Enable	0 → interruptions masquées 1 → interruptions autorisées si ERL et EXL sont tous les deux à 0
1	EXL	EXception Level	1 → MIPS en mode exception à l'entrée dans le kernel, le MIPS est en mode kernel, interruptions masquées
2	ERL	ERror Level	1 → MIPS en mode Erreur Fatale par exemple, si le MIPS a une exception au boot (on ne verra pas ça) le MIPS est en mode kernel, interruptions masquées
4	UM	User Mode	0 → MIPS en mode kernel 1 → MIPS en mode user si ERL et EXL sont tous les deux à 0

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

Cours 10 / slide 12

- Le champ `XCODE` qui contient le code de la cause d'entrée dans le noyau est codé sur 4 bits entre les bits 2 et 5.
- Les valeurs les plus importantes sont 0 et 8 (interruption et syscall). Les autres valeurs sont des exceptions, c'est-à-dire des fautes faites par le programme.

0	0000_b	interruption	un contrôleur de périphérique à lever un signal IRQ
4	0100_b	ADEL	lecture non-alignée (p. ex. `lw` a une adresse impaire)
8	1000_b	syscall	exécution de l'instruction `syscall`

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` ?

Cours 10 / slide 13

- Une exception, c'est une erreur du programme, telle qu'une division par 0, une lecture non alignée ou une instruction illégale. Il est important que le gestionnaire d'exception sache quelle est l'instruction fautive. C'est pour cette raison que le registre EPC contient l'adresse de l'instruction fautive. Le gestionnaire pourra lire l'instruction et éventuellement corriger le problème.
- A titre indicatif, ce n'est pas la question, mais pour les syscall, c'est aussi l'adresse de l'instruction `syscall` qui est stockée dans `C0_EPC` , or pour le retour de `syscall` , on souhaite aller à l'instruction suivante. Il faut donc incrémenter la valeur de `C0_EPC` de 4 (les instructions font 4 octets) pour connaître l'adresse de retour.

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?

Cours 10 / slide 9

- Les trois instructions sont

`mtc0 $GPR, $C0`	`M`ove `T`o `C`oprocessor `0`	`$GPR` → COPRO_0(`$C0`)
`mfc0 $GPR, $C0`	`M`ove `F`rom `C`oprocessor `0`	`$GPR` ← COPRO_0(`$C0`)
`eret`	`E`xpection `RET`urn	`PC` ← `EPC` ; `c0_sr.EXL` ← `0`

  Attention à l'ordre des registres dans les instructions. L'ordre est toujours le même, c'est d'abord le registre $GPR puis le registre $C0, le sens de l'échange est défini par l'opcode de l'instruction (move `TO` ou move `FROM` coprocessor 0).

- Bien sûr que `syscall` peut être utilisé en mode user, puisque c'est comme ça qu'on entre dans le kernel pour les demandes de services.

8. Quelle est l'adresse d'entrée dans le noyau?

Cours 10 / slide 13

- C'est `0x80000180` . Il n'y a qu'une adresse pour toutes les causes `syscall` , exception et interruption.
- (slides 16 et 17) Il y a aussi l'adresse de la fonction `kinit()` qui est la fonction appelée par le code de boot (à l'adresse `0xBFC00000`) pour entrer dans le noyau.

9. Que se passe-t-il quand le MIPS entre dans le noyau, lors de l'exécution de l'instruction `syscall` ?

Cours 10 / slide 13

- L'instruction `syscall` induit beaucoup d'opérations élémentaires dans le MIPS:
  - `EPC` ← `PC` (adresse de l'instruction `syscall`)
  - `c0_SR.EXL` ← `1` (ainsi les bits `c0_SR.UM` et `c0_SR.IE` ne sont plus utilisés)
  - `c0_cause.XCODE` ← `8`
  - `PC` ← `0x80000180`

10. Quelle instruction utilise-t-on pour sortir du noyau et entrer dans l'application ? Dîtes précisément ce que fait cette instruction dans le MIPS.

Cours 10 / slide 13

- C'est l'instruction `eret` qui permet de sortir du noyau. C'est la seule instruction permettant de sortir du noyau.
  - `PC` ← `EPC`
  - `c0_SR.EXL` ← `0` (ainsi les bits `c0_SR.UM` et `c0_SR.IE` sont à nouveau utilisés)

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. Pour répondre aux questions, vous devez avoir lu les transparents 33 à 53 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 ?

   > *Cours 10 / slide 38*
   >
   > - `__attribute__ ((section (".start")))`
   >   La syntaxe est un peu curieuse avec les doubles underscore et les doubles parenthèses.

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 pas 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 }
   ```

   > *Cours 10 / slide 28*
   >
   > - La ligne 11 contient `. = ALIGN(4)`, c'est équivalent à la directive `.align 4` de l'assembleur. Cela permet de déplacer le pointeur de remplissage de la section de sortie courante (c'est-à-dire ici `.kdata`) sur une frontière de 2^4 octets (une adresse multiple de 16). Cette contrainte est liée aux caches que nous ne verrons pas ici.
   > - La ligne 12 permet de créer la variable de ldscript `__bss_origin` et de l'initialiser à l'adresse courante, ce sera donc l'adresse de début de la zone `bss`.
   > - La ligne 15 permet de créer la variable `__bss_end` qui sera l'adresse de fin de la zone `bss` (en fait c'est la première adresse qui suit juste `bss`.

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` ?

   > *Cours 10 / slide 53*
   >
   > - `extern` : informe le compilateur que la variable définie existe ailleurs. Grâce à son type, le compilateur sait s'en servir.
   > - `volatile` : informe le compilateur que la variable peut changer de valeur toute seule et que donc il doit toujours accéder en mémoire à chaque fois que le programme le demande. Il ne peut donc pas optimiser les accès mémoire en utilisant les registres.
   > - `__tty_regs_map` est un tableau à 2 cases (puisque `NTTYS` = `2`).
   >   Chaque case est une structure de 4 entiers, donc `0x10` octets (16 octets).
   >   `read` est le troisième champ, c'est le troisième entier de la structure, donc en `+8` par rapport au début.
   >   En conséquence `__tty_regs_map[1].read` est en `0xd0200018`

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?

   > *Cours 10 / slide 42*

- C'est un appel de fonction, il faut donc respecter la convention d'appel des fonctions
  - Les registres `$4` à `$7` contiennent les arguments de la fonction
  - Le pointeur de pile doit pointer sur la case réservée pour le premier argument et les cases suivantes sont réservées arguments suivants.
  - Ce n'est pas rappelé ici, mais, **pour l'application user**, il y a **au plus** 4 arguments (entier ou pointeur) pour tous les syscalls. Le gestionnaire de syscall ajoute un cinquième argument avec le numéro de service qu'il a reçu dans `$2`. En conséquence, le pointeur de pile pointe au début d'une zone vide de 4 entiers suivi d'un 5e avec le numéro du service.
  - L'intérêt d'ajouter le numéro de service comme cinquième argument, c'est qu'il est possible de faire une fonction unique qui gère un ensemble de syscalls avec un `switch/case` sur le numéro de service. On ne le fait pas dans cette version.

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` ?

Cours 10 / slide 40

- La fonction `syscall()` a 5 a arguments
- Elle reçoit ses 4 premiers arguments dans les registres $4 à $7 et le 5e (le numéro de service) dans la pile.
- La ligne 3 sert à dire que syscall est une étiquette utilisée dans un autre fichier. `.globl` signifie **glob**al **l**abel. Si on la retire, il y aura un problème lors de l'édition de lien. `syscall()` ne sera pas trouvé par l'éditeur de liens.
- Le noyau attend le numéro de service dans `$2`. Or le numéro du service est le 5e argument de la fonction `syscall()`. La ligne 5 permet d'aller le chercher dans la pile.
- oui, ce code de la fonction `syscall()` qui fait appel à l'instruction `syscall` aurait pu être mis dans un fichier en assembleur, mais cela aurait demandé d'avoir un fichier de plus, pour une seule fonction. Dans une version plus évoluée du système, il y aura un d'autres fonctions assembleur, alors on créera un fichier assembleur pour les réunir.

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?

Cours 9 / slide 24 et Cours 10 / slides 64 et 65

- C'est l'éditeur de lien qui est en charge du placement en mémoire du code et des données, et c'est dans le fichier ldscript `kernel.ld` ou `user.ld` que le programmeur peut imposer ses choix.
- Pour placer une fonction à une place, la méthode que vous avez vu consiste
  - à créer une section grâce à la directive `.section` en assembleur ou à la directive `__attribute__((section()))` en C
  - puis à positionner la section créée dans la description des `SECTIONS` du ldscript.

2. Regardons comment la fonction `kinit()` appelle la fonction `__start()`, il y a deux fichiers impliqués `kinit.c` et `hcpua.S`, les commentaires ont été rétirés.

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

hcpua.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 ? À quoi sert `.globl app_load` ? 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` ?

- La fonction `_start` est au début de la section `.text` (code de l'utilisateur). Le noyau connait cette adresse parce qu'elle est définit dans son fichier `ldscript`.
- `.globl app_load` est nécessaire parce que ce label de fonction est défini dans le fichier `hcpua.S` mais il est utilisé dans un autre (`kinit.c`).
- Les registres utilisés par `app_load` sont `$4`, `$26`, `$29` du banc GPR et `$12` (`c0_sr`) et `$14` (`c0_epc`) du banc de registres système.
- `$26` est un registre temporaire pour le noyau, il peut l'utiliser sans le sauver avant et donc sans le restaurer.
- Il y a 4 registres affectés, dans l'ordre :
  - Le registre système `$14` nommé `c0_epc`, il reçoit l'adresse `_start`, c'est-à-dire l'adresse de la fonction `_start()`.
  - `$26` affecté à `0x12`, c'est un registre temporaire pour le noyau.
  - Le registre système `$12` nommé `c0_sr`, il reçoit la valeur `0x12`, donc les bits `UM`, `EXL` et `IE` prennent respectivement les valeurs `1`, `1` et `0`
    - UM = 1 et IE = 0, signifie que l'on est normalement en mode `user` avec les interruptions masquées, **mais** comme `EXL`=`1`, alors on reste en mode `kernel` avec interruptions masquées.
  - Le registre GPR `$29` reçoit l'adresse de la première adresse **après** la section `.data`. C'est le haut de la pile.
- L'exécution de l'instruction `eret` mettra `EXL` à `0` pour rendre les bits `UM` et `IE` actifs et passer en mode `user` (ici avec interruptions masquées).

3. Que faire avant l'exécution de la fonction `main()` du point de vue de l'initialisation? Et au retour de la fonction `main()`?

Cours 10 / slide 38

- Comme dans la fonction `kinit()`, il faut explicitement initialiser les variables globales non initialisées dans le programme C.
- Si on sort de la fonction `main()`, l'application s'achève. Cela signifie qu'il faut appeler la fonction `exit()` qui effectue l'appel système SYSCALL_EXIT. Cette appel est réalisé au cas où l'application n'aurait pas explicitement exécuté `exit()`. Dans ce cas la valeur rendue par l'application est la valeur de 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?

Cours 10 / slide 17

- Il y en a 3 (si on excepte le signal `reset` qui redémarre tout le système:
  1. Les appels système donc l'exécution de l'instruction `syscall`.
  2. Les exceptions donc les "erreur" de programmation (division par 0, adressage mémoire incorrect, etc.).
  3. Les interruptions qui sont des demandes d'intervention provenant des périphériques.
- L'instruction `syscall` initialise les 4 bits `XCODE` du registre `c0_cause` avec un code indiquant la raison de l'entrée dans le noyau. Le noyau doit analyser ce champ `XCODE`.

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. Ligne 16, la directive `.org DEP` (`.org` pour `origine`) permet de placer le pointeur de remplissage de la section courante à `DEP` octets du début de la section, ici `DEP = 0x180`. Aurait-on pu remplacer le `.org 0x180` par `.space 0x180` ? Expliquer les lignes 25 à 28.

`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,    not_syscall
```

Cours 10 / slide 41 (mais il n'y a pas ces détails)

- La section `kentry` est placée à l'adresse `0x80000000` or l'entrée du noyau est `0x80000180` (l'entrée du noyau est l'adresse à laquelle le processeur *saute* lors de l'exécution `syscall`), il faut donc déplacer le pointeur de remplissage de la section `ktentry` de `0x180`. La directive `.space 0x180` réserve `0x180`, si on met cette directive au tout début de la section, c'est équivalent.
- Commentaire du code
  - Ligne 25 : `$26` ← `c0_cause`
    → donc le registre `$26` GPR réservé au kernel prend la valeur du registre de cause.
  - Ligne 26 : `$26` ← `$26 & 0b00111100`
    → C'est un masque qui permet de ne conserver que les 4 bits du champ `XCODE`.
  - Ligne 27 : `$27` ← `0b00100000`
    → On initialise le registre GPR réservé au kernel $27 avec la valeur attendue dans $26 s'il s'agit d'une cause `syscall`.
  - Ligne 28 : si `$26` ≠ `$27` goto not_syscall
    → Si ce n'est pas un `syscall`, on va plus loin, sinon on continue en séquence.

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[]` définit 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_TTY_PUTC    1
 3 #define SYSCALL_TTY_GETC    2
 4 #define SYSCALL_TTY_PUTS    3
 5 #define SYSCALL_TTY_GETS    4
 6 #define SYSCALL_CLOCK       5
 7 #define SYSCALL_NR          32
```

`kernel/ksyscalls.c`

```
void *syscall_vector[] = {
    [0 ... SYSCALL_NR - 1] = unknown_syscall,
    [SYSCALL_EXIT]      = exit,
    [SYSCALL_TTY_PUTC]  = tty_putc,
    [SYSCALL_TTY_GETC]  = tty_getc,
    [SYSCALL_TTY_PUTS]  = tty_puts,
    [SYSCALL_TTY_GETS]  = tty_gets,
```

```
    [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
```

Cours 10 / slide 42

- État de la pile après l'exécution des lignes 36 à 43

```
      +----------+
      |    $31   |  Nous allons exécuter jal un peu plus et perdre $31, il faut le sauver
      +----------+
      |  C0_EPC  |  C'est l'adresse de retour du syscall
      +----------+
      |  C0_SR   |  le registre status est modifié plus loin, il faut le sauver pour le restaurer
      +----------+
      |    $2    |  C'est le numéro de syscall qui pourra être accédé par la fonction appelée en 5e argument
      +----------+
      |          |  place réservée pour le 4e argument actuellement dans $7
      +----------+
      |          |  place réservée pour le 3e argument actuellement dans $6
      +----------+
      |          |  place réservée pour le 2e argument actuellement dans $5
      +----------+
$29 → |          |  place réservée pour le 1e argument actuellement dans $4
      +----------+
```

- L'instruction ligne 44 met `0` dans le registre `c0_sr`. Ce qui a pour conséquence de mettre à `0` les bits `UM`, `EXL` et `IE`. On est donc en mode kernel avec interruptions masquées.
  - *Notez qu'interdire les interruptions pendant l'exécution des syscall est un choix important. Pour le moment, ce n'est pas un problème puisque nous ne traitons pas les interruptions, mais si nous les traitions, elles seraient masquées. En conséquence, il serait interdit aux fonctions qui traitent les appels système d'exécuter des attentes longues (comme une boucle qui attend le changement d'état d'un registre de périphérique) car sinon, le noyau serait figé (plus rien ne bougerait). Nous verrons comment faire au prochain cours.*

- Commentaire du code lignes 46 à 53
  - Ligne 46 : `$26` ← l'adresse du tableau syscall_vector
    → On s'apprête à y faire un accès indexé par le registre `$2`
  - Ligne 47 : `$2` ← `$2 & 0x1F`
    → pour éviter de sortir du tableau si l'utilisateur à mis n'importe quoi dans `$2`.
    On ne fait pas un modulo et donc `SYSCALL_NR` doit être une puissance de 2 !
  - Ligne 48 : `$2` ← `$2 * 4`
    → Les cases du tableau sont des pointeurs et font 4 octets
  - Ligne 49 : `$2` ← `$26 + $2`
    → `$2` contient désormais l'adresse de la case contenant la fonction correspondante au service n° `$2`
  - Ligne 50 : `$2` ← MEM[`$2`]
    → $2 contient l'adresse de la fonction à appeler
  - Ligne 51 : `jal $2`
    → appel de la fonction de service
    On rappelle que `$4` à `$7` contiennent les 4 premiers argument, mais qu'il y a de place pour ces arguments dans la pile.

- Les lignes 53 à 59 restaurent l'état des registres `$31`, `c0_status`, `c0_epc` et le pointeur de pile puis on sort du noyau avec l'instruction `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?

   Cours 9 / slide 26

   - Le rôle principal d'un Makefile est de décrire le mode d'emploi pour construire un fichier dit **`cible`** à partir d'un ou plusieurs fichiers **`source`** (dits de dépendance) en utilisant des commandes du `shell`. Ce rôle pourrait tout aussi bien être occupé par un script `shell` et d'ailleurs, dans le premier TP, nous avons vu un usage du Makefile dans lequel nous avions rassemblé plusieurs scripts `shell` sous forme de règles.
   - Le second rôle d'un Makefile est de permettre la reconstruction partielle du fichier **`cible`** lorsque quelques fichiers **`source`** changent (pas tous). Pour ce rôle, le Makefile exprime toutes les étapes de construction de la **`cible`** finale et des **`cibles`** intermédiaires sous forme d'un arbre dont les feuilles sont les fichiers **`sources`**.

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
```

Cours 9 / slides 27 et 56

- La `cible` finale est : `kernel.x`
- Les `cibles` intermédiaires sont : `kernel.ld`, `obj/hcpua.o`, `obj/kinit.o`, `obj/klibc.o` et `obj/harch.o`.
- La `source` est : `hcpua.S`
- Les variables automatiques servent à extraire des noms dans la définition de la dépendance (`cible : dépendances`)
  - dans la première règle :
    - `$@` = `cible` = `kernel.x`
    - `$^` = l'ensemble des dépendances = `kernel.ld`, `obj/hcpua.o`, `obj/kinit.o`, `obj/klibc.o` et `obj/harch.o`
  - dans la seconde règle :
    - `$@` = `cible` = `obj/hcpua.o`
    - `$<` = la première des dépendances = `hcpua.S`

3. Dans le TP, à partir de la deuxième étape, nous avons trois répertoires de sources `kernel`, `ulib` et `uapp`. Chaque répertoire contient une 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 qui exécute
├── common ──────── répertoire des fichiers commun kernel / user
├── kernel ──────── Répertoire des fichiers composant le kernel
│   └── Makefile    : description des actions possibles sur le code kernel : compilation et nettoyage
├── uapp ────────── Répertoire des fichiers de l'application user seule
│   └── Makefile    : description des actions possibles sur le code user : compilation et nettoyage
└── ulib ────────── Répertoire des fichiers des bibliothèques système liés avec l'application user
    └── Makefile    : description des actions possibles sur le code user : compilation et nettoyage
```

Ce n'est pas dit dans le cours, mais la question contient la réponse...

```
compil:
    make -C kernel compil
    make -C ulib   compil
    make -C uapp   compil
```

Dernière modification le 14 déc. 2021 à 05:19:23)