1. 1. Cours et documents annexes
2. 2. Objectifs généraux
3. 3. Principe pédagogique
4. 4. Fonctionnement des séances
5. 5. Séances de TME
    1. ? 1. Boot et premier programme en mode kernel?
    2. ? 2. Application simple en mode utilisateur?
    3. ? 3. Gestionnaire d'interruption?

Architecture des ordinateurs L3S5

Cette page est dédiée aux dernières séances du module Architecture des ordinateurs (LU3NI029). Vous trouverez sur cette page : (i) les objectifs généraux de ces séances en lien avec ce que vous avez déjà vu dans les premières séances de l'UE, (ii) une explication du principe pédagogique choisi consistant à construire un début de système d'exploitation sur un petit *ordinateur* (SoC) à partir de rien, et enfin (iii) la description des séances.

1. Cours et documents annexes

- **Cours démarrage**
    1. Architecture d'un SoC minimal (SoC = ordinateur intégré sur une puce)
    2. Chaîne de compilation du langage C puisque vous allez utiliser le langage C
    3. Présentation des piles de couches logicielles pour comprendre comment l'application communique avec le système d'exploitation
    4. Présentation du prototype virtuel du SoC que vous allez utiliser en TP

- **Documentation MIPS32 architecture et assembleur (mode user)**
    1. Registres de l'architecture externe accessible en mode user (p. 2)
    2. Espace d'adressage du MIPS32 (p. 4)
    3. Syntaxe et principales directives du langage assembleur (p. 5)
    4. Codage des instructions utilisateur du MIPS32 (p. 9)
    5. Instructions accessible en mode utilisateur (p. 11)
    6. Appels système de simulateur de processeur **Mars** (p. 22)
    7. Convention d'appel des fonctions (p. 24)

- **Documentation MIPS32 architecture et assembleur (coté kernel)**
    1. Modes d'exécution du processeur MIPS
    2. Registres protégés utilisables seulement en mode kernel
    3. Découpage de l'espace d'adressage
    4. Instructions protégées
    5. Cause d'entrée et de sortie du noyau du système d'exploitation
    6. Fonctionnement du registre d'état `c0_sr`
    7. Fonctionnement du registre de cause `c0_cause`

2. Objectifs généraux

Les 8 premières séances de l'UE décrivent l'architecture externe du MIPS (celle visible du programmeur) et la programmation structurée en assembleur (avec des fonctions et une pile). Les programmes réalisés utilisent des

structures de données simples telles que les tableaux à une dimension et les enregistrements (les *struct* du C) non récursifs. L'accès aux entrées-sorties se fait par des demandes de services en utilisant l'instruction `syscall`. Les programmes sont exécutés sur le simulateur de processeur MARS permettant d'observer l'évolution des registres du processeur et l'évolution des segments de mémoire utilisés par le code, les data et la pile.

Il s'avère que ce que fait l'instruction `syscall` n'est pas détaillé et l'architecture de l'ordinateur, au centre duquel se trouve le processeur MIPS, est juste présentée. C'est pourquoi le but des trois dernières séances est d'étudier plus en détail l'architecture d'un ordinateur simple, de type ?microcontrôleur à base de MIPS, et d'y exécuter une application au-dessus d'un embryon de système d'exploitation qui exécute les appels système, c'est-à-dire les services accessibles par l'instruction `syscall`.

Concernant le matériel, il est donc composé d'un processeur MIPS connecté à une mémoire et quelques périphériques. La mémoire contient le code et les données. Les périphériques sont les composants permettant les entrées-sorties (p. ex. le terminal écran-clavier) ou alors les composants offrant un service spécifique (p. ex. le *timer* qui compte le temps). Il s'agit d'apprendre à :

1. manipuler les deux modes d'exécution du processeur (mode *kernel* et mode *user*) ;
2. communiquer avec les contrôleurs de périphériques grâce à des registres de commandes accessibles dans l'espace d'adressage du processeur en utilisant les instructions *load*/*store* (`lw`/`sw`);
3. utiliser les lignes d'interruption des contrôleurs de périphériques (signal électrique à deux états : *ON* et *OFF*) pour prévenir le processeur d'un événement.

Concernant le logiciel, il est écrit en langage C et en assembleur. Il est composé d'un empilement de couches logicielles. Il y a tout d'abord le code de démarrage du processeur (le *boot*), puis le noyau du système d'exploitation, puis la bibliothèque système (la *libc*), et enfin l'application de l'utilisateur. Le code de *boot* initialise l'ordinateur. Le noyau gère les ressources matérielles et exécute les appels système (`syscall`). La bibliothèque système (la *libc*) est un ensemble de fonctions standards tel que `fprintf()` invoquant l'instruction `syscall`. Et enfin, l'application de l'utilisateur est un algorithme utilisant les fonctions de la *libc*. Il s'agit d'apprendre à :

1. programmer en langage C et utiliser une chaîne de compilation standard (compilateur + éditeur de liens) via un `Makefile` ;
2. exécuter les programmes sur un simulateur d'ordinateur complet avec processeur, mémoire et contrôleurs de périphériques ;
3. comprendre quelques services proposés par un système d'exploitation simple tels que les pilotes de périphériques, le gestionnaire de syscall, l'exécution de plusieurs fonctions en temps partagé (en les exécutant à tour de rôle très rapidement).

3. Principe pédagogique

Pour présenter les concepts des systèmes d'exploitation, la méthode employée en général est *top-down*. On vous présente les services des systèmes (gestion des fichiers, gestion des processus, gestion des communications interprocessus, etc.), puis on vous présente comment un système open source tel que Linux fait pour rendre ces services. C'est très intéressant, mais le système pris comme base est tellement complexe, qu'il est juste possible de voir qu'une petite partie, et certains étudiants perdent la vue d'ensemble. Pour l'UE d'architecture des ordinateurs, c'est une approche impossible parce qu'elle est trop éloignée de l'architecture matérielle.

Si Linux est trop complexe alors pourquoi ne pas prendre un petit système ad hoc, mais en conservant l'approche *top-down* ? Oui, cela peut être envisagé, c'est d'ailleurs ce qui a été fait dans un ancien module. Toutefois, c'est encore difficile, parce que pour bien comprendre comment fonctionne un service du système d'exploitation, il faut

avoir une vue d'ensemble du système et ce n'est pas simple à présenter.

Nous avons choisi, une approche *bottom-up*. Nous partons d?une feuille blanche, et nous ajoutons progressivement les services en limitant le nombre de fichiers et la taille des codes. Chaque nouveau service qui s'ajoute s'appuie sur les services précédemment construits.

Pour conclure, si on devait vous apprendre comment est faite une voiture. L'approche *top-down* consiste à prendre une voiture du commerce et à la démonter pour voir de quoi elle est faite. L'approche *bottom-up* consiste à assembler des éléments pour construire une toute petite voiture (genre 2CV :-)).

4. Fonctionnement des séances

Le but des TD est de préparer le travail que vous devez faire dans le TP. L'idée générale des TP est de créer, très progressivement, un tout petit système d'exploitation. Évidemment, dans le temps imparti, il n'est pas envisageable de créer un système complexe. Ce système est petit, mais il se veut simple à comprendre.

Toutes les séances sont structurées de la même manière. Chaque séance est découpée en étapes qui doivent être suivies dans l'ordre. Chaque étape est indépendante des autres du point de vue des fichiers, c'est-à-dire qu'elle n'utilise pas les fichiers des étapes précédentes et donc s'il y a des fichiers en commun, ceux-ci sont répliqués. Le code fourni est toujours fonctionnel et il y a toujours un `Makefile` pour produire l'exécutable et le faire tourner sur le simulateur du prototype de l'ordinateur. Le code est très commenté et il n'y a pas ? ou peu ? de "trous" à remplir.

Chaque étape introduit un petit nombre de concepts. Dans la première étape de la première séance, il n'y a que 2 fichiers sources (`hcpu.S` et `kernel.ld`) et 1 `Makefile` simple (de type *collection de Shell script*). Dans les étapes suivantes, on ajoute progressivement des fichiers et des services.

Le travail demandé pour chaque étape est le suivant :

1. Il faut répondre à des questions simples portant sur le code et sur l'architecture dans un compte-rendu au format *Markdown* prérempli. Un modèle de compte-rendu est téléchargeable sur la page du TME. Les réponses aux questions sont en général dans le code ou dans les commentaires du code ou dans les réponses aux questions du TD qui a préparé la séance de TP ou enfin, dans les diapositives du cours.
   Le but de ces questions est de pointer les difficultés introduites dans l'étape et de les associer à des éléments de réponses vus précédents. Par exemple, dans la première étape, une question est : - *A quelle adresse démarre le MIPS?*. La réponse est dans le cours et dans le TD et dans le fichier `ldscript.ld`.

2. Il faut ajouter une fonctionnalité au programme dans un ou plusieurs fichiers et décrire cet ajout dans le compte-rendu. Cet ajout peut nécessiter la création d'un nouveau fichier source.
   Le but de cet ajout est de s'approprier le code grâce à une petite modification et d'en voir toutes les implications. Par exemple, dans la première étape de la première séance, il est demandé d'afficher deux messages au lieu d'un.

5. Séances de TME

? 1. Boot et premier programme en mode kernel?

Dans cette séance, il s'agit de comprendre comment un ordinateur simple démarre et comment le programmeur peut interagir avec le monde extérieur via les contrôleurs de périphériques. Il s'agit également d'utiliser le langage C pour les programmes et donc la chaîne de compilation *GCC* et il s'agit d'introduire l'usage d'un `Makefile`. Nous décrivons aussi les fonctions C pour les entrées-sorties telles que `printf()` et `gets()`. Pour toutes les étapes, le nom de l'exécutable produit est `kernel.x`.

L'architecture de l'ordinateur utilisé dans cette séance est composée d'un processeur MIPS, d'une mémoire multisegment et d'un contrôleur de terminal qui peut contrôler jusqu'à 4 terminaux indépendants.

? 2. Application simple en mode utilisateur?

Dans cette séance, nous utilisons les deux modes d'exécution du processeur, le mode *kernel* et le mode *user*. En mode *kernel*, le processeur a droit à toutes les instructions et à tout l'espace d'adressage. En mode *user*, le processeur est bridé, certaines instructions sont interdites et seule une partie de l'espace d'adressage est autorisé. Nous allons voir comment se passe le passage d'un mode à l'autre. Dans toutes les étapes, le code se répartie dans deux exécutables: `kernel.x` et `user.x`. Nous allons voir en particulier comment fonctionne le gestionnaire d'appel système.

? 3. Gestionnaire d'interruption?

Dans cette troisième séance, il s'agit de comprendre comment fonctionne le gestionnaire des interruptions. Pour cela, nous allons ajouter dans l'architecture deux autres composants : un timer qui compte le temps (ou ici qui compte les cycles) et un concentrateur d'interruption. Ce concentrateur permet de mixer toutes les lignes d'interruptions des contrôleurs de périphériques pour n'en produire qu'une seule à destination du processeur MIPS.