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 5 sections. Certaines questions de sections différentes sont semblables, c'est normal. 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, (2) une liste des objectifs principaux de l'étape, (3) une liste des fichiers avec un bref commentaire sur chaque fichier, (4) une liste de questions simples dont les réponses sont dans le code, le cours ou le TD et enfin (5) un exercice de codage.

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

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

Récupération du code du TP

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

```
AS5
??? bin
?   ??? almo1.x
?   ??? gcc
?   ??? Source-me.sh
?   ??? test
?   ??? tracelog
??? tp1
?   ??? 1_hello_boot
?   ??? 2_init_asm
?   ??? 3_init_c
?   ??? 4_nttys
?   ??? 5_driver
?   ??? Makefile
??? tp2
```

```
??? 1_klibc
??? 2_appk
??? 3_syscalls
??? 4_libc
??? Makefile
```

Objectif de la séance

Cette séance illustre le cours2. Les applications de l'utilisateur s'exécutent en mode user. Dans la séance précédente, nous avons vu que les registres de commande des contrôleurs de périphériques sont placés dans l'espace d'adressage du processeur. Les adresses de ces registres ont été placées dans la partie de l'espace d'adressage interdite en mode user. Ainsi, une application n'a pas un accès direct aux périphériques, elle doit utiliser des appels système (avec l'instruction syscall) pour demander au noyau du système d'exploitation. C'est ce que nous allons voir. Le code est désormais découpé en 4 couches logicielles :

1. `1_klibc` le code de boot (utilisé seulement au démarrage);
2. `2_appk` le noyau du système d'exploitation, ici pour l'essentiel, la fonction d'initialisation `kinit()` et le gestionnaire des appels systèmes;
3. `3_syscalls` la bibliothèque de fonctions standards (libc);
4. `4_libc` l'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. Dans la section A3, nous verrons le code de gestion des changements de mode; Le MIPS propose deux modes d'exécution.

Questions

1. Le MIPS propose deux modes d'exécution, rappelez quels sont ces deux modes et à quoi ils servent? (*Nous l'avons dit dans le descriptif de la séance*).
2. Commencez par rappeler ce qu'est l'espace d'adressage du MIPS, puis dîtes ce que veut dire qu'une adresse X mappée en mémoire, et enfin dîtes si une adresse X mappée en mémoire est toujours accessible (en lecture ou en écriture) quelque soit le mode d'exécution du MIPS.
3. Le MIPS propose des registres à usage général (GPR *General Purpose Register*) pour les calculs ($0 à $31). Le MIPS propose un deuxième banc de registres à l'usage du système d'exploitation, ce sont les registres système. Comment sont-ils numérotés? Chaque registre porte un nom correspondant à son usage, quels sont ceux que vous connaissez, donner leur nom, leur numéro et leur rôle? Peut-on faire des calculs avec des registres? Quelles sont les instructions qui permettent de les manipuler?
4. Le registre status est composé de plusieurs champs de bits qui ont chacun une fonction spécifique. Décrivez le contenu du registre status et le rôle des bits de l'octet 0(seulement ceux vu en cours).
5. Le registre cause est contient la cause d'appel du kernel. Dites à quel endroit est stockée cette cause et donnez la signification des codes 0, 4 et 8
6. Le registre `EPC` est un registre 32 bits qui contient une adresse. Vous devriez l'avoir décrit dans la question 2, mais expliquez pourquoi ce doit être l'adresse de l'instruction qui provoque une exception qui doit être

stocké dans `EPC`?

7. Nous avons vu trois instructions ne sont pas utilisable lorsque le MIPS est en mode kernel, lesquelles? Que font-elles? Est-ce que `syscall` peut-être utilisée en mode user?
8. Quelle est l'adresse d'entrée dans le noyau?
9. Que se passe-t-il quand le MIPS entre dans le noyau, après l'exécution de l'instruction `syscall`?
10. Quelle instruction utilise-t-on pour sortir du noyau et entrer dans l'application ? Dîtes précisément ce que fait cette instruction dans le MIPS?

A2. Langage C pour la programmation système

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

Questions

1. En assembleur, vous utilisez les sections prédéfinies `.data` et `.text` pour placer respectivement les data et le code ou alors vous pouvez créer vos propres sections avec la directive `.section` (nous avons utilisé cette possibilité pour la section `.boot`). Il est aussi possible d'imposer ou de créer des sections en langage C avec le mot clé `__attribute__`. Ce mot clé du C permet de demander certains comportements au compilateur. Il y a en a beaucoup (si cela vous intéresse vous pouvez regarder dans la ?doc de GCC sur les attributs. En cours, nous avons vu un attribut permettant de désigner ou créer une section dans laquelle est mise la fonction concernée. Quelle était la syntaxe de cet attribut (regardez sur le slide 37).
2. En C, vous savez que les variables globales sont toujours initialisées, soit explicitement dans le programme lui-même, soit implicitement à la valeur `0`. Les variables globales initialisées sont placées dans la section `.data` (ou plutôt dans l'une des sections `data` : `.data`, `.sdata`, `.rodata`, etc.) et elles sont présentes dans le fichier objet (`.o`) produit 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`. C'est grâce au fichier ldscript que nous pouvons mapper l'ensemble des segments en mémoire.
   Pour pouvoir initialiser à `0` les segments `bss` par programme, il nous faut connaître l'adresse de début et de fin en mémoire. Le code ci-dessous est le fichier ldscript du kernel `kernel.ld` (nous avons retier les commentaires pour la circonstance.
   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. Est-ce une adresse d'entier? Est-ce une adresse de tableau d'entiers? Ou encore, est-ce une 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 au moins les toutes premières instructions du code de boot (démarrage de l'ordinateur) et l'entrée dans le noyau après l'exécution d'un syscall. Dans ce dernier cas, le gestionnaire de syscall écrit en assembleur a besoin d'appeler une fonction écrite en langage C. Le gestionnaire de syscall trouve l'adresse de la fonction C qu'il doit appeler puis il place cette adresse dans un registre, par exemple `$2`. Il suffit qu'il exécute l'instruction `jal $2` pour appeler la fonction. Que doivent contenir les registres `$4` à `$7` et comment doit-être la pile?
5. Vous avez appris à écrire des programmes assembleur, mais parfois il est plus simple, voire il est nécessaire de mélanger le code C et le code assembleur. Dans l'exemple ci-dessous, nous voyons comment la fonction `kinit()` procède pour entrer dans la fonction placée à l'adresse `__crt0` définie dans le fichier `kernel.ld`. Remarquez la syntaxe, ici `volatile` permet de dire au compilateur d'insérer le code tel que sans le modifier. Notez aussi l'absence de `,` entre les chaînes de caractères. Le premier argument de `__asm__` est une chaîne de caractères unique dans laquelle les instructions sont séparées par de `\n`. Il peut y avoir d'autres arguments, nous ne les verrons pas.
   Dans quelle section se trouve l'adresse `__crt0`? Combien vaut elle? Est-ce que cette valeur est imposée par le processeur MIPS comme l'adresse de boot ou d'entrée dans le kernel? Quelle fonction est à cette adresses? Pourquoi doit-on écrire ce code en assembleur?

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

6. Dans le code C de la question précédente, à quoi servent les lignes 12 à 16? Pourquoi faire des déclarations `extern`?

A3. Passage entre les modes kernel et user

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

Questions

1. Comment imposer le placement d'adresse d'une fonction ou d'une variable en mémoire?
2. Dans la question **A2.5**, nous avons vu comment la fonction `kinit` appelle la fonction `__start()` grâce à un bout de code en assembleur. Nous allons voir maintenant quelles sont les conditions de cet appel. Dans le code de la question **A2.5**, `$26` est un registre de travail pour le kernel. Quels sont les autres registres modifiés? Expliquez pour chacun la valeur affectée.
3. Que faire avant et après l'exécution de la fonction `main()` du point de vue de l'initialisation?
4. Nous avons vu que le noyau est sollicité par des événements, quels sont-ils?Quel est le comportement exact de l'instruction `syscall`? Comment le noyau fait-il pour connaitre la cause de son appel?
5. `$26` et `$27` sont deux registres temporaires que le noyau se réserve pour faire des calcul 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 `$26` et `$27` par l'utilisateur ne provoque pas d'exception du MIPS, mais si le noyau est appelé, il modifie ces registres et donc l'utilisateur perd leur valeur. Le code assembleur ci-après contient les instructions exécutées par le noyau quelque-soit la cause. Les commentaires présents dans le code ont été volontairement retirés. La section `.kentry` sera placée à l'adresse `0x80000000` par l'éditeur de lien. La directive `.org` (ligne 16) permet de déplacer le pointeur de remplissage de la section courante du nombre d'octets donnés en argument, ici `0x180`. Pouvez-vous dire pourquoi ? Expliquer les lignes 25 à 28.

   kernel/hcpu.S

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

6. Le gestionnaire de syscall est la partie du code qui gère le comportement du noyau lors de l'exécution de l'instruction `syscall`. C'est un code en assembleur présent dans le fichier `kernel/hcpu.S` que nous allons observer. Pour vous aider dans la compréhension de ce code, vous devez imaginer que l'instruction `syscall` est un peu comme un appel de fonction. Ce code utilise un tableau de pointeurs de fonctions nommé `syscall_vector` définit dans le fichier `kernel/ksyscalls.c`. Les lignes `36` à `43` sont chargées d'allouer dans la pile. Dessinez l'état de la pile après l'exécution de ces instructions. Que fait l'instruction lige `44` et quelle conséquence cela a? Que font les lignes `46` à `52`? Et enfin que font les lignes `54` à `60`

   kernel/ksyscalls.c

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

kernel/hcpu.S

```
34 ksyscall:
 35
 36     addiu   $29,    $29,    -8*4
 37     mfc0    $27,    $14
 38     mfc0    $26,    $12
 39     addiu   $27,    $27,    4
 40     sw      $31,    7*4($29)
 41     sw      $27,    6*4($29)
 42     sw      $26,    5*4($29)
 43     sw      $2,     4*4($29)
 44     mtc0    $0,     $12
 45
 46     la      $26,    syscall_vector
 47     andi    $2,     $2,     SYSCALL_NR-1
 48     sll     $2,     $2,     2
 49     addu    $2,     $26,    $2
 50     lw      $2,     ($2)
 51     li      $26,    0xFF00
 52     jalr    $2
 53
 54     lw      $26,    5*4($29)
 55     lw      $27,    6*4($29)
 56     lw      $31,    7*4($29)
 57     mtc0    $26,    $12
 58     mtc0    $27,    $14
 59     addiu   $29,    $29,    8*4
 60     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 là quoi sert un Makefile?
2. Comment appeler un makefile depuis une autre Makefile?
3. comment décrire une règle explicite?
4. comment utiliser les variables automatiques du C?

B. Travaux pratiques

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

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

B1. Ajout d'une bibliothèque de fonctions standard pour le kernel (klic)

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

Objectifs

- Ajouter une bibliothèque de fonctions standards
- Utiliser un Makefile avec des règles explicites

Fichiers

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

Questions

1. En ouvrant tous les fichiers dessiner le graphe de dépendance de `kernel.x` vis-à-vis de ses sources?
2. ?

Exercices

- Ajout de la fonction cpuid() qui lit le registre $15 du coprocesseur système.

B2. Programme utilisateur mais exécuté en mode kernel

Nous allons désormais avoir deux exécutables: le noyau et l'application. Dans cette étape, nous allons voir comment le noyau fait pour appeler l'application, alors que celle-ci n'est pas compilée en même temps que le noyau.

Objectifs

Fichiers

```
2_appk/
??? Makefile        : Makefile racine qui invoque les Makefiles des sous-répertoires et qui exéc
??? kernel ?????????? Répertoire des fichiers composant le kernel
?   ??? kinit.c     : fichier contenant la fonction de démarrage du noyau
?   ??? harch.h     : API du code dépendant de l'architecture
?   ??? harch.c     : code dépendant de l'architecture du SoC
```

```
?   ??? hcpu.h      : prototype de la fonction clock()
?   ??? hcpu.S      : code dépendant du cpu matériel en assembleur
?   ??? klibc.h     : API de la klibc
?   ??? klibc.c     : fonctions standards utilisées par les modules du noyau
?   ??? kernel.ld   : ldscript décrivant l'espace d'adressage pour l'édition de liens du kernel
?   ??? Makefile    : description des actions possibles sur le code kernel : compilation et nett
??? user ??????????? Répertoire des fichiers composant l'application user
    ??? crt0.c      : fonctions d'interface entre kernel et user, pour le momment : crt0()
    ??? main.c      : fonction principale de l'application
    ??? user.ld     : ldscript décrivant l'espace d'adressage pour l'édition de liens du user
    ??? Makefile    : description des actions possibles sur le code user : compilation et nettoy
```

Questions

1. Question ?

B3. Programme utilisateur utilisé en mode user mais sans libc

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

Objectifs

Fichiers

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

Questions

1. Question ?

B4. Accès aux registres de contrôle des terminaux `TTY`

Objectifs

Fichiers

```
4_libc/
??? Makefile        : Makefile racine qui invoque les Makefiles des sous-répertoires et qui exéc
??? common ?????????? répetoire des fichiers commun kernel / user
?   ??? syscalls.h  : API la fonction syscall et des codes de syscalls
??? kernel ?????????? Répertoire des fichiers composant le kernel
?   ??? kinit.c     : fichier contenant la fonction de démarrage du noyau
?   ??? harch.h     : API du code dépendant de l'architecture
?   ??? harch.c     : code dépendant de l'architecture du SoC
?   ??? hcpu.h      : prototype de la fonction clock()
?   ??? hcpu.S      : code dépendant du cpu matériel en assembleur
?   ??? klibc.h     : API de la klibc
?   ??? klibc.c     : fonctions standards utilisées par les modules du noyau
?   ??? kpanic.h    : déclaration du tableau de dump des registres en cas d'exception
?   ??? kpanic.c    : fonction d'affichage des registres avant l'arrêt du programme
?   ??? ksyscalls.c : Vecteurs des syscalls
?   ??? kernel.ld   : ldscript décrivant l'espace d'adressage pour l'édition de liens du kernel
?   ??? Makefile    : description des actions possibles sur le code kernel : compilation et nett
??? uapp ???????????? Répertoire des fichiers de l'application user seule
?   ??? main.c      : fonction principale de l'application
?   ??? Makefile    : description des actions possibles sur le code user : compilation et nettoy
??? ulib ???????????? Répertoire des fichiers des bibliothèques système liés avec l'application
    ??? crt0.c      : fonctions d'interface entre kernel et user, pour le momment : crt0()
    ??? libc.h      : API pseudo-POSIX de la bibliothèque C
    ??? libc.c      : code source de la libc
    ??? main.c      : fonction principale de l'application
    ??? user.ld     : ldscript décrivant l'espace d'adressage pour l'édition de liens du user
    ??? Makefile    : description des actions possibles sur le code user : compilation et nettoy
```

Questions

1. Question ?