1. Récupération du code du TP
2. Questions à traiter en TD
3. 1. Premier programme en assembleur dans la seule section de boot
4. 2. Saut dans le code du noyau en assembleur
5. 3. Saut dans la fonction kinit() du noyau en langage C
6. 4. Accès aux registres de contrôle des terminaux TTY
7. 5. Premier petit pilote pour le terminal

Boot et premier programme en mode kernel

Cette page décrit la séance complète TD et TME. Elle commence par des exercices à faire sur papier et puis elle continue et se termine par des questions sur le code et quelques exercices de codage simples à écrire et tester sur votre machine virtuelle. La partie pratique est découpé en 5 étapes. Pour chaque étape, nous donnons une brève description, suivie d'une liste des objectifs principaux et d'une liste des fichiers présents. Un bref commentaire est ajouté pour les fichiers.

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*
- Document sur l'assembleur du MIPS et la convention d'appel des fonctions : *recommandé*
- Configuration de l'environnement des TP : *obligatoire*
- Documentation sur le mode kernel du MIPS32 : *optionnel pour cette séance*

Récupération du code du TP

- Vous devez commencer par récupérer l'archive code du tp1
- Assurez vous que vous avez déjà sourcé le fichier `Source-me.sh` (il doit être dans votre `.bashrc`
- Placez cette archive dans le répertoire AS5 et décompressez l'archive **`tar xvzf tp1.tgz`**
- Après décompression, vous devriez obtenir avec la commande **`tree -L 1 tp1`**

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

Questions à traiter en TD

- Analyse de l'architecture
- fonction write buffer en assembleur et en C
- Makefile et make recursif
- Les struct et les tableaux de structs
- fonction read buffer en C
- les usages de extern/globl et de volatile
- compilation conditionnelle

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

ajouter des analyses des fichiers objets et des traces

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
??? hcpu.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**
  Les réponse sont dans le cours ou dans les fichiers sources

  - Dans quel fichier se trouve la description de l'espace d'adressage du MIPS ? Que trouve-t-on dans ce fichier ?
  - Dans quel fichier se trouve le code de boot et pourquoi avoir nommé ce fichier ainsi ?
  - A quelle adresse démarre le MIPS ? Où peut-on le vérifier ?
  - Que produit `gcc` quand on utilise l'option `-c` ?
  - Que fait l'éditeur de liens ? Comment est-il invoqué ?
  - De quels fichiers a besoin l'éditeur de liens pour fonctionner ?
  - Dans quelle section se trouve le code de boot pour le compilateur ? *(la réponse est dans le code assembleur)*
  - Dans quelle section se trouve le message hello pour le compilateur ?
  - Dans quelle section se trouve le code de boot dans le code exécutable ?
  - Dans quelle région de la mémoire le code de boot est placé ?
  - Comment connaît-on l'adresse du registre de sortie du contrôleur de terminal TTY ?
  - 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 ?
  - Pourquoi terminer le programme par un `dead: j dead` ?

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/
??? hcpu.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
??? Makefile     : description des actions possibles sur le code : compilation, exécution
```

- **Questions**
  Les réponse sont dans le cours ou dans les fichiers sources

  - Regarder dans le fichier `hcpu.S`, dans quelle section est désormais le code de boot ?
  - Le code de boot ne fait que sauter dans la fonction kinit avec l'instruction `j`, il n'y a pas de retour, ce n'est donc pas un `jal`, mais pourquoi ne pas avoir utiliser `j init` et donc pourquoi passer par un registre ?
  - Dans `kernel.ld`, la définition de la mémoire est plus complète, elle contient 3 régions : pour le code de boot `boot_region` pour le code du noyau `ktext_region` et pour les données globales du noyau `kdata_region`. Ces régions ne contiennent qu'une section de sorties (resp. `.boot`, `.ktext` et `.kdata`) remplies avec les sections d'entrées produites par le compilateur. Que signifie `*(.*data*)` ?
  - Quelle est la valeur de `__kdata_end` ? Pourquoi, selon vous, mettre 2 «_» au début des variables ?
  -

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/
??? hcpu.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 fon
??? Makefile     : description des actions possibles sur le code : compilation, exécution
```

- **Questions**
  Les réponse sont dans le cours ou dans les fichiers sources

    - Question ?

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 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/
??? hcpu.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 fon
??? Makefile     : description des actions possibles sur le code : compilation, exécution
```

- **Questions**
  Les réponse sont dans le cours ou dans les fichiers sources

    - Question ?

5. Premier petit pilote pour le terminal

Dans l'étape 4, nous accédons au registre 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 créér uun début de pilote pour le terminal TTY.
- Savoir comment décrire une API en C

Fichiers

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

```
??? Makefile     : description des actions possibles sur le code : compilation, exécution
```

- **Questions**
  Les réponse sont dans le cours ou dans les fichiers sources

    - ♦ Question ?

- Quel est le nom de la directive assembleur permettant de déclarer une section