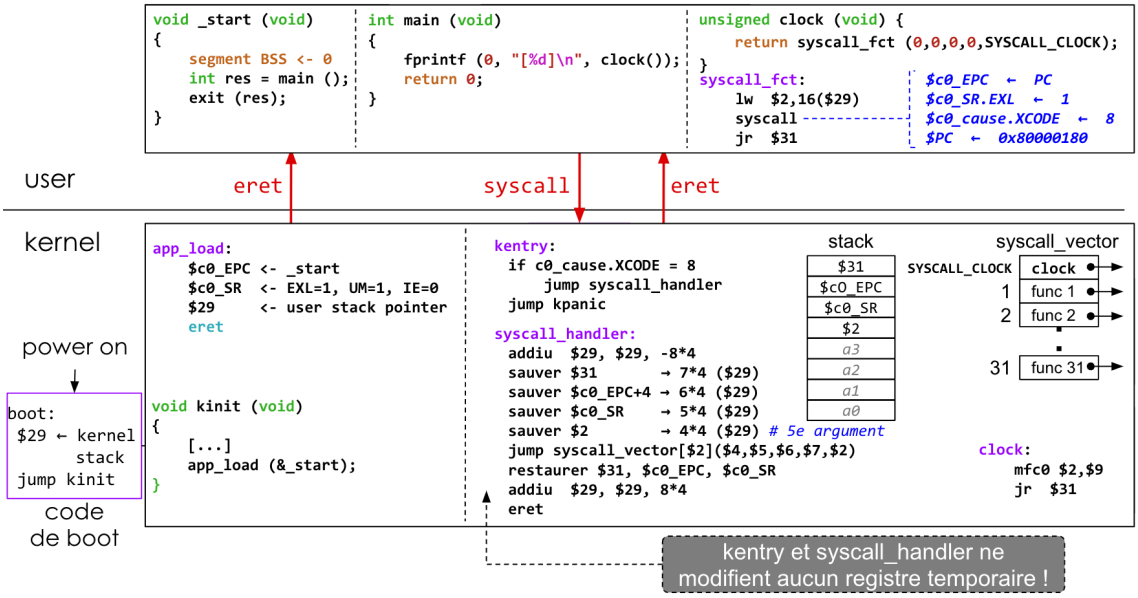
Application simple en mode utilisateur

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)

   > - 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 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)

   > - 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** 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 ?

   - Les registres du coprocesseur `0` 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 ce qui se trouve dans le registres `$14` sans préciser 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 les 3 principaux

`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

   - Il y en a d'autres, dont certains seront utilisés plus tard

`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)

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`)

`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`)

   - 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 $C0, le sens de l'échange est défini par l'opcode de l'instruction (move `TO` ou move `FROM` coprocessor 0).
     `$C0` peut être `c0_sr` (i.e. $12) ou `c0_cause` (i.e. $13) ou `c0_epc` (i.e. $14)
   - non, il n'est pas possible d'utiliser d'autres instructions pour les manipuler, on peut juste les lire et les écrire en utilisant les instructions `mtc0` et `mfc0`

   - `c0_epc = c0_epc + 4`

     ```
     mfc0    $8, $14
     addiu   $8, $8, 4
     mtc0    $8, $14
     ```

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.)

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
4	UM	User Mode	0 → MIPS en mode kernel 1 → MIPS en mode user, seulement si ERL et EXL sont tous les deux à 0

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.)

   - 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 codes les plus importantes à connaitre sont 0 et 8 (interruption et syscall). Les autres codes sont pour les 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`

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)

   - Une exception, c'est dû à une erreur du programme, telle que la lecture d'un mot à une adresse non mappée, 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 `c0_epc` contient l'adresse de l'instruction fautive. Le gestionnaire d'exceptions dans le noyau 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 la vraie adresse de retour du `syscall`.

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)

- Les trois instructions sont `mtc0`, `mfc0` (déjà vues au dessus) et `eret`

`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`

- 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.

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)

   - Au démarrage, le boot saute à l'adresse de la fonction `kinit()` pour entrer dans le noyau.
   - En dehors du démarrage, c'est `0x80000180`. Il n'y a qu'une adresse pour toutes les causes `syscall`, exceptions et interruptions.

10. Que se passe-t-il lorsqu'on entre dans le noyau après de l'exécution de l'instruction `syscall` ? (C10 S15)

    - L'instruction `syscall` induit 4 opérations élémentaires dans le MIPS:
      - `c0_epc` ← `PC` (sauvegarde dans `c0_epc` 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` (c'est la cause `syscall`)
      - `PC` ← `0x80000180` (c'est l'adresse d'entrée dans le noyau)
    - Ces 4 opérations élémentaires sont réalisées par l'instruction `syscall` !

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)

    - C'est l'instruction `eret` qui permet de sortir du noyau.
      C'est la seule instruction permettant de sortir du noyau.
      - `PC` ← `c0_epc` (c'est *l'équivalent* du `jr $31` pour sortir d'une fonction)
      - `c0_sr.EXL` ← `0` (ainsi les bits `c0_sr.UM` et `c0_sr.IE` sont à nouveau utilisés)
    - Ces 2 opérations élémentaires sont réalisées par l'instruction `eret` !

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)

   - C'est l'éditeur de lien qui est en charge du placement en mémoire du code et des données, et c'est dans les fichiers ldscript `kernel.ld` ou `user.ld` que le programmeur peut imposer ses choix de placement dans l'espace d'adressage.
   - Pour placer une fonction à une adresse précise, la méthode que vous avez vu consiste
     - à créer une section grâce à la directive `.section` en assembleur ou grâce à la directive `__attribute__((section()))` pour les programmes C, dans les deux cas le programmeur choisit un nom de section.
     - puis à positionner la section ainsi créée dans la description des `SECTIONS` du fichier ldscript concerné.

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 `hcpua.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 de lien
        app_load (&_start);         # appel de la fonction app_load definie dans hcpua.S
    }

hcpua.S:
    .globl app_load
    app_load:
        mtc0    $4,     $14         # $4 contient l'argument
        li      $26,    0x12        # $26 <--  0x12 == 0b00010010
        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)

   - La fonction `_start()` est au début de la section `.text` (qui contient le code de l'utilisateur). Le noyau connait cette adresse parce qu'elle est définit dans son fichier ldscript `kernel.ld`.

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

   - `.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`). Il faut donc le rendre e`xtern`.

   Quels sont les registres utilisés dans le code de `app_load` ?
   Que savez-vous de l'usage de `$26` ?

- 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 du coprocesseur `0`.
- `$26` est un registre GPR temporaire pour le noyau, il peut l'utiliser sans le sauver avant et donc sans le restaurer.

Quels sont les registres modifiés ? Expliquez pour chacun la valeur affectée.

- Il y a 4 registres affectés, dans l'ordre :
  - Le registre du coprocesseur 0 `$14` nommé `c0_epc`, il reçoit l'adresse `_start`, c'est-à-dire l'adresse de la fonction `_start()`.
  - `$26` affecté par `0x12` ($26 c'est un registre temporaire pour le noyau, on peut l'écraser sans sauver sa valeur).
  - Le registre du coprocesseur 0 `$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.
      Ici les interruptions sont masquées même quand exécute l'application parce que le noyau ne contient pas encore le gestionnaire des interruptions. En effet, on construit le noyau par morceau et le gestionnaire des interruptions est vu au cours suivant.
  - Le registre GPR `$29` reçoit l'adresse de la première adresse en haut de la région `data_region`. C'est le haut de la pile pour l'utilisateur.

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

- 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 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)

   - Comme dans la fonction `kinit()`, il faut explicitement initialiser les variables globales non initialisées dans le programme C. En effet, le programmeur suppose que les variables globales non initialisées explicitement dans le programme C sont à 0. Comme ces variables ne sont pas explicitement initialisées, elles n'occupent pas de place dans le fichier exécutable, on sait juste ou elles sont placées en mémoire.
   - 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 la fonction `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 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)

   - 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. 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 *déplassement*) 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)

   - 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.

   Expliquer les lignes 25 à 28. (C10 S20+S26+S31)

   - Commentaire du code
     - Ligne 25 : `$26` ← `c0_cause`
       → donc le registre GPR `$26` réservé au kernel prend la valeur du registre de cause.
     - Ligne 26 : `$26` ← `$26 & 0b00.1111.00`
       → C'est un masque qui permet de ne conserver que les 4 bits du champ `XCODE`.
     - Ligne 27 : `$27` ← `0b 00.1000.00`
       → 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 'kpanic'
       → Si ce n'est pas un `syscall`, on va à la fonction `kpanic`, sinon on continue en séquence.

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:
- sauf que la fonction est définie par un *numéro de syscall* contenu dans le registre GPR `$2`;
- les arguments sont bien dans les registres $4 à $7, mais il y en a 4 au maximum;
- toutefois, la fonction appelante de syscall n'a pas réservé d'espace dans la pile pour les arguments, il faudra le faire;
- enfin, 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)

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

```
      +----------+
      |    $31   |  Nous allons exécuter jal et perdre $31, il faut le sauver
      +----------+
      |  C0_EPC  |  C'est l'adresse de retour du syscall
      +----------+
      |  C0_SR   |  le registre status est modifié, il faut le sauver pour le restaurer
      +----------+
      |    $2    |  numéro de syscall qui peut être lu par la fonction appelée (5e arg)
      +----------+
      |          |  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`.

3. Langage C pour la programmation système

La programmation en C, vous connaissez, mais quand on programme pour le noyau il y a des éléments de syntaxe ou des besoins spécifiques que vous ne connaissez peut-être pas. Pour répondre aux questions, vous devez avoir lu les transparents de l'annexe du cours 10, dans lesquels une séquence complète de code est détaillée du boot à exit.

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 ? (C10 S30 C10 annexe S8)

   - `__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).

   ```
   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 }
   ```

   Expliquez ce que font les lignes 11, 12 et 15 ? (C10 S32)

   - 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];
```

Si `NTTYS` est une macro dont la valeur est `2`, quelle est l'adresse en mémoire `__tty_regs_map[1].read` ?

- `__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 de la structure, c'est un entier, donc en `+8` par rapport au début de la strucrure.
  En conséquence `__tty_regs_map[1].read` est en `0xd0200018`

À quoi servent les mots clés `extern` et `volatile` ? (C10 annexe S23 et connaissance du C)

- `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.

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? (Connaissance assembleur)

   - 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, ...) // tty identifiant du terminal
2  {                                     // fmt chaine format, suivie d'arguments optionnels
3      int res;
4      char buffer[PRINTF_MAX];
5      va_list ap;
6      va_start (ap, fmt);                                  // définit le dernier argument non-optionnel
7      res = vsnprintf(buffer, sizeof(buffer), fmt, ap); // remplit le buffer avec la chaîne à afficher
8      res = syscall (tty, (int)buffer, 0, 0, SYSCALL_TTY_PUTS);  // ←  appel système
9      va_end(ap);
10     return res;
11 }
12
13 void exit (int status)
14 {
15     syscall( status, 0, 0, 0, SYSCALL_EXIT);                  // ← appel système
16 }
```

Le code de la fonction `syscall()` en **assembleur** est dans le fichier **C** : `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` ? (C10 S31)

- 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 d'autres fonctions assembleur, alors on créera un fichier assembleur pour les réunir.

4. Génération du code exécutable (optionnel)

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? (C9 annexe S5 à S7)

   - 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`**. Les commandes d'une règle ne sont exécutées que si la date de la cible est plus ancienne que la date de l'une des sources dont elle dépend.

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. Les lignes qui suivent sont des extraits de `1_klibc/Makefile` (le Makefile de l'étape-1). 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. (C9 annexe S5 à S7)

   ```
   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
   ```

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

   ```
   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
   ```

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

Dernière modification le 27 nov. 2023 à 13:59:07)