Boot et premier programme en mode kernel

Pour les travaux pratiques, vous devez d'abord répondre à des questions qui 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 questions ont déjà été traitées en TD, c'est normal. Ensuite, vous passez aux exercices pratiques.

La partie pratique est découpée en 5 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.

Vous allez compiler vos programmes pour le MIPS et les exécuter le prototype virtuel du SoC nommé **almo1** présenté en cours. Il est composé d'un MIPS, d'une mémoire et d'un contrôleur de terminal TTY. Le simulateur d'**almo1** se présente sous la forme d'un exécutable qui simule le comportement du SoC. Le MIPS exécute le programme se trouvant dans les bancs de mémoire du SoC. La simulation est complète depuis le signal reset et précise au cycle et au bit (cela signifie que si c'était une vraie machine, vous auriez exactement le même comportement de vos programmes).

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

- Description des objectifs de cette séance et des suivantes : *obligatoire*
- Cours de démarrage présentant l'architecture matérielle et logicielle que vous allez manipuler *obligatoire*
- Éléments d'information sur les outils de la chaîne de compilation *recommandé*
- Document sur l'assembleur du MIPS et la convention d'appel des fonctions : *recommandé*
- Documentation sur le mode kernel du MIPS32 : *optionnel pour cette séance*
- Configuration de l'environnement des TP : *obligatoire si vous êtes sur votre machine personelle, **sinon lisez seulement la suite***

Mise en place de l'environnement des TP

- **Configuration et test de l'environnement de travail sur les machines de la PPTI**
    - Éditez votre fichier `$HOME/.bashrc` et ajoutez au début:
      `source /Infos/lmd/2024/licence/ue/LU3IN029-2024oct/kO6/bin/SourceMe.sh`
      Ce script modifie quelques variables d'environnement telle que PATH qui permet de définir les répertoires dans lesquels le shell bash trouve ses exécutables (ici, les outils de la chaîne de compilation du MIPS et le simulateur **almo1**).

- Exécutez pour cette fois le .bashrc (parce que vous venez juste de le modifier):
  `source $HOME/.bashrc`
  Vous pouvez aussi ouvrir un nouveau terminal, celui-ci exécutera le script `.bashrc` avant d'afficher le prompt.

- **Récupération des codes pour le tp1**
  - Ouvrez un `terminal`
  - Créer un répertoire nommé `kO6` à la racine de votre compte:
    `mkdir ~/kO6`
    Dans le texte des TP, nous supposons que c'est là que vous mettrez vos codes, vous pouvez choisir un autre emplacement, mais dans les textes, ce sera `~/kO6`
  - Allez dans le répertoire `~/kO6`:
    `cd ~/kO6`
  - Copier les codes du tp1:
    `cp -rp /Infos/lmd/2024/licence/ue/LU3IN029-2024oct/kO6/tp1 .`
  - Exécutez la commande: **`cd tp1 ; tree -L 1`**.
    Vous devriez obtenir ceci:

```
.
|-- 0_test_almo1
|-- 1_hello_boot
|-- 2_init_asm
|-- 3_init_c
|-- 4_nttys
|-- 5_driver
`-- Makefile
```

- **Pour tester que tout fonctionne**
  - Allez dans le répertoire `0_test_almo1`:
    `cd tp1/0_test_almo1`
  - Exécuter la commande:
    `make exec`
    Vous devez voir 2 fenêtres `xterm` apparaître avec le message `Hello Word` et une petite fenêtre avec une roue tournante
  - **Pour arrêter le simulateur, vous devez taper `CTRL-C` dans le terminal où vous avez démarrer le simulateur**.
  - Si vous regarder dans le répertoire, le compilateur et le simulateur ont créés plusieurs fichiers (les exécutables MIPS et d'autres), pour faire le ménage et revenir aux seuls fichiers source, vous devez taper:
    `make clean`

1. Premier programme en assembleur dans la seule section de boot

Nous commençons par un petit programme de quelques lignes en assembleur, placé entièrement dans la région mémoire du boot, qui réalise l'affichage du message "Hello World". C'est un tout tout petit programme, mais pour obtenir l'exécutable, vous devrez utiliser tous les outils de la chaîne de cross-compilation MIPS et pour l'exécuter vous devrez exécuter le simulateur du prototype. C'est simple, mais c'est nouveau pour beaucoup d'entre vous

Objectifs

- produire un exécutable à partir d'un code en assembleur.

- savoir comment afficher un caractère sur un terminal.
- analyse d'une trace d'exécution

Fichiers

- Allez dans le répertoire `1_hello_boot`: `cd ~/k06/tp1/1_hello_boot`
  Il y a 3 fichiers.

```
1_hello_boot
??? hcpua.S      : code dépendant du cpu matériel en assembleur
??? kernel.ld    : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
??? Makefile     : description des actions possibles sur le code : compilation, exécution
```

Questions (certaines déjà vues en TD, mais vous devez ouvrir les fichiers pour répondre)

1. Dans quel fichier se trouve la description de l'espace d'adressage du MIPS ? Que trouve-t-on dans ce fichier (il faut l'ouvrir pour répondre) ?
2. Dans quel fichier se trouve le code de boot ? (h pour hardware, cpu = Central Processor Unit)
3. À quelle adresse démarre le MIPS ? Où peut-on le vérifier ?
4. Que produit `gcc` quand on utilise l'option `-c` ? (C9 S35)
5. Que fait l'éditeur de liens ? Comment est-il invoqué ? (C9 S37)
6. De quels fichiers a besoin l'éditeur de liens pour fonctionner ? (C9 S37)
7. Dans quelle section se trouve le code de boot pour le compilateur ? *(la réponse est dans le code assembleur, on fera un autre choix plus tard)*
8. Dans quelle section se trouve le message `"hello ..."` pour le compilateur ? Ce choix est particulier, mais ce message est en lecture seule, alors pourquoi pas...
9. Dans quelle section se trouve le code de boot dans le code exécutable produit par l'éditeur de liens ? (la réponse est dans le fichier `kernel.ld`)
10. Dans quelle région de la mémoire le code de boot est-il placé ? (la réponse est dans `kernel.ld`)
11. Comment connaît-on l'adresse du registre de sortie du contrôleur de terminal `TTY` ? (la réponse est dans `kernel.ld' et sur cette page)
12. Le code de boot se contente d'afficher un message, comment sait-on que le message est fini et que le programme doit s'arrêter ? (ou quel est le caractère de fin de chaîne en C ?)
13. Pourquoi terminer le programme par un `dead: j dead` ? Notez qu'on ne peut pas encore faire un *`syscall exit`* parce qu'il n'y a pas de gestionnaire de syscall et surtout parce `syscall` est une instruction appelée par une application utilisateur, et qu'on n'est pas dans une application utilisateur.
14. Quelle est la différence entre `#include "file.h"` et `#include <file.h>` ? Quelle option du compilateur C permet de spécifier les répertoires lesquels se trouvent les fichiers include ? Si vous ne savez pas interrogez Internet ! (C9 annexe S4+S10)
15. Comment définir une macro-instruction C uniquement si elle n'est pas déjà définie ? Écrivez un exemple. Si vous ne savez pas regardez l'usage de `#ifndef` (C9 annexe S10)
16. Comment être certain de ne pas inclure plusieurs fois le même fichier `.h` ? Vous devriez avoir trouvé l'explication en répondant à la question précédente. (C9 annexe S10)

Exercices

- Exécutez le programme en lançant le simulateur avec `make exec`, qu'observez-vous ?
- Exécutez le programme en lançant le simulateur avec `make debug`.
  Cela exécute le programme pour une courte durée et cela produit deux fichiers `trace0.s` et `label0.s`.
  `trace0.s` contient la trace des instructions assembleur exécutées par le processeur.
  Ouvrez `trace.0.s` et repérez ce qui est cité ici

- On voit la séquence chronologique des instructions exécutées
- La première colonne nous informe que les adresses lues sont dans l'espace Kernel (ici, K)
- La seconde colonne sont les numéros de cycles (ce n'est pas +1 à chaque instruction parce qu'il y a des caches et que cela prend du temps d'aller chercher les instructions dans la mémoire).
- La troisième sont les adresses. Notez que la première instruction lue est bien en `0xbfc00000`
- La quatrième le code binaire des instructions (sur 4 octets bien sûr)
- Le reste de la ligne contient l'instruction désassemblée (décodée)
- Lorsque les adresses ont un nom, c'est-à-dire qu'une étiquette leur a été attribuée, celle-ci est indiquée.

`label0.s` contient la séquence des appels de fonctions de l'exécutions. C'est en fait un extrait de la trace.
Ouvrez les fichiers `trace0.s` et `label0.s` et interprétez ce que vous voyez.
A quoi correspond les messages `READ` et `WRITE` dans le fichier `trace0.s`?

- Modifiez le code de `hcpua.S` afin d'afficher le message "Au revoir\n" après le message "Hello...". Vous devez avoir un message en plus, et pas seulement étendre le premier.

2. Saut dans le code du noyau en assembleur

Rendez-vous dans le répertoire `tp1/2_init_asm`.
Dans ce deuxième programme, nous restons en assembleur, mais nous avons deux fichiers source :

1. le fichier contenant le code de boot ;
2. le fichier contenant le code du noyau. Ici, le code du noyau c'est juste une *fonction* `kinit()`. Ce n'est pas vraiment une fonction car on n'utilise pas la pile.

Objectifs

- Savoir comment le programme de boot fait pour sauter à l'adresse de la routine kinit.
- Avoir un fichier kernel.ld un peu plus complet.

Fichiers

```
2_init_asm/
??? hcpua.S      : code dépendant du cpu matériel en assembleur
??? kernel.ld    : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
??? kinit.S      : fichier contenant le code de démarrage du noyau, ici c'est une routine kinit.
??? Makefile     : description des actions possibles sur le code : compilation, exécution, nettc
```

Questions

1. Regarder dans le fichier `hcpua.S`, dans quelle section est désormais le code de boot ? (on a changé par rapport à l'exercice précédent)
2. Le code de boot ne fait que sauter à l'adresse `kinit` avec l'instruction **`jr`**, il n'y a pas de retour, ce n'est donc pas un **`jal`**. Où est défini l'étiquette `kinit` ? Comment le code de boot connait-il cette adresse ? Pourquoi ne pas avoir utilisé `j init` et donc pourquoi passer par un registre ?
3. Dans `kernel.ld`, que signifie `*(.*data*)` ? (C9 S38+S39)
4. Quelle est la valeur de `__kdata_end` ? Pourquoi mettre 2 «_» au début des variables du `ldscript` ? (?réponse)

Exercices

- Exécutez le programme sur le simulateur. Est-ce différent de l'étape 1 ?
- Modifiez le code, comme pour l'étape 1, afin d'afficher un second message ?

3. Saut dans la fonction kinit() du noyau en langage C

Rendez-vous dans le répertoire `tp1/3_init_c`.
Dans ce troisième programme, nous faisons la même chose que pour le deuxième mais `kinit()` est désormais écrit en langage C. Cela change peu de choses, sauf une chose importante `kinit()` est une fonction et donc il faut absolument une pile d'exécution avant d'y entrer. Il faut donc initialiser **`$29`**.

Objectifs

- Savoir comment et où déclarer la pile d'exécution du noyau au démarrage (cela changera plus tard).
- Savoir comment afficher un caractère sur un terminal **depuis un programme C**.

Fichiers

```
3_init_c/
??? hcpua.S      : code dépendant du cpu matériel en assembleur
??? kernel.ld    : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
??? kinit.c      : fichier en C contenant le code de démarrage du noyau, ici c'est la fonction k
??? Makefile     : description des actions possibles sur le code : compilation, exécution, netto
```

Questions

1. Quand faut-il initialiser la pile ? Dans quel fichier est-ce ? Quelle est la valeur du pointeur initial ?
2. Dans quel fichier le mot clé `volatile` est-il utilisé ? Rappeler son rôle.

Exercices

- Exécutez le programme sur le simulateur. Est-ce différent de l'étape 1 ?
- Ouvrez les fichiers `kinit.o.s` et `kernel.x.s`, le premier fichier est le désassemblage de `kinit.o` et le second est le désassemblage de `kernel.x`. Dans ces fichiers, vous avez plusieurs sections. Les sections `.MIPS.abiflags`, `.reginfo` et `.pdr` ne nous sont pas utiles (elles servent au chargeur d'application, elles contiennent des informations sur le contenu du fichier et cela ne nous intéresse pas).
  Notez l'adresse de `kinit` dans les deux fichiers, sont-ce les mêmes ? Sont-elles dans les mêmes sections ? Expliquez pourquoi (c'est simple si vous avez compris le rôle de l'éditeur de liens).
- Modifiez le code de `kinit.c`, et comme pour l'étape 1, afficher un second message ?

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

Rendez-vous dans le répertoire `tp1/4_nttys`.
Le prototype de SoC que nous utilisons pour les TP est configurable. Il est possible par exemple de choisir le nombre de terminaux texte (TTY). Par défaut, il y en a un, mais nous pouvons en avoir jusqu'à 4. Nous allons modifier le code du noyau pour s'adapter à cette variabilité. En outre, pour le moment, nous ne faisions qu'écrire sur le terminal, maintenant, nous allons aussi lire le clavier.

Objectifs

- Savoir comment compiler un programme C avec du code conditionnel.
- Savoir comment décrire en C l'ensemble des registres d'un contrôleur de périphérique et y accéder.

Fichiers

```
4_nttys/
??? hcpua.S      : code dépendant du cpu matériel en assembleur
??? kernel.ld    : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
??? kinit.c      : fichier en C contenant le code de démarrage du noyau, ici c'est la fonction k
??? Makefile     : description des actions possibles sur le code : compilation, exécution, nettc
```

Questions

1. Dans le fichier `kinit.c`, dans la boucle `while` de la fonction `kinit()`, il est question d'un loopback, à quoi cela sert-il ? (Vous devez lire le code et les commentaires de cette fonction pour comprendre).
2. Dans le fichier `kinit.c`, on trouve `__tty_regs_map[ tty%NTTYS ].write = *s`, expliquez le modulo. Pour répondre à cette question, vous devez avoir compris comment sont rangés les registres de commande dans le contrôleur de TTY (C9 S10), et comprendre que le programmeur pourrait mettre n'importe quoi dans la variable `tty` et qu'on ne veut pas de bug !
3. Exécutez le programme sur le simulateur. Qu'observez-vous ? Est-ce que les deux fenêtres ont le même comportement vis-à-vis du clavier ?

Exercices

- Modifiez le code pour afficher un message sur le terminal `xterm1`, il y a toujours une attente sur le premier terminal.
- Modifiez le code pour que le programme affiche les touches tapées au clavier sur les deux terminaux. C'est-à-dire, ce que vous tapez sur le terminal `xterm0` s'affiche sur ce même terminal, et pareil pour `xterm1`. L'idée est de ne plus faire d'attente bloquante sur le registre `TTY_STATUS` de chaque terminal. Pour que cela soit plus amusant, changez la casse (minuscule ?? majuscule) sur le terminal `proc1_term1` (si vous tapez `bonjour 123`, il affiche `BONJOUR 123` et inversement).

B5. Premier petit pilote pour le terminal

Rendez-vous dans le répertoire `tp1/5_driver`.
Dans l'étape précédente, nous accédons aux registres de périphérique directement dans la fonction `kinit()`, ce n'est pas très simple. C'est pourquoi nous allons ajouter un niveau d'abstraction qui représente un *début* de pilote de périphérique (device driver). Ce pilote, même tout petit constitue une couche logicielle avec une API.

Objectifs

- Savoir comment créer un début de pilote pour le terminal `TTY`.
- Savoir comment décrire une API en C
- Savoir appeler une fonction en assembleur depuis le C

Fichiers

```
5_driver/
??? harch.c      : code dépendant de l'architecture du SoC, pour le moment c'est juste le pilote
```

```
??? harch.h      : API du code dépendant de l'architecture
??? hcpu.h       : prototype de la fonction clock()
??? hcpua.S      : code dépendant du cpu matériel en assembleur
??? kernel.ld    : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
??? kinit.c      : fichier en C contenant le code de démarrage du noyau, ici c'est la fonction k
??? Makefile     : description des actions possibles sur le code : compilation, exécution, netto
```

Questions

1. Les fonctions du driver de TTY ont pour prototype:

```
/**
 * \brief     read any char from the tty until count is reached
 * \param     tty   tty number (between 0 and NTTYS-1)
 * \param     buf   buffer where the read char are put
 * \param     count number of char read must be lower or equal to buf size
 * \return    the number of read chars
 */
int tty_read (int tty, char *buf, unsigned count);

/**
 * \brief     write count chars of the buffer to the tty
 * \param     tty   tty number (between 0 and NTTYS-1)
 * \param     buf   buffer pointer
 * \return    the number of written char
 */
int tty_write (int tty, char *buf, unsigned count);
```

   Elles sont dans `harch.h` et le code est dans le fichier `harch.c`. Dans `harch.h`, il n'y a pas La structure décrivant la carte des registres du `TTY`, laquelle est déclarée dans le `.c`. Pourquoi avoir fait ainsi ?
2. Le MIPS dispose d'un compteur de cycles internes. Ce compteur est dans un banc de registres accessibles uniquement quand le processeur fonctionne en mode `kernel`. Nous verrons ça au prochain cours, mais en attendant nous allons quand même exploiter ce compteur dans une fonction `unsigned clock(void)` dont le code est dans `hcpua.S`. Pourquoi avoir mis la fonction dans `hcpua.S` ? Rappeler, pourquoi avoir mis `.globl clock`
3. Compilez et exécutez le code avec `make exec`. Observez. Ensuite ouvrez le fichier `kernel.x.s` et regardez où a été placée la fonction `clock()`.
   Est-ce un problème si `kinit()` n'est plus au début du segment `ktext` ? Posez-vous la question de qui a besoin de connaître l'adresse de `kinit()`

Exercices

- les fonctions `tty_read()` et `tty_write()` n'offrent pas un service suffisant, alors nous avons ajouté les fonctions `tty_getc()`, `tty_gets()` et `tty_puts()` qui permettent respectivement de lire un caractère sur un TTY, de lire une chaîne de caractères ou d'écrire une chaîne de caractères. Ces fonctions seront déplacées dans un autre fichier plus tard, mais pour le moment, elles sont dans le fichier `kinit.c`.
- Afin de vous *détendre un peu*, vous allez créer un petit jeu `guess`
  - `guess` tire un nombre entre '0' et '9' et vous devez le deviner en faisant des propositions. `guess` vous dit si c'est trop grand ou trop petit. Ce programme ne va révolutionner votre vie de programmeur(se), mais bon, c'est probablement le premier programme que vous allez écrire et faire tourner sur une machine sans système d'exploitation.

  Étapes
  - Vous créez deux fichiers `guess.c` et `guess.h`.
    - `guess.c` contient le jeu il y au moins une fonction `guess()`

    - ◊ `guess.h` contient les déclarations externes de `guess.c`
- ♦ `kinit()` doit lancer `guess()`
- ♦ `guess()`
    - ◊ vous demande de taper une touche pour démarrer le jeu.
    - ◊ effectue un tirage d'un nombre en utilisant la fonction `clock()` et ne gardant que le chiffre de poids faible (ce n'est pas aléatoire, mais c'est mieux que rien)
    - ◊ exécute en boucle jusqu'à réussite
        - · demande d'un chiffre
        - · comparaison avec le tirage et affichage des messages `"trop grand"`, `"trop petit"` ou `"gagné"`
- ♦ Vous devrez modifier le Makefile puisque vous avez un fichier à compiler en plus.
- ♦ Si c'est trop facile, vous pouvez complexifier en utilisant des nombres à 2 chiffres ou plus.