

INDEX

DOC → [Config] [MIPS U] [MIPS K] [markdown] [CR.md]
 COURS → [1 (+code)] [2 (+outils)] [3] [4] [5] [6] [7] [8] [9]
 TME → [1] [2] [3] [4] [5] [6] [7] [8] [9]
 CODE → [gcc + soc] [1] [2] [3] [4] [5] [6] [7] [8] [9]

- A. Travaux dirigés
 A1. Analyse de l'architecture
 A2. Programmation assembleur
 A3. Programmation en C
 A4. Compilation
 A5. Les modes d'exécution du MIPS
 A6. Langage C pour la programmation système
 A7. Passage entre les modes kernel et user
 A8. Génération du code exécutable
 B. Travaux pratiques
 B1. Saut dans la fonction kinit() du noyau en langage C
 B2. Premier petit pilote pour le terminal
 B3. Ajout d'une bibliothèque de fonctions standards pour le kernel (klibc)
 B4. Ajout de la librairie C pour l'utilisateur

1 - Du boot au premier programme user

Ce TME est sans doute le plus chargé de tous les TME du module parce qu'il porte sur le tout démarrage du système et il va jusqu'à l'exécution d'une application utilisant des services du noyau grâce aux syscalls. Le code reste néanmoins petit parce que le nombre de services est faible. Il y a également des questions sur la chaîne de compilation, mais là encore la complexité est raisonnable.

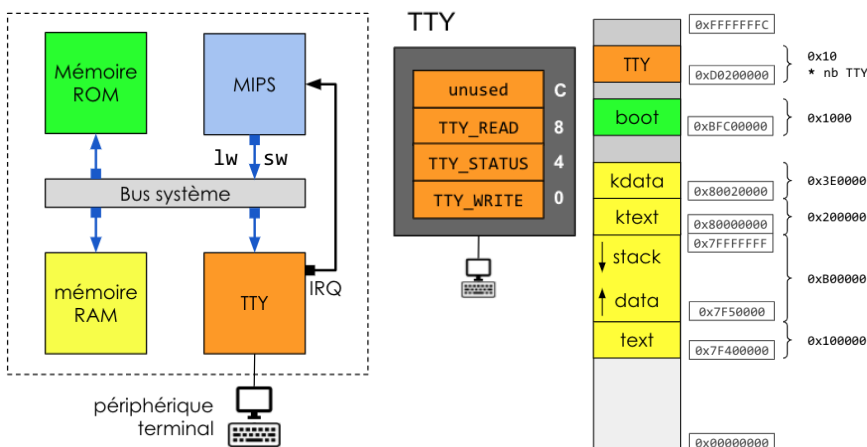
Si vous avez des difficultés, tant sur le code que sur les outils, il est important de relire le cours, de poser des questions, de faire des recherches sur internet, etc. puisque vous aurez besoin d'avoir compris ces bases pour aller plus loin. Soyez proactif, vous seul savez ce que vous ne comprenez pas. 😊

A. Travaux dirigés

A1. Analyse de l'architecture

Les trois figures ci-dessous donnent des informations sur l'architecture du prototype **almo1** sur lequel vous allez travailler.

- À gauche, vous avez un schéma de connexion simplifié.
- Au centre, vous avez la représentation des 4 registres internes du contrôleur de terminal **TTY** nécessaires pour commander un couple écran-clavier.
- À droite, vous avez la représentation de l'espace d'adressage implémenté pour le prototype.



Questions

- Il y a deux mémoires dans **almo1** : RAM et ROM. Qu'est-ce qui les distinguent et que contiennent-elles ?
 - La ROM est une mémoire morte, c'est-à-dire en lecture seule. Elle contient le code de démarrage du prototype.
 - La RAM est une mémoire vive, c'est-à-dire pouvant être lue et écrite. Elle contient le code et les données.
- Qu'est-ce l'espace d'adressage du MIPS ? Quelle taille fait-il ? Quelles sont les instructions du MIPS permettant d'utiliser ces adresses ? Est-ce synonyme de mémoire ?
 - L'espace d'adressage du MIPS est l'ensemble des adresses que peut former le MIPS.
 - Les adresses sont sur 32 bits et désignent chacune un octet, il y a donc 2^{32} octets.
 - On accède à l'espace d'adressage avec les instructions load/store (**lw**, **lh**, **lb**, **lhu**, **lbu**, **sw**, **sh**, **sb**).
 - Non, les mémoires sont des composants contenant des cases de mémoire adressable. Les mémoires sont placées (on dit aussi « *mappées* » dans l'espace d'adressage).
- Qu'est-ce l'espace d'adressage de l'application ?
 - L'espace d'adressage de l'application est l'ensemble des adresses que peut utiliser l'application.
 - Ici, c'est la même chose de l'espace d'adressage du MIPS, même si certains segments ne sont utilisables que lorsque le MIPS est en mode kernel.
 - Dans un SoC plus évolué, l'application pourrait ne pas avoir accès à tout.
- Dans quel composant matériel se trouve le code de démarrage et à quelle adresse est-il placé dans l'espace d'adressage et pourquoi à cette adresse ?
 - Le code de boot est dans la mémoire ROM.
 - Il commence à l'adresse **0xBFC00000** parce que c'est l'adresse qu'envoie le MIPS au démarrage.
- Quel composant permet de faire des entrées-sorties dans almo1 ? Citez d'autres composants qui pourraient être présents dans un autre SoC ?
 - Ici, c'est le composant **TTY** qui permet de sortir des caractères sur un écran et de lire des caractères depuis un clavier.

- On peut avoir aussi un contrôleur de disque, un contrôleur vidéo, un port réseau Ethernet, un port USB, des entrées analogiques (pour mesurer des tensions), etc.
- 6. Il y a 4 registres dans le contrôleur de `TTY`, à quelles adresses sont-ils placés dans l'espace d'adressage ? Comme ce sont des registres, est-ce que le MIPS peut les utiliser comme opérandes pour ses instructions (comme `add`, `or`, etc.) ? Dans quel registre faut-il écrire pour envoyer un caractère sur l'écran du terminal (implicitement à la position du curseur) ? Que contiennent les registres `TTY_STATUS` et `TTY_READ` ? Quelle est l'adresse de `TTY_WRITE` dans l'espace d'adressage ?
 - Le composant `TTY` est placé à partir de l'adresse `0xD0200000`.
 - Non, ce sont des registres de périphériques placés dans l'espace d'adressage et donc accessibles par des instructions `load/store` uniquement.
 - Pour écrire un caractère sur l'écran, il faut écrire le code ASCII du caractère dans le registre `TTY_WRITE`.
 - `TTY_STATUS` contient 1 s'il y a au moins un caractère en attente d'être lu, `TTY_READ` contient le code ASCII du caractère tapé au clavier si `TTY_STATUS==1`.
- 7. Le contrôleur de `TTY` peut contrôler de 1 à 4 terminaux. Chaque terminal dispose d'un ensemble de 4 registres (on appelle ça une carte de registres, ou en anglais une `register map`). Ces ensembles de 4 registres sont placés à des adresses contiguës. S'il y a 2 terminaux (`TTY0` et `TTY1`), à quelle adresse est le registre `TTY_READ` de `TTY1` ?
 - Si les adresses utilisées par `TTY0` commencent à `0xd0200000` alors celles de `TTY1` commencent à l'adresse `0xd0200010` et donc `TTY_READ` est à l'adresse `0xd0200018`.
- 8. Que représentent les flèches bleues sur le schéma ? Pourquoi ne vont-elles que dans une seule direction ?
 - Ces flèches représentent les requêtes d'accès à la mémoire, c'est-à-dire les `loads` et les `stores` qui sont émis par le MIPS lors de l'exécution des instructions `lw`, `sw`, etc. Les requêtes sont émises par le MIPS et reçues par les composants mémoires ou périphériques.
 - On ne représente pas les données qui circulent, mais juste les requêtes, pour ne pas alourdir inutilement le schéma. Implicitement, si le MIPS envoie une requête de lecture alors il y aura une donnée qui va revenir, c'est obligatoire, alors on ne la dessine pas, car ce n'est pas intéressant. En revanche, le fait que le MIPS soit le seul composant à émettre des requêtes est une information intéressante.
- 9. Que fait le contrôleur DMA et donner des différences par rapport au contrôleur de `TTY` ?
 - Le contrôleur DMA fait des déplacements de mémoire comme `memcpy()`, mais il le fait plus vite que la fonction `memcpy()`.
 - La différence la plus importante, c'est qu'il peut faire des lectures et des écritures dans la mémoire. On dit que c'est un initiateur.

A2. Programmation assembleur

L'usage du code assembleur est réduit au minimum. Il est utilisé uniquement où c'est indispensable. C'est le cas du code de démarrage. Ce code ne peut pas être écrit en C pour au moins une raison importante. Le compilateur C suppose la présence d'une pile et d'un registre du processeur contenant le pointeur de pile, or au démarrage les registres sont vides (leur contenu n'est pas significatif). Dans cette partie, nous allons nous intéresser à quelques éléments de l'assembleur qui vous permettront de comprendre le code en TP.

Questions

- Nous savons que l'adresse du premier registre du `TTY` est `0xd0200000` est qu'à cette adresse se trouve le registre `TTY_WRITE` du `TTY0`. Le code permettant d'écrire le code ASCII `'x'` sur le terminal 0 pourrait-être

```
lui    $4, 0xD020
ori    $4, $4, 0x0000 // cette instruction ne sert a rien puisque on ajoute 0, mais je la mets pour le
cas general
ori    $5, $0, 'x'
sb     $5, 0($4)      // Notez que l'immédiat 0 devant ($4) n est pas obligatoire mais on s'obligera à
le mettre
```

Le problème avec le code précédent est que l'adresse du `TTY` est un choix de l'architecte du prototype et s'il décide de placer le `TTY` ailleurs dans l'espace d'adressage, il faudra réécrire le code. Il est préférable d'utiliser une étiquette pour désigner cette adresse : on suppose désormais que l'adresse du premier registre du `TTY` se nomme `__tty_regs_map`. Le code assembleur ne connaît pas l'adresse, mais il ne connaît que le symbole. Ainsi, pour écrire `'x'` sur le terminal 0, nous devons utiliser la macro instruction `la $r, label`. Cette macro-instruction est remplacée lors de l'assemblage du code par une suite composée de deux instructions `lui` et `ori`. Il existe aussi la macro instruction `li` qui demande de charger une valeur sur 32 bits dans un registre. Pour être plus précis, les macro-instructions

```
la $r, label
li $r, 0x87654321
```

sont remplacées par

```
lui $r, label>>16
ori $r, $r, label & 0xFFFF
lui $r, 0x8765
ori $r, $r, 0x4321
```

Réécrivez le code de la question précédente en utilisant `la` et `li`

```
la    $4, __tty_regs_map
li    $5, 'x'
sb    $5, 0($4)
```

- En assembleur pour sauter à une adresse de manière inconditionnelle, on utilise les instructions `j label` et `jr $r`. Ces instructions permettent-elles d'effectuer un saut à n'importe quelle adresse ?
 - `j label` malgré sa forme assembleur effectue un saut relativement au `PC` puisque le `label` n'est pas entièrement encodé dans l'instruction binaire (cf. cours sur les sauts). Cette instruction réalise : `PC ← (PC & 0xF0000000) | (ZeroExtend(label, 32) << 2)`. Les 4 bits de poids forts du `PC` sont conservés, le saut est bien relatif au `PC` (`ZeroExtend?` désigne ici le fait d'étendre le label sur 32 bits en ajoutant des zéros en tête, à changer s'il y a une meilleure syntaxe).

- A l'inverse, `jr $r` effectue un saut absolu puisque cette instruction réalise `PC ← $r`. Autrement dit, si l'on veut aller exécuter du code n'importe où en mémoire, il faut utiliser `jr`.
 - 3. Vous avez utilisé les directives `.text` et `.data` pour définir les sections où placer les instructions et les variables globales, mais il existe la possibilité de demander la création d'une nouvelle section dans le code objet produit par le compilateur avec la directive `.section name, "flags"`
 - `name` est le nom de la nouvelle section. On met souvent un `.name` pour montrer que c'est une section et
 - `"flags"` informe du contenu : `"ax"` pour des instructions, `"ad"` pour des données (ceux que ça intéresse pourront regarder là <https://frama.link/20UzK0FP>)
- Écrivez le code assembleur créant la section `".mytext"` et suivi de l'addition des registres `$5` et `$6` dans `$4`

```
.section .mytext, "ax"
addu $4, $5, $6
```

- 4. À quoi sert la directive `.globl label` ?
 - `globl` signifie `glob al label`. Cette directive permet de dire que le `label` est visible en dehors de son fichier de définition. Ainsi il est utilisable dans d'autres programmes assembleur ou d'autres programmes C.
- 5. Écrivez une séquence de code qui affiche la chaîne de caractère `"Hello"` sur `TTY0`. Ce n'est pas une fonction et vous pouvez utiliser tous les registres que vous voulez. Vous supposez que `__tty_regs_maps` est déjà défini.

```
.data
hello: .asciiz "Hello"
.text
la      $4, hello           // $4 ← address of string
la      $5, __tty_regs_map  // $5 ← address of tty's registers map

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 ok it must be printed
```

- 6. En regardant le dessin de l'espace d'adressage du prototype **almo1**, dites à quelle adresse devra être initialisé le pointeur de pile pour le kernel. Rappelez pourquoi c'est indispensable de le définir avant d'appeler une fonction C et écrivez le code qui fait l'initialisation, en supposant que l'adresse du pointeur de pile vaut celle que représente le nom `__kdata_end`
 - La pile va être initialisée juste à la première adresse au-delà de la zone kdata : `0x80020000 + 0x003E0000 = 0x80400000`
 - La première chose que fait une fonction, c'est décrémenter le pointeur de pile pour écrire `$31`, etc. Il faut donc que le pointeur ait été défini.

```
la $29, __kdata_end
```

A3. Programmation en C

Vous savez déjà programmer en C, mais vous allez voir des syntaxes ou des cas d'usage que vous ne connaissez peut-être pas encore. Les questions qui sont posées ici n'ont pas toutes été vues en cours, mais vous connaissez peut-être les réponses, sinon ce sera l'occasion d'apprendre.

Questions

1. Quels sont les usages du mot clé `static` en C ?
 1. Déclarer `static` une variable globale ou une fonction en faisant précéder leur définition du mot clé `static` permet de limiter la visibilité de cette variable ou de cette fonction au seul fichier de déclaration. Notez que par défaut les variables et les fonctions du C ne sont pas `static`, il faut le demander explicitement. C'est exactement l'inverse en assembleur où tout label est implicitement `static` ; il faut demander avec la directive `.globl` de le rendre visible.
 2. Déclarer `static` une variable locale permet de la rendre persistante, c'est-à-dire qu'elle conserve sa valeur entre deux appels. Cette variable locale n'est pas dans le contexte de la fonction (celui-ci est dans la pile et il est libéré en sortie de fonction). Une variable locale `static` est en fait allouée comme une variable globale mais son usage est limité à la seule fonction où elle est définie.
2. Pourquoi déclarer des fonctions ou des variables `extern` ?
 - Les déclarations `extern` permettent d'informer que le compilateur qu'une variable ou qu'une fonction existe et est définie ailleurs. Le compilateur connaît ainsi le type de la variable ou du prototype des fonctions, il sait donc comment les utiliser. En C, par défaut, les variables et les fonctions doivent être déclarées / leur existence et type doit être connus avant leur utilisation.
 - Il n'y a pas de déclaration `extern` en assembleur parce que ce n'est pas un langage typé. Pour l'assembleur, un label c'est juste une adresse donc un nombre.
3. Comment déclarer un tableau de structures en variable globale ? La structure est nommée `test_s`, elle a deux champs `int` nommés `a` et `b`. Le tableau est nommé `tab` et a 2 cases.

```
struct test_s {
    int a;
    int b;
};
struct test_s tab[2];
```

4. Quelle est la différence entre `#include "file.h"` et `#include <file.h>` ?
 - Avec `#include "file.h"`, le préprocesseur recherche le fichier dans le répertoire local.
 - Avec `#include <file.h>`, le préprocesseur recherche le fichier dans les répertoires standards tel que `/usr/include` et dans les répertoires spécifiés par l'option `-I` du préprocesseur. Il peut y avoir plusieurs fois `-I` dans la commande, par exemple `-I dir1 -I dir2 -I dir3`.
5. Comment définir une macro-instruction C uniquement si elle n'est pas déjà définie ? Écrivez un exemple.

- En utilisant, une directive `#ifndef`, par exemple :

```
#ifndef MACRO
#define MACRO
#endif
```

6. Comment être certain de ne pas inclure plusieurs fois le même fichier `.h` ?

- En utilisant ce que nous venons de voir dans la question précédente : on peut définir une macro instruction différente au début de chaque fichier `.h` (en utilisant le nom du fichier comme nom de macro pour éviter les collisions de nom). On peut alors tester l'existence de cette macro comme condition d'inclusion du fichier.

```
// Debut du fichier filename.h
#ifndef _FILENAME_H_
#define _FILENAME_H_

[... contenu du fichier ...]

#endif
// Fin de fichier filename.h
```

7. Supposons que la structure `tty_s` et le tableau de registres de `TTY` soient définis comme suit. Écrivez une fonction C `int getchar(void)` bloquante qui attend un caractère tapé au clavier sur le `TTY0`. Nous vous rappelons qu'il faut attendre que le registre `TTY_STATUS` soit différent de 0 avant de lire `TTY_READ`.

```
struct tty_s {
    int write;           // tty's output address
    int status;          // tty's status address something to read if not null)
    int read;            // tty's input address
    int unused;          // unused address
};
extern volatile struct tty_s __tty_regs_map[NTTYS];
```

```
int getchar(void)
{
    while (__tty_regs_map[0].status == 0);
    return __tty_regs_map[0].read;
}
```

8. Savez-vous à quoi sert le mot clé `volatile` ? Nous n'en avons pas parlé en cours, mais c'est nécessaire pour les adresses des registres de périphérique, une idée ... ?

- `volatile` permet de dire à `gcc` que la variable en mémoire peut changer à tout moment, elle est volatile. Ainsi quand le programme demande de lire une variable `volatile` le compilateur doit toujours aller la lire en mémoire. Il ne doit jamais chercher à optimiser en utilisant un registre afin de réduire le nombre de lecture mémoire (load). De même, quand le programme écrit dans une variable `volatile`, cela doit toujours provoquer une écriture dans la mémoire (store).
- Ainsi, les registres de périphériques doivent toujours être impérativement lus ou écrits à chaque fois que le programme le demande, parce que c'est justement ces lectures et ces écritures qui commandent le périphérique.

A4. Compilation

Pour obtenir le programme exécutable, nous allons utiliser :

- `gcc -o file.o -c file.c`
 - Appel du compilateur avec l'option `-c` qui demande à `gcc` de faire le préprocessing puis la compilation c pour produire le fichier objet `file.o`
- `ld -o bin.x -Tkernel.ld files.o ...`
 - Appel de l'éditeur de lien pour produire l'exécutable `bin.x` en assemblant tous les fichiers objets `.o`, en les plaçant dans l'espace d'adressage et résolvant les liens entre eux (quand un `.o` utilise une fonction ou une variable définie dans un autre `.o`).
- `objdump -D file.o > file.o.s` ou `objdump -D bin.x > bin.x.s`
 - Appel du désassembleur qui prend les fichiers binaires (`.o` ou `.x`) pour retrouver le code produit par le compilateur à des fins de debug ou de curiosité.

Questions

Le fichier `kernel.ld` décrit l'espace d'adressage et la manière de remplir les sections dans le programme exécutable.

```
_tty_regs_map    = 0xd0200000 ;
_boot_origin     = 0xbfc00000 ;
_boot_length     = 0x00001000 ;
_ktext_origin    = 0x80000000 ;
_ktext_length    = 0x00020000 ;
[... question 1 ...]
_kdata_end       = _kdata_origin + _kdata_length ;

MEMORY {
    boot_region : ORIGIN = _boot_origin, LENGTH = _boot_length
    ktext_region : ORIGIN = _ktext_origin, LENGTH = _ktext_length
[... question 2 ...]
}

SECTIONS {
    .boot : {
        *(.boot)
    } > boot_region
[... question 3 ...]
    .kdata : {
        *(.data*)
    }
```

```
} > kdata_region
}
```

1. Le fichier commence par la déclaration des variables donnant des informations sur les adresses et les tailles des régions de mémoire. Ces symboles n'ont pas de type et ils sont visibles de tous les programmes C, il faut juste leur donner un type pour que le compilateur puisse les exploiter, c'est ce que nous avons fait pour `extern volatile struct tty_s __tty_regs_map[NTTYS]`. En regardant dans le dessin de la représentation de l'espace d'adressage, complétez les lignes de déclaration des variables pour la région `kdata_region`

```
__kdata_origin   = 0x80020000 ;
__kdata_length   = 0x003E0000 ;
```

2. Le fichier contient ensuite la déclaration des régions (dans `MEMORY(...)`) qui vont être remplies par les sections trouvées dans les fichiers objets. Comment modifier cette partie (la zone `[... question 2 ...]`) pour ajouter les lignes correspondant à la déclaration de la région `kdata_region` ?

```
kdata_region : ORIGIN = __kdata_origin, LENGTH = __kdata_length
```

3. Enfin le fichier contient comment sont remplies les régions avec les sections. Complétez les lignes correspondant à la description du remplissage de la région `ktext_region`. Vous devez la remplir avec les sections `.text` issus de tous les fichiers.

```
.ktext : {
    *(.text)
} > ktext_region
```

Nous allons systématiquement utiliser des Makefiles pour la compilation du code, mais aussi pour lancer le simulateur du prototype **almo1**. Pour cette première séance, les Makefiles ne permettent pas de faire des recompilations partielles de fichiers. Les Makefiles sont utilisés pour agréger toutes les actions que nous voulons faire sur les fichiers, c'est-à-dire : compiler, exécuter avec ou sans trace, nettoyer le répertoire. Nous avons recopié partiellement le premier Makefile pour montrer sa forme et poser quelques questions, auxquels vous savez certainement répondre.

```
# Tools and parameters definitions
# -----
NTTY ?= 2 # default number of ttys

CC      = mipsel-unknown-elf-gcc # compiler
LD       = mipsel-unknown-elf-ld # linker
OD       = mipsel-unknown-elf-objdump # desassembler
SX       = almo1.x # prototype simulator

CFLAGS   = -c # stop after compilation, then produce .o
CFLAGS += -Wall -Werror # gives almost all C warnings and considers them to be errors
CFLAGS += -mips32r2 # define of MIPS version
CFLAGS += -std=c99 # define of syntax version of C
CFLAGS += -fno-common # do not use common sections for non-static vars (only bss)
CFLAGS += -fno-builtin # do not use builtin functions of gcc (such as strlen)
CFLAGS += -fomit-frame-pointer # only use of stack pointer ($29)
CFLAGS += -G0 # do not use global data pointer ($28)
CFLAGS += -O3 # full optimisation mode of compiler
CFLAGS += -I. # directories where include files like <file.h> are located
CFLAGS += -DNTTYS=$(NTTY) # define NTTYS with the number of ttys in the prototype

# Rules (here they are used such as simple shell scripts)
# -----
help:
@echo "\nUsage : make <compil|exec|clean> [NTTY=num]\n"
@echo "      compil  : compiles all sources"
@echo "      exec    : executes the prototype"
@echo "      clean   : clean all compiled files\n"

compil:
$(CC) -o hcpu.o $(CFLAGS) hcpu.S
@$(OD) -D hcpu.o > hcpu.o.s
$(LD) -o kernel.x -T kernel.ld hcpu.o
@$(OD) -D kernel.x > kernel.x.s

exec: compil
$(SX) -KERNEL kernel.x -NTTYS $(NTTY)

clean:
-rm *.o* *.x* ~~ *.log.* proc?_term? 2> /dev/null || true
```

4. Au début du fichier se trouve la déclaration des variables du Makefile, quelle est la différence entre `=`, `?` et `+=` ?

- `=` fait une affectation simple
- `?` fait une affectation de la variable si elle n'est pas déjà définie comme variable d'environnement du shell ou dans la ligne de commande de make, par exemple avec `FROM`
- `+=` concatène la valeur courante à la valeur actuelle, c'est une concaténation de chaîne de caractères.

5. Où est utilisé `CFLAGS` ? Que fait `-DNTTYS=$(NTTY)` et pourquoi est-ce utile ici ?

- La variable `CFLAGS` est utilisée par `gcc`, il y a ici toutes les options indispensables pour compiler mais il en existe beaucoup, ce qui fait des tonnes de combinaison d'options !
- `-DNTTYS=$(NTTY)` permet de définir et donner une valeur à une macro (ici définition `NTTYS` avec la valeur `$(NTTY)`) comme le fait un `#define` dans un fichier C. Cette commande évite donc d'ouvrir les codes pour les changer.

6. Si on exécute `make` sans cible, que se passe-t-il ?

- C'est la première cible qui est choisie, donc ici c'est équivalent à `make help`. Cela affiche l'usage pour connaître les cibles disponibles.

7. à quoi servent `@` et `-` au début de certaines commandes ?

- `@` permet de ne pas afficher la commande avant son exécution. On peut rendre ce comportement systématique en ajoutant la règle `.SILENT:` n'importe où dans le fichier.
- `-` permet de ne pas stopper l'exécution des commandes même si elles rendent une erreur, c'est-à-dire une valeur de sortie différente de 0.

A5. Les modes d'exécution du MIPS

Dans cette section, nous allons nous intéresser à ce que propose le processeur MIPS concernant les modes d'exécution. Ce sont des questions portant sur l'usage des modes en général et le comportement du MIPS vis-à-vis de ces modes en particulier. Dans la section **A7**, nous verrons le code de gestion des changements de mode dans le noyau.

Questions

- Le MIPS propose deux modes d'exécution, rappelez quels sont ces deux modes et à quoi ils servent? (Nous l'avons dit dans le descriptif de la séance).
 - Il y a le mode kernel et le mode user.
 - Le mode kernel est utilisé par le noyau alors que le mode user est utilisé par l'application.
- Commencez par rappeler ce qu'est l'espace d'adressage du MIPS et dites ce que signifie « une adresse X est mappée en mémoire ». Dites si une adresse X mappée en mémoire est toujours accessible (en lecture ou en écriture) quelque soit le mode d'exécution du MIPS.
 - L'espace d'adressage du MIPS, c'est l'ensemble des adresses que peut produire le MIPS
 - On dit qu'une adresse est mappée en mémoire, s'il y a bien une case mémoire pour cette adresse.
 - Non X n'est pas toujours accessible, si $X < 0x80000000$ elle est bien accessible quelque soit le mode d'exécution du MIPS, mais si $X \geq 0x80000000$ alors X n'est accessible que si le MIPS est en mode kernel.
- Le MIPS propose des registres à usage général (GPR *General Purpose Register*) pour les calculs (\$0 à \$31). Le MIPS propose un deuxième banc de registres à l'usage du système d'exploitation, ce sont les registres système (dit du coprocesseur 0). Comment sont-ils numérotés? Chaque registre porte un nom correspondant à son usage, quels sont ceux que vous connaissez: donner leur nom, leur numéro et leur rôle? Peut-on faire des calculs avec des registres? Quelles sont les instructions qui permettent de les manipuler?
 - Les registres système sont numérotés de \$0 à \$31, comme les registres GPR, ce qui peut induire une certaine confusion
 - Nous avons vu 3, mais il y en a d'autres que nous verrons plus loin.

| | | |
|------------------------|------|---|
| <code>cr_sr</code> | \$12 | contient essentiellement le mode d'exécution du MIPS et le bit d'autorisation des interruptions |
| <code>cr_cause</code> | \$13 | contient la cause d'appel du noyau |
| <code>cr_epc</code> | \$14 | contient l'adresse de l'instruction ayant provoqué l'appel du noyau ou l'adresse de l'instruction suivante |
| <code>cr_bar</code> | \$8 | contient l'adresse mal formée si la cause est une exception due à un accès non aligné (p.ex. <code>lw</code> a une adresse non multiple de 4) |
| <code>cr_count</code> | \$9 | contient le nombre de cycles depuis le démarrage du MIPS |
| <code>cr_procid</code> | \$15 | contient le numéro du processeur (utile pour les architectures multicores) |

 - non, il n'est pas possible de faire des calculs sur ces registres.
 - On peut juste les lire et les écrire en utilisant les instructions `mtc0` et `mfc0`
- Le registre status est composé de plusieurs champs de bits qui ont chacun une fonction spécifique. Décrivez le contenu du registre status et le rôle des bits de l'octet 0 (seulement les bits vus en cours).

| | | | |
|---|-----|------------------|---|
| 0 | IE | Interrupt Enable | 0 → interruptions masquées 1 → interruptions autorisées si ERL et EXL sont tous les deux à 0 |
| 1 | EXL | EXception Level | 1 → MIPS en mode exception à l'entrée dans le kernel le MIPS est en mode kernel, interruptions masquées |
| 2 | ERL | ERror Level | 1 → au démarrage du MIPS et certaines erreurs de la mémoire le MIPS est en mode kernel, interruptions masquées |
| 4 | UM | User Mode | 0 → MIPS en mode kernel 1 → MIPS en mode user si ERL et EXL sont tous les deux à 0 |

- Le registre cause est contient la cause d'appel du kernel.
Dites à quel endroit est stockée cette cause et donnez la signification des codes 0, 4 et 8
 - Le champ `xCODE` qui contient le code de la cause d'entrée dans le noyau est codé sur 4 bits entre les bits 2 et 5.
 - Les valeurs les plus importantes sont 0 (interruption et syscall). Les autres valeurs sont considérées comme des exceptions.

| | | | |
|---|-------------------|--------------|--|
| 0 | 0000 _b | interruption | un contrôleur de périphérique à lever un signal IRQ |
| 4 | 0100 _b | ADEL | lecture non-alignée (p. ex. <code>lw</code> a une adresse impaire) |
| 8 | 1000 _b | syscall | exécution de l'instruction <code>syscall</code> |
- Le registre `C0_EPC` est un registre 32 bits qui contient une adresse. Vous devriez l'avoir décrit dans la question 2. Expliquez pourquoi ce doit être l'adresse de l'instruction qui provoque une exception qui doit être stockée dans `C0_EPC` ?
 - Une exception, c'est une erreur du programme, telle qu'une division par 0, une lecture non alignée ou une instruction illégale. Il est important que le gestionnaire d'exception sache quelle est l'instruction fautive. C'est pour cette raison que EPC contient l'adresse de l'instruction fautive. Le gestionnaire pourra lire l'instruction et éventuellement corriger le problème.
 - A titre indicatif, ce n'est pas la question, mais pour les syscall, c'est aussi l'adresse de l'instruction `syscall` qui est stockée dans `C0_EPC`, or pour le retour de `syscall`, on souhaite aller à l'instruction suivante. Il faut donc incrémenter la valeur de `C0_EPC` de 4 (les instructions font 4 octets) pour connaître l'adresse de retour.

- Nous avons vu trois instructions utilisables **seulement** lorsque le MIPS est en mode kernel, lesquelles? Que font-elles? Est-ce que l'instruction `syscall` peut-être utilisée en mode user?

- Les trois instructions sont

| | | |
|-------------------------------|---|---------------------------------------|
| <code>mtc0 \$GPR, \$C0</code> | <code>M ove T o C oprocessor 0</code> | <code>\$GPR → COPRO_0(\$C0)</code> |
| <code>mfc0 \$GPR, \$C0</code> | <code>M ove F rom C oprocessor 0</code> | <code>\$GPR ← COPRO_0(\$C0)</code> |
| <code>eret</code> | <code>E xpection RET urn</code> | <code>PC ← EPC ; c0_sr.EXL ← 0</code> |

- Bien sûr que `syscall` peut être utilisé en mode user, puisque c'est comme ça qu'on entre dans le kernel pour les demandes de services.

8. Quelle est l'adresse d'entrée dans le noyau?

- C'est `0x80000180`. Il n'y a qu'une adresse pour toutes les causes `syscall`, exception et interruption.
- Il y a aussi l'adresse de la fonction `kinit()` qui est la fonction appelée par le code de boot (lequel est à l'adresse `0xBFC00000`) pour entrer dans le noyau.

9. Que se passe-t-il quand le MIPS entre dans le noyau, après l'exécution de l'instruction `syscall`?

- L'instruction `syscall` induit beaucoup d'opérations élémentaires dans le MIPS:
 - `EPC ← PC` (adresse de l'instruction `syscall`)
 - `c0_sr.EXL ← 1` (ainsi les bits `c0_sr.UM` et `c0_sr.IE` ne sont plus utilisés)
 - `c0_cause.XCODE ← 8`
 - `PC ← 0x80000180`

10. Quelle instruction utilise-t-on pour sortir du noyau et entrer dans l'application ? Dites précisément ce que fait cette instruction dans le MIPS.

- C'est l'instruction `eret` qui permet de sortir du noyau.
 - `PC ← EPC`
 - `c0_sr.EXL ← 0` (ainsi les bits `c0_sr.UM` et `c0_sr.IE` sont à nouveau utilisés)

A6. Langage C pour la programmation système

La programmation en C, vous connaissez, mais quand on programme pour le noyau, c'est un peu différent. Il y a des éléments de syntaxe ou des besoins spécifiques.

Questions

1. En assembleur, vous utilisez les sections prédéfinies `.data` et `.text` pour placer respectivement les data et le code ou alors vous pouvez créer vos propres sections avec la directive `.section` (nous avons utilisé cette possibilité pour la section `.boot`). Il est aussi possible d'imposer ou de créer des sections en langage C avec le mot clé `__attribute__`. Ce mot clé du C permet de demander certains comportements au compilateur. Il y a en a beaucoup (si cela vous intéresse vous pouvez regarder dans la [doc de GCC sur les attributs](#)). En cours, nous avons vu un attribut permettant de désigner ou créer une section dans laquelle est mise la fonction concernée. Quelle était la syntaxe de cet attribut (regardez sur le slide 37).

- `__attribute__((section("crt0")))`
Remarquez la syntaxe un peu curieuse avec les doubles underscore et les doubles parenthèses.

2. En C, vous savez que les variables globales sont toujours initialisées, soit explicitement dans le programme lui-même, soit implicitement à la valeur `0`. Les variables globales initialisées sont placées dans la section `.data` (ou plutôt dans l'une des sections `data` : `.data`, `.sdata`, `.rodata`, etc.) et elles sont présentes dans le fichier objet (`.o`) produit par le compilateur. En revanche, les variables globales non explicitement initialisées ne sont pas présentes dans le fichier objet. Ces dernières sont placées dans un segment de la famille `.bss`. Le fichier `ldscript` permet de mapper l'ensemble des segments en mémoire. Pour pouvoir initialiser à `0` les segments `bss` par programme, il nous faut connaître les adresses de début et de fin où ils sont placés en mémoire.

Le code ci-dessous est le fichier `ldscript` du kernel `kernel.ld` (nous avons retiré les commentaires mais ils sont dans les fichiers). Expliquez ce que font les lignes 11, 12 et 15.

```
1 SECTIONS
2 {
3     .boot : {
4         *(.boot)
5     } > boot_region
6     .ktext : {
7         *(.text*)
8     } > ktext_region
9     .kdata : {
10        *(.*data*)
11        . = ALIGN(4);
12        __bss_origin = .;
13        *(.*bss*)
14        . = ALIGN(4);
15        __bss_end = .;
16    } > kdata_region
17 }
```

- La ligne 11 contient `. = ALIGN(4)`, c'est équivalent à la directive `.align 4` de l'assembleur. Cela permet de déplacer le pointeur de remplissage de la section de sortie courante (c'est-à-dire ici `.kdata`) sur une frontière de 2^4 octets (une adresse multiple de 16). Cette contrainte est liée aux caches que nous ne verrons pas ici.
- La ligne 12 permet de créer la variable de `ldscript` `__bss_origin` et de l'initialiser à l'adresse courante, ce sera donc l'adresse de début de la zone `bss`.
- La ligne 15 permet de créer la variable `__bss_end` qui sera l'adresse de fin de la zone `bss` (en fait c'est la première adresse qui suit juste `bss`).

3. Nous connaissons les adresses des registres de périphériques. Ces adresses sont déclarées dans le fichier `ldscript` `kernel.ld`. Ci-après, nous avons la déclaration de la variable de `ldscript` `__tty_regs_map`. Cette variable est aussi utilisable dans les programmes C, mais pour être utilisable par le compilateur C, il est nécessaire de lui dire quel type de variable c'est, par exemple une adresse d'entier ou une adresse de tableau d'entiers, Ou encore, une adresse de structure.

Dans le fichier `kernel.ld`:

```
__tty_regs_map = 0xd0200000 ; /* tty's registers map, described in devices.h */
```

Dans le fichier `harch.c` :

```
12 struct tty_s {
13     int write;           // tty's output address
14     int status;         // tty's status address something to read if not null)
15     int read;           // tty's input address
16     int unused;         // unused address
17 };
18
19 extern volatile struct tty_s __tty_regs_map[NTTYS];
```

À quoi servent les mots clés `extern` et `volatile` ?

Si `NTTYS` est une macro dont la valeur est `2`, quelle est l'adresse en mémoire `__tty_regs_map[1].read` ?

- `extern` : informe le compilateur que la variable définie existe ailleurs. Grâce à son type, le compilateur sait s'en servir.
- `volatile` : informe le compilateur que la variable peut changer de valeur toute seule et que donc il doit toujours accéder en mémoire à chaque fois que le programme le demande. Il ne peut donc pas optimiser les accès mémoire en utilisant les registres.
- `__tty_regs_map` est un tableau à 2 cases (puisque `NTTYS = 2`). Chaque case est une structure de 4 entiers, donc `0x10` octets.
`read` est le troisième champ, c'est le troisième entier de la structure, donc en `+8` par rapport au début.
En conséquence `__tty_regs_map[1].read` est en `0xd0200018`.

4. Certaines parties du noyau sont en assembleur. Il y a au moins les toutes premières instructions du code de boot (démarrage de l'ordinateur) et l'entrée dans le noyau après l'exécution d'un syscall. Le gestionnaire de syscall est écrit en assembleur et il a besoin d'appeler une fonction écrite en langage C. Ce que fait le gestionnaire de syscall est:

- trouver l'adresse de la fonction C qu'il doit appeler pour exécuter le service demandé;
- placer cette adresse dans un registre, par exemple `$2`;
- exécuter l'instruction `jal` (ici, `jal $2`) pour appeler la fonction. Que doivent contenir les registres `$4` à `$7` et comment doit-être la pile?
 - C'est un appel de fonction, il faut donc respecter la convention d'appel des fonctions
 - Les registres `$4` à `$7` contiennent les arguments de la fonction
 - Le pointeur de pile doit pointer sur la case réservée pour le premier argument et les cases suivantes sont réservées arguments suivants.
 - Ce n'est pas rappelé ici, mais il y a **au plus** 4 arguments (entier ou pointeur) pour tous les syscalls. En conséquence, le pointeur de pile pointe au début d'une zone vide de 4 entiers.

5. Vous avez appris à écrire des programmes assembleur, mais parfois il est plus simple, voire nécessaire, de mélanger le code C et le code assembleur. Dans l'exemple ci-dessous, nous voyons comment la fonction `syscall()` est écrite. Cette fonction utilise l'instruction `syscall`.

Deux exemples d'usage de la fonction `syscall()` pris dans le fichier `04_libc/ulib/libc.c`

```
1 int fprintf (int tty, char *fmt, ...)
2 {
3     int res;
4     char buffer[PRINTF_MAX];
5     va_list ap;
6     va_start (ap, fmt);
7     res = vsnprintf(buffer, sizeof(buffer), fmt, ap);
8     res = syscall (tty, (int)buffer, 0, 0, SYSCALL_TTY_PUTS);
9     va_end(ap);
10    return res;
11 }
12
13 void exit (int status)
14 {
15     syscall( status, 0, 0, 0, SYSCALL_EXIT);           // never returns
16 }
```

Le code de cette fonction est dans le fichier `04_libc/ulib/crt0.c`

```
1 //int syscall (int a0, int a1, int a2, int a3, int syscall_code)
2 __asm__ (
3     ".globl syscall\n"
4     "syscall:\n"
5     "    lw $2,16($29)\n"
6     "    syscall\n"
7     "    jr $31\n"
8 );
```

Combien d'arguments a la fonction `syscall()` ? Comment la fonction `syscall()` reçoit-elle ses arguments ? A quoi sert la ligne 3 de la fonction `syscall()` et que se passe-t-il si on la retire ? Expliquer la ligne 5 de la fonction `syscall()`. Aurait-il été possible de mettre le code de la fonction `syscall()` dans un fichier `.S` ?

- La fonction `syscall()` a 5 arguments
- Elle reçoit ses 4 premiers arguments dans les registres `$4` à `$7` et le 5e (le numéro de service) dans la pile.
- La ligne 3 sert à dire que `syscall` est une étiquette utilisée dans un autre fichier. `.globl` signifie **global label**. Si on la retire, il y aura un problème lors de l'édition de lien. `syscall()` ne sera pas trouvé par l'éditeur de liens.
- Le noyau attend le numéro de service dans `$2`. Or le numéro du service est le 5e argument de la fonction `syscall()`. La ligne 5 permet d'aller le chercher dans la pile.
- oui, ce code de la fonction `syscall()` qui fait appel à l'instruction `syscall` aurait pu être mis dans un fichier en assembleur, mais cela aurait demandé d'avoir un fichier de plus, pour une seule fonction. Dans une version plus évoluée du système, il y aura un d'autres fonctions assembleur, alors on créera un fichier assembleur pour les réunir.

A7. Passage entre les modes kernel et user

Le noyau et l'application sont deux exécutables compilés indépendamment mais pas qui ne sont pas indépendants. Vous savez déjà que l'application appelle les services du noyau avec l'instruction `syscall`, voyons comment cela se passe vraiment depuis le code C. Certaines questions sont proches de celles déjà posées, c'est volontaire.

Questions

1. Comment imposer le placement d'adresse d'une fonction ou d'une variable en mémoire?

- C'est l'éditeur de lien qui est en charge du placement en mémoire du code et des données, et c'est dans le fichier `ldscript` `kernel.ld` ou `user.ld` que le programmeur peut imposer ses choix.
- Pour placer une fonction à une place, la méthode que vous avez vu consiste
 - à créer une section grâce à la directive `.section` en assembleur ou à la directive `__attribute__((section()))` en C
 - puis à positionner la section créée dans la description des `SECTIONS` du `ldscript`.

2. La fonction `kinit()` appelle la fonction `__start()` : `kernel/kinit.c`

```
void kinit (void)
{
    kprintf (banner);

    // put bss sections to zero. bss contains uninitialised global variables
    extern int __bss_origin; // first int of bss section (defined in ldscript kernel.ld)
    extern int __bss_end;   // first int of above bss section (defined in ldscript kernel.ld)
    for (int *a = &__bss_origin; a != &__bss_end; *a++ = 0);

    extern int __start;      // __start is the entry point of the app (defined in kernel.ld)
    app_load (&__start);    // function to start the user app (defined in hcpua.S)
}
```

`kernel/hcpua.S`

```
.globl app_load // ----- void app_load (void * fun) called by kinit()
app_load:      // call when we exit kinit() function to go to user code

mtc0 $4, $14 // put __start address in c0_EPC
li $26, 0x12 // define next status reg. value
mtc0 $26, $12 // UM <- 1, IE <- 0, EXL <- 1
la $29, __data_end // define new user stack pointer
eret // j EPC and EXL <- 0
```

Dans le code précédent, `$26` est un registre de travail pour le kernel. Quels sont les autres registres modifiés? Expliquez pour chacun la valeur affectée.

- Il y a 3 registres affectés, dans l'ordre :
 - Le registre système `$14` nommé `c0_epc`, il reçoit l'adresse `__crt0`, c'est-à-dire l'adresse de la fonction `__start()`.
 - Le registre système `$12` nommé `c0_sr`, il reçoit la valeur `0x12`, donc les bits `UM`, `EXL` et `IE` prennent respectivement les valeurs `1`, `1` et `0`
 - `UM = 1` et `IE = 0`, signifie que l'on est normalement en mode `user` avec les interruptions masquées, mais comme `EXL = 1`, alors on reste en mode `kernel` avec interruptions masquées. L'exécution de l'instruction `eret` mettra `EXL` à `0` pour rendre les bits `UM` et `IE` actifs et passer en mode `user` (ici avec interruptions masquées).
 - Le registre GPR `$29` reçoit l'adresse de la première adresse après la section `.data`. C'est le haut de la pile.

3. Que faire avant l'exécution de la fonction `main()` du point de vue de l'initialisation? Et au retour de la fonction `main()`?

- Comme dans la fonction `kinit()`, il faut explicitement initialiser les variables globales non initialisées dans le programme C.
- Si on sort de la fonction `main()`, l'application s'achève. Cela signifie qu'il faut appeler la fonction `exit()` qui effectue l'appel système `EXIT`. Cette appel est réalisé au cas où l'application n'aurait pas explicitement exécuté `exit()`.

4. Nous avons vu que le noyau est sollicité par des événements, quels sont-ils? Nous rappelons que l'instruction `syscall` initialise le registre `c0_cause`, comment le noyau fait-il pour connaître la cause de son appel?

- Il y en a 3 (si on excepte le signal `reset` qui redémarre tout le système:
 1. Les appels système donc l'exécution de l'instruction `syscall`.
 2. Les exceptions donc les "erreur" de programmation (division par 0, adressage mémoire incorrect, etc.).
 3. Les interruptions qui sont des demandes d'intervention provenant des périphériques.
- L'instruction `syscall` initialise les 4 bits `XCODE` du registre `c0_cause` avec un code indiquant la raison de l'entrée dans le noyau. Le noyau doit analyser ce champ `XCODE`.

5. `$26` et `$27` sont deux registres temporaires que le noyau se réserve pour faire des calculs sans qu'il ait besoin de les sauvegarder dans la pile. Ce ne sont pas des registres système comme `c0_sr` ou `c0_epc`. En effet, l'usage de ces registres (`$26` et `$27`) par l'utilisateur ne provoque pas d'exception du MIPS. Toutefois si le noyau est appelé alors il modifie ces registres et donc l'utilisateur perd leur valeur.

Le code assembleur ci-après contient les instructions exécutées à l'entrée dans le noyau, quelle que soit la cause. Les commentaires présents dans le code ont été volontairement retirés (ils sont dans les fichiers du TP). La section `.kentry` est placée à l'adresse `0x80000000` par l'éditeur de lien. La directive `.org` (ligne 16) permet de déplacer le pointeur de remplissage de la section courante du nombre d'octets donnés en argument, ici `0x180`. Pouvez-vous dire pourquoi? Expliquer les lignes 25 à 28.

`kernel/hcpua.S`

```
15 .section .kentry, "ax"
16 .org 0x180
22
23 kentry:
24
25 mfc0 $26, $13
26 andi $26, $26, 0x3C
27 li $27, 0x20
28 bne $26, $27, kpanic
```

- La section `.kentry` est placée à l'adresse `0x80000000` or l'entrée du noyau est `0x80000180`, il faut donc déplacer le pointeur de remplissage de la section `.kentry` de `0x180`. Remarquez qu'on aurait pu utiliser une directive `.space 0x180`.

- Commentaire du code
 - Ligne 25 : `$26 ← c0_cause`
→ donc le registre `$26` GPR réservé au kernel prend la valeur du registre de cause.
 - Ligne 26 : `$26 ← $26 & 0b00111100`
→ C'est un masque qui permet de ne conserver que les 4 bits du champ `XCODE`.
 - Ligne 27 : `$27 ← 0b00100000`
→ On initialise le registre GPR réservé au kernel `$27` avec la valeur attendue dans `$26` s'il s'agit d'une cause `syscall`.
 - Ligne 28 : si `$26 ≠ $27` goto kpanic
→ Si ce n'est pas un `syscall`, on va plus loin, sinon on continue en séquence.

6. Le gestionnaire de `syscall` est la partie du code qui gère le comportement du noyau lors de l'exécution de l'instruction `syscall`. C'est un code en assembleur présent dans le fichier `kernel/hcpua.S` que nous allons observer. Pour vous aider dans la compréhension de ce code, vous devez imaginer que l'instruction `syscall` est un peu comme un appel de fonction. Ce code utilise un tableau de pointeurs de fonctions nommé `syscall_vector` défini dans le fichier `kernel/ksyscalls.c`. Les lignes 47 à 54 sont chargées d'allouer de la place dans la pile.

- Dessinez l'état de la pile après l'exécution de ces instructions.
- Que fait l'instruction ligne 55 et quelle conséquence cela a-t-il?
- Que font les lignes 57 à 62?
- Et enfin que font les lignes 64 à 70?

Les commentaires ont été laissés, vous devez juste mettre à quoi ça sert, sans détailler ligne à ligne.

common/syscalls.h

```
#define SYSCALL_EXIT      0      /* see exit()   in ulib/libc.c */
#define SYSCALL_READ      1      /* see read()   in ulib/libc.c */
#define SYSCALL_WRITE     2      /* see write()  in ulib/libc.c */
#define SYSCALL_CLOCK     3      /* see clock()  in ulib/libc.c */
#define SYSCALL_NR       32
```

kernel/ksyscalls.c

```
void *syscall_vector[] = {
    [0 ... SYSCALL_NR - 1] = unknown_syscall, /* default function */
    [SYSCALL_EXIT]         = exit,
    [SYSCALL_READ]         = tty_read,
    [SYSCALL_WRITE]        = tty_write,
    [SYSCALL_CLOCK]        = clock,
};
```

kernel/hcpua.S

```
45 syscall_handler:
46
47     addiu    $29,    $29,    -8*4          // context for $31 + EPC + SR + syscall_code + 4 args
48     mfc0     $27,    $14                  // $27 ← EPC (addr of syscall instruction)
49     mfc0     $26,    $12                  // $26 ← SR (status register)
50     addiu    $27,    $27,    4            // $27 ← EPC+4 (return address)
51     sw       $31,    7*4($29)            // save $31 because it will be erased
52     sw       $27,    6*4($29)            // save EPC+4 (return address of syscall)
53     sw       $26,    5*4($29)            // save SR (status register)
54     sw       $2,     4*4($29)            // save syscall code (useful for debug message)
55     mtc0     $0,     $12                  // SR ← kernel-mode without INT (UM=0 ERL=0 EXL=0 IE=0)
56
57     la       $26,    syscall_vector      // $26 ← table of syscall functions
58     andi     $2,     $2,    SYSCALL_NR-1 // apply syscall mask
59     sll      $2,     $2,    2             // compute syscall index (multiply by 4)
60     addu     $2,     $26,    $2           // $2 ← & syscall_vector[$2]
61     lw       $2,     ($2)                // at the end: $2 ← syscall_vector[$2]
62     jalr     $2
63
64     lw       $26,    5*4($29)            // get old SR
65     lw       $27,    6*4($29)            // get return address of syscall
66     lw       $31,    7*4($29)            // restore $31 (return address of syscall function)
67     mtc0     $26,    $12                  // restore SR
68     mtc0     $27,    $14                  // restore EPC
69     addiu    $29,    $29,    8*4          // restore stack pointer
70     eret
```

- État de la pile après l'exécution des lignes 36 à 43

| | | |
|--------|--------|--|
| + | -----+ | |
| | \$31 | Nous allons exécuter jal un peu plus et perdre \$31, il faut le sauver |
| + | -----+ | |
| | C0_EPC | C'est l'adresse de retour du syscall |
| + | -----+ | |
| | C0_SR | le registre status est modifié plus loin, il faut le sauver pour le restaurer |
| + | -----+ | |
| | \$2 | C'est le numéro de syscall qui pourra être accédé par la fonction appelée en 5e argument |
| + | -----+ | |
| | | place réservée pour le 4e argument actuellement dans \$7 |
| + | -----+ | |
| | | place réservée pour le 3e argument actuellement dans \$6 |
| + | -----+ | |
| | | place réservée pour le 2e argument actuellement dans \$5 |
| + | -----+ | |
| \$29 → | | place réservée pour le 1e argument actuellement dans \$4 |
| + | -----+ | |

- L'instruction ligne 44 met `0` dans le registre `c0_sr`. Ce qui a pour conséquence de mettre à `0` les bits `UM`, `EXL` et `IE`. On est donc en mode kernel avec interruptions masquées.

- Notez qu'interdire les interruptions pendant l'exécution des syscall est contraignant. Pour le moment, ce n'est pas important puisque nous ne traitons pas les interruptions, mais si nous les traitons, elles seraient masquées. En conséquence, il serait interdit aux fonctions qui traitent les appels système d'exécuter des attentes longues (comme une boucle qui attend le changement d'état d'un registre de périphérique) car sinon, le noyau serait bloqué (plus rien ne bougerait).
- Commentaire du code lignes 46 à 53
 - Ligne 46 : `$26` ← l'adresse du tableau `syscall_vector`
→ On s'apprête à y faire un accès indexé par le registre `$2`
 - Ligne 47 : `$2` ← `$2 & 0x1F`
→ pour éviter de sortir du tableau si l'utilisateur a mis n'importe quoi dans `$2`
 - Ligne 48 : `$2` ← `$2 * 4`
→ Les cases du tableau sont des pointeurs et font 4 octets
 - Ligne 49 : `$2` ← `$26 + $2`
→ `$2` contient désormais l'adresse de la case contenant la fonction correspondante au service n° `$2`
 - Ligne 50 : `$2` ← `MEM[$2]`
→ `$2` contient l'adresse de la fonction à appeler
 - Ligne 51 : `jal $2`
→ appel de la fonction de service
On rappelle que `$4` à `$7` et qu'il y a de place pour ces arguments dans la pile.
- Les lignes 53 à 59 restaurent l'état des registres `$31`, `c0_status`, `c0_epc` et le pointeur de pile puis on sort du noyau avec l'instruction `eret`.

A8. Génération du code exécutable

Pour simuler le logiciel, il faut produire deux exécutables. Nous utilisons, ici, un Makefile hiérarchique et des règles explicites. Cela sort du cadre de l'architecture, mais vous avez besoin de ce savoir-faire pour comprendre le code, alors allons-y.

Questions

1. Rappelez à quoi sert un Makefile?

- Le rôle principal d'un Makefile est de décrire le mode d'emploi pour construire un fichier dit **cible** à partir d'un ou plusieurs fichiers **source** (dits de dépendance) en utilisant des commandes du **shell**. Ce rôle pourrait tout aussi bien être occupé par un script **shell** et d'ailleurs, dans le premier TP, nous avons vu un usage du Makefile dans lequel nous avons rassemblé plusieurs scripts **shell** sous forme de règles.
- Le second rôle d'un Makefile est de permettre la reconstruction efficace du fichier **cible** lorsqu'un seul fichier **source** change. Pour ce rôle, le Makefile exprime toutes les étapes de constructions de la **cible** finale et des **cibles** intermédiaires sous forme d'un arbre dont les feuilles sont les fichiers **sources**.

2. Vous n'allez pas à avoir à écrire un Makefile complètement. Toutefois, si vous ajoutez des fichiers source, vous allez devoir les modifier en ajoutant des règles. Nous avons vu brièvement la syntaxe utilisée dans les Makefiles de ce TP au cours n°1. Les lignes qui suivent sont des extraits de `03_klibc/Makefile` (le Makefile de l'étape1). Dans cet extrait, quelles sont la **cible** finale, les **cibles** intermédiaires et les **sources**? A quoi servent les variables automatiques de make? Dans ces deux règles, donnez-en la valeur.

```
kernel.x : kernel.ld obj/hcpu.o obj/kinit.o obj/klibc.o obj/harch.o
$(LD) -o $@ -T $^
$(OD) -D $@ > $@.s

obj/hcpua.o : hcpua.S hcpu.h
$(CC) -o $@ $(CFLAGS) $<
$(OD) -D $@ > $@.s
```

- La **cible** finale est : `kernel.x`
- Les **cibles** intermédiaires sont : `kernel.ld`, `obj/hcpu.o`, `obj/kinit.o`, `obj/klibc.o` et `obj/harch.o`.
- La **source** est : `hcpua.S`
- Les variables automatiques servent à extraire des noms dans la définition de la dépendance (**cible : dépendances**)
 - dans la première règle :
 - `$@` = **cible** = `kernel.x`
 - `$^` = l'ensemble des dépendances = `kernel.ld`, `obj/hcpu.o`, `obj/kinit.o`, `obj/klibc.o` et `obj/harch.o`
 - dans la seconde règle :
 - `$@` = **cible** = `obj/hcpu.o`
 - `$<` = la première des dépendances = `hcpua.S`

3. Dans le TP, à partir de la deuxième étape, nous avons trois répertoires de sources `kernel`, `ulib` et `uapp`. Chaque répertoire contient un fichier **Makefile** différent destiné à produire une **cible** différente grâce à une règle nommée **compil**, c.-à-d. si vous tapez `make compil` dans un de ces répertoires, cela compile les sources locales. Il y a aussi un Makefile dans le répertoire racine `04_libc`. Dans ce dernier Makefile, une des règles est destinée à la compilation de l'ensemble des sources dans les trois sous-répertoires. Cette règle appelle récursivement la commande **make** en donnant en argument le nom du sous-répertoire où descendre : `make -C <répertoire> [cible]` est équivalent à `cd <répertoire>; make [cible] ; cd ..`. Ecrivez la règle **compil** du fichier `04_libc/Makefile`.

```
04_libc/
├── Makefile           : Makefile racine qui invoque les Makefiles des sous-répertoires et qui exécute
├── common             : répertoire des fichiers commun kernel / user
├── kernel             : Répertoire des fichiers composant le kernel
│   └── Makefile       : description des actions possibles sur le code kernel : compilation et nettoyage
├── uapp              : Répertoire des fichiers de l'application user seule
│   └── Makefile       : description des actions possibles sur le code user : compilation et nettoyage
├── ulib              : Répertoire des fichiers des bibliothèques système liés avec l'application user
│   └── Makefile       : description des actions possibles sur le code user : compilation et nettoyage
```

```
compil:
make -C kernel compil
```

```
make -C ulib    compil
make -C uapp    compil
```

B. Travaux pratiques

Pour les travaux pratiques, vous devez d'abord répondre aux questions, elles ont pour but de vous faire lire le code et revoir les points du cours. Les réponses sont dans le cours ou dans les fichiers sources. Certaines ont déjà été traitées dans la partie TD, c'est normal. Ensuite, vous passez aux exercices pratiques.

Pour récupérer le code, référez-vous à la section **Récupération du code du TP** de la page principale de ce site.

Les premières étapes du TP sont uniquement dans le noyau et le MIPS est alors en mode kernel puis, à la fin, les applications de l'utilisateur s'exécutent en mode user, au-dessus d'une petite libc, à laquelle vous devez ajouter un service de copie de mémoire (`memcpy`).

B1. Saut dans la fonction kinit() du noyau en langage C

Dans ce premier programme, le code de boot entre dans le noyau par la fonction C `kinit()`, c'est une fonction et donc il faut absolument une pile d'exécution. C'est un tout petit programme, mais pour obtenir l'exécutable, vous devrez utiliser tous les outils de la chaîne de cross-compilation MIPS et pour l'exécuter vous devrez exécuter le simulateur du prototype.

Objectifs

- produire un exécutable à partir d'un code en assembleur et en C
- savoir comment afficher un caractère sur un terminal.
- savoir analyser une trace d'exécution
- Savoir comment et où déclarer la pile d'exécution du noyau.
- Savoir comment afficher un caractère sur un terminal depuis un programme C.

Fichiers

```
01_init_c/
├── hcpua.S      : code dépendant du cpu matériel en assembleur
├── kernel.ld    : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
├── kinit.c      : fichier en C contenant le code de démarrage du noyau, ici c'est la fonction kinit().
└── Makefile    : description des actions possibles sur le code : compilation, exécution, nettoyage, etc.
```

Questions

- Dans quel fichier se trouve la description de l'espace d'adressage du MIPS ? Que trouve-t-on dans ce fichier ?
 - C'est dans le fichier `kernel.ld`.
 - On y trouve:
 - la définition de variables du ldscript. Ce sont essentiellement des adresses dans l'espace d'adressage, mais pas seulement, il y a aussi la taille des régions.
 - On trouve ensuite la déclaration des régions mémoires.
 - et enfin la définition des sections de sortie qui seront mises dans le fichier binaire produit et dans quelle région elles sont placées.
- Dans quel fichier se trouve le code de boot et pourquoi, selon vous, avoir nommé ce fichier ainsi ?
 - Le code de boot est dans le fichier `hcpua.S`. Il a été nommé ainsi parce que c'est du code qui dépend du hardware et qu'il concerne le cpu.
- À quelle adresse démarre le MIPS ? Où peut-on le vérifier ?
 - L'adresse de démarrage est `0xBFC00000`.
 - On peut le vérifier dans le fichier `kernel.ld`. Il y a une définition des régions mémoires, dont une région commençant à cette adresse-là, et c'est dans cette région que l'on met le code de boot.
- Que produit `gcc` quand on utilise l'option `-c` ?
 - L'option `-c` demande à `gcc` de s'arrêter après avoir produit le fichier objet.
 - Il produit donc un fichier au format `.o`.
- Que fait l'éditeur de liens ? Comment est-il invoqué ?
 - L'éditeur de liens rassemble toutes les sections produites par le compilateur, et donc présentes dans les fichiers objet `.o`, et il les place dans de nouvelles sections, elles-mêmes placées dans les régions de la mémoire, conformément au fichier ldscript (ici `kernel.ld`).
 - L'éditeur de liens est appelé par `gcc` si on n'a pas l'option `-c` ou directement par `ld` (ici `mipsel_unknown_ld`).
- De quels fichiers a besoin l'éditeur de liens pour fonctionner ?
 - L'éditeur de liens a besoin des fichiers objets `.o` et du fichier ldscript (ici, `kernel.ld`).
- Dans quelle section se trouve le code de boot ? (la réponse est dans le code assembleur)
 - Le code de boot a été mis dans une section `.boot`.
- Dans quelle région de la mémoire le code de boot est-il placé ? (la réponse est dans `kernel.ld`)
 - Le code de boot est placé dans la région `boot_region`.
- Comment connaît-on l'adresse du registre de sortie du contrôleur de terminal `TTY` ?
 - Le fichier `kernel.ld` déclare une variable `__tty_regs_map` initialisée avec l'adresse de où sont placés les registres de contrôles du `TTY`. Le premier registre à l'adresse `__tty_regs_map` est l'adresse du registre de sortie `TTY_WRITE`.

10. Quand faut-il initialiser la pile ? Dans quel fichier est-ce ? Quelle est la valeur du pointeur initial ?

- Il faut initialiser le pointeur avant d'appeler `kinit()`
- C'est dans le fichier `hcpua.S`
- `$29 ← __kdata_end`, c'est-à-dire `0x80400000`

11. Dans quel fichier le mot clé `volatile` est-il utilisé ? Rappeler son rôle.

- Il est utilisé dans `kinit.c` pour informer le compilateur que la variable `__tty_regs_map` doit toujours être lue en mémoire et ne peut jamais être "optimisée" dans un registre. Les écritures doivent aussi toujours toutes avoir lieu. Cette variable désigne les registres du contrôleur de terminal. Quand le programme accède en lecture ou écriture à cette variable, il veut accéder au terminal, il faut vraiment qu'il y ait des load/store dans le programme assembleur correspondant au programme source.

Exercices

- Exécutez le programme en lançant le simulateur avec `make exec`, qu'observez-vous ?

- On voit une fenêtre `xterm` qui affiche un message et c'est tout. Dans le terminal de lancement de `make exec`, on voit le compteur de cycles avancer.

- Exécutez le programme en lançant le simulateur avec `make debug`.

Cela exécute le programme pour une courte durée et cela produit deux fichiers `trace0.s` et `label0.s`.

- `trace0.s` contient la trace des instructions assembleur exécutées par le processeur.

Ouvrez `trace.0.s` et repérez ce qui est cité ici

- On voit la séquence des instructions exécutées
- La première colonne nous informe que les adresses lues sont dans l'espace Kernel
- La seconde colonne sont les numéros de cycles
- La troisième sont les adresses des instructions
- La quatrième le code binaire des instructions
- Le reste de la ligne contient l'instruction désassemblée
- Lorsque les adresses ont un nom, c'est à dire qu'une étiquette leur a été attribuée, celle-ci est indiquée.

```

K 12: <boot> ----- ./hcpua.S
K 12: 0xbfc00000 0x3c1d8040 lui sp,0x8040
K 13: 0xbfc00004 0x27bd0000 addiu sp,sp,0
K 14: 0xbfc00008 0x3c1a8000 lui k0,0x8000
K 15: 0xbfc0000c 0x275a0028 addiu k0,k0,40
K 26: 0xbfc00010 0x03400008 jr k0
K 27: 0xbfc00014 0x00000000 nop
K 37: <kinit> ----- ./kinit.c
K 37: 0x80000028 0x27bdffe8 addiu sp,sp,-24
K 38: 0x8000002c 0xafbf0014 sw ra,20(sp)
K 39: <-- WRITE MEMORY @ 0x803ffffc BE=1111 <-- 0
K 48: 0x80000030 0x3c048002 lui a0,0x8002
K 49: 0x80000034 0x0c000000 jal 80000000 <puts>
K 50: 0x80000038 0x24840000 addiu a0,a0,0
K 60: <puts> ----- ./kinit.c

```

- `label0.s` contient la séquence des appels de fonctions de l'exécution. C'est en fait un extrait de la trace. Ouvrez le fichier `label0.s` et interprétez ce que vous voyez.

```

K 12: <boot> ----- ./hcpua.S
K 37: <kinit> ----- ./kinit.c
K 60: <puts> ----- ./kinit.c

```

- Si vous ouvrez le Makefile, vous pouvez voir que le mode d'optimisation du compilateur est `O1` (regardez la définition de `CFLAGS`). Si vous demandez une optimisation en `O2` ou `O3`, et que vous exécutez à nouveau votre programme en mode debug, qu'observez-vous dans la trace d'exécution ?

- La fonction `puts` a disparue, elle a été *inlinée* par le compilateur ! Par conséquent, parfois pour voir le debug, il faut demander au compilateur de ne pas optimiser, mais parfois aussi en faisant ça, le bug disparaît et là, on pleure...

- Ouvrez les fichiers `kinit.o.s` et `kernel.x.s`, le premier fichier est le désassemblage de `kinit.o` et le second est le désassemblage de `kernel.x`. Dans ces fichiers, vous avez plusieurs sections. Les sections `.MIPS.abiflags`, `.reginfo` et `.pdr` ne nous sont pas utiles (elles servent au chargeur d'application, elles contiennent des informations sur le contenu du fichier et cela ne nous intéresse pas).

Notez l'adresse de `kinit` dans les deux fichiers, sont-ce les mêmes ? Sont-elles dans les mêmes sections ? Expliquez pourquoi.

- Dans `kinit.o.s`, l'adresse de `kinit` est `0` alors que dans `kernel.x.s` l'adresse est `0x80000000`.
- Dans `kinit.o.s`, `kinit` est dans la section `.text` alors que dans `kernel.x.s` `kinit` est dans la section `.ktext`.
- La raison est que
 - dans `kinit.o`, `kinit` n'a pas encore été placé, le compilateur commence toutes ses sections à 0, donc `kinit` est dans la section `.text` et elle commence à 0.
 - dans `kernel.x.s` `kinit` est placé et mis dans la section `.ktext` comme le fichier `kernel.ld` le demande.

- Modifiez le code de `kinit.c` et affichez un second message ?

- C'est toujours du copier-coller, mais parfois on a des surprises :-)

B2. Premier petit pilote pour le terminal

Le prototype de SoC que nous utilisons pour les TP est configurable. Il est possible par exemple de choisir le nombre de terminaux texte (TTY). Par défaut, il y en a un mais, nous pouvons en avoir jusqu'à 4. Nous allons modifier le code du noyau pour s'adapter à cette variabilité. En outre, nous allons ajouter un niveau d'abstraction qui représente un début de pilote de périphérique (device driver). Ce pilote, même tout petit, constitue une couche logicielle avec une API.

Objectifs

- Savoir comment créer un début de pilote pour le terminal `TTY`.
- Savoir comment écrire une API en C
- Savoir appeler une fonction en assembleur depuis le C

Fichiers

```
02_driver/
├── harch.c      : code dépendant de l'architecture du SoC, pour le moment c'est juste le pilote du TTY
├── harch.h      : API du code dépendant de l'architecture
├── hcpu.h       : prototype de la fonction clock()
├── hcpu.a.S     : code dépendant du cpu matériel en assembleur
├── kernel.ld    : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
├── kinit.c      : fichier en C contenant le code de démarrage du noyau, ici c'est la fonction kinit().
└── Makefile     : description des actions possibles sur le code : compilation, exécution, nettoyage, etc.
```

Questions

1. Le code du driver du TTY est dans le fichier `harch.c` et les prototypes sont dans `harch.h`. Si vous ouvrez `harch.h` vous allez voir que seuls les prototypes des fonctions `tty_read()` et `tty_write()` sont présents. La structure décrivant la carte des registres du `TTY` est déclarée dans le `.c`. Pourquoi avoir fait ainsi ?
 - Le noyau n'a pas besoin de savoir comment sont organisés les registres dans le TTY. Il a juste besoin de savoir comment écrire ou lire un message. Plus c'est cloisonné, moins il y a de risque de problèmes. En outre, cela simplifie un hypothétique portage sur une autre architecture.
2. Le MIPS dispose d'un compteur de cycles internes. Ce compteur est dans un banc de registres accessibles uniquement quand le processeur fonctionne en mode `kernel`. Nous verrons ça au prochain cours, mais en attendant nous allons quand même exploiter ce compteur. Pourquoi avoir mis la fonction dans `hcpu.a.S` ? Rappel, pourquoi avoir mis `.globl clock`
 - La fonction qui lit ce registre (`$9` qui ne désigne pas un registre GPR du processeur !) est nécessairement en assembleur car elle utilise des instructions particulières et dépend du matériel, elle est donc mise dans `hcpu.a.S`.
 - `.globl clock` permet de faire en sorte que la fonction soit visible par les autres fichiers C.
3. Compilez et exécutez le code avec `make exec`. Observez. Ensuite ouvrez le fichier `kernel.x.s` et regardez où a été placée la fonction `clock()`. Est-ce un problème si `kinit()` n'est plus au début du segment `ktext` ? Pour répondre, posez-vous la question de qui a besoin de connaître l'adresse de `kinit()`
 - Non, ce n'est pas un problème puisque ça fonctionne. Le code de boot a besoin de l'adresse de `kinit()` mais on l'obtient avec la macro `la`. C'est l'éditeur de lien qui fera en sorte que dans le code binaire l'adresse de `kinit()` mise dans le registre `$26` soit la bonne. Notez que cela fonctionne parce qu'on fait l'édition de lien du noyau et du code de boot pour fabriquer un seul binaire (`kernel.x`), ce qui n'est pas le cas dans un vrai système, savez-vous pourquoi ?

Exercices

- Écrire une fonction `void Capitalize(void)` appelée par la fonction `kinit()` qui lit une phrase terminée par un `\n` et la réécrit en ayant mis en majuscule la première lettre de chaque mot. Vous mettrez cette nouvelle fonction dans le fichier `kinit.c` (ce ne devrait pas être sa place mais c'est juste un exercice). Notez que vous ne pouvez pas utiliser la fonction `toupper()` parce que c'est une fonction de la `glibc` (la bibliothèque de la librairie de fonctions standards) et que là vous ne l'avez pas. Vous n'êtes pas sur Linux :-)
- Dans cette fonction `void Capitalize(void)`, pour chaque caractère lu au clavier, vous devez tester s'il est compris entre 'a' et 'z' et si oui, ajouter ('A' - 'a') ...

B3. Ajout d'une bibliothèque de fonctions standards pour le kernel (klibc)

Objectifs de l'étape

Le noyau gère les ressources matérielles et logicielles utilisées par les applications. Il a besoin de fonctions standards pour réaliser des opérations de base, telles qu'une fonction `print` ou une fonction `rand`. Ces fonctions ne sont pas très originales, mais elles recèlent des subtilités que vous ne connaissez peut-être pas encore, vous pouvez les regarder par curiosité. En outre, nous allons utiliser un Makefile définissant un graphe de dépendance explicite entre les fichiers cibles et les fichiers sources avec des règles de construction.

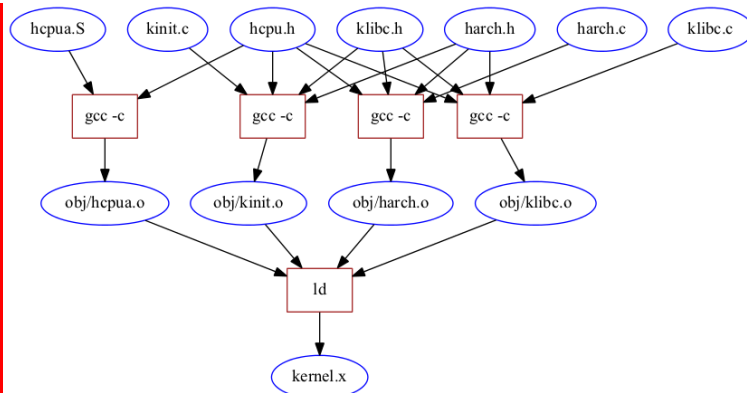
Fichiers

```
03_klibc/
├── kinit.c      : fichier contenant la fonction de démarrage du noyau
├── harch.h      : API du code dépendant de l'architecture
├── harch.c      : code dépendant de l'architecture du SoC
├── hcpu.h       : prototype de la fonction clock()
├── hcpu.a.S     : code dépendant du cpu matériel en assembleur
├── kernel.ld    : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
├── klibc.h      : API de la klibc
├── klibc.c      : fonctions standards utilisées par les modules du noyau
└── Makefile     : description des actions possibles sur le code : compilation, exécution, nettoyage, etc.
```

Questions

1. Ouvrez le fichier Makefile, En ouvrant tous les fichiers dessiner le graphe de dépendance de `kernel.x` vis-à-vis de ses sources?

```
kernel.x : kernel.ld obj/hcpu.o obj/kinit.o obj/klibc.o obj/harch.o
obj/hcpu.o : hcpu.S hcpu.h
obj/kinit.o : kinit.c klibc.h harch.h hcpu.h
obj/klibc.o : klibc.c klibc.h harch.h hcpu.h
obj/harch.o : harch.c klibc.h harch.h hcpu.h
```

Ce fichier est produit par **Graphviz** à partir du fichier `makefile.dot` et la commande `dot -T png Makefile.dot -oMakefile.png`

```

digraph G {
  node [shape=box color=brown]
  gcc1[label="gcc -c"];
  gcc2[label="gcc -c"];
  gcc3[label="gcc -c"];
  gcc4[label="gcc -c"];
  ld[label="ld"];
  node [shape=ellipse color=blue]
  "hcpua.S" , "hcpu.h" --> gcc1 --> "obj/hcpu.o" --> ld --> "kernel.x"
  "kinit.c" , "klibc.h" , "harch.h" , "hcpu.h" --> gcc2 --> "obj/kinit.o" --> ld
  "klibc.c" , "klibc.h" , "harch.h" , "hcpu.h" --> gcc3 --> "obj/klibc.o" --> ld
  "harch.c" , "klibc.h" , "harch.h" , "hcpu.h" --> gcc4 --> "obj/harch.o" --> ld
}
  
```

1. Dans quel fichier se trouvent les codes dépendant du MIPS ?

- Ils sont dans le fichier `hcpua.S`

Exercices

- Le numéro du processeur est dans les 12 bits de poids faible du registre \$15 (`c0_cpuid`) du coprocesseur système (à côté des registres `c0_epc`, `c0_sr`, etc.). Ajoutez la fonction `int cpuid(void)` qui lit le registre `c0_cpuid` et qui rend un entier contenant juste les 12 bits de poids faible. Vous pouvez vous inspirer fortement de la fonction `int clock(void)`. Comme il n'y a qu'un seul processeur dans cette architecture, `cpuid` rend toujours 0. Écrivez un programme de test (vous devrez modifier les fichiers `hcpu.h`, `hcpua.S` et `kinit.c`)

hcpua.S

```

.globl cpuid
cpuid:
    mfc0    $2, $15
    andi    $2, $2, 0xFFF
    jr      $31
  
```

hcpu.h

```

/**
 * \brief    cpu identifier
 * \return   a number
 */
extern unsigned cpuid (void);
  
```

B4. Ajout de la librairie C pour l'utilisateur

Objectifs de l'étape

L'application utilisateur n'est pas censée utiliser directement les appels système. Elle utilise une librairie de fonctions standards (la `libc` POSIX, mais également d'autres) et ce sont ces fonctions qui réalisent les appels système. Toutes les fonctions de la `libc` n'utilisent pas les appels système. Par exemple, les fonctions `int rand(void)` ou `int strlen(char *)` (rendent, respectivement, un nombre pseudoaléatoire et la longueur d'une chaîne de caractères) n'ont pas besoin du noyau. Les librairies font partie du système d'exploitation mais elles ne sont pas dans le noyau.

Le terme « librairie » vient de l'anglais « library » qui signifie bibliothèque. On utilise souvent le mot librairie même si le sens en français n'est pas le même que celui en anglais. Disons que, dans notre contexte, les deux mots sont synonymes.

Normalement, les librairies système sont des « vraies » librairies au sens `gcc` du terme. C'est-à-dire des archives de fichiers objet (`.o`). Ici, nous allons simplifier et ne pas créer une vraie librairie, mais seulement un fichier objet `libc.o` contenant toutes les fonctions. Ce fichier objets doit être lié avec le code de l'application.

L'exécutable de l'application utilisateur est donc composé de deux parties : d'un côté, le code de l'application et, de l'autre, le code de la librairie `libc` (+ `crt0`). Nous allons répartir le code dans deux répertoires `uapp` pour les fichiers de l'application et `ulib` pour les fichiers qui ne sont pas l'application, c'est-à-dire la `libc`, le fichier `crt0.c` mais aussi le fichier ldscript `user.ld`.

On rappelle que le fichier `crt0.c` contient le code d'entrée dans l'application avec la fonction `_start()` appelée par la fonction `kinit()`. C'est aussi, dans ce fichier que l'on met le code assembleur de la fonction `syscall_fct()` permettant de revenir dans le noyau. En conséquence, `crt0.c`, c'est le pont entre le noyau et l'application.

Fichiers

```

04 libc/
├── Makefile
  
```

: Makefile racine qui invoque les Makefiles des sous-répertoires et qui exécute

| | |
|-------------|---|
| common | répertoire des fichiers commun kernel / user |
| syscalls.h | : API la fonction syscall et des codes de syscalls |
| kernel | Répertoire des fichiers composant le kernel |
| kinit.c | : fichier contenant la fonction de démarrage du noyau |
| harch.h | : API du code dépendant de l'architecture |
| harch.c | : code dépendant de l'architecture du SoC |
| hcpu.h | : prototype de la fonction clock() |
| hcpua.S | : code dépendant du cpu matériel en assembleur |
| hcpuc.c | : code dépendant du cpu matériel en c |
| klibc.h | : API de la klibc |
| klibc.c | : fonctions standards utilisées par les modules du noyau |
| ksyscalls.c | : Vecteurs des syscalls |
| kernel.ld | : ldscript décrivant l'espace d'adressage pour l'édition de liens du kernel |
| Makefile | : description des actions possibles sur le code kernel : compilation et nettoyage |
| uapp | Répertoire des fichiers de l'application user seule |
| main.c | : fonction principale de l'application |
| Makefile | : description des actions possibles sur le code user : compilation et nettoyage |
| ulib | Répertoire des fichiers des bibliothèques système liés avec l'application user |
| crt0.c | : fonctions d'interface entre kernel et user, pour le moment : _start() |
| libc.h | : API pseudo-POSIX de la bibliothèque C |
| libc.c | : code source de la libc |
| user.ld | : ldscript décrivant l'espace d'adressage pour l'édition de liens du user |
| Makefile | : description des actions possibles sur le code user : compilation et nettoyage |

Questions

- Pour ce petit système, dans quel fichier sont placés tous les prototypes des fonctions de la libc? Est-ce ainsi pour POSIX sur LINUX?
 - Ils sont tous dans le fichier `libc.h`.
 - Non, pour POSIX, les prototypes de fonctions de la libc sont répartis dans plusieurs fichiers suivant leur rôle. Il y a `stdio.h`, `string.h`, `stdlib.h`, etc. Nous n'avons pas voulu ajouter cette complexité.
- Dans quel fichier se trouve la définition des numéros de services tels que `SYSCALL_EXIT` ?
 - Ils sont dans le fichier `common/syscall.h`.
- Dans quel fichier se trouve le vecteur de syscall, c'est-à-dire le tableau `syscall_vector[]` contenant les pointeurs sur les fonctions qui réalisent les services correspondants aux syscall ?
 - Il est dans le fichier `kernel/ksyscall.c`.
- Dans quel fichier se trouve le gestionnaire de syscalls ?
 - Il est dans le fichier `kernel/hcpua.S`.

Exercice

Pour finir ce TME (un peu long 😊), vous allez juste ajouter une boucle d'affichage des caractères ASCII au début de la fonction `main()` en utilisant la fonction de la libc `fputc(tty,c)` (avec `tty` à 0 pour un affichage sur le terminal 0, et `c` la variable contenant le caractère à afficher, qui prendra toutes les valeurs entre 32 et 127).

- Je vous donne le code dans le corrigé, mais ça fait seulement 2 lignes, alors je pense que vous n'en aurez pas besoin ! 😊

```
for (int c = 32; c < 127; c++) {
    fputc (0, c);
}
```

- Ensuite, quand ça marche, exécutez le programme en mode debug (`make debug` au lieu de `make exec`) et ouvrez le fichier `trace0.s`. A quel cycle, commence la fonction `main()` ?
 - 7351 (sur ma machine, peut-être que cela peut varier si, entre temps, j'ai mis une autre version du simulateur)
- Recompilez le kernel en utilisant le mode `-O0` (lettre O suivie du chiffre zéro), réexécutez l'application en mode debug et regardez à nouveau à quelle cycle commence la fonction `main()` ?
 - 20603 (sur ma machine) c'est presque 3 fois plus lent !!!
- Pour finir, recompilez à nouveau le noyau en utilisant le mode `-O3`, réexécutez encore l'application en mode debug et regardez combien de cycles sont nécessaires pour exécuter la fonction `fputc()`. Pour ça, vous ouvrez le fichier `trace0.s`, vous cherchez le premier appel de `fputc()` (vous notez le cycle) et vous cherchez l'instruction `eret` qui marque la sortie du kernel (vous notez le cycle) et vous faites la différence ? Profitez en pour voir l'entrée dans le kernel, l'analyse de la cause, l'utilisation du vecteur de syscall, etc.
 - 7391 : appel de `fputc()`
 - 7589 : exécution de `eret`
 - 7589 - 7391 = 198 cycles (pour afficher 1 caractère !)
- Refaites le calcul pour le deuxième appel de `fputc()`, que constatez-vous ? Avez-vous une explication ?
 - 7620 : appel de `fputc()`
 - 7686 : exécution de `eret`
 - 7686 - 7620 = 66 cycles, c'est plus rapide, c'est à cause des caches que nous verrons plus tard !