DOCS [Start][Config][User][Kernel] ? COURS [9] [10] [11] ? TD [?9][?10][?11] ? TP [*9*][?10][?11] ? ZIP [gcc...][9][10][11]

Boot et premier programme en mode kernel

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.

IMPORTANT
Avant de faire cette séance, vous devez avoir lu 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*
- Configuration de l'environnement des TP : *obligatoire*
- Document sur l'assembleur du MIPS et la convention d'appel des fonctions : *recommandé, normalement déjà lu*
- Documentation sur le mode kernel du MIPS32 : *optionnel pour cette séance*

Récupération du code du TP

- **Vous devez avoir installé le simulateur du prototype almo1 et la chaine de cross-compilation MIPS (Config sections 2.2 et 3.2)**
- Téléchargez **l'archive code du tp1** et placez là dans le répertoire **`~/kO6`** (ou dans le répertoire que vous avez choisi, relisez la page sur la configuration si ce n'est pas clair).
- Ouvrez un `terminal`
- Allez dans le répertoire `kO6` : **`cd ~/kO6`**
- Décompressez l'archive du tp1 (dans le répertoire **`kO6`**) : **`tar xvzf tp1.tgz`**
- Exécutez la commande **`cd ; tree -L 1 kO6/tp1/`**.
  (si vous n'avez pas `tree` sur votre Linux, vous pouvez l'installer, c'est un outil utile, mais pas indispensable pour ces TP)
  Vous devrez obtenir ceci:

```
kO6/tp1
??? 1_hello_boot
??? 2_init_asm
??? 3_init_c
??? 4_nttys
??? 5_driver
??? Makefile
```

Avant de commencer

- 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.

- Vous devez avoir récupérer l'archive tp1.tgz pour pouvoir faire cette partie, si ce n'est pas la cas, retournez lire la section `Récupération du code du TP` en haut de cette page. La variable `shell` `$kO6` doit être définie dans votre environnement si vous avez suivi les consignes de la page Config sections 2.2.

- Si vous avez bien suivi les étapes de configuration de l'environnement et de récupération du code alors le code se trouve dans `~/kO6/tp1/`, et ouvrez un terminal et allez-y. Dans le répertoire `~/kO6/tp1/` vous avez 5 sous-répertoires et un Makefile. Le fichier `~/kO6/tp1/Makefile` permet de faire le ménage en appelant les Makefiles des sous-répertoires avec la cible `clean`, il est simple, mais c'est un Makefile hiérarchique. Ouvrez-le par curiosité.

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

```
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, nettc
```

Questions

1. Dans quel fichier se trouve la description de l'espace d'adressage du MIPS ? Que trouve-t-on dans ce fichier ?
2. Dans quel fichier se trouve le code de boot et pourquoi, selon vous, avoir nommé ce fichier ainsi ?
3. À quelle adresse démarre le MIPS ? Où peut-on le vérifier ?
4. Que produit `gcc` quand on utilise l'option `-c` ?
5. Que fait l'éditeur de liens ? Comment est-il invoqué ?
6. De quels fichiers a besoin l'éditeur de liens pour fonctionner ?
7. Dans quelle section se trouve le code de boot pour le compilateur ? *(la réponse est dans le code assembleur)*
8. Dans quelle section se trouve le message "hello" pour le compilateur ? Ce choix est particulier, mais ce message est en lecture seule.
9. Dans quelle section se trouve le code de boot dans le code exécutable ? (la réponse est dans `hcpua.S`)

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 ?)
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 appeler par une application utilisateur, et qu'il n'y en a pas encore.

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 des instructions exécutées
    - ♦ La première colonne nous informe que les adresses lues sont dans l'espace Kernel
    - ♦ La seconde colonne sont les numéros de cycles
    - ♦ La troisième sont les adresses
    - ♦ La quatrième le code binaire des instructions
    - ♦ Le reste de la ligne contient l'instruction désassemblée
    - ♦ 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 le fichier `label0.s` et interprétez ce que vous voyez.

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

2. Saut dans le code du noyau en assembleur

Dans le deuxième programme, nous restons en assembleur, mais nous avons deux fichiers source : (1) le fichier contenant le code de boot et (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, netto
```

Questions

1. Regarder dans le fichier `hcpua.S`, dans quelle section est désormais le code de boot ?
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 `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*)` ?
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

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.

Objectifs

- Savoir comment et où déclarer la pile d'exécution du noyau.
- 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.
- 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`

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, netto
```

Questions

1. Dans le fichier `kinit.c`, il est question d'un loopback, à quoi cela sert-il ?
2. Dans le fichier `kinit.c`, on trouve `__tty_regs_map[ tty%NTTYS ].write = *s`, expliquez le modulo.
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 second terminal, 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 `proc0_term0` s'affiche sur ce même terminal, et pareil pour `proc0_term1`. 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

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. Le code du driver du TTY est dans le fichier `harch.c` et les prototypes sont dans `harch.h`. Si vous ouvrez `harch.h` vous allez voir que seuls les prototypes des fonctions `tty_gets()` et `tty_puts()` sont présents. La structure décrivant la carte des registres du `TTY` 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. 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

- 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.