INSA Lyon
4ème année
Spécialité Informatique

Systèmes d'exploitation avancés

Ordonnancement CPU, Gestion mémoire, Appels Système, Virtualisation

Kevin Marquet

Septembre 2014

Table des matières

Chapitre 1

Introduction

Ce document décrit les travaux pratiques associés au cours de systèmes d'exploitation avancés. Il guide l'implémentation d'un petit noyau de système d'exploitation. Ce système d'exploitation est destiné à être exécuté sur une plateforme embarquée : un Raspberry Pi. Avant l'implémentation proprement dite, une partie de ce sujet détaille donc quelques manipulations permettant de prendre en main les outils de développement : debugger et émulateur. Tous les développements effectués dans le cadre de cette partie du projet seront faits **exclusivement sous Linux**.

1.1 Déroulement

Ce semestre comprend 2 périodes :

Semaines 39 à 44 Travail en binômes

Semaines 47 à 4 Travail en héxanômes

Le graphe représenté sur la figure 1.1 donne une vue d'ensemble des travaux pratiques que vous allez réaliser durant ce semestre. Il se lit comme suit :

- Les nœuds verts sont des composants fournis.
- Les nœuds blancs montrent ce que vous allez faire pendant la première période, en binôme. Chacun de ces noeuds correspond grosso modo à une séance de TP.
- Les autres nœuds montrent ce que vous ferez par la suite, en hexanômes. Parmi ceux-ci, on distingue des travaux obligatoires (en gris) et d'autres optionnels (en pointillés rouge). Voyez la partie II pour les détails sur cette période.

Vous pouvez cliquer sur les liens indiqués au sein de chaque nœud pour accéder directement au chapitre correspondant. Ces chapitres listent les notions et compétences associés au chapitre, décrivent les travaux à effectuer, donnent des rappels succints sur les notions, ainsi que des références vers des explications plus détaillées.

1.2 Évaluation au cours du semestre

Pour rappel, la note de l'UE SEA, est calculé de la façon suivante : 0.6 x <note DS> + 0.4 x <note TP>. Les évaluations de TPs auxquelles vous aurez droit pendant le semestre sont les suivants :

Pendant la première période vous devez valider auprès d'un prof vos réponses aux questions des chapitres 3 à 5. Concrètement : au minimum à la fin de chaque exercice, vous appelez un prof et vous lui expliquez vos réponses et ce que vous avez compris. Le rôle du prof est de s'assurer que vous avez bien les compétences indiquées au début de chaque chapitre. Cela donnera lieu à une première note de TP. Si vous validez tous les exercices, vous aurez 17. À cette note, vous retranchez 5 points par exercice non validé, et vous ajoutez 2 points par fonctionnalité non demandé ajoutée au noyau. Ces deux points sont laissés à l'appréciation de votre enseignant de TP.

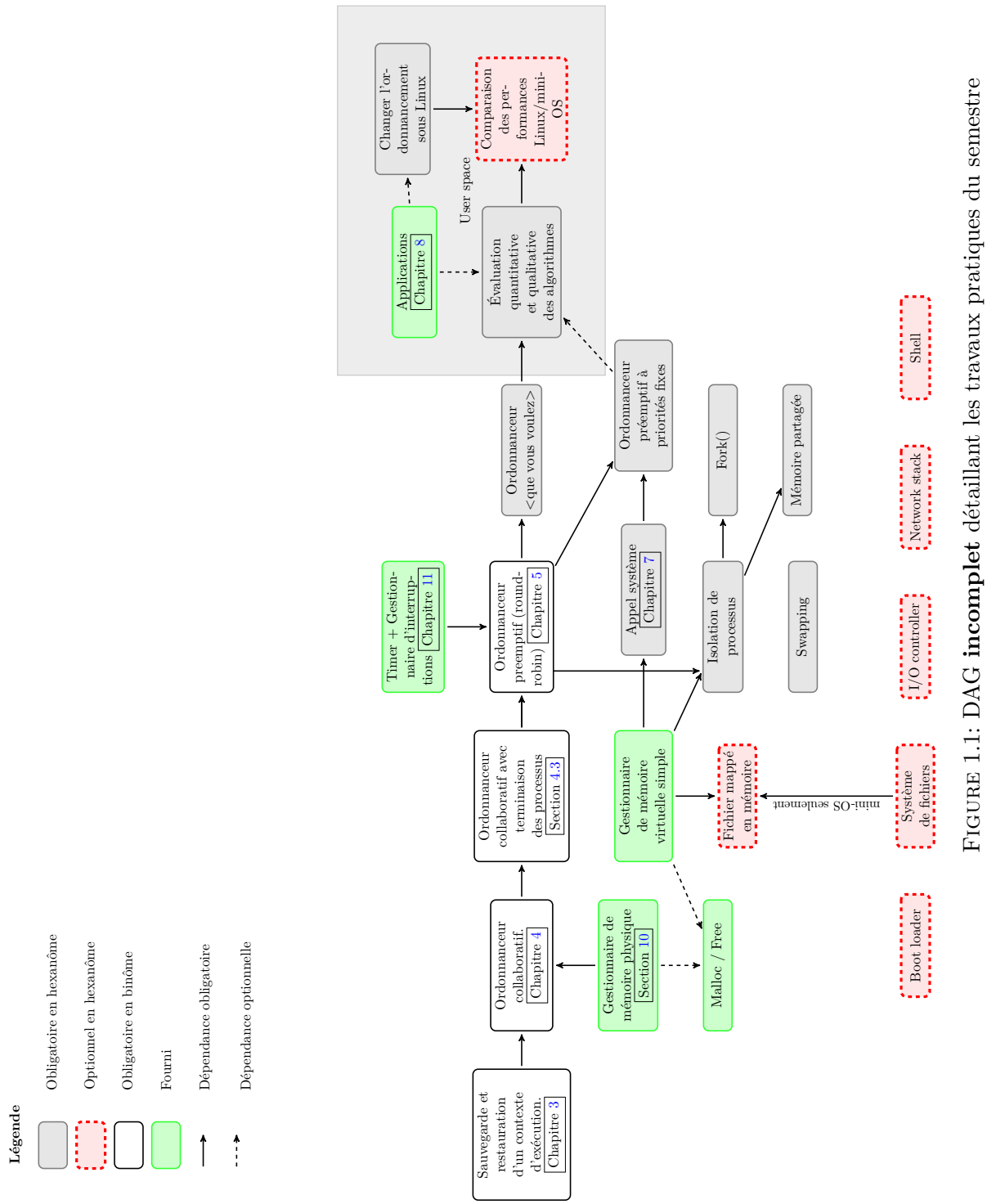

FIGURE 1.1: DAG **incomplet** détaillant les travaux pratiques du semestre

Au terme de la deuxième période un exposé présentant ce que vous avez fait et ce que vous avez compris. Voyez la Section 6.2 pour les détails. Cette note comptera pour 60% de la note de TP.

La note de DS prendra en compte les éléments suivants :

Au terme de la première période un QCM sur Moodle, portant sur le cours, les lectures et les TP donnera lieu à une première note de DS ;

Au mois de décembre un deuxième QCM sur Moodle portant sur le cours et les TPs donnera lieu à une deuxième note de DS.

En fin de semestre , le traditionnel DS portant principalement sur le cours, mais aussi sur les TPs, les questions posées dans le présent sujet et les question posées dans le poly de cours.

Ce qu'il est important de noter, c'est que les questions des QCM porteront sur tous les modules obligatoires du TP. Il s'agit donc de discuter entre vous au sein de l'hexanôme pour comprendre ce que chacun a fait.

Première partie

Semaines 39 à 44 : travail en binômes

Chapitre 2

Prise en main de l'environnement de compilation, exécution, et débogage

Ce chapitre décrit la prise en main des outils permettant de compiler et déboguer un programme destiné à s'exécuter sur la plateforme embarquée. En premier lieu, vous vous familiariserez avec le debogueur GDB et comment s'en servir pour déboguer un programme s'exécutant sur le PC. Puis vous verrez comment (cross-)compiler un programme pour qu'il s'exécute sur la plateforme cible, le Raspberry Pi, et enfin comment débogueur ce programme.

2.1 Ce que vous allez apprendre dans ce chapitre

PL / Code Generation : Concepts.

Concept	**Addressed ?**
Procedure calls and method dispatching	**[No]**
Separate compilation; linking	**[Yes]**
Instruction selection	**[No]**
Instruction scheduling	**[No]**
Register Allocation	**[No]**
Peephole optimization	**[No]**

PL / Code Generation : Skills.

1	Identify all essential steps for automatically converting source code into assembly or other low-level languages	**[Familiarity]**
2	Generate the low-level code for calling functions/methods in modern languages	**[Familiarity]**
3	Discuss why separate compilation requires uniform calling conventions	**[Not acquired]**
4	Discuss why separate compilation limits optimization because of unknown effects of calls	**[Not acquired]**
5	Discuss opportunities for optimization introduced by naive translation and approaches for achieving optimization, such as instruction selection, instruction scheduling, register allocation, and peephole optimization	**[Not acquired]**

2.2 Cross-compilation

Wikipedia.en :

> *A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.*

Voyez la figure 2.1. Dans le cas d'une compilation pour la machine hôte (votre PC contenant un processeur Intel, sous Linux), le fichier binaire contiendra de l'assembleur 8086. Dans votre cas, vous allez utiliser un cross-compilateur afin de produire de l'assembleur ARM pour la Raspberry Pi. Ce cross-compilateur est un port du

compilateur GCC, et les outils sont disponibles sur vos machine dans `/opt/4if-LS/arm-none-eabi-gcc/bin`. Ce chemin doit normalement déjà être présent dans votre `PATH`. Sinon, changez celui-ci.

Rappelez-vous... la cross-compilation

En 3if-Archi, vous pouviez, dans les options de l'IDE (IAR) configurer les options du cross-compilateur.

Point d'entrée de votre programme Le point d'entrée de votre code, c'est à dire l'endroit du code où le processeur va sauter après le boot est la fonction `kmain()`. Pour comprendre le boot du Raspberry Pi et comprendre pourquoi le processeur saute à cet endroit là, voyez l'encart page 12.

2.3 Émulation de la plateforme

Vous avez lu l'encart 2.3 de rappel sur la cross-compilation ? Eh bien sur les Raspberry Pi, c'est pas pareil. Enfin, disons qu'on n'a pas pris les semaines nécessaires pour faire marcher le connecteur JTAG via les pins de la Raspbery Pi. Mais pas grave, on va voir une autre manière de développer. On ne vous aura pas initié à GDB pour rien...

Solution donc : on va émuler la Raspberry Pi. C'est à dire qu'on va utiliser un émulateur. Wikipedia nous dit :

> an emulator is hardware or software or both that duplicates (or emulates) the functions of one computer system (the guest) in another computer system (the host), different from the first one, so that the emulated behavior closely resembles the behavior of the real system (the guest)

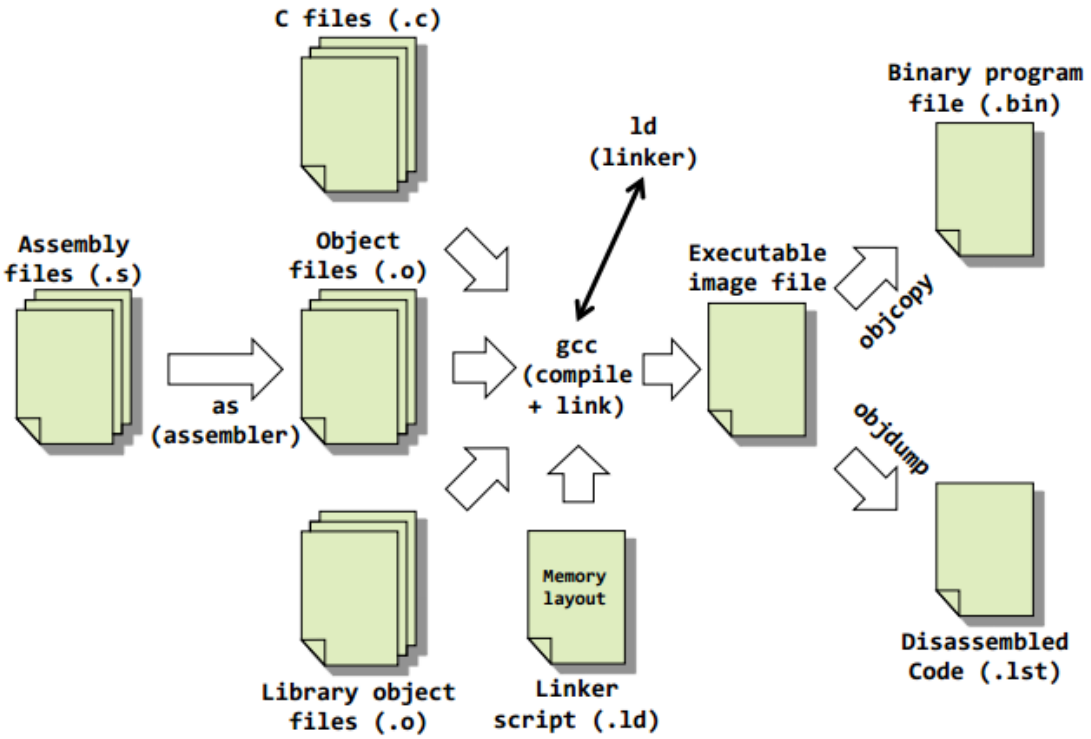

FIGURE 2.1: Chaîne de compilation

Rappelez-vous... le déboguage d'un program cross-compilé

En 3IF-Archi, on exécutait le programme pas à pas, sur la plateforme cible, à travers un connecteur JTAG et en utilisant l'interface de l'IDE IAR Workbench

Rappelez-vous... GDB

Vous avez déjà utilisé plusieurs fois gdb. Pour vous rafraîchir la mémoire, vous pouvez chercher *gdb cheat sheet* dans votre moteur de recherche préféré.

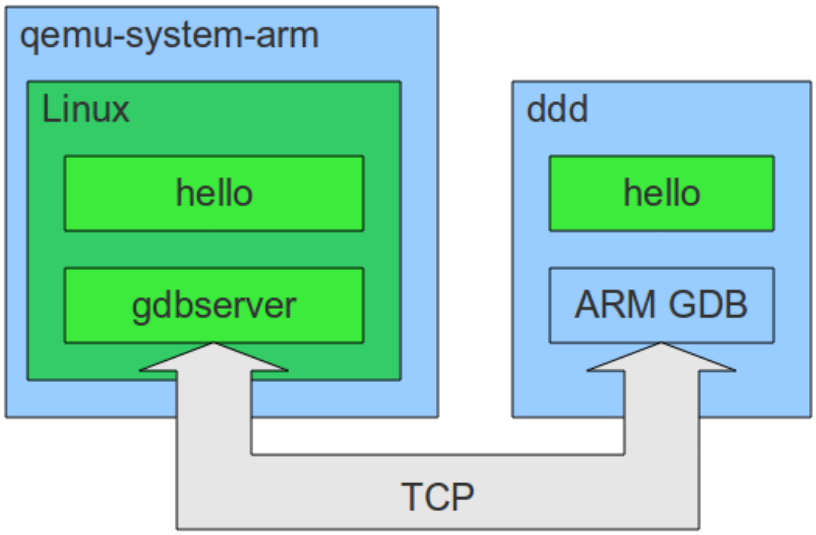

FIGURE 2.2: Remote debugging

L'émulateur qu'on va utiliser (qemu) est donc un logiciel capable d'éxécuter, sur le PC, le code binaire ARM. Et pour pouvoir déboguer ce programme, l'émulateur va se laisser piloter par GDB (via l'interface gdbserver) comme illustré par la figure 2.2. Sur cette figure, le programme compilé pour ARM et émulé par qemu s'appelle `hello` ; `ddd` est une surcouche graphique à gdb (pour notre part, on utilisera gdb directement).

Les scripts pour exécuter pas à pas vos programmes cross-compilés vous sont fournis. D'ailleurs, voyons tout de suite un exemple.

Exercice : Prise en main des outils

Mettez en place votre code de la façon suivante :

1. Téléchargez l'archive `ospie-start.tgz` sur :
   http://kevinmarquet.net/teaching/operating-systems/systeme-dexploitation-avances
2. Décompressez-la : `tar xzf ospie-start.tgz` et placez-vous dans le répertoire créé : `cd ospie-start`
3. Créez le fichier `kernel.c` et retapez-y le code de la figure 2.3.
4. Le fichier `Makefile` fourni permet de compiler — tapez '`make`'.
5. Placez-vous dans le répertoire débug et exécutez '`./run-qemu.sh`', qui lance l'émulateur.
6. Dans un **autre** terminal, depuis ce même répertoire, tapez `run-gdb.sh`.

7. C'est depuis le shell de gdb, que vous venez de lancer, que vous contrôlerez l'exécution de votre programme sur l'émulateur. Quand vous taperez (`continue` ou `c`), votre noyau s'initialisera et s'arrêtera au début de `kmain()`. Exécutez le programme pas à pas.
8. Le Makefile que l'on vous fournit produit une version désassemblée de votre mini-OS dans le fichier `kernel.list`. Éditez ce fichier et retrouvez-y les fonctions de votre fichier `kernel.c`

Vous aurez remarqué que le débogueur est arm-none-eabi-gdb, autrement dit, un gdb capable de lire comprendre l'assembleur ARM.

```
int
divide(int dividend, int divisor)
{
  int result = 0;
  int remainder = dividend;

  while (remainder >= divisor) {
    result++;
    remainder -= divisor;
  }

  return result;
}

int
compute_volume(int rad)
{
  int rad3 = rad * rad * rad;

  return divide(4*355*rad3, 3*113);
}

int
kmain( void )
{
  int radius = 5;
  int volume;

  volume = compute_volume(radius);

  return volume;
}
```

FIGURE 2.3: Un premier bout de code pour observer

2.4 Exécution sur la plateforme

(Vous n'aurez pas besoin de faire ces manipulations pendant les 2 premières séances)

Vous aurez besoin d'un Rasberry Pi, une alimentation, une carte SD et un cordon d'alimentation.

Pour exécuter votre programme sur le Raspberry Pi, remplacez juste le fichier `kernel.img` de votre carte SD que vous avez par le fichier `kernel.img` que le `Makefile` a compilé pour vous (dans le répertoire `SD_Card` pour la suite du TP).

Pour comprendre : le boot du Raspberry Pi

— Quand le Raspberry est mis en route, le processeur ARM n'est pas alimenté, mais le GPU (processeur graphique) oui. À ce point, la mémoire (SDRAM) n'est pas alimentée ;
— Le GPU exécute le premier bootloader, qui est constitué d'un petit programme stocké dans la ROM du micro-contrôleur. Ces quelques instruction lisent la carte SD, et chargent le deuxième bootloader stocké dans le fichier `bootcode.bin` dans le cache L2 ;
— L'excution de ce programme allume la SDRAM puis charge le troisième bootloader (fichier `loader.bin`) en RAM et l'exécute ;
— Ce programme charge `start.elf` à l'adresse zéro, et le processeur ARM l'exécute ;
— `start.elf` ne fait que charger `kernel.img`, dans lequel votre code se trouve ! Petit détail : le `start.elf` qu'on utilise sur la carte est celui d'une distribution Linux qui saute à l'adresse 0x8000 car des informations (paramétrage du noyau, configuration de la MMU) sont stockés entre les adresses 0x0 et 0x8000.

Plus d'infos :

— http://thekandyancode.wordpress.com/2013/09/21/how-the-raspberry-pi-boots-up/
— http://raspberrypi.stackexchange.com/questions/10442/what-is-the-boot-sequence
— http://www.raspberrypi.org/forums/viewtopic.php?f=63&t=6685

2.5 Documentation

Pour s'y retrouver dans les processeurs ARM, ce qu'il faut au moins avoir compris, c'est que ARM6 n'est pas pareil que ARMv6. ARM6 désigne une génération de processeurs alors que ARMv6 se réfère à un jeu d'instructions. Les deux ne vont pas de pair : les machines sur lesquelles vous allez travailler sont des processeurs ARM11 implémentant le jeu d'instruction ARMv6T. Ensuite, il suffit d'aller voir :

— http://en.wikipedia.org/wiki/ARM_architecture
— http://en.wikipedia.org/wiki/List_of_ARM_microprocessor_cores

La doc technique qui peut être utile est rangée dans `ospie-start/doc/hard/`. Vous y trouverez notamment :

— l'ARM Architecture Reference Manual (le "ARM ARM") décrivant le jeu d'instructions ARM : `ARM_Architecture_Reference Manual.pdf`
— la documentation du microntrôleur du Raspberry Pi : `BCM2835-ARM-Peripherals.pdf`
— la doc du processeur proprement dit : `DDI0301H_arm1176jzfs_r0p7_trm.pdf`

Plein d'info sur les Raspberry : http://elinux.org/RPi_Hub

Chapitre 3

Sauvegarde et restauration d'un contexte d'exécution

3.1 Ce que vous allez apprendre dans ce chapitre

OS / Concurrency : Concepts.

Concept	Addressed ?
States and state diagrams (cross reference SF/State-State Transition-State Machines)	[No]
Structures (ready list, process control blocks, and so forth)	[No]
Dispatching and context switching	[Yes]
The role of interrupts	[No]
Managing atomic access to OS objects	[No]
Implementing synchronization primitives	[No]
Multiprocessor issues (spin-locks, reentrancy) (cross reference SF/Parallelism)	[No]

OS / Concurrency : Skills.

1	Describe the need for concurrency within the framework of an operating system	[Not acquired]
2	Demonstrate the potential run-time problems arising from the concurrent operation of many separate tasks	[Not acquired]
3	Summarize the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each	[Usage]
4	Explain the different states that a task may pass through and the data structures needed to support the management of many tasks	[Not acquired]
5	Summarize techniques for achieving synchronization in an operating system (e.g., describe how to implement a semaphore using OS primitives)	[Not acquired]
6	Describe reasons for using interrupts, dispatching, and context switching to support concurrency in an operating system	[Usage]
7	Create state and transition diagrams for simple problem domains	[Not acquired]

3.2 Retour à un contexte

Un point d'exécution est caractérisé par l'état courant de la pile d'appel et des registres du processeur ; on parle de contexte. Dans les systèmes d'exploitation, les hyperviseurs, les machines virtuelles etc. le passage d'un contexte à l'autre est effectué par le dispatcher. C'est celui-ci que vous allez implémenter dans ce chapitre.

Contexte d'exécution

Cette section est un rappel du cours pour ce qui est des notions mais donne des détails à propos de l'architecture des processeurs de la famille ARM11 (i.e. jeu d'instruction ARMv6).

Pour s'exécuter, les procédures d'un programme en langage C sont compilées en code machine. Ce code machine exploite lors de son exécution des registres et une pile d'exécution. Voir l'encart de la présente page détaillant le processeur du Raspberry Pi.

Processeur du Raspberry Pi

La carte "Raspberry Pi" comprend un micro-contrôleur. Celui-ci contient un processeur ARM1176JZF, de la famille ARM11, jeu d'instruction ARMv6. Ses registres sont composés de :

- 13 registres généraux **R0** à **r12** ;
- Un registre **r13** servant de pointeur de pile. Il est aussi appelé **sp** (pour Stack Pointer) ;
- Un registre **r14**, aussi appelé **lr** (pour Link Register). Il a deux fonctions :
  - Lorsqu'un saut ou un appel de fonction est réalisé (typiquement grâce à l'instruction `'BL'`), ce registre contient l'adresse de retour de la fonction,
  - Lorsqu'une interruption arrive, l'adresse de l'instruction du programme interrompu est sauvegardé dans ce registre ;
- Le registre `CPSR` est le registre de statut (Status Register).

La documentation complète des architectures ARM et de notre processeur en particulier est en ligne, n'hésitez pas à y jeter un œil...

Le compilateur génère les instructions assembleur équivalentes au code C. Lorsqu'une procédure est appelée dans le code C, le compilateur génère des instructions sauvegardant les registres du microprocesseur au sommet de la pile, puis les arguments. Puis, le compilateur génère une instruction de branchement vers l'adresse de la fonction appelée, typiquement une instruction `bl`. Attention, une fois la fonction appelée exécutée, le processeur devra exécuter l'instruction suivant ce branchement. Pour cela, l'instruction `bl` sauvegarde cette adresse dans le registre `lr` avant de faire le saut proprement dit.

Ces conventions (ordre d'empilement typiquement) sont définies dans le document ARM Procedure Call Standard [1].

Sauvegarder les valeurs des registres suffit à mémoriser un contexte. Restaurer les valeurs de ces registres permet de se retrouver dans le contexte sauvegardé.

Attention, une fois **sp** restauré, les accès aux variables locales (allouées dans la pile d'exécution donc) ne sont plus possibles, ces accès étant réalisés par indirection à partir de la valeur de **sp**.

L'accès aux registres du processeur peut se faire par l'inclusion de code assembleur au sein du code C ; voir l'encart de la présente page sur l'assembleur en ligne.

Lier de l'assembleur et du code C

Le compilateur GCC autorise l'inclusion de code assembleur au sein du code C via la construction `asm()`. De plus, les opérandes des instructions assembleur peuvent être exprimées en C. Deux exemples :

- `__asm("mov %0, sp" : "=r"(varSP));` pour sauvegarder le registre `sp` dans une variable nommée `varSP` ;
- Le code suivant effectue l'opération inverse, à savoir lire la variable `varSP` et copier son contenu dans le registre `sp` : `__asm("mov sp, %0" : : "r"(varSP));`

Pour plus de détail sur la syntaxe et la sémantique de cette commande, voyez http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html Attention, cette construction est hautement non portable.

Un autre moyen d'intégrer de l'assembleur dans du C est d'écrire des fichiers `.s` tels que `vectors.s`, de les compiler avec un compilateur d'assembleur (e.g. GNU AS) et de lier le fichier objet obtenu avec les autres grâce au linker.

Exercice : observation du code à l'exécution

Question 3-1

— Observez les valeurs de **sp** en exécutant le programme simple que vous avez utilisé en Section 2.3 ;

1. les plus curieux peuvent se documenter ici : http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf

— Comparez la valeur de **sp** avec les adresses des première et dernière variables locales déclarées dans ces fonctions ;
— Comparez cette valeur avec les adresses des premier et dernier paramètres de ces fonctions ;
— Dessinez la mémoire autour de l'adresse contenue dans **sp** en y reportant toutes les adresses ;
— Dans quel sens croît la pile ?
— Observez l'utilisation du registre **lr** avant et après chaque appel de fonction, ainsi qu'au retour de chacune. Quelles instructions assembleur le met à jour implicitement ? Au besoin, re-lisez le rôle du registre **lr** dans l'encart page précédente.

Validation Faites valider votre travail avant de passer à la suite !

3.3 Un dispatcher pour coroutines

Nous allons implanter un mécanisme de coroutines. Les coroutines sont des procédures qui s'exécutent dans des contextes séparés. Ainsi une procédure `ping` peut « rendre la main » à une procédure `pong` sans pour autant terminer son exécution, et la procédure `pong` peut faire de même avec la procédure `ping` ensuite ; `ping` reprendra son exécution dans le contexte dans lequel elle était avant de passer la main.

Dans l'exercice décrit ci-après, vous allez commencer par vous placer dans un cas très simple où seuls deux contextes existent, les deux coroutines étant simplistes :

1. elles ne prennent pas de paramètres ;
2. elles ne contiennent aucune variable locale.

Vu la simplicité des deux fonctions considérées, considérez dans l'exercice ci-après qu'il vous suffit de sauvegarder l'adresse de l'instruction en cours d'exécution, ainsi que le pointeur de pile pour pouvoir reprendre l'exécution d'un processus.

Gestion de la mémoire

De la mémoire doit être allouée pour toutes les structures de données (pile d'excution, PCB...). Cependant, vous n'avez pas (encore) écrit le gestionnaire mémoire de votre petit système d'exploitation. Nous vous en fournissons un très simple qui ne virtualise pas la mémoire mais utilise la mémoire physique directement. Pour allouer de la mémoire, utilisez la fonction

```
void* phyAlloc_alloc(unsigned int size)
```

Cette fonction alloue `size` octets de données. Pour libérer la mémoire, utilisez

```
void phyAlloc_free(void* ptr, unsigned int size)
```

Ces deux fonctions sont déclarées dans `phyAlloc.h` et définies dans `phyAlloc.c`. Le gestionnaire mémoire est initialisé dans `init_hw()`.

Exercice : Dispatcher pour coroutines

Question 3-2 Remplacez le code de votre fonction `kmain` par celui de la figure 3.1. Que fait ce programme ? Ne tenez pas compte pour le moment de l'appel à `init_hw()` : cette fonction ne fait qu'initialiser des structures de données et du matériel, notamment le gestionnaire mémoire.

Question 3-3 Définissez la structure de données `struct ctx_s` dans un fichier `sched.h`.

Question 3-4 Dans le fichier `sched.c`, définissez la variable globale `current_ctx` qui pointe en permanence sur le contexte en cours d'exécution : soit `ctx_ping`, soit `ctx_pong` dans notre exemple.

```
#include "sched.h"
#include "dispatcher.h"

struct ctx_s ctx_ping;
struct ctx_s ctx_pong;
struct ctx_s ctx_init;

void
ping()
{
  while ( 1 ) {
    switch_to(&ctx_pong);
  }
}

void
pong()
{
  while ( 1 ) {
    switch_to(&ctx_ping);
  }
}

//----------------------------------------------------------------------
int
kmain ( void )
{
  init_hw();
  init_ctx(&ctx_ping, ping, STACK_SIZE);
  init_ctx(&ctx_pong, pong, STACK_SIZE);

  current_ctx = &ctx_init;
  switch_to(&ctx_ping);

  /* Pas atteignable vues nos 2 fonctions */
  return(0);
}
```

FIGURE 3.1: Ce code doit tourner sans modification

Question 3-5 Quelle taille de pile vous semble-t'il raisonnable d'allouer ? Argumentez et définissez une macro `STACK_SIZE` que vous pourrez utiliser à chaque création de contexte.

Question 3-6 Définissez dans un fichier `sched.c` la fonction :

```
void init_ctx(struct ctx_s* ctx, func_t f, unsigned int stack_size)
```

qui initialise un contexte (dont une pile...). Attention, veillez à avoir lu l'encart page précédente sur la gestion de la mémoire pour l'allocation de la pile d'exécution. Le type `func_t` est un pointeur de fonction, défini de la manière suivante : `typedef void (*func_t) ( void);`

Notes pour cette question :

— dans le code fourni, le contexte `ctx_init` est un contexte qui n'est plus utile une fois la première coroutine appelée. Il ne servira que dans les deux premiers prochains exercices. Il sert uniquement à la simplicité de mise en oeuvre.

— faisez gaffe, les contextes dans lesquels s'exécutent `ping()` et `pong()` sont déclarés dans `kernel.c` et sont donc alloués statiquement. Il vous est donc inutile de gérer leur allocation dynamique (vous inquiétez pas hein, vous vous chargerez plus tard de l'allocation des contextes, quand vous gèrerez plusieurs processus) ;

Question 3-7 Modifiez le Makefile pour qu'il compile votre fichier `sched.c`. Il n'y a qu'à l'ajouter à la variable `C_FILES`.

(Rappelez vous...) Les attributs de fonction

En C, vous pouvez, dans la déclaration d'une fonction ou d'une variable, spécifier un ou plusieurs attributs. Un attribut est une information donné au compilateur pour l'aider à optimiser sa génération de code, ou lui indiquer de compiler le code d'un certaine manière. L'attribut qui nous intéresse ici est l'attribut `naked`. Une fonction déclarée comme telle sera compilée sans les prologue ni épilogue qui, au début et à la fin de chaque fonction, sauvegarde (resp. restore) des registres qui seront utilisés dans la fonction ; charge au programmeur de le faire. C'est typiquement utilisé pour déclarer un traitant d'interruption. Exemple : `void __attribute__((naked)) timer_handler();`

En 3if-archi, vous avez déjà utilisé ce genre de fonctionnalités, mais sous IAR, lorsque vous avez ajouté à vos fonctions des décorations comme `#pragma vector=TIMERA0_VECTOR` et le mot-clef `__interrupt`. Sous GCC, pour obtenir la même chose chose, il aurait fallu écrire `__attribute__((__interrupt__(TIMER0_A0_VECTOR)))`.

Question 3-8 Assurez-vous d'avoir compris ce que fait l'attribut `naked` (voyez l'encart 7). Observez la différence entre le code assembleur d'une fonction déclarée avec et sans cet attribut (par exemple en allant voir dans le fichier `kernel.list`).

Question 3-9 Déclarez puis définissez la primitive suivante qui permet de changer de contexte :

```
void __attribute__ ((naked)) switch_to(struct ctx_s* ctx);
```

Cette primitive, lorsqu'elle est appelée, va :

1. Sauvegarder le contexte courant ;
2. Changer de contexte courant (faire pointer `current_ctx` vers le contexte `ctx` passé en paramètre) ;
3. Restaurer ce nouveau contexte ;
4. Sauter à l'adresse de retour du contexte restauré.

Attention, pensez à relire la description du registre `lr` ! Les deux seules instructions assembleur dont vous aurez besoin sont `mov` (illustrée précédemment) et `bx` (dont l'usage est simple – voyez l'ARM ARM).

NB : je vous rappelle que modifier un pointeur ne veut pas dire modifier la structure de donnée pointée !

Question 3-10 Compilez et exécutez pas à pas le programme. Vérifiez qu'il passe bien successivement d'un contexte à l'autre.

3.4 Sauvegarde des variables locales

On complexifie : remplacez vos fonctions `ping` et `pong` par celles de la figure 3.2.

Remarquez que ces deux fonctions possèdent maintenant des variables locales, et effectuent des calculs. Il ne suffit donc plus de sauvegarder et restaurer les pointeurs de pile et d'instruction mais tous les registres. Vous pourriez sauvegarder ces registres dans les structures `struct ctx_s`, mais je vous suggère de plutôt les sauvegarder dans la pile d'exécution. C'est ce qui est fait de manière traditionnelle dans les systèmes d'exploitation, les machines virtuelles Java, etc. Le principe est simple : sur un appel à `switch_to(...)`, votre OS empile tous les registres sur le haut de pile puis passe la main au contexte passé en paramètre ; les registres de ce dernier sont dépilés et restaurés avant que celui-ci reprenne son exécution.

```
void
ping()
{
  int cpt = 0;

  while ( 1 ) {
    cpt ++;
    switch_to(&ctx_pong);
  }
}

void
pong()
{
  int cpt = 1;

  while ( 1 ) {
    cpt += 2 ;
    switch_to(&ctx_ping);
  }
}
```

FIGURE 3.2: Avec les variables locales

Exercice : Dispatcher v2 (le retour)

Question 3-11 Faites les modifications nécessaires pour autoriser ce fonctionnement, grâce à ces deux instructions assembleur :

- utilisez l'instruction assembleur `push` pour sauvegarder les registres sur la pile d'excution ;
- utilisez l'instruction assembleur `pop`.

Vérifiez le bon fonctionnement pas à pas du programme.

Chapitre 4

Ordonnanceur collaboratif

4.1 Ce que vous allez apprendre dans ce chapitre

OS / Concurrency : Concepts.

Concept	Addressed ?
States and state diagrams (cross reference SF/State-State Transition-State Machines)	[Yes]
Structures (ready list, process control blocks, and so forth)	[Yes]
Dispatching and context switching	[Yes]
The role of interrupts	[No]
Managing atomic access to OS objects	[No]
Implementing synchronization primitives	[No]
Multiprocessor issues (spin-locks, reentrancy) (cross reference SF/Parallelism)	[No]

OS / Concurrency : Skills.

1	Describe the need for concurrency within the framework of an operating system	[Not acquired]
2	Demonstrate the potential run-time problems arising from the concurrent operation of many separate tasks	[Usage]
3	Summarize the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each	[Usage]
4	Explain the different states that a task may pass through and the data structures needed to support the management of many tasks	[Not acquired]
5	Summarize techniques for achieving synchronization in an operating system (e.g., describe how to implement a semaphore using OS primitives)	[Not acquired]
6	Describe reasons for using interrupts, dispatching, and context switching to support concurrency in an operating system	[Usage]
7	Create state and transition diagrams for simple problem domains	[Not acquired]

OS / Scheduling and Dispatch : Concepts.

Concept	Addressed ?
Preemptive and non-preemptive scheduling (cross reference SF/Resource Allocation and Scheduling, PD/Parallel Performance)	[No]
Schedulers and policies (cross reference SF/Resource Allocation and Scheduling, PD/Parallel Performance)	[No]
Processes and threads (cross reference SF/computational paradigms)	[Yes]
Deadlines and real-time issues	[No]

SF / Resource Allocation and Scheduling : Skills.

1	Define how finite computer resources (e.g., processor share, memory, storage and network bandwidth) are managed by their careful allocation to existing entities	[Not acquired]
2	Describe the scheduling algorithms by which resources are allocated to competing entities, and the figures of merit by which these algorithms are evaluated, such as fairness	[Not acquired]
3	Implement simple schedule algorithms	[Usage]
4	Measure figures of merit of alternative scheduler implementations	[Not acquired]

4.2 Passons à un nombre de processus supérieur à 2

La primitive `switch_to` du mécanisme de coroutines impose au programmeur d'expliciter le nouveau contexte à activer. À travers la série de question ci-après, vous allez donc déclarer et définir une interface de manipulation de processus :

```
int create_process(func_t f, void *args, unsigned int stack_size);
void __attribute__ ((naked)) ctx_switch();
```

La primitive `create_process()` ajoute à l'ancienne `init_ctx()` l'allocation dynamique de la structure. La primitive `ctx_switch()` permet au contexte courant de passer la main à un autre contexte, ce dernier étant déterminé par l'ordonnanceur. Un des objectifs de cet ordonnanceur est de choisir, lors d'un changement de contexte, le nouveau processus à activer.

Pour cela, l'ordonnanceur a besoin d'information sur les processus. Comme on l'a vu en cours, pour chaque processus, ces données sont regroupées dans un PCB (Process Control Block). Ces PCBs doivent être stockés, par exemple sous la forme d'une structure chaînée circulaire comme vous l'avez vu en cours.

Un PCB doit également contenir un pointeur de fonction et un pointeur pour les arguments de la fonction. Cette fonction sera celle qui sera appelée lors de la première activation du processus. On suppose que le pointeur d'arguments est du type `void*`. La fonction appelée aura tout loisir d'effectuer une coercition de la structure pointée dans le type attendu.

Exercice : Réalisation de l'ordonnanceur collaboratif

Question 4-1 Faites une sauvegarde de vos fichiers. Puis remplacez le contenu de kernel.c par celui dela figure 4.1. Ce code devra tourner sans modification. Observez les différences avec l'ancien.

Question 4-2 Définissez un type de donnée pour l'état d'un processus dans le fichier `sched.h`.

Question 4-3 Proposez une structure de données `struct pcb_s` pour un PCB (toujours dans `sched.h`).

Question 4-4 Créez la fonction `init_pcb` dans le fichier `sched.c` qui initialise la structure définie ci-dessus.

Question 4-5 Toujours dans le fichier `sched.c`, implémentez l'ordonnanceur, au travers des quatre fonctions suivantes :

- `void create_process(func_t f, void* args, unsigned int stack_size)` Cette fonction alloue un nouveau PCB, l'ajoute à la liste chaînée des PCBs, et l'initialise en appelant `init_pcb` ;
- `void start_current_process()` est appelée pour lancer un processus (pour la première fois). Elle appelle la fonction qui a été donnée comme point d'entrée du processus (le paramètre `f` de `create_process()`).
- `void elect()` choisit le prochain processus et fait pointer la variable globale `current_process` sur son PCB. Elle ne sauvegarde ni ne restaure les contextes d'exécution ! C'est ctx_switch qui s'en chargera.
- `void start_sched()` initialise quelques variables globales.

Question 4-6 Écrivez, dans le fichier `sched.c`, la fonction `void __attribute__ ((naked)) ctx_switch()` qui permet de passer la main au prochain processus en 3 étapes :

1. sauvegarde le contexte du processus en cours d'exécution
2. demande au scheduler d'élire un nouveau processus
3. restaure le contexte du processus élu

Question 4-7 Que se passe-t'il lors de la première invocation de `ctx_switch()` si l'on n'y prend pas garde ? Remédiez éventuellement à ce problème.

```c
#include "sched.h"

void
funcA()
{
  int cptA = 0;

  while ( 1 ) {
    cptA ++;
    ctx_switch();
  }
}

void
funcB()
{
  int cptB = 1;

  while ( 1 ) {
    cptB += 2 ;
    ctx_switch();
  }
}

//----------------------------------------------------------------------
int
kmain ( void )
{
  init_hw();
  create_process(funcB, NULL, STACK_SIZE);
  create_process(funcA, NULL, STACK_SIZE);

  start_sched();
  ctx_switch();

  /* Pas atteignable vues nos 2 fonctions */
  return 0;
}
```

FIGURE 4.1: Ce code doit tourner sans modification

Question 4-8 Après que le registre de piles ait été initialisé sur une nouvelle pile d'exécution, les éventuelles variables locales et arguments de la fonction `ctx_switch()` seraient-ils utilisables ?

4.3 Terminaison des processus

Lorsqu'un programme se termine, son contexte d'exécution ne doit plus pouvoir être utilisé, et les structures de données inutiles doivent être désallouées.

Question 4-9 Ajoutez le support pour la terminaison propre d'un processus et testez-le sur une modification de `kmain.c`. Pour commencer, regardez du côté de `start_current_process()` : lorsqu'un processus est terminé, on revient de l'appel à `pcb->entry_point()`.

Chapitre 5

Ordonnanceur préemptif

5.1 Ce que vous allez apprendre dans ce chapitre

OS / Concurrency : Concepts.

Concept	Addressed ?
States and state diagrams (cross reference SF/State-State Transition-State Machines)	[Yes]
Structures (ready list, process control blocks, and so forth)	[No]
Dispatching and context switching	[No]
The role of interrupts	[Yes]
Managing atomic access to OS objects	[No]
Implementing synchronization primitives	[No]
Multiprocessor issues (spin-locks, reentrancy) (cross reference SF/Parallelism)	[No]

OS / Concurrency : Skills.

1	Describe the need for concurrency within the framework of an operating system	[Not acquired]
2	Demonstrate the potential run-time problems arising from the concurrent operation of many separate tasks	[Not acquired]
3	Summarize the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each	[Not acquired]
4	Explain the different states that a task may pass through and the data structures needed to support the management of many tasks	[Usage]
5	Summarize techniques for achieving synchronization in an operating system (e.g., describe how to implement a semaphore using OS primitives)	[Not acquired]
6	Describe reasons for using interrupts, dispatching, and context switching to support concurrency in an operating system	[Usage]
7	Create state and transition diagrams for simple problem domains	[Not acquired]

OS / Scheduling and Dispatch : Concepts.

Concept	Addressed ?
Preemptive and non-preemptive scheduling (cross reference SF/Resource Allocation and Scheduling, PD/Parallel Performance)	[Yes]
Schedulers and policies (cross reference SF/Resource Allocation and Scheduling, PD/Parallel Performance)	[Yes]
Processes and threads (cross reference SF/computational paradigms)	[No]
Deadlines and real-time issues	[No]

OS / Scheduling and Dispatch : Skills.

1	Compare and contrast the common algorithms used for both preemptive and non-preemptive scheduling of tasks in operating systems, such as priority, performance comparison, and fair-share schemes	**[Not acquired]**
2	Describe relationships between scheduling algorithms and application domains	**[Not acquired]**
3	Discuss the types of processor scheduling such as short-term, medium-term, long-term, and I/O	**[Not acquired]**
4	Describe the difference between processes and threads	**[Not acquired]**
5	Compare and contrast static and dynamic approaches to real-time scheduling	**[Usage]**
6	Discuss the need for preemption and deadline scheduling	**[Not acquired]**
7	Identify ways that the logic embodied in scheduling algorithms are applicable to other domains, such as disk I/O, network scheduling, project scheduling, and problems beyond computing	

SF / Resource Allocation and Scheduling : Skills.

1	Define how finite computer resources (e.g., processor share, memory, storage and network bandwidth) are managed by their careful allocation to existing entities	**[Not acquired]**
2	Describe the scheduling algorithms by which resources are allocated to competing entities, and the figures of merit by which these algorithms are evaluated, such as fairness	**[Not acquired]**
3	Implement simple schedule algorithms	**[Usage]**
4	Measure figures of merit of alternative scheduler implementations	**[Not acquired]**

5.2 Ordonnancement sur interruption

L'ordonnanceur programmé jusqu'ici est un ordonnancement avec partage volontaire du processeur : un contexte passe la main grâce à un appel explicite à `switch_to()`. Nous allons maintenant programmer un ordonnancement préemptif avec partage involontaire du processeur : l'ordonnanceur va être capable d'interrompre le processus en cours d'exécution et de changer de processus. Cet ordonnancement est basé sur la génération d'interruptions. Une interruption déclenche l'exécution d'une fonction associée à l'interruption (un gestionnaire d'interruption ou handler). Nous vous fournissons les primitives suivantes (dans `hw.c/hw.h`) :

```
void init_hw();
void DISABLE_IRQ();
void ENABLE_IRQ();
void set_tick_and_enable_timer();
```

La primitive `init_hw()` effectue plusieurs tâches d'initialisation du matériel. Notamment, cette primitive configure un timer pour qu'il génère une interruption toute les 10ms. Le vecteur d'interruption est configuré pour sauter au label `irq` dans le fichier `vectors.s`.

Les deux primitives `DISABLE_IRQ()` et `ENABLE_IRQ()` permettent de délimiter des zones de code devant être exécutées de manière non interruptible. Au démarrage, les interruptions sont désactivées.

`set_tick_and_enable_timer` permet quant à elle de ré-armer le timer. Pensez à l'utiliser, le matériel le désactive après chaque interruption. Attention : après cet appel, le timer génèrera un signal sur sa ligne d'interruption mais le bit GIE (rappelez-vous la 3IF...) doit être positionné par un `ENABLE_IRQ()` pour que leprocesseur en tienne compte.

Exercice : Première version de l'ordonnanceur préemptif

Question 5-1 Modifiez votre fonction `start_sched()` pour activer les interruptions.

Question 5-2 Dans la questions suivante, vous allez implémenter la fonction `ctx_switch_from_irq`. Afin de debugger plus facilement, vous devriez mettre un point d'arrêt dans cette fonction. Pour cela, comprenez la commande du fichier `debug/run-gdb.sh` et modifiez le fichier `debug/gdbinit`.

Question 5-3 Faites ce qu'il faut pour que, sur une interruption du timer, le gestionnaire d'interruption exécute la fonction `ctx_switch_from_irq()` (que vous aurez créé).

Il vous faut maintenant implémenter la fonction `ctx_switch_from_irq()`. Or, dans les ARM pas tout jeunes comme celui du Raspberry Pi, lorsqu'une interruption a lieu, le processeur se met dans un mode d'exécution

particulier — le mode `irq` dans notre cas. Les privilèges changent donc (certaines instructions ne peuvent être exécutées dans certains modes d'exécution) et le processeur utilise une **nouvelle version matérielle** de certains registres afin de ne pas modifier la valeur de ceux du mode d'exécution qui s'est vu interrompre (System pour nous).

Question 5-4 Lisez les sections A2.2 Processor modes et A2.3 Registers de l'ARM ARM. Quels registres ont une version matérielle différente dans le mode d'exécution IRQ que dans le mode System ?

Vous aurez remarqué que le pointeur de pile n'est plus le même, il a été changé par le processeur ! C'est un problème car vous avez besoin de sauvegarder l'adresse de l'instruction en cours d'exécution ainsi que le registre de status dans la pile d'exécution du processus interrompu et non dans la pile du mode `irq`. Vous pourriez le faire en quelques lignes d'assembleur mais le plus simples est d'utiliser les instructions suivantes :

```
__asm("sub  lr, lr, #4");
__asm("srsdb  sp!, #0x13");
__asm("cps #0x13");
```

Explications :

- La première instruction décrémente **`lr`** de façon à ce qu'il pointe effectivement vers l'instruction interrompue. En effet, lorsqu'une interruption survient, le processeur sauvegarde l'adresse (+4 !) de l'instruction en cours d'exécution avant l'interruption dans le registre **`lr`** du mode `irq`. Pour quelques détails, lisez la petite section A2.6.8 Interrupt request (IRQ) exception de l'ARM ARM.
- La deuxième instruction est une instruction SRS qui sauvegarde **`lr`** et le registre de statut **`cpsr`** dans la pile du mode d'exécution `System` du processeur.
- La troisième passe le processeur en mode `System`. Après cette instruction, les registres (y compris **`sp`** !) sont donc à nouveau ceux du processus interrompu.

Si vous avez suivi, vous aurez compris qu'il vous faudra restaurer le registre de statut et l'adresse de retour à la fin de `ctx_switch_from_irq()`. Pour cela, vous devrez utiliser l'instruction `RFE` décrite dans la section A4.1.59 RFE de l'ARM ARM.

Question 5-5 Lisez la description de l'instruction SRS (section A4.1.90). Puisqu'on a utilisé SRSDB, donnez la configuration de l'instruction RFE à utiliser pour que les 2 soient cohérentes.

Question 5-6 Que veut dire le caractère '!' si on l'utilise dans l'instruction `RFE` ? Mettez à jour votre instruction RFE le cas échéant.

Question 5-7 Implémentez la fonction `ctx_switch_from_irq()`.

5.3 Gestion des sections critiques

Exercice : Protection des structures de données partagées

Votre ordonnanceur est maintenant préemptif, il reste à isoler les sections critiques de code ne devant pas être interrompues par un gestionnaire d'interruptions.

Question 5-8 Ajoutez les appels nécessaires à `DISABLE_IRQ()` et `ENABLE_IRQ()` dans le code de l'ordonnanceur.

Deuxième partie

Semaines 45 à 52 : projet (en hexanômes)

Chapitre 6

Présentation du projet

6.1 Objectifs

L'objectif général de ce projet est d'illustrer les notions vues en cours associées à l'ordonnancement et aux systèmes temps-réel. Vous illustrerez ces notions à la fois au-dessus de Linux (la distribution Raspbian est présente sur les cartes SD initialement : pensez à sauvegarder le fichier `kernel.img` si vous l'écrasez) et dans votre mini-OS et dans les deux cas, on s'intéresse à l'exécution sur Raspberry Pi.

Sur chacun des systèmes, vous devrez tester différentes politiques d'ordonnancement permettant de mettre en évidence :

- le besoin de synchronization entre processus ;
- le besoin d'ordonnancement à priorités ;
- l'impact de la politique d'ordonnancement sur les performances ;
- les différents critères de choix des algorithmes d'ordonnancement ;
- le fait que les processus sont bornés en temps ou en I/O.

Concrètement, vous devrez :

- Terminer votre mini-OS (aller jusqu'au bout du sujet précédent) ;
- Implémenter, dans votre mini-OS, différentes politiques d'ordonnancement différentes de round-robin, dont au moins une à priorités ;
- Implémenter un ensemble de processus fonctionnant au-dessus de votre mini-OS, permettant d'illustrer les points ci-dessus.
- Implémenter un ensemble de processus fonctionnant au-dessus de Linux, permettant d'illustrer les différences entre l'ordonnanceur `SCHED_OTHER` et les ordonnanceurs temps-réel `SCHED_RR` et `SCHED_FIFO`.
- Préparer exposé et démonstrations (voir plus bas).

6.2 Déroulement et rendu

Le projet dure 5 séances. Une bonne partie de la dernière séance sera consacrée à l'évaluation. Cette évaluation prendra la forme d'un exposé oral pendant lequel vous présenterez votre projet à l'aide de quelques slides et de démonstrateurs. Bien que ces critères soient susceptibles d'évoluer, cet oral sera noté de la manière suivante :

Chapitre 7

Appels système

7.1 Ce que vous allez apprendre dans ce chapitre

OS / Operating System Principles : Concept

Concept	Addressed ?
Structuring methods (monolithic, layered, modular, micro-kernel models)	[No]
Abstractions, processes, and resources	[No]
Concepts of application program interfaces (APIs)	[No]
Application needs and the evolution of hardware/software techniques	[No]
Device organization	[No]
Interrupts : methods and implementations	[Yes]
Concept of user/system state and protection, transition to kernel mode	[Yes]

OS / Operating System Principles : Skills.

1	Explain the concept of a logical layer.	[Not acquired]
2	Explain the benefits of building abstract layers in hierarchical fashion.	[Usage]
3	Describe the value of APIs and middleware.	[Not acquired]
4	Describe how computing resources are used by application software and managed by system software.	[Not acquired]
5	Contrast kernel and user mode in an operating system.	[Usage]
6	Discuss the advantages and disadvantages of using interrupt processing.	[Usage]
7	Explain the use of a device list and driver I/O queue.	[Not acquired]

7.2 Implémentons les appels système

Cette partie vous guide dans l'implémentation des appels système ; voyez l'encart 7.2 pour un rappel de ce que sont les appels système. Le mécanisme permettant d'implémenter ces appels système est relativement simple. Il s'agit de déclencher une *interruption logicielle* grâce à l'instruction assembleur `SWI` puis de traiter cette interruption dans un traitant d'interruption (avec les privilèges noyau).

Appels système

Wikipedia : *Un appel système (en anglais, system call, abrégé en syscall) est une fonction primitive fournie par le noyau d'un système d'exploitation et utilisée par les programmes utilisateur.*
Rappelons, de manière simplifiée, que les programmes *utilisateur* ont des privilèges moindres que le noyau. TODO : exemple privilège

Exercice : Appel système, sans paramètre

Vous allez commencer par écrire un mécanisme d'appel système simple, qui permette d'exécuter du mode noyau via des fonctions sans paramètres.

Question 7-1 Écrivez une fonction

```
void syscall(int number)
```

qui déclenche une interruption logicielle, traitée par la fonction

```
void syscall_handler()
```

Ce traitant d'interruption doit appeler la fonction correspondant au numéro passé en paramètre.

Question 7-2 Donner un exemple d'appel système Linux qui ne serait pas possible avec ce mécanisme simpliste.

Exercice : Appel système, avec paramètres

Afin de pallier aux limitations de l'appel système sans paramètres, vous allez ajouter le support de ceux-ci. Cependant, afin de ne pas gérer un nombre variables de paramètres, nous allons nous limiter à 2 paramètres. Ces deux paramètres ne serviront pas forcément.

Question 7-3 Modifiez votre fonction `syscall` pour qu'elle prenne 2 paramètres. Puis modifiez votre traitant d'interruption pour qu'il récupère ces paramètres afin de les passer

Question 7-4 Donner un exemple d'appel système Linux qui ne serait pas possible avec seulement 2 paramètres ?

Exercice : Implémentation d'appels système classiques

Question 7-5 Rappelez-vous : qu'appelle-t-on un quantum de temps ?

Question 7-6 Implémentez l'appel système suivant :
— `void wait(int nq)` qui fait attendre un processus `nq` quantums de temps.

Question 7-7 Implémentez votre fonction `yield()` comme un appel système.

Question 7-8 Décrivez le problème posé potentiellement par l'interruption de votre noyau en cours de traitement d'appel système.

Question 7-9 Assurez-vous que votre noyau ne peut pas être intérrompu en cours de traitement d'un appel système.

Chapitre 8

Suggestions d'applications

Voici quelques exemples d'applications que vous pouvez utiliser ou implémenter pour illustrer les concepts vus en cours. Gardez à l'esprit que vous cherchez à illustrer ces concepts, et pas "juste" à programmer un jeu vidéo ou à jouer de la musique.

8.0.1 Clignotage de la LED

Les fonctions `led_on()` et `led_off()` permettent d'allumer et éteindre la LED. Assurez-vous que votre mini-OS fonctionne avec deux processus, l'un éteignant la LED régulièrement, l'autre l'allumant. Sous Linux, vous pouvez utiliser http://wiringpi.com/.

8.0.2 Lecteur MIDI

On vous fournit en ligne une archive comprenant un lecteur midi fonctionnel pour votre mini-OS. Attention, ce code est pas (encore) tout à fait safe et certains hexanômes ont passé du temps de debug là-dessus, parfois sans succès (sorry...) ; pour d'autres ça a marché facilement donc faîtes juste attention à pas y passer trop de temps.

8.0.3 Installer des logiciels sous Linux

Vous êtes libres d'installer les logiciels que vous voulez sous Linux (lecteur vidéo etc.). Également, on vous fournit une archive `pmidi.tgz` comprenant les sources d'un lecteur midi fonctionnant au dessus de Linux. Pour compiler, make. Pour exécuter, tapez `play_midi_file` <filename>. Attention, pour cela vous aurez besoin d'installer le paquet `timidity++`.

Pour installer des paquet, vous aurez besoin de connecter le Raspberry Pi à internet. Pour cela, servez-vous d'un portable comme passerelle. Ci-dessous, les commandes à taper pour une passerelle sous Linux (ce n'est pas garanti de marcher à tous les coups, utilisez vos cours de réseau) :

— Sur le Raspberry Pi :

```
sudo ifconfig eth0 192.168.0.2 netmask 255.255.255.0
sudo route add default gw 192.168.0.1
```

— Sur le PC :

```
sudo ifconfig eth0 192.168.0.1 netmask 255.255.255.0
echo 1 > /proc/sys/net/ipv4/ip_forward /sbin/iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
/sbin/iptables -A FORWARD -i eth0 -o eth1 -m state --state RELATED,ESTABLISHED -j ACCEPT
/sbin/iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
```

Attention : pour jouer du son sous Linux et l'entendre, il vous faut indiquer à la couche ALSA que vous voulez que le son sorte par la sortie casque et pas HDMI :
`sudo amixer cset numid=3 1`.

8.0.4 Lecteur WAV

On vous fournit en ligne un lecteur de .wav (archive `wav_player.tgz`), fonctionnant uniquement au-dessus de Linux (désolé, pas eu le temps de coder le lecteur .wav pour le mini-OS, même si vous pouvez reprendre les couches basses du lecteur de fichiers MIDI). Attention, un décodeur lecteur wav, ça ne demande pas beaucoup de ressources, donc pas forcément utile pour vous.

8.0.5 Un clavier pour votre mini-OS

Un tutoriel sur le web http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input01.html explique comment récupérer les appuis de touches par polling. Vous pourriez donc avoir un processus dont c'est le rôle.

8.0.6 Synthétiseur de son (ou autre action suite à l'appui sur une touche)

Dans votre mini-OS, vous pouvez jouer un son différent selon la touche du clavier (ou afficher/éteindre la LED). Sous Linux, c'est aussi possible : il existe plein de logiciel libre disponible.

8.0.7 Jeux : casse-briques, tétris etc.

C'est pas compliqué, si vous utilisez les bonnes librairies (en tous cas au-dessus de Linux...). Et parfait pour illustrer problèmes de latence, de synchronisation (surtout si vous jouez de la musique en même temps), de performances... mais ça demande peut-être un peu de boulot. En même temps, vous êtes 6 :)

8.0.8 Lecture vidéo

Au-dessus de Linux, vous devez pouvoir vous en sortir. Au-dessus de votre mini-OS, c'est déconseillé : on n'a pas pris le temps d'écrire ou récupérer un driver vidéo (mais n'hésitez pas :))

8.1 Synchronisation entre contextes

On introduit un mécanisme de synchronisation entre contextes à l'aide de sémaphores. Un sémaphore est une structure de données composée :

— d'un compteur ;
— d'une liste de contextes en attente sur le sémaphore.

Le compteur peut prendre des valeurs entières positives, négatives, ou nulles. Lors de la création d'un sémaphore, le compteur est initialisé à une valeur donnée positive ou nulle ; la file d'attente est vide.

Un sémaphore est manipulé par les deux actions atomiques suivantes :

— `sem_down()` (traditionnellement aussi nommée `wait()` ou `P()`). Cette action décrémente le compteur associé au sémaphore. Si sa valeur est négative, le processus appelant se bloque dans la file d'attente.
— `sem_up()` (aussi nommée `signal()`, `V()`, ou `post()`) Cette action incrémente le compteur. Si le compteur est négatif ou nul, un processus est choisi dans la file d'attente est devient actif.

Deux utilisations sont faites des sémaphores :

— la protection d'une ressource partagée (par exemple l'accès à une variable, une structure de donnée, une imprimante...). On parle de sémaphore d'exclusion mutuelle ;

Rappel de cours

Typiquement le sémaphore est initialisé au nombre de processus pouvant concurremment accéder à la ressource (par exemple 1) et chaque accès à la ressource est encadré d'un couple

```
sem_down(S) ;
  <accès à la ressource>
```

```
    sem_up(S) ;
```

— la synchronisation de processus (un processus doit en attendre un autre pour continuer ou commencer son exécution).

Rappel de cours

(Par exemple un processus 2 attend la terminaison d'un premier processus pour commencer.) On associe un sémaphore à l'événement, par exemple `findupremier`, initialisé à 0 (l'événement n'a pas eu lieu) :

```
Processus 1 :                          Processus 2 :
    <action 1>                             sem_down(findupremier) ;
    sem_up(findupremier) ;                 <action 2>
```

Bien souvent, on peut assimiler la valeur positive du compteur au nombre de processus pouvant acquérir librement la ressource ; et assimiler la valeur négative du compteur au nombre de processus bloqués en attente d'utilisation de la ressource. Un exemple classique est donné dans l'encart page suivante.

```
#define N 100                          /* nombre de places dans le tampon */

struct sem_s mutex, vide, plein;

sem_init(&mutex, 1);                 /* controle d'acces au tampon */
sem_init(&vide, N);                  /* nb de places libres */
sem_init(&plein, 0);                 /* nb de places occupees */

void producteur (void)
{
  objet_t objet ;

  while (1) {
    produire_objet(&objet);            /* produire l'objet suivant */
    sem_down(&vide);                   /* dec. nb places libres */
    sem_down(&mutex);                  /* entree en section critique */
    mettre_objet(objet);               /* mettre l'objet dans le tampon */
    sem_up(&mutex);                    /* sortie de section critique */
    sem_up(&plein);                    /* inc. nb place occupees */
  }
}

void consommateur (void)
{
  objet_t objet ;

  while (1) {
    sem_down(&plein);                  /* dec. nb emplacements occupes */
    sem_down(&mutex);                  /* entree section critique */
    retirer_objet (&objet);            /* retire un objet du tampon */
    sem_up(&mutex);                    /* sortie de la section critique */
    sem_up(&vide);                     /* inc. nb emplacements libres */
    utiliser_objet(objet);             /* utiliser l'objet */
```

```
  }
}
```

Le classique producteur consommateur

Une solution du problème du producteur consommateur au moyen de sémaphores est donnée ici. Les deux utilisations types des sémaphores sont illustrées. Persuadez-vous qu'il n'est pas possible pour le producteur (resp. le consommateur) de prendre le sémaphore `mutex` avant le sémaphore `plein` (resp. `vide`).
Testez votre implantation des sémaphores sur un exemple comme celui-ci.
Ajoutez une boucle de temporisation dans le producteur que le changement de contexte puisse avoir lieu avant que le tampon ne soit plein.
Essayez d'inverser les accès aux sémaphores `mutex` et `plein`/`vide` ; que constatez-vous ? Votre implémentation peut-elle détecter de tels comportements ?

Exercice : Implémentation des sémaphores

Question 8-1 Donnez la déclaration de la structure de donnée associée à un sémaphore.

Question 8-2 Proposez une implantation de la primitive

```
void sem_init(struct sem_s* sem, unsigned int val);
```

Question 8-3 En remarquant qu'un contexte donnée ne peut être bloqué que dans une unique file d'attente d'un sémaphore, ajouter une structure de données à votre ordonnanceur pour qu'il puisse gérer les processus bloqués.

Question 8-4 Proposez une implantation des deux primitives

```
void sem_up(struct sem_s* sem);
void sem_down(struct sem_s* sem);
```

8.2 Prévention des interblocages

On ajoute aux sémaphores introduit précédemment un mécanisme d'exclusion mutuel sous la forme de simples verrous :

— un verrou peut être libre ou verrouillé par un contexte ; ce contexte est dit propriétaire du verrou ;
— la tentative d'acquisition d'un verrou non libre est bloquante.

L'interface de manipulation des verrous est la suivante :

```
void mtx_init(struct mtx_s* mutex);
void mtx_lock(struct mtx_s* mutex);
void mtx_unlock(struct mtx_s* mutex);
```

Comparés aux sémaphores, l'utilisation des verrous est plus contraignantes : seul le contexte propriétaire du verrou peut le libérer et débloquer un contexte en attente du verrou. De manière évidente, les verrous peuvent être simulés par des sémaphores dont la valeur initiale du compteur serait 1.

Exercice : Le dîner des philosophes

L'académique et néanmoins classique problème des philosophes est le suivant : cinq philosophes attablés en cercle autour d'un plat de spaghettis mangent et pensent alternativement sans fin (faim ?). Une fourchette

est disposée entre chaque couple de philosophes voisins. Un philosophe doit préalablement s'emparer des deux fourchettes qui sont autour de lui pour manger.

Vous allez élaborer une solution à ce problème en attachant un processus à l'activité de chacun des philosophes et un verrou à chacune des fourchettes.

Montrez qu'une solution triviale peut mener à un interblocage, aucun des philosophes ne pouvant progresser.

Question 8-5 Comment le système peut-il prévenir de tels interblocages ?

Vous considèrerez que

— un contexte est bloqué sur un verrou ;
— un verrou bloque un ensemble de contextes ;
— un contexte détient un ensemble de verrous.

Considérez aussi les situations dans lesquelles toutes les activités ne participent pas à l'interblocage. Par exemple, une sixième activité indépendante existe en dehors des cinq philosophes.

Question 8-6 Modifiez l'interface de manipulation des verrous pour que le verrouillage retourne une erreur en cas d'interblocage :

```
void mtx_init(struct mtx_s* mutex);
int mtx_lock(struct mtx_s* mutex);
void mtx_unlock(struct mtx_s* mutex);
```

Chapitre 9

Allocation dynamique de mémoire

9.1 Une première bibliothèque standard

La bibliothèque C standard fournit un ensemble de fonctions permettant l'accès aux services du système d'exploitation. Parmi ces services, on trouve l'allocation et la libération de mémoire, au travers les deux primitives suivantes :

void *malloc (unsigned size) ; La fonction `malloc()` de la bibliothèque retourne un pointeur sur un bloc d'au moins size octets.

void free (void *ptr) ; La fonction `free()` permet de libérer le bloc préalablement alloué pointé par `ptr`, quand il n'est plus utile.

Cette partie du sujet consiste à implémenter vos propres fonctions d'allocation et libération de mémoire (qu'on appelera `gmalloc` et `gfree`). Dans un premier temps, nous réaliserons une implémentation simple et efficace de ces primitives. Dans un deuxième temps, vous les optimiserez.

La mise en place de l'environnement de développement vous est laissée. Vous pouvez vous inspirer de ce qui vous est fourni pour la partie ordonnancement (Makefile...).

9.1.1 Principe

Lors de la création d'un processus, un espace mémoire lui est alloué, contenant la pile d'exécution de ce processus, le code de celui-ci, les variables globales, ainsi que le tas, dans lequel les allocations dynamiques effectuées au travers `gmalloc()` sont effectuées. Ce tas mémoire est accessible le champs `heap` de la structure associé au processus (voir le chapitre 3). Au début de l'exécution du processus, ce tas ne contient aucune donnée. Il va se remplir et se vider au grè des appels à `gmalloc()` et `gfree()` : à chaque appel à `gmalloc()`, un bloc va être alloué dans le tas, à chaque appel à `gfree`, un bloc va être libéré, menant à une fragmentation du tas.

Dans notre implantation de ces fonctions, l'ensemble des blocs mémoire libres va être accessible au moyen d'une liste chaînée. Chaque bloc contient donc un espace vide, la taille de cette espace vide et un pointeur sur le bloc suivant. Le dernier bloc pointera sur le premier.

9.1.2 Implémentation de gmalloc()

Lors d'un appel à `gmalloc()`, on cherche dans la liste de blocs libres un bloc de taille suffisante. L'algorithme first-fit consiste à parcourir cette liste chaînée et à s'arrêter au premier bloc de taille suffisante. Un algorithme best-fit consiste à utiliser le "meilleur" bloc libre (selon une définition de "meilleur' donnée'). Nous allons implémenter le first-fit.

Si le bloc a exactement la taille demandée, on l'enlève de la liste et on le retourne à l'utilisateur. Si le bloc est trop grand, on le divise en un bloc libre qui est gardé dans la liste chaînée, et un bloc qui est retourné à l'utilisateur. Si aucun bloc ne convient, on retourne un code d'erreur.

9.1.3 Implémentation de gfree()

La libération d'un espace recherche l'emplacement auquel insérer ce bloc dans la liste des blocs libres. Si le bloc libéré est adjacent à un bloc libre, on les fusionne pour former un bloc de plus grande taille. Cela évite une fragmentation de la mémoire et autorise ensuite de retourner des blocs de grande taille sans faire des appels au système.

9.2 Optimisations de la bibliothèque

Votre bibliothèque peut maintenant être optimisée. Implémentez donc les améliorations que vous saurez trouver/imaginer, en vous inspirant entre autres des points suivants :

Pré-allocation Une des optimisations effectuées par la librairie C standard sous Unix est la mise en place de listes chaînés de blocs d'une certaine taille. Gardez en mémoier que ce système est efficace pour de petites tailles.

Détection d'utilisation illégale Les primitives telles qu'elles sont définies peuvent être mal utilisées, et votre implémentation peut favoriser la détection de ces utilisations frauduleuses. Quelques exemples d'utilisation frauduleuse :

— Passage à `gfree()` d'un pointeur ne correspondant pas à un précédent `gmalloc()`
— Supposition de remplissage d'un segment alloué à zéro
— Débordement d'écriture
— Utilisation d'un segment après l'avoir rendu par `gfree()`

Outils Vous pouvez favoriser le débogage en fournissant des outils tels que l'affichage des listes chaînées ou l'affichage des blocs alloués.

Spécialisation aux applications Puisque vous connaissez l'utilisation qui sera faite de votre bibliothèque, vous pouvez spécialiser son fonctionnement. Bien sûr vous perdrez en généricité, il faut savoir dans quelle mesure.

Troisième partie

Code fourni

Chapitre 10

Gestion de la mémoire physique

L'ensemble du code et des données utilisées par votre mini-OS est, jusqu'à présent, alloué statiquement, c'est à dire par le compilateur dans le fichier exécutable du noyau. Ces allocations correspondent soit aux variables globales, soit aux variables locales déclarées "static", soit encore au code. Toutefois, comme dans de nombreux systèmes d'exploitation, votre mini-OS pourra allouer dynamiquement de la mémoire supplémentaire selon ses besoins. Cela permettra par exemple de créer des processus.

La première étape pour permettre l'allocation dynamique de mémoire est de gérer la mémoire physique, c'est-à-dire la mémoire physiquement présente dans l'ordinateur. Pour cela, nous vous fournissons un petit gestionnaire de mémoire physique très simple. Il maintient simplement une liste de blocs libres. Vous pouvez allouer et libérer de la mémoire grâce aux fonctions suivantes :

— `char* malloc_alloc(uint32_t size)` qui alloue `size` octets consécutifs dans la mémoire physique
— `void malloc_free(void* ptr)` qui libère la zone pointée par `ptr`

Chapitre 11

Interruptions

Rappelez-vous, en 3IF vous avez configuré un timer sur les cartes TI à base de MSP430. Vous avez dû positioner un vecteur d'interruption et écrire le gestionnaire (le handler) d'interruption associé. Sur le Raspberry, cela fonctionne exactement de la même manière, mais nous avons fait le job à votre place. Ce que vous avez besoin de savoir, c'est que :

— le timer est réglé pour déclenché une interruption toutes les 10 ms.

— lorsqu'une interruptoin se produit, la fonction appelée est `ctx_swicth()`. Pour changer afin d'appeler une autre fonction lors d'une interruption, ça se passe dans `vectors.s`