

Démarrage

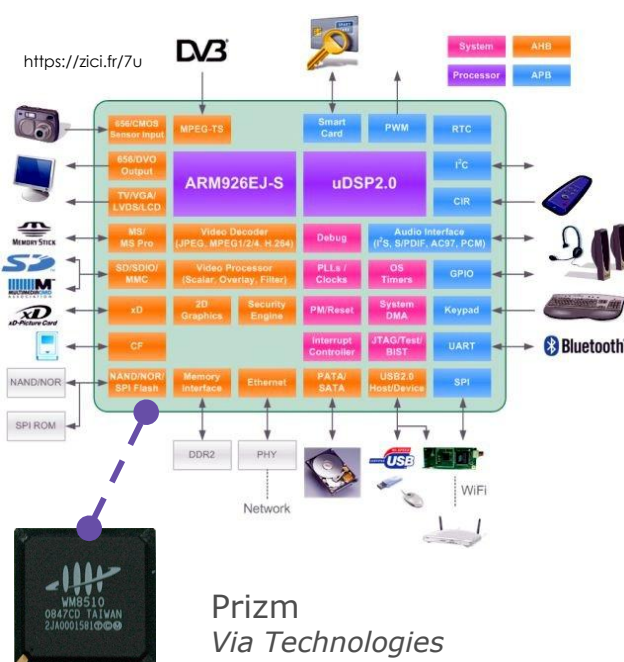
LU3IN029 Archi L3 S5

franck.wajsburt@lip6.fr

LU3IN029 — 2021 — Démarrage de l'ordinateur

1

SoC



- Avec la miniaturisation un seul chip peut contenir un système entier : ce sont des System on Chip ou SoC.
- Les SoC sont présents 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 la représentation de son interface vers ses périphériques.
- Le dernier né d'Apple (le M1max) à base de core ARM contient ... 57 milliards de transistors !

LU3IN029 — 2021 — Démarrage de l'ordinateur

2

Questions du jour



Tous les SoC ne sont pas aussi grands, il y a toutes les tailles

→ Que trouve-t-on dans un SoC au minimum?

Vous utilisez le langage C avec un compilateur pour produire un exécutable,

→ Quelles sont les étapes pour produire un exécutable ?

Les applications que vous écrivez s'appuient sur un système d'exploitation (OS)

→ Quelle est la place et le rôle d'un OS ?

Enfin, vous allez exécuter vos programmes sur un prototype virtuel de SoC

→ Qu'est-ce que cela signifie pratiquement ?

Plan

- Architecture d'un SoC minimal
- Chaîne de compilation C
- Système d'exploitation
- Prototypage virtuel

Architecture d'un SoC minimal

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

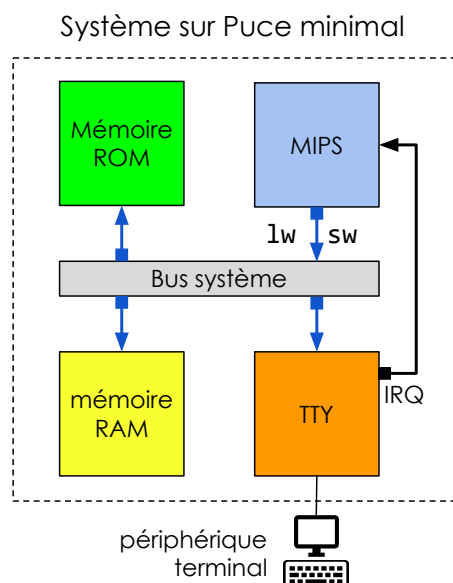
→ Un processeur, une mémoire, et quoi d'autre ... ?

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

→ Est-ce la même chose ?

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*)



Un cœur 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)

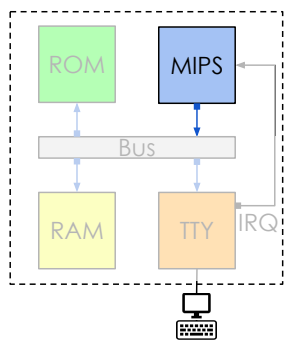
contenant le code du programme, ses données et sa (ou ses) pile(s) d'exécution de fonctions

Un contrôleur de TTY (terminal)

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



0x80000000 → 0xFFFFFFFF
segments pour le noyau
pour son code et ses
données et le contrôle
des périphériques

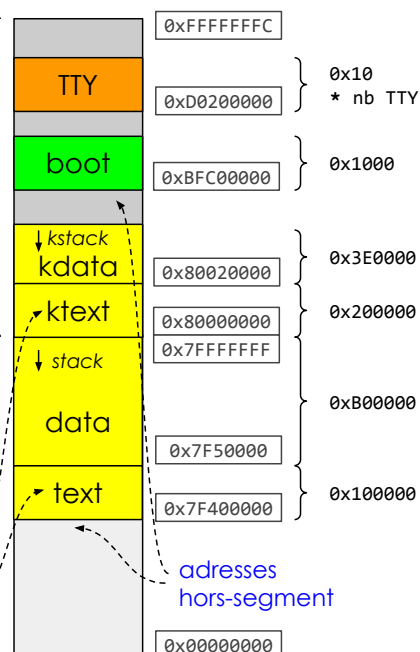
L'espace d'adressage
est coupé en 2 parties

0x00000000 → 0x7FFFFFFF
segment pour les apps
de l'utilisateur

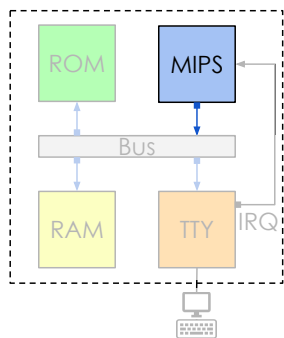
segments

Espace d'adressage du prototype

L'espace d'adressage est l'ensemble des adresses que peut produire le MIPS32. Les segments d'adresses représentés ici sont ceux du prototype utilisé en TP.



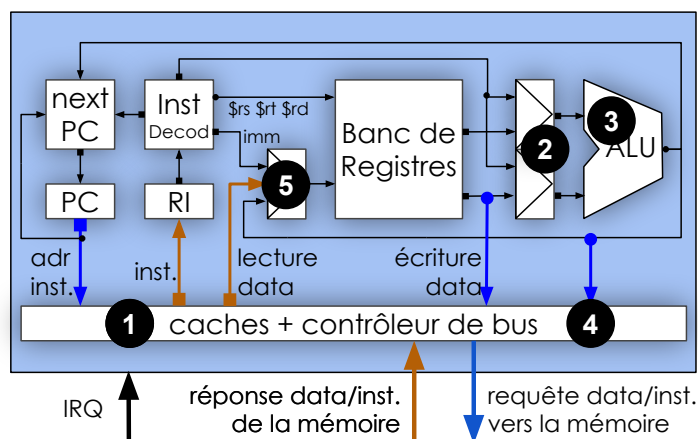
- Le segment de **boot** commence en 0xBFC00000. Cette adresse est imposée par le MIPS. Le segment fait 0x1000 octets (4 kio)
- Le segment des **TTY** commence en 0xD0200000. C'est un segment de 16 octets par TTY.
- ktext** et **kdata** sont les segments utilisés par le noyau du système d'exploitation
 - ktext** commence en 0x80000000 et sa taille est 0x200000 (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.



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.

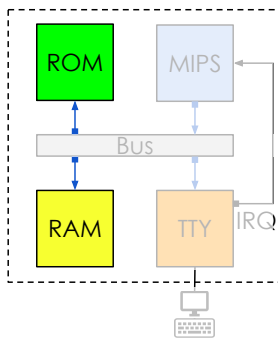


Le MIPS a **2 modes** d'exécution : **kernel** et **user**

- 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 binaires vues au début du module 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



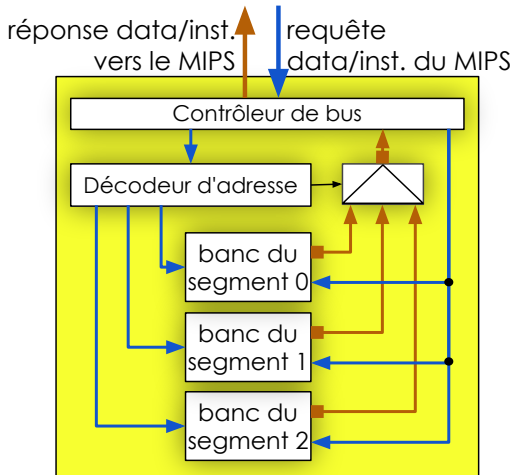
Mémoires RAM et ROM

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).

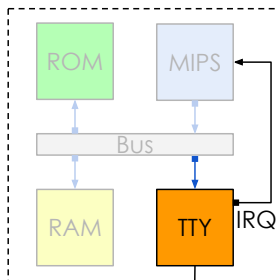
Les requêtes contiennent

- 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 :

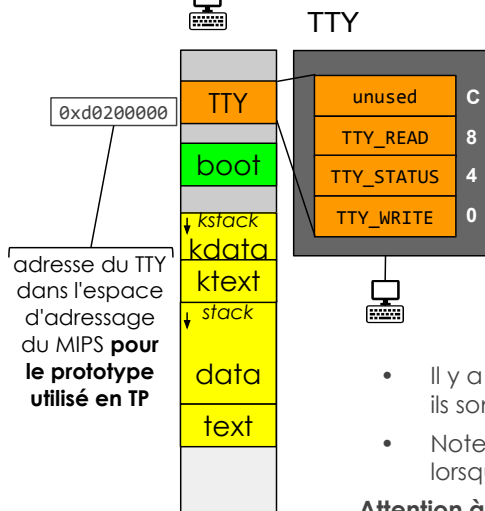
- une donnée sur 32 bits si c'est une lecture (la sélection du ou des octets concernés par la lecture est faite par le MIPS lui-même).
- une acquittement si c'est une écriture.



Périphériques (Devices)

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

- Le TTY est un périphérique d'entrées-sorties texte (écran-clavier)
- Les périphériques contiennent des registres de contrôle qui sont placés dans l'espace d'adressage du MIPS pour qu'il puisse les accéder par 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

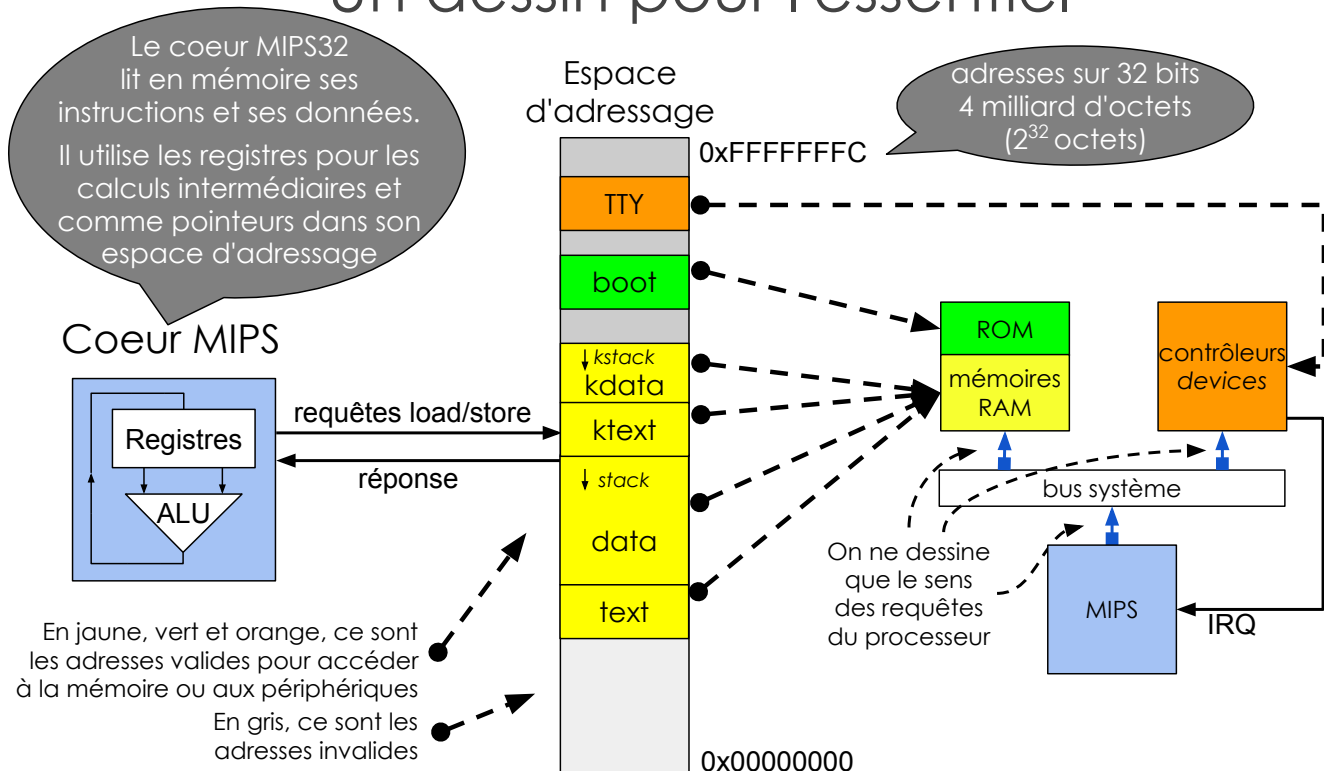
0xd0200008 C'est un registre **en lecture seule**.

Il contient le code ASCII du dernier caractère tapé au clavier.

- Il y a autant de jeu de 4 registres que de 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 !

Un dessin pour l'essentiel



Ce qu'il faut retenir

Un SoC (ordinateur) est composé au minimum

- d'un processeur pour exécuter le programme, ici ce sont des MIPS
- d'une mémoire RAM contenant un ou plusieurs bancs de mémoire représentant des segments de l'espace d'adressage du processeur :
 - pour le code du programme utilisateur et pour le système d'exploitation;
 - pour les données du programme utilisateur et du système d'exploitation;
 - et pour les piles d'exécution des fils d'exécution des programmes.
- d'une mémoire ROM contenant le code de démarrage du processeur.
- 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.

Chaîne de compilation

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

→ On doit compiler... mais ça veut dire quoi au juste ?

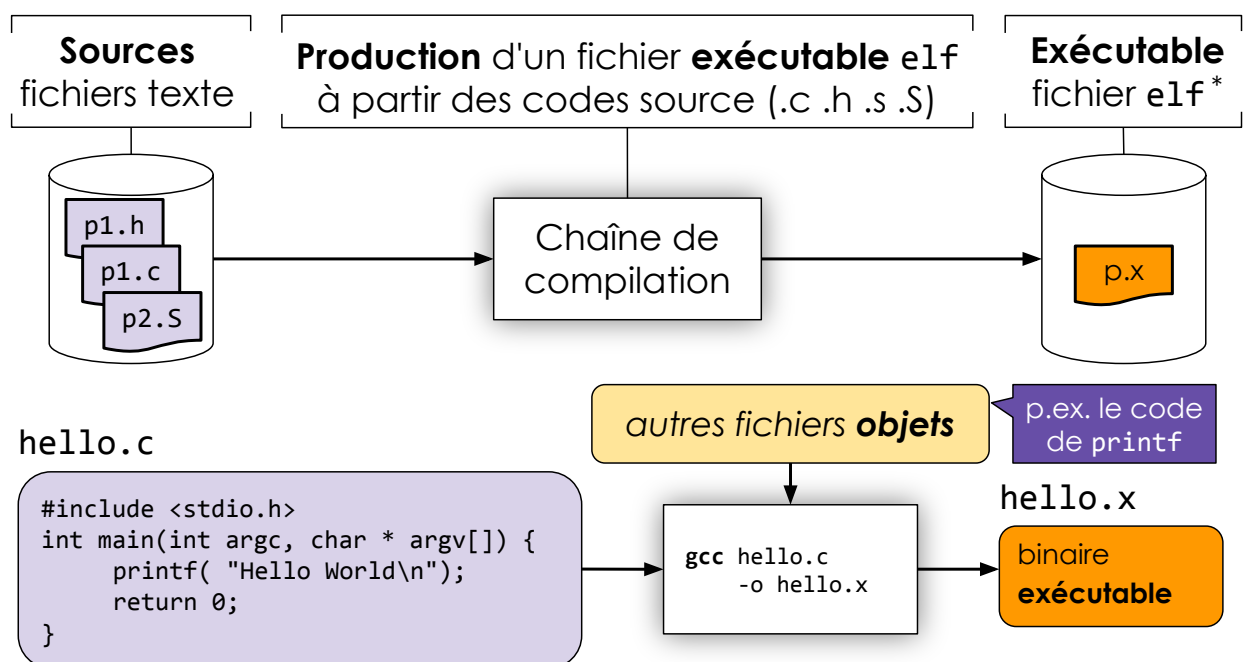
Comment imposer le placement du code et des données en mémoire ?

→ Les adresses sont celles du matériel, comment les imposer au compilateur ?

Comment automatiser la production du code à partir des sources ?

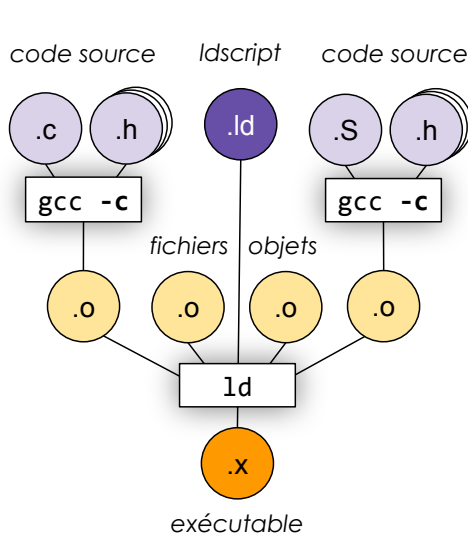
→ Comment faire quand il y a plusieurs fichiers source ?

Chaîne de compilation



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`)
 - Unification de tous les fichiers objets pour produire un exécutable

Chaîne de compilation GNU

GNU propose une chaîne d'outils de compilation permettant de produire un exécutable à partir de programmes source :

`gcc -c` préprocesseur + compilateur (`gcc -c file.c → .o`)
`ld` Éditeur de liens (ou `gcc file.o → .x`)

mais pas seulement, il y a d'autres outils :

`cpp` préprocesseur seul (ou `gcc -E file.c → .c`, idem pour `.S`)
`as` Assembleur seul (ou `gcc -S file.c → .s`)
`objdump` désassembleur (`elf .o` ou `.x → .s`)
`gdb` debugger (pour exécuter en pas à pas à condition que le code ait été compilé avec l'option `-g`, cela nécessite l'intervention du système d'exploitation)
`nm` liste des symboles présents dans un fichier au format elf
`readelf` affichage du contenu d'un fichier au format elf

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, ld, objdump`

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 | <code>as</code> | ⇒ <code>mipsel-unknown-elf-as</code> |
| • compilateur | <code>gcc</code> | ⇒ <code>mipsel-unknown-elf-gcc</code> |
| • linker | <code>ld</code> | ⇒ <code>mipsel-unknown-elf-ld</code> |
| • désassembleur | <code>objdump</code> | ⇒ <code>mipsel-unknown-elf-objdump</code> |

Donc vous compilez sur votre PC pour exécuter sur une autre machine, ici le MIPS

Préprocesseur du C

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

- efface les commentaires
- interprète les directives : **#directive** (le # doit être en début de ligne)

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 du compilateur

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` \Rightarrow `writable, allocated, executable`

`type` : `@prognobit` | `@nobits` \Rightarrow resp. avec et sans data

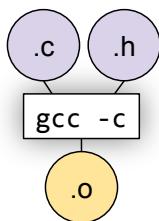
Et les deux macros que vous pourrez désormais utiliser :

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

Compilateur C

(<https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/>)

Doc de la version 7.2 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 en mémoire.

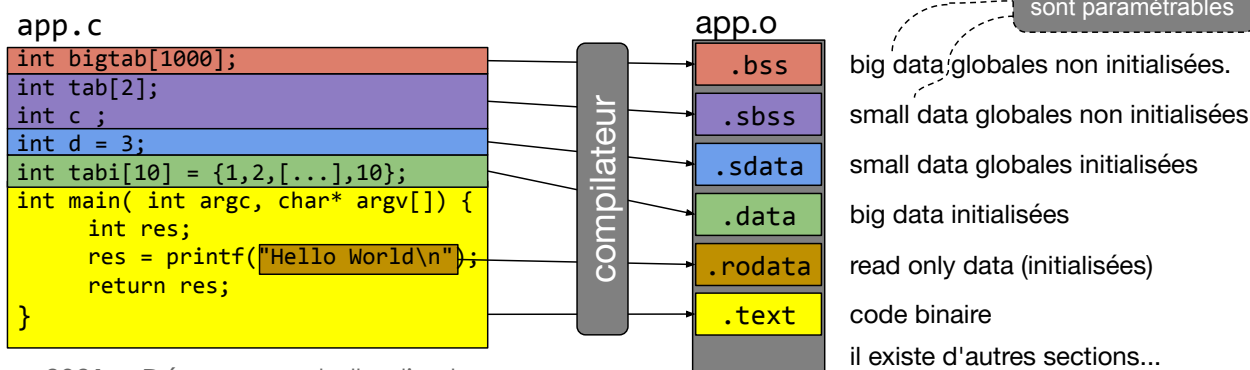
Compilateur C

Le compilateur place le code binaire et les données globales dans des sections typées

- 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`) car ce sont des données qui n'existent qu'à l'exécution du programme, et qui seront placées dans la pile d'exécution à une position en mémoire choisie par le noyau du système d'exploitation.

- Les sections produites par le compilateur commencent toutes à l'adresse 0**

Les tailles big/small sont paramétrables



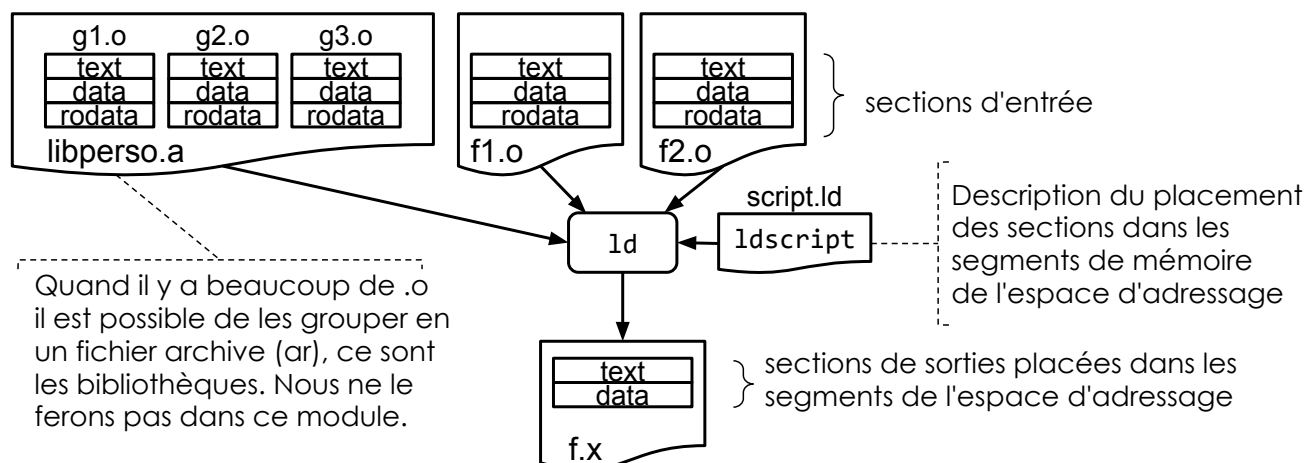
Edition de liens

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 unir) pour produire un fichier exécutable complet (au format elf).**

`ld -o f.x f1.o f2.o -lperson -Tscript.ld`



Edition de lien : 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 comment remplir la mémoire, c'est un script pour l'éditeur de lien ld.

```
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)

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

Edition de lien : 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 de 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 prend tous les fichiers

définition : concaténation dans la section de sortie **.boot** de l'ensemble des sections **.boot** trouvées dans tous les fichiers objets (puisque l'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 bleu) remplissent la section de sortie (en rouge)

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

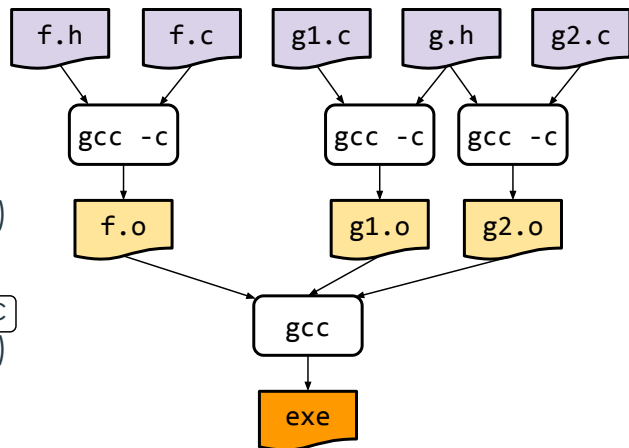
région (nous mettons ici une seule section de sortie par région)

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 **exe** 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** dépend 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.



Fichier Makefile & Commande make

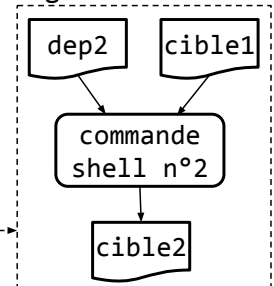
Un **Makefile** est un fichier contenant la méthode de construction d'un fichier cible à partir de ces sources. Un fichier **Makefile** est interprété par la commande **make**

La commande **make** a deux objectifs (l'objectif 1 est LE plus important)

1. Décrire la méthode de construction permettant de la rejouer complètement ou en partie après un changement des sources.
2. Permettre une reconstruction sélective en n'exécutant que les étapes de construction nécessaires pour produire la cible lorsque seule une partie des sources a été modifiée

Un fichier **Makefile** est constitué d'un ensemble de règles

règle de Makefile



Makefile

```
cible1 : dep1
← TAB → commandes shell n°1
cible2 : cible1 dep2
← TAB → commandes shell n°2
```

puisque cible2 dépend de cible1 alors

```
bash
$> make cible2
commandes shell n°1
commandes shell n°2
```

Vous verrez pendant les TME 3 manières d'utiliser un Makefile :

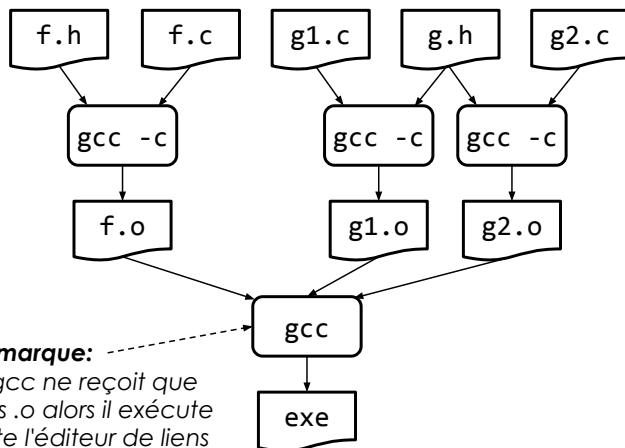
(1.) collection de scripts ; (2.) avec des règles explicites ; (3.) avec des règles implicites

1. Makefile : collection de *shell scripts*

Le Makefile peut contenir l'ensemble des *shell scripts* nécessaires à produire l'exécutable. Le Makefile contient alors autant de règles qu'il y a scripts à réaliser :

- 1 pour produire l'exécutable,
- 1 pour faire le ménage,
- 1 pour exécuter, etc.

L'appel de `make` sans cible prend par défaut la première cible



Remarque:
Si `gcc` ne reçoit que des `.o` alors il exécute juste l'éditeur de liens

```
help: # règle par défaut qui affiche une aide
@echo "compil : compilation exec"
@echo "clean : efface les .o"
@echo "cleanall : tout sauf les sources"
@echo "exec : test du programme"
```

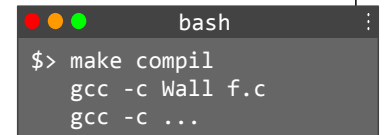
compil : # règle pour générer l'exécutable

```
gcc -c -Wall f.c
gcc -c -Wall g1.c
gcc -c -Wall g2.c
gcc f.o g1.o g2.o -o exec
```

exec : # règle pour lancer et tester l'exécutable
exe arg1 arg2

clean :
rm *.o

cleanall : clean
rm exe

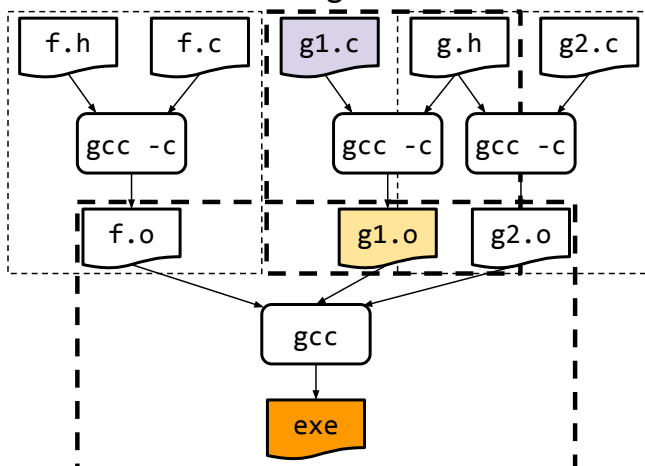


2. Makefile : règles explicites

make lit d'abord le Makefile entièrement, puis construit le graphe de construction, puis produit la cible demandée uniquement si sa date est antérieure à l'une de ces dépendances.

Ici la règle par défaut est **dep** qui n'est pas un fichier

4 règles



Makefile

```
dep: exe

all : clean dep

f.o : f.c f.h
    gcc -c -Wall f.c

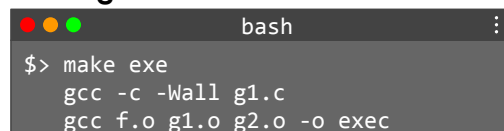
g1.o : g1.c g.h
    gcc -c -Wall g1.c

g2.o : g2.c g.h
    gcc -c -Wall g1.c

exe : f.o g1.o g2.o
    gcc f.o g1.o g2.o -o exec

clean :
    rm *.o
```

si seul **g1.h** a été modifié →



Compilation des codes source OS et app

```
.
├─ Makefile
├─ common
│  └─ syscalls.h
│  └─ thread.h
├─ kernel
│  └─ Makefile
│  └─ harch.c
│  └─ harch.h
│  └─ hcpu.h
│  └─ hcpua.S
│  └─ hcpuc.c
│  └─ kernel.ld
│  └─ kinit.c
│  └─ klibc.c
│  └─ klibc.h
│  └─ ksyscalls.c
│  └─ kthread.c
├─ uapp
│  └─ Makefile
│  └─ main.c
└─ ulib
   └─ Makefile
   └─ crt0.c
   └─ libc.c
   └─ libc.h
   └─ thread.c
   └─ 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 tp3.
- 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*)

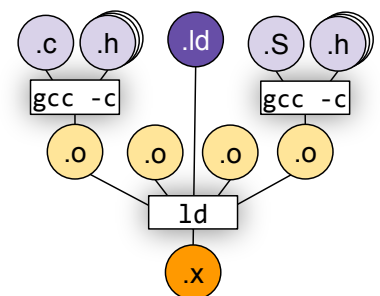
```
make -C kernel compil NTYS=1 NCPUS=1
makedepend [...] sources
mipsel-unknown-elf-gcc -o obj/hcpua.o -c [...] hcpua.S
mipsel-unknown-elf-gcc -o obj/ksyscalls.o -c [...] ksyscalls.c
mipsel-unknown-elf-gcc -o obj/harch.o -c [...] harch.c
mipsel-unknown-elf-gcc -o obj/hcpuc.o -c [...] hcpuc.c
mipsel-unknown-elf-gcc -o obj/klibc.o -c [...] klibc.c
mipsel-unknown-elf-gcc -o obj/kinit.o -c [...] kinit.c
mipsel-unknown-elf-gcc -o obj/kthread.o -c [...] kthread.c
mipsel-unknown-elf-ld -o ../kernel.x -T kernel.ld kernel_objets

make -C ulib compil NTYS=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
mipsel-unknown-elf-gcc -o obj/thread.o -c [...] thread.c

make -C uapp compil NTYS=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 -NTYS 1 -NCPUS 1
```

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

Système d'exploitation

A quoi sert le système d'exploitation (OS) et comment l'utiliser ?

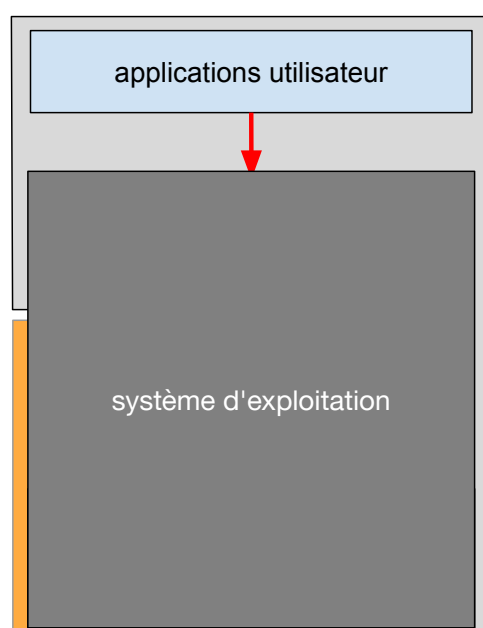
→ L'OS permet de gérer les ressources matérielles, il est accessible par une API standard et composé de plusieurs couches logicielles.

C'est quoi une couche logicielle et c'est quoi une API ?

→ Comment une application peut utiliser des fonctions définies ailleurs, dans une autre programme binaire ?

Et comment démarre un ordinateur ?

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 :

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

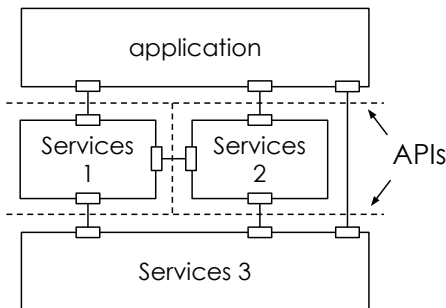
avec

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

Couches logicielles et API

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

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

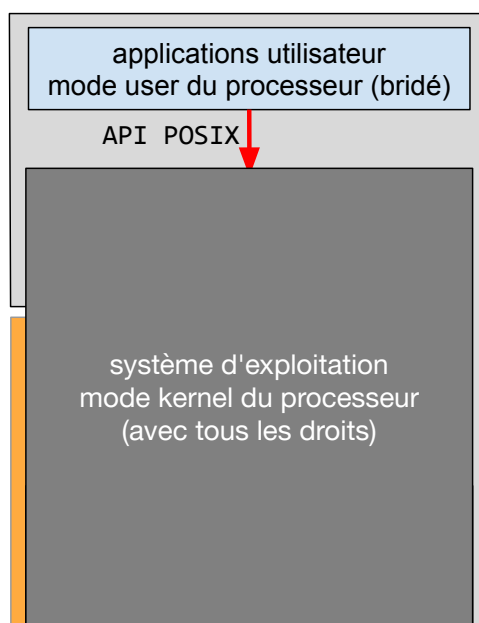


- Une couche logicielle rassemble un ensemble de services cohérents et propose une interface nommée API (Application Programming Interface) permettant d'accéder aux services.
- une API est un contrat définissant un mode d'emploi des services offerts par la couche logicielle.
- Une API en C est un ensemble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

Les APIs permettent (1.) de réduire la complexité des programmes, (2.) d'améliorer leur fiabilité et (3.) de simplifier leurs évolutions.

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.

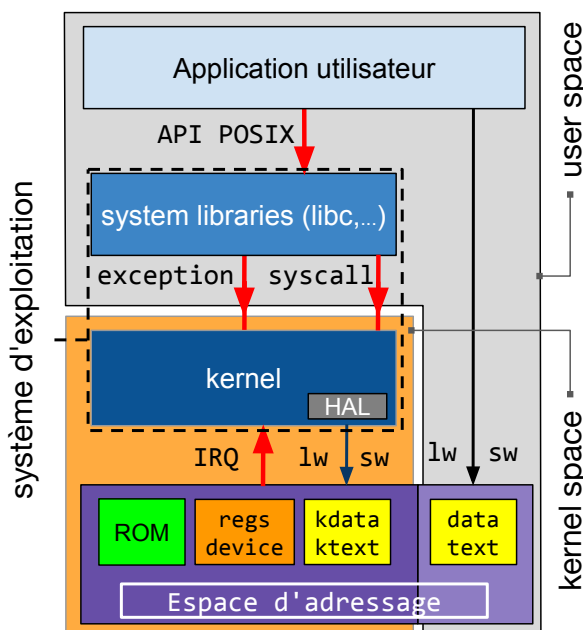
- POSIX est un ensemble d'APIs, vous connaissez déjà la libc (printf(), read(), etc.) mais il y en a d'autres comme les PThreads.

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

C'est possible grâce aux 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 = bibliothèques + kernel



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

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

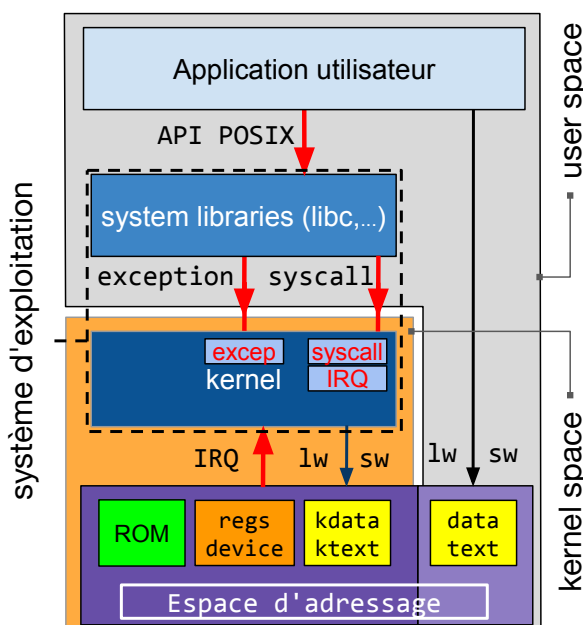
- l'**espace kernel** représente les ressources accessibles seulement en mode kernel.
- l'**espace user** représente les ressources accessibles à l'application, c.-à-d. les bibliothèques système et ses segments de code, de données et de pile.

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 implémentent 3 **gestionnaires** pour les **syscalls**, les **exceptions** et les **IRQ** des périphériques.

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

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

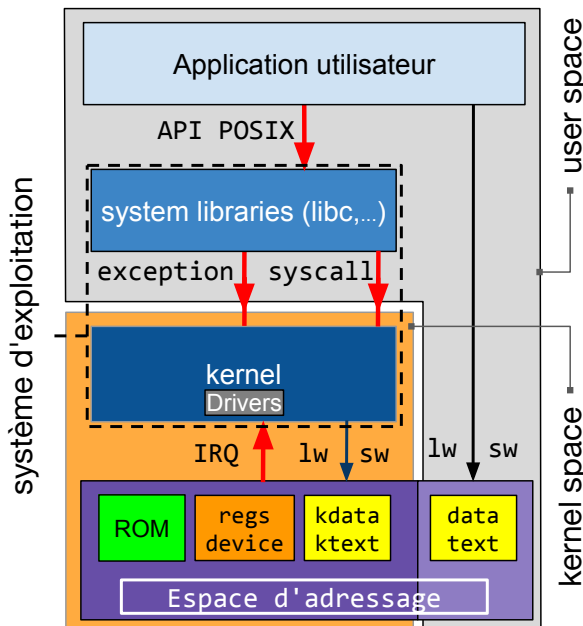
Le **gestionnaire d'exception** gère les erreurs du programme ou les allocations à la volée, telles que:

- 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).

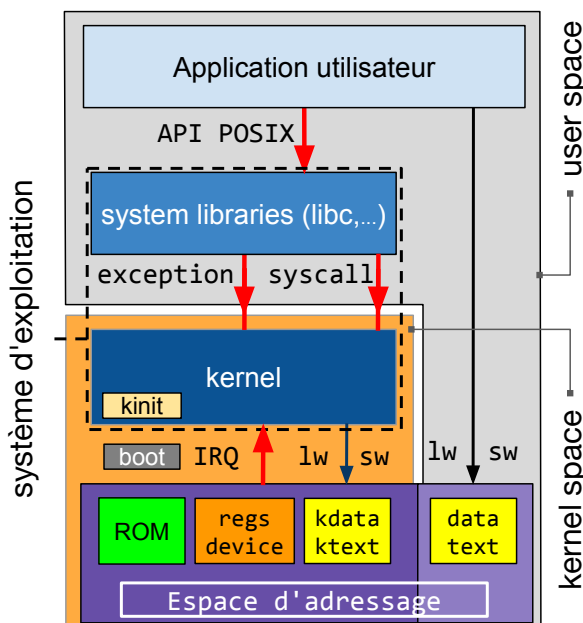
Un **Pilote** (Driver en anglais) est le code dans le noyau qui **contrôle un périphérique**, c'est-à-dire qui écrit les commandes, lit les réponses, traite les erreurs, et réagit aux IRQ (Interrupt ReQuests).

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 le 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.

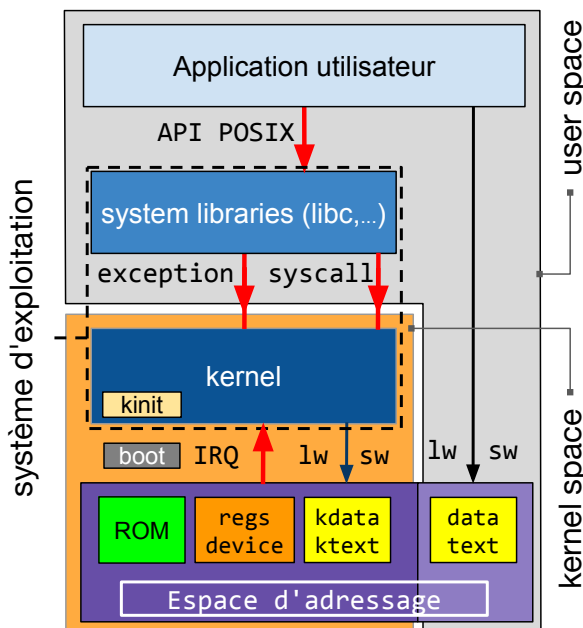
- Le code de boot est nécessairement dans une mémoire persistante, en ROM.
- Pour le MIPS, il est à l'adresse **0xBFC00000**
- Les premières instructions sont toujours en assembleur.
- Il doit au minimum initialiser le pointeur de pile pour pouvoir sauter dans la fonction d'entrée du noyau **kinit()**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

Dans un vrai système, le code de boot doit :

- scanner (découvrir) le matériel pour connaître 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 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



LU3NI029 — 2021 — Démarrage de l'ordinateur

Le prototype que vous allez utiliser :

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

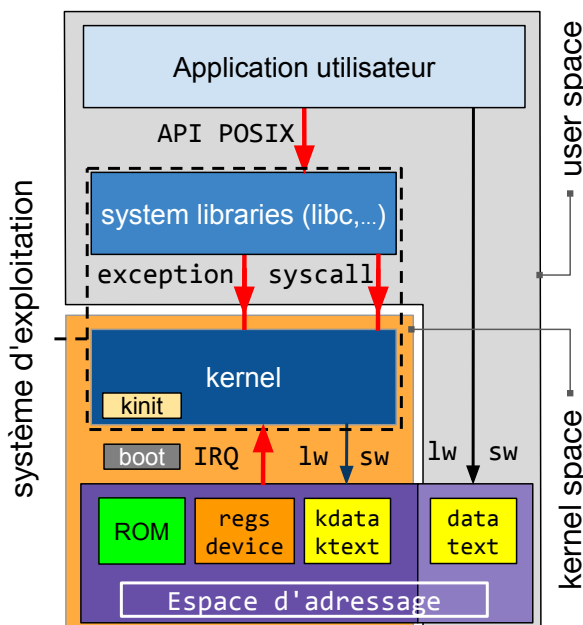
Le processus de boot doit donc être plus simple.

- 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 cores) 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 et ce n'est pas irréaliste, c'est ce qui se passe dans les micro-contrôleurs qui sont des SoC intégrant des 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 (p. ex. Arduino).

39

Objectif final & initial pour cette UE



LU3NI029 — 2021 — Démarrage de l'ordinateur

À la fin de 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 d'événements (**syscall**, **exception**, et **interruption**)
 - et un ordonnanceur de **threads**

Au début :

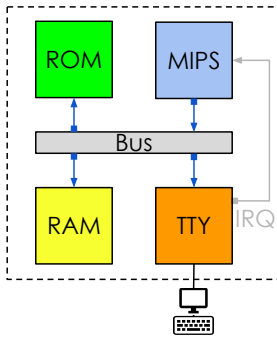
Nous allons juste voir comment le système démarre. Nous allons présenter 4 parties :

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 encore d'application donc pas de bibliothèques système.

40

Premier objectif

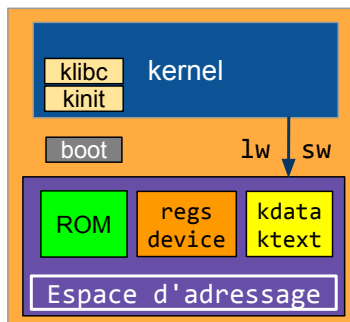


Les schémas représentent ce qui va être utilisé au début :

- Tous les composants sont utilisés mais pas les interruptions
- Il y a uniquement du code pour le noyau (pas d'application)

Dans la première étape

- La ROM contient un code de **boot** (écrit en assembleur) qui
 - initialise le pointeur de pile,
 - puis saute dans la fonction **kinit()** écrite en C. **kinit()** signifie **kernel init**, c'est l'entrée du noyau
- La RAM contient seulement le code du noyau avec :
 - La fonction d'initialisation du noyau : **kinit()**
 - Un pseudo-driver de TTY (**read()** et **write()**)
 - Une petite bibliothèque de fonctions standard pour le kernel : **klibc** (**printf()**, **memcpy()**, etc.)
 - Une fonction **game()** appelée par **kinit()** qui exécute un petit jeu utilisant le TTY.



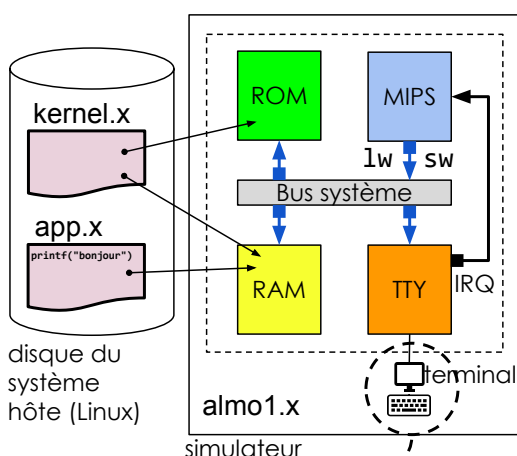
Ce qu'il faut retenir

- 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 un type de machine virtuelle précise au bit et au cycle
- Avec quoi allons nous charger les mémoires du prototype ?
→ avec le code et les données du noyau et de l'application, oui mais.
- Comment allons nous analyser l'exécution du code ?

Simulateur du SoC

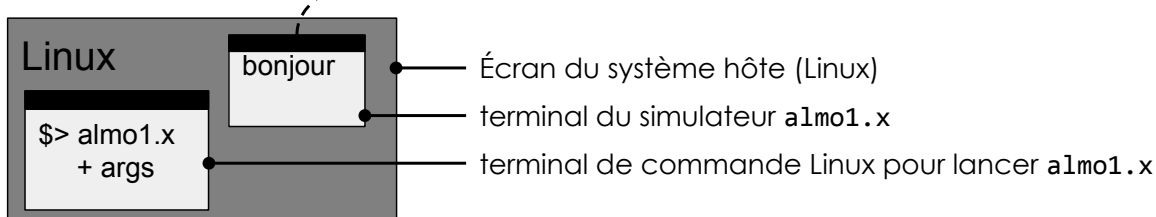


Le simulateur du SoC que nous utilisons se nomme **almo1.x**. Il est configurable, nous pouvons choisir le nombre de périphériques et de MIPS (ici c'est 1)

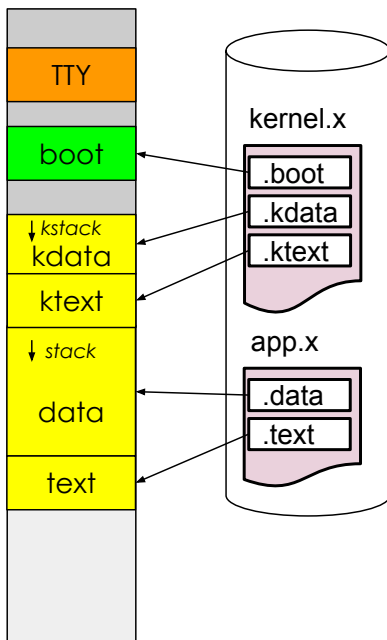
almo1.x prend en paramètre :

- les fichiers binaires MIPS du kernel et de l'application
- les paramètres du matériel
- le nombre de cycles à simuler
- la demande de trace d'exécution

almo1.x remplit les segments de mémoire avec ce qui est présent dans les fichiers binaires puis il envoie un **signal reset** au MIPS qui commence à exécuter la première instruction



Chargement des mémoires



Les mémoires du SoC almo sont remplies avant le démarrage de la simulation par les sections 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)

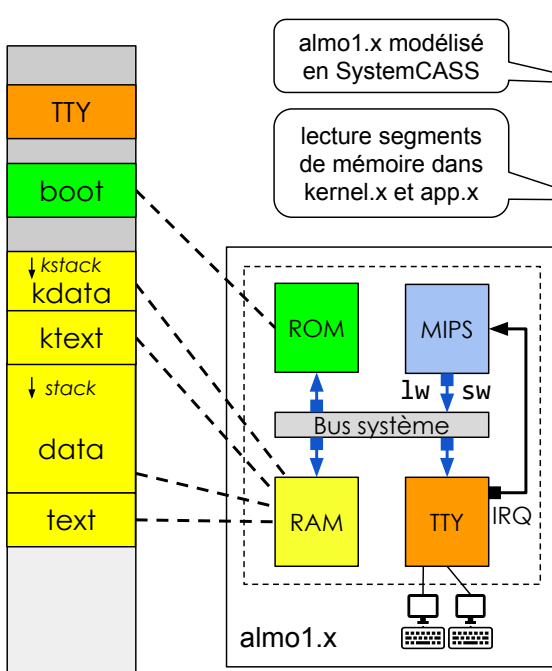
Les deux binaires ont été créés par l'éditeur de lien avec :

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



`almo1.x` modélisé en SystemCASS

lecture segments de mémoire dans `kernel.x` et `app.x`

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

```
SystemCASS
Cycle Accurate System Simulator
[...]
```

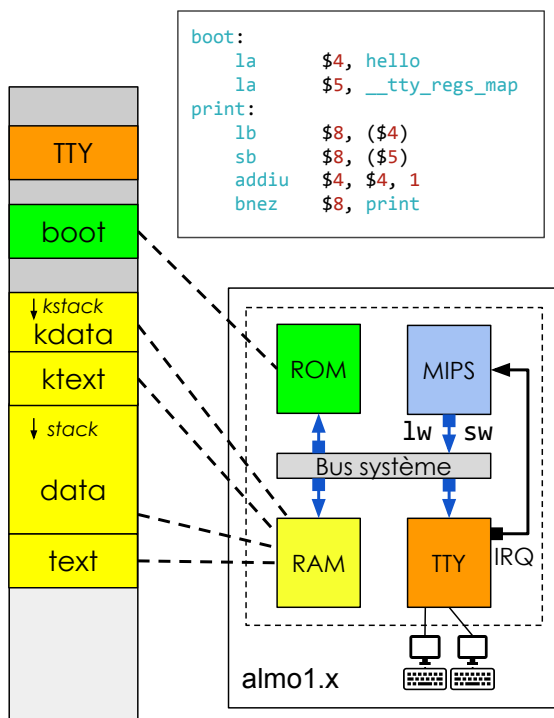
```
Loading at 0xbfc00000 size 0x1000: .boot
Loading at 0x80000000 size 0x20000: .ktext .MIPS.abiflags
Loading at 0x80020000 size 0x3e0000: .kdata
Loading at 0x7f400000 size 0x100000: .text .MIPS.abiflags
Loading at 0x7f500000 size 0xb00000: .data
```

```
### cycle = 38000000 / frequency = 3984.06Khz
```

Après le chargement, le MIPS démarre et il 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

\$> make debug

almo1.x -KERNEL kernel.x -DEBUG 0 -NCYCLES 10000 -NTTY 1 [...]

SystemASS

Cycle Accurate System Simulator

[...]

generate trace[cpu].s, may take a while...

Chaque MIPS a son fichier de trace l'extension .s c'est pour la colorisation de l'éditeur de texte

\$> head trace0.s

mode	cycle	adresse	instruction	code assembleur
K	12:	<boot>:-		
K	12:	0xbfc00000	0x3c04bfc0	lui a0,0xbfc0
K	13:	0xbfc00004	0x24840028	addiu a0,a0,40
K	14:	0xbfc00008	0x3c05d020	lui a1,0xd020
K	15:	0xbfc0000c	0x24a50000	addiu a1,a1,0
K	26:	<print>:-		
K	26:	0xbfc00010	0x80880000	lb t0,0(a0)
K	27:	0xbfc00014	0xa0a80000	sb t0,0(a1)
K	37:	--> READ MEMORY @ 0xbfc00028	BE---1 --> 0x6c6c6548	
K	39:	<-- WRITE MEMORY @ 0xd0200000	BE---1 <-- 0x48	

et 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.

Conclusion

- Ce que nous avons vu...
- 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 lien
- 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 une 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

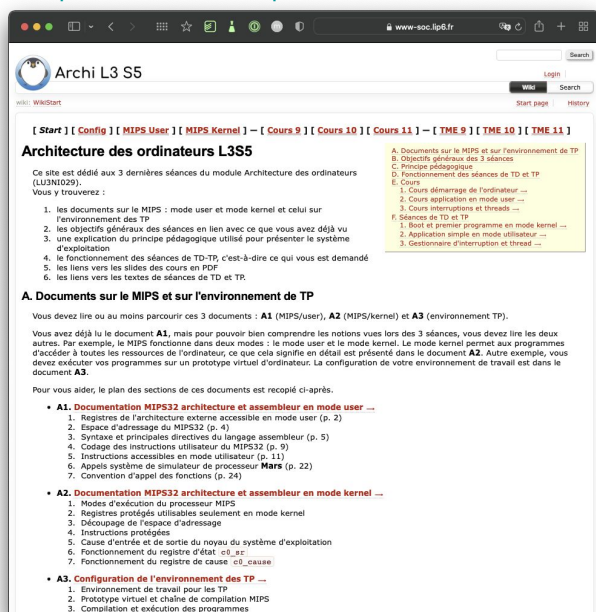
- L'idée est que vous compreniez le mieux possible comment fonctionne un ordinateur du point de vue du logiciel en regardant comment il utilise le matériel (processeur, mémoire et périphériques).
- Pour cela, la méthode choisie est que vous suiviez le démarrage d'une application utilisateur depuis le 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 que 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 travailler à la PPTI, mais vous aurez aussi une archive qui devrait tourner sur toutes les distributions de Linux sur machine réelle ou virtuelle (VirtualBox)

Ressources et préparation des TME

Site dédié pour cette partie de l'UE :

<https://www-soc.lip6.fr/trac/archi-l3s5>

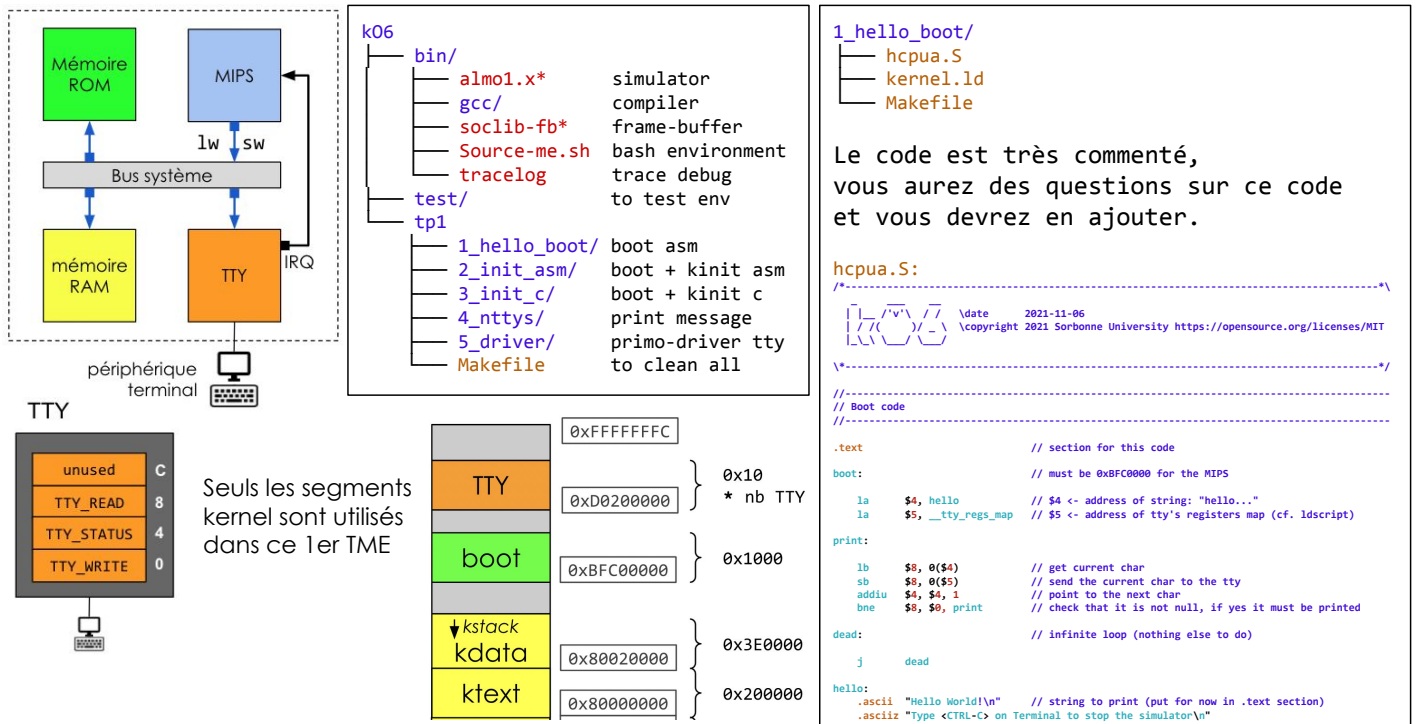


Vous y trouverez :

- 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**
- les **objectifs généraux** des séances en lien avec ce que vous avez déjà vu dans la première partie de l'UE ;
- une explication du **principe pédagogique** utilisé pour présenter l'architecture
- le **fonctionnement des séances** de TD-TP, c'est-à-dire ce que vous ai demandé
- les liens vers les slides des **cours** en PDF
- les liens vers les textes de **TD et TP**.

Vous devez lire ces pages avant les TD...

TD + TME 1



Annexe

- Les arguments utilisés du compilateur
- Les règles génériques du Makefile
- Les 4 types de directives du préprocesseur du C

Compilateur C : arguments utilisés

(<https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc.pdf>)

L'application gcc permet d'appeler le préprocesseur, le compilateur, l'assembleur et l'éditeur de liens. La [documentation de gcc 7.2.0](#) (utilisé en TP) fait plus 900 pages. Nous n'allons voir que quelques arguments... (ceux utilisés en TME)

gcc -c	: stoppe gcc après le compilateur produit un .o ou un .s
-Wall	: tous les warnings du langage C (ou presque) sont relevés
-Werror	: tous les warnings sont considérés comme des erreurs
-mips32r2	: informe le compilateur du type de MIPS32 révision 2
-std=c99	: permet, entre autres, la déclaration des var. locales n'importe où
-fno-common	: ne pas utiliser une section common pour les var. glob. non-static
-fno-builtin	: ne pas utiliser les fonctions "builtin" de gcc (p. ex. strcpy())
-fomit-frame-pointer	: demande d'utiliser \$29 comme seul pointeur de pile
-G0	: ne pas utiliser de global pointer (\$28) pour les variables globales
-O3	: demande l'optimisation maximale (il existe : -O0, -O1, -O2, -Os)
-o <file>	: nom du fichier de sortie <file>
-I <dir1:dir2..>	: ajoute <dir1:dir2..> aux répertoires de recherche des #include
-T <ldscript>	: informe l'éditeur de lien du nom du ldscript (si ld est invoqué)

Makefile : règles génériques et variables

Le format Makefile est très riche (<https://www.gnu.org/software/make/manual/>)

nous n'allons pas tout utiliser (il y a des tutos p. ex sur [codes-sources](#))

- Il permet d'exprimer des règles génériques
 - La dépendance de la cible utilise le caractère % qui représente un nom de fichier quelconque
 - Les commandes utilisent des variables automatiques dont la valeur est extraite de la ligne de dépendance
 - \$@ : cible
 - \$< : première dépendance
 - \$^ : toutes les dépendances
 - \$* : %
- on peut ajouter des règles de commandes
 - clean ou all et le .PHONY permet de dire à make que ces cibles ne sont pas des fichier
- on peut utiliser des variables
 - CFLAGS, BIN, etc.

```
CFLAGS = -Wall
OBJ     = f.o g1.o g2.o
BIN     = exe
.PHONY  = all clean

all      : clean $(EXE)

%.o      : %.c
          gcc -c $(CFLAGS) $<

$(BIN)   : $(OBJ)
          gcc $^ -o $@

clean    :
          rm $(OBJ)

f.o      : f.c f.h
g1.o     : g1.c g.h
g2.o     : g2.c g.h
```

1. Préprocesseur : expansion de macro

(<https://gcc.gnu.org/onlinedocs/cpp/Macros.html>)

```
#define MACRO
#define MACRO DÉFINITION
#define MACRO(a1,a2,...,an) DÉFINITION_AVEC_ARGUMENTS
#undef MACRO
```

- Attention, on ne peut pas mettre de commentaire derrière une définition
- Une définition de macro est sur une seule ligne ou utiliser le caractère \
- Exemples

```
#define DEBUG
#define ROUGE 4
#define MAX(a,b) ((a)>(b)?(a):(b))
#define INCV(v) do{v.x++;v.y++;}while(0)
[...]
```

```
struct v_st {int x, y} v1, v2;
if (MAX(4, 2*i+j) < ROUGE)
    INCV(v1);
else
    INCV(v2);
[...]
```

→ noter les ()
→ si plusieurs instructions dans la définition de la macro

2. Préprocesseur : inclusion de fichiers

(<https://gcc.gnu.org/onlinedocs/cpp/Header-Files.html>)

`#include` permet d'inclure un fichier dans un autre

- `#include <fichier.h>` ou `#include "fichier.h"`
 - inclut *fichier.h* dans le fichier contenant la directive `#include`
 - avec `" "` : *fichier.h* est recherché dans le répertoire courant
 - avec `< >` : *fichier.h* est recherché dans les répertoires standards tel que `/usr/include` et dans les répertoires donnés par l'argument `-I`
- Bon usage :
 - N'inclure que des `.h` ⇒ **jamais** des `.c`
 - Se prémunir contre la double inclusion (évite la redéfinition des macros)

```
#ifndef _FICHIER_H_
#define _FICHIER_H_
...
#endif
```

3. Préprocesseur : compilation conditionnelle

(<https://gcc.gnu.org/onlinedocs/cpp/Conditionals.html>)

Permet de sélectionner le code à inclure par exemple pour adapter le code à la machine ou traiter le debug

Directives : `#if` `#ifdef` `#ifndef` `#else` `#elif` `#endif`

<pre>#ifdef DEBUG code 1 #else code 2 #endif</pre>	<pre>#if (defined DEBUG) code 1 #else code 2 #endif</pre>	<pre>#if VERBOSE > 1 code 1 #elif VERBOSE > 0 code 2 #else code 3 #endif</pre>
<pre>#if 0 code #endif</pre>	<pre>#if !(defined __MIPS__ defined MIPS32) code 1 #endif</pre>	

4. Préprocesseur : autres directives et macros

Contrôle de la compilation : (<https://gcc.gnu.org/onlinedocs/cpp/Diagnostics.html>)

`#error "message"` : affiche "message" et stoppe la compilation
`#warning "message"` : affiche "message"

Macros prédéfinies : (<https://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html>)

<code>__FILE__</code>	:	MACRO	contenant	le	nom	du	fichier	courant
<code>__LINE__</code>	:	"	"		le	numéro	de	ligne
<code>__DATE__</code>	:	"	"		la	date		
<code>__TIME__</code>	:	"	"		l'heure			
<code>__FUNC__</code>	:	"	"		la	fonction	courante	

Mise entre guillemets : (<https://gcc.gnu.org/onlinedocs/cpp/Stringizing.html>)

`#define str(s) #s` : `str(bonjour)` → `"bonjour"`

Concaténation de 2 arguments : (<https://gcc.gnu.org/onlinedocs/cpp/Concatenation.html>)

`#define concat(a,b) a##b` : `concat(A,B)` → `AB`