INSA Lyon
4ème année
Spécialité Informatique

Systèmes d'exploitation avancés

Ordonnancement CPU, Gestion mémoire, Appels Système, Virtualisation

Kevin Marquet

Septembre 2014

Table des matières

Chapitre 1

Introduction

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

1.1 Déroulement

Ce semestre comprend 2 périodes :

Semaines 39 à 44 Travail en binômes

Semaines 47 à 4 Travail en héxanômes

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

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

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

1.2 Évaluation au cours du semestre

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

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

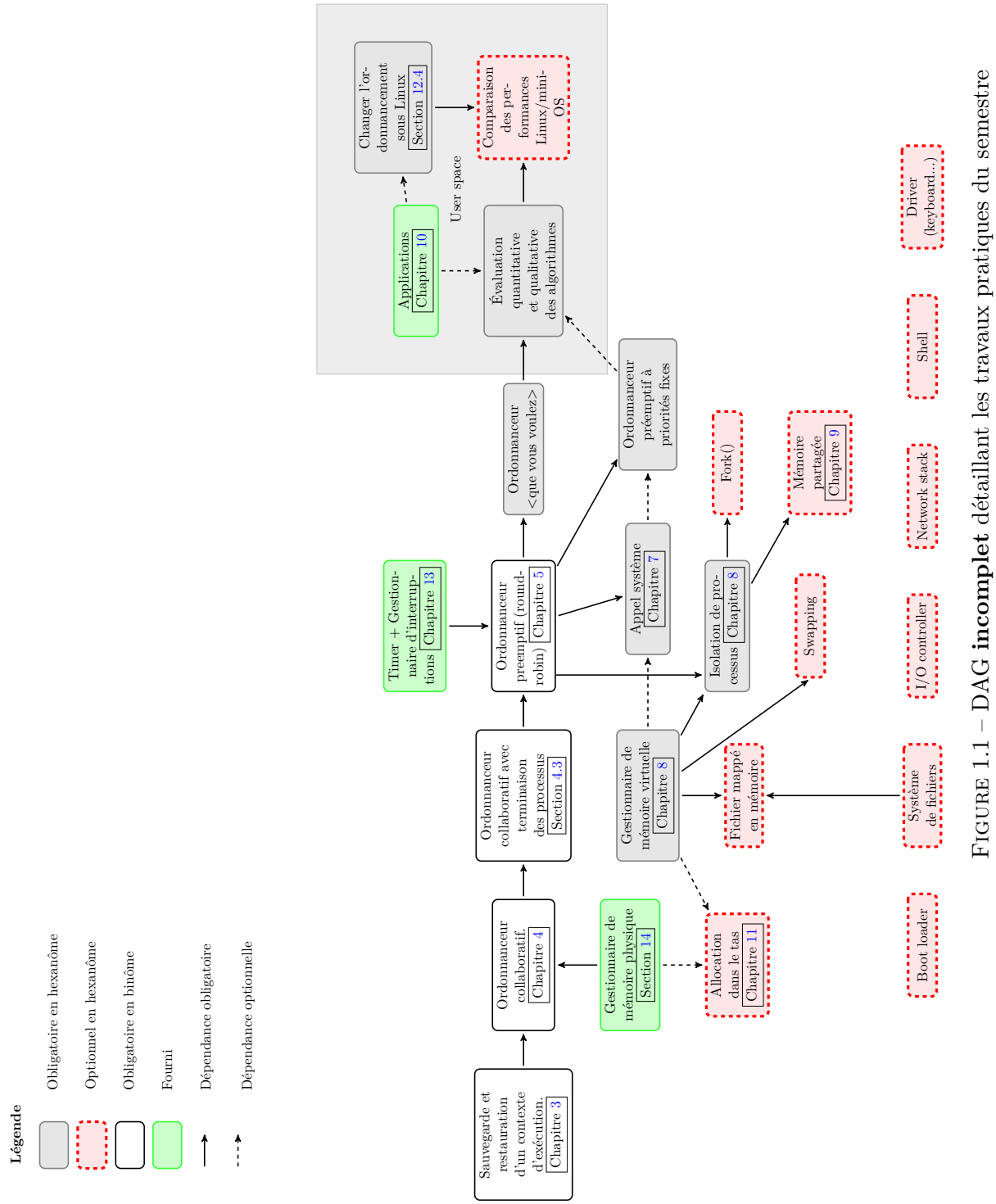

FIGURE 1.1 – DAG **incomplet** détaillant les travaux pratiques du semestre

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

La note de DS prendra en compte les éléments suivants :

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

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

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

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

Première partie

Semaines 39 à 44 : travail en binômes

Chapitre 2

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

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

2.1 Ce que vous allez apprendre dans ce chapitre

PL / Code Generation : Concepts.

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

PL / Code Generation : Skills.

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

2.2 Cross-compilation

Wikipedia.en :

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

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

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

Rappelez-vous... la cross-compilation

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

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

2.3 Émulation de la plateforme

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

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

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

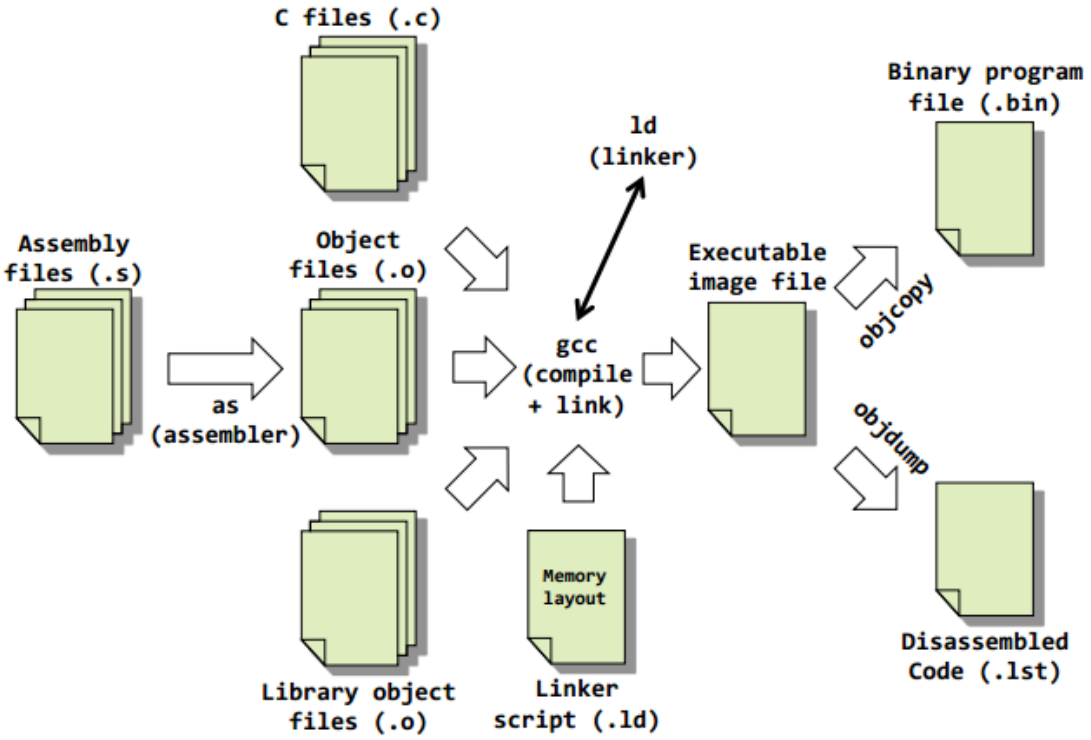

FIGURE 2.1 – Chaîne de compilation

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

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

Rappelez-vous... GDB

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

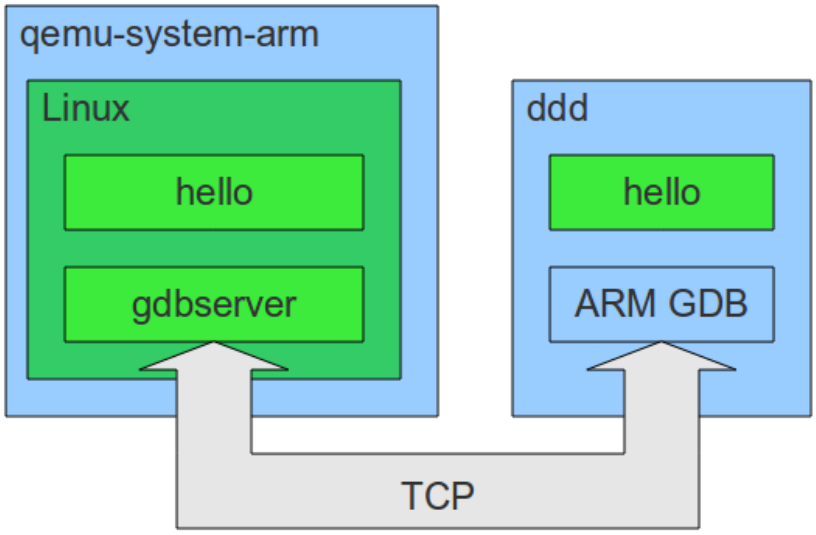

FIGURE 2.2 – Remote debugging

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

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

Exercice : Prise en main des outils

Mettez en place votre code de la façon suivante :

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

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

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

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

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

  return result;
}

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

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

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

  volume = compute_volume(radius);

  return volume;
}
```

FIGURE 2.3 – Un premier bout de code pour observer

2.4 Exécution sur la plateforme

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

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

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

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/

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 instruction lisent la carte SD, et chargent le deuxième bootloader stocké dans le fichier `bootcode.bin` dans le cache L2 ;
- L'excution de ce programme allume la SDRAM puis charge le troisième bootloader (fichier `loader.bin`) en RAM et l'exécute ;
- Ce programme charge `start.elf` à l'adresse zéro, et le processeur ARM l'exécute ;
- `start.elf` ne fait que charger `kernel.img`, dans lequel votre code se trouve ! Petit détail : le `start.elf` qu'on utilise sur la carte est celui d'une distribution Linux qui saute à l'adresse 0x8000 car des informations (paramétrage du noyau, configuration de la MMU) sont stockés entre les adresses 0x0 et 0x8000. Il est possible de changer cela facilement (on ne le fera pas pendant ce TP) : http://www.raspberrypi.org/documentation/configuration/config-txt.md

Plus d'infos :

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

2.5 Documentation

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

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

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

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

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

Chapitre 3

Sauvegarde et restauration d'un contexte d'exécution

3.1 Ce que vous allez apprendre dans ce chapitre

OS / Concurrency : Concepts.

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

OS / Concurrency : Skills.

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

3.2 Retour à un contexte

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

Contexte d'exécution

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

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

Processeur du Raspberry Pi

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

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

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

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

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

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

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

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

Lier de l'assembleur et du code C

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

- `__asm("mov %0, r0" : "=r"(X));` pour sauvegarder le registre `r0` dans une variable nommée `X` ;
- Le code suivant effectue l'opération inverse, à savoir lire la variable `X` et copier son contenu dans le registre `r0` : `__asm("mov r0, %0" : : "r"(X));`

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

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

Exercice : observation du code à l'exécution

Question 3-1

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

— Observez les valeurs de **sp** en exécutant le programme simple que vous avez utilisé en Section 2.3 ;
— Comparez la valeur de **sp** avec les adresses des première et dernière variables locales déclarées dans ces fonctions ;
— Comparez cette valeur avec les adresses des premier et dernier paramètres de ces fonctions ;
— Dessinez la mémoire autour de l'adresse contenue dans **sp** en y reportant toutes les adresses ;
— Dans quel sens croît la pile ?
— Observez l'utilisation du registre **lr** avant et après chaque appel de fonction, ainsi qu'au retour de chacune. Quelles instructions assembleur le met à jour implicitement ? Au besoin, re-lisez le rôle du registre **lr** dans l'encart page précédente.

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

3.3 Un dispatcher pour coroutines

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

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

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

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

Gestion de la mémoire

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

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

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

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

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

Exercice : Dispatcher pour coroutines

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

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

```
#include "hw.h"
#include "sched.h"

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

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

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

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

  current_ctx = &ctx_init;
  switch_to(&ctx_ping);

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

FIGURE 3.1 – Ce code doit tourner sans modification

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

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

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

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

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

Notes pour cette question :

— dans le code fourni, le contexte `ctx_init` est un contexte qui n'est plus utile une fois la première coroutine appelée. Il ne servira que dans les deux premiers prochains exercices. Il sert uniquement à la simplicité de mise en oeuvre.
— faisez gaffe, les contextes dans lesquels s'exécutent `ping()` et `pong()` sont déclarés dans `kernel.c` et sont donc alloués statiquement. Il vous est donc inutile de gérer leur allocation dynamique (vous inquiétez pas hein, vous vous chargerez plus tard de l'allocation des contextes, quand vous gèrerez plusieurs processus) ;

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

(Rappelez vous...) Les attributs de fonction

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

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

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

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

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

Cette primitive, lorsqu'elle est appelée, va :

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

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

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

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

3.4 Sauvegarde des variables locales

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

Remarquez que ces deux fonctions possèdent maintenant des variables locales, et effectuent des calculs. Il ne suffit donc plus de sauvegarder et restaurer les pointeurs de pile et d'instruction mais tous les registres. Vous

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

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

void
pong()
{
  int cpt = 1;

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

FIGURE 3.2 – Avec les variables locales

pourriez sauvegarder ces registres dans les structures `struct ctx_s`, mais je vous suggère de plutôt les sauvegarder dans la pile d'exécution. C'est ce qui est fait de manière traditionnelle dans les systèmes d'exploitation, les machines virtuelles Java, etc. Le principe est simple : sur un appel à `switch_to(...)`, votre OS empile tous les registres sur le haut de pile puis passe la main au contexte passé en paramètre ; les registres de ce dernier sont dépilés et restaurés avant que celui-ci reprenne son exécution.

Exercice : Dispatcher v2 (le retour)

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

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

Vérifiez le bon fonctionnement pas à pas du programme.

Chapitre 4

Ordonnanceur collaboratif

4.1 Ce que vous allez apprendre dans ce chapitre

OS / Concurrency : Concepts.

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

OS / Concurrency : Skills.

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

OS / Scheduling and Dispatch : Concepts.

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

SF / Resource Allocation and Scheduling : Skills.

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

4.2 Passons à un nombre de processus supérieur à 2

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

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

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

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

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

Exercice : Réalisation de l'ordonnanceur collaboratif

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

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

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

Question 4-4 Créez la fonction `init_pcb` dans le fichier `sched.c` qui initialise la structure définie ci-dessus. Attention à l'initialisation du pointeur de pile des PCB.

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

- `void create_process(func_t f, void* args, unsigned int stack_size)` Cette fonction alloue un nouveau PCB, l'ajoute à la liste chaînée des PCBs, et l'initialise en appelant `init_pcb` ;
- `void start_current_process()` est appelée pour lancer un processus (pour la première fois). Elle appelle la fonction qui a été donnée comme point d'entrée du processus (le paramètre `f` de la fonction `create_process()`). L'intérêt de cette fonction sera surtout visible quand vous gérerez la terminaiason des processus. Indice : il est plus simple d'initialiser le pointeur d'instruction courante des PCB avec l'adresse de cette fonction que de gérer des cas conditionnels dans `ctx_switch()`.
- `void elect()` choisit le prochain processus et fait pointer la variable globale `current_process` sur son PCB. Elle ne sauvegarde ni ne restaure les contextes d'exécution ! C'est ctx_switch() qui s'en chargera.
- `void start_sched()` initialise quelques variables globales.

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

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

```
#include "sched.h"
#include "hw.h"

void
funcA()
{
  int cptA = 0;

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

void
funcB()
{
  int cptB = 1;

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

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

  start_sched();
  ctx_switch();

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

FIGURE 4.1 – Ce code doit tourner sans modification

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

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

4.3 Terminaison des processus

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

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

Chapitre 5

Ordonnanceur préemptif

5.1 Ce que vous allez apprendre dans ce chapitre

OS / Concurrency : Concepts.

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

OS / Concurrency : Skills.

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

OS / Scheduling and Dispatch : Concepts.

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

OS / Scheduling and Dispatch : Skills.

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

SF / Resource Allocation and Scheduling : Skills.

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

5.2 Ordonnancement sur interruption

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

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

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

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

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

Exercice : Première version de l'ordonnanceur préemptif

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

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

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

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

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

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

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

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

Explications :

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

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

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

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

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

5.3 Gestion des sections critiques

Exercice : Protection des structures de données partagées

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

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

Deuxième partie

Semaines 45 à 52 : projet (en hexanômes)

Chapitre 6

Présentation du projet

6.1 Objectifs

L'objectif général de ce projet est d'implémenter certaines fonctionnalités des systèmes d'exploitation et en acquérir les concepts. Plus concrètement, 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 cela, vous pouvez :
  - Vous inspirer d'une ou plusieurs boîtes rouges de la figure 1.1.
  - Vous pouvez implémenter une fonctionnalité de votre choix, en vous inspirant par exemple de l'annexe B.

  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 6.2 pour les détails du rendu. N'hésitez pas à réaliser des parties complémentaires d'autres hexanômes et à mettre le travail en commun.

Certains travaux d'implémentation sont obligatoirement à faire au sein de votre mini-OS (enfin vous pouvez implémentez un nouvel ordonnanceur dans le noyau Linux mais en quelques séances de Projet, ça risque d'être compliqué). D'autres peuvent être faits au-dessus de Linux ou votre mini-OS. Pour ceux qui implémenteront des choses au-dessus de Linux et qui ont besoin d'installer des paquets, voyez le chapitre 12.

6.2 Déroulement et rendu

Le projet dure 7 séances dont 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.

6.3 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 7

Appels système

7.1 Ce que vous allez apprendre dans ce chapitre

OS / Operating System Principles : Concept

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

OS / Operating System Principles : Skills.

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

7.2 Principe des appels système

Un appel système (en anglais, *system call*, abrégé en *syscall*) est une 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 nécessitant certains *privilèges*. Lisez l'encart ci-dessous pour un rappel de ce que sont les privilèges. La norme POSIX spécifie par exemple les appels systèmes `fork()` ou encore `read()`.

Ce chapitre vous guide dans l'implémentation d'un mécanisme d'appels système. Le principe permettant d'implémenter ces appels système est relativement simple :

— Lorsqu'un appel système est réalisé par un programme utilisateur, la fonction appelée doit déclencher une *interruption logicielle* grâce à l'instruction assembleur `SWI` ;

— le noyau traite cette interruption dans un traitant d'interruption, avec les privilèges noyau.

Les appels système que vous allez implémenter sont donnés dans la table 7.1.

7.3 Exercice : Appel système, sans paramètre

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

Modes d'exécution et privilèges

Un programme peut s'exécuter avec des *privilèges* plus ou mains 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* utilsateur ou en mode privilégié.
Chez ARM, il existe un mode utilisateur – le mode `USER` – mais plusieurs mode privilégiés :

- le mode `System`, dans lequel devrait s'exécuter le noyau.
- plusieurs modes pour gérer les exceptions :
  - le mode `SVC`. C'est dans ce mode que votre noyau commence à s'exécuter. C'est aussi le mode dans lequel se place le processeur lorsqu'il reçoit une interruption logicielle (`SWI` – *SoftWare Interrupt*)
  - le mode `Abort` dans lequel se place le CPU lorsque survient une erreur d'accès aux données (voyez le chapitre 8.
  - les modes `IRQ` et `FIQ` (*a priori*, vous n'utilisez pas ce dernier) dans lesquels se place automatiquement le CPU lorsqu'il reçoit une interruption du matériel (autre que la MMU pour signaler une erreur d'accès).

NB : dans le code d'initialisation du noyau (fichier `vectors.s`) que l'on vous a fournit, le processeur ne passe jamais en mode `System`. Vous pouvez le faire, ou non.
Par exemple, changer le registre de statut du processeur nécessite d'exécuter une instruction `msr`, ce qui n'est possible que si le processeur est dans un mode privilégié.

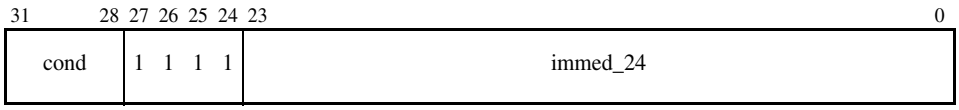

FIGURE 7.1 – Format de l'instruction SWI

Nom	Numéro	Description
`void sys_reboot()`	1	Le système redémarre
void sys_wait(unsigned int nbQuantums)	2	Le processus qui effectue cet appel attend `nbQuantums` avant d'être à nouveau éligible par l'ordonnanceur.

TABLE 7.1 – Les appels système que vous devez implémenter

Question 7-1 Déclarez et définissez la première fonction de la liste des appels système de la table 7.1. Pour le moment, elle ne fait que déclencher une interruption logicielle au moyen de l'instruction assembleur `SWI`, sans opérandes, en indiquant juste à GCC que le registre `lr` va être modifié par l'instruction assmenbleur : `__asm(‘‘SWI 0’’ : : : "lr")`. Ne vous occupez pas du résultat de l'exécution de cette instruction pour l'instant.

Question 7-2 Un traitant d'interruption logicielle vous est donné figure 7.2. Copiez le dans le fichier `vectors.s` et déclarez la fonction `SWIHandler` comme il convient dans des fichiers `syscall.c`/`syscall.h`. Laissez son corps vide pour le moment mais vérifiez en exécutant votre code pas à pas que l'exécution de `sys_reboot()` amène bien le processeur à exécuter cette fonction.

Question 7-3 Afin que votre mécanisme puisse traiter plus d'un appel système, votre fonction `SWIHandler()` doit récupérer le numéro de l'appel système spécifié dans le table 7.1, puis appeler une fonction qui réalise l'appel système associé à ce numéro. Pour passer le numéro de l'appel au noyau, copiez-le dans le registre R0 avant d'exécuter “SWI 0” : `__asm("mov r0, %0" : : "r"(numSysCall) : "r0")`. Implémentez ce mécanisme pour la valeur 1 : `SWIHandler()` appelle une fonction nommée `doSysCallReboot()` si le numéro d'interruption logicielle est 1.

Redémarrage du microcontrôleur

Redémarrer logiciellement le système entier – noyau + matériel – n'est pas si simple. On pourrait simplement faire brancher le CPU à l'adresse 0x8000 mais on ne sait pas dans quel état les périphériques seraient laissés. Ce qu'on veut, c'est envoyer le signal `reset` au CPU. Pour cela, les instructions ci-dessous configurent le watchdog du microcontroller (Broadcom BCM2835) puis attend 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);
```

```
swi:
    b SWIHandler
```

FIGURE 7.2 – Traitant pour les interruptions logicielles

7.4 Exercice : Appel système, avec paramètres

Afin de pallier aux limitations de l'appel système sans paramètres, vous allez ajouter le support de ceux-ci. Le passage des paramètres au noyau va se faire simplement via les registres généraux.

Question 7-4 Rappelez-vous : qu'appelle-t-on un quantum de temps ?

Question 7-5 Implémentez le deuxième appel système de la table 7.1.

7.5 Exercice : Non-Preemption du noyau

Question 7-6 Décrivez le problème posé potentiellement par l'interruption de votre noyau en cours de traitement d'appel système.

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

Chapitre 8

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

Mais sur les processeurs ARM, il est très difficile de dissocier les deux pour une bonne raison : il est impossible de remplir la TLB via l'OS. Seul le matériel (la MMU) peut le faire, en parcourant la table des pages. On y revient dans la suite...

Les 4 prochaines sections donnent des détails sur le mécanisme de pagination et comment gérer cela sur le processeur du Raspberry Pi ; les exercices commencent ensuite.

8.1 Rappels sur la mémoire virtuelle

La figure 8.1 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 *addresse 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).

8.2 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/TTBR0`.

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

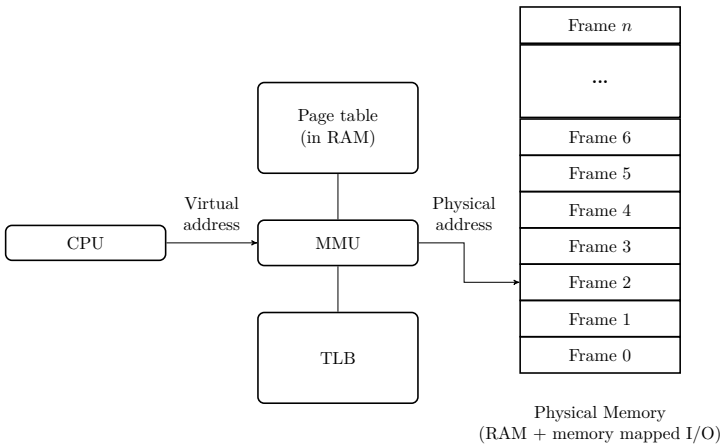

FIGURE 8.1 – Schéma simpliste de la traduction d'une adresse virtuelle en une adresse physique

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

8.3 La MMU

8.3.1 Activation de la MMU

Les instructions nécessaires pour configurer et activer la MMU vous sont données figure 8.2. 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 8.3.2)
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`.

8.3.2 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 << (32 - N)` sont traduites par la table des pages pointées par `TTBR0` ;
— les adresses comprises entre `1 << (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 8.2 – 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`.

8.4 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 8.3. Attention l'implémentation ne gère ni sections ni supersections, uniquement des pages (de taille 4Ko).

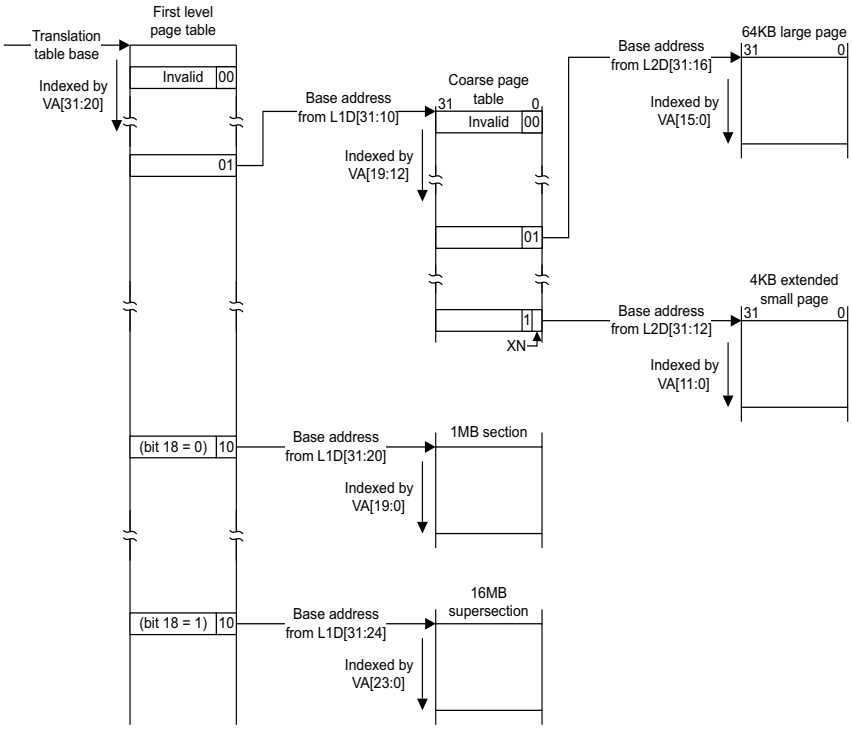

FIGURE 8.3 – Vue générale de la translation d'adresse en mode ARMv6

La traduction d'adresse effectuée est récapitulée, de manière précise, par la figure 8.4

8.4.1 Table des pages de niveau 1

Le format d'une entrée de la table de niveau 1 (une PDE) est détaillé dans la figure 8.5. En fonction des deux bits de poids faibles, l'entrée prend la forme d'une des deux premières lignes (dans votre impélementation, 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 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.

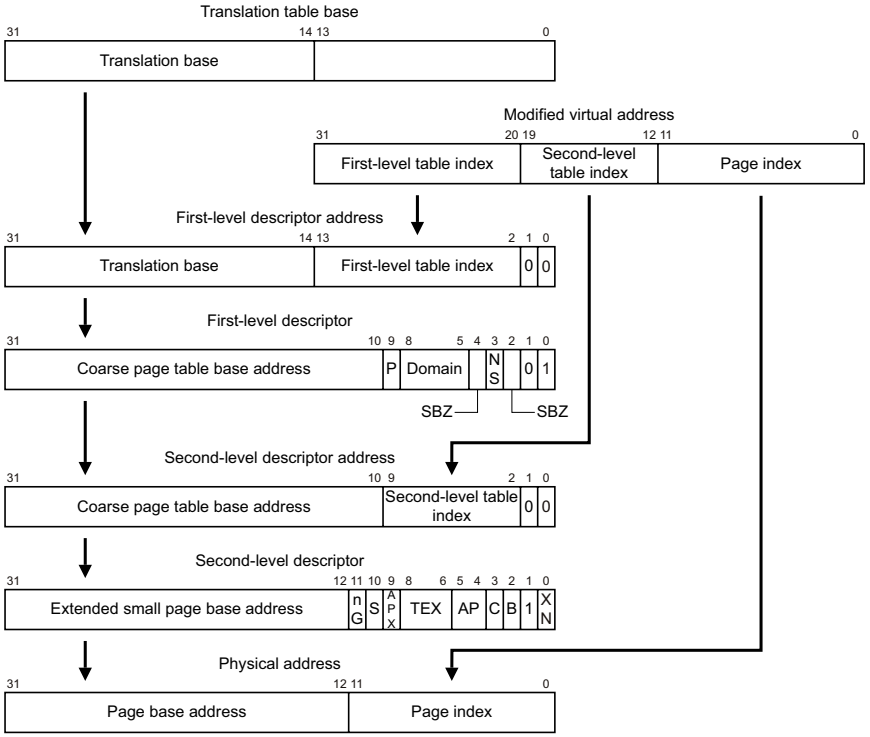

FIGURE 8.4 – Translation d'adresse en mode ARMv6, pages de 4KB

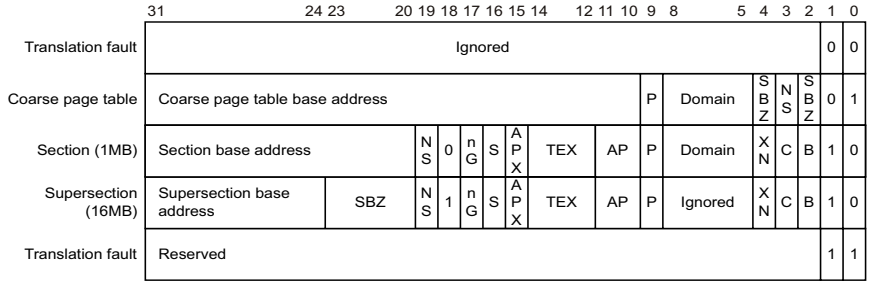

FIGURE 8.5 – Une entrée de la table des pages de niveau 1

8.4.2 Tables des pages de niveaux 2

Le format d'une entrée d'une table de niveau 2 est détaillé dans la figure 8.6. 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 gére 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).

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 8.1. 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 8.1 – Permissions d'accès

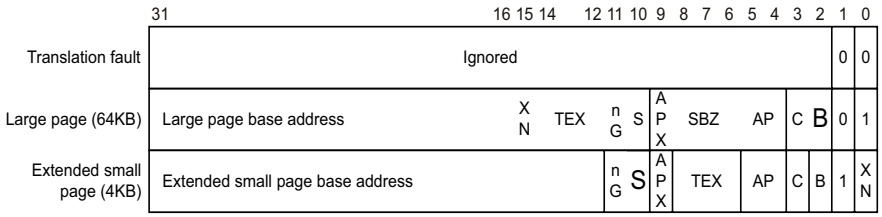

FIGURE 8.6 – Une entrée d'une table des pages de niveau 2

8.5 Organisation de la mémoire physique

L'organisation de la RAM du Raspberry Pi est illustrée par la figure 8.7. On y voit notamment qu'une partie de l'espace d'adressage est dédié aux périphériques (rappelez-vous la 3IF...). À cette organisation, il faut ajouter :

— À l'adresse 0, on trouve la table des vecteurs d'interruption
— À l'adresse 0x8000, le code de boot du noyau (fichier `vectors.s`)
— À l'adresse 0x38000 la pile (descendante !) du mode FIQ
— À l'adresse 0x40000 la pile (descendante !) du mode IRQ
— À l'adresse 0x48000 la pile (descendante !) du mode SVC

8.6 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

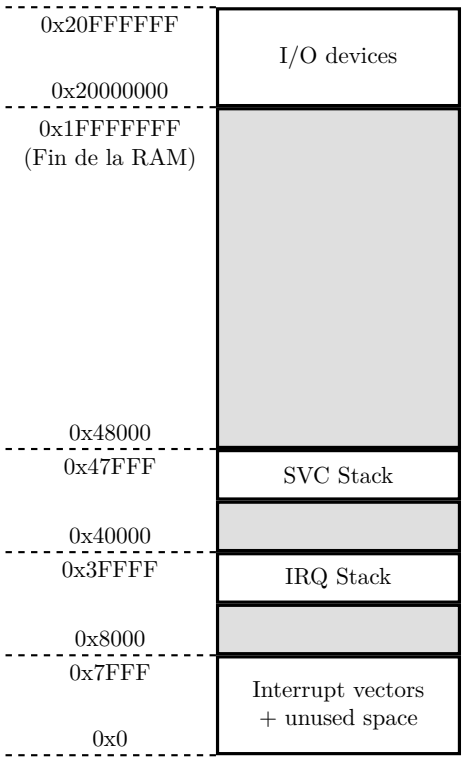

FIGURE 8.7 – Organisation de l'espace d'adressage physique

La série de question ci-dessous vous guide dans l'implémentation de ce mécanisme.

Question 8-1 Votre implémentation se fera principalement dans un fichier `vmem.c`. Créez ce fichier et modifiez le Makefile pour qu'il le compile.

Question 8-2 En vous basant sur les explications des sections précédentes, en particulier la figure 8.4 :
- 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_COUN` 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_COUN` qui vaut cette valeur.
- Définissez la macro `FIRST_LVL_TT_SIZE`, valant la taille en octets de la table de niveau 1.
- Définissez la macro `TOTAL_TT_SIZE` qui vaut la taille de la table des pages en entier.

Question 8-3 À quelle adresse allez-vous placer la table des page de niveau 1 de votre OS ? Ou seront placées les tables de niveau 2 ?

Question 8-4 En vous basant sur les explications des sections précédentes, en particulier la figure 8.6, 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 8-5 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 8-6 Implémentez la fonction `unsigned int init_kern_translation_table(void)` qui initialise la table des pages de l'OS. Attention, étant donné que vous ne relocalisez pas le code et les données statiques, votre table des pages doit traduire une addresse logique par elle-même pour toutes les adresses logiques auxquelles on trouve les données du noyau, soit entre 0 et la fin de la table des pages. 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. Un plan d'adressage facile à mettre en place parce que les adresses tombent aux bons endroits, est :

- adresse logique = adresse physique pour toute adresse physique entre 0x0 et 0x500000 (cette adresse est un peu grande, on n'a pas besoin de tout ça pour mettre les données noyau mais ça permet de tomber juste sur le nombre d'entrées dans la table de niveau 1).
- adresse logique = adresse physique pour toute adresse physique entre 0x20000000 et 0x20FFFFFF
- défaut de traduction sinon (pour le moment).

Question 8-7 Configurez et activez la MMU à l'aide des fonctions données dans la figure 8.2 puis testez votre mécanisme de pagination.

8.7 Exercice : allocation dynamique de pages

Les questions suivantes vous guident dans l'implémentation des deux fonctions suivantes :

- `uint8_t* vMem_Alloc(unsigned int nbPages)` qui alloue `nbPages` consécutives dans l'espace d'adressage virtuel de votre OS.
- `void vMem_Free(uint8_t* ptr, unsigned int nbPages)` qui libère les `nbPages` pages de de mémoire virtuelle pointées par `ptr`.

Afin que votre OS sache quelles pages de mémoire physiques (les *frames*) sont encore disponibles, il faut le doter d'une table d'occupation des frames. Chaque i-eme champ de cette table est un `uint8_t` indiquant si la i-eme frame est libre (0) ou non (1). Lors de l'allocation, votre code devra parcourir cette table pour trouver `nbPages` frames libres. Lors de la libération, il devra marquer les frames libérées comme libres.

Question 8-8 En regardant la figure 8.7, 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 8-9 Du coup, quelle taille fait la table d'occupation des frames ?

Question 8-10 À quelle adresse allez-vous placer la table d'occupation des frames ?

Question 8-11 Initialisez la table d'occupation des frames.

Question 8-12 Implémentez `vMem_alloc()` et `vMem_free()`.

8.8 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'insruction 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 8.4 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 8-13 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.

Question 8-14 Remplacez l'allocateur utilisé dans votre ordonnanceur par celui que vous venez d'implémenter.

Question 8-15 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 8-16 Votre traitant d'interruption doit gérer les fautes d'accès. Par exemple en terminant le processus fautif.

8.9 Annexe : Détail du c1/Control register

Explication de la valeur du registre c1/Control register, étant donné que $75864189 = 0 \times 485987d = (00000100100000101100101000111101)_2$

- 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**

8.10 Annexes : la protection chez ARM

8.10.1 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.

Chapitre 9

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 10

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.

10.1 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.

10.2 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/.

10.3 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 printf...\n");
```

10.4 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.

10.5 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`.

10.6 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.

10.7 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.

10.8 Jeux : casse-briques, tétris etc.

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

10.9 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.

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

Un sémaphore est manipulé par les deux actions atomiques suivantes :

— `sem_down()` (traditionnellement aussi nommée `wait()` ou `P()`). Cette action décrémente le compteur associé au sémaphore. Si sa valeur est négative, le processus appelant se bloque dans la file d'attente.

— `sem_up()` (aussi nommée `signal()`, `V()`, ou `post()`) Cette action incrémente le compteur. Si le compteur est négatif ou nul, un processus est choisi dans la file d'attente est devient actif.

Deux utilisations sont faites des sémaphores :

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

> **Rappel de cours**
>
> Typiquement le sémaphore est initialisé au nombre de processus pouvant concurremment accéder à la ressource (par exemple 1) et chaque accès à la ressource est encadré d'un couple
>
> ```
> sem_down(S) ;
>   <accès à la ressource>
> sem_up(S) ;
> ```

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

> **Rappel de cours**
>
> (Par exemple un processus 2 attend la terminaison d'un premier processus pour commencer.) On associe un sémaphore à l'événement, par exemple `findupremier`, initialisé à 0 (l'événement n'a pas eu lieu) :
>
> ```
> Processus 1 :
>     <action 1>
>     sem_up(findupremier) ;
>
> Processus 2 :
>     sem_down(findupremier) ;
>     <action 2>
> ```

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

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

struct sem_s mutex, vide, plein;

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

void producteur (void)
{
  objet_t objet ;

  while (1) {
    produire_objet(&objet);           /* produire l'objet suivant */
    sem_down(&vide);                  /* dec. nb places libres */
```

```
    sem_down(&mutex);                /* entree en section critique */
    mettre_objet(objet);             /* mettre l'objet dans le tampon */
    sem_up(&mutex);                  /* sortie de section critique */
    sem_up(&plein);                  /* inc. nb place occupees */
  }
}

void consommateur (void)
{
  objet_t objet ;

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

Le classique producteur consommateur

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

Exercice : Implémentation des sémaphores

Question 10-1 Donnez la déclaration de la structure de donnée associée à un sémaphore.

Question 10-2 Proposez une implantation de la primitive

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

Question 10-3 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 10-4 Proposez une implantation des deux primitives

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

10.10 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 débloquer un contexte en attente du verrou. De manière évidente, les verrous peuvent être simulés par des sémaphores dont la valeur initiale du compteur serait 1.

Exercice : Le dîner des philosophes

L'académique et néanmoins classique problème des philosophes est le suivant : cinq philosophes attablés en cercle autour d'un plat de spaghettis mangent et pensent alternativement sans fin (faim ?). Une fourchette est disposée entre chaque couple de philosophes voisins. Un philosophe doit préalablement s'emparer des deux fourchettes qui sont autour de lui pour manger.

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

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

Question 10-5 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 10-6 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 11

Allocation dynamique de mémoire

11.1 Une première bibliothèque standard

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

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

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

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

11.1.1 Principe

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

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

11.1.2 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.

11.1.3 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.

11.2 Optimisations de la bibliothèque

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

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

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

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

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

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

Chapitre 12

Linux sur Raspberry Pi

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

12.1 Installation et exécution de Linux

Un Raspberry démmare 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 de la façon suivante :

— Téléchargez l'image au bout du lien suivant : TODO

— Copiez-là sur la carte SD :

Sous Linux dd TODO

Sous MacOS dd TODO

12.2 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

12.3 Installation de logiciel

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

12.4 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. Typoiquement, voud devez observer l'effet d'un changement de la niceness, d'un changement d'ordonnanceur (utilisation de l'ordonnaceur 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

Code fourni

Chapitre 13

Interruptions

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

— le timer est réglé pour 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 `vectors.s`.

Chapitre 14

Gestionnaire de mémoire physique simple

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

Annexe A

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 B

Competences

Les pages ci-dessous listent les notions (*Topics*) et compétences (*Skills*) auxquelles on s'intéresse dans l'unité d'enseignement SEA. En plus des compétences détaillées dans les chapitres correspondant aux parties grises de la figure 1.1, vous devez démontrer l'usage de 3 compétences choisies dans cette liste.

OS/Overview of Operating Systems

[2 Core-Tier1 hours]

Topics:

- Role and purpose of the operating system
- Functionality of a typical operating system
- Mechanisms to support client-server models, hand-held devices
- Design issues (efficiency, robustness, flexibility, portability, security, compatibility)
- Influences of security, networking, multimedia, windowing systems

Learning Outcomes:

1. Explain the objectives and functions of modern operating systems. [Familiarity]
2. Analyze the tradeoffs inherent in operating system design. [Usage]
3. Describe the functions of a contemporary operating system with respect to convenience, efficiency, and the ability to evolve. [Familiarity]
4. Discuss networked, client-server, distributed operating systems and how they differ from single user operating systems. [Familiarity]
5. Identify potential threats to operating systems and the security features design to guard against them. [Familiarity]

OS/Operating System Principles

[2 Core-Tier1 hours]

Topics:

- Structuring methods (monolithic, layered, modular, micro-kernel models)
- Abstractions, processes, and resources
- Concepts of application program interfaces (APIs)
- The evolution of hardware/software techniques and application needs
- Device organization
- Interrupts: methods and implementations
- Concept of user/system state and protection, transition to kernel mode

Learning Outcomes:

1. Explain the concept of a logical layer. [Familiarity]
2. Explain the benefits of building abstract layers in hierarchical fashion. [Familiarity]
3. Describe the value of APIs and middleware. [Assessment]
4. Describe how computing resources are used by application software and managed by system software. [Familiarity]
5. Contrast kernel and user mode in an operating system. [Usage]
6. Discuss the advantages and disadvantages of using interrupt processing. [Familiarity]
7. Explain the use of a device list and driver I/O queue. [Familiarity]

OS/Concurrency

[3 Core-Tier2 hours]

Topics:

- States and state diagrams (cross-reference SF/State and State Machines)
- Structures (ready list, process control blocks, and so forth)
- Dispatching and context switching
- The role of interrupts
- Managing atomic access to OS objects
- Implementing synchronization primitives
- Multiprocessor issues (spin-locks, reentrancy) (cross-reference SF/Parallelism)

Learning Outcomes:

1. Describe the need for concurrency within the framework of an operating system. [Familiarity]
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. [Familiarity]
4. Explain the different states that a task may pass through and the data structures needed to support the management of many tasks. [Familiarity]
5. Summarize techniques for achieving synchronization in an operating system (e.g., describe how to implement a semaphore using OS primitives). [Familiarity]
6. Describe reasons for using interrupts, dispatching, and context switching to support concurrency in an operating system. [Familiarity]
7. Create state and transition diagrams for simple problem domains. [Usage]

OS/Scheduling and Dispatch

[3 Core-Tier2 hours]

Topics:

- Preemptive and non-preemptive scheduling (cross-reference SF/Resource Allocation and Scheduling, PD/Parallel Performance)
- Schedulers and policies (cross-reference SF/Resource Allocation and Scheduling, PD/Parallel Performance)
- Processes and threads (cross-reference SF/Computational paradigms)
- Deadlines and real-time issues

Learning Outcomes:

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. [Usage]
2. Describe relationships between scheduling algorithms and application domains. [Familiarity]
3. Discuss the types of processor scheduling such as short-term, medium-term, long-term, and I/O. [Familiarity]
4. Describe the difference between processes and threads. [Usage]
5. Compare and contrast static and dynamic approaches to real-time scheduling. [Usage]
6. Discuss the need for preemption and deadline scheduling. [Familiarity]
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. [Usage]

OS/Memory Management

[3 Core-Tier2 hours]

Topics:

- Review of physical memory and memory management hardware
- Working sets and thrashing
- Caching (cross-reference AR/Memory System Organization and Architecture)

Learning Outcomes:

1. Explain memory hierarchy and cost-performance trade-offs. [Familiarity]
2. Summarize the principles of virtual memory as applied to caching and paging. [Familiarity]
3. Evaluate the trade-offs in terms of memory size (main memory, cache memory, auxiliary memory) and processor speed. [Assessment]
4. Defend the different ways of allocating memory to tasks, citing the relative merits of each. [Assessment]
5. Describe the reason for and use of cache memory (performance and proximity, different dimension of how caches complicate isolation and VM abstraction). [Familiarity]
6. Discuss the concept of thrashing, both in terms of the reasons it occurs and the techniques used to recognize and manage the problem. [Familiarity]

OS/Security and Protection

[2 Core-Tier2 hours]

Topics:

- Overview of system security
- Policy/mechanism separation
- Security methods and devices
- Protection, access control, and authentication
- Backups

Learning Outcomes:

1. Articulate the need for protection and security in an OS (cross-reference IAS/Security Architecture and Systems Administration/Investigating Operating Systems Security for various systems). [Assessment]
2. Summarize the features and limitations of an operating system used to provide protection and security (cross-reference IAS/Security Architecture and Systems Administration). [Familiarity]
3. Explain the mechanisms available in an OS to control access to resources (cross-reference IAS/Security Architecture and Systems Administration/Access Control/Configuring systems to operate securely as an IT system). [Familiarity]
4. Carry out simple system administration tasks according to a security policy, for example creating accounts, setting permissions, applying patches, and arranging for regular backups (cross-reference IAS/Security Architecture and Systems Administration). [Usage]

OS/Virtual Machines

[Elective]

Topics:

- Types of virtualization (including Hardware/Software, OS, Server, Service, Network)
- Paging and virtual memory
- Virtual file systems
- Hypervisors
- Portable virtualization; emulation vs. isolation
- Cost of virtualization

Learning Outcomes:

1. Explain the concept of virtual memory and how it is realized in hardware and software. [Familiarity]
5. Differentiate emulation and isolation. [Familiarity]
6. Evaluate virtualization trade-offs. [Assessment]
2. Discuss hypervisors and the need for them in conjunction with different types of hypervisors. [Usage]

OS/Device Management

[Elective]

Topics:

- Characteristics of serial and parallel devices
- Abstracting device differences
- Buffering strategies
- Direct memory access
- Recovery from failures

Learning Outcomes:

1. Explain the key difference between serial and parallel devices and identify the conditions in which each is appropriate. [Familiarity]
2. Identify the relationship between the physical hardware and the virtual devices maintained by the operating system. [Usage]
3. Explain buffering and describe strategies for implementing it. [Familiarity]
4. Differentiate the mechanisms used in interfacing a range of devices (including hand-held devices, networks, multimedia) to a computer and explain the implications of these for the design of an operating system. [Usage]
5. Describe the advantages and disadvantages of direct memory access and discuss the circumstances in which its use is warranted. [Usage]
6. Identify the requirements for failure recovery. [Familiarity]
7. Implement a simple device driver for a range of possible devices. [Usage]

OS/File Systems

[Elective]

Topics:

- Files: data, metadata, operations, organization, buffering, sequential, nonsequential
- Directories: contents and structure
- File systems: partitioning, mount/unmount, virtual file systems
- Standard implementation techniques
- Memory-mapped files
- Special-purpose file systems
- Naming, searching, access, backups
- Journaling and log-structured file systems

Learning Outcomes:

1. Describe the choices to be made in designing file systems. [Familiarity]
2. Compare and contrast different approaches to file organization, recognizing the strengths and weaknesses of each. [Usage]
3. Summarize how hardware developments have led to changes in the priorities for the design and the management of file systems. [Familiarity]
4. Summarize the use of journaling and how log-structured file systems enhance fault tolerance. [Familiarity]

OS/Real Time and Embedded Systems

[Elective]

Topics:

- Process and task scheduling
- Memory/disk management requirements in a real-time environment
- Failures, risks, and recovery
- Special concerns in real-time systems

Learning Outcomes:

1. Describe what makes a system a real-time system. [Familiarity]
2. Explain the presence of and describe the characteristics of latency in real-time systems. [Familiarity]
3. Summarize special concerns that real-time systems present, including risk, and how these concerns are addressed. [Familiarity]

OS/Fault Tolerance

[Elective]

Topics:

- Fundamental concepts: reliable and available systems (cross-reference SF/Reliability through Redundancy)
- Spatial and temporal redundancy (cross-reference SF/Reliability through Redundancy)
- Methods used to implement fault tolerance
- Examples of OS mechanisms for detection, recovery, restart to implement fault tolerance, use of these techniques for the OS's own services

Learning Outcomes:

1. Explain the relevance of the terms fault tolerance, reliability, and availability. [Familiarity]
2. Outline the range of methods for implementing fault tolerance in an operating system. [Familiarity]
3. Explain how an operating system can continue functioning after a fault occurs. [Familiarity]

OS/System Performance Evaluation

[Elective]

Topics:

- Why system performance needs to be evaluated (cross-reference SF/Performance/Figures of performance merit)
- What is to be evaluated (cross-reference SF/Performance/Figures of performance merit)
- Systems performance policies, e.g., caching, paging, scheduling, memory management, and security
- Evaluation models: deterministic, analytic, simulation, or implementation-specific
- How to collect evaluation data (profiling and tracing mechanisms)

Learning Outcomes:

1. Describe the performance measurements used to determine how a system performs. [Familiarity]
2. Explain the main evaluation models used to evaluate a system. [Familiarity]