

INSA Lyon  
4ème année  
Spécialité Informatique

## **Systèmes d'exploitation avancés**

Ordonnancement CPU, Gestion mémoire, Appels Système, Virtualisation

Kevin Marquet  
30 septembre 2015

# Table des matières

<b>Table des matières</b>	<b>2</b>
<b>1 Introduction</b>	<b>5</b>
1.1 Déroulement . . . . .	5
1.2 Évaluation au cours du semestre . . . . .	5
<b>I Semaines 39 à 44 : travail en binômes</b>	<b>8</b>
<b>2 Prise en main de l'environnement de compilation, exécution, et débogage</b>	<b>9</b>
2.1 Ce que vous allez apprendre dans ce chapitre . . . . .	9
2.2 Cross-compilation . . . . .	9
2.3 Émulation de la plateforme . . . . .	10
2.4 Exercice : écrire (un peu) d'assembleur ARM . . . . .	14
2.5 Exercice : l'attribut naked . . . . .	15
2.6 Exercice : utiliser les tests fournis . . . . .	15
2.7 Exercice : exécution sur la plateforme . . . . .	15
2.8 Documentation . . . . .	16
<b>3 Modes d'exécution du processeur</b>	<b>17</b>
3.1 Les modes d'exécution ARM . . . . .	17
3.2 Organisation de la mémoire physique . . . . .	18
3.3 Exercice : changement de mode d'exécution . . . . .	18
<b>4 Appels système</b>	<b>21</b>
4.1 Ce que vous allez apprendre dans ce chapitre . . . . .	21
4.2 Appels système . . . . .	21
4.3 Exercice : un premier appel système qui ne retourne pas . . . . .	22
4.4 Exercice : un appel système qui retourne . . . . .	25
4.5 Exercice : un appel système avec passage de paramètres . . . . .	27
4.6 Exercice : un appel système qui retourne une valeur . . . . .	28
<b>5 Process dispatching</b>	<b>30</b>
5.1 Ce que vous allez apprendre dans ce chapitre . . . . .	30
5.2 Introduction . . . . .	30
5.3 Exercice : un dispatcher pour coroutines . . . . .	31
5.4 Exercice : des coroutines avec des variables locales . . . . .	33
<b>6 Ordonnancement collaboratif (round-robin)</b>	<b>36</b>
6.1 Ce que vous allez apprendre dans ce chapitre . . . . .	36
6.2 Ordonnancement . . . . .	37

6.3	Exercice : ordonnancement round-robin . . . . .	37
6.4	Exercice : terminaison des processus . . . . .	38
6.5	Exercice facultatif : terminaison non explicite d'un processus . . . . .	39
<b>7</b>	<b>Ordonnanceur préemptif</b>	<b>40</b>
7.1	Ce que vous allez apprendre dans ce chapitre . . . . .	40
7.2	Ordonnancement sur interruption . . . . .	41
<b>II</b>	<b>Semaines 45 à 52 : projet (en hexanômes)</b>	<b>44</b>
<b>8</b>	<b>Présentation du projet</b>	<b>45</b>
8.1	Objectifs . . . . .	45
8.2	Déroulement et rendu . . . . .	46
8.3	Organisation du travail au sein de l'hexanôme . . . . .	46
<b>9</b>	<b>Gestion de la mémoire virtuelle : pagination et protection</b>	<b>47</b>
9.1	Rappels sur la mémoire virtuelle . . . . .	47
9.2	Le coprocesseur . . . . .	47
9.3	La MMU . . . . .	48
9.4	Pagination . . . . .	49
9.5	Exercice : Pagination sans allocation dynamique . . . . .	52
9.6	Exercice : allocation dynamique de pages . . . . .	53
9.7	Exercice : isolation entre processus . . . . .	55
9.8	Annexe : Détail du c1/Control register . . . . .	56
9.9	Annexes : la protection chez ARM . . . . .	57
<b>10</b>	<b>Partage de la mémoire</b>	<b>60</b>
<b>11</b>	<b>Suggestions d'applications</b>	<b>61</b>
11.1	Sortie vidéo . . . . .	61
11.2	Clignotage de la LED . . . . .	61
11.3	Sortie série . . . . .	61
11.4	Lecteur WAV . . . . .	61
11.5	Installer des logiciels sous Linux . . . . .	62
11.6	Un clavier pour votre mini-OS . . . . .	62
11.7	Synthétiseur de son (ou autre action suite à l'appui sur une touche) . . . . .	62
11.8	Jeux : casse-briques, tétis etc. . . . .	62
11.9	Synchronisation entre contextes . . . . .	62
11.10	Prévention des interblocages . . . . .	64
<b>12</b>	<b>Allocation dynamique de mémoire</b>	<b>66</b>
12.1	Une première bibliothèque standard . . . . .	66
12.2	Optimisations de la bibliothèque . . . . .	67
<b>13</b>	<b>Linux sur Raspberry Pi</b>	<b>68</b>
13.1	Installation et exécution de Linux . . . . .	68
13.2	Accès à internet . . . . .	68
13.3	Installation de logiciel . . . . .	68
13.4	L'ordonnancement sous Linux . . . . .	68

<b>III Annexes</b>	<b>69</b>
<b>A Interruptions</b>	<b>70</b>
<b>B Gestionnaire de mémoire physique simple</b>	<b>71</b>
<b>C Installation des outils nécessaires à la réalisation du projet</b>	<b>72</b>
C.1 Installation de Qemu pour Raspberry Pi . . . . .	72
C.2 Installation de GCC pour ARM . . . . .	72
C.3 Installation de GDB pour ARM . . . . .	72
<b>D FAQ</b>	<b>74</b>
<b>E Aide-mémoire gdb</b>	<b>75</b>
<b>Glossary</b>	<b>78</b>

# 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**.

Éléments de lecture :

- Quand vous rencontrez une **notion**, vous pouvez cliquer sur son nom afin d'avoir sa définition ;
- Un glossaire en fin de document regroupe la liste des **notions**.

### 1.1 Déroulement

Ce semestre comprend 2 périodes :

**Semaines 39 à 44** Travail en binômes

**Semaines 47 à 4** Travail en hexanômes

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.
- Les autres nœuds montrent ce que vous ferez par la suite, en hexanômes. Parmi ceux-ci, distinguez les travaux obligatoires (en gris) des 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 succincts 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 :  $0.6 \times \text{<note DS>} + 0.4 \times \text{<note TP>}$ . 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 2 à 7. Concrètement : 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 compris. Cela donnera lieu à une première note de TP :

- si vous validez tous les exercices, vous aurez 17.

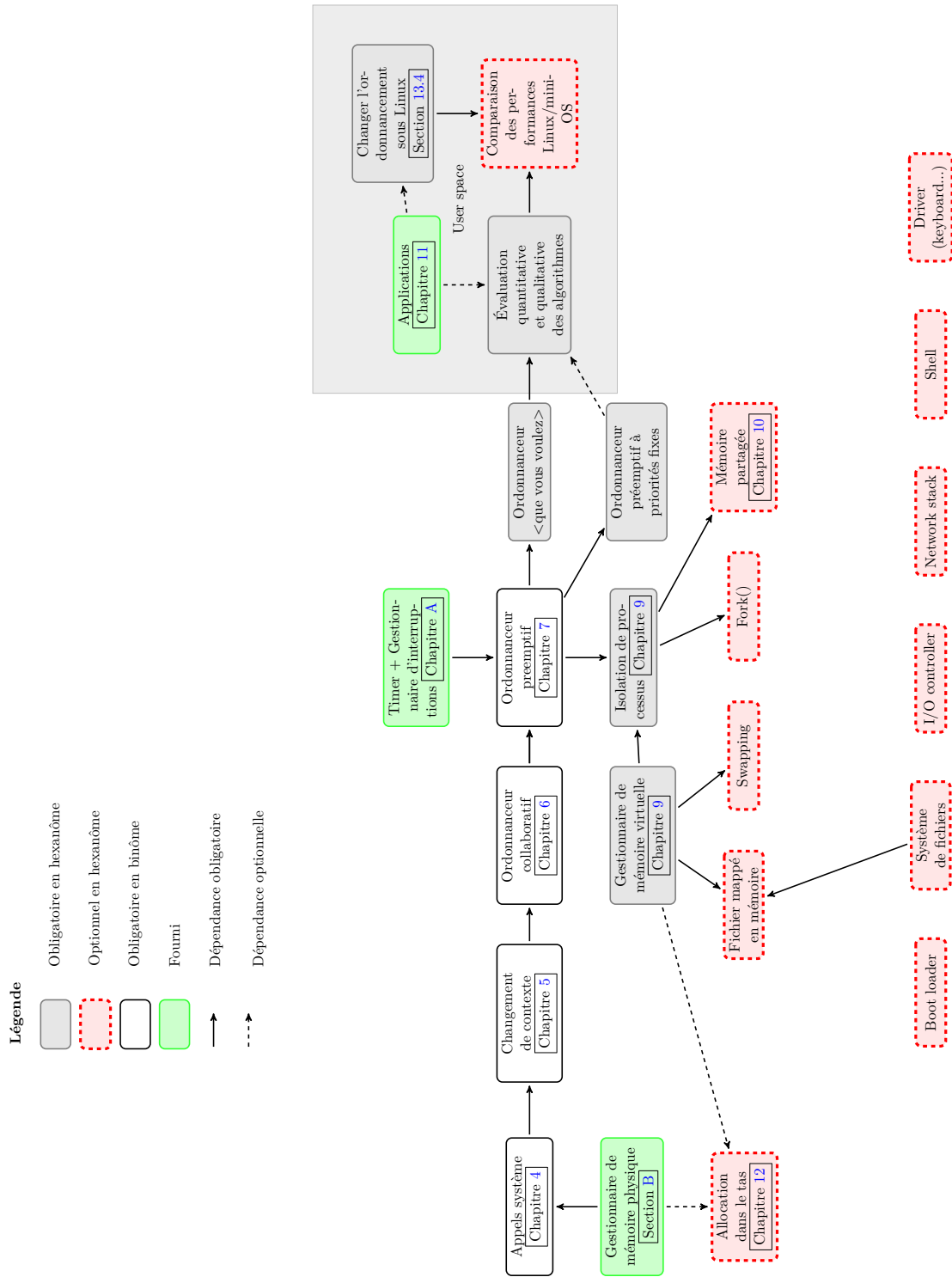


FIGURE 1.1 – DAG incomplet détaillant les travaux pratiques du semestre

- 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.

**Au terme de la deuxième période** un exposé présentant ce que vous avez fait et ce que vous avez compris. Voyez la Section 8.2 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 débogueur GDB et apprendrez à vous 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 déboguer 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]

## 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-compileur afin de produire de l'assembleur ARM pour la Raspberry Pi. Ce cross-compileur 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.

### **Encart 1: Rappelez-vous... la cross-compilation**

En 3if-Archi, vous utilisiez aussi un cross-compileur mais pour MSP430.

**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 7.

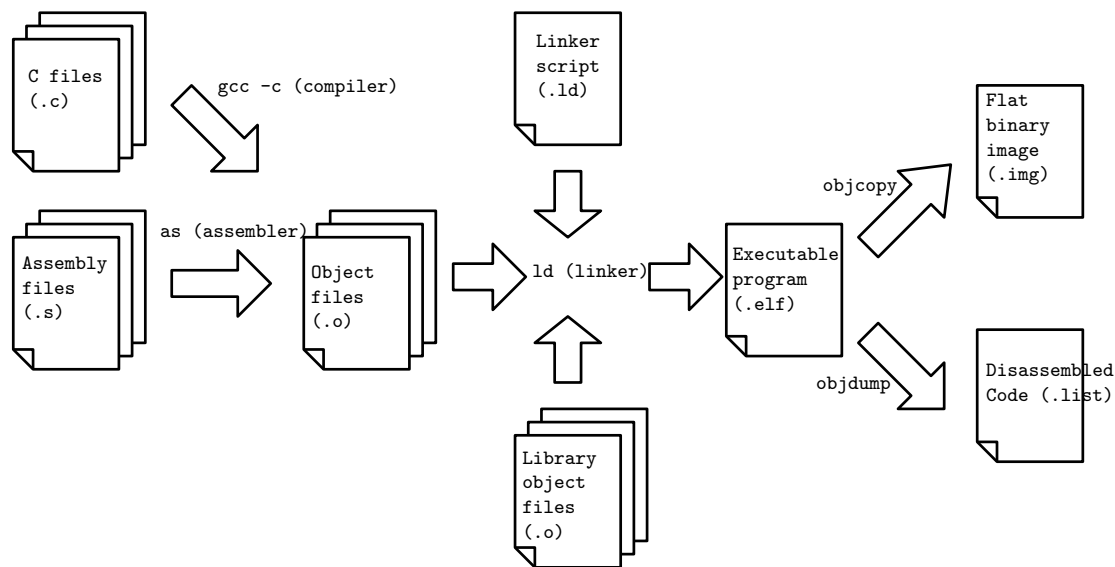


FIGURE 2.1 – Chaîne de compilation

## 2.3 Émulation de la plateforme

### Encart 2: Rappelez-vous... le débogage d'un program cross-compilé

En 3IF-Archi, on exécutait le programme pas à pas, sur la plateforme cible, à travers un connecteur JTAG, en utilisant mspdebug pour parler JTAG, et gdb par-dessus pour contrôler l'exécution du programme.

Vous avez lu l'encart 2 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 Raspberry Pi. Mais pas grave, on va voir une autre manière de développer. On ne vous a pas initié à GDB pour rien...

### Encart 3: 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.

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)

L'émulateur qu'on va utiliser (QEmu) est donc un logiciel capable d'interpréter, sur le PC, le code binaire ARM. Et pour pouvoir déboguer ce programme, l'émulateur va se laisser piloter par GDB, comme illustré par la figure 2.2. Des scripts pour lancer ces outils avec les bonnes options vous sont fournis dans l'archive de départ. Les quelques prochaines questions vous guident dans la prise en main de tout ça.

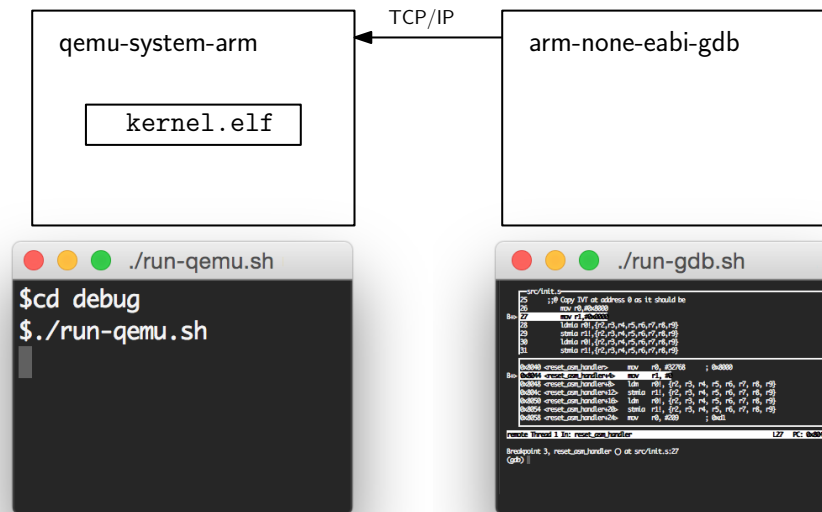


FIGURE 2.2 – Remote debugging : QEmu exécute notre programme en lui faisant croire qu’il s’exécute vraiment sur une Raspberry Pi, et obéit aux commandes que GDB lui envoie par le réseau. Vous aurez remarqué que la commande de débogage est `arm-none-eabi-gdb`, autrement dit, un `gdb` capable de lire et de comprendre l’assembleur ARM.

## Exercice : Prise en main des outils

**Question 2-1** Mettez en place votre code de la façon suivante :

1. Téléchargez l’archive `sysif.tgz` à l’adresse suivante :  
<http://kevinmarquet.net/teaching/operating-systems/systeme-dexploitation-avancees>
2. Décompressez-la : `tar xzf sysif.tgz` et placez-vous dans le répertoire créé : `cd sysif`
3. Créez dans ce répertoire un fichier `kmain.c` recopiez-y le code de la figure 2.3.
4. Le `Makefile` vous permet maintenant de compiler votre noyau — tapez `'make'`.
5. Placez-vous dans le répertoire `tools` et exécutez `'./run-qemu.sh'`, qui lance l’émulateur.
6. Dans un **autre** terminal, allez également dans `tools` et lancez `'./run-gdb.sh'`.
7. C’est depuis `gdb` que vous contrôlerez interactivement l’exécution de votre programme dans l’émulateur. Tapez `continue` (ou `c`) pour lancer l’exécution du code. Votre noyau s’initialise et s’arrête au début de `kmain()`. Exécutez le programme pas à pas avec la commande `next`.

**Question 2-2** Le `Makefile` que l’on vous fournit produit une version désassemblée de votre mini-OS dans le fichier `build/kernel.list`. Ouvrez ce fichier et notez les adresses suivantes :

- début et fin du code (section `.text`);
- début et fin des variables globales non-initialisées (section `.bss`);
- début des fonctions de votre fichier `kmain.c`.

## Exercice : observation des appels de fonction

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.

```
1 void
2 dummy()
3 {
4     return;
5 }
6
7 int
8 div(int dividend, int divisor)
9 {
10     int result = 0;
11     int remainder = dividend;
12
13     while (remainder >= divisor) {
14         result++;
15         remainder -= divisor;
16     }
17
18     return result;
19 }
20
21 int
22 compute_volume(int rad)
23 {
24     int rad3 = rad * rad * rad;
25
26     return div(4*355*rad3, 3*113);
27 }
28
29 int
30 kmain( void )
31 {
32     int radius = 5;
33     int volume;
34
35     dummy();
36     volume = compute_volume(radius);
37
38     return volume;
39 }
```

FIGURE 2.3 – Un premier bout de code pour observer

**Question 2-3** Lisez l’encart page suivante détaillant le processeur du Raspberry Pi puis exécutez le programme une instruction assembleur à la fois<sup>1</sup> en notant :

- Quels registres sont utilisés par la fonction `div()` ?
- Les valeurs de ces registres : `print/x $reg` pour afficher la valeur du registre `reg` (par exemple `r0`), ou `info registers` pour afficher la valeur de tous les registres.
- Quel registre le compilateur a-t’il décidé d’utiliser pour la variable `result` ?

---

1. vous utiliserez donc la commande `gdb stepi`, ou sa version courte `si`

**Encart 4: 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 :
  - Lors d'un saut ou d'un appel de fonction (typiquement grâce à l'instruction 'BL'), ce registre mémorise l'adresse de retour de la fonction,
  - Lorsqu'une interruption arrive, il mémorise l'adresse de l'instruction interrompue.
- 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...

**Exercice : appels de fonction**

Pour traduire en assembleur un morceau de code C qui appelle une procédure, 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** (pour *branch-and-link*). 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<sup>2</sup>.

**Question 2-4** Re-exécutez le programme de la Figure 2.3 pas à pas en observant les valeurs de **sp** au fil de cette exécution, et en comparant le contenu de la mémoire autour de cette adresse avec les adresses des premier et dernier paramètres de chaque fonction.

- Dans quel sens croît la pile : adresses croissantes ou décroissantes ?
- Quelle est la plus petite adresse mémoire que la pile atteint ?
- **sp** pointe-t'il sur la première case vide ou la dernière case pleine ?

**Question 2-5** Arrêtez l'exécution du programme à la ligne 18 (i.e. **return result;**) et dessinez le contenu de la pile d'exécution à ce moment. Pour cela, il vous faudra notamment déterminer la base de la pile, en regardant, par exemple, comment est initialisé **sp** avant l'appel à **kmain**.

**Question 2-6** Re-exécutez le programme pas à pas en observant les valeurs du registre **lr** avant et après chaque appel de fonction, ainsi qu'au retour de chacune grâce à la commande GDB **x**. Quelle instruction assembleur le met à jour implicitement ? Au besoin, relisez le rôle du registre **lr** dans l'encart de la présente page.

---

2. les plus curieux peuvent se documenter ici : [http://infocenter.arm.com/help/topic/com.arm.doc.ih0042e/IHI0042E\\_aapcs.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ih0042e/IHI0042E_aapcs.pdf)

## 2.4 Exercice : écrire (un peu) d'assembleur ARM

### Encart 5: Lier de l'assembleur et du code C

Le compilateur GCC permet de mélanger dans un même programme du code C du code assembleur, via la directive `__asm()`. De plus, on peut faire interagir les deux grâce à des *opérandes* d'entrée ou de sortie. Deux exemples :

- La ligne `__asm("mov %0, r0" : "=r"(X));` permet de copier le (contenu du) registre `r0` vers une variable nommée `X`;
- Le code suivant effectue l'opération inverse, à savoir lire la variable `X` (ou, plus généralement, évaluer l'expression `X`) et copier sa valeur dans le registre `r0` : `__asm("mov r0, %0" : : "r"(X));` Vous aurez remarqué que l'opérande d'entrée vient après un *second* caractère «deux points».

Attention à deux choses :

1. le compilateur est susceptible de traduire une même directive `__asm` en *plusieurs* instructions assembleur, et d'utiliser les registres qu'il veut. Pensez donc à bien regarder de près le code généré.
2. le compilateur traduit chaque directive `__asm` de manière complètement séparée, sans tenir compte des lignes voisines. Il peut donc y avoir des interférences inattendues.

En particulier, il se peut qu'un registre dont vous espériez maîtriser le contenu se retrouve écrasé. Dans ce cas, la solution est de spécifier dans chaque directive, après un troisième caractère «deux points» une liste de registres *réservés* que le compilateur n'aura alors pas le droit d'utiliser. Cette situation est illustrée dans l'exemple ci-dessous, qui cherche à copier le contenu des registres `r0` et `r1` dans deux variables `var0` et `var1`.

```
// probleme: la premiere ligne peut tres bien ecraser R1 au passage...
__asm("mov %0, r0" : "=r"(var0));
// ... et donc on n'est pas certain de retrouver la bonne valeur dans var1
__asm("mov %0, r1" : "=r"(var1));

// solution: indiquer quels registres sont precieux
__asm("mov %0, r0" : "=r"(var0) : : "r0", "r1");
__asm("mov %0, r1" : "=r"(var1) : : "r0", "r1");
```

Pour plus de détails sur la syntaxe de l'assembleur en ligne, reportez-vous à la documentation de GCC : <http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html>

Un autre moyen d'intégrer de l'assembleur avec du C est d'écrire des fichiers `.s` (par exemple notre `init.s`) et de les traduire en langage machine à l'aide d'un *assembleur* (e.g. GNU AS). On peut ensuite lier ensemble les fichiers objet obtenus grâce au *linker*. C'est d'ailleurs ce que fait notre Makefile.

Vous aurez à écrire quelques lignes d'assembleur pendant la suite du TP. Lisez l'encart 5 puis répondez aux deux questions ci-dessous.

**Question 2-7** Faites la manip suivante :

- au début de `kmain()`, faites sauter le processeur à l'adresse de la fonction `dummy()` grâce à l'instruction `b <label-ou-adresse-ou-nomfonction>;`
- exécutez ce programme instruction par instruction.

Votre programme retourne-il correctement de `dummy()` ? Pourquoi ? Corrigez pour que le retour s'effectue correctement.

**Question 2-8** Grâce à l'assembleur en ligne, vous pouvez avoir accès aux registres du processeur directement dans du code C. Écrivez :

- la ligne qui permet de mettre dans le registre `R2` le contenu de la variable `radius`.
- la ligne qui permet de mettre dans la variable `radius` le contenu du registre `R3`.

## 2.5 Exercice : l'attribut `naked`

### Encart 6: (Rappelez vous...) Les attributs de fonction

Certains compilateurs C comme GCC supportent, au sein de la déclaration d'une fonction ou d'une variable, la spécification d'un ou plusieurs **attribut**. Un attribut est une information donnée au compilateur pour l'aider à optimiser sa génération de code, ou lui indiquer de compiler le code d'une certaine manière. L'attribut qui nous intéresse ici est l'attribut `naked`. Une fonction déclarée comme telle sera compilée sans **prologue** ni **épilogue**. Dans une fonction classique, le code généré débute par un **prologue** qui sauvegarde certains registres sur la pile. Puis vient le **corps** de la fonction, qui correspond aux instructions du programme source. Finalement, le code de la fonction se termine par un **épilogue** qui restaure les registres sauvegardés. Dans une fonction `naked`, le compilateur ne génère ni prologue ni épilogue. S'il faut sauvegarder des registres, charge au programmeur de le faire.

Cette syntaxe est notamment utile pour déclarer un **traitant d'interruption**, dont on ne veut pas qu'il modifie la pile! Exemple : `void __attribute__((naked)) timer_handler();`

En 3if-archi, vous avez déjà utilisé ce genre de fonctionnalités, justement dans le TP sur les interruptions !

**Question 2-9** Lisez l'encart 6. Relevez les différences entre le code assembleur d'une fonction déclarée avec et sans cet attribut (par exemple en allant voir dans le fichier `kernel.list`).

## 2.6 Exercice : utiliser les tests fournis

Pour vous aider à valider votre code, nous vous fournissons au fil du projet différents programmes de test, dans le répertoire `sysif/test`. Dans cet exercice, vous allez prendre en main ce dispositif sur un exemple simple.

**Question 2-10** Compilez votre noyau avec la commande `make KMAIN=./test/kmain-bidule.c` (et ouvrez le fichier `.c` correspondant dans votre éditeur de texte.) Exécutez le programme en mode pas à pas dans `gdb`. Vérifiez que vous passez bien deux fois par la fonction `bidule()`.

**Question 2-11** Vous allez maintenant faire la même vérification, de façon non-interactive. Redémarrez `QEmu`, puis depuis le répertoire `tools`, tapez la commande `./run-gdb.sh ../test/bidule-called-twice.gdb`. Lisez le script `gdb` jusqu'à vous convaincre qu'il vérifie la même chose que la question précédente.

**Question 2-12** nous vous fournissons un outil pour automatiser toute cette manipulation : compilation, lancement de `Qemu`, et de `gdb` en mode non-interactif. Lisez le script `run-test.sh`, et exécutez-le avec les bons paramètres de ligne de commande.

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

## 2.7 Exercice : exécution sur la plateforme

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

Vous aurez besoin d'un Raspberry 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).

Si la carte SD dont vous disposez ne contient plus les fichiers initiaux (la distribution Linux Raspbian) ou si vous voulez réinstaller Linux dessus, voyez la page suivante : <http://www.raspberrypi.org/documentation/installation/installing-images/>

**Encart 7: 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 microcontrôleur. Ces quelques instructions lisent la carte SD, et chargent le *deuxième bootloader* stocké dans le fichier `bootcode.bin` dans le cache L2 ;
- L'exécution 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. Il est possible de changer cela facilement (on ne le fera pas pendant ce TP) : <http://www.raspberrypi.org/documentation/configuration/config-txt.md>

Le code qui se retrouve à l'adresse 0x8000 est celui du fichier `init.s` (car le compilateur cherche le label `_start`). Les premières lignes de ce fichier ne font que configurer la table des vecteurs d'interruption. C'est un peu compliqué car à chaque interruption correspond une instruction, pas une adresse à laquelle sauter comme sur d'autres architectures comme le MSP430. L'instruction pourrait être un simple `branch` mais dans le cas présent il faut copier ces instructions à partir de l'adresse 0 car c'est là que le CPU va chercher l'instruction à exécuter.

Sinon, plus d'infos sur le processus de boot :

- <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>

## 2.8 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/ARM_architecture)
- [http://en.wikipedia.org/wiki/List\\_of\\_ARM\\_microprocessor\\_cores](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 microcontrôleur 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](http://elinux.org/RPi_Hub)



## Chapitre 3

# Modes d'exécution du processeur

Un programme peut s'exécuter avec des *privilèges* plus ou moins grands. Au plus simple, on distingue ainsi les privilèges du code tournant dans *l'espace utilisateur* de ceux du code s'exécutant dans *l'espace noyau*. Concrètement, un processeur peut être configuré (via le registre de statut) pour s'exécuter en *mode* utilisateur ou en mode privilégié.

### 3.1 Les modes d'exécution ARM

Chez ARM, il existe un mode utilisateur – le mode **USER** – mais plusieurs mode privilégiés :

- le mode **System** dans lequel devra s'exécuter votre noyau.
- plusieurs modes pour gérer les exceptions :
  - le mode **SVC** (pour *Supervisor*). C'est le mode dans lequel se place le processeur lorsqu'il reçoit une interruption logicielle (causée par l'instruction **SWI** – *SoftWare Interrupt*)
  - le mode **Abort** dans lequel se place le CPU lorsque survient une erreur d'accès aux données (signalée par la MMU par exemple, vous verrez ça dans le chapitre 9).
  - les modes **IRQ** et **FIQ** (*a priori*, vous n'utiliserez pas ce dernier) dans lesquels se place automatiquement le CPU lorsqu'il reçoit une interruption du matériel.

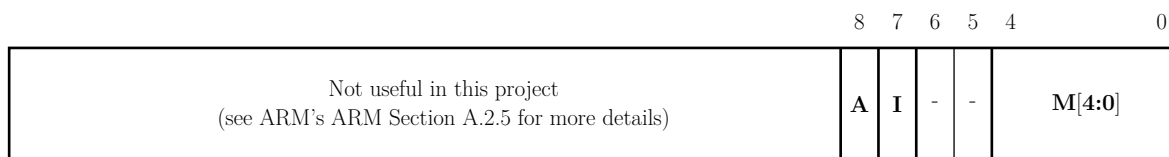


FIGURE 3.1 – Détail des registres de statut

Pour changer le mode d'exécution du processeur, il suffit de modifier les bits 0 à 4 du registre de statut, appelé **CPSR** (*Current Program Status Register*). Pour cela, deux moyens :

- l'instruction **cps #<num>** (*Change Processor State*) copie la valeur **num** dans les bits 0 à 4 de **CPSR**.
- l'instruction **msr <reg>** copie le contenu entier du registre **<reg>** dans le **CPSR**.
- pour lire le registre de statut, c'est l'instruction **mrs** qu'il faut utiliser. Exemple : **mrs r0, CPSR**.

La figure 3.1 détaille **CPSR**, l'utilité de chaque bit est donné ci-dessous :

**M[4:0]** sont les bits déterminant dans quel mode d'exécution se trouve le processeur. La table 3.2 indique quelle valeur correspond à quel mode d'exécution ;

**I** indique si les interruptions (*IRQ*) sont activées (valeur 0), or non (1) ;

**A** indique si les *data aborts* sont activées ou non. Ce ne sera utile qu'au chapitre 9.

Bits	Mode d'exécution
0b10000	User
0b10001	FIQ
0b10010	IRQ
0b10011	SVC (Supervisor)
0b10111	Abort
0b11111	System

TABLE 3.2 – Les bits de mode d'exécution du CPSR

Deux choses importantes, résumées par la figure 3.3 :

- chaque mode d'exécution dispose de son propre registre SP (**r13**) et son propre registre LR (**r14**). Les modes **User** et **System** partagent ces registres.
- chaque mode d'exécution privilégié, sauf **System**, est un mode dit *d'Exception*, et dispose d'un registre supplémentaire, **SPSR** (*Saved Program Status Register*).

Lorsque le processeur subit une exception (par exemple, une interruption), il passe du mode **User** ou **System** à un mode d'exception et fait deux choses :

1. il copie la valeur de **CPSR** dans le registre **SPSR** du mode d'exception
2. il copie la valeur du **PC** dans le registre **LR** du mode d'exception.

NB : si le processeur est en mode **System**, vous pouvez passer explicitement à un mode d'exception via l'instruction **cps**, ces 2 points ne sont pas réalisés.

## 3.2 Organisation de la mémoire physique

Maintenant que vous avez compris qu'il existe plusieurs modes d'exécution et que certains disposent d'un pointeur de pile à eux, jetez un oeil à la figure 3.4 qui illustre l'organisation initiale de la RAM du Raspberry Pi. Par exemple :

- À l'adresse 0, on trouve la table des vecteurs d'interruption
- À l'adresse 0x8000, le code de boot du noyau (fichier **init.s**);
- La pile svc commence à l'adresse du label **\_\_svc\_stack\_end\_\_**;
- Au-delà de l'adresse 0x20000000, l'espace d'adressage est dédié aux périphériques (rappelez-vous la 3IF et les registres des périphériques *mappés* en mémoire).

## 3.3 Exercice : changement de mode d'exécution





















**Question 3-1** Observez la valeur de **CPSR** au début de **kmain()**. Dans quel mode d'exécution se trouve le processeur ?

**Question 3-2** Modifiez votre programme pour qu'il passe le processeur en mode SVC au début de **kmain()**. Notez les valeurs de **CPSR** et **SP** avant et après ce changement de mode. Vérifiez que les valeurs sont les bonnes.

**Question 3-3** Notez la valeur de **LR** avant et après avoir passé le processeur dans le mode SVC au début de **kmain()**. Utilisez la commande GDB **x** ou regardez dans **kernel.list** pour en déduire où il pointe.

**Question 3-4** Que se passe-t-il si vous essayez de passer le processeur dans le mode **User** puis **SVC** ? Pourquoi ?

**Question 3-5** Que se passe-t-il si vous essayez d'accéder au registre **SPSR** au début de **kmain()** ? Pourquoi ?

Modes						
<div> <div>Privileged modes</div> <div>Exception modes</div> </div>						
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	 R8_fiq
R9	R9	R9	R9	R9	R9	 R9_fiq
R10	R10	R10	R10	R10	R10	 R10_fiq
R11	R11	R11	R11	R11	R11	 R11_fiq
R12	R12	R12	R12	R12	R12	 R12_fiq
SP	R13	 R13_svc	 R13_abt	 R13_und	 R13_irq	 R13_fiq
LR	R14	 R14_svc	 R14_abt	 R14_und	 R14_irq	 R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		 SPSR_svc	 SPSR_abt	 SPSR_und	 SPSR_irq	 SPSR_fiq


 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

FIGURE 3.3 – Les différents bancs de registres visibles dans chaque mode d'exécution. Figure extraite du manuel de référence ARM §A2.3 page 43 : allez lire les explications correspondantes.

**Question 3-6** Dans le fichier `utils.gdb`, nous vous fournissons une macro `print_sr` qui affiche sous une forme lisible le registre de statut. Elle vous dit par exemple le mode d'exécution actuel, la valeur des drapeaux et des bits de masquage d'interruption. Les bits à 0 sont affichés en minuscules et les bits à 1 sont en majuscule. Essayez cette macro à différents points d'exécution de votre noyau (et surtout pensez à l'utiliser par la suite).

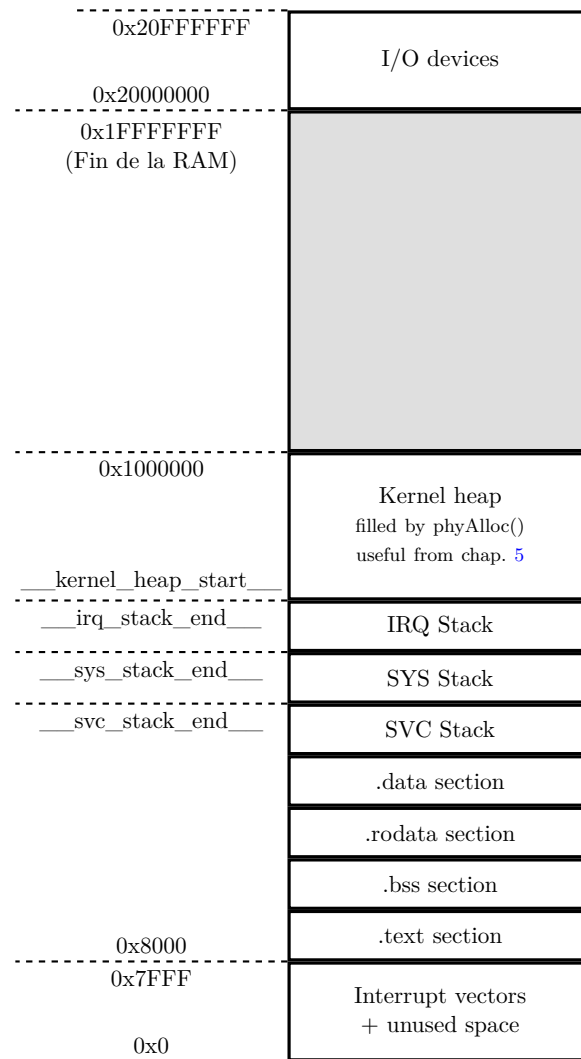


FIGURE 3.4 – Organisation de l'espace d'adressage physique initial. Les adresses (numériques ou symboliques) de début et fin de chaque section sont définies par le *linker script*, c'est à dire le fichier `sysif.ld`. Les sections `.bss`, `.text`, `.data`, `.rodata` sont construites par le compilateur.

# Chapitre 4

## Appels système

Un **appel système** (en anglais, *system call*, abrégé en *syscall*) est une fonction primitive fournie par le noyau du système d'exploitation, et qui rend un service nécessitant certains **privilèges** inaccessibles aux applications utilisateur. Par exemple sous Linux, pour ouvrir un fichier on doit invoquer l'appel système `open()` <sup>1</sup>.

Ce chapitre vous guide dans l'implémentation d'un mécanisme d'appels système.

### 4.1 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]

### 4.2 Appels système

Le principe, illustré par la figure 4.1, est relativement simple :

- Pour invoquer un appel système, l'application provoque une *interruption logicielle* <sup>2</sup>. Nous allons pour cela utiliser l'instruction `SWI`. Voyez donc l'encart 8 ;
- le noyau attrape cette interruption dans un traitant d'interruption, avec les privilèges noyau.

Les différents exercices qui suivent vous guident progressivement vers des implémentations de plus en plus complètes de ce principe. Tout le code de ce chapitre doit être placé dans les fichiers `syscall.c` et `syscall.h`.

---

1. en pratique, on utilise souvent des fonctions *wrapper* plus faciles d'usage, par exemple `fopen()` en C, ou les `fstream` en C++, qui se chargent d'interagir avec le noyau.

2. Comme on veut que le CPU passe dans un mode d'exécution privilégié, on ne peut pas simplement sauter dans le noyau par un appel de fonction classique.

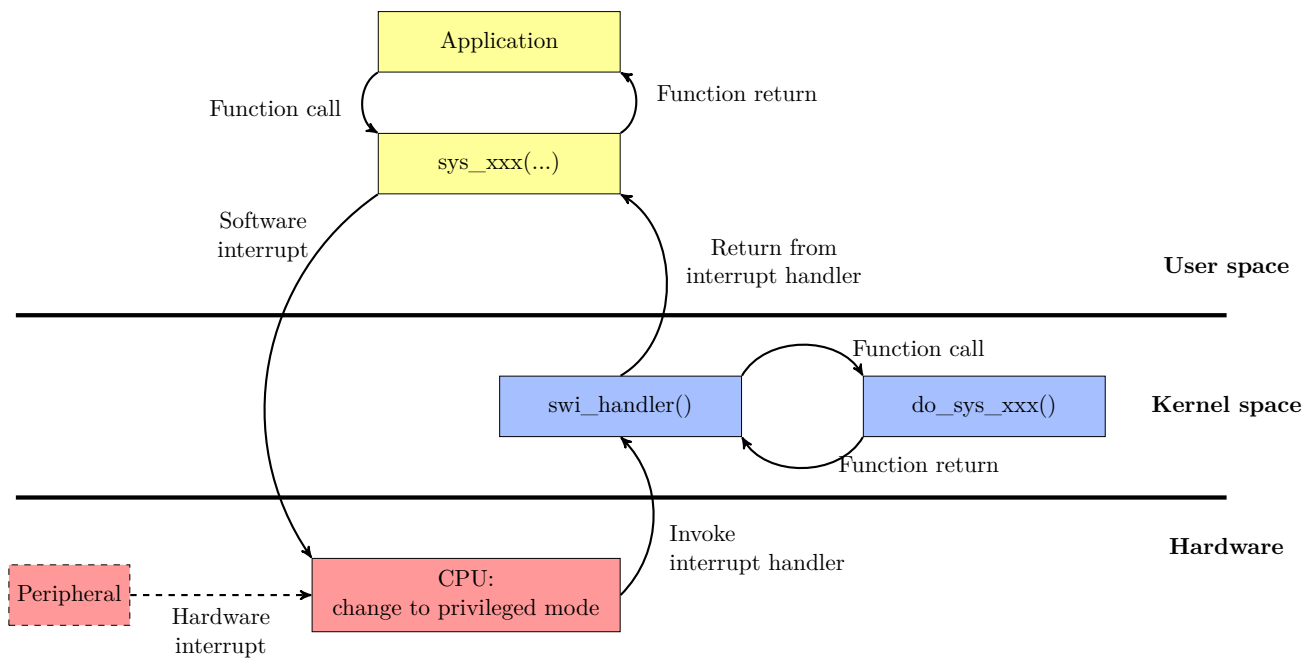


FIGURE 4.1 – Overview of a system call.

**Encart 8: L'instruction ARM SWI**

L'instruction SWI permet de demander l'exécution de code en mode Superviseur (SVC). Son effet est le suivant :

1. CPSR[4:0] := 0b10011, c'est à dire le processeur passe en mode SVC ;
2. L'adresse de l'instruction qui suit le SWI est copiée dans LR\_svc, i.e. le registre LR du mode SVC ; c'est à cette adresse-là qu'il faudra retourner après avoir traité l'appel système.
3. La valeur de CPSR\_user est copiée dans SPSR\_svc, i.e. le registre SPSR du mode SVC ;
4. PC reçoit l'adresse du traitant d'interruption correspondant aux interruptions logicielles

Pour plus de détails, reportez-vous au manuel de référence ARM §A2.6.4 page 58 (et aussi §A4.1.107 page 360).

### 4.3 Exercice : un premier appel système qui ne retourne pas

Vous allez commencer par écrire un mécanisme d'appel système simpliste, qui permet juste de demander au noyau d'exécuter d'une certaine fonction, sans lui passer de paramètres, ni récupérer de valeur de retour, car cette fonction ne retourne même pas. Plus précisément, il s'agit d'implémenter l'appel système ci-dessous :

Nom	Numéro	Fonction
<code>void sys_reboot()</code>	1	Redémarre le système

Le principe est le suivant :

- Côté utilisateur, le fonction positionne le numéro d'appel système dans le registre r0, puis fait une SWI.
- Côté noyau, le plus simple est de sauter dans une fonction C `swi_handler()` qui va :
  1. regarder le numéro de l'appel système.
  2. appeler la fonction permettant de traiter l'appel système correspondant à ce numéro.

## Nomenclature

- Vous nommerez vos fonctions côté utilisateur `sys_bidule`
- Vous nommerez chaque fonction traitant un appel en particulier `do_sys_bidule`

**Question 4-1** Déclarez et définissez la fonction `void sys_reboot()` qui implémente le côté utilisateur de l'appel système.

Notez au passage que l'instruction SWI requiert un opérande immédiat, mais nous n'allons pas nous en servir dans ce projet. Nous utiliserons donc toujours la syntaxe `SWI #0`.

**Question 4-2** Déclarez (dans `syscall.c`) une fonction `void swi_handler(void)`, complètement vide pour l'instant. Exécutez votre programme en mode pas-à-pas, et constatez qu'elle n'est jamais invoquée. Modifiez le code fourni pour qu'il saute dans votre fonction `swi_handler()` chaque fois que l'application fera SWI.

**Question 4-3** Implémentez la fonction `swi_handler()`, qui appelle `do_sys_reboot()` si et seulement si le numéro de l'appel système est 1. Pour cela :

- Afin de déboguer plus facilement, vous devriez mettre un point d'arrêt dans cette fonction. Pour cela, comprenez le script `run-gdb.sh` et modifiez le fichier `init.gdb`.
- Lisez l'encart 9 pour l'implémentation de `do_sys_reboot()`.
- si le numéro de l'appel système n'est pas 1, alors le noyau n'a rien d'intelligent à faire. Plutôt que d'ignorer silencieusement l'erreur, appelez la fonction `PANIC()` pour vous aider à déboguer. voir encart 10.

**Question 4-4** À ce stade, votre OS doit être capable d'exécuter correctement le programme suivant. Vérifiez en mode pas à pas que vous passez bien par `sys_reboot()`, puis par `swi_handler()`, puis par `do_sys_reboot()`, et que celle-ci redémarre effectivement le système : on finit donc par *revenir* à `kmain()`.

```
#include "syscall.h"
```

```
void kmain( void )
{
    // Note: kmain() starts with cpu in SYSTEM mode

    __asm("cps 0x10"); // switch CPU to USER mode
    // *****
    // Userland starts here

    sys_reboot();
}
```

**Question 4-5** Faites la même manip à l'aide de `run-test.sh` : votre noyau doit passer correctement le test `sys-reboot-does-reboot.gdb` sur `test/kmain-reboot.c`

**Encart 9: Redémarrage du microcontrôleur**

Redémarrer logiciellement le système entier (noyau + matériel) n'est pas si simple. Tant que vous travaillerez sur émulateur, vous pouvez vous contenter de sauter à l'adresse 0x8000 où se trouve le code d'initialisation mémoire qui vous est fourni. (pour les plus curieux, il s'agit de la routine `reset_asm_handler` dans `init.s`)

Mais sur Raspberry Pi, ce n'est pas très satisfaisant, car on ne sait pas dans quel état les périphériques seront laissés. Ce qu'on veut, c'est envoyer au CPU un signal matériel de `reset`. Pour cela, les instructions ci-dessous configurent le watchdog du microcontrôleur (Broadcom BCM2835) puis attendent que celui-ci envoie le signal `reset` au CPU.

```
const int PM_RSTC = 0x2010001c;
const int PM_WDOG = 0x20100024;
const int PM_PASSWORD = 0x5a000000;
const int PM_RSTC_WRCFG_FULL_RESET = 0x00000020;

PUT32(PM_WDOG, PM_PASSWORD | 1);
PUT32(PM_RSTC, PM_PASSWORD | PM_RSTC_WRCFG_FULL_RESET);
while(1);
```



**Encart 10: Programmation défensive dans le noyau : les macros PANIC() et ASSERT()**

Afin de vous faciliter la tâche de débogage, nous vous fournissons dans `util.c` et `util.h` quelques fonctions utilitaires classiques. Lorsque vous détectez une situation inattendue (par exemple, un appel système avec un numéro inconnu) c'est probablement signe qu'un bug s'est glissé dans votre code noyau ou dans l'application. Ça ne servirait donc à rien de continuer l'exécution, autant s'arrêter le plus tôt possible.

C'est exactement le rôle de la fonction `kernel_panic()`, qui n'est là que pour servir de sentinelle. L'idée est de mettre un point d'arrêt sur cette fonction (rajoutez le tout de suite dans votre `gdbinit`) et de l'invoquer lorsque les choses vont mal.

```
void kernel_panic(char* string, int number)
{
    for(;;)
    {
        // do nothing
    }
}
```

Vous aurez évidemment plusieurs endroits dans votre noyau susceptibles d'invoquer cette fonction. Pour pouvoir identifier individuellement chacun de ces endroits, nous vous fournissons également une macro `PANIC()`, qui invoquera cette fonction avec en paramètres le nom de fichier et le numéro de ligne courants :

```
#define PANIC() do { kernel_panic(__FILE__, __LINE__) ; } while(0)
```

Enfin, nous vous fournissons aussi une macro `ASSERT()` qui vérifie qu'une expression booléenne est bien vraie, et qui panique sinon :

```
#define ASSERT(exp) do { if(!(exp)) PANIC(); } while(0)
```

En pratique, vous n'aurez probablement pas besoin d'appeler vous-même `kernel_panic()` à la main, mais seulement les deux macros `PANIC()` et `ASSERT()`.

Vérifiez que vous avez compris comment utiliser ces deux macros en les invoquant sur des exemples simplistes :

```
13 kmain()
14 {
15     ...
16     ASSERT( 1+1 == 2); // will do nothing
17
18     ASSERT( 1+1 == 3); // will panic
19     ...
20 }
```

et assurez-vous que vous obtenez dans `gdb` un message de ce genre :

```
Breakpoint 5, kernel_panic (string=0x96d8 "src/kmain.c", number=18)
```

## 4.4 Exercice : un appel système qui retourne

Implémenter *Reboot* était facile : vous pouviez faire tous les crétinismes que vous vouliez côté noyau, le but étant de rebooter c'était pas bien grave. Les questions suivantes vous guident dans l'implémentation de l'appel système suivant :

Nom	Numéro	Fonction
<code>void sys_nop()</code>	2	Ne fait rien, puis retourne

Bien sûr, le code utilisateur qui appelle cette fonction doit ensuite continuer son exécution comme si de rien n'était.

**Question 4-6** Implémentez une version naïve de cette fonction, et vérifiez qu'elle est invoquée correctement.

**Question 4-7** Un problème qui se pose à ce stade, est de retourner au code utilisateur appelant. Pourquoi est-ce un problème dans votre version actuelle ? Exécutez votre programme en mode pas à pas pour bien comprendre ce qui se passe.

**Question 4-8** Pour régler ce problème, vous allez ajouter au début de `swi_handler()` une *sauvegarde* des registres utilisateur, et vous allez les *restaurer* lorsque vous aurez fini. Plus précisément, vous allez les empiler (sur la pile SVC) grâce à l'instruction `stmfd`, puis les dépiler avec `ldmfd`. Lisez l'encart 11, et pour plus de détails reportez-vous au manuel de référence ARM<sup>3</sup>. Pourquoi le registre `spsr` n'apparaît-il pas du côté droit de la figure illustrant l'instruction `ldmfd` ?

**Question 4-9** Réglez donc le problème identifié ci-dessus, en sauvegardant (resp. restaurant) le contexte utilisateur dans (resp. depuis) la pile SVC au début et à la fin de `swi_handler()`.

**Question 4-10** Est-il nécessaire de sauvegarder explicitement le registre de statut également ? Pourquoi ?

**Question 4-11** À ce stade, votre OS doit être capable d'exécuter correctement le programme suivant. Vérifiez dans GDB que vous revenez bien de `sys_nop`, puis que vous rebootez.

```
#include "syscall.h"

void kmain( void )
{
    // Note: kmain() starts with cpu in SYSTEM mode

    __asm("cps 0x10"); // switch CPU to USER mode
    // *****
    // Userland starts here

    sys_nop();

    // this must be reachable
    sys_reboot();
}
```

Cependant, il reste un dernier détail à régler pour que votre implémentation soit correcte.

**Question 4-12** Ajoutez une boucle infinie autour de l'appel à `sys_nop()`, et affichez (dans gdb) le registre `SP_svc` à chaque entrée dans `swi_handler()`. Que constatez-vous ? Étudiez le code machine de `swi_handler` de plus près (dans `kernel.list`) pour comprendre d'où vient ce problème, et réglez-le dans votre programme C.

**Question 4-13** Vérifiez avec `run-test.sh` et le programme `kmain-nop-reboot.c` que votre noyau passe bien les tests suivants :

- `sys-nop-does-return.gdb`
- `swi-handler-preserves-SP.gdb`

---

3. pour `stmfd` voir §A4.1.97 page 339, et pour `ldmfd` voir §A4.1.22 page 190. Vous trouverez aussi des explications utiles au §2.6, pages 54 et suivantes.

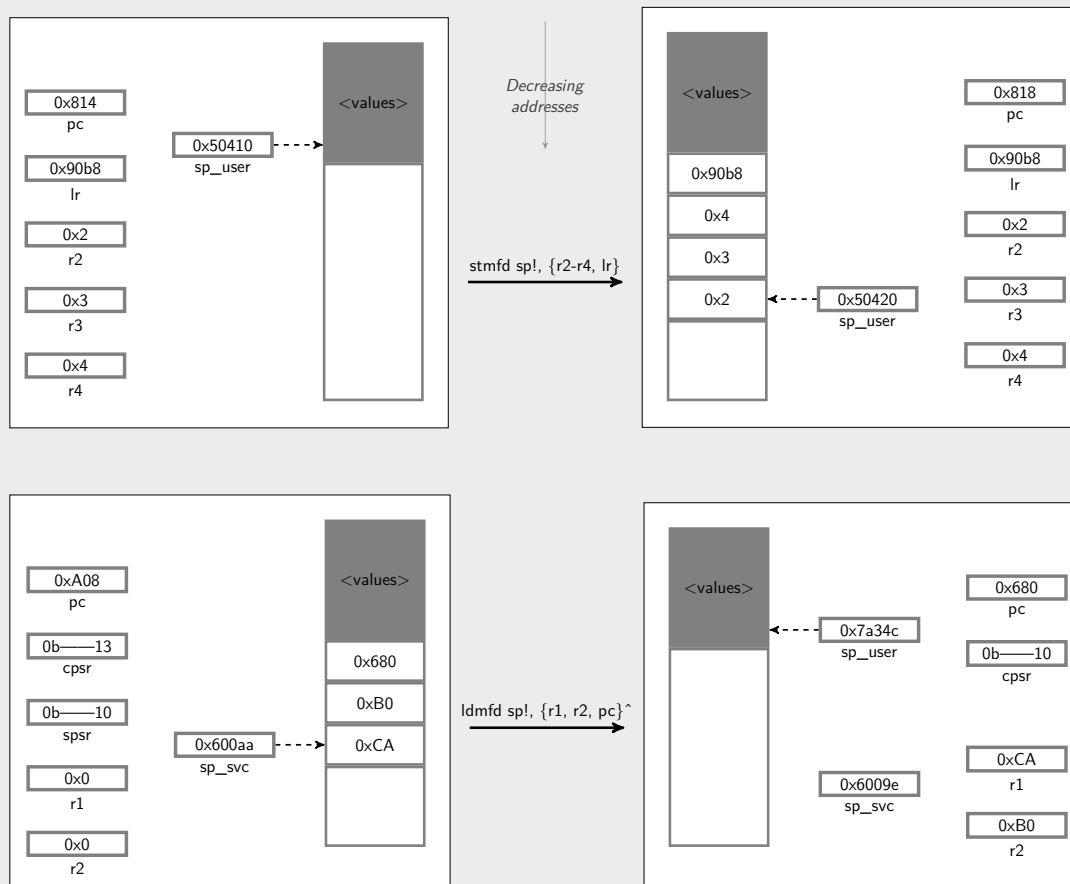
**Encart 11: Sauvegarde et restauration de contexte**

L'ensemble des informations caractérisant un point d'exécution est appelé contexte.

Dans le cas de votre mini-OS (comme dans bien des systèmes), un contexte contient l'état courant de la pile d'appel et des registres du processeur.

Donc par définition, mais sous réserve que la mémoire contenant la pile d'exécution ne soit pas modifiée, sauvegarder les registres suffit à mémoriser un contexte. En quelque sorte, on prend une photo de l'état du processeur à un moment donnée. À tout moment ultérieur, restaurer les valeurs de ces registres permettra de reprendre l'exécution du programme dont le contexte a été sauvegardé.

Il existe plusieurs manières différentes de sauvegarder et restaurer les registres. Utilisez, au moins dans une première version de votre OS, deux spécialisations des instructions STM (pour *STore Multiple*) et LDM (*LoaD Multiple*), illustrées par la figure ci-dessous. FD veut dire *full descending*, où *descending* signifie que la pile grandit lorsque SP diminue, et *full* signifie que SP pointe vers une case pleine. Attention, le rôle du caractère '^' est important : s'il est présent, l'instruction copie le contenu du registre SPSR dans CPSR. Et enfin, le '!' spécifie que le registre SP doit être mis à jour afin de pointer tout de suite après sur la dernière case pleine (pas besoin de décrémenter SP soi-même).



## 4.5 Exercice : un appel système avec passage de paramètres

Vous allez maintenant ajouter le support nécessaire pour que l'application puisse invoquer un appel système en lui passant des paramètres. Choisissons déjà une convention d'appel de part et d'autre de SWI. Comme le processeur va changer de mode, et donc de pile, le plus simple est de passer les arguments via les registres :

- Côté utilisateur, chaque fonction `sys_bidule()` va placer les valeurs des arguments dans des registres généraux, puis faire SWI.
- Côté noyau, `swi_handler()` commence par empiler tous les registres (sur la pile SVC), puis elle mémorise l'emplacement du sommet de pile, et le transmet à la méthode `do_sys_bidule`, qui l'utilisera pour

retrouver les arguments en mémoire.

Ainsi, votre noyau pourra à sa guise consulter les valeurs des registres utilisateur (en allant voir dans la pile) tout en disposant quand même des registres CPU pour s'exécuter.

Une telle convention d'appel est appelée une *Application Binary Interface*, ou *ABI*. Selon Wikipedia fr :

*Une ABI décrit une interface bas niveau entre les applications et le système d'exploitation, entre une application et une bibliothèque ou bien entre différentes parties d'une application. Une ABI diffère d'une API, puisqu'une API définit une interface entre du code source et une bibliothèque, de façon à assurer que le code source fonctionnera (compilera, si applicable) sur tout système supportant cette API.*

**Question 4-14** Implémentez l'appel système suivant, avec pour l'instant une implémentation côté noyau complètement vide.

Nom	Numéro	Fonction
<code>void sys_settime(uint64_t date_ms)</code>	3	Sets system date, in milliseconds

**Question 4-15** Invoquez `sys_settime()` avec des arguments facilement reconnaissables, et dessinez l'état de la pile SVC au début de la fonction `do_sys_settime()`.

Il serait donc tentant d'aller piocher directement nos paramètres en sommet de pile. Mais cette solution naïve ne va pas marcher à tous les coups.

**Question 4-16** Rajoutez quelques variables locales dans votre fonction `do_sys_settime()`, initialisées à des valeurs facilement reconnaissables, et aussi quelques calculs inutiles entre ces variables. Affichez de nouveau l'état de la pile en début de fonction. Que concluez-vous ?

**Question 4-17** Corrigez votre fonction `swi_handler()` en conséquence, et terminez l'implémentation de `do_sys_settime()`. Une fois que vous avez correctement reconstruit l'entier `date_ms`, vous pouvez le passer à la primitive `set_date_ms()` de `hw.h/hw.c` (allez voir son implémentation au passage).

**Question 4-18** À ce stade, votre OS doit être capable d'exécuter correctement un programme qui appelle `sys_settime()`. Vérifiez en mode pas à pas que votre paramètre est bien passé par les registres R1 et R2, puis sauvegardé dans la pile, puis reconstruit dans `do_sys_settime()`, puis finalement passé en argument à `set_date_ms()`.

**Question 4-19** Vérifiez avec `run-test.sh` et le programme `kmain-settime.c` que votre noyau passe bien le test `sys-settime-passes-argument.gdb`

## 4.6 Exercice : un appel système qui retourne une valeur

Last but not least, implémentons un appel système qui retourne une valeur. De nouveau, il faut choisir une convention pour le passage de la valeur de retour. Je vous propose le principe suivant : de la même façon que les paramètres sont passés aux fonctions `do_sys_bidule` par la pile SVC, ces fonctions vont modifier les valeurs contenues dans la pile, de telle sorte que la restauration de contexte survenant à la fin de `handle_swi()` placera dans `r0` la bonne valeur de retour. Si la valeur de retour est sur plus de 32 bits, utilisez aussi le registre `r1`.

La fonction `sys_bidule` pourra lire ces registres pour reconstituer la valeur à retourner à l'application.

**Question 4-20** Implémentez l'appel système suivant :

Nom	Numéro	Fonction
<code>uint64_t sys_gettime()</code>	4	Return system date, in milliseconds.

**Question 4-21** Vérifiez avec `run-test.sh` et le programme `kmain-gettime.c` que votre noyau passe bien le test `sys-gettime-returns-value.gdb`

# Chapitre 5

## Process dispatching

Jusqu'ici, on n'a eu affaire qu'à du code séquentiel, c'est à dire avec un seul fil d'exécution. Sur une vraie machine, on veut typiquement exécuter plusieurs programmes à la fois, donc gérer *plusieurs* fils d'exécution distincts. Dans ce chapitre ainsi que dans les deux chapitre suivants, vous allez implémenter les mécanismes nécessaires au partage du temps processeur entre plusieurs fils d'exécution.

### 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)	[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]

### 5.2 Introduction

Dans les systèmes d'exploitation, les hyperviseurs, les machines virtuelles etc., le passage d'un fil d'exécution à l'autre est effectué par un composant appelé le **dispatcher**. C'est celui-ci que vous allez implémenter dans ce chapitre. Tout le code de ce chapitre doit être placé dans les fichiers `sched.c` et `sched.h`.

### 5.3 Exercice : un dispatcher pour coroutines

Un système d'exploitation permet à plusieurs fils d'exécution (appelons-les dès maintenant des **processus**) s'exécuter de façon concurrente. L'objectif de cet exercice est d'implanter un tel mécanisme de partage du CPU pour deux processus collaboratifs. Chaque processus pourra *rendre la main*, permettant ainsi au système d'exploitation d'allouer la ressource CPU à l'autre processus. Pour cela, l'OS maintient, pour chaque processus, une structure de données appelée **Process Control Block (PCB)**, ce qui lui permet de sauvegarder le **contexte d'exécution** et l'**état d'un processus** quand celui-ci n'est pas en cours d'exécution.

À la fin de ce chapitre, votre noyau sera capable d'exécuter des applications de ce genre :

```
void user_process_1()
{
    int v1=5;
    while(1)
    {
        v1++;
        sys_yieldto(p2);
    }
}

void user_process_2()
{
    int v2=-12;
    while(1)
    {
        v2-=2;
        sys_yieldto(p1);
    }
}
```

Vous allez donc devoir implémenter, entre autres, l'appel système suivant :

Nom	Numéro	Fonction
<code>void sys_yieldto(struct pcb_s* dest)</code>	5	Passer la main à un autre processus

Vous aurez remarqué que ce paramètre **dest** est d'un type que vous n'avez pas encore défini. Les questions qui suivent sont justement là pour vous y amener progressivement. Le principe de l'implémentation est relativement simple : en réalité, le travail est déjà fait en grande partie !

En effet, la première instruction de votre **swi\_handler()** consiste à empiler la plupart des registres utilisateur. Si vous ne les touchez pas, alors ce seront les mêmes registres qui seront dépilés en sortant de **swi\_handler()**, et on retournera donc dans le même processus. C'est comme ça qu'on avait implémenté les appels système. Mais si, entre temps, votre noyau a remplacé les valeurs dans la pile par des valeurs décrivant à un *autre* contexte, alors c'est cet autre contexte qui va reprendre son exécution ! Vous aurez donc réalisé tout simplement un *changement de contexte*.

**Question 5-1** Commencez par déclarer un type **struct pcb\_s** avec aucun champ, et écrire les fonctions **sys\_yieldto()** et **do\_sys\_yieldto()**. Pour l'instant, cette dernière récupère son paramètre **dest** mais ne fait rien.

**Question 5-2** Ajoutez à votre **struct pcb\_s** les champs nécessaires pour contenir une copie de tous les registres sauvegardés au début de **swi\_handler()**. Ajoutez à **do\_sys\_yieldto()** les instructions nécessaires pour faire effectivement ce changement de contexte. Comme le processus appelant nous indique vers qui il veut sauter, on sait ce qu'il faut charger dans la pile. Par contre, pour savoir où recopier le contexte du processus appelant, vous aurez besoin de déclarer dans **sched.c** une variable globale **struct pcb\_s \*current\_process**, et de la faire pointer en permanence sur le processus en cours d'exécution.

**Question 5-3** Mais, tel quel, ça ne peut pas marcher : lors du premier appel à `sys_yieldto()`, votre noyau va sauvegarder le contexte de `kmain()` vers le pointeur `current_process`, qui n'est pas initialisé ! Déclarez donc, toujours dans `sched.c`, une `struct pcb_s kmain_process` qui servira exactement à ça. Vous aurez aussi besoin d'une fonction `sched_init()` pour faire pointer `current_process` sur cette structure.

Vous avez maintenant les moyens de sauter d'un processus à l'autre, mais à condition que les deux processus soient déjà lancés ! En effet, pour l'instant le tout premier appel `sys_yieldto(p1)` ; dans `kmain` ne peut pas marcher, puisque `p1` n'est pas initialisé. Du coup, la première fois qu'il sera rechargé dans le CPU, les registres seront fantaisistes. Pour les registres de données, ce n'est pas très grave : le nouveau processus se débrouillera très bien pour initialiser ses variables au début de son exécution. Par contre, il est important d'initialiser correctement le champ LR du PCB, pour qu'il soit restauré correctement (dans le PC) en sortie de `swi_handler()` et qu'on saute ainsi au bon endroit.

Recopiez dans `kmain.c` les deux fonctions `user_process_N` données en début de chapitre, ainsi que le code suivant :

```
#include "util.h"
#include "syscall.h"
#include "sched.h"

struct pcb_s pcb1, pcb2;

struct pcb_s *p1, *p2;

void user_process_1 and _2 ...

void kmain( void )
{
    sched_init();

    p1=&pcb1;
    p2=&pcb2;

    // initialize p1 and p2
    // [your code goes yere]

    __asm("cps 0x10"); // switch CPU to USER mode
    // *****

    sys_yieldto(p1);

    // this is now unreachable
    PANIC();
}
```

**Question 5-4** Remplacez le `[your code goes here]` par du code qui initialise le registre LR de chaque PCB avec l'adresse de la bonne fonction.

**Question 5-5** À ce stade, ça marche presque ! Exécutez votre programme en mode pas à pas, et observez le flot de contrôle passer successivement de `kmain`, à `user_process_1`, à `user_process_2`, à `user_process_2`, à `user_process_2...`

Le problème, c'est qu'en réalité on n'a pas sauvegardé tous les registres ! Rappelez vous : à l'exercice 4.5, on avait empilé tous les registres pour pouvoir travailler les mains libres. Mais le registre LR qu'on sauvegarde ainsi au début de `swi_handler` est `LR_svc`, qui contient l'adresse de retour d'exception. Le registre `LR_user`, lui, n'est pas directement visible depuis le mode SVC. Pour des appels système «simples», on n'avait pas besoin



de s'en préoccuper, puisqu'on retournait toujours vers le même processus. Mais maintenant qu'on passe d'un processus à l'autre, on va être obligé de sauvegarder/restaurer également `LR_user`.

**Question 5-6** Distinguez dans votre PCB les deux champs `LR_svc` et `LR_user`, et ajoutez à votre changement de contexte la sauvegarde et restauration de `LR_user`.

**Question 5-7** Exécutez le programme en pas à pas, et vérifiez que maintenant le flot de contrôle passe bien `kmain`, à `user_process_1`, puis à `user_process_2`, retour à `user_process_1`, puis `user_process_2` et ainsi de suite indéfiniment.

## 5.4 Exercice : des coroutines avec des variables locales

**Question 5-8** Dans `gdb`, observez l'évolution des variables locales de vos deux processus. Que constatez-vous ? Étudiez le code machine de vos deux fonctions pour comprendre ce qui se passe.

Le problème, c'est que pour l'instant on n'a toujours pas réellement des processus indépendants : certes leurs registres cpu sont maintenant distincts, et sont correctement sauvegardés/restaurés, mais ils partagent tous la même *pile d'exécution* ! Chaque processus y range soigneusement ses variables, donc si la même pile est partagée par tous les processus, alors ils vont chacun écraser les variables des voisins, et tout le monde va calculer n'importe quoi.

La solution à ce problème est d'allouer une pile d'exécution distincte à chaque processus, et de sauvegarder/restaurer également le pointeur de pile lors du changement de contexte. C'est ce que vous allez faire dans cet exercice.

**Question 5-9** On va maintenant attribuer à chaque processus non seulement un PCB mais une pile d'exécution individuelle. Pour ne pas répéter inutilement du code, vous allez tout d'abord ajouter à `sched.c` une fonction `create_process` qui va se charger d'allouer dynamiquement (voir à ce sujet l'encart 12) un `struct pcb_s`, de l'initialiser, et de le retourner à l'appelant.

Plus précisément, la signature de cette fonction sera `struct pcb_s* create_process(func_t* entry)`, prenant en paramètre un pointeur de fonction particulier et retournant un pointeur de pcb. Pour les processus utilisateur, vous définirez dans `sched.h` votre type `func_t` de la manière suivante : `typedef int (func_t) (void)` ; Autrement dit, un processus utilisateur est une fonction sans paramètre, et retournant un entier.

Vous pouvez maintenant supprimer vos deux variables `pcb1` et `pcb2`, et ré-écrire votre fonction `kmain()` comme ci-dessous. Remarque : pour l'instant nos processus ne retournent rien, et ne peuvent même pas se terminer du tout. On est donc obligé de faire une conversion explicite de types (en VO un *type cast*) pour que le compilateur accepte notre programme. On aurait aussi pu déclarer `func_t` autrement, et économiser le cast. Cependant, au chapitre suivant vous allez implémenter la terminaison de processus, et le renvoi d'une valeur de retour. Donc autant déclarer dès maintenant `func_t` avec sa forme définitive.

```

void kmain( void )
{
    sched_init();

    p1=create_process((funct_t*) &user_process_1);
    p2=create_process((funct_t*) &user_process_2);

    __asm("cps 0x10"); // switch CPU to USER mode
    // *****

    sys_yieldto(p1);

    // this is now unreachable
    PANIC();
}

```

### Encart 12: Gestion de la mémoire

À partir d'ici, vous aurez besoin d'allouer dynamiquement de la mémoire pour y ranger toutes vos structures de données (pile d'exécution, PCB...). Cependant, vous n'avez pas encore écrit le gestionnaire mémoire de votre système d'exploitation. Nous vous fournissons pour l'instant un allocateur très simple qui gère des blocs de mémoire physique. Pour allouer de la mémoire, utilisez la fonction

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

Cette fonction alloue `size` octets de données, et renvoie un pointeur sur le début de la zone allouée. Pour libérer cette zone, utilisez

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

Ces deux fonctions sont déclarées dans `kheap.h` et définies dans `kheap.c`. Avant de pouvoir les utiliser, il faut initialiser le gestionnaire mémoire en appelant `kheap_init()`.

**Question 5-10** Si vous n'avez pas lu l'encart 12, lisez-le, puis initialisez la mémoire dans votre fonction `sched_init()` (c'est important que ce soit là car les tests supposent ça).

**Question 5-11** Allouez à chaque processus une pile d'exécution de 10Ko, ce sera largement suffisant pour ce projet<sup>1</sup>. Ajoutez à votre PCB un champ `sp` pour y stocker le pointeur de pile du processus. Initialement, la pile de chaque processus ne contient aucune donnée. À quelle adresse devez-vous donc faire pointer ce champ `sp` ?

**Question 5-12** Comme pour `LR_user`, il ne vous reste plus qu'à sauvegarder/restaurer le registre `SP_user` vers le bon `PCB->sp` lors des changements de contexte.

À ce stade, votre programme doit maintenant s'exécuter correctement. Vérifiez en mode pas à pas que vous sautez bien indéfiniment d'un processus à l'autre, et que leurs variables locales sont correctement préservées de part et d'autre des changements de contexte : au cours de l'exécution, `v1` est de plus en plus grande, et `v2` de plus en plus négative.

**Question 5-13** En toute rigueur, il reste un dernier détail à régler pour que votre implémentation soit vraiment complète. À chaque changement de contexte, il faut également sauvegarder le registre d'état du processus, c'est à dire `CPSR_user`, qui a été copié pour vous par le CPU dans `SPSR_svc` au moment du SWI. Ajoutez à votre changement de contexte la sauvegarde et restauration de ce registre.

1. Si on se dit que dans chaque fonction, on a 5 variables locales + 5 arguments + 10 mots pour calculer, on a besoin à la louche de 80 octets par stack frame. Si on imbrique une centaine d'appels, on a besoin de 8Ko pour toute la pile.

**Question 5-14** Vérifiez avec `run-test.sh` et le programme `kmain-yieldto.c` que votre noyau passe bien les tests suivants :

- `sys-yieldto-jumps-to-dest.gdb`
- `sys-yieldto-preserves-locals.gdb`
- `sys-yieldto-preserves-status-register.gdb`

**Question 5-15** Dans un OS classique, le noyau ne permet pas à une application de choisir à qui elle passe la main. On trouve donc typiquement un appel système `yield()` sans paramètre, mais pas d'appel `yield_to(dest)`. Pourquoi ? Listez au moins deux bonnes raisons de ne pas offrir cette fonctionnalité.

## Chapitre 6

# Ordonnancement collaboratif (round-robin)

Lorsque de nombreux processus s'exécutent sur la machine, ce ne serait pas sérieux de les laisser choisir individuellement à qui chacun passe la main. Au contraire, c'est un des rôles importants du système d'exploitation que de choisir intelligemment, lors de chaque changement de contexte, quel processus va être exécuté ensuite. Le composant qui se charge de cette décision s'appelle l'**ordonnanceur**. Dans ce chapitre, vous allez implémenter un ordonnanceur simple.

### 6.1 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]

## 6.2 Ordonnancement

L'ordonnanceur a besoin d'information sur les processus. Comme on l'a vu précédemment, les informations dont l'OS a besoin concernant chaque processus sont regroupées dans un PCB (Process Control Block). Mais dans le cas général, un nombre  $N$  de processus peuvent co-s'exécuter. L'**ordonnanceur** a pour rôle de :

- maintenir une ou plusieurs structures de données contenant les PCB. Plusieurs structures de données peuvent être utilisées pour ce faire ; vous allez dans ce chapitre utiliser une liste chaînée de PCB.
- choisir, quand il faut (pour l'instant, quand un processus rend la main), le processus à exécuter parmi les  $N$ . On parle de l'**élection** d'un processus. Moults algorithmes existent ayant différentes propriétés (équité, famine, efficacité, etc.). Dans ce chapitre vous allez simplement implémenter un ordonnanceur **round-robin** : le processus élu sera le processus suivant, dans la liste chaînée, celui qui s'exécutait jusque là.

## 6.3 Exercice : ordonnancement round-robin

L'objectif de cet exercice est d'implémenter la liste de PCB et la fonction d'élection round-robin de l'ordonnanceur. Concrètement, à la fin de l'exercice, votre noyau sera capable d'exécuter le programme `test/kmain-yield.c`, reproduit ci-dessous page 38.

**Question 6-1** Une solution simple pour organiser les PCB consiste à les chaîner directement entre eux, sans ajout d'une structure supplémentaire. Proposez une modification de la `struct pcb_s` pour ce faire.

**Question 6-2** Implémentez l'ordonnanceur round-robin. Entre autres, vous devrez :

- modifier `create_process()` qui doit maintenant ajouter le PCB nouvellement créé à la liste chaînée ; vous pouvez continuer à le retourner aussi, mais la fonction `kmain()` n'est plus censée avoir accès aux PCB.
- implémenter une fonction `void elect()`, qui choisit le prochain processus et fait pointer la variable globale `current_process` sur son PCB.
- ajouter un nouvel appel système `sys_yield()` sans paramètre, qui permet à une application de rendre la main.
- penser à chaîner `kmain_process` avec les autres processus, ou pas.

**Question 6-3** À ce stade, votre OS doit être capable d'exécuter correctement le genre de programme ci-dessous. Vérifiez à l'aide de `gdb` que vous passez bien par chacun des processus à son tour, et que les différentes variables augmentent au bon rythme.

```

#include "util.h"
#include "syscall.h"
#include "sched.h"

#define NB_PROCESS 5

void user_process()
{
    int v=0;
    for(;;)
    {
        v++;
        sys_yield();
    }
}

void kmain( void )
{
    sched_init();

    int i;
    for(i=0;i<NB_PROCESS;i++)
    {
        create_process(&user_process);
    }

    __asm("cps 0x10"); // switch CPU to USER mode
    // *****

    while(1)
    {
        sys_yield();
    }
}

```

**Question 6-4** Vérifiez avec `run-test.sh` et le programme `kmain-yield.c` que votre noyau passe correctement les test suivants :

- `round-robin-distinct-stacks.gdb`
- `round-robin-fairness.gdb`

## 6.4 Exercice : 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 6-5** Vous pourriez être tentés d'attendre que la fonction principale d'un processus se termine pour gérer la terminaison de ce processus et désallouer les structures de données associées à ce processus. Pourquoi n'est-ce pas possible ?

Une solution simple consiste pour gérer la terminaison d'un processus consiste à faire un appel système à la fin du processus, dans le code USER. Vous allez donc implémenter l'appel système suivant :

Nom	Numéro	Fonction
<code>void sys_exit(int status)</code>	7	Termine le processus courant avec le code de retour <code>status</code> .

Quelques subtilités nous obligent à vous guider dans sa réalisation côté noyau (fonction `do_sys_exit()` si vous n'êtes pas passé à côté des consignes de nomenclature), via les quelques questions ci-dessous. C'est pas pour être sympa, c'est parce que sinon vous allez tous nous poser les mêmes questions et on n'aime pas répéter les mêmes trucs 15 fois.

**Question 6-6** Comment pouvez-vous, dans `do_sys_exit()` supprimer de la liste chaînée le PCB du processus qui vient de se terminer ? Que pensez-vous de l'efficacité de cette opération dans le cas où un grand nombre de processus s'exécutent ? Est-ce gênant ?

**Question 6-7** Les deux solutions suivantes fonctionnent, vous êtes libres d'implémenter celle que vous voulez :

- Marquer le processus comme `TERMINATED` et gérer sa terminaison dans la fonction `select()`. Dans ce cas il vous faut ajouter la notion d'état dans les PCB, et modifier `select()` afin de supprimer un éventuel processus `TERMINATED` rencontré lors du parcours de la liste chaînée ;
- Double-chaîner la liste de PCB. Ça ne vous empêche pas d'ajouter l'état d'un processus dans la structure de PCB.

**Question 6-8** Modifiez votre `kmain.c` pour que les processus utilisateurs appellent `sys_exit()` au bout d'un moment, et vérifiez en mode pas à pas que votre OS se comporte correctement.

```
void user_process()
{
    int v=0;
    while(v<5)
    {
        v++;
        sys_yield();
    }

    sys_exit();
}
```

**Question 6-9** Modifiez la structure de PCB de manière à stocker le code de retour. Pour récupérer ce code de retour côté noyau, souvenez-vous de l'appel système `sys_gettime()` qui retourne une valeur.

**Question 6-10** Lorsque l'ordonnanceur n'a plus aucun processus à exécuter, il n'y a plus qu'à appeler la fonction `terminate_kernel()` puisque pour l'instant, rien ne peut conduire à la création d'un processus à part le processus lui-même.

## 6.5 Exercice facultatif : terminaison non explicite d'un processus

Obliger le programmeur d'application à appeler `sys_exit()` est un problème : en cas d'oubli, le système tout entier devient inutilisable. Bon, on est bien d'accord que le mauvais comportement d'un processus peut de toute façon rendre inutilisable le système en entier puisque votre ordonnancement est collaboratif pour l'instant. Mais la remarque vaut pour votre futur ordonnanceur préemptif.

Le démarrage d'un processus ne doit plus se faire par simple saut vers son `pcb->entry()`, mais vers une nouvelle fonction `start_current_process()`. Cette fonction commence par appeler la fonction `pcb->entry()` puis, au retour de celle-ci, effectue l'appel système `sys_exit()`, de manière transparente pour le processus utilisateur.

# Chapitre 7

## Ordonnanceur préemptif

L'ordonnanceur que vous avez programmé jusqu'ici permet un partage du processeur dit *collaboratif*, c'est à dire que le noyau attend que chaque processus rende volontairement la main. Même si certains OS utilisent cette approche, la plupart implémentent plutôt un ordonnancement dit *préemptif*, c'est à dire que le noyau reprend la main *de force* au bout d'un certain temps. C'est ce que vous allez implémenter dans ce chapitre.

### 7.1 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]

## 7.2 Ordonnancement sur interruption

Votre ordonnanceur sera capable, à la fin de ce chapitre, d'interrompre le processus en cours d'exécution afin de passer la main à un autre processus. Pour cela, vous allez utiliser un *timer* matériel et lui faire envoyer au processeur des *requêtes d'interruption* à intervalles réguliers. Une requête d'interruption est un événement assez similaire à un SWI : elle provoque un changement de mode d'exécution, et fait sauter le processeur vers certaine fonction du noyau (un **traitant d'interruption** ou *interrupt handler*) dans laquelle vous ferez le changement de contexte. Pour plus de détails sur les interruptions, lisez l'encart 13, et reportez-vous au manuel de référence ARM §2.6.8 page 62.

Afin de vous faire gagner du temps, et d'éviter les redondances inutiles avec vos TP d'Architecture de l'année dernière, nous vous fournissons les quelques primitives ci-dessous pour interagir avec le matériel (allez les voir de plus près dans `hw.c/hw.h`). Attention : elles ne fonctionnent correctement que si le processeur est dans un mode d'exécution privilégié !

```
void timer_init();
void set_next_tick_default();
void ENABLE_TIMER_IRQ();
void DISABLE_TIMER_IRQ();
void DISABLE_IRQ();
void ENABLE_IRQ();
```

La fonction `timer_init()` effectue plusieurs tâches d'initialisation du timer matériel. Elle fait aussi appel à deux primitives que vous devrez invoquer vous aussi à chaque fois que vous voudrez ré-armer le timer : `set_next_tick_default()` qui configure l'expiration du timer dans 10ms, et `ENABLE_TIMER_IRQ()` pour que le timer envoie au cpu une requête d'interruption quand il arrivera à échéance. Le vecteur d'interruption est configuré pour sauter au label `irq_asm_handler` dans le fichier `init.s`. Comme pour SWI, vous devrez aller modifier le code assembleur pour qu'il saute vers une fonction `irq_handler` qui sera écrite en C.

Les deux primitives `DISABLE_IRQ()` et `ENABLE_IRQ()` permettent de configurer le comportement du processeur vis-à-vis des requêtes d'interruption (grâce au bit I du registre d'état<sup>1</sup>). Au démarrage du système, les interruptions sont désactivées.

1. cf manuel de référence ARM §2.5 page 49

**Encart 13: Interruption matérielle**

Petit rappel d'architecture : le processeur exécute des cycles fetch-decode-execute, en vérifiant avant d'exécuter chaque instruction si une interruption n'est pas survenue. Si une interruption est en attente, le processeur saute vers le vecteur d'interruption correspondant.

Dans notre cas, le fonctionnement du processeur ARM vis-à-vis des interruptions matérielles est globalement similaire à ce qu'on a déjà vu pour les exceptions logicielles (encart 8 page 22). Lorsqu'il reçoit le signal IRQ, le CPU fait plusieurs choses :

1. sauvegarder CPSR en le copiant dans SPSR\_irq
2. passer en mode IRQ ;
3. désactiver les interruptions (en passant à zéro le bit I du CPSR)
4. sauvegarder PC dans LR\_irq (voir ci-dessous).
5. charger dans PC l'adresse du traitant d'interruption

Attention, remarque importante : au moment où le traitant d'interruption est appelé, le pipeline du processeur fait que PC contient déjà l'adresse de l'instruction suivante. Il faut donc décrémenter LR\_irq de 4 pour récupérer la «bonne» adresse de retour, sinon on va «sauter» une instruction à chaque fois.

Pour plus de détails, reportez-vous au manuel de référence ARM §A2.6.8 page 62.

**Exercice : Prise en main des interruptions**

**Question 7-1** Modifiez votre code pour activer les interruptions.

**Question 7-2** Faites en sorte que, sur une interruption du timer, le processeur saute vers une fonction `handle_irq()`, qui ne fait rien pour l'instant.

**Question 7-3** Implémentez `irq_handler()` pour qu'elle ré-arme le timer, et qu'elle retourne correctement au code interrompu. Vérifiez en mode pas à pas que vous revenez bien au bon endroit, et que vous restaurez correctement le mode d'exécution.

**Exercice : Ordonnanceur préemptif**

**Question 7-4** Quels registres ont une version matérielle différente dans le mode d'exécution IRQ que dans le mode SVC ? Servez-vous de la figure 3.3.

**Question 7-5** Avec cette information en tête, modifiez votre `irq_handler` pour ne plus simplement revenir dans le processus interrompu, mais pour faire au passage un changement de contexte. Deux manières de procéder :

1. Le plus simple est de procéder comme dans `swi_handler` pour la sauvegarde/restauration des registres, et comme dans `yield` pour l'échange des contextes sur la pile. Le pointeur de pile sera celui du mode d'exécution IRQ mais cela ne change rien au mécanisme. Pour cette solution, dotez-vous de deux fonctions remplissant le même rôle que `void context_save_to_pcb()` et `void context_load_from_pcb()` mais compatible avec le mode IRQ plutôt que SVC.
2. Vous pouvez aussi vous débrouiller pour éviter le copier-coller et invoquer votre code existant (`do_sys_yield()`), à condition de vous mettre dans le bon mode d'exécution, et de correctement sauvegarder/restaurer tous les registres du programme utilisateur.

À ce stade, votre OS doit être capable d'exécuter correctement le programme suivant. Vérifiez avec `gdb` que vous passez bien par les trois processus utilisateur à tour de rôle, et indéfiniment. Vous pouvez pour ça mettre des points d'arrêt dans chacune des trois boucles infinies, puis vous aider de la commande `continue N` pour ignorer les  $N$  prochains passages sur un même point d'arrêt.

```
void user_process_1()
{
    int v1=5;
    while(1)
    {
        v1++;
    }
}

void user_process_2()
{
    int v2=-12;
    while(1)
    {
        v2-=2;
    }
}

void user_process_3()
{
    int v3=0;
    while(1)
    {
        v3+=5;
    }
}

void kmain( void )
{
    sched_init();

    create_process(&user_process_1);
    create_process(&user_process_2);
    create_process(&user_process_3);

    timer_init();
    ENABLE_IRQ();

    __asm("cps 0x10"); // switch CPU to USER mode
    // *****

    while(1)
    {
        sys_yield();
    }
}
```

## Deuxième partie

**Semaines 45 à 52 : projet (en hexanômes)**

# Chapitre 8

## Présentation du projet

### 8.1 Objectifs

L'objectif général de ce projet est d'implémenter certaines fonctionnalités des systèmes d'exploitation et en acquérir les concepts. Vous devez vous organiser en hexanôme pour faire la démonstration lors du rendu final que :

- Vous avez implémenté et compris les modules en gris de la figure 1.1.
- Vous avez implémenté et compris une fonctionnalité supplémentaire. :

Pour le deuxième point, vous pouvez vous inspirer d'une ou plusieurs boîtes rouges de la figure 1.1. Attention, elles sont pour la plupart assez "volumineuses". Vous pouvez aussi vous inspirer des fonctionnalités suivantes :

- un mécanisme `fork()` / `join()`
- un tas par processus (`malloc()` / `free()` quoi)
- un chargeur de binaire en format elf (plus dur)
- un driver clavier. Vous pouvez récupérer le code de l'université de Cambridge et juste le repackager pour qu'il compile avec votre noyau. Attention, il ne fonctionne pas avec tous les claviers, et pas ceux de la salle 219
- un driver pour un périphérique quelconque (on a des caméras par exemple, mais je ne sais pas à quel point c'est compliqué)
- une version 64 bits de votre OS, fonctionnant sur Raspberry Pi B+ (on en a)
- une version multi-cœur de votre OS sur Raspberry Pi B+ (on en a). Mais attention, le point précédent est un pré-requis!
- une optimisation des perfs, chiffrée bien sur. Par exemple :
  - montrer qu'on peut diminuer le temps du *context switch*, en se servant des instructions `srs/rfe`, ou en évitant de sauvegarder des registres inutiles.
  - montrer qu'on peut diminuer le temps du *context switch*, en évitant d'invalider complètement la TLB à chaque appel système. Plusieurs moyens pour cela :
    - se servir de la *TrustZone*, voir Section 9.9
    - verrouiller certaines entrées de la TLB, voir la doc du processeur Section 3.2.31.
- se servir du DMA pour optimiser les copies mémoire. Voir Section 7.4 de la doc processeur.
- se servir de l'unité de calcul flottant pour optimiser... euh... les calculs flottants. Voir Chapitres 20 et 21 de la doc processeur.
- mettre au point un lien JTAG pour programmer/debugger, ce qui permettrait d'éviter de devoir enlever/-remettre la carte SD chaque fois qu'on veut exécuter un truc. Cf. <http://sysprogs.com/VisualKernel/tutorials/raspberry/jtagsetup/>
- ...

Mais dans tous les cas, vous devez valider ce choix auprès de vos profs de TP, histoire que vous ne vous lanciez pas dans une implémentation trop complexe.

Vous ferez la démonstration de ce que vous avez fait et appris lors de la dernière séance. Voyez la section 8.2 pour les détails du rendu. N'hésitez pas à réaliser des parties complémentaires d'autres hexanômes et à mettre le travail en commun.

## 8.2 Déroulement et rendu

Le projet dure 7 séances ; la dernière sera entièrement consacrée à l'évaluation. Cette évaluation prendra la forme d'un exposé oral pendant lequel vous présenterez votre projet à l'aide d'au moins quelques scénarios de démonstration. En plus de ces démonstrations, vous pouvez utiliser ce que vous voudrez : quelques slides, des vidéos, le tableau et la craie... bref ce que vous voulez pour montrer les compétences acquises et les travaux d'implémentation réalisés. Attention, durant cette dernière séance, vous devrez assister aux présentations des autres hexanômes, aucun travail ne pourra être fait avant de présenter ce jour là.

Votre rendu ce jour là durera une heure, devant l'ensemble de votre groupe de projet. Profs et étudiant auront tout loisir pour poser des questions sur les démos, les réalisations, la compréhension.

Pour les groupes comprenant 5 hexanômes au moins, l'évaluation durera jusque au moins 13h. Si vous avez une impossibilité (*e.g.* cours de langue), merci de prévenir à l'avance.

## 8.3 Organisation du travail au sein de l'hexanôme

Vous vous organisez comme vous voulez mais deux choses peuvent vous aider :

- lire la page suivante sur les pièges à éviter :

<http://www.cs.cornell.edu/Courses/cs4410/2014fa/howtolose.php>

- vous rappeler que toutes les notions et compétences associées aux boîtes grises doivent être comprises par tout le monde, y compris ceux qui ne travaillent pas directement sur l'implémentation proprement dite. Passer une heure par semaine à vous parler sur l'état d'avancement de chacun, les principes, les choix réalisés, l'interfaçage de chaque partie du code avec le reste etc. est un minimum pour espérer comprendre le projet dans son ensemble.

## Chapitre 9

# Gestion de la mémoire virtuelle : pagination et protection

Ce chapitre vous guide dans l'implémentation d'un gestionnaire de mémoire physique fournissant à la fois un mécanisme de pagination de la mémoire, ainsi qu'un mécanisme d'isolation des processus. Attention, ces deux concepts sont orthogonaux :

- on peut isoler des processus les uns des autres sans utiliser de mécanisme de pagination ;
- on peut paginer la mémoire sans isoler les processus entre eux.

Mais sur les processeurs ARM, il est très difficile de dissocier les deux, vous verrez pourquoi.

Les premières sections de ce chapitre donnent des détails sur le mécanisme de pagination et comment gérer cela sur le processeur du Raspberry Pi. VOus avez donc un peu de lecture avant de commencer les exos.

### 9.1 Rappels sur la mémoire virtuelle

La figure 9.1 illustre comment, dans beaucoup de systèmes informatiques, une adresse manipulée par le processeur est traduite pour accéder à l'endroit de la mémoire où est réellement stocké l'information recherchée.

L'adresse manipulée par le processeur est appelée *adresse virtuelle*. Elle est traduite grâce à la MMU (*Memory Management Unit*) en une adresse physique. Pour ce faire la MMU effectue une recherche dans la table des pages, située en RAM. Afin de ne pas payer le coût (en temps) nécessaire à la recherche dans la table des pages, la MMU utilise un cache de traduction : la TLB (*Translation Lookaside Buffer*). Ce composant est composé principalement d'une mémoire associative (et n'est donc pas en RAM).

### 9.2 Le coprocesseur

Le microprocesseur ARM1176JZF-S présent dans le Raspberry Pi possède un *co-processeur*, de son petit nom CP15 (pour *Control Processor*), permettant de contrôler certaines fonctionnalités, parmi lesquelles le système de cache, le DMA, la gestion des performances et, ce qui nous intéresse ici : la MMU.

Le contrôle de ces composants s'effectue grâce à l'écriture dans les registres du coprocesseur, nommés *c0* à *c15*. Mais attention, il existe plusieurs versions matérielles de chacun de ces registres. Dans la suite, on se réfère à la version matérielle par la nomenclature suivante : `<numéro de registre>/<nom du registre>`, *e.g.* `c2/TTBRO`.

Ces registres sont accessibles à l'aide des deux instructions suivantes :

- `MCR{cond} P15, <Opcode_1>, <Rd>, <CRn>, <CRm>, <Opcode_2>` Écrire un registre
- `MRC{cond} P15, <Opcode_1>, <Rd>, <CRn>, <CRm>, <Opcode_2>` Lire un registre

Le champ `Rd`, spécifie le registre du processeur (r0 à r12 donc) impliqué dans le transfert. Le champ `CRn` spécifie le *numéro* du registre du coprocesseur auquel on veut accéder, par exemple `c0`. Les champs `CRm` et

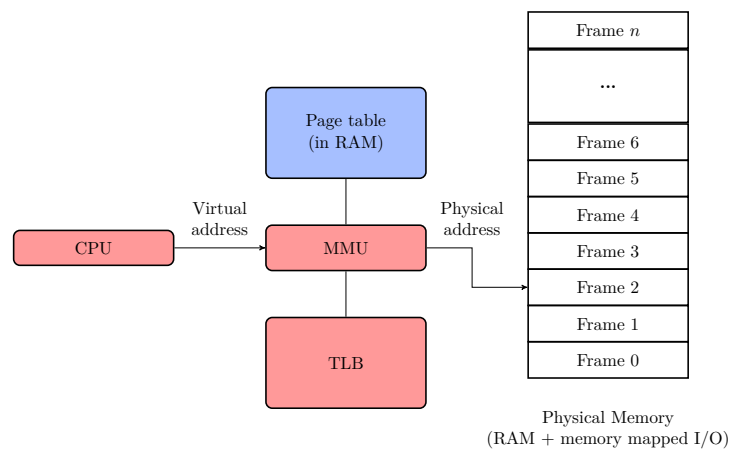


FIGURE 9.1 – Schéma simpliste de la traduction d’une adresse virtuelle en une adresse physique. La table des pages est en RAM (donc dans une des *frames*...), est remplie par l’OS, mais la traduction est un mécanisme matériel.

`Opcode_2` indique l’opération que l’on souhaite réaliser ainsi que le registre matériel. Par exemple, l’instruction `mcr p15, 0, r0, c2, c0, 0` spécifie d’écrire la valeur contenue dans `r0` dans le registre `c2/TTBR0`.

Le contrôle de la MMU s’effectue à l’aide d’un registre de 32 bits accessible en lecture seulement, un registre accessible en écriture seulement, et 22 registres de 32 bits accessibles en lecture/écriture. Les registres du coprocesseur auxquels vous aurez besoin d’accéder sont détaillés dans la suite du sujet.

## 9.3 La MMU

### 9.3.1 Activation de la MMU

Les instructions nécessaires pour configurer et activer la MMU vous sont données figure 9.2. Ce code est expliqué ci-dessous.

L’activation et la désactivation de la MMU se font via le positionnement d’un bit (le bit M, numéro 0) du registre de contrôle du coprocesseur (`c1/Control register`). Cependant, pour activer la MMU, la documentation ARM indique qu’il est nécessaire de respecter la séquence suivante :

1. Configurer les registres du coprocesseur
2. Configurer les descripteurs de table des pages (voir 9.3.2)
3. Désactiver et invalider le cache d’instruction. Il est ensuite possible de le réactiver quand on active la MMU. Je vous donne le code pour ça, il s’agit de la ligne : `mcr p15, 0, r0, c7, c7, 0` du fichier `init.s`.
4. Activer la MMU en positionnant le bit M du registre `c1/Control Register`.

### 9.3.2 Configuration des descripteurs de table des pages

Le processeur contient deux registres pointant vers 2 tables de pages, `TTBR0` et `TTBR1`, ainsi qu’un registre de contrôle, `TTBCTR`. L’espace d’adressage est divisé en 2 régions :

- les adresse comprises entre  $1 \ll (32 - N)$  sont traduites par la table des pages pointées par `TTBR0` ;
- les adresses comprises entre  $1 \ll (32 - N)$  et 4GB sont traduites par la tables des pages pointée par `TTBR1` ;

$N$  est configuré via `c2/TTBCR`. Si  $N=0$  (ce sera votre cas), alors tout l’espace d’adressage est géré via `c2/TTBR0`. Sinon, l’espace d’adressage réservé à l’OS et aux I/O est situé dans la partie haute de l’espace d’adressage (via `c2/TTBR1` donc) et les tâches dans la partie basse (`TTBR0`).



```

void
start_mmu_C()
{
    register unsigned int control;

    __asm("mcr p15, 0, %[zero], c1, c0, 0" : : [zero] "r"(0)); //Disable cache
    __asm("mcr p15, 0, r0, c7, c7, 0"); //Invalidate cache (data and instructions) */
    __asm("mcr p15, 0, r0, c8, c7, 0"); //Invalidate TLB entries

    /* Enable ARMv6 MMU features (disable sub-page AP) */
    control = (1<<23) | (1 << 15) | (1 << 4) | 1;
    /* Invalidate the translation lookaside buffer (TLB) */
    __asm volatile("mcr p15, 0, %[data], c8, c7, 0" : : [data] "r" (0));
    /* Write control register */
    __asm volatile("mcr p15, 0, %[control], c1, c0, 0" : : [control] "r" (control));
}

void
configure_mmu_C()
{
    register unsigned int pt_addr = MMUTABLEBASE;
    total++;

    /* Translation table 0 */
    __asm volatile("mcr p15, 0, %[addr], c2, c0, 0" : : [addr] "r" (pt_addr));

    /* Translation table 1 */
    __asm volatile("mcr p15, 0, %[addr], c2, c0, 1" : : [addr] "r" (pt_addr));

    /* Use translation table 0 for everything */
    __asm volatile("mcr p15, 0, %[n], c2, c0, 2" : : [n] "r" (0));

    /* Set Domain 0 ACL to "Manager", not enforcing memory permissions
     * Every mapped section/page is in domain 0
     */
    __asm volatile("mcr p15, 0, %[r], c3, c0, 0" : : [r] "r" (0x3));
}

```

FIGURE 9.2 – Les deux fonctions permettant de configurer et activer la MMU

En cas de *TLB miss*, les bits de poids fort de l'adresse logique sont utilisés pour décider si TTBR0 ou TTBR1 est utilisé pour traduire cette adresse.

Les 3 registres TTBR0, TTBR1, TTBCR, existent en 2 exemplaires matériels (ce sont des *banked registers*). L'une ou l'autre version est utilisée selon que le micro-contrôleur s'exécute dans l'état **Secure** ou **Non-secure**.

## 9.4 Pagination

La traduction des adresses peut se faire, sur notre processeur, en mode ARMv5 ou ARMv6. L'implémentation qui vous est fournie fonctionne en mode ARMv6. Ceci est configuré via le bit XP (23) du registre c1/control register.

Dans le processeur ARM des Raspberry Pi, c'est la MMU qui se charge de remplir la TLB avec des paires <Adresse logique, Adresse physique>. Contrairement à un processeur Intel, l'OS ne peut pas remplir directement la TLB avec les paires qu'il veut. Lorsqu'une adresse logique n'est pas trouvée dans la TLB, la MMU parcourt la table des pages afin de trouver l'adresse physique correspondante. Cette table des pages est située

en mémoire, elle est remplie par le système d'exploitation, et permet la traduction d'une adresse logique en une adresse physique selon le mécanisme décrit dans la figure 9.3. Attention l'implémentation ne gère ni sections ni supersections, uniquement des pages (de taille 4Ko).

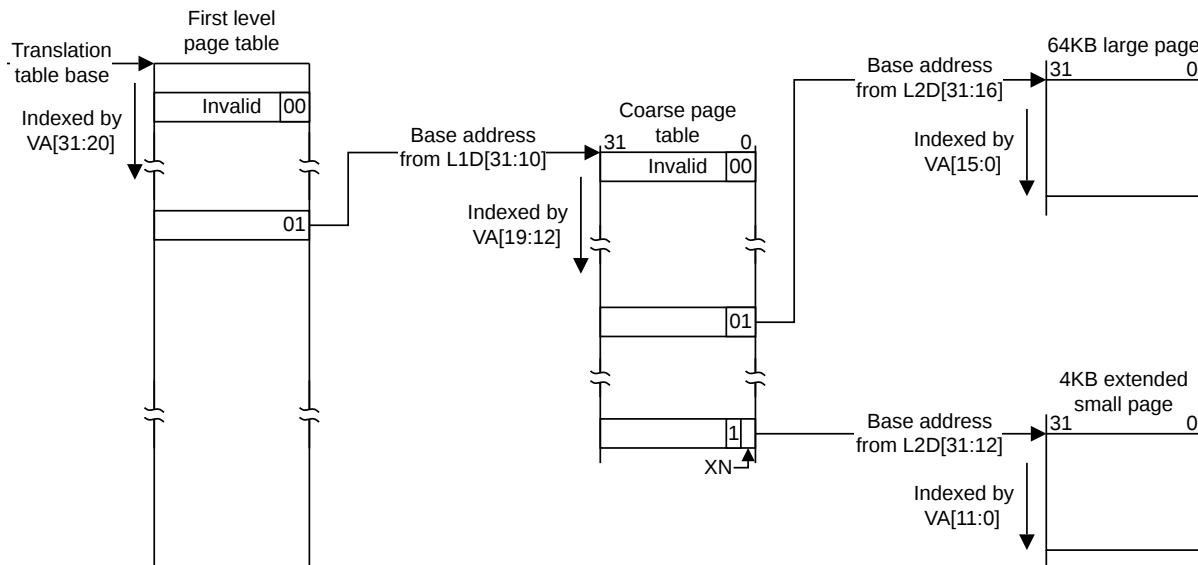


FIGURE 9.3 – Vue générale de la traduction d'adresse en mode ARMv6

La traduction d'adresse effectuée est récapitulée, de manière précise, par la figure 9.4

#### 9.4.1 Table des pages de niveau 1

Le format d'une entrée de la table de niveau 1 (une PDE) est détaillé dans la figure 9.5. En fonction des deux bits de poids faibles, l'entrée prend la forme d'une des deux premières lignes (dans votre implémentation, seules ces 2 lignes sont valides, on ne gère pas de section et supersection).

Les bits de la seule ligne ayant du sens ont le rôle suivant :

**P** Pas supporté dans notre processeur (section 6.11.1 de la doc processeur).

**Domain** la MMU effectue un contrôle d'accès à gros grain : ces bits indiquent un *domaine* auquel appartient la zone mémoire. Le registre c3 du coprocesseur associe des droits à chacun de ces domaines. Initialisez à 0 pour que toutes les pages appartiennent au même domaine. Plus d'infos en Section 6.5.1 de la doc sur le processeur.

**SBZ** "Should Be Zero" (gestion de compatibilité ascendante).

**NS** Bit relatif à l'extension *TrustZone* des ARMv6. Laissez à 0 ("Secure"). Plus d'infos section 6.6.3 de la doc sur le processeur.

#### 9.4.2 Tables des pages de niveaux 2

Le format d'une entrée d'une table de niveau 2 est détaillé dans la figure 9.7. Encore une fois, en fonction des deux bits de poids faibles, l'entrée prend la forme d'une des ligne mais l'implémentation fournie ne gère pas les pages de taille de 64KB. Les bits de la seule ligne ayant du sens ont le rôle suivant :

**nG (Not-Global)** détermine si cette entrée est globale, c'est à dire valide pour tous les processus, ou si elle est valide pour un seul processus. Initialisez à 0 tant que vous n'êtes pas à l'isolation entre processus.

**S (Shared)** détermine si l'entrée concerne une page partagée ou non.

**XN (eXecute-Never)** détermine si la région contient du code excutable (0) ou non (1).

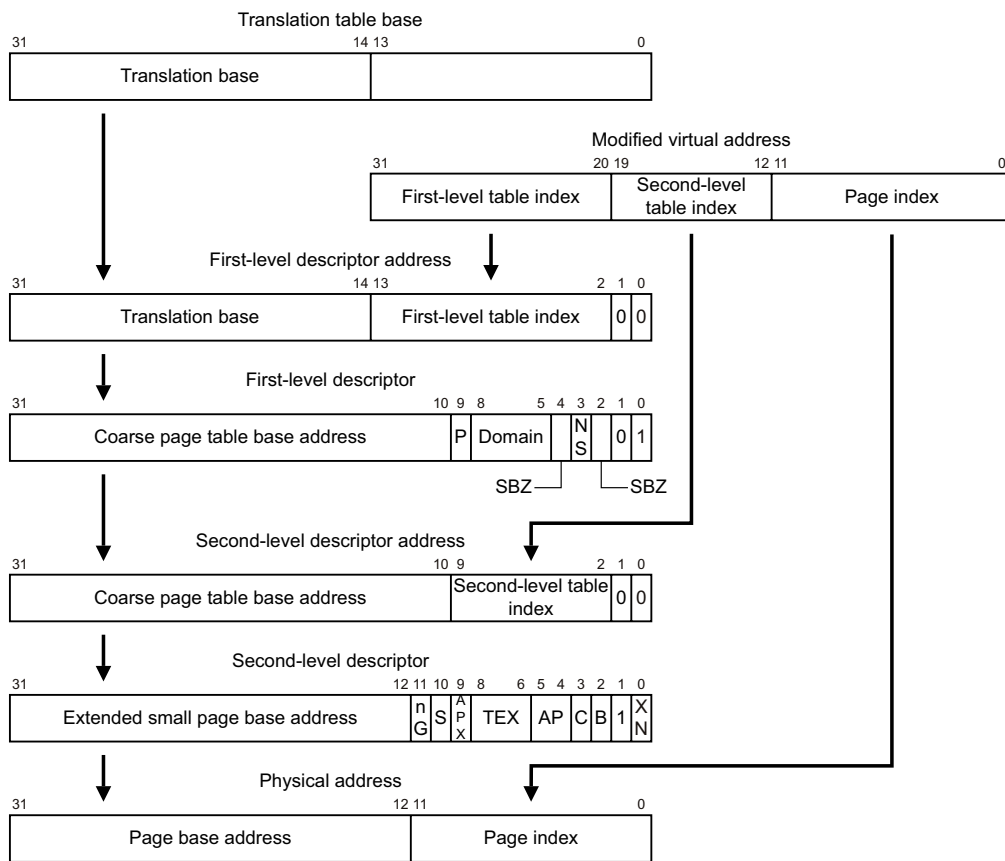


FIGURE 9.4 – Traduction d'adresse en mode ARMv6, pages de 4KB

	31	24	23	20	19	18	17	16	15	14	12	11	10	9	8	5	4	3	2	1	0	
Translation fault	Ignored																				0	0
Coarse page table	Coarse page table base address														P	Domain	S B Z	N S	S B Z	0	1	
	Section base address				N S	0	n G	S	A P X	TEX	AP	P	Domain	X N	C	B	1	0				
Supersection (16MB)	Supersection base address		SBZ		N S	1	n G	S	A P X	TEX	AP	P	Ignored	X N	C	B	1	0				
Translation fault	Reserved																				1	1

FIGURE 9.5 – Une entrée de la table des pages de niveau 1. Seules les lignes 1 et 2 sont utiles car on n'utilise ni *Section* ni *Supersection* dans ce projet. Donc soit les bits 1 et 0 sont à 00 et cela veut dire qu'il n'existe pas de traduction pour l'adresse virtuelle. Soit ils sont à 01, et la ligne 2 (Coarse page table) décrit le format de l'entrée.

**TEX, C et B** Concerne politique de cache et type de région :

- Initialiser à **TEX=000**, **C=0**, **B=1** si l'espace concerné correspond à un périphérique mappé en mémoire ;
- Initialiser à **TEX=001**, **C=0**, **B=0** sinon (mémoire partageable, pas de mise en cache)

Plus d'explications section 6.6.1 de la doc sur le processeur.

**APX** Si **ForceAP=1**, l'OS peut se servir de ce bit pour optimiser le remplacement des pages. Mais dans votre implémentation, **ForceAP=0** donc la signification de **APX** bit dépend de **AP**, cf. ci-dessous. Initialisez par exemple à 0.

**AP** Avec **APX**, la signification de ces bits est donnée par la table 9.6. Initialisez par exemple à 01.

APX	AP[1:0]	Privileged permissions	User permissions
0	b00	No access, recommended use. Read-only when S=1 and R=0 or when S=0 and R=1, deprecated.	No access, recommended use. Read-only when S=0 and R=1, deprecated.
0	b01	Read/write.	No access.
0	b10	Read/write.	Read-only.
0	b11	Read/write.	Read/write.
1	b00	Reserved.	Reserved.
1	b01	Read-only.	No access.
1	b10	Read-only.	Read-only.
1	b11	Read-only.	Read-only.

TABLE 9.6 – Permissions d'accès

	31	16	15	14	12	11	10	9	8	7	6	5	4	3	2	1	0	
Translation fault	Ignored																0	0
Large page (64KB)	Large page base address				X N	TEX	n G	S	A P X	SBZ			AP	C	B	0	1	
Extended small page (4KB)	Extended small page base address						n G	S	A P X	TEX		AP	C	B	1	X N		

FIGURE 9.7 – Une entrée d'une table des pages de niveau 2. Si les bits 1 et 0 contiennent '00', c'est une faute de traduction. S'ils contiennent '01', l'entrée correspond une Large page, 64 Ko, pas utile dans ce projet. Si le bit 0 est à '1', alors l'entrée correspond à une page de 4 Ko, ce qu'on utilise dans ce projet, et la ligne 3 décrit le format de l'entrée.

## 9.5 Exercice : Pagination sans allocation dynamique

Afin d'implémenter votre mécanisme de pagination, il vous faut :

- initialiser la table des pages : table de niveau 1 et tables de niveau 2
- initialiser la table des frames
- activer la MMU
- fournir des primitives d'allocation et libération des pages

La série de question ci-dessous vous guide dans l'implémentation de ce mécanisme. Votre implémentation se fera principalement dans un fichier `vmem.c` (et le `.h` associé).

**Question 9-1** En vous basant sur les explications des sections précédentes, en particulier la figure 9.4 :

- Rappelez la taille d’une page. Définissez la macro `PAGE_SIZE` qui vaut cette valeur.
- Combien d’entrées contient une table des pages de niveau 2 ? Définissez la macro `SECON_LVL_TT_COUNT` qui vaut cette valeur.
- Définissez la macro `SECON_LVL_TT_SIZE`, valant la taille en octets d’une table de niveau 2.
- Combien d’entrées contient la table des pages de niveau 1 ? Définissez la macro `FIRST_LVL_TT_COUNT` qui vaut cette valeur.
- Définissez la macro `FIRST_LVL_TT_SIZE`, valant la taille en octets de la table de niveau 1.

**Question 9-2** En vous basant sur les explications des sections précédentes, en particulier la figure 9.7, écrivez sur papier le champ de bit nécessaire pour une entrée d’une table des pages de niveau 2, en considérant que cette page est en lecture/écriture pour tous.

**Question 9-3** Donnez la ligne de code qui met dans la variable `uint32_t device_flags` la valeur du champ de bit pour les entrées des tables des pages de deuxième niveau lorsque les adresses physiques concernées sont entre `0x20000000` et `0x20FFFFFF`, c’est à dire là où sont mappés les périphériques.

**Question 9-4** Pour l’allocation des pages (niveau 1 et 2), vous allez devoir vous servir de la primitive `uint8_t* kAlloc_aligned(unsigned int size, unsigned int pwr_of_2)` du fichier `kheap.c`, qui alloue `size` octets à une adresse alignée à  $2^{pwr\_of\_2}$ . Pourquoi ?

**Question 9-5** Il vous faut maintenant implémenter la fonction `unsigned int init_kern_translation_table(void)` qui alloue puis initialise la table des pages de l’OS. Étant donné que vous ne relocalisez pas le code et les données statiques, votre table des pages doit traduire une adresse logique par elle-même pour toutes les adresses logiques auxquelles on trouve les données du noyau. Par exemple, l’adresse physique correspondant à l’adresse logique `0x8000` vaut : `0x8000`. Toutes les autres pages physiques sont vides au début de l’exécution de votre noyau. Nous vous proposons de traduire de cette manière :

- adresse logique = adresse physique pour toute adresse physique entre `0x0` et `__kernel_heap_end__` (re-voyez la figure 3.4 au besoin) ;
- adresse logique = adresse physique pour toute adresse physique entre `0x20000000` et `0x20FFFFFF` ;
- défaut de traduction sinon (pour le moment).

**Question 9-6** Dans une fonction `void vmem_init()`, initialisez la mémoire physique, configurez la MMU, activez les interruptions ainsi que les *data aborts* et enfin activez la MMU. Utilisez les fonctions données dans la figure 9.2. L’appel à `vmem_init()` se fait directement dans `sched_init()`, de cette façon :

```
#if VMEM
    vmem_init();
#else
    kheap_init();
#endif
```

**Question 9-7** Testez votre mécanisme de pagination. Aidez-vous de la fonction `vmem_translate()` donnée en figure 9.8, qui traduit une adresse virtuelle en adresse physique de la même manière que le fait la MMU.

## 9.6 Exercice : allocation dynamique de pages

Les questions suivantes vous guident dans l’implémentation des deux appels système suivants :

Nom	Numéro	Fonction
<code>void* sys_mmap(unsigned int size)</code>	8	Alloue <code>size</code> octets consécutifs dans l'espace d'adressage du processus courant. Retourne l'adresse du premier octet.
<code>void sys_munmap(void* addr, unsigned int nb_pages)</code>	9	Libère <code>size</code> octets préalablement alloués dans l'espace d'adressage du processus courant à partir de <code>addr</code> .

Pour allouer *une* page de mémoire virtuelle à un processus, il suffit de :

1. Trouver 1 page libre dans l'espace d'adressage du processus courant ;
2. Trouver 1 frame libre dans la mémoire physique ;
3. Insérer la traduction de l'adresse de la page vers l'adresse de la frame dans la table des pages du processus courant.

Afin que votre OS sache quelles *frames* sont encore disponibles, il faut le doter d'une table d'occupation des frames. Par exemple, chaque *i*-eme champ de cette table est un `uint8_t` indiquant si la *i*-eme frame est libre (0) ou non (1).

**Question 9-8** En regardant la figure 3.4, on voit que l'espace d'adressage physique accessible par l'ARM va de 0 à 0x20FFFFFF. Quelle est la taille de l'espace d'adressage physique ? Et quelle est la taille de l'espace d'adressage logique ?

**Question 9-9** Du coup, quelle taille fait la table d'occupation des frames ?

**Question 9-10** Allouez puis initialisez la table d'occupation des frames, dans `vmem_init()`. À ce stade, certaines frames doivent être marquées occupées car correspondent au code, données, structures noyaux etc.

**Question 9-11** Implémentez la fonction `uint8_t* vmem_alloc_for_userland(pcb_t* process)` qui alloue une page dans l'espace d'adressage du processus donné en paramètre.

**Question 9-12** On veut maintenant pouvoir allouer plus d'une page à un processus. Les pages allouées doivent-elles être situées consécutivement dans l'espace d'adressage virtuelle de ce processus ? Et les frames ?

**Question 9-13** Ajoutez un paramètre `unsigned int size` à la fonction `vmem_alloc_for_userland(...)`. Elle alloue maintenant `size` octets dans l'espace d'adressage du processus donné en paramètre. Contentez-vous d'arrondir au nombre de page supérieur nécessaire.

**Question 9-14** Implémentez la fonction `void vmem_free(uint8_t* vAddress, pcb_t* process, unsigned int size)` qui libère les pages allouées par `vmem_alloc_for_userland(...)`.

**Question 9-15** Mettez en place l'appel système `sys_mmap()`. La fonction `do_sys_mmap()` (côté noyau donc) se sert bien sûr de `vmem_alloc_for_userland(...)`.

**Question 9-16** Mettez en place l'appel système `sys_munmap()`. La fonction `do_sys_munmap()` (côté noyau donc) se sert bien sûr de `vmem_free(...)`.

**Question 9-17** Testez vos appels système `sys_mmap()` et `sys_munmap`.

## 9.7 Exercice : isolation entre processus

Vous allez maintenant gérer l'isolation entre processus. C'est à dire qu'un processus ne pourra accéder qu'aux pages de mémoire lui appartenant. Pour cela, deux solutions sont possibles :

1. Chaque processus a sa propre table des pages. Quelques précisions :
  - Lors d'un context-switch, l'OS doit invalider toutes les entrée de la TLB et fait pointer `c2/TTBR0` sur la table des pages du processus élu.
  - Le pointeur vers la table des pages est stocké dans le PCB des processus. Le pointeur de table des pages de l'OS est stocké dans une variable globale.
  - Pour invalider les entrées de la TLB, l'instruction suivante suffit : `MCR p15,0,<Rd>,c8, c6,0`, peu importe `<Rd>`.
  - Si un processus tente d'accéder à une page qui n'existe pas dans l'espace d'adressage du processus, la MMU va générer une *Translation fault*.
2. Indiquer dans la TLB le processus propriétaire de chaque page. Quelques précisions :
  - L'identifiant du propriétaire d'un bout de mémoire est un entier appelé *ASID* – *Address Space Identifier*. Chaque processus possède le sien dans son PCB et votre OS possède le sien dans une variable globale.
  - Votre OS doit maintenir un ASID courant lors des divers *context switch* et appels système. Cet ASID courant est stocké dans `c13/Context ID register`, bits 0 à 7.
  - Lors d'un accès mémoire, en cas de *TLB miss*, la MMU parcourt la table des pages, trouve l'adresse physique, et stocke dans la TLB la paire `<(ASID, adresse logique); adresse physique>`.
  - Lors d'un accès mémoire, en cas de *TLB hit*, la MMU compare l'ASID de l'entrée de la TLB avec l'ASID courant (stocké dans `c13/Context ID register`). Si les deux ne correspondent pas, la MMU génère une *Permission fault*.
  - Changer l'identifiant courant se fait grâce à l'instruction suivante : `MCR p15, 0, <Rd>, c13, c0, 1`, qui copie dans `c13/Context ID register` le contenu du registre `<Rd>`.
  - Attention, pour activer la vérification de l'ASID par la MMU, il faut que le bit `nG` de l'entrée de la TLB correspondant à l'adresse en cours d'accès soit à 1 : re-voyez la section 9.4 au besoin.

Je vous laisse libre de la solution à implémenter dans les questions suivantes.

Lors d'une faute de traduction ou une faute d'accès, plusieurs choses sont effectuées par le matériel. D'abord, certains registres du coprocesseur sont mis à jour :

- `c5/Data Fault Status Register (DFSR)` contient la cause de la faute. Les bits 0 à 3 valent :
  - 0111 si c'est une *Translation fault* de page
  - 0110 si c'est une *Access fault* sur une page
  - 1111 si c'est une *Permission fault* sur une page
 Pour lire ce registre, utilisez l'instruction `MRC p15, 0, <Rd>, c5, c0, 0` qui copie dans le registre `<Rd>` du processeur le contenu de `c5/DFSR`.
- `c6/Fault Address Register (FAR)` contient l'adresse virtuelle dont la tentative d'accès a généré une faute. Pour lire, ce registre, utilisez l'instruction `MRC p15, 0, <Rd>, c6, c0, 0` qui copie dans le registre `<Rd>` du processeur le contenu de `c6/FAR`.

Puis la MMU notifie le processeur de la faute via une interruption *Data abort*, et celui-ci va alors brancher au traitant d'interruption correspondant, c'est à dire au label `data` de `vectors.s`.

**Question 9-18** Déclarez et définissez une fonction `data_handler` et faites en sorte que celui-ci soit exécuté sur une interruption *data abort*. Dans ce traitant, pour l'instant, identifiez la cause de l'interruption et terminez l'exécution de votre noyau. Cette fonction vous sera utile pour débogger la suite.

**Question 9-19** Pour isoler, le PCB de chaque processus doit-il être alloué dans l’espace d’adressage de chaque processus, ou continuer à résider uniquement dans l’espace d’adressage du noyau ? Pourquoi ? Le cas échéant, modifier votre fonction `create_process()`.

**Question 9-20** Pour isoler, la pile d’exécution de chaque processus doit-elle être allouée dans l’espace d’adressage de chaque processus, ou continuer à résider uniquement dans l’espace d’adressage du noyau ? Pourquoi ? Le cas échéant, modifier votre fonction `create_process()`.

**Question 9-21** Montrez que la traduction qui est faite d’une même adresse logique donne deux résultats différents pour deux processus différents, ainsi que pour l’OS.

**Question 9-22** Votre traitant d’interruption `data_handler` doit maintenant gérer les fautes d’accès. Par exemple en terminant le processus fautif.

**Question 9-23** Avec ce que vous savez maintenant du fonctionnement de la MMU et du remplissage de la TLB : pourrait-on isoler les processus entre eux sans paginer sur le processeur du Raspberry Pi ?

## 9.8 Annexe : Détail du c1/Control register

Explication de la valeur du registre c1/Control register, étant donné que  $75864189 = 0 \times 485987d = (00000100100001011001010001111101)_2$

- 29 FA : 0 → AP désactivé ; ForceAP=0
- 28 TR : 0 → TEX remap=0, désactivé
- 25 EE : 0 → E bit in the CPSR IS SET TO 0 on an exception
- 24 VE : 0 → Interrupt vectors are fixed
- 23 XP : 1 → ARMv6 mode (!= ARMv5) ; subpage hardware support disabled
- 22 U : 0 → Support pour accès aux données non alignées désactivées
- 21 FI : 0 → Latence normale pour les FIQ
- 18 IT : 1 → \*deprecated\*
- 16 DT : 1 → \*deprecated\*
- 15 L4 : 1 → Loads to PC do not set the T bit ??
- 14 RR : 0 → Stratégie de remplacement du cache normale
- 13 V : 0 → Vecteurs d’interruption normaux
- 12 I : 1 → Cache d’instruction activé ??
- 11 Z : 0 → Prédiction de branchement désactivée
- 10 F : 1 → “Should be 0” ??
- 9 R : 0 → \*deprecated\*
- 8 S : 0 → \*deprecated\*
- 7 B : 0 → Little-endian
- 6-4 SBO : 111 → “Should be 1”
- 3 W : 1 → “Not implemented”
- 2 C : 1 → Cache de données niveau 1 activé
- 1 A : 0 → Détection de non-alignement non activé
- 0 M : 1 → **Activation de la MMU**



## 9.9 Annexes : la protection chez ARM

### 9.9.1 L'extension *TrustZone*

Les différents modes d'exécution (IRQ, SYSTEM, SVC, USER...) du processeur permettent d'autoriser ou pas l'exécution de certaines instructions et de changer la version de certains registres utilisés. Mais cela ne fournit pas de protection en dehors du processeur. Par exemple, cela ne garantit aucunement qu'un processus en espace utilisateur n'écrase pas des données de l'OS. Pour cela, vous avez implémenté un mécanisme d'isolation associé à la pagination. Cependant, cela vous force à invalider toute la TLB à chaque appel système. L'extension *TrustZone* permet de se passer de cela en implémentant une séparation entre deux mondes : le monde dit *Secure* dans lequel typiquement l'OS s'exécutera, et le monde *Non-secure* dans lequel les processus utilisateurs s'exécuteront. Cela impacte plusieurs composants du micro-contrôleur, en particulier bien sûr la gestion de la mémoire.

**Passage de Secure à Non-secure** Le bit NS du registre SCR (*Secure Configuration Register*) du coprocesseur permet de contrôler le passage d'un monde à l'autre.

```

uint32_t
vmem_translate(uint32_t va, struct pcb_s* process)
{
    uint32_t pa; /* The result */

    /* 1st and 2nd table addresses */
    uint32_t table_base;
    uint32_t second_level_table;

    /* Indexes */
    uint32_t first_level_index;
    uint32_t second_level_index;
    uint32_t page_index;

    /* Descriptors */
    uint32_t first_level_descriptor;
    uint32_t* first_level_descriptor_address;
    uint32_t second_level_descriptor;
    uint32_t* second_level_descriptor_address;

    if (process == NULL)
    {
        __asm("mrc p15, 0, %[tb], c2, c0, 0" : [tb] "=r"(table_base));
    }
    else
    {
        table_base = (uint32_t) process->page_table;
    }

    table_base = table_base & 0xFFFFC000;

    /* Indexes */
    first_level_index = (va >> 20);
    second_level_index = ((va << 12) >> 24);
    page_index = (va & 0x00000FFF);

    /* First level descriptor */
    first_level_descriptor_address = (uint32_t*) (table_base | (first_level_index << 2));
    first_level_descriptor = *(first_level_descriptor_address);

    /* Translation fault */
    if (! (first_level_descriptor & 0x3)) {
        return (uint32_t) FORBIDDEN_ADDRESS;
    }

    /* Second level descriptor */
    second_level_table = first_level_descriptor & 0xFFFFC00;
    second_level_descriptor_address = (uint32_t*) (second_level_table | (second_level_index << 2));
    second_level_descriptor = *((uint32_t*) second_level_descriptor_address);

    /* Translation fault */
    if (! (second_level_descriptor & 0x3)) {
        return (uint32_t) FORBIDDEN_ADDRESS;
    }

    /* Physical address */
    pa = (second_level_descriptor & 0xFFFFF000) | page_index;

    return pa;
}

```

FIGURE 9.8 – Cette fonction traduit l’adresse virtuelle en une adresse physique de la en se basant sur la table des pages du processus donné en paramètre, de la meme façon que le ferait la MMU.

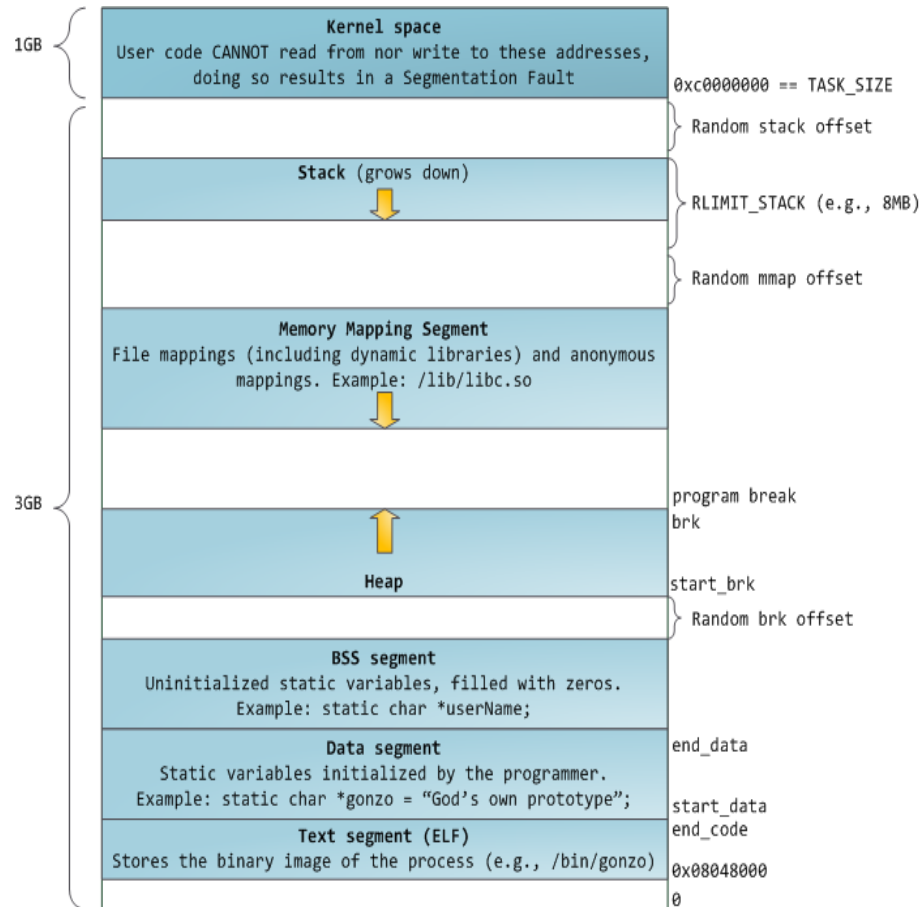


FIGURE 9.9 – Organisation of memory space in Linux

## Chapitre 10

# Partage de la mémoire

Ce chapitre n'a manifestement pas eu le temps d'être rédigé correctement. Cependant, le principe du partage de pages a été vu en cours et, surtout, vos profs de TP sont là pour ça.

# Chapitre 11

## 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.

### 11.1 Sortie vidéo

Pour votre mini-OS, un petit driver pour la sortie vidéo est dispo sur la page du cours. Il fournit les primitives `put_pixel_RGB24(...)` et `FramebufferInitialize()` à partir desquelles vous pouvez faire ce que vous voulez : programmer une sortie texte, afficher des photos, pourquoi pas jouer de la vidéo...

Sous Linux, c'est cadeau.

### 11.2 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/>.

### 11.3 Sortie série

Les fichiers `uart.c` et `uart.h` permettent d'afficher du texte via la sortie texte de l'émulateur. Il suffit de :

- Déclarer un buffer de communication :

```
#define UART_BUFFER_SIZE 256u
static char uart_buffer[UART_BUFFER_SIZE];
```
- Affichez des trucs, comme ça :

```
uart_send_str("Enfin un semblant de printf...\n");
```

### 11.4 Lecteur WAV

On vous fournit en ligne deux fichiers permettant de jouer des fichiers `.wav` sur la sortie JACK pour votre mini-OS. Ils s'utilisent de la manière suivante :

- Mettez le fichier `tune.wav` que vous voulez jouer à la racine de votre projet ;
- Faîtes en sorte de compiler `pwm.c` et lier le fichier objet résultant à votre noyau ;
- Ajouter à votre Makefile la règle suivante :

```
tune.o : tune.wav
$(CC)-ld -s -r -o $@ -b binary $^
```

- modifier votre Makefile pour que `tune.o` soit lié à votre noyau lors de l'édition de lien.
- l'appel à `audio_test()` joue `tune.wav`. À vous d'adapter le code pour montrer ce que vous voulez.

## 11.5 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.
```

## 11.6 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.

## 11.7 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.

## 11.8 Jeux : casse-briques, téttris 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 :)

## 11.9 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.



```

    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 */
    }
}

```

#### Encart 14: 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 11-1** Donnez la déclaration de la structure de donnée associée à un sémaphore.

**Question 11-2** Proposez une implantation de la primitive

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

**Question 11-3** 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 11-4** Proposez une implantation des deux primitives

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

## 11.10 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 débloquent 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 11-5** 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 11-6** 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 12

## Allocation dynamique de mémoire

### 12.1 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 appellera `gmalloc` et `gfree`). Dans un premier temps, nous réaliserons une implémentation simple et efficace de ces primitives. Dans un deuxième temps, vous les optimiserez.

#### 12.1.1 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 5). 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.

#### 12.1.2 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.

### 12.1.3 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.

## 12.2 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 chaînées de blocs d'une certaine taille. Gardez en mémoire 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 généralité, il faut savoir dans quelle mesure.

# Chapitre 13

## Linux sur Raspberry Pi

Ce chapitre présente quelques aspects techniques relatifs à l'installation, et l'exécution de Linux sur les Raspberry Pi.

### 13.1 Installation et exécution de Linux

Un Raspberry démarre sur le système présent sur la carte SD. La distribution Linux doit donc être présente sur cette carte mémoire. Si vous avez une carte SD sans distribution Linux, ou bien que vous avez écrasé certains fichiers, vous pouvez re-installer une distribution.

### 13.2 Accès à internet

Vous pouvez connecter vos Raspberry Pi en filaire via les prises Ethernet suivantes :

- Salle 208 : B6P4A10, B6P4A11, B5P7A13, B5P5B1, B5P5B12
- Salle 219 : B5P6B3, B5P6A8, B5P6A7, B5P6A1

### 13.3 Installation de logiciel

Une fois connecté à internet, pour installer des paquets sous TODO

### 13.4 L'ordonnancement sous Linux

Histoire de voir que les concepts sont vraiment les mêmes dans un vrai OS que dans votre mini-OS, il s'agit dans ce chapitre de manipuler l'ordonnanceur Linux. Typiquement, vous devez observer l'effet d'un changement de la niceness, d'un changement d'ordonnanceur (utilisation de l'ordonnanceur à priorités fixes). PEU d'indication, vous êtes libres d'illustrer ça de la manière que vous voulez. Quelques pointeurs :

- La page de manuel de [sched\\_setscheduler](#)
- Le chapitre 14 du livre [Understanding the Linux kernel](#)

## Troisième partie

### Annexes

## Annexe A

# Interruptions

Rappelez-vous, en 3IF vous avez configuré un timer sur les cartes TI à base de MSP430. Vous avez dû positionner un vecteur d'interruption et écrire le **traitant 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 déclencher une interruption toutes les 10 ms ;
- lorsqu'une interruption se produit, le code boucle sur le label `irq`. Pour changer afin d'appeler une fonction lors d'une interruption, ça se passe dans `init.s`.

## Annexe B

# Gestionnaire de mémoire physique simple

Le gestionnaire de mémoire physique qui vous est fourni est tellement simple qu'il vous servira de documentation.

## Annexe C

# Installation des outils nécessaires à la réalisation du projet

### C.1 Installation de Qemu pour Raspberry Pi

D'abord, téléchargez la branche qui va bien du Qemu spécial Raspberry :

```
— git clone https://github.com/Torlus/qemu.git
— cd qemu
— git checkout -b raspberry origin/rpi
```

Ensuite, configurez et compilez :

```
— ./configure --target-list="arm-softmmu"
— make
```

Le binaire est ensuite dans `arm-softmmu/qemu-system-arm`

Modifiez le fichier `tools/run-qemu.sh` en conséquence, ou ajoutez un lien symbolique au bon endroit (ou modifiez votre `PATH`, tout ça).

### C.2 Installation de GCC pour ARM

**Ubuntu** installez le paquet `gcc-arm-none-eabi`

**Mac OS X** avec MacPorts, installez le paquet `arm-none-eabi-gcc`

### C.3 Installation de GDB pour ARM

**Linux** Sous Ubuntu, le paquet à installer s'appelle `gdb-arm-none-eabi` ; mais attention, la version courante d'Ubuntu (15.04 pour encore quelques semaines) a dans ses repos la version 7.8 de GDB. Or cette version est buggée et ne vous autorisera pas à utiliser le système de test que nous vous fournissons. Il vous faut dans ce cas recompiler gdb. C'est ultra simple :

```
— Téléchargez la version 7.10, par exemple depuis http://www.gnu.org/software/gdb/download/.
— tar xzf gdb-7.10.tar.gz
— cd gdb-7.10
— ./configure --prefix=<install_dir> --target=arm-none-eabi --enable-tui
— make
— make install
— Ajoutez <install_dir>/bin à votre PATH
```



**Mac OS X** Avec MacPorts, installez le paquet `arm-none-eabi-gdb`

## Annexe D

# FAQ

### **Why does the GPU control the first stages of the boot process of the Raspberry Pi boot on the GPU?**

The SoC included in the Raspberry Pi is a Broadcom BCM2835. It is a multimedia applications processor that might have started out as a GPU only until the designers figured out how to attach an ARM CPU. Since the GPU already works there is not much reason to redesign the silicon die complete, just create a way for the CPU to interface with the GPU. An other reason is that the GPU directly accesses the RAM for low latency.

### **The assembly code seems sub-optimal, why ?**

Have you checked the optimisation level ? -O0 ?

## Annexe E

# Aide-mémoire gdb

Dans ce projet, vous allez devoir passer un temps considérable à faire la *mise au point* (*debugging*) de votre programme avec `gdb`. Vous gagnerez un temps précieux si vous vous donnez les moyens de l'utiliser efficacement. Pour vous aider, nous vous fournissons plusieurs ressources, prenez le temps de les regarder :

**GDB quick reference** Les deux pages suivantes sont une *reference card* rassemblant les commandes les plus utilisées. La plupart de ces commandes vous seront utiles !

**Commandes «maison»** Vous trouverez dans le répertoire `tools` un fichier de configuration `init.gdb` qui vous évite de retaper tout le temps les mêmes commandes. N'hésitez pas à modifier ce fichier tout au long du projet (il est fait pour ça) par exemple en rajoutant des breakpoints aux endroits qui vous intéressent.

Vous aurez remarqué que ce `init.gdb` fait référence à un autre fichier (`utils.gdb`) dans lequel nous vous avons écrit quelques commandes `gdb` spécifiques à ce projet. Prenez le temps de les lire et de les comprendre, elles vous feront gagner un temps précieux. Vous êtes d'ailleurs encouragés à écrire vous-même vos propres commandes !

**Documentation** Le manuel de référence de `gdb` est disponible sur le web : <https://sourceware.org/gdb/current/onlinedocs/gdb/>

Essential Commands

<code>gdb program [core]</code>	debug <i>program</i> [using coredump <i>core</i> ]
<code>b [file:]function</code>	set breakpoint at <i>function</i> [in <i>file</i> ]
<code>run [arglist]</code>	start your program [with <i>arglist</i> ]
<code>bt</code>	backtrace: display program stack
<code>p expr</code>	display the value of an expression
<code>c</code>	continue running your program
<code>n</code>	next line, stepping over function calls
<code>s</code>	next line, stepping into function calls

Starting GDB

<code>gdb</code>	start GDB, with no debugging files
<code>gdb program</code>	begin debugging <i>program</i>
<code>gdb program core</code>	debug coredump <i>core</i> produced by <i>program</i>
<code>gdb --help</code>	describe command line options

Stopping GDB

<code>quit</code>	exit GDB; also <code>q</code> or EOF (eg <code>C-d</code> )
<code>INTERRUPT</code>	(eg <code>C-c</code> ) terminate current command, or send to running process

Getting Help

<code>help</code>	list classes of commands
<code>help class</code>	one-line descriptions for commands in <i>class</i>
<code>help command</code>	describe <i>command</i>

Executing your Program

<code>run arglist</code>	start your program with <i>arglist</i>
<code>run</code>	start your program with current argument list
<code>run ... &lt;inf&gt;outf</code>	start your program with input, output redirected
<code>kill</code>	kill running program
<code>tty dev</code>	use <i>dev</i> as stdin and stdout for next <code>run</code>
<code>set args arglist</code>	specify <i>arglist</i> for next <code>run</code>
<code>set args</code>	specify empty argument list
<code>show args</code>	display argument list
<code>show env</code>	show all environment variables
<code>show env var</code>	show value of environment variable <i>var</i>
<code>set env var string</code>	set environment variable <i>var</i>
<code>unset env var</code>	remove <i>var</i> from environment

Shell Commands

<code>cd dir</code>	change working directory to <i>dir</i>
<code>pwd</code>	Print working directory
<code>make ...</code>	call “ <code>make</code> ”
<code>shell cmd</code>	execute arbitrary shell command string

[ ] surround optional arguments ... show one or more arguments

Breakpoints and Watchpoints

<code>break [file:]line</code>	set breakpoint at <i>line</i> number [in <i>file</i> ]
<code>b [file:]line</code>	eg: <code>break main.c:37</code>
<code>break [file:]func</code>	set breakpoint at <i>func</i> [in <i>file</i> ]
<code>break +offset</code>	set break at <i>offset</i> lines from current stop
<code>break -offset</code>	
<code>break *addr</code>	set breakpoint at address <i>addr</i>
<code>break</code>	set breakpoint at next instruction
<code>break ... if expr</code>	break conditionally on nonzero <i>expr</i>
<code>cond n [expr]</code>	new conditional expression on breakpoint <i>n</i> ; make unconditional if no <i>expr</i>
<code>tbreak ...</code>	temporary break; disable when reached
<code>rbreak regex</code>	break on all functions matching <i>regex</i>
<code>watch expr</code>	set a watchpoint for expression <i>expr</i>
<code>catch event</code>	break at <i>event</i> , which may be <code>catch</code> , <code>throw</code> , <code>exec</code> , <code>fork</code> , <code>vfork</code> , <code>load</code> , or <code>unload</code> .
<code>info break</code>	show defined breakpoints
<code>info watch</code>	show defined watchpoints

<code>clear</code>	delete breakpoints at next instruction
<code>clear [file:]fun</code>	delete breakpoints at entry to <i>fun</i> ()
<code>clear [file:]line</code>	delete breakpoints on source line
<code>delete [n]</code>	delete breakpoints [or breakpoint <i>n</i> ]

<code>disable [n]</code>	disable breakpoints [or breakpoint <i>n</i> ]
<code>enable [n]</code>	enable breakpoints [or breakpoint <i>n</i> ]
<code>enable once [n]</code>	enable breakpoints [or breakpoint <i>n</i> ]; disable again when reached
<code>enable del [n]</code>	enable breakpoints [or breakpoint <i>n</i> ]; delete when reached

<code>ignore n count</code>	ignore breakpoint <i>n</i> , <i>count</i> times
-----------------------------	---

<code>commands n</code>	execute GDB <i>command-list</i> every time breakpoint <i>n</i> is reached.
<code>[silent]</code>	<code>[silent]</code> suppresses default display
<code>command-list</code>	
<code>end</code>	end of <i>command-list</i>

Program Stack

<code>backtrace [n]</code>	print trace of all frames in stack; or of <i>n</i> frames—innermost if <i>n</i> >0, outermost if <i>n</i> <0
<code>bt [n]</code>	
<code>frame [n]</code>	select frame number <i>n</i> or frame at address <i>n</i> ; if no <i>n</i> , display current frame
<code>up n</code>	select frame <i>n</i> frames up
<code>down n</code>	select frame <i>n</i> frames down
<code>info frame [addr]</code>	describe selected frame, or frame at <i>addr</i>
<code>info args</code>	arguments of selected frame
<code>info locals</code>	local variables of selected frame
<code>info reg [rn]...</code>	register values [for regs <i>rn</i> ] in selected frame; <code>all-reg</code> includes floating point
<code>info all-reg [rn]</code>	

Execution Control

<code>continue [count]</code>	continue running; if <i>count</i> specified, ignore this breakpoint next <i>count</i> times
<code>c [count]</code>	
<code>step [count]</code>	execute until another line reached; repeat <i>count</i> times if specified
<code>s [count]</code>	
<code>stepi [count]</code>	step by machine instructions rather than source lines
<code>si [count]</code>	
<code>next [count]</code>	execute next line, including any function calls
<code>n [count]</code>	
<code>nexti [count]</code>	next machine instruction rather than source line
<code>ni [count]</code>	
<code>until [location]</code>	run until next instruction (or <i>location</i> )
<code>finish</code>	run until selected stack frame returns
<code>return [expr]</code>	pop selected stack frame without executing [setting return value]
<code>signal num</code>	resume execution with signal <i>s</i> (none if 0)
<code>jump line</code>	resume execution at specified <i>line</i> number
<code>jump *address</code>	or <i>address</i>
<code>set var=expr</code>	evaluate <i>expr</i> without displaying it; use for altering program variables

Display

<code>print [/f] [expr]</code>	show value of <i>expr</i> [or last value \$] according to format <i>f</i> :
<code>p [/f] [expr]</code>	
<code>x</code>	hexadecimal
<code>d</code>	signed decimal
<code>u</code>	unsigned decimal
<code>o</code>	octal
<code>t</code>	binary
<code>a</code>	address, absolute and relative
<code>c</code>	character
<code>f</code>	floating point
<code>call [/f] expr</code>	like <code>print</code> but does not display <code>void</code>
<code>x [/Nuf] expr</code>	examine memory at address <i>expr</i> ; optional format spec follows slash
<code>N</code>	count of how many units to display
<code>u</code>	unit size; one of
<code>b</code>	individual bytes
<code>h</code>	halfwords (two bytes)
<code>w</code>	words (four bytes)
<code>g</code>	giant words (eight bytes)
<code>f</code>	printing format. Any <code>print</code> format, or
<code>s</code>	null-terminated string
<code>i</code>	machine instructions
<code>disassem [addr]</code>	display memory as machine instructions

Automatic Display

<code>display [/f] expr</code>	show value of <i>expr</i> each time program stops [according to format <i>f</i> ]
<code>display</code>	display all enabled expressions on list
<code>undisplay n</code>	remove number(s) <i>n</i> from list of automatically displayed expressions
<code>disable disp n</code>	disable display for expression(s) number <i>n</i>
<code>enable disp n</code>	enable display for expression(s) number <i>n</i>
<code>info display</code>	numbered list of display expressions

Expressions

<i>expr</i>	an expression in C, C++, or Modula-2 (including function calls), or:
<i>addr@len</i>	an array of <i>len</i> elements beginning at <i>addr</i>
<i>file::nm</i>	a variable or function <i>nm</i> defined in <i>file</i>
<i>{type}addr</i>	read memory at <i>addr</i> as specified <i>type</i>
<b>\$</b>	most recent displayed value
<b>\$n</b>	<i>n</i> th displayed value
<b>\$\$</b>	displayed value previous to <b>\$</b>
<b>\$\$n</b>	<i>n</i> th displayed value back from <b>\$</b>
<b>\$_</b>	last address examined with <b>x</b>
<b>\$_</b>	value at address <b>\$_</b>
<b>\$var</b>	convenience variable; assign any value
<b>show values</b> [ <i>n</i> ]	show last 10 values [or surrounding <i>\$n</i> ]
<b>show conv</b>	display all convenience variables

Symbol Table

<b>info address</b> <i>s</i>	show where symbol <i>s</i> is stored
<b>info func</b> [ <i>regex</i> ]	show names, types of defined functions (all, or matching <i>regex</i> )
<b>info var</b> [ <i>regex</i> ]	show names, types of global variables (all, or matching <i>regex</i> )
<b>whatis</b> [ <i>expr</i> ]	show data type of <i>expr</i> [or <b>\$</b> ] without evaluating; <b>p</b> <i>type</i> gives more detail
<b>p</b> <i>type</i> [ <i>expr</i> ]	
<b>p</b> <i>type type</i>	describe type, struct, union, or enum

GDB Scripts

<b>source</b> <i>script</i>	read, execute GDB commands from file <i>script</i>
<b>define</b> <i>cmd</i> <i>command-list</i>	create new GDB command <i>cmd</i> ; execute script defined by <i>command-list</i>
<b>end</b>	end of <i>command-list</i>
<b>document</b> <i>cmd</i> <i>help-text</i>	create online documentation for new GDB command <i>cmd</i>
<b>end</b>	end of <i>help-text</i>

Signals

<b>handle</b> <i>signal act</i>	specify GDB actions for <i>signal</i> :
<b>print</b>	announce signal
<b>noprint</b>	be silent for signal
<b>stop</b>	halt execution on signal
<b>nostop</b>	do not halt execution
<b>pass</b>	allow your program to handle signal
<b>nopass</b>	do not allow your program to see signal
<b>info signals</b>	show table of signals, GDB action for each

Debugging Targets

<b>target</b> <i>type param</i>	connect to target machine, process, or file
<b>help target</b>	display available targets
<b>attach</b> <i>param</i>	connect to another process
<b>detach</b>	release target from GDB control

Controlling GDB

<b>set</b> <i>param value</i>	set one of GDB's internal parameters
<b>show</b> <i>param</i>	display current setting of parameter
Parameters understood by <b>set</b> and <b>show</b> :	
<b>complaint</b> <i>limit</i>	number of messages on unusual symbols
<b>confirm</b> <i>on/off</i>	enable or disable cautionary queries
<b>editing</b> <i>on/off</i>	control <b>readline</b> command-line editing
<b>height</b> <i>lpp</i>	number of lines before pause in display
<b>language</b> <i>lang</i>	Language for GDB expressions ( <b>auto</b> , <b>c</b> or <b>modula-2</b> )
<b>listsize</b> <i>n</i>	number of lines shown by <b>list</b>
<b>prompt</b> <i>str</i>	use <i>str</i> as GDB prompt
<b>radix</b> <i>base</i>	octal, decimal, or hex number representation
<b>verbose</b> <i>on/off</i>	control messages when loading symbols
<b>width</b> <i>cpl</i>	number of characters before line folded
<b>write</b> <i>on/off</i>	Allow or forbid patching binary, core files (when reopened with <b>exec</b> or <b>core</b> )
<b>history</b> ...	groups with the following options:
<b>h</b> ...	
<b>h exp</b> <i>off/on</i>	disable/enable <b>readline</b> history expansion
<b>h file</b> <i>filename</i>	file for recording GDB command history
<b>h size</b> <i>size</i>	number of commands kept in history list
<b>h save</b> <i>off/on</i>	control use of external file for command history
<b>print</b> ...	groups with the following options:
<b>p</b> ...	
<b>p address</b> <i>on/off</i>	print memory addresses in stacks, values
<b>p array</b> <i>off/on</i>	compact or attractive format for arrays
<b>p demangl</b> <i>on/off</i>	source (demangled) or internal form for C++ symbols
<b>p asm-dem</b> <i>on/off</i>	demangle C++ symbols in machine-instruction output
<b>p elements</b> <i>limit</i>	number of array elements to display
<b>p object</b> <i>on/off</i>	print C++ derived types for objects
<b>p pretty</b> <i>off/on</i>	struct display: compact or indented
<b>p union</b> <i>on/off</i>	display of union members
<b>p vtbl</b> <i>off/on</i>	display of C++ virtual function tables
<b>show commands</b>	show last 10 commands
<b>show commands</b> <i>n</i>	show 10 commands around number <i>n</i>
<b>show commands +</b>	show next 10 commands

Working Files

<b>file</b> [ <i>file</i> ]	use <i>file</i> for both symbols and executable; with no arg, discard both
<b>core</b> [ <i>file</i> ]	read <i>file</i> as coredump; or discard
<b>exec</b> [ <i>file</i> ]	use <i>file</i> as executable only; or discard
<b>symbol</b> [ <i>file</i> ]	use symbol table from <i>file</i> ; or discard
<b>load</b> <i>file</i>	dynamically link <i>file</i> and add its symbols
<b>add-sym</b> <i>file addr</i>	read additional symbols from <i>file</i> , dynamically loaded at <i>addr</i>
<b>info files</b>	display working files and targets in use
<b>path</b> <i>dirs</i>	add <i>dirs</i> to front of path searched for executable and symbol files
<b>show path</b>	display executable and symbol file path
<b>info share</b>	list names of shared libraries currently loaded

Source Files

<b>dir</b> <i>names</i>	add directory <i>names</i> to front of source path
<b>dir</b>	clear source path
<b>show dir</b>	show current source path
<b>list</b>	show next ten lines of source
<b>list -</b>	show previous ten lines
<b>list</b> <i>lines</i>	display source surrounding <i>lines</i> , specified as:
[ <i>file</i> :] <i>num</i>	line number [in named file]
[ <i>file</i> :] <i>function</i>	beginning of function [in named file]
+ <i>off</i>	<i>off</i> lines after last printed
- <i>off</i>	<i>off</i> lines previous to last printed
* <i>address</i>	line containing <i>address</i>
<b>list</b> <i>f,l</i>	from line <i>f</i> to line <i>l</i>
<b>info line</b> <i>num</i>	show starting, ending addresses of compiled code for source line <i>num</i>
<b>info source</b>	show name of current source file
<b>info sources</b>	list all source files in use
<b>forw</b> <i>regex</i>	search following source lines for <i>regex</i>
<b>rev</b> <i>regex</i>	search preceding source lines for <i>regex</i>

GDB under GNU Emacs

<b>M-x</b> <b>gdb</b>	run GDB under Emacs
<b>C-h</b> <b>m</b>	describe GDB mode
<b>M-s</b>	step one line ( <b>s</b> tep)
<b>M-n</b>	next line ( <b>n</b> ext)
<b>M-i</b>	step one instruction ( <b>s</b> tepi)
<b>C-c</b> <b>C-f</b>	finish current stack frame ( <b>f</b> inish)
<b>M-c</b>	continue ( <b>c</b> ont)
<b>M-u</b>	up <i>arg</i> frames ( <b>u</b> p)
<b>M-d</b>	down <i>arg</i> frames ( <b>d</b> own)
<b>C-x</b> <b>&amp;</b>	copy number from point, insert at end
<b>C-x</b> <b>SPC</b>	(in source file) set break at point

GDB License

<b>show copying</b>	Display GNU General Public License
<b>show warranty</b>	There is NO WARRANTY for GDB. Display full no-warranty statement.

Copyright ©1991,'92,'93,'98,2000 Free Software Foundation, Inc.  
Author: Roland H. Pesch

The author assumes no responsibility for any errors on this card.

This card may be freely distributed under the terms of the GNU General Public License.

Please contribute to development of this card by annotating it. Improvements can be sent to bug-gdb@gnu.org.

GDB itself is free software; you are welcome to distribute copies of it under the terms of the GNU General Public License. There is absolutely no warranty for GDB.

# Glossary

## **appel système**

fonction primitive fournie par le noyau d'un système d'exploitation. Une telle fonction est utilisée par les programmes utilisateurs pour demander un service au noyau. [21](#)

## **attribut**

Information donnée au compilateur pour l'aider à optimiser sa génération de code, ou lui indiquer de compiler le code d'une certaine manière. Il existe des attributs de fonction, de variable, et de type. [15](#)

## **contexte d'exécution**

Une photo des registres CPU prise à un certain point du programme. Recharger un contexte dans le CPU permet de reprendre l'exécution du programme à partir de ce point (puisque le registre PC fait partie du contexte) [31](#)

## **dispatcher**

The part of the operating system responsible for saving and restoring execution contexts [30](#)

## **élection**

TODO [37](#)

## **épilogue**

Quelques instructions assembleur générées par le compilateur à la toute fin de chaque fonction, afin de restituer la frame de la fonction appelante. [15](#)

## **état d'un processus**

TODO [31](#)

## **mode d'exécution**

TODO [78](#)

## **notion**

Une notion, en philosophie, est une connaissance élémentaire, souvent tirée d'observations empiriques. [5](#)

## **ordonnanceur**

TODO [36](#), [37](#)

## **PCB**

Process Control Block [31](#)

## **privilège**

TODO. Et dire que ça inclut le [mode d'exécution](#), mais pas seulement. [21](#)

## **processus**

Programme en cours d'exécution [31](#)

**prologue**

Quelques instructions assembleur générées par le compilateur au tout début de chaque fonction, afin de préparer l'espace dédié à l'exécution de la fonction sur la pile d'exécution. [15](#)

**requête d'interruption, ou IRQ (*Interrupt Service Routine*)**

Signal matériel envoyé au CPU par un périphérique pour lui notifier un évènement. Voir aussi [traitant d'interruption](#). [79](#)

**round-robin**

Politique d'ordonnancement consistant à exécuter les processus chacun leur tour [37](#)

**traitant d'interruption, ou ISR (*Interrupt Service Routine*)**

procédure invoquée automatiquement par le CPU lorsqu'il reçoit une [requête d'interruption](#). Voir aussi l'encart [13](#) page [42](#). [15](#), [41](#), [70](#), [79](#)