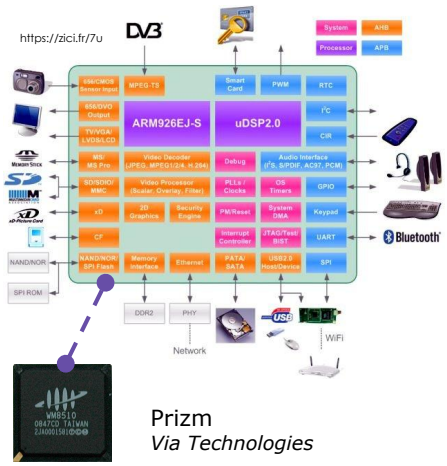
Démarrage du Système

LU3INx29 Architecture des ordinateurs 1

franck.wajsburt@lip6.fr

20 nov 2023

SoC



Prizm
Via Technologies

- System on Chip ou SoC :
  un circuit contenant un système entier !
- Les SoC sont présents partout :
  dans les smartphones, les tablettes,
  les ordinateurs, les voitures, etc.
  et même dans les écouteurs (sans fil).
- Dans un SoC, on trouve au moins :
  un cœur de processeur, de la mémoire
  et des contrôleurs de périphériques

Sur la droite vous avez un exemple de SoC et son interface vers ses périphériques.

Le M3 Ultra d'Apple™ à base de core ARM contient … 92 milliards de transistors !

Questions pour cette séance

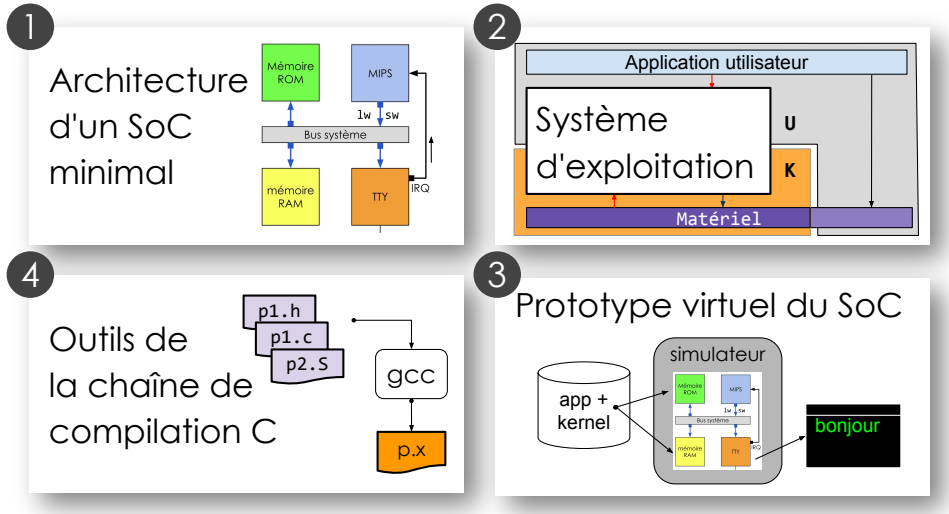
Que trouve-t-on dans un SoC au minimum ?
→ Un cœur et …

Comment produire un exécutable à partir d'un code C ?
→ Avec un compilateur et …

Quelle est la place et le rôle du système d'exploitation ?
→ Entre l'application utilisateur et le SoC …

Qu'est-ce qu'un prototype virtuel de SoC ?
→ Une application qui simule le fonctionnement d'un SoC …

Plan de la séance



Architecture d'un SoC minimal

Quels sont les composants de base d'un SoC minimal ?

→ Un cœur (le processeur), de la mémoire et des contrôleurs de périphériques et un bus pour router les requêtes de lecture/écriture

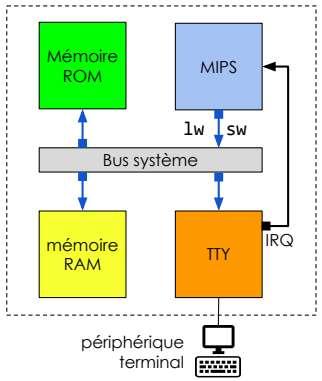
Quelle est la différence entre l'espace d'adressage et la mémoire ?

→ L'espace d'adressage est un intervalle d'adresses.
La mémoire est un composant matériel contenant des cases de mémoire accessible dans l'espace d'adressage

SoC minimal

Ce schéma est un exemple de SoC minimal mais réaliste, c'est celui qui sera utilisé en TP.
SoC nommé **almo** (*créée pour l'étude de l'architecture logicielle et matérielle des ordinateurs*)

Système sur Puce minimal



Un processeur, le cœur de calcul (MIPS)
qui exécute le code des programmes sur des données présentes en mémoire et qui accède aux autres composants par des load/store

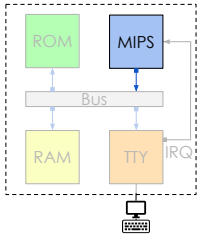
Un composant de mémoire morte (ROM)
contenant le code de démarrage

Un composant de mémoire vive (RAM)
contenant le code du programme, ses données et sa (*ou ses*) pile(s) d'exécution de fonctions

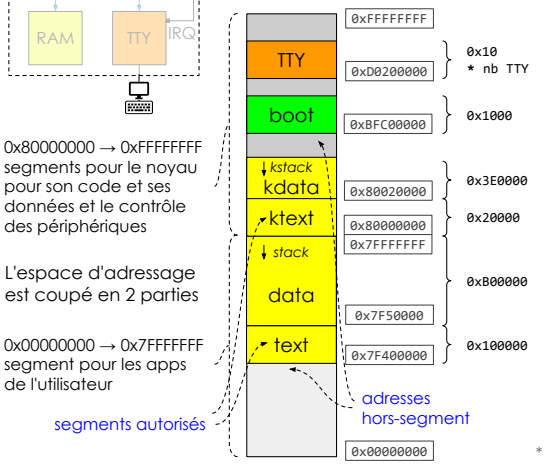
Un contrôleur de terminal (TTY)
qui contrôle le couple écran-clavier, qui peut envoyer des requêtes d'interruption (IRQ) vers le MIPS (IRQ : Interrupt ReQuest) pour prévenir qu'un caractère tapé au clavier est en attente d'être lu.

Un bus système
qui transmet les requêtes de lecture et d'écriture du MIPS vers les mémoires et vers le TTY

Espace d'adressage du prototype



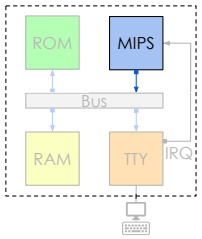
L'espace d'adressage du MIPS est l'ensemble des adresses que peut produire le MIPS. Les segments d'adresses ci-dessous sont ceux du prototype en TP.



- Le segment de **boot** commence en **`0xBFC00000`**
  Cette adresse est imposée par le MIPS.
  Le segment fait ici **`0x1000`** octets (4 kio)
- Le segment des **TTY** commence en **`0xD0200000`**
  C'est un segment de 16 octets par TTY.
- **ktext** et **kdata** sont les segments utilisés par le noyau du système d'exploitation
  - **ktext** commence en **`0x80000000`** et sa taille est **`0x20000`** (132 kio)
  - **kdata** commence en **`0x80020000`** et sa taille est **`0x3E0000`** (~ 4 Mio)
- **text** et **data** sont les segments de l'application
  - **text** commence en **`0x7F400000`** et sa taille est **`0x100000`** (1 Mio)
  - **data** commence en **`0x7F500000`** et sa taille est **`0xB00000`** (~ 11 Mio) ce segment contient les données globales du programme et la (ou les) pile(s) d'exécution des fonctions.

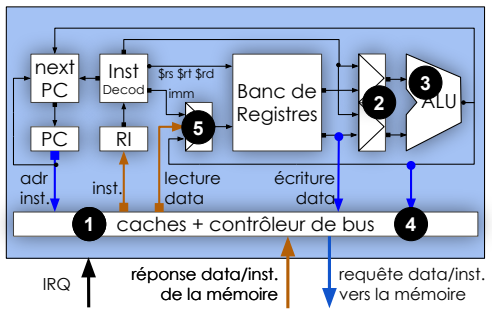
** Sur le dessin, les couleurs des segments désignent les composants qui les gèrent.*

Le cœur MIPS32



Le MIPS32 est un processeur RISC qui exécute environ 1 instruction par cycle. Les calculs sont faits dans les registres internes. La mémoire est juste lue/écrite via un contrôleur de bus qui arbitre entre les accès data et les accès instructions.

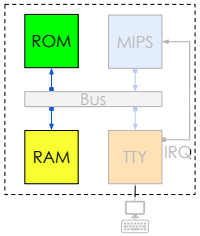
Notez la présence de caches qui stockent les data et les instructions lues le plus fréquemment mais nous ne les présenterons pas dans ce module.



Le MIPS a **2 modes** d'exécution : **kernel** et **user**
- Le mode **kernel a droit à tout**
- Le mode **user est bridé** car une partie de la mémoire et des instructions sont interdites.

Le MIPS exécute les instructions (`add`, `lw`, `beq`, …) en plusieurs étapes

1. Lecture de l'instruction à l'adresse du PC et rangement dans le registre instruction (RI)
2. Décodage de l'instruction et calcul des opérandes à partir de RI et des registres
3. exécution de l'opération dans l'ALU
4. Si c'est une instruction load/store (lw/sw) Lecture ou écriture mémoire
5. Écriture du résultat dans le banc de registres



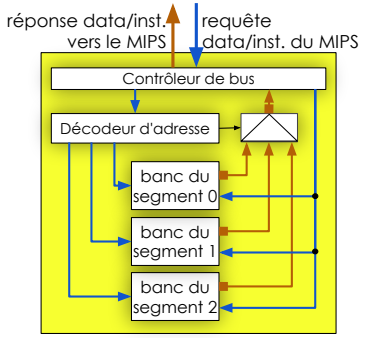
Mémoires RAM et ROM

Les mémoires RAM et ROM stockent les instructions et les data des programmes

- Les mémoires sont des tableaux d'octets et chaque octet a sa propre adresse
- Les mémoires reçoivent des requêtes du MIPS de lecture (*load*) ou d'écriture (*store*)

RAM et ROM sont deux types de mémoire

- Le contenu d'une ROM est persistant. ⇒ Elle ne peut pas être écrite par le MIPS
- Une RAM peut être lue et écrite ⇒ Son contenu est non significatif au démarrage.



Chaque mémoire gère une ou plusieurs segments d'adresses dans des bancs de mémoire physique (un banc de mémoire est un composant matériel).

Les requêtes contiennent

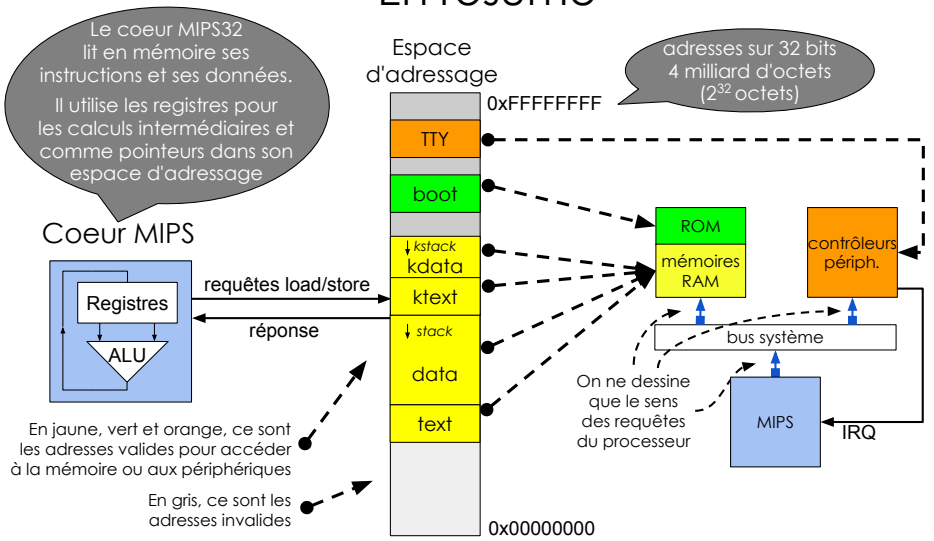
- une adresse sur 32 bits
- une commande (read/write)
- une donnée sur 32 bits si c'est une écriture
- un masque de 4 bits désignant les octets concernés pour les écritures (bit enable)

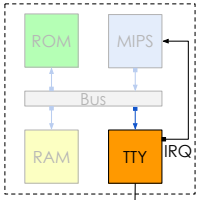
Le décodeur sélectionne le banc concerné par l'adresse

La mémoire produit une réponse avec :

- une donnée sur 32 bits si c'est une lecture (la sélection du ou des octets concernés par la lecture est faite par le MIPS lui-même).
- un acquittement si c'est une écriture.

En résumé

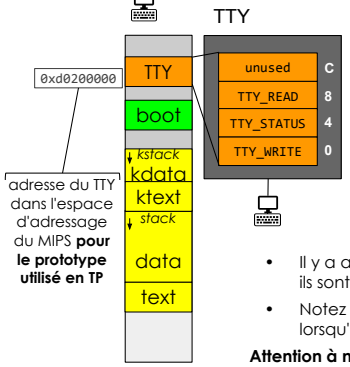




Périphériques (*Devices*)

Un contrôleur de périphérique (device en anglais) est un composant matériel qui propose un service spécifique (entrée-sortie, timer, etc.)

- Le TTY est un contrôleur périphérique d'entrées-sorties texte (écran-clavier)
- Les contrôleurs de périphériques contiennent des registres de contrôle placés dans l'espace d'adressage du MIPS pour qu'il puisse les commander par des load/store
- Le TTY contient 4 registres par TTY à partir de l'adresse `0xd0200000` (3 sont utilisés ici)



`TTY_WRITE` est le registre de sortie vers l'écran du terminal TTY

`0xd0200000` C'est un registre **en écriture seule**.
Pour écrire un caractère à l'écran, il faut écrire son code ASCII dans le registre TTY_WRITE. Le code est envoyé au terminal et le caractère s'affiche là où se trouve le curseur, lequel avance automatiquement.

`TTY_STATUS` est le registre d'état qui informe de la présence de caractères envoyés par le clavier et en attente d'être lu.

`0xd0200004` C'est un registre **en lecture seule** qui vaut `0` s'il n'y a pas de caractères en attente et `1` s'il y a au moins un caractère en attente d'être lu.

`TTY_READ` est le registre d'entrée du clavier

`0xd0200008` C'est un registre **en lecture seule**.
Il contient le code ASCII du dernier caractère tapé au clavier.

- Il y a autant de jeu de 4 registres que de contrôleur TTY dans la plateforme, ils sont placés à des adresses consécutives : pour 2 TTYs de `0xd0200000` à `0xd020001C`
- Notez la présence d'une ligne d'interruption (IRQ) que le TTY active (2 états ON/OFF) lorsqu'il a un caractère en attente (nous détaillerons son usage au dernier cours)

Attention à ne pas confondre registres du MIPS et registres de périphérique !

Ce qu'il faut retenir

Un SoC est composé au minimum

- d'un cœur de processeur pour exécuter les instructions, ici ce sont des MIPS
- d'une mémoire ROM contenant le code de démarrage du processeur.
- d'une mémoire RAM contenant un ou plusieurs bancs de mémoire représentant des segments de l'espace d'adressage du processeur :
  - pour les instructions du programme utilisateur et du système d'exploitation ;
  - pour les données du programme utilisateur et du système d'exploitation ;
  - et pour les piles d'exécution des fils d'exécution de programmes.
- d'un contrôleur d'entrées-sorties configurable par des registres accessibles par lecture/écriture dans l'espace d'adressage (mais ce n'est pas de la mémoire).
- d'un bus système routant (acheminant) les requêtes de lecture/écriture du processeur vers les composants gérant les adresses concernées.
- de lignes d'interruption permettant aux contrôleurs d'entrées-sorties de prévenir de la terminaison d'une commande demandée.

Système d'exploitation

Qu'est-ce qu'une API ?

→ Un contrat définissant un mode d'emploi de services spécifiques.

A quoi sert le système d'exploitation (OS) ?

→ Gérer les ressources matérielles et logicielles au travers d'APIs

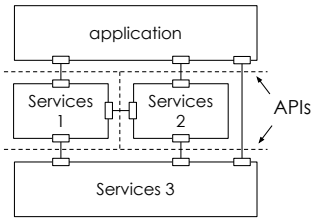
Comment le SoC utilisé en TP démarre-t-il ?

→ Exécution du code de boot puis entrée dans le noyau

Couches logicielles et API

(https://www.wikiwand.com/fr/Architecture_logicielle#/Architecture_en_couches)

Les programmes sont composés d'un empilement de couches logicielles.

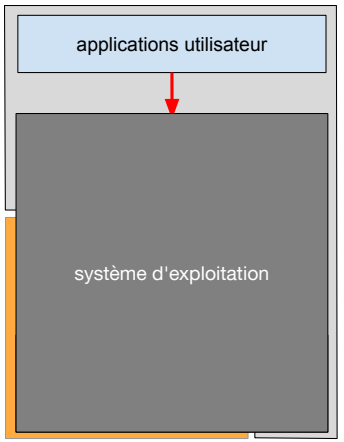


- Une couche logicielle rassemble un ensemble de services cohérents et propose une interface nommée API (Application Programming Interface) permettant d'accéder aux services.
- une API est un contrat définissant un mode d'emploi des services offerts par la couche logicielle.
- Une API en C est déclarés dans un fichier `.h`,
  C'est un ensemble de :
  → fonctions ; types ; #define ; structures de données ; etc.

Les APIs permettent

1. de réduire la complexité des programmes (moins de code à écrire)
2. d'améliorer la fiabilité des programmes car l'implémentation des API est sûr
3. de simplifier l'évolution des programmes (programmation modulaire)

Système d'exploitation et Applications



Les applications ne s'exécutent généralement pas directement sur le matériel, elles utilisent les services d'un système d'exploitation.

Les **services** du système d'exploitation sont nombreux :

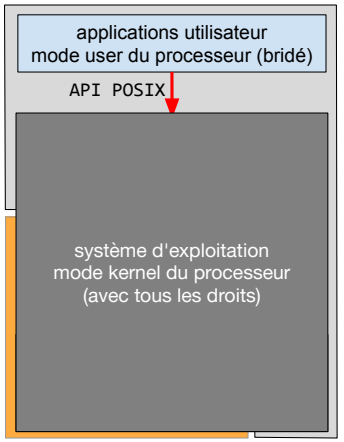
- **création** et **destruction** des applications ;
- **allocation** équitable des ressources matérielles pour les applications : mémoire, périphériques, fichiers, ports réseau, processeur(s), etc. ;
- **communication** et **synchronisation** des applications entre elles.

avec

- garantie de la **sécurité** des applications : c'est-à-dire la **confidentialité** et l'**intégrité** des données ainsi que la **disponibilité** des ressources ;
- garantie de la **sûreté** de fonctionnement du matériel par l'usage d'API spécifiques ;

API POSIX

https://www.wikiwand.com/fr/POSIX



Les systèmes d'exploitation implémentent souvent l'**API POSIX** (Portable Operating System Interface) pour simplifier le portage des applications sur plusieurs systèmes.

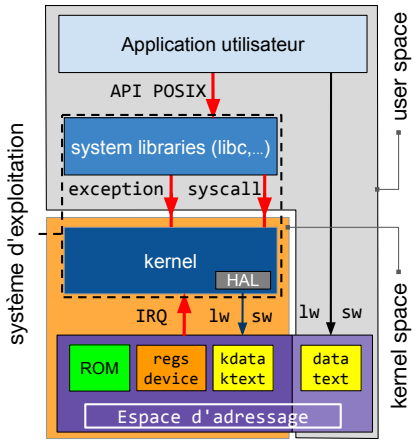
- **POSIX** est en réalité un ensemble d'APIs, vous connaissez déjà la **libc** (printf(), read(), etc.) mais il y en a d'autres comme les **Pthreads**.

Le système d'exploitation doit garantir que les applications accèdent exclusivement aux ressources autorisées.

C'est possible en utilisant les modes d'exécution du processeur : les modes **kernel** et **user**.

- En mode **kernel**, le processeur peut utiliser toutes les adresses de l'espace d'adressage et toutes les instructions.
- En mode **user**, le processeur est bridé, certaines adresses et instructions sont interdites

Système d'exploitation = librairies + kernel



Ce schéma est une vue simplifiée des composants d'un système d'exploitation.

Il y a 2 espaces de droits : **user** & **kernel**

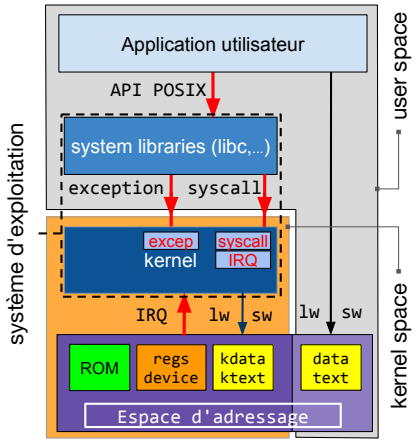
- l'**espace kernel** représente les ressources accessibles seulement en mode kernel.
- l'**espace user** représente les ressources accessibles à l'application, c.-à-d. les bibliothèques système et ses segments de code, de données et de pile.

Le système d'exploitation est composé de **2 parties**:

1. Les **bibliothèques système** qui implémentent l'API POSIX et qui s'exécutent en **mode user**.
2. Le **kernel** (noyau) qui implémentent les services tels que l'allocation des ressources matérielles et qui s'exécute en **mode kernel**.

Le noyau propose également une API pour l'accès aux ressources matérielles afin de faciliter le portage sur des architectures différentes. C'est la couche d'abstraction matérielle ou **HAL** (Hardware Abstraction Layer).

Les 3 gestionnaires du système d'exploitation



Le noyau implémente 3 **gestionnaires de services** pour les **syscalls**, les **exceptions** et les **IRQ** des périphériques.

Le **gestionnaire de syscall** offre les services nécessaires à l'API POSIX, tels que :

- les commandes de périphériques
- l'allocation de mémoire
- le lancement et l'arrêt d'application

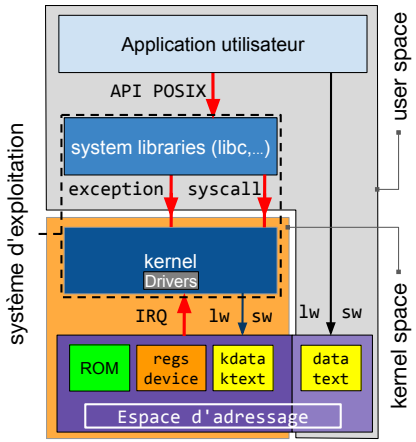
Le **gestionnaire d'exception** gère les erreurs du programme ou les allocations à la volée, telles que:

- la division par 0
- la violation de privilège (accès interdit en mode **user**)
- les instructions inconnues
- les erreurs de segmentation (accès mémoire hors segment)

Le **gestionnaire d'interruption** gère les requêtes (IRQ) (toujours attendus) venant des périphériques, tels que:

- les fins de commande (lorsque le travail est fait)
- les **ticks** d'horloge (pour que l'OS compte le temps)

Pilote (*Driver*)



Quand une application veut utiliser un périphérique (device), elle utilise une fonction de la libc (p. ex. `printf()`) qui réalise un appel système (`syscall write`) et c'est le noyau qui commande le périphérique en écrivant dans ses registres de contrôle (nous avons vu ceux du TTY).

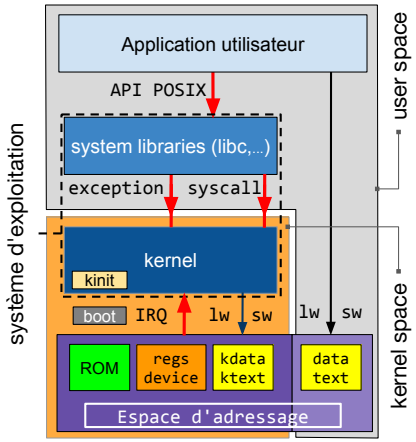
> Un **Pilote** (Driver en anglais) est le code dans le noyau qui **contrôle un périphérique**, c'est-à-dire qui écrit les commandes du périphérique, lit ses réponses, traite ses erreurs, et réagit à ses IRQ (Interrupt ReQuests).

Il existe une **variété innombrable de périphériques**, et donc de « Pilote de Périphérique » (Device Driver). Il est impossible qu'un noyau dispose de tous les pilotes, d'autant qu'ils sont écrits par les vendeurs de périphériques, et non pas par les développeurs du noyau du système.

Tous les noyaux proposent une API pour les pilotes avec des fonctions standards (`open()`, `read()`, write(), close(), etc.) que les vendeurs de périphériques doivent implémenter.

Dans notre cas, nous aurons un pilote pour le TTY même s'il est très simple.

Boot standard



Quand un SoC démarre :

- Les mémoires RAM sont vides (contenu non significatif).
- Les périphériques sont dans un état non fonctionnel (ils n'ont pas reçu de commandes et ils ne peuvent pas émettre d'interruptions).

Le code de boot contient les premières instructions exécutées par le processeur.

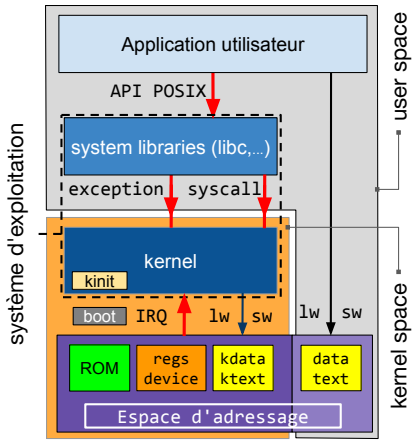
- Le code de boot est nécessairement dans une mémoire persistante, en ROM.
- Pour le MIPS, il est à l'adresse **`0xBFC00000`**
- Les premières instructions sont toujours en assembleur.
- Il doit au minimum initialiser le pointeur de pile pour pouvoir sauter dans la fonction d'entrée du noyau **`kinit()`** (**k**ernel **init**ialisation) écrite en langage C.

Dans un vrai système, le code de boot doit :

- scanner (découvrir) le matériel pour connaitre la taille des mémoires et trouver les périphériques présents ;
- puis, lire le *boot loader*, placé au début du disque dur, qui charge le système d'exploitation présent dans l'un des fichiers du disque ou sur le réseau.

Le code de boot ne fait pas partie du système d'exploitation puisque son but est de le charger.

Boot du prototype de TP



Le prototype que vous allez utiliser :

- n'a pas de contrôleur de disque.
- n'a pas la possibilité de « découvrir » le matériel

Le processus de boot doit donc être plus simple.

- Le simulateur du prototype prend en paramètre les fichiers binaires contenant le noyau et l'application et remplit directement les RAM avec les instructions et les données globales présentes dans ces fichiers.
- Le matériel (les adresses, la taille des segments de mémoire, le nombre de périphériques ou de procs) est décrit dans des variables du fichier ldscript du kernel (donc connu par le code du kernel) et dans des `#define` du code kernel.

Cela simplifie beaucoup le démarrage, c'est ce qui se passe dans les micro-contrôleurs (ex: Arduino) intégrant des mémoires RAM en technologie flash dont le contenu est persistant comme pour les ROM et dont le matériel est connu au moment de la compilation du programme.

Ce qu'il faut retenir

- Une API définit une interface standard de services.
  Ici, une API prend la forme d'un ensemble de fonctions, de types, de variables, etc. défini dans des fichiers include (`.h`).
- Les applications utilisent un système d'exploitation pour accéder aux ressources matérielles grâce à des API utilisateur comme POSIX.
- Un système d'exploitation est composé de 2 parties : les bibliothèques système qui implémentent l'API utilisateur (POSIX) et le noyau (kernel) qui gère le matériel.
- Les applications et les bibliothèques système s'exécutent en mode user (mode sans privilège : c'est-à-dire sans pouvoir accéder aux ressources protégées)
- Le noyau utilise le mode kernel du processeur pour s'exécuter (mode privilégié).
- Le noyau rend ses services grâce à 3 gestionnaires :
  gestionnaire d'appel système (syscall), gestionnaire d'exception (erreur) et gestionnaire d'interruptions (IRQ)
- Le processeur démarre en mode kernel pour exécuter le code de boot qui entre dans le noyau, lequel initialise le matériel et ses structures internes avant de démarrer la première application (et la seule pour cette UE).

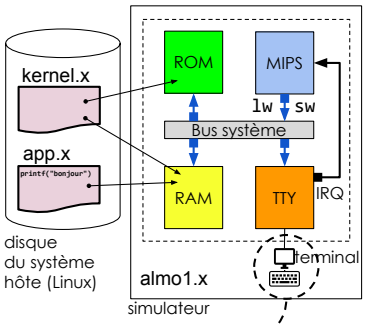
Prototypage Virtuel

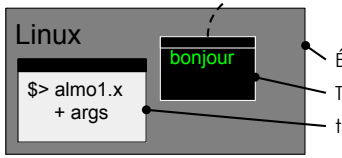
A quoi ressemble le prototype virtuel du SoC que nous utilisons ?
→ C'est une application qui simule le comportement du SoC…

Avec quoi allons-nous charger les mémoires du prototype ?
→ Avec 2 exécutables : le noyau (kernel.x) et l'application (app.x)...

Comment allons-nous analyser l'exécution du code ?
→ Le simulateur produit une trace d'exécution des instructions…

Simulateur du SoC `almo1.x`





Le simulateur du SoC utilisé en TP se nomme **`almo1.x`**
Ce simulateur est configurable, on peut choisir le nombre de périphériques et même de MIPS (ici `1`)

`almo1.x` prend en paramètre :

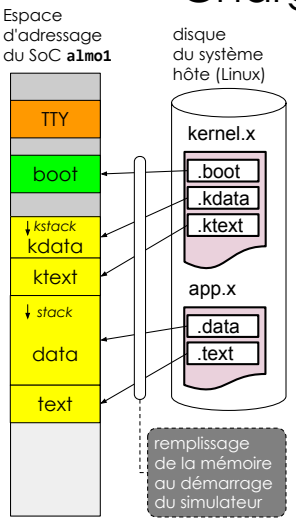
- les fichiers binaires MIPS du kernel et de l'application
- les paramètres du matériel (p. ex. nombre de TTY)
- le nombre de cycles à simuler (p. ex. 1 000 000)
- la demande de trace d'exécution des instructions

`almo1.x` au démarrage :

- **remplit les segments** de mémoire (**`.text`** et **`.data`**) avec ce qu'il trouve dans les fichiers binaires
- **active le reset** : le signal de démarrage du MIPS
- **démarre** l'exécution de la première instruction qu'il va chercher à l'adresse **`0xBFC00000`**

Écran du système hôte (Linux)
TTY du simulateur **`almo1.x`**
terminal de commande Linux pour lancer **`almo1.x`**

Chargement des mémoires



Les mémoires du SoC almo sont remplies avant le démarrage de la simulation par les sections (`.text`, `.data`, etc.) que le simulateur `almo1.x` trouve dans les fichiers produits par la chaîne de cross-compilation MIPS.

En supposant que :

- `kernel.x` est le code binaire du noyau du système d'exploitation
- `app.x` est le code binaire de l'application (contenant le code de l'application utilisateur et le code de la librairie C du système d'exploitation)

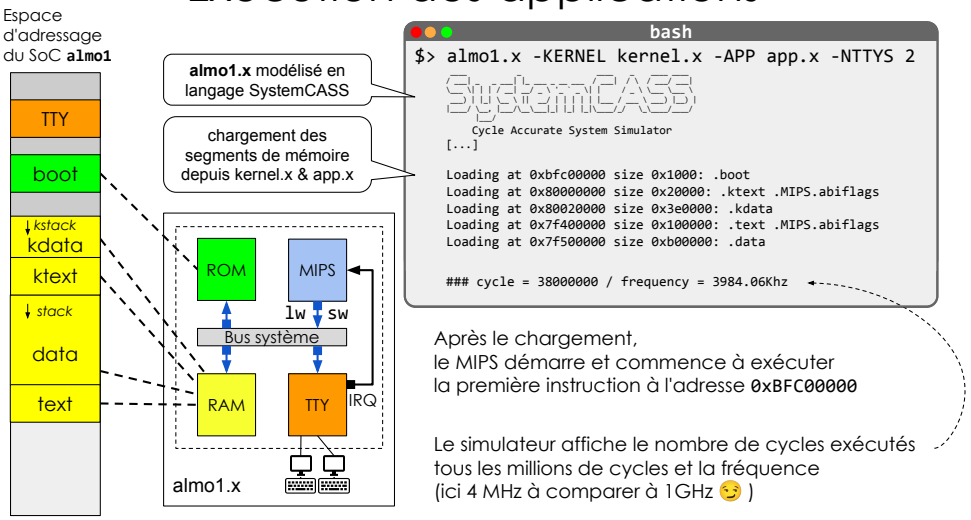
Les deux binaires ont été créés par l'éditeur de liens avec :

- `kernel.x` contient les trois sections `.ktext` `.kdata` et `.boot`
- `app.x` contient les deux sections `.text` `.data`

```
$> almo1.x -KERNEL kernel.x -APP app.x -NTTYS 2
```

Sur un ordinateur, ce chargement des mémoires fait par l'exécution d'un ***bootloader logiciel,*** *il en existe plusieurs niveaux, ici le chargement des mémoires est fait par le matériel, comme dans le cas des micro-contrôleurs, avant le démarrage du processeur du prototype.*

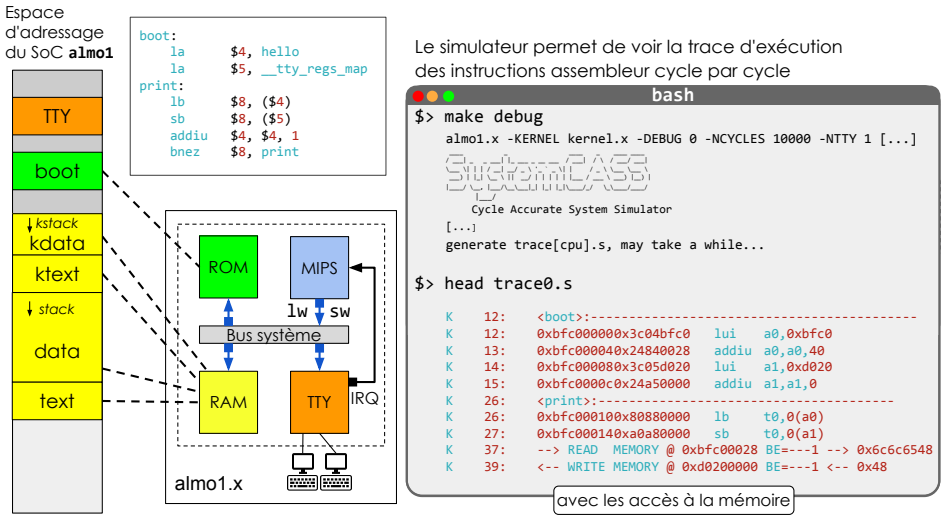
Exécution des applications



Après le chargement,
le MIPS démarre et commence à exécuter
la première instruction à l'adresse `0xBFC00000`

Le simulateur affiche le nombre de cycles exécutés
tous les millions de cycles et la fréquence
(ici 4 MHz à comparer à 1GHz 😉)

Trace d'exécution



Le simulateur permet de voir la trace d'exécution des instructions assembleur cycle par cycle

```
bash
$> make debug
    almo1.x -KERNEL kernel.x -DEBUG 0 -NCYCLES 10000 -NTTY 1 [...]
    Cycle Accurate System Simulator
    [...]
    generate trace[cpu].s, may take a while...

$> head trace0.s

    K    12:    <boot>:----------------------------------------
    K    12:    0xbfc000000x3c04bfc0   lui    a0,0xbfc0
    K    13:    0xbfc000040x24840028   addiu  a0,a0,40
    K    14:    0xbfc000080x3c05d020   lui    a1,0xd020
    K    15:    0xbfc0000c0x24a50000   addiu  a1,a1,0
    K    26:    <print>:---------------------------------------
    K    26:    0xbfc000100x80880000   lb     t0,0(a0)
    K    27:    0xbfc000140xa0a80000   sb     t0,0(a1)
    K    37:    --> READ  MEMORY @ 0xbfc00028 BE=---1 --> 0x6c6c6548
    K    39:    <-- WRITE MEMORY @ 0xd0200000 BE=---1 <-- 0x48
```

avec les accès à la mémoire

Ce qu'il faut retenir

- Le prototype virtuel est une modélisation du SoC que l'on peut simuler cycle par cycle pour exécuter l'application et le noyau.
- Les composants présents dans le prototype sont partiellement configurables au lancement du simulateur, par exemple le nombre de TTY.
- Le simulateur prend en paramètre deux fichiers binaires obtenus par gcc. L'un contient le code du noyau, l'autre contient le code de l'application. Le simulateur extrait le contenu des sections `.text` , `.data`, `.ktext`, `.kdata` et `.boot` de ces fichiers et initialise les segments concernés de la RAM
- Le simulateur permet de voir la trace d'exécution des instructions et les accès mémoire, mais il ne permet pas de voir l'évolution des registres.

Chaîne de compilation

Comment passer d'un programme C à un code binaire exécutable ?

→ 2 étapes : compilation et édition de liens

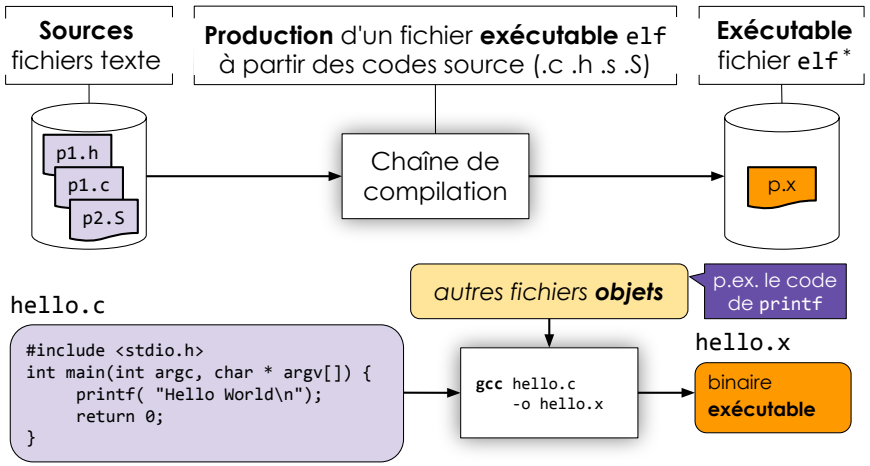
Comment imposer le placement du code et des données en mémoire ?

→ Grâce au fichier `ldscript` utilisé par l'éditeur de lien

Comment automatiser la production du code à partir des sources ?

→ En utilisant un Makefile

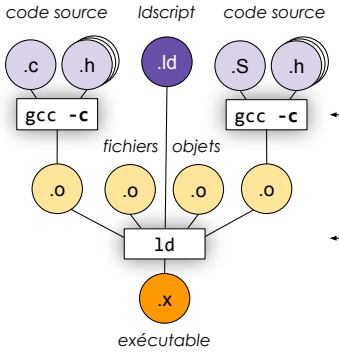
Chaîne de compilation



 * Executable and Linking Format

Étapes de la chaîne de compilation

Selon Wikipedia *Compiler un programme C consiste à transformer un code source [...] en un fichier binaire exécutable* ***mais il y a plusieurs étapes.***



Pour faire simple, on peut distinguer 2 grandes étapes :

1. La **compilation** (`.c + .h`)→(`.o`) **ou** (`.S + .h`)→(`.o`)
   - Préprocessing (*transformation du code source*)
     - Inclusion des .h (récursif) contenant les déclarations des variables et des fonctions.
     - Expansion des macro-instructions (`#define`)
   - Compilation et génération du code binaire
     - Analyse syntaxique et optimisations
     - Génération du code objet (code binaire de la cible)
2. L'**édition des liens** (`.o + .ld`) → (`.x`)
   - Fusion de tous les fichiers objets pour produire un fichier binaire exécutable

Compilation native et Compilation croisée

Compilation native

C'est lorsque le compilateur produit du code pour la machine et le système d'exploitation où est fait la compilation :

⇒ `gcc`, `as` (assembleur), `ld` (éditeur de liens), `objdump` (désassembleur)

Donc vous compilez sur votre PC pour exécuter sur votre PC

Compilation croisée

C'est lorsque le compilateur produit du code pour une autre machine ou un autre système d'exploitation que la machine où est faite la compilation :

nom : `cpu-os-format-tool` ⇒ Pour la plateforme utilisé en TP

- assembleur `as` ⇒ `mipsel-unknown-elf-as`
- compilateur `gcc` ⇒ `mipsel-unknown-elf-gcc`
- linker `ld` ⇒ `mipsel-unknown-elf-ld`
- désassembleur `objdump` ⇒ `mipsel-unknown-elf-objdump`

Ici vous compilez sur un PC Intel pour exécuter sur un SoC MIPS

Préprocessing du C

Le préprocesseur transforme le code C et produit un nouveau code C

- efface les commentaires
- interprète les directives : ***#directive***

Il y a 4 usages des directives *

1. Expansion de macro instructions (`#define, #undef`, *etc.*)
   permet le remplacement d'un identifiant (macro) par sa définition.
   Ces définitions peuvent être paramétrées avec des arguments
2. Inclusion de fichiers (`#include`)
   permet d'inclure des déclarations de fonctions, de types, de variable ou des définitions de macros dans un fichier .c, .S ou .h
3. Compilation conditionnelle (`#if, #ifdef, #else, #elif, #endif`, *etc.*)
   permet de supprimer une partie des lignes de code source dans certaines conditions
4. Directives pour le compilateur (`#warning, __FILE__`, *etc.*)
   permet d'informer ou d'interroger les variables internes du compilateur

 ** voir le détails de ces usages en annexe*

Langage assembleur

(https://sourceware.org/binutils/docs/as/)

Directives que vous les connaissez déjà :

- `.globl` *label* : indique que ce label est visible des autres fichiers
- `.word .ascii .asciiz .byte` : permet d'allouer de la place, c'est suivi des valeurs
- `.space` *size* : alloue de la place non initialisée
- `.align` *n* : déplace le ptr. de remplissage à la prochaine adresse 2^n
- `.text .data` : indique la section à remplir

Directive pour créer une section dans le fichier objet (ici avec le nom ***name***) :

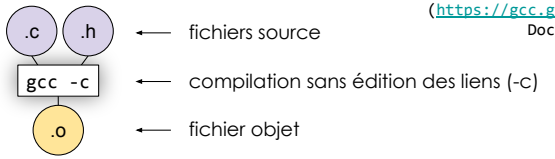
Par défaut, le code est mis dans une section `.text` et les données une section `.*data*`, mais il est possible de créer d'autres sections pour contrôler précisément le placement en mémoire

```
.section name[, "flags"][, "type"]
                flags   : wax  ⇒ writable, allocated, executable
                type    : @progbit | @nobits ⇒ resp. avec et sans data
```

Et les deux macros que vous pourrez désormais utiliser :

- `la $r, label` : $r ← *label* (la = load address) l'assembleur remplace `la` par `lui+ori`
- `li $r, imm32` : $r ← *imm32*, (li = load immediate) l'assembleur remplace `li` par `lui+ori`

Compilation du langage C



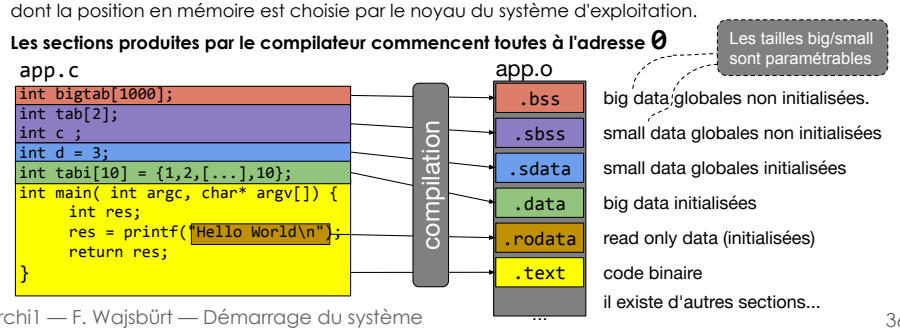
(https://gcc.gnu.org/onlinedocs/gcc-7.1.0/gcc/)
Doc de la version 7.1 utilisée en TP

- Le compilateur produit un code binaire (`.o`) pour un processeur cible à partir du code source C (`.c`) ou assembleur (`.s`) déjà « *préprocessé* »
- La compilation se déroule en plusieurs étapes :
  - les analyses : lexicale, syntaxique et sémantique
  - la génération d'un code intermédiaire (indépendant du processeur)
  - les optimisations pour gagner en vitesse ou en taille
  - la génération du code binaire pour le processeur cible (`.o`)
- **Le fichier généré n'est pas exécutable** parce qu'il manque des choses.
  - En effet, le code des fonctions telles que `printf()` n'est pas dans le fichier produit par le compilateur, parce que ce code est ailleurs.
  - Les adresses dans le fichier objet sont "fausses" parce que les sections ne sont pas encore à leur place dans l'espace d'adressage.

Fichier objet

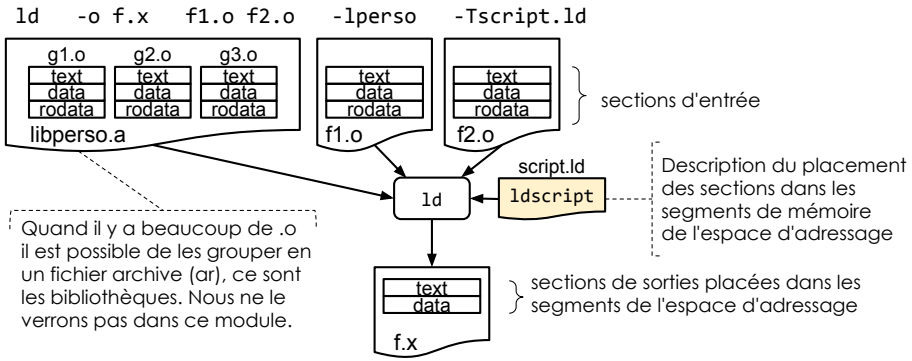
Le compilateur met le code et les données globales dans des sections typées du fichier objet

- Une section est un segment d'adresses destinée à contenir une catégorie d'information
  - Le code est dans une section `.text`
  - Les données globales sont placées dans différentes sections (`.*data*`, `.*bss*`, etc.) en fonction de leur taille et du fait qu'elles sont initialisées ou pas.
  - Les données globales non explicitement initialisées dans le code de l'application, sont initialisées à 0 au lancement de l'application.
- Il n'y a pas de sections pour les variables locales (tel que `int res` dans l'exemple) car ce sont des données qui n'existent qu'à l'exécution du programme, et qui seront placées dans la pile d'exécution dont la position en mémoire est choisie par le noyau du système d'exploitation.
- **Les sections produites par le compilateur commencent toutes à l'adresse 0**



Edition de liens

Le compilateur produit des fichiers objets (`.o`) avec du code binaire incomplet,
les sections ne sont pas placées dans l'espace d'adressage par le compilateur lui-même
et donc les adresses de saut ou de variables globales dans le `.o` ne sont pas connues.
⇒ **Il faut lier les `.o` (les réunir) pour produire un fichier exécutable complet (au format elf).**



Edition de liens : fichier ldscript

```
__tty_regs_map   = 0xd0200000 ;
__boot_origin    = 0xbfc00000 ;
__boot_length    = 0x00001000 ;
__ktext_origin   = 0x80000000 ;
__ktext_length   = 0x00020000 ;
__kdata_origin   = 0x80020000 ;
__kdata_length   = 0x003E0000 ;
__kdata_end      = __kdata_origin + __kdata_length;
```

déclaration des variables du ldscript

Nous verrons plus loin comment le code C et le code assembleur peuvent accéder aux valeurs des variables définies dans le fichier ldscript. Ce sera nécessaire par exemple pour définir la position initiale du pointeur de pile en `__kdata_end`

```
MEMORY {
    boot_region  : ORIGIN = __boot_origin,  LENGTH = __boot_length
    ktext_region : ORIGIN = __ktext_origin, LENGTH = __ktext_length
    kdata_region : ORIGIN = __kdata_origin, LENGTH = __kdata_length
}
```

Le fichier **ldscript** décrit
la manière de remplir la mémoire,
c'est un script pour l'éditeur de liens ld.

Il y a plusieurs syntaxes possibles. Il faut comprendre que le ldscript décrit l'espace d'adressage et comment celui-ci est rempli avec les sections produites par le compilateur.

région est le nom donné par l'éditeur de liens aux segments de l'espace d'adressage : région = segment

```
SECTIONS {
    .boot : {
        *(.boot)
    } > boot_region
    .ktext : {
        *(.text*)
    } > ktext_region
    .kdata : {
        *(.*data*)
        . = ALIGN(4);
        __bss_origin = .;
        *(.*bss*)
        . = ALIGN(4);
        __bss_end = .;
    } > kdata_region
}
```

description des régions de l'espace d'adressage

Pour chaque, il y a l'adresse d'origine et la taille en octet

description du remplissage des sections de sortie (en rouge) avec les sections d'entrée (en bleu) produites par le compilateur et prises dans l'ordre et placement dans les régions (en vert)

Edition de liens : fichier ldscript

```
__tty_regs_map   = 0xd0200000 ;
__boot_origin    = 0xbfc00000 ;
__boot_length    = 0x00001000 ;
__ktext_origin   = 0x80000000 ;
__ktext_length   = 0x00020000 ;
__kdata_origin   = 0x80020000 ;
__kdata_length   = 0x003E0000 ;
__kdata_end      = __kdata_origin + __kdata_length;
```

déclaration des variables du ldscript

Les noms choisis pour ces variables sont quelconques mais ils sont préfixés par convention par un double underscore « _ _ » pour éviter les conflits de noms avec les variables ou fonctions du programmes.
(https://www.gnu.org/software/libc/manual/html_node/Reserved-Names.html)

```
MEMORY {
    boot_region  : ORIGIN = __boot_origin,  LENGTH = __boot_length
    ktext_region : ORIGIN = __ktext_origin, LENGTH = __ktext_length
    kdata_region : ORIGIN = __kdata_origin, LENGTH = __kdata_length
}
```

```
SECTIONS {
    .boot : {
        *(.boot)
    } > boot_region
    .ktext : {
        *(.text*)
    } > ktext_region
    .kdata : {
        *(.*data*)
        . = ALIGN(4);
        __bss_origin = .;
        *(.*bss*)
        . = ALIGN(4);
        __bss_end = .;
    } > kdata_region
}
```

syntaxe : fichier-objet(section) * est un caractère *joker* qui désigne tous les fichiers

définition : concaténation dans la section de sortie `.boot` de l'ensemble des sections `.boot` trouvées dans tous les fichiers objets (puisqu'on a mis une *) donnés en argument à l'éditeur de liens et placement dans la région boot_region

section de sortie (en rouge)

représente le pointeur d'adresse de remplissage dans la région courante

Les sections d'entrée (en bleue) remplissent la section de sortie (en rouge)

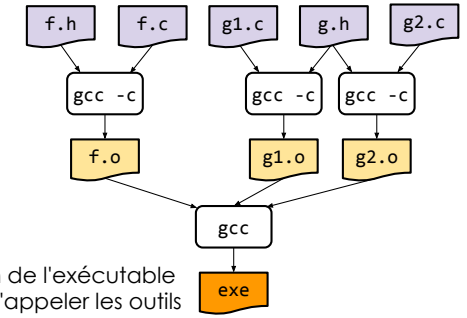
Le pointeur d'adresse de remplissage est déplacé pour être multiple de 2^4

région (nous mettons ici une seule section de sortie par région)

Compilation séparée

On veut faire la compilation de plusieurs fichiers sources indépendants
Puis les réunir en un seul fichier exécutable : exe

Ce graphe représente les dépendances de exec et comment l'obtenir:



- `exe` dépend de `f.o`, `g1.o` et `g2.o`
  et pour l'obtenir il faut utiliser `gcc`
  (il n'y a pas, ici, les arguments de `gcc`*)*
- `f.o` dépends de `f.c` et `f.h`
  et pour l'obtenir il faut aussi utiliser `gcc`
  (il n'y a pas, ici, les arguments de gcc)
- etc.

Pour décrire la méthode de construction de l'exécutable
On utilise des Makefiles qui permettent d'appeler les outils de compilation dans le bon ordre, vous les verrez en TP
Il y a quelques détails en annexe de ce cours que vous êtes invités à lire 😉

Compilation des codes source OS et app

```
.
├── Makefile
├── common
│   └── syscalls.h
├── kernel
│   ├── Makefile
│   ├── harch.c
│   ├── harch.h
│   ├── hcpu.h
│   ├── hcpua.S
│   ├── hcpuc.c
│   ├── kernel.ld
│   ├── kinit.c
│   ├── klibc.c
│   ├── klibc.h
│   └── ksyscalls.c
├── uapp
│   ├── Makefile
│   └── main.c
└── ulib
    ├── Makefile
    ├── crt0.c
    ├── libc.c
    ├── libc.h
    └── user.ld
```

Nous n'allons pas détailler le fonctionnement des outils de compilation et des makefiles, mais vous aurez le code source commenté que vous pourrez lire.

- A gauche se trouve la version la plus complexe du code du dernier TP.
- En dessous, un extrait de la séquence des commandes invoquées par le Makefile hiérarchique placé à la racine (*en haut de la liste à gauche*)

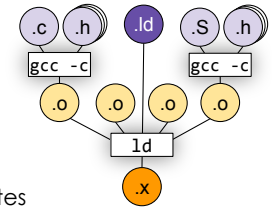
```
make -C kernel compil NTTYS=1 NCPUS=1
    makedepend [...] sources
    mipsel-unknown-elf-gcc -o obj/hcpua.o -c [...] hcpua.S
    mipsel-unknown-elf-gcc -o obj/ksyscalls.o -c [...] ksyscalls.c
    mipsel-unknown-elf-gcc -o obj/harch.o -c [...] harch.c
    mipsel-unknown-elf-gcc -o obj/hcpuc.o -c [...] hcpuc.c
    mipsel-unknown-elf-gcc -o obj/klibc.o -c [...] klibc.c
    mipsel-unknown-elf-gcc -o obj/kinit.o -c [...] kinit.c
    mipsel-unknown-elf-ld -o ../kernel.x -T kernel.ld kernel_objets
make -C ulib compil NTTYS=1 NCPUS=1
    makedepend [...] sources
    mipsel-unknown-elf-gcc -o obj/libc.o -c [...] libc.c
    mipsel-unknown-elf-gcc -o obj/crt0.o -c [...] crt0.c
make -C uapp   compil NTTYS=1 NCPUS=1
    makedepend [...] sources
    mipsel-unknown-elf-gcc -o obj/main.o -c [...] main.c
    mipsel-unknown-elf-ld -o ../user.x -T ../ulib/user.ld user_objets

almo1.x -KERNEL kernel.x -APP user.x -NTTYS 1 -NCPUS 1
```

compilation du **noyau** de l'OS (système d'exploitation)

compilation des bibliothèques système et de l'application

Ce qu'il faut retenir



- La chaîne de compilation comporte 2 étapes importantes
  1. Compilation des codes sources en code objet avec une étape préliminaire de traitement des macro-instructions
  2. Édition des liens des fichiers objets pour produire l'exécutable
- L'éditeur de liens a besoin d'une description des régions de la mémoire et de la manière de remplir ces régions par les sections des fichiers objets
- Le Makefile sert à décrire comment est fabriqué un exécutable à partir des sources et plus généralement, il sert à automatiser les commandes shell des étapes de fabrication du binaire exécutable à partir des sources.

Conclusion

Ce que nous avons vu

et descriptions des expériences

Nous avons vu

- qu'un SoC contient au minimum un processeur, une mémoire et un contrôleur de périphérique.
- que les composants sont accessibles par des load/store dans l'espace d'adressage
- que les programmes exécutables sont obtenus par la chaîne de compilation gcc en deux étapes importantes : compilation et édition de liens
- que l'espace d'adressage est décrit dans un fichier ldscript
- qu'un Makefile permet de décrire la méthode de construction d'un exécutable
- qu'une API est un interface qui définit un contrat, dans notre cas sous forme d'un fichier .h
- que le système d'exploitation implémente une API (POSIX) pour l'écriture des applications
- que le système d'exploitation est responsable de l'allocation des ressources matérielles et logicielles nécessaires aux applications.
- que le système est composé d'un noyau et de bibliothèques système (p. ex. libc POSIX)
- que le noyau contient 3 gestionnaires : syscall, exception et interruption
- que le MIPS démarre à l'adresse `0xBFC00000`
- que le noyau définit une API de pilote pour la commande des périphériques
- que le SoC est modélisé dans un prototype virtuel simulable cycle par cycle depuis le reset
- que le simulateur charge les mémoires avant d'exécuter les instructions

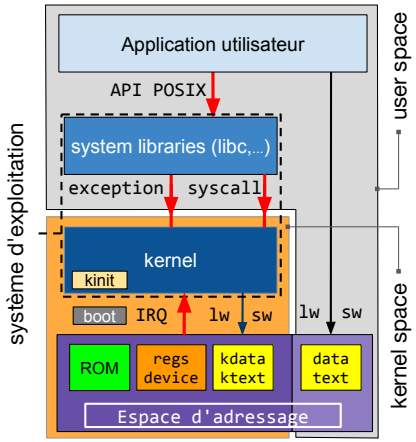
Quelles sont les expériences en TME ?

Principes

- L'idée est que vous compreniez le mieux possible comment fonctionne un ordinateur du point de vue du logiciel en regardant comment il utilise le matériel (processeur, mémoire et périphériques).
- Pour cela, la méthode choisie est que vous suiviez le démarrage d'une application utilisateur depuis le signal de démarrage du processeur :
  → boot → kernel initialisation → application → kernel (appels système) → application ...
- Le système d'exploitation que vous allez suivre n'est pas Linux, c'est un tout petit système, mais même petit, il ne serait pas raisonnable de vous le faire écrire.
- Les TP sont composés de quelques étapes (moins de 5) qui introduisent chacune un concept ou un mécanisme avec le moins de code possible.
- Pour chaque étape, vous avez tous les codes source et tous les Makefile fonctionnels
- Vous aurez des questions dont les réponses sont dans le code ou dans les cours ou les TD
- Vous devrez modifier le code pour ajouter une fonctionnalité très simple, mais qui doit vous permettre de vous approprier le code

Le simulateur tourne sur Linux, vous pourrez travaillez à la PPTI, mais vous avez aussi une archive qui devrait tourner sur toutes les distributions de Linux sur machine réelle ou virtuelle (VirtualBox)

Objectifs pour cette UE



Pour cette UE :

Nous construisons **progressivement** un embryon de système d'exploitation avec :

1. une mini-**libc** pseudo-POSIX
2. un petit **kernel** avec
   - ses 3 gestionnaires de services (syscall, exception, et interruption)

Au début pour le premier TP :

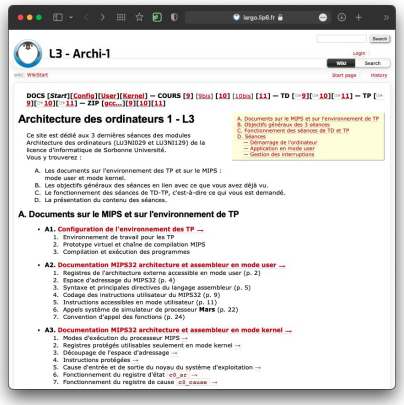
Nous allons juste voir comment le système démarre. Nous allons présenter 4 parties :

1. code de démarrage du système (**boot**)
2. fonction d'initialisation du kernel (**kinit**)
3. code d'accès au périphérique TTY
4. bibliothèque de fonctions standards pour le kernel

Ces parties sont volontairement très simples, Il n'y a pas d'application à ce niveau de construction de l'OS et donc pas de bibliothèques système.

Ressources et préparation des TME

Site dédié pour cette partie de l'UE :
https://www-soc.lip6.fr/trac/archi-l3s5

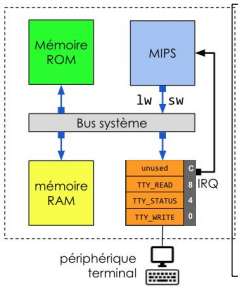


Vous y trouverez :

A. les documentations sur le **MIPS** et la **config**.
   - Documentation MIPS32 architecture et assembleur en **mode user**
   - Documentation MIPS32 architecture et assembleur en **mode kernel**
   - Configuration de l'**environnement des TP**
B. les **objectifs généraux** des séances en lien avec ce que vous avez déjà vu dans la première partie de l'UE ;
C. une explication du **principe pédagogique** utilisé pour présenter l'architecture
D. le **fonctionnement des séances** de TD-TP, c'est-à-dire ce qui vous ait demandé
E. les liens vers les slides des **cours** en PDF
F. les liens vers les textes de **TD et TP**.

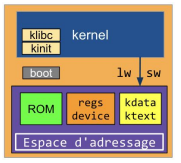
Vous devez lire ces pages avant les TD...

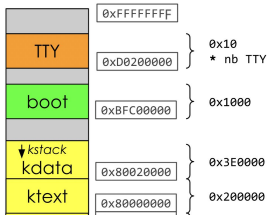
TD + TME 1




```
k06
├── bin/
│   ├── almo1.x*        simulator
│   ├── gcc/            compiler
│   ├── soclib-fb*      frame-buffer
│   ├── Source-me.sh    bash environment
│   └── tracelog        trace debug
└── tp1
    ├── 0_test_alom1/   to test the env.
    ├── 1_hello_boot/   boot asm
    ├── 2_init_asm/     boot + kinit asm
    ├── 3_init_c/       boot + kinit c
    ├── 4_nttys/        print message
    ├── 5_driver/       primo-driver tty
    └── Makefile        to clean all
```

Seuls les segments du kernel sont utilisés dans ce 1er TME






```
1_hello_boot/
├── hcpua.S
├── kernel.ld
└── Makefile
```

Le code est très commenté, vous aurez des questions sur ce code et vous devrez ajouter quelques lignes

hcpua.S:

```
/*-------------------------------------------------------------------------*\
 |_ /'v'\ /_/   \date      2021-11-06
 / /( )\ / _ \   \copyright 2021 Sorbonne University https://opensource.org/licenses/MIT
 |_\ \__/ \_/
\*-------------------------------------------------------------------------*/

//-------------------------------------------------------------------------
// Boot code
//-------------------------------------------------------------------------

.text                       // section for this code

boot:                       // must be 0xBFC00000 for the MIPS

    la      $4, hello       // $4 <- address of string: "hello..."
    la      $5, __tty_regs_map  // $5 <- address of tty's registers map (cf. ldscript)

print:

    lb      $8, 0($4)       // get current char
    sb      $8, 0($5)       // send the current char to the tty
    addiu   $4, $4, 1       // point to the next char
    bne     $8, $0, print   // check that it is not null, if yes it must be printed

dead:                       // infinite loop (nothing else to do)

    j       dead

hello:
    .asciiz "Hello World!\n"    // string to print (put for now in .text section)
    .asciiz "Type <CTRL-C> on Terminal to stop the simulator\n"
```