

INSA Lyon
4ème année
Spécialité Informatique

**Gestion des ressources physiques des systèmes
informatiques**

Ordonnancement CPU

Kevin Marquet

Septembre 2013

Table des matières

Table des matières	2
1 Introduction	3
2 Prise en main de l'environnement de compilation et debug	5
2.1 Prise en main du debugger GDB	5
2.2 Cross-compilation	9
2.3 Émulation de la plateforme	10
2.4 Exécution sur la plateforme	11
3 Processus et ordonnancement	13
3.1 Retour à un contexte	13
3.2 Un dispatcher pour processus collaboratifs très simples	15
3.3 Un dispatcher pour processus collaboratifs moins simples	17
3.4 Création d'un ordonnanceur collaboratif	17
3.5 Ordonnancement sur interruptions	20
3.6 Synchronisation entre contextes	22
3.7 Prévention des interblocages	25

Chapitre 1

Introduction

Ce document décrit les premiers travaux pratiques associés au cours de systèmes d'exploitation avancés. Il guide l'implémentation d'un élément essentiel des noyaux de système d'exploitation à savoir un ordonnanceur, complété d'un mécanisme de synchronisation d'exclusion mutuelle entre processus. Ce micro-OS 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 : un debugger ainsi qu'un émulateur.

Attention, tous les développements effectués dans le cadre de cette partie du projet seront faits **exclusivement sous Linux**.

Chapitre 2

Prise en main de l’environnement de compilation et debug

Ce chapitre décrit la prise en main des outils permettant de compiler et debugger un programme destiné à s’exécuter sur la plateforme embarquée. En premier lieu, vous vous familiariserez avec le debugger GDB et comment s’en servir pour debugger 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 debugger ce programme.

2.1 Prise en main du debugger GDB

Wikipedia.fr :

Un débogueur (ou débogueur, de l’anglais *debugger*) est un logiciel qui aide un développeur à analyser les bugs d’un programme. Pour cela, il permet d’exécuter le programme pas-à-pas, d’afficher la valeur des variables à tout moment, de mettre en place des points d’arrêt sur des conditions ou sur des lignes du programme ...

Rappelez-vous... le debugger

En 3IF, dans le module d’architecture, vous avez utilisé un tel outil, mais il était intégré à l’IDE.

2.1.1 Petite démo

Compiler un programme pour le debugging Pour pouvoir lancer l’exécution d’un programme sous `gdb`, il faut “préparer” celui-ci, en utilisant l’option spéciale `-g` lors de la compilation.

Dans cet exemple, vous allez debugger le programme de la figure 2.1.1, une fois ce code copié dans le fichier `prog.c` :

On compile donc le programme à l’aide de la commande :

```
gcc -Wall -g prog.c -o prog
```

et lancez `gdb` sur ce programme ainsi :

```
#include <stdio.h>

int max(int a, int b)
{
    int c=a;
    if (a<b){ c=b; };
    return c;
}

int f(int x)
{
    return 2*x+max(x+4,2*x);
}

int main()
{
    printf("d et d\n", f(3), f(5));
    return 0;
}
```

FIGURE 2.1: prog.c

```
gdb prog
<< message d'accueil... >>
(gdb)
```

Exécuter le programme sous gdb On peut exécuter le programme à l'aide de la commande `run` de gdb :

```
(gdb) run
Starting program: ./prog
13 et 20
```

Program exited normally.

L'exécution se déroule comme si le programme tournait "normalement". Lorsque l'utilisateur de `gdb` a la main, il peut choisir de terminer la session en tapant `quit`.

On peut exploiter `gdb` pour examiner le programme à différentes étapes de son exécution : pour cela, il faut introduire des *points d'arrêt* où l'exécution s'interrompt.

Poser des points d'arrêt C'est la commande `breakpoint` qui permet d'indiquer des points d'arrêt. Vous pouvez fournir un numéro de ligne dans le code source, ou bien le nom d'une fonction (l'exécution s'interromptra alors à chaque appel à cette fonction).

Dès lors, si vous lancez l'exécution, `gdb` interrompt l'exécution du programme et redonne la main à l'utilisateur lorsqu'il rencontre un point d'arrêt :

```
(gdb) break f
Breakpoint 1 at 0x804840b: file code_gdb.c, line 15.
```

```
(gdb) run
Starting program: ./prog

Breakpoint 1, f (x=5) at code_gdb.c:15
15         return 2*x+max(x+4,2*x);
```

NB : vous pouvez également introduire des “*watchpoints*”, à l’aide de la commande `watch`, qui ont pour effet d’interrompre l’exécution lorsque la valeur d’une variable est modifiée : on “surveille” en quelque sorte cette variable.

Examiner la situation lors d’un point d’arrêt Lorsque l’exécution du programme est interrompue, vous pouvez examiner l’état de la mémoire à ce moment là, par exemple en affichant la valeur d’une variable à l’aide de la commande `print` :

```
(gdb) print x
$1 = 5
```

On peut également utiliser la commande `list` pour se remémorer l’endroit dans le code où l’exécution a été interrompue :

```
(gdb) list
10         return c;
11     }
12
13     int f(int x)
14     {
15         return 2*x+max(x+4,2*x);
16     }
17
18     int main()
19     {
```

Exécuter le programme La commande `step` permet d’avancer pas à pas dans l’exécution, afin de bien contrôler l’évolution du programme. Elle permet de passer à la ligne suivante dans le source :

```
(gdb) step
max (a=9, b=10) at code_gdb.c:5
5         int c=a;
(gdb) step
7         if (a<b){
(gdb) step
8             c=b;
(gdb) print c
$2 = 9
(gdb) step
10        return c;
(gdb) print c
$3 = 10
```

À noter qu'il existe aussi la commande `next`, qui elle ne “descend” pas dans les appels de fonctions ; ainsi, si l'on a interrompu l'exécution juste avant un appel de la forme `f(a,b)`, `next` relance le programme et l'interrompt après l'appel à `f()` (dans le code de la fonction appelante), alors que `step` s'arrête à la première ligne du code définissant la fonction `f`. Également, la fonction `stepi` exécute une instruction machine, par opposition à une ligne de code comme `step`.

L'instruction `continue`, elle, relance l'exécution jusqu'au prochain point d'arrêt.

Examiner la pile d'exécution : la commande `backtrace` permet d'afficher la pile d'exécution, indiquant à quel endroit l'on se trouve au sein des différents appels de fonctions. Ici, le processeur est en train d'exécuter la fonction `max`, qui a été appelée par `f`, elle-même invoquée par la fonction `main` :

```
(gdb) backtrace
#0  max (a=9, b=10) at code_gdb.c:10
#1  0x8048423 in f (x=5) at code_gdb.c:15
#2  0x804844f in main () at code_gdb.c:20
```

Effacer un point d'arrêt : `clear` en indiquant un numéro de ligne ou un nom de fonction, `delete` en indiquant le numéro du breakpoint (`delete` tout court efface – après confirmation – tous les points d'arrêt).

Remarque : pourquoi faut-il utiliser une option spéciale de `gcc` afin de pouvoir utiliser `gdb` ? Parce que de nombreuses informations inutiles lors de l'exécution du programme ne sont pas mises, par défaut : ainsi il est a priori inutile de savoir, au cours de l'exécution du programme, à quel endroit le processeur se trouve dans le code source, ou bien quel est le nom de la variable qu'on est en train de modifier. Par contre ce genre de renseignement est utile à `gdb` afin que l'utilisateur “s'y retrouve”. 1

2.1.2 Quelques commandes importantes sous gdb

Entre parenthèses, les abréviations que l'on peut utiliser à la place des commandes en toutes lettres.

`quit (q)` quitter `gdb`

`run (r)` lancer l'exécution

`break,watch,clear,delete (b,w,c1,d)` introduire un point d'arrêt, ou bien “surveiller” une variable

`step,next,continue (s,n,c)` avancer d'un pas (en entrant ou pas dans les sous-fonctions), relancer jusqu'au prochain point d'arrêt

`print,backtrace,list (p,bt,l)` afficher la valeur d'une variable, la pile d'exécution, afficher l'endroit où l'on se trouve dans le code. Notamment, `info registers` ou encore `print $<register_name>` permet d'afficher la valeur des registres du processeur.

`help <cmd>` afficher l'aide sur la commande `<cmd>`. Vous pouvez bien sûr taper `help help`.

2.1.3 Scripts gdb

Si vous lancez la commande `gdb -x <nom_fichier>.gdb <executable>`, `gdb` évaluera toutes les commandes `gdb` contenues dans le fichier `<nom_fichier>.gdb`. Cela est très utile

pour ne pas avoir à taper systématiquement les même commandes à chaque exécution, et rendre ainsi les sessions de debug plus efficaces (voir exemple 1 ci-dessous). Dans la suite du TP, un petit fichier contenant les commandes de base pour debugger votre programme vous est fourni. N'hésitez pas à aller le modifier !

Listing 2.1: Exemple 1 : afficher le contenu de `mavariab` chaque fois qu'on exécute la fonction `mafonction`

```
break mafonction
run
print mavariab
```

Une autre fonctionnalité très utile est de pouvoir définir vos propres commandes. Vous pouvez par exemple n'afficher des infos de debug que dans certaines conditions, ou afficher plein d'infos sans avoir besoin de retaper des dizaines de lignes de code. L'exemple 2 ci-dessous crée une commande `pregs` qui affichera le contenu des registres `r0` et `sp` à chaque fois que vous l'invoquerez.

Exemple 2 :

```
define pregs
  print "Contenu de r0:"
  print $r0
  print "Pointeur de pile:"
  print $sp
end
```

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 2.2. 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`.

Rappelez-vous... la cross-compilation

En 3if-Archi, vous pouvez, 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 `notmain.c`. Pour comprendre le boot du Raspberry Pi et comprendre pourquoi le processeur saute à cet endroit là, voyez l'encart page 12.

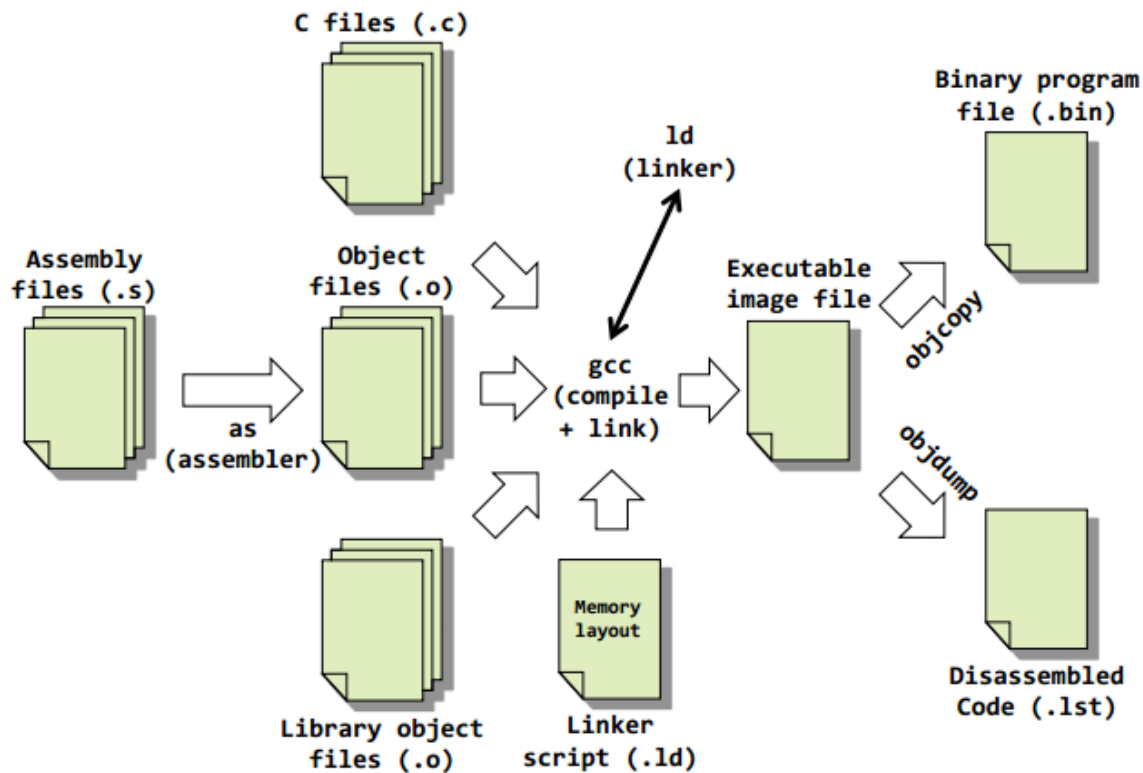


FIGURE 2.2: Chaîne de compilation

2.3 Émulation de la plateforme

Rappelez-vous... le debug 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

Voyez avez lu l'encart ci-dessous de rappel sur la cross-compilation ? Eh bien sur les Raspberry Pi, c'est pas pareil. Enfin, disons qu'on n'a pas pris les semaines nécessaires pour faire marcher le connecteur JTAG via les pins de la Raspberry Pi. Mais pas grave, on va voir une autre manière de développer. Vous vous doutez bien qu'on vous a 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)

L'émulateur qu'on va utiliser est donc un logiciel capable d'exécuter, sur le PC, le code binaire ARM. Et pour pouvoir debugger ce programme, l'émulateur va se laisser piloter par

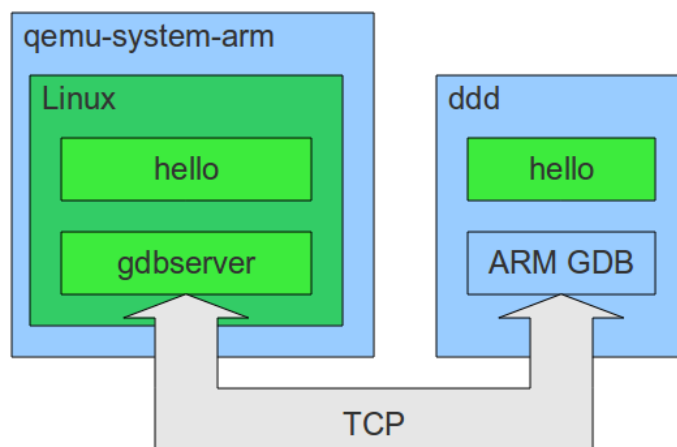


FIGURE 2.3: Remote debugging

GDB (via l'interface `gdbserver`) comme illustré par la figure 2.3. 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 1. Prise en main des outils

Téléchargez l'archive `exemple_gdb.tgz` sur <http://perso.citi-lab.fr/kmarquet/4if-GRP.html>.

Décompressez-la et placez-vous dedans. Éditez le fichier `notmain.c` pour observer le code. Tapez `'make'` pour compiler ce programme. Placez-vous dans le répertoire debug et exécutez `'run-qemu.sh'`. Dans un **autre** terminal, depuis ce même répertoire, tapez `debug.sh`. Exécutez le programme pas à pas, en essayant les différentes commandes de la Section 2.1.2. On remarque que le debugger est *arm-none-eabi-gdb*, autrement dit `gdb` compilé explicitement pour comprendre l'assembleur ARM.

2.4 Exécution sur la plateforme

Vous n'aurez pas besoin de faire ces manipulations pendant les 2 premières séances, mais vous aurez besoin d'un Raspberry Pi, une alimentation, une carte SD et un cordon d'alimentation.

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

Pour comprendre : le boot du Raspberry Pi

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

Chapitre 3

Processus et ordonnancement

À la fin de chaque exercice, pensez à copier le répertoire dans lequel vous travaillez, histoire de garder une trace de ce qui marche.

3.1 Retour à un contexte

Pour certaines applications, l'exécution du programme doit être reprise en un point particulier. Un point d'exécution est caractérisé par l'état courant de la pile d'appel et des registres du processeur ; on parle de *contexte*.

Contexte d'exécution

Cette section est un rappel du cours pour ce qui est des notions mais donne les détails (registres, principalement) pour ce qui est de l'architecture de la famille ARM11 (i.e. architecture 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

Le Raspberry Pi comprend un micro-contrôleur . Celui-ci contient un processeur ARM1176JZF. 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...

Grossièrement, lorsqu'une procédure est appelée, les registres du microprocesseur sont sauvés au sommet de la pile, puis les arguments sont empilés. Puis le processeur branche au code de la fonction appelée et sauvegarde l'adresse de retour (registre `lr`) sur la pile ; grâce à l'instruction `bl`). Ces conventions dépendent bien sûr du compilateur.

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.

Assembleur en ligne dans du code C

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

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

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

Par ailleurs, un autre moyen d'intégrer de l'assembleur dans du C est d'écrire des fichiers `.s` tels que `vectors.s`.

Exercice 2. Premières observations

Question 2.1

- 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.
- Dans quel sens croît la pile ?
- Observez l'utilisation du registre `lr`. Quelle instruction assembleur le met à jour implicitement ? Au besoin, re-lisez le rôle du registre `lr` dans l'encart page précédente.

3.2 Un dispatcher pour processus collaboratifs très simples

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'exécution, 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 `uint32* AllocateMemory(unsigned int size)`. Cette fonction alloue `size * 32` bits de données. Pour libérer la mémoire, utilisez `int FreeAllocatedMemory(uint32_t* address)`. Ces deux fonctions sont déclarées dans `allocateMemory.h`.

Exercice 3. Dispatcher simple

Question 3.1

- Mettez en place votre code de la façon suivante :
 1. Téléchargez l'archive `ordonnancement_collaboratif.tgz` sur <http://perso.citi-lab.fr/kmarquet/4if-GRP.html> ;
 2. Décompressez-la : `tar xzf ordonnancement_collaboratif.tgz` ;
 3. Le fichier `Makefile` fourni permet de compiler — tapez `'make'` — le fichier `notmain.c`. Il ne compile pas ; c'est normal, vous n'avez plus qu'à lire les questions qui suivent et implémenter ce qu'il faut !
- Éditez `notmain.c`. Que fait ce programme ?

Question 3.2

Définissez la structure de données `struct ctx_s` dans un fichier `process.h`.

Question 3.3

Déclarez la variable globale `current_ctx` qui *pointe* en permanence sur le contexte en cours d'exécution : soit `ctx_A`, soit `ctx_B` dans notre exemple.

Question 3.4

Définissez dans le fichier `process.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. 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.
- faites gaffe, les contextes pour nos mini-processus exécutant `ping()` et `pong()` sont déclarés dans `notmain.c` et ils 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 + d'un processus);

Question 3.5

Déclarez (dans un fichier `dispatcher.h`) et définissez (dans un fichier `dispatcher.c`) la primitive suivante qui permet de changer de contexte :

```
void 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.

Attention, pensez à relire la description du registre `lr` ! La seule instruction assembleur dont vous avez besoin est `mov`, illustrée précédemment.

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

Question 3.6

Compilez (vous aurez besoin de modifier le `makefile` pour cela) et exécutez pas à pas le programme. Vérifiez qu'il passe bien successivement d'un contexte à l'autre.

3.3 Un dispatcher pour processus collaboratifs moins simples

On complexifie : cette fois, il s'agit de passer d'un processus exécutant `funcA()` à un processus exécutant `funcB()` (voir figure 3.1).

NB : histoire d'éviter les copier-coller qui marchent mal depuis ce pdf, le fichier est disponible en ligne.

Remarquez que ces deux fonctions possèdent maintenant des variables locales, et effectuent des calculs. Il ne suffit donc plus de sauvegarder et restaurer `pc` et `sp` mais tous les registres.

Exercice 4. Dispatcher v2 (le retour)

Question 4.1

Faites les modifications nécessaires pour autoriser ce fonctionnement. Vérifiez le bon fonctionnement pas à pas du programme figure 3.1.

3.4 Création d'un ordonnanceur collaboratif

La primitive `switch_to` du mécanisme de coroutines impose au programmeur d'explicitement le nouveau contexte à activer. Par ailleurs, une fois l'exécution de la fonction associée à un contexte terminée, il n'est pas possible à la primitive `init_ctx()` d'activer un autre contexte, aucun autre contexte ne lui étant connu. À travers la série de question ci-après, vous allez donc déclarer et définir une nouvelle interface avec laquelle les contextes ne sont plus directement manipulés dans « l'espace utilisateur » :

```
int create_process(int stack_size, func_t f, void *args);
void yield();
```

La primitive `create_process()` ajoute à l'ancienne `init_ctx()` l'allocation dynamique la structure mémorisant le contexte. La primitive `yield()` permet au contexte courant de passer la main à un autre contexte, ce dernier étant déterminé par l'ordonnancement. Un des objectifs de cet ordonnancement est de choisir, lors d'un changement de contexte, le nouveau contexte 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. Les informations de contexte non présente dans le PCB seront sauvegardées, lors d'un changement de contexte, dans la pile d'exécution correspondant à ce PCB.

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

```

#include "process.h"
#include "dispatcher.h"

struct ctx_s ctx_A;
struct ctx_s ctx_B;
struct ctx_s ctx_init;

void
funcA ()
{
    int cptA = 0;

    while ( 1 ) {
        cptA ++;
        switch_to(&ctx_B);
    }
}

void
funcB ()
{
    int cptB = 1;

    while ( 1 ) {
        cptB += 2 ;
        switch_to(&ctx_A);
    }
}

//-----
int
notmain ( void )
{
    init_ctx(&ctx_B, funcB, STACK_SIZE);
    init_ctx(&ctx_A, funcA, STACK_SIZE);

    current_ctx = &ctx_init;
    switch_to(&ctx_A);

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

```

FIGURE 3.1: Ce code doit tourner sans modification

fonction appelée aura tout loisir pour effectuer une coercition de la structure pointée dans le type attendu.

Exercice 5. Réalisation de l'ordonnanceur collaboratif

Question 5.1

Remplacez les appels à `switch_to()` par des appels à `yield()` dans `notmain.c`, et supprimer les déclarations de `ctx_A` et `ctx_B`.

Question 5.2

Proposer un type de donnée pour l'état d'un processus.

Question 5.3

Renommez la structure de donnée `struct ctx_s` en `pcb_s` et modifiez là pour qu'elle ne contienne que les infos d'un PCB.

Question 5.4

Renommez la fonction `init_ctx()` en `init_pcb` et modifiez là pour qu'elle initialise la structure définie ci-dessus.

Question 5.5

Le fichier `sched.c` implémente l'ordonnanceur au travers des fonctions `create_process()` et `void yield()` décrites ci-dessus, ainsi que `void start_current_process()`. Quelques détails :

- `void create_process(func_t f, void* args)` Cette fonction alloue un nouveau PCB, l'ajoute à la liste chaînée des PCBs, et l'initialise en appelant `init_pcb`
- `void yield()` Cette fonction positionne la variable globale `next_running` qui est le prochain processus à s'exécuter et appelle `ctx_switch()` (voir la question suivante).
- `void start_current_process()` est appelée pour lancer un processus (pour la première fois). Elle positionne le contexte courant et appelle la fonction qui a été donnée comme point d'entrée du processus (le paramètre `f` de `create_process()`).

Définissez ces 3 fonctions. Assurez-vous que vous avez prévu le cas de la première invocation de `yield()`... Ah oui, également : modifiez `current_ctx` en `current_process` (et ajustez son type bien sûr).

Question 5.6

Modifiez `switch_to()` (qui devient `ctx_switch` donc) qui réalise le comportement décrit plus haut. Quelques détails :

- Cette fonction sauvegarde les registres du processus en cours d'exécution dans la pile d'exécution. Pour cela, vous pouvez utiliser l'instruction assembleur `push`.
- Cette fonction restaure le contexte du processus choisi par `yield()`. Pour cela, vous pouvez utiliser l'instruction assembleur `pop`.

- Si ce processus est activé pour la première fois, au lieu de revenir avec un `return;` la fonction appelle `start_current_context()` pour « lancer » la première exécution...

Attention, après que le registre de piles ait été initialisé sur une nouvelle pile d'exécution, les variables locales et les arguments de la fonction `ctx_switch()` sont inutilisables.

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 5.7

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

3.5 Ordonnancement sur interruptions

L'ordonnancement développé jusque ici est un ordonnancement avec partage volontaire du processeur : un contexte passe la main grâce à un appel explicite à `yield()`. Nous allons maintenant développer 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'interruptions ou *handler*). On vous fournit 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()` permet d'initialiser le matériel ; elle doit être invoquée pour mettre en place la génération des interruptions. Cette primitive configure un timer pour qu'il génère une interruption toute les 10ms.

Les deux primitives `DISABLE_IRQ()` et `ENABLE_IRQ()` permettent de délimiter des zones de code devant être exécutées de manière non interruptible.

`set_tick_and_enable_timer` permet quant à elle de ré-armer le timer. Attention : pensez à réarmer le timer, le matériel le désactive après chaque interruption. Attention (bis) : après cet appel, la *ligne* d'interruption reste désactivée et doit être activée par un `ENABLE_IRQ()` pour que la génération d'une interruption puisse avoir lieu.

La nouvelle interface que va fournir votre ordonnanceur est la suivante :

```
int create_process(int stack_size, func_t f, void* args);
void start_sched();
```

La fonction `start_sched()` va initialiser le matériel.

Exercice 6. Première version de l'ordonnanceur préemptif

(Rappelez vous...) Les attributs

En C, vous pouvez, dans la déclaration d'une fonction ou d'une variable, spécifier un ou plusieurs *attributs*. Un attribut est une information donnée au compilateur pour l'aider à optimiser sa génération de code, ou lui indiquer de compiler le code d'une certaine manière. L'attribut qui nous intéresse ici est l'attribut `naked`. Une fonction déclarée comme telle sera compilée sans les *prologue* et *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 dans les déclarations de traitant d'interruption. Un exemple : `void __attribute__((naked)) timer_handler();`
 En 3if-archi, on utilisait le mot-clef `__interrupt`, synonyme de `__attribute__((__interrupt__(TIMER0_A0_VECTOR)))`, supporté par mspgcc.

Important : Récupérez le bon fichier `vectors.s` en ligne. Celui-ci configure le vecteur

d'interruption pour que, sur une interruption timer (toutes les 10ms), le processeur saute à `ctx_switch()`.

Question 6.1

Proposez une implantation de `start_sched()`.

Question 6.2

Jusqu'à présent, le choix du processus (on parle de son *élection*) devant s'exécuter — on l'appelait précédemment `next_running` — se faisait *avant* d'appeler `ctx_switch()`. À présent, ce n'est plus possible : le processus en cours d'exécution pouvant se faire interrompre n'importe quand, vous devez d'abord sauvegarder le *contexte* interrompu avant d'éventuellement utiliser des registres pour exécuter du code tel que l'ordonnancement. Assurez-vous d'avoir compris ce que fait l'attribut `naked` (voyez l'encart 3.5) et effectuez les modifications suivantes :

- la fonction `ctx_switch()` doit être déclarée `naked` ;
- prenez cela en compte : votre code doit sauter au bon endroit à la fin de votre `ctx_switch()`.
- l'élection du prochain processus à s'exécuter doit être appelée dans `ctx_switch()`. Placez-le dans une fonction `void schedule()`.
- Pensez à ré-armer le timer !

Question 6.3

Dans les ARM pas tout jeunes, lorsqu'une interruption a lieu, le processeur se met dans un *mode d'exécution* particulier — le mode `fiq` dans notre cas. Cela a les conséquences suivantes :

- les *privilèges* changent (certaines instructions ne peuvent être exécutées dans certains modes d'exécution) ;
- le processeur utilise une **nouvelle version matérielle** de certains registres. Dans notre cas, `sp`, `lr`, `cpsr`.

- le processeur a sauvegardé l'adresse de l'instruction en cours d'exécution avant l'interruption dans le registre `lr` du mode `fiq`. Au début de votre fonction `ctx_switch()`, il vous faut donc ajouter 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. C'est dans la documentation...
- La deuxième instruction sauvegarde `lr` et le registre de statut `cpsr` dans la pile du mode d'exécution initial du processeur (mode `System`).
- La troisième passe en mode `System`. Les registres (y compris `sp...`) sont donc, après cela, ceux du processus interrompu.
- De même, l'instruction suivante vous permettra de terminer proprement votre `ctx_switch()` :

```
__asm("rfe fd sp!");
```

Cette instruction fait l'inverse de `srsdb` : elle charge dans `pc` la valeur située en haut de pile, et dans `cpsr` la valeur un mot avant le haut de pile (le processeur repasse donc en mode `fiq` si vous avez utilisé les 3 instructions données au précédent item... et que vous n'avez rien oublié sur la pile entre-temps!). NB : `sp` est mis à jour (utilisation du '!')

Prenez cela en compte dans votre implémentation de `ctx_switch()`. Euh... ça va ? Pas trop perdu ?

Exercice 7. Protection des structures de données partagées

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

Question 7.1

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

3.6 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 et devient actif.

Deux utilisations sont faites des sémaphores :

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

Rappel de cours

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

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

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

Rappel de cours

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

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

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

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

struct sem_s mutex, vide, plein;

sem_init(&mutex, 1);                          /* controle d'accès 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/vidé`; que constatez-vous? Votre implémentation peut-elle détecter de tels comportements?

Exercice 8.

Question 8.1

Donnez la déclaration de la structure de donnée associée à un sémaphore.

Question 8.2

Proposez une implantation de la primitive

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

Question 8.3

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

Exercice 9.

Proposez une implantation des deux primitives

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

3.7 Prévention des interblocages

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

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

L'interface de manipulation des verrous est la suivante :

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

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

Exercice 10.

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.

Exercice 11.

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.

Exercice 12.

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);
```