# Application simple en mode utilisateur



Le TP est découpé en 4 étapes. Pour chaque étape, nous donnons :

1. une brève description avec une liste des objectifs principaux de l'étape,
2. une liste des fichiers avec un bref commentaire sur chaque fichier,
3. une liste de questions simples dont les réponses sont dans le code, le cours ou le TD, 4
4. un petit exercice de codage.

**IMPORTANT**
**Avant de faire cette séance, vous devez avoir regardé les documents suivants** :

> - Description des objectifs de cette séance et des suivantes : *déja lu*
> - Cours sur l'exécution d'une application en mode user *obligatoire*
> - Éléments d'information sur les outils de la chaîne de compilation *recommandé*
> - Document sur l'assembleur du MIPS et la convention d'appel des fonctions : *recommandé, mais déjà lu*
> - Documentation sur le mode kernel du MIPS32 : *obligatoire pour compléter le cours*

**Récupération du code du tp2**

- Ouvrez un `terminal`
- Allez dans le répertoire des TPs: `cd ~/k06`
  *... ou bien le répertoire que vous avez choisi pour faire les TPs.*
- Copiez les codes du **tp2**:
  `cp -rp /Infos/lmd/2024/licence/ue/LU3IN029-2024oct/k06/tp2 .`
- Exécutez la commande : `cd tp2 ; tree -L 1`.
  Vous devriez obtenir ceci :

```
.
├── 1_klibc
├── 2_appk
├── 3_syscalls
├── 4_libc
└── Makefile
```

**Objectif de la séance**

Les applications de l'utilisateur s'exécutent en mode user. Dans la séance précédente, nous avons vu que les registres de commande des contrôleurs de périphériques sont placés dans l'espace d'adressage du processeur. Les adresses de ces registres ont été placées dans la partie de l'espace d'adressage interdite en mode user. Ainsi, une application n'a pas un accès direct aux périphériques, elle doit utiliser des appels système (avec l'instruction `syscall`) pour demander au noyau du système d'exploitation de faire l'accès. C'est ce que nous allons voir.

Vous allez suivre 4 étapes :

- **`1_klibc`**
  → Ajout d'une librairie de fonctions standards pour le noyau (klibc), mais pas d'application utilisateur ;
- **`2_appk`**
  → Appel d'une application utilisateur par la fonction d'initialisation `kinit()`, mais sans gestionnaire des appels systèmes dans le noyau, il n'est donc pas possible d'utiliser `syscall` ;
- **`3_syscalls`**
  → Ajout du gestionnaire des appels système et une application, mais **sans** la librairie de fonctions standards utilisateur (libc);
- **`4_libc`**
  → Ajout d'une librairie de fonctions standards utilisateur minimaliste (libc) et d'une application.

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 et vous guidez, un peu, pour l'exercice. Les réponses sont dans le cours ou dans les fichiers sources. Certaines ont déjà été traitées en TD, c'est normal. Ensuite, vous passez aux exercices pratiques.

Le code se trouve normalement dans `k06/tp2/` (sinon téléchargez-le), ouvrez un terminal et allez-y. Dans ce répertoire, vous avez 4 sous-répertoires et un Makefile. Le fichier `k06/tp2/Makefile` qui fait un appel récursif à `make` et qui permet de faire le ménage en appelant les Makefiles des sous-répertoires avec la cible `clean`.

## 1. 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` (générateur de nombres pseudo-aléatoires). 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, même si ce n'est pas le but du TP. 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**

```
1_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()
├── hcpua.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**

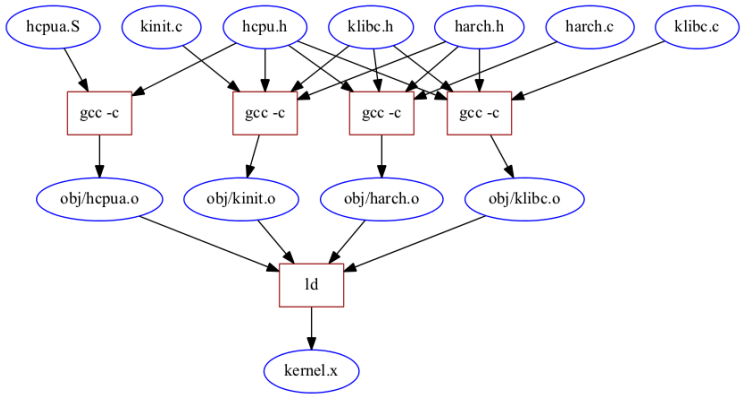
1. Ouvrez le fichier Makefile (vous pouvez regarder les dépendances en ouvrant quelques fichiers sources), puis dessiner le graphe de dépendance de `kernel.x` vis-à-vis de ses sources?
   La réponse peut-être visible avec la commande `dot -Tpng Makefile.dot -oMakefile.png` à partir du fichier `Makefile.dot` reproduit ci-après *( `dot` est un outil de graphviz ... essayez c'est magique :-)*

   `Makefile.dot`

   ```
   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/hcpua.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
   }
   ```

   ```
   kernel.x : kernel.ld obj/hcpua.o obj/kinit.o obj/klibc.o obj/harch.o
   obj/hcpua.o : hcpua.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
   ```

   `dot -Tpng Makefile.dot -oMakefile.png`

2. Dans quel fichier se trouvent les codes dépendant du MIPS (donc écrits en assembleur) ?
   - Ils sont dans le fichier `hcpua.S`

**Exercices**

- Dans un SoC, on peut avoir plusieurs processeurs. Le noyau a besoin de savoir sur quelle instance de processeur il s'exécute. Le numéro du processeur est dans les 12 bits de poids faible du registre du **coprocesseur-0 $15** ( `c0_cpuid` ) (à côté des registres `c0_epc` , `c0_sr` , etc. ce n'est pas le registre GPR $15). 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 inspirez fortement de la fonction `int clock(void)` . Comme il n'y a qu'un seul processeur dans cette architecture, `cpuid` rend toujours `0` .
  Ecrivez un programme de test (vous devrez modifier les fichiers `hcpu.h` pour y mettre le prototype de la fonction, `hcpua.S` pour y mettre le code et `kinit.c` pour appeler la fonction)

**hcpua.S**

```
.globl cpuid
cpuid:
    mfc0    $2, $15
    andi    $2, $2, 0xFFF  // masque pour ne conserver que les 12 bits de poids faible
    jr      $31
```

**hcpu.h**

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

## 2. Programme utilisateur mais exécuté en mode kernel

**Objectifs de l'étape**

Nous allons désormais avoir deux exécutables: le noyau et l'application. Dans cette étape, nous allons voir comment le noyau fait pour appeler l'application, alors même que celle-ci n'est pas compilée en même temps que le noyau. Nous allons passer du noyau à l'application à la fin de la fonction `kinit()` .

Nous allons donc entrer dans l'application, en revanche, dans cette étape, nous n'allons pas mettre en place la gestion des syscalls. **C'est-à-dire qu'il ne sera pas possible de revenir dans le noyau depuis l'application**. C'est bien entendu une étape intermédiaire, parce qu'il faudra absolument pouvoir invoquer le noyau depuis l'application pour accéder aux périphériques.

**Pour pouvoir quand même accéder aux registres de périphériques, nous allons EXCEPTIONNELLEMENT exécuter l'application en mode kernel. Ainsi, l'application pourra accéder aux adresses de l'espace d'adressage réservées au mode `kernel`.**

Nous avons deux exécutables à compiler et donc deux `Makefile`s de compilation. Nous avons aussi un `Makefile` qui invoque récursivement les `Makefile`s de compilation.

**Fichiers**

```
2_appk/
├── Makefile        : Makefile racine qui invoque les Makefiles des sous-répertoires et qui exécute
├── 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
│   ├── klibc.h     : API de la klibc
│   ├── klibc.c     : fonctions standards utilisées par les modules du noyau
│   ├── 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
└── user ────────── Répertoire des fichiers composant l'application user
    ├── crt0.c      : fonctions d'interface entre kernel et user, pour le moment : _start()
    ├── main.c      : fonction principale de l'application
    ├── 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**

1. Combien de fichiers de type ldscript avons-nous en tout ?
   - Il en faut deux désormais, un pour le kernel `kernel/kernel.ld` et un pour l'application `user/user.ld`
2. Dans quel fichier se trouve la première fonction de l'application et comment s'appelle-t-elle?
   - Dans le fichier `user/crt0.c`, c'est la fonction `_start()`.
3. Quelle est la fonction du noyau qui appelle cette fonction et dans quel fichier?
   - C'est la fonction `kinit()`, dans le fichier `kernel/kinit.c`.
4. Comment le noyau fait-il pour démarrer l'application en mode `kernel`? (la réponse est dans la fonction de la question précédente).
   - Dans la fonction `kinit()`, on appelle `app_load(&_start)`, c'est une fonction écrite en assembleur, donc forcément dans `hcpua.S`. Dans cette fonction on initialise `c0_epc` avec l'adresse de `_start()`, on initialise le bit `UM` du registre status `c0_sr` avec `0`. Ainsi, après l'exécution de `eret`, nous serons en mode `kernel`.

     ```
     .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,    2                // define next status reg. value
         mtc0    $26,    $12              // UM <- 0, IE <- 0, EXL <- 1
         eret                             // j EPC and EXL <- 0
     ```

**Exercice**

- Vous n'allez pas faire grand-chose pour cette étape parce qu'il est impossible de revenir dans le noyau après l'entrée dans l'application... tant qu'il n'y a pas le gestionnaire de syscalls dans le noyau. En conséquence, affichez juste un second message depuis la fonction `main()`.

## 3. Programme utilisateur utilisé en mode user mais sans libc

**Objectifs de l'étape**

Le programme utilisateur doit absolument s'exécuter en mode user et il doit passer par des appels système pour accéder aux services du noyau. Les services, ici, sont limités (l'accès au TTY, exit et clock), il n'empêche que pour gérer ces appels, il faut analyser la cause d'appels à l'entrée du noyau et un gestionnaire de `syscall`. Il faut aussi le gestionnaire d'exceptions, parce que s'il y a une erreur de programmation, le noyau doit afficher quelque chose pour aider le programmeur.

Le passage de l'application au noyau par le biais de l'instruction `syscall` impose que les numéros de services soient identiques pour le noyau et pour l'application. Ces numéros de service (comme `SYSCALL_TTY_PUTS`, `SYSCALL_EXIT` sont définis dans le fichier `syscall.h` communs au noyau et à l'application. Ce fichier est mis dans un répertoire à part nommé `common`. Il n'y a qu'un seul fichier ici, mais dans un système plus élaboré, il y en a d'autres.

**Fichiers**

```
3_syscalls/
├── 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
│   ├── kpanic.h    : déclaration du tableau de dump des registres en cas d'exception
│   ├── kpanic.c    : fonction d'affichage des registres avant l'arrêt du programme
│   ├── 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
└── user ──────────── Répertoire des fichiers composant l'application user
    ├── crt0.c      : fonctions d'interface entre kernel et user, pour le moment : _start()
    ├── main.c      : fonction principale de l'application
    ├── 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**

1. Dans quel fichier se trouve la définition des numéros de services tels que `SYSCALL_EXIT` ?
   (*Ces numéros sont communs au noyau et à l'application, alors ils ont été mis dans un répertoire commun aux deux exécutables.*)
   - Ils sont dans le fichier `common/syscall.h`. C'est dit dans l'énoncé !
2. 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 ? C'est un tableau indexé par les numéros de syscall qui permet de connaitre la fonction à exécuter pour chaque numéro de service.
   - Il est dans le fichier `kernel/ksyscall.c`.
3. Dans quel fichier se trouve le gestionnaire de syscalls ? (*c'est de l'assembleur*) et pourquoi est-ce de l'assembleur ?
   - Il est dans le fichier `kernel/hcpua.S`.
   - Le code du gestionnaire est totalement spécifique au MIPS, on ne peut pas l'écrire en C.

**Exercice**

- Vous allez ajouter un appel système nommé `SYSCALL_CPUID` qui rend le numéro du processeur. Nous allons lui attribuer le numéro 4 (notez que ces numéros de services n'ont rien à voir avec les numéros utilisés pour le simulateur MARS). Pour ajouter un appel système, vous devez modifier les fichiers :
  - `kernel/hcpua.S` : pour ajouter `cpuid(void)` déjà écrite à l'étape précédente;
  - `kernel/hcpu.h` : pour le prototype de la fonction `cpuid() (afin que gcc la connaisse pour la compilation de` kernel/ksyscall.c`;
  - `common/syscalls.h` : pour ajouter le numéro de syscall `SYSCALL_CPUID`;
  - `kernel/ksyscall.c` : pour ajouter la fonction `cpuid()` dans le vecteur de syscall.
- Modifiez aussi la fonction `main()` pour utiliser votre nouvel appel système. Vous pouvez utiliser le mode debug pour voir dans la trace d'exécution l'appel à

## 4. 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 pas uniquement) 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 pseudo aléatoire 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 objet 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 utilisateur
- `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`.

**Fichiers**

```
4_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**

1. 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.

**Exercice**

- Vous allez juste ajouter la fonction `int cpuid()` dans la librairie `libc`.
- Au premier TP, vous deviez créer un petit jeu 'guess', vous pouvez en faire une application utilisateur, en utilisant cette fois les fonctions de la `libc`.

*Dernière modification le 16 nov. 2024 à 16:25:25)*