

Advanced Operating Systems

Privileges, system calls, context-switching, CPU scheduling, memory paging, memory allocation, isolation,
virtualisation, process synchronisation

Kevin Marquet

September 18, 2018

Contents

Contents	2
1 Introduction	4
2 Compiling, executing, and debugging	6
2.1 What you will learn in this chapter	6
2.2 Cross-compilation	6
2.3 Emulating the Raspberry Pi	7
2.4 Exercise: writing (a bit of) ARM assembly	12
2.5 Exercise : the <i>naked</i> attribute	13
2.6 Exercise : use the provided tests	13
2.7 Documentation	14
3 CPU execution modes	15
3.1 The ARM execution mode	15
3.2 Physical memory organisation	16
3.3 Exercise: changing the execution mode	17
4 System calls	19
4.1 What you will learn in this chapter	19
4.2 System calls, the principle	19
4.3 Exercise : a first system call that never returns	21
4.4 Exercise: a system cal that returns	22
4.5 Exercise : a system call with parameters passing	24
4.6 Exercise : a syscall that returns a value	25
5 Process dispatching	27
5.1 What you will learn in this chapter	27
5.2 Introduction	27
5.3 Exercise : a dispatcher for coroutines	28
5.4 Exercise: coroutines with local variables	30
6 Collaborative scheduling (round-robin)	32
6.1 Ce que vous allez apprendre dans ce chapitre	32
6.2 Scheduling	33
6.3 Exercise: round-robin scheduling	33
6.4 Exercise : process termination	34
6.5 Optional exercise : non explicit termination of a process	35
7 Preemptive scheduling	36
7.1 What you will learn in this chapter	36

7.2	Scheduling on interruption	37
7.3	Exercise: handling interruptions	38
8	Virtual memory management: paging	41
8.1	Virtual memory	41
8.2	The co-processor	41
8.3	The MMU	42
8.4	Paging	43
8.5	Exercise : paging without dynamic allocation	46
9	Dynamic allocation	50
9.1	Exercice : compréhension	50
9.2	Exercice : implémentation	51
10	Process isolation	52
10.1	Exercice : gérer une faute de traduction	52
10.2	Exercice : compréhension	52
10.3	Exercice : implémentation	54
11	Shared memory	55
12	Process synchronization	56
12.1	Prévention des interblocages	58
13	Allocation dynamique de mémoire	60
13.1	Une première bibliothèque standard	60
13.2	Optimisations de la bibliothèque	61
I	Annexes	62
A	Interruptions	63
B	Gestionnaire de mémoire physique simple	64
C	Installation des outils nécessaires à la réalisation du projet	65
C.1	Installation de Qemu pour Raspberry Pi	65
C.2	Installation de GCC pour ARM	65
C.3	Installation de GDB pour ARM	65
D	FAQ	67
E	Appendix about virtual memory	68
E.1	Detail concerning register <i>c1/Control register</i>	68
E.2	Isolation in ARM's processors	69
F	Aide-mémoire gdb	70
	Glossary	73

Chapter 1

Introduction

This document describes the practical works associated with the course "Advanced Operating Systems". It will guide you all along the implementation of a small operating system intended to be executed on a Raspberry Pi platform. The entire implementation will be made on top of a Linux distribution.

Some elements to help you read this document:

- When you encounter a [notion](#), you can click on the word in order to get a small definition of it ;
- A glossary at the end of the document group together all the [notions](#).

The graph on the Figure [1.1](#) gives an overview of the system you will implement. It is to be read in the following manner:

- Green nodes are small parts that you don't have to implement, we give them to you, because we are good to you.
- White nodes are parts you can expect to implement. You will be closely guided in their implementation.
- Red nodes are examples of optionnal nodes. You don't have to even try to implement them, but if you are passionate, dont hesitate.
- You can click on the links given inside each node in order to access directly the corresponding chapter. Each chapter list the concepts and skills associated to the chapter, describe how to implement a part of the OS, explain the concepts, and give links towards more detailed explanations.

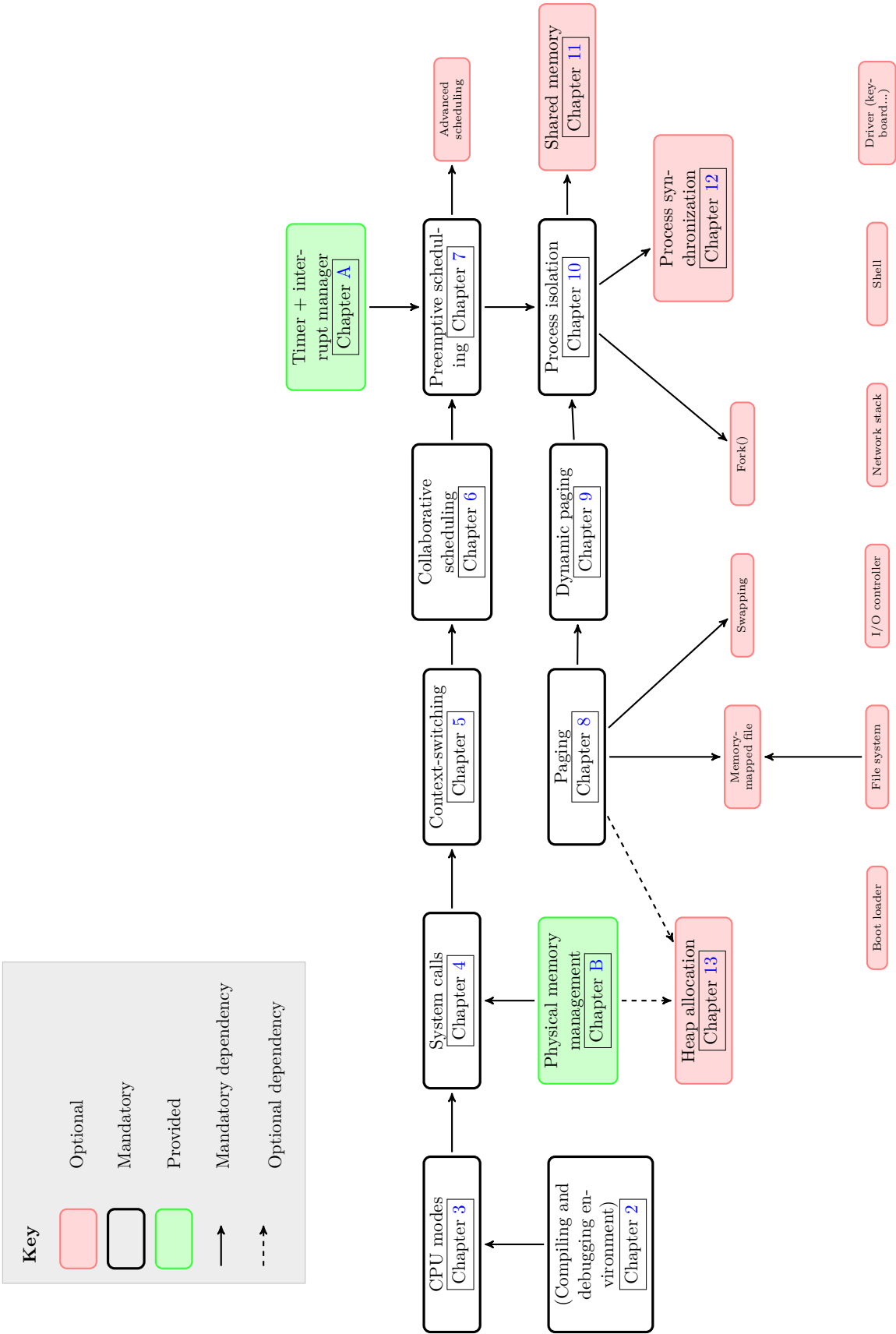


Figure 1.1: DAG detailing the practical works of the module.

Chapter 2

Compiling, executing, and debugging

This chapter describes how to use tools allowing to compile and debug programs that are to be executed on the embedded platform. First, you will familiarize yourself with the GDB debugger, then you will cross-compile a program so that it can execute on the platform, and at last how to debug this program.

2.1 What you will learn in this chapter

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.

See Figure 2.1. In the case of a compilation for the host machine — that is to say your PC, containing a processor Intel executing Linux —, the binary file will contain x86 assembly. But you will use a **cross-compiler** in order to produce ARM assembly for the Raspberry Pi. This cross-compiler is a port of the GCC compiler. You can either install it on your own machine, or work on the provided computers. On those machines, all tools are available in /opt/AOS but you have to add to your PATH the following repositories in order to be able to execute the tools :

- /opt/AOS/arm-none-eabi-gcc/bin

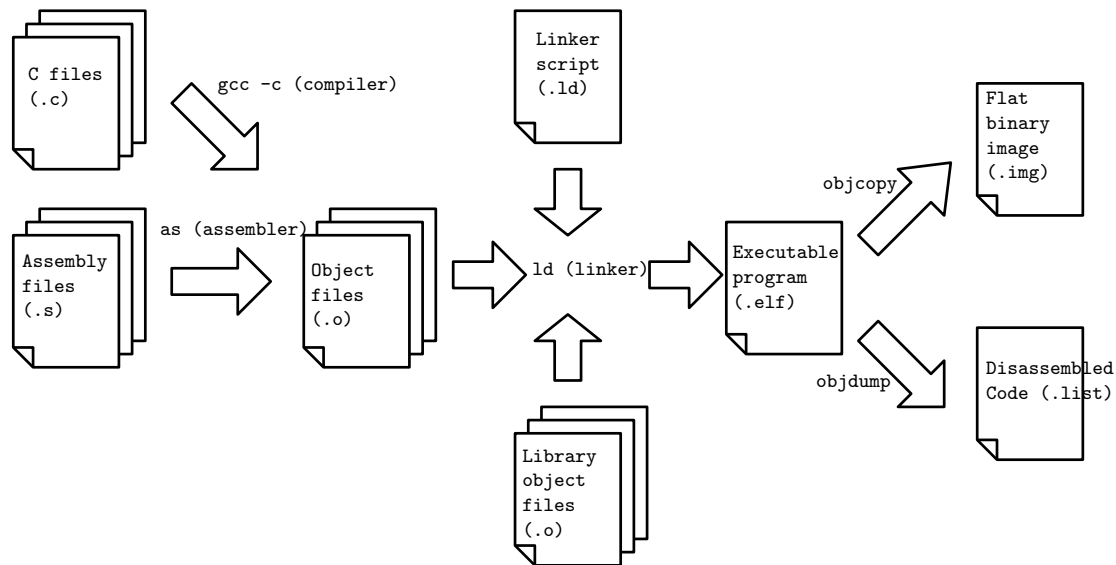


Figure 2.1: Compilation chain

- /opt/A0S/arm-none-eabi-gcc/arm-none-eabi/bin
- /opt/A0S/qemu-rpi/bin

Entry point of your code The entry point of your code, *i.e.* the place where the processor will jump after booting, is the function `kmain()`. To understand the boot of the Raspberry Pi, see the insert [4](#).

2.3 Emulating the Raspberry Pi

Our solution to avoid plugging the Raspberry Pi each time we want to test a single line of code is to *emulate* the RasPi. Wikipedia tells us:

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)

The emulator we will use is Qemu. It is a piece of software able to interpret, on the PC, the ARM binary code. And in order to debug this program, the emulator will let itself be driven by GDB, as illustrated by Figure [2.2](#). Some scripts to ease the use of these tools are provided in the archive file. The next questions will guide you in controlling them.

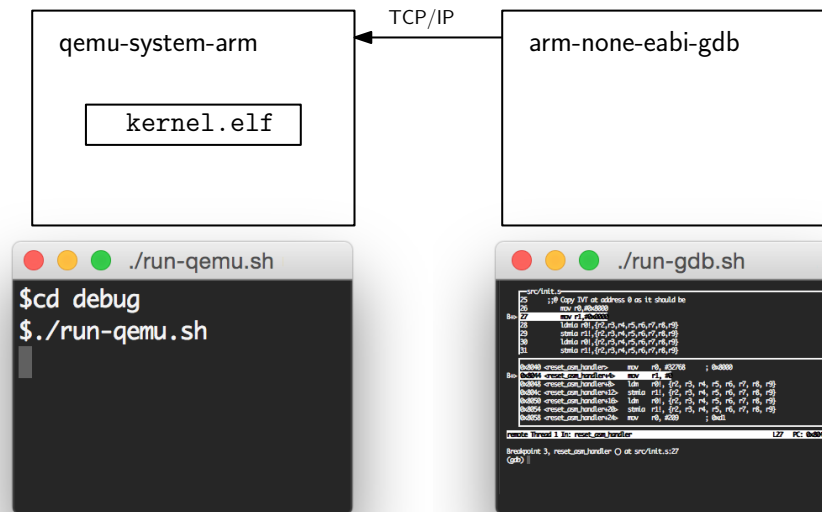


Figure 2.2: Remote debugging: Qemu executes our program, acting as the real platform, and obeys the commands sent by GDB via a TCP socket. Note that the command used to launch the debugger is `gdb-multiarch`, a port of `gdb` able to read and understand various assembly languages.

Exercise: controlling the tools

Question 2-1 Make sure you have the commands `gcc-arm-none-eabi`, `gdb-arm-none-eabi` and `qemu-system-arm`. Packages should be available on your distribution, but you can also compile from source, it is quite straightforward. In the latter case, beware of the `--target-list` option when building `qemu`, and the `--target` option when building `gdb`.

Question 2-2 Then, setup the code in the following manner :

1. Download the archive `barepyOS.tgz` on Moodle
2. Uncompress it : `tar xzf barepyOS.tgz` and change directory: `cd barepyOS`
3. Create in this directory a file `kmain.c` and copy to it the code of Figure 2.3.
4. The `Makefile` allows you to compile your kernel by typing `'make'`. Try it.
5. Move to directory `tools` et execute `'./run-qemu.sh'`. This launch the emulator.
6. In an **other** terminal, go to `tools` and launch `'./run-gdb.sh'`.
7. From GDB, you can control the interactive execution of your program inside the emulator. Type `continue` (or `c`, it's a shortcut) to launch the execution of the code. Your kernel initialize itself and stop at the beginning of `kmain()`. Execute the program step by step with the following commands:
 - `next` (or `n`) execute one line of C and does not enter function calls;
 - `step` (or `s`) execute one line of C and enter function calls;
 - `stepi` (or `si` execute one assembly instruction)
 - at any moment you can visualize the execution stack thanks to the command `backtrace` (or `bt`);
 - if you just type "enter", the previous command is executed once again;

8. A gdb macro allows you to restart your kernel without quitting gdb and killing Qemu: type `reset`, then `continue`.
9. to quit qemu, type `Ctrl-a, x`

Question 2-3 The Makefile we provide to you is a disassembled version of your mini-OS in the file `build/kernel.list`. Open this file and note the following addresses :

- beginning and end of the code (section `.text`) ;
- beginning and end of non-initialized variables (section `.bss`) ;
- beginning of the functions of your file `kmain.c`.

```
void
dummy()
{
    return;
}

int
div(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 div(4*355*rad3, 3*113);
}

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

    dummy();
    volume = compute_volume(radius);

    return volume;
}
```

Figure 2.3: A first piece of code to observe

Exercise: observation of function calls

Procedures of a C program are compiled to machine code which use registers and execution stack. Let's see how they are used.

Question 2-4 Read the insert [1](#) on this page detailing the processor of the Raspberry Pi we target. Then, execute the program one instruction at a time ¹, and note:

- Which registers are used by the fonction `div()`
- Thanks to the command `print/x $reg` (e.g. `print/x $r0`), or the command `info registers`, you can print the value of registers. Just try these commands, they will be useful.
- Simple question: who has chosen to use these particular registers ?

Frame 1: Processor of the Raspberry Pi

This is now the time to familiarize yourself with some important notions and technic terms:

The Raspberry Pi *board* includes a **microcontroller** **Broadcom BCM2836** ;

This microcontroller includes an ARM **Cortex-A7 processor** ;

The Cortex-A7 processor of the BCM2836 includes **4 cores**. Note that the *design* of the ARM Cortex-A7 allows 1 to 4 cores) ;

The processor is part of the **Cortex-A family**, implementing an **ARMv7-A instruction set**. Its registers are composed of:

- 13 general registers `r0` to `r12` ;
- One register `r13` used as stack pointer. It is also called `sp` (for *Stack Pointer*).
- One register `r14`, also called `lr` (for *Link Register*). It has two fonctions :
 - When a jump or a function call arised (typically thanks to the instruction '`BL`'), this register memorise the return address of the function,
 - When an interrupt arises, it memorises the address of the instruction that has been interrupted.
- The register `CPSR` is the status register. It holds the result of the last comparison, the fact that interrupts are activated or not, etc.

The complete documentation of ARM architectures and of our particular processor is online. Don't hesitate to have a look. It's a safe way to find questions your teacher will have a hard time to answer.

Exercise: function calls

The cross-compiler is responsible for producing ARM assembly code equivalent to the C code you wrote. The assembly code for a function call comprises :

1. some instructions that save some registers of the microprocessor on top of the execution stack, so that the called function can use them.
2. some instructions that push the arguments on top of the execution stack the
3. one instruction that makes the processor jumping to the address of the called function. On ARM, it is typically a `bl` instruction (for *branch-and-link*). Beware, after a function has been called and executed entirely, the processor will execute the instruction following this jump. **This address is stored in the register `lr` by the processor when it executes a `bl` instruction, just before jumping to the called function.**

¹use the gdb command `stepi`, or its shortcut `si`

These conventions (*e.g.* the stacking order, number of registers saved...) are defined in the document *ARM Procedure Call Standard*².

Question 2-5 Re-execute the program of Figure 2.3 step by step, **starting from the function `kmain()`** ((you are not supposed to look at the initialisation code from the file `init.s` for now). Just before the return of `kmain()`):

- observe the address stored in `sp`
- observe the content of few stack cells thanks to the following commands: `print/x *((int*) $sp)`, `print/x *((int*) $sp + 1)`, `print/x *((int*) $sp + 2)`, `print/x *((int*) $sp + 3)`. You can also use the gdb command `x/4x ((int*) $sp)` (change the value 4 if you want to see the content of more cells ; change 4x to 4d if you want decimal format).
- At which address is stored the local variable `radius` ? And `volume` ?

Question 2-6 Compare `sp` at the beginning of `kmain()` and at the end of `compute_volume()`. In which direction is the stack growing: increasing addresses or decreasing addresses ?

Question 2-7 `sp` points to the first empty place or the last full place ?

Question 2-8 Re-execute the program step by step and look the values of the register `lr` before and after each function call, as well as at the return of each one thanks to the GDB command `x`. If you have read correctly the explanations at the beginning of the section, you should know which instruction update it implicitly. Which one is this ? If needed, re-read the rôle of register `lr` in the insert on the previous page.

²most curious of you can read this: http://infocenter.arm.com/help/topic/com.arm.doc.ih0042e/IHI0042E_aapcs.pdf

2.4 Exercise: writing (a bit of) ARM assembly

Frame 2: Linking assembly and C code

The compiler GCC allows to mix inside the same program C code and assembly code, via the directive `__asm()`. In addition, one can make both interacting thanks to I/O *operands*. Two examples:

- The line `__asm("mov %0, r0" : "=r"(X));` allows to copy the (content of) register r0 to the variable X;
- The following code performs the inverse operation, naming reading the variable X (and more generally evaluate the expression X) and copying its value in the register r0: `__asm("mov r0, %0" : : "r"(X));` You will have noticed that the entry operand comes after a *second* character « colon ».

Beware of two things:

1. the compiler is liable to translate the same directive `__asm` to *several* assembly instructions, and use the registers it wants. Think of examining carefully the generated code.
2. the compiler translates each directive `__asm` in a separated manner, without taking into account the lines nearby. Unlikely interferences can therefore happen.

In particular, it may happen that the content of a register may be erased. In this case, the solution is to specify in each directive, after a third character « colon », a list of *reserved* registers that the compiler cannot use. This situation is illustrated in the example below, which copies the content of registers r0 and r1 in two variables `var0` and `var1`.

```
// Problem: the first line can erase R1...
__asm("mov %0, r0" : "=r"(var0));
// ... therefore, it is not sure that var1 ends up with the good value
__asm("mov %0, r1" : "=r"(var1));

// solution: indicate which registers are precious
__asm("mov %0, r0" : "=r"(var0) : : "r0", "r1");
__asm("mov %0, r1" : "=r"(var1) : : "r0", "r1");
```

For more details concerning the syntax of inline assembly, please report to the GCC documentation: <http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html>.

Another means of integrating assembly with C is to write some `.s` files (for instance, our file `init.s`) and translate them to machine language thanks to an *assembler* (e.g. GNU `as`). One can then link together the object files obtained thanks to the *linker*. It is therefore what is made by our Makefile.

You will have to write few lines of assembly in the following of the work. Read the insert 2 then answer the two questions below.

Question 2-9 Do the following manipulation:

- at the beginning of `kmain()`, make the processor jump to the address of function `dummy()` thanks to this inline assembly code: `__asm("b <label-ou-adresse-ou-nomfonction>".` Yes, this is a simple *branch* instruction, not a *branch-and-link*.
- execute this program instruction by instruction.

Does your program return correctly of the function `dummy()` ? Why ? Make sure you have read insert on page 10 then fix the assembly instruction you have written so that it returns correctly.

Question 2-10 Thanks to inline assembly, you can have access to the processor directly through C code. Write:

- the line which allows you to put to the register R2 the content of variable `radius`.
- the line which allows you to put to the variable `radius` the content of register R3.

2.5 Exercise : the *naked* attribute

Frame 3: The function attributes

Some C compilers like GCC supports, inside the declaration of a function or a variable, the specification of one or several **attributes**. An attribute is an information given to the compiler to guide the code generation. The attribute we are interesting in here is the attribute `naked`. A function declared as such is compiled without **prologue** nor **epilogue**. Indeed, in a classical function, the generated code starts with a prologue which saves some registers onto the stack. then comes the body of the function, which corresponds to the instructions of the source program. Finally, the code of the function ends with an epilogue which restores the registers saved. In a function naked, the compiler generates neither prologue nor epilogue; if some registers are to be saved, it is the responsibility of the programmer to do it.

This syntax is notably useful to declare an **interrupt handler**, which we expect not to modify the stack.

Example : `void __attribute__((naked)) timer_handler();`

Question 2-11 Read the insert 3. Note the differences between the assembly code of a function declared with and without this attribute (for instance, looking at the file `kernel.list`).

2.6 Exercise : use the provided tests

In order to help you validating your code, we provide, all along the project, various test programs, in the directory `/test`. In this exercise, you will learn how to use this mechanism on simple example.

Question 2-12 Compile your kernel with the code `make KMAIN=./test/kmain-bidule.c` (and open the corresponding `.c` file in your text editor). Execute the program step by step in GDB. Verify that the control flow goes two times through the function `bidule()`.

Question 2-13 You will now do the same verification, non-interactively. Restart Qemu. Then, from the directory `tools`, type the command `./run-gdb.sh ./test/bidule-called-twice.gdb`. Read the gdb script until you are convinced that it verifies the same thing that at the previous question.

Question 2-14 We provide the script `run-test.sh` to automatise the whole handling: compilation, Qemu launch, and GDB in non-interactive mode. Read it and execute it with the good command line parameters, for instance from the main directory: `./tools/run-test.sh ./test/kmain-bidule.c ./test/bidule-called-twice.gdb`

Frame 4: Understanding the boot of the Raspberry Pi

- When the Raspberry is powered on, the ARM processor is not supplied, but the GPU (graphical processor) yes. At this point, the memory (SDRAM) is not powered ;
- The GPU executes the *first bootloader*, constituted of a small program stored in the ROM of the microcontrlleur. These few instructions read the SD card, and load the *second bootloader* stored in the file `bootcode.bin` in the L2 cache ;
- The execution of this program switchon the SDRAM then load the *third bootloader* (file `loader.bin`) in RAM and executes ;
- This program load `start.elf` at address zero, and the ARM processeur executes it ;
- `start.elf` only loads `kernel.img`, in which you code resides ! Small detail: the `start.elf` file we use on the card is the one of a Linux distribution which jumps at address `0x8000` because necessary information (kernel configuration, MMU configuration...) are stored between address `0x0` and `0x8000`. It is possible to change this easily (we won't during the project): <http://www.raspberrypi.org/documentation/configuration/config-txt.md>

The code that resides at address `0x8000` is the one of the file `init.s` (because the compiler looks after the label `_start`). The first lines of this file only configure the interrupt vector table. It is a bit complicated because to each interruption corresponds an instruction, not an address to which jumping as on other architectures like the MSP430. The instruction could be a simple branch in this case it is needed to copy these instructions from address `0` because the CPU first executes the instruction at address `0`.

More information on the boot process:

- <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>

2.7 Documentation

In order to get your bearings in the ARM denomination, you need to understand that *ARM6* is **not** the same than *ARMv6*. *ARM6* designates aprocessor generation whereas *ARMv6* refers to an instruction set. Both are not paired: processors on which you will work are ARM11 processors implementing the instruction set *ARMv6T*. Then, you only need to see:

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

The technical documentation which may be useful is put inside `barepyOS/doc/hard/`. You will find notably:

- the *ARM Architecture Reference Manual* (the “ARM ARM”) describing the ARM instruction set: `ARM_Architecture_Reference_Manual.pdf`;
- the documentation of the Raspberry Pi microcontroller: `BCM2835-ARM-Peripherals.pdf`. (there does not exist a specific datasheet for the BCM2836 but the only significant difference is the replacement of the processor;
- the documentation of the Cortex-A7 processor: `DDI0464F_cortex_a7_mpcore_r0p5_trm.pdf`.
- a quick reference card of the ARM instruction set

Lots of information on the Raspberry Pi : http://elinux.org/RPi_Hub, and https://en.wikipedia.org/wiki/Raspberry_Pi

Chapter 3

CPU execution modes

A program can execute with **privileges** more or less important. Basically, privileges can be divided in two classes: those of the code running in the *user space*, and those of the code running in the *kernel space*. Concretely, a processor can be configured (via the register status) in order to execute in a *user mode* or in *privileged mode*.

3.1 The ARM execution mode

ARM processors propose 1 user mode – the mode **User** – but several privileged modes:

- the mode **System** in which you code will have to execute.
- several modes to handle exceptions:
 - the mode **SVC** (for *Supervisor*). It is the mode in which the processor executes when a software interrupt arises (caused by the instruction **SWI** – *SoftWare Interrupt*)
 - the mode **Abort** in which the CPU run when a data access arises (signaled by the MMU for instance, you'll see that in chapter 10).
 - the modes **IRQ** and **FIQ** (*a priori*, you will not use the latter) in which the CPU executes when it receives a hardware interrupt.

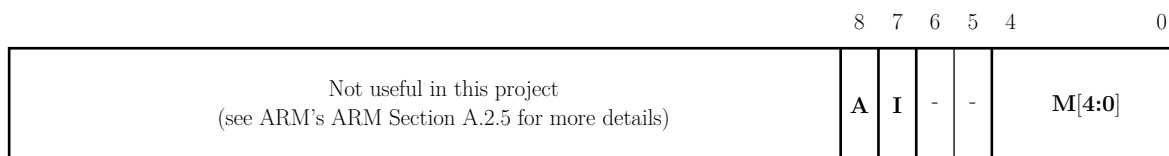


Figure 3.1: The status registers detailed

In order to change the execute mode of the processor, it suffices to modify the five bits 0 to 4 of the status register, called **CPSR** (*Current Program Status Register*). In this goal, two means exist :

- the instruction **cps #<num>** (*Change Processor State*) copies the value of **num** in the bits 0 to 4 of **CPSR**.
- write directly the status register via the instruction **msr CPSR, <reg>** which copies the whole content of the register **<reg>** in **CPSR**.

To read the status register, the instruction **mrs** must be used. Example : **mrs r0, CPSR**.

Figure 3.1 details CPSR, the role of each bit is given below:

M[4:0] are the bits determining in which execution mode the processor is currently executing. Table 3.2 indicates the value associated to which execution mode ;

I indicate if interrupts (*IRQ*) are activated (value 0), or not (1) ;

A indicate if *data aborts* are activated or not. This will be useful at chapter 8.

Bits	Execution mode
0b10000	User
0b10001	FIQ
0b10010	IRQ
0b10011	SVC (Supervisor)
0b10111	Abort
0b11111	System

Table 3.2: Values coding the execution mode of the processor

Two important things, summed up by Figure 3.3 :

- each execution mode has its own register **SP** (**r13**) and its own register **LR** (**r14**). The execution modes **User** and **System** share the same register;
- each privileged execution mode, except **System**, is said to be an **exception** mode, and has at its disposal one more register, **SPSR** (*Saved Program Status Register*).

When the processor is signaled that an **exception** has been raised (for instance, an interrupt), it changes its execution mode from **User** or **System** to an exception mode, and makes two things :





















1. it copies the value of **CPSR** into the register **SPSR** of the exception mode.
2. it copies the value of **PC** into the register **LR** of the exception mode. Small detail that will be important at some point: at this moment of the fetch-decode-execute cycle of the processor, **PC** has already been incremented by 4, and already points to the instruction *following* the one that is being executed.

NB: if the processor is in **System** mode, you can pass explicitly to an exception mode via the instruction **cps**. **In this case, these 2 actions are not performed.**

3.2 Physical memory organisation

Now that you have understood that several execution modes exist and that some of them have their own stack pointer, have a look at Figure 3.4. It shows the initial organisation of the RAM of the Raspberry Pi. You can see:

- At address 0, the interrupt vector table,
- At address 0x8000, the booting code of the kernel (file **init.s**) ;
- The execution stack of the **SVC** mode starts at the address of the label **__svc_stack_end__** ;
- Above address 0x20000000, the address space is dedicated to peripherals. It is said that peripherals' registers are mapped in memory.

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


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

Figure 3.3: The various register sets visible in each execution mode. Figure extracted from the ARM reference manual §A2.3 page 43. Go read the corresponding explanations.

3.3 Exercise: changing the execution mode

Question 3-1 Observe the value of `CPSR` at the beginning of `kmain()`. In which execution mode is the processor ?

Question 3-2 Modify your program so that it changes the processor to `SVC` mode at the beginning of `kmain()`. Note the values of `CPSR` and `SP` before and after this change. Verify that the values are the good ones.

Question 3-3 Note the value of `LR` before and after having changed the processor mode to `SVC` at the beginning of `kmain()`. Use the GDB command `x AND` look inside the file `kernel.list` in order to deduce where this register points to.

Question 3-4 What happens if you try to change the processor to `User` mode then to `SVC` ? Why ?

Question 3-5 What happens if you try to access the register `SPSR` at the beginning of `kmain()` (beware, you cannot access a status register through a `mov`, you have to use the `mrs` instruction) ? Why ?

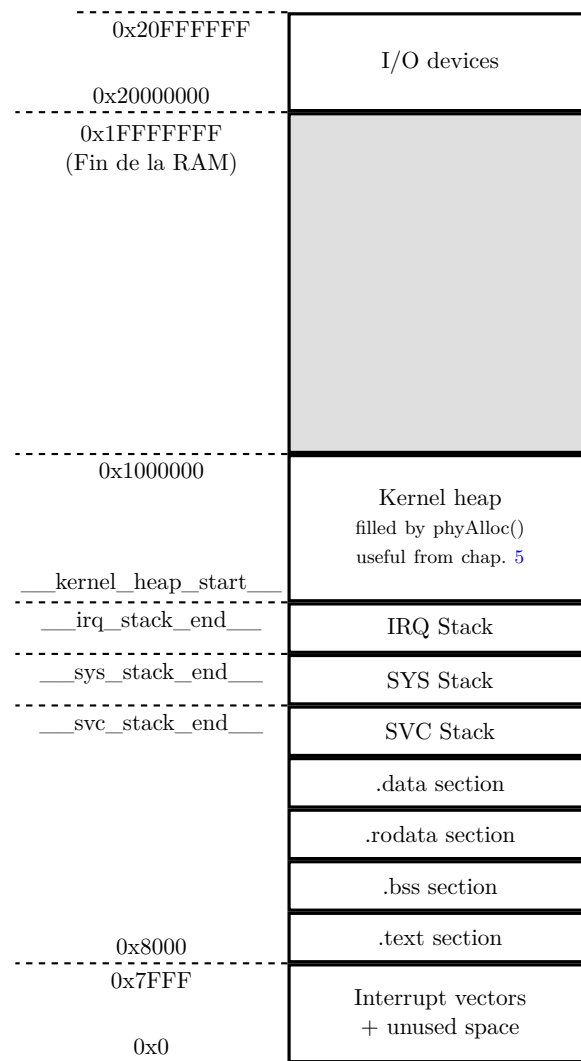


Figure 3.4: Organisation of the initial physical address space. Start and end addresses (numerical or symbolic) of each section are defined by the *linker script*, that is to say the file `barepy0S.ld`. Sections `.bss`, `.text`, `.data`, `.rodata` are built by the compiler

Question 3-6 In the file `utils.gdb`, we provide a macro `print_sr` that can prints in a readable form the status register. It indicates the current execution mode, the value of various flags and of the bit indicating of interrupts are masked or not. Bits equal to 0 are in lower case, and bits to '1' are in upper case. Try this macro at various execution points of your kernel (and above all, think of using it in the following).

Chapter 4

System calls

A **system call** (for *system call*) is a primitive function, provided by the kernel of the operating system, able to execute a piece of code requiring some **privileges** inaccessible to user programs. For instance, under Linux, in order to open a file, one must invoke the system call `open()` ¹.

This chapter guides you in the implementation of a system call mechanism.

4.1 What you will learn in this chapter

OS / Operating System Principles : Concept

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

OS / Operating System Principles : Skills.

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

4.2 System calls, the principle

The mechanism, illustrated by Figure 4.1, is quite simple:

- to invoke() a system call, the application triggers a *software interrupt* ². In this goal, we will use the instruction `SWI`. See the insert 5 ;
- the kernel catches this interrupt in an interrupt handler, with kernel privileges.

More precisely, you will implement system calls in the form of the following one:

¹in practice, one often use *wrapper* functions easier to use, for instance the function `fopen()` in C, or the `fstream` in C++.

Those functions interacts with the kernel.

²as we want the CPU to run in a privileged execution mode, one cannot simply jump in the kernel with a mere classical function call

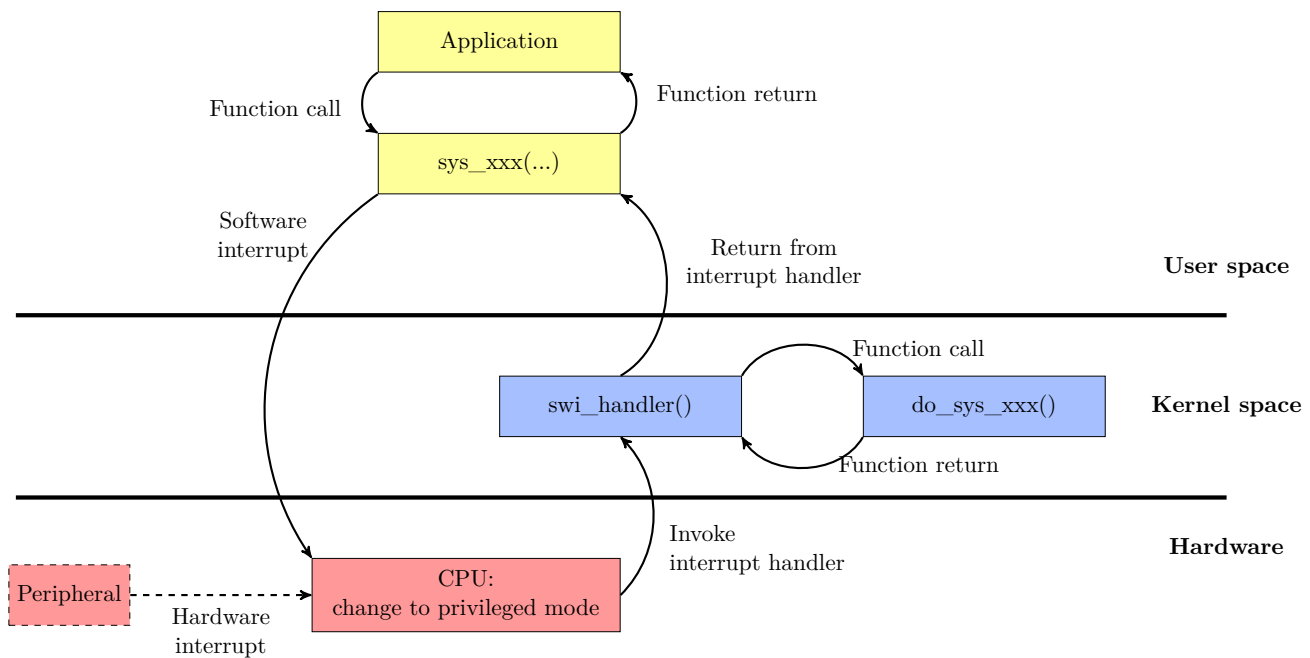


Figure 4.1: Overview of a system call.

Name	Number	Role
<code>void sys_XXX()</code>	<i>N</i>	The kernel makes something that is impossible for a userland process

The principle of your syscall mechanism will be the following:

- function `sys_XXX` is the userland part of the syscall. It is called by a process;
- this function sets `r0` to the system call number corresponding to this syscall (denoted *N* in the table above), then it triggers a SWI;
- the software interrupt will be handled, on kernel side, by a function `C C_swi_handler()` which:
 1. gets from `r0` the number of the system call `sys_XXX()`
 2. calls the correct function `do_sys_XXX` that implements the corresponding service.

The exercises in the following guide you in the implementation of various system calls

All the code of this chapter should be placed in the files `syscall.c` and `syscall.h` (create them now in the directory `src`).

Frame 5: The ARM instruction SWI

The instruction SWI allows to ask for the execution of code in *supervisor* mode (SVC). Its effect is the following:

1. `CPSR[4:0] := 0b10011`, that is to say, the processor mode change to SVC ;
2. The address of the instruction following the SWI is copied in `LR_svc`, i.e. the register `LR` of the SVC mode ; it's at this address that the CPU will have to jump after the system call has been handled.
3. the value of `CPSR_user` is copied to `SPSR_svc`, i.e. the register `SPSR` of SVC mode ;
4. the address of the function that handles software interrupts is copied to `PC`.

For more details, report to the reference manual ARM §A2.6.4 page 58 (and also §A4.1.107 page 360).

4.3 Exercice : a first system call that never returns

You will start by writing a simplistic system call mechanism. This system call requires no parameters and returns no value because it doesn't even return. More precisely, you will implement the following system call:

Name	Number	Function
<code>void sys_reboot()</code>	1	Reboot the system

Question 4-1 Declare and define the function `void sys_reboot()` that implements the user side of the system call.

Note that the instruction `SWI` require an immediate operand, but we will not use it in the project. In practice, you will always use the syntaxe `SWI #0`.

Question 4-2 Declare (in `syscall.c`) a function `void C_swi_handler(void)`, completely empty for the moment. Modify the code provided in `init.s` so that it jumps to your function `C_swi_handler()` each time a software interrupt is issued rather than infinitely looping. Execute your program step-by-step, to understand what happens.

Question 4-3 Implement the function `C_swi_handler()` that calls `do_sys_reboot()` if and only if the number of the system call is 1. In this goal:

- In order to debug more easily, you should put a breakpoint in this function. To do this, read and understand the script `run-gdb.sh` and modify the file `init.gdb`.
- Read the insert 6 for the implementation of `do_sys_reboot()`.
- If the number of the system call is not 1, then the kernel has nothing smart to do. Rather than ignoring silently the error, call function `PANIC()` in order to help you debugging. See the insert 7.

Question 4-4 At this point, your OS must be capable of executing correctly the following program. Verify step by step that you go through `sys_reboot()`, then `C_swi_handler()`, then `do_sys_reboot()`, and that this function reboots the system: the control flow comes back to `kmain()`.

```
#include "syscall.h"

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

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

    sys_reboot();
}
```

Question 4-5 Do the same thing using `run-test.sh` : your kernel must successfully pass the test `sys-reboot-does-reboot.gdb` on `test/kmain-reboot.c`

Frame 6: Rebooting the microcontroller

Rebooting the entire system (noyau + matériel) is not so simple. As long as you work on top of an emulator, you can just jump to address `0x0000` where resides the interrupt handler `reset` (for most curious: it is the routine `asm_reset_handler` in `init.s`).

Frame 7: Defensive programming inside the kernel: macros PANIC() and ASSERT()

In order to ease debugging, we provide you, in files `util.c` and `util.h`, a few utility classical functions. When you detect an unexpected situation (for instance, a system call with unknown number), a bug probably appeared in your kernel code, or in the application. It would be useless to continue execution, better stop as soon as possible.

This is exactly the role of function `kernel_panic()`, which only exists to serve as a sentinelle. The idea is to put a breakpoint on this function (add it now in your `gdbinit`) and invoke it when something bad is happening.

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

You can place a call to this function in several places of your code. In order to identify individually each of this place we provide you with a macro `PANIC()`, which will invoke this function with as parameters the name of the file and the number of the line of code that made the call:

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

At last, we also provide you with a macro `ASSERT()`. It verifies that a boolean expression is true, and panics if the expression is false:

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

In practice, you will probably never call yourself `kernel_panic()` by hand, but only the two macros `PANIC()` et `ASSERT()`.

Verify that you have understood how to use these two macros by invoking them on simplistic examples:

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

and ensure that you obtain, in `gdb`, a message of this kind:

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

4.4 Exercice: a system cal that returns

Implementing *Reboot* was easy: you could make mistakes on the kernel side, it was not leading to anything wrong because the goal was anyway to reboot. Following questions guide you in the implementation of the system call below:

Name	Number	Function
<code>void sys_nop()</code>	2	Does nothing, then returns

Of course, the user code that calls this function must then continue its execution normally.

Question 4-6 Implement a naive version of this function, and check that it is invoked correctly.

Question 4-7 One remaining problem is to return to the calling user code. Why is it a problem in your current version ? Execute program step by step in order to understand what happens.

Question 4-8 To solve this problem, you will add, at the beginning of function `C_swi_handler()`, a *copy* of user registers, and you will *restore* them when the system call is finished. More precisely, you will push them (onto the SVC stack) thanks to the instruction `stmfd`, then pop them with function `ldmfd`. Read the insert8 (and for more details, report to the reference manual ARM³) ; why the register `spsr` doesn't appear on the right side of figure illustrating instruction `ldmfd` ?

Question 4-9 Solve the problem described above by saving (resp. restoring) the registers of the user context inside (resp. from) SVC stack at the beginning and at the end of `C_swi_handler()`.

Question 4-10 Is it necessary to save explicitly the status register also ? Why ?

Question 4-11 At this point, your OS must be capable of executing correctly the following program. Verify in GDB that your kernel really comes back from `sys_nop`, then *que vous rebootez*.

```
#include "syscall.h"

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

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

    sys_nop();

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

Question 4-12 However, one last detail remains in order for your implementation to be correct. Add an infinite loop around the call to `sys_nop()`, and print (using gdb) the register `SP_svc` at the beginning of `C_swi_handler()`. If you let the loop execute, what do you think will happen ? Look at the code of function `C_swi_handler()` (in `kernel.list`), in order to understand how to fix this problem... and fix it.

Question 4-13 Check with `run-test.sh` and the program `kmain-nop-reboot.c` that your kernel pass the following tests:

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

³for `stmfd` see §A4.1.97 page 339, and for `ldmfd` see §A4.1.22 page 190. You can also find useful explanations at §2.6, pages 54.

Frame 8: Saving and restoring contexts

The set of information defining an execution point is called a context.

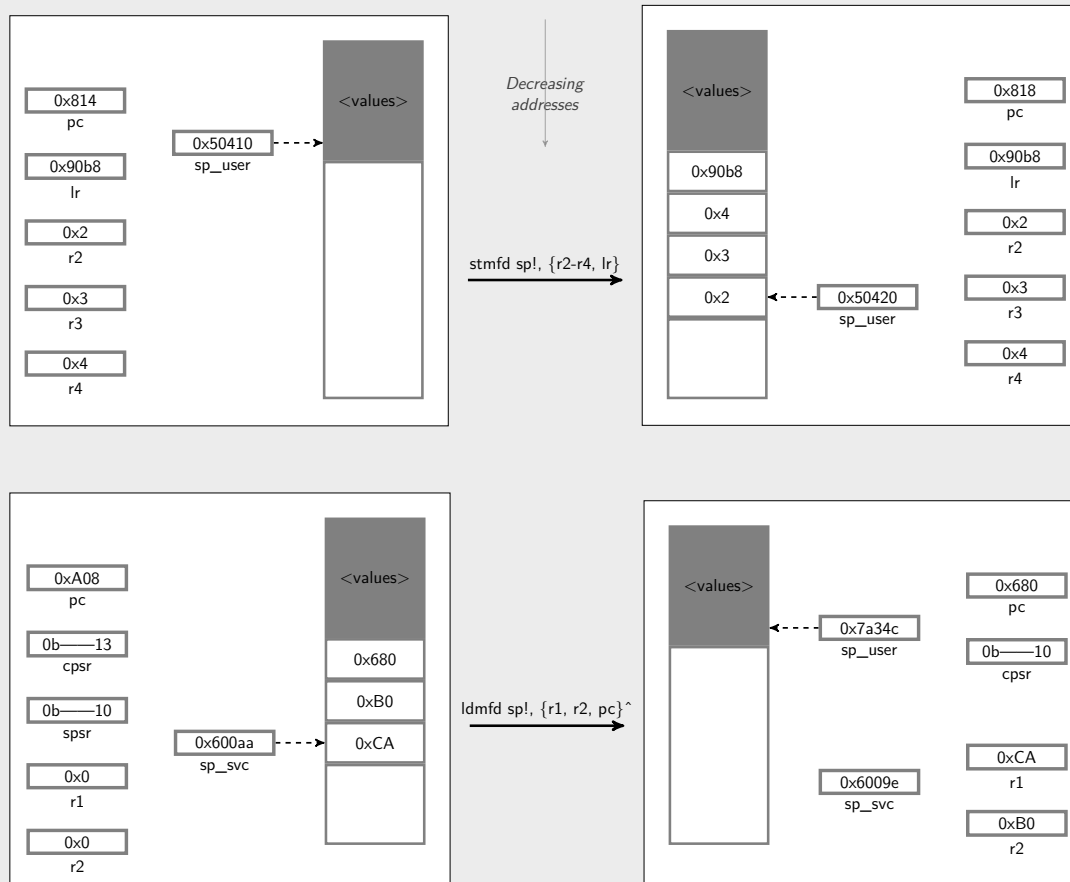
In your mini-OS (as in many OSes), a context contains the processor registers, as well as the current state of the execution stack. So, by definition, if the memory containing the execution stack is not modified, saving the registers is enough to memorise a context.

In other words, at some point, the state of the processor is photographed. Later, restoring the values of these registers will allow to resume the execution of the program whose context has been saved.

There exists many ways of saving and restoring registers. Use, at least in the first version of your OS, two specialisations of the instructions STM (for *STore Multiple*) and LDM (for *LoaD Multiple*), whose functioning is illustrated in the Figure above.

FD means full descending, where descending means that the stack grows when SP decreases, and full means that SP points to the last full stack location.

Beware ! The role of character '^' is important: if it is present, the instruction copies the content of SPSR to CPSR. Finally, the character '!' specifies that SP must be updated in order to point immediatly after on the last full stack location (*i.e.* no need to decrease SP yourself).



4.5 Exercise : a system call with parameters passing

You will now add a syscall that takes parameters as input. For now, let us chose a calling convention that will be respected in user-space (just before the execution of SWI) and kernel-space (after the execution of SWI). Because the processor will change its execution mode, SP will change, so the easiest way is to pass paramters via the registers.

- On user side, each function `sys_bidule()` will place the values of parameters in general registers, then execute the instruction SWI.

- On kernel side, `C_swi_handler()` starts by pushing all registers (on the SVC stack), then it memorises the location of the stack pointer, and passes it to the function `do_sys_bidule()`, which will use it to find the parameters in memory.

This way, your kernel can access the values of user registers (by accessing the stack), while using CPU registers to execute all the same.

Such a calling convention is called an *Application Binary Interface*, or *ABI*. Citing Wikipedia.en :

In computer software, an application binary interface (ABI) is the interface between two program modules, one of which is often a library and/or operating system and the other one is usually an application created by a regular programmer. In contrast to an API, which defines structures and methods one can use at software level, an ABI defines the structures and methods used to access external, already compiled libraries/code at the level of machine code. It does this by determining in which binary format information should be passed from one program component to the next, or to the operating system in the case of a system call. Thus it sets details such as the calling convention.

Question 4-14 Implement the following system call with, for the moment, an empty kernel-side implementation.

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

Question 4-15 Invoke `sys_settime()` with easily recognizable parameters, and draw a figure of the SVC stack at the beginning of function `do_sys_settime()`.

It is tempting to go and grab parameters directly on the top of the stack. But this naive solution will not work every time.

Question 4-16 Add some local variables in your function `do_sys_settime()`, initialised to values easily recognisable, and also some useless computations. Print again the state of the stack at the beginning of the function. What do you conclude ?

Question 4-17 Fix your function `C_swi_handler()` and complete the implementation of `do_sys_settime()`. Once you have correctly assembled the integer `date_ms`, you can pass it to the primitive `set_date_ms()` of `hw.h/hw.c` (also : go and have a look at its implementation).

Question 4-18 For now, your OS must be capable of executing correctly a program that calls `sys_settime()`. Check step by step that your parameter is correctly passed through registers R1 and R2, then saved in the stack, then re-built in `do_sys_settime()`, and finally passed as parameter to `set_date_ms()`.

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

4.6 Exercise : a syscall that returns a value

Last but not least, you will implement a system call that returns a value. Again, you have to choose a convention for parameters' passing. I propose the following principle : as parameters are passed to function `do_sys_bidule()` via the SVC stack, these functions will modify the values contained in the stack, so that restoring the context at the end of `swi_handler()` will put in `r0` the correct return value. If the return value requires more than 32 bits, use also `r1`.

Function `ys_bidule` can read these registers in order to build the value that must be returned to the application.

Question 4-20 Implement the following system call:

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

Question 4-21 Check with `run-test.sh` and the program `kmain-gettime.c` that your kernel pass successfully the test `sys-gettime-returns-value.gdb`.

Chapter 5

Process dispatching

Until now, you have only executed one program at a time. On a real computing system, one want most of the time execute several programs at a time, necessitating to manage several **processes**. In this chapter and the two following ones, you will implement the mechanisms necessary to share the processor between several processes.

5.1 What you will learn in this chapter

OS / Concurrency : Concepts.

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

OS / Concurrency : Skills.

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

5.2 Introduction

In operating systems, hypervisors, virtual machines, the **dispatcher** is responsible for allowing one **process** to access the CPU. In this chapter, you will implement this piece of software. Put all your code in the files `sched.c` and `sched.h`.

5.3 Exercise : a dispatcher for coroutines

An operating system allows several processes to execute *concurrently*. The term *concurrently* is used because processes have to share hardware resources (*e.g.* the CPU). At some points, the operating system must stop the execution of a process, and execute another one.

For now, we consider the execution of only two processes. Each process voluntarily returns the control flow to the operating system, allowing it to do “what is necessary” for the other process to execute. The two processes are called **coroutines**.

In this goal, the operating system maintains, for each process, a data structure called **Process Control Block (PCB)**, in which it saves the **execution context** and the **process state** when process is not executing.

At the end of the chapter, your kernel will be able to execute applications of this kind:

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

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

You will have to implement, among other things, the following system call:

Name	Number	Function
<code>void sys_yieldto(struct pcb_s* dest)</code>	5	Give hand to another process

Notice that you don’t have defined the type of parameter `dest` yet. The following questions will lead you to his definition. The principle is quite simple; in fact, the job is already almost done ! Indeed, the first instruction of your `C_swi_handler()` consists in stacking most user registers. If you don’t change the values in the stack, then the values of the registers will be the same at the end of `C_swi_handler()` than before the system call, and the same process will execute. It’s because the system call mechanism already save/restore registers from the stack. But if you replace the content of the stack with values of the *other* coroutine, before restoring the registers, then this other coroutine will resume its execution ! This operation is called a **context switch**.

Question 5-1 Declare a type `struct pcb_s` with no field, and write the functions `sys_yieldto()` and `do_sys_yieldto()`. For the moment, the latter gets its parameter `dest` but does nothing.

Question 5-2 Add the necessary fields for your `struct pcb_s` to contain a copy of all registers saved at the beginning of `C_swi_handler()`. Add to `do_sys_yieldto()` the instructions necessary to make the context-switch. Because the calling process indicates to which it wants to give access to the CPU, the OS knows the addresses that must be pushed onto the stack and be restored. But, the context of the calling process needs to be copied somewhere. In this goal, you will declare in `sched.c` a global variable `struct pcb_s *current_process`, and you will make it permanently point on the process being executed.

Question 5-3 But as it is, it cannot work: during the first call to `sys_yieldto()`, your kernel will save the context of `kmain()` in the location pointed to by `current_process...` which is a non initialized pointer ! So, declare (in `sched.c`) a `struct pcb_s kmain_process` which will serve exactly for the first context-switch. You will also need a function `sched_init()` that will initialize `current_process`.

Now you can jump from one process to another, if the processes are already started ! Indeed, for the moment, the very first call to `sys_yieldto(p1);` in `kmain` cannot work, because `p1` is not initialized. The first time it will be reloaded in the CPU, the registers will therefore be unreliable, at the very least . For data registers, it is not really a problem: the new process will initialize its variables at the beginning of its execution. On the contrary, it is important to initialize the field `LR` of the PCB, so that it can be restored correctly (in `PC`) at the end of `C_swi_handler()`, so that the CPU jumps to the right location.

Copy in `kmain.c` the two functions `user_process_N` given at the beginning of the chapter, as well as the following code:

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

struct pcb_s pcb1, pcb2;

struct pcb_s *p1, *p2;

void user_process_1 and _2 ...

void kmain( void )
{
    sched_init();

    p1=&pcb1;
    p2=&pcb2;

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

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

    sys_yieldto(p1);

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

Question 5-4 Replace the [your code goes here] by some code that initializes the field `LR` of each PCB the address of the correct function.

Question 5-5 At this stage, it almost works ! Execute your program step by step, and observe the control flow going successively from `kmain` to `user_process_1`, to `user_process_2`, to `user_process_2`, to `user_process_2...`

The problem is that we didn't save all registers ! Remember : at exercise 4.5, we had pushed all registers onto the stack. But the register `LR` that we save at the beginning of `C_swi_handler()` is `LR_svc`, containing the return address of the exception mode `SVC`. The register `LR_user` is not directly visible from the mode `SVC`. For «simple» system calls, we didn't have to handle this because we always returned to the same process. But now that we go from one process to another, we need to save/restore also `LR_user`.

Question 5-6 Distinguish in your PCB the two fields `LR_svc` and `LR_user`, and add to your context-switch mechanism the saving and the restoring of `LR_user`.

Question 5-7 Execute the program step by step and check that now the control flow goes from `kmain()` to `user_process_1()`, then to `user_process_2()`, then back to `user_process_1()`, then `user_process_2()` etc.

5.4 Exercise: coroutines with local variables

Question 5-8 In gdb, observe the evolution of the local variables of your two processes. What do you see ? Study the machine code of your two functions in order to understand what happens.

The problem is that for now we do not have really independant processes: their cpu registers are now distinct, and correctly saved/restored, but they all share the same *execution stack* ! Each process puts its variables in it, so if the same stack is shared by all processes, each one will erase the variables of each other, and evrybody will compute things with neither rythme nor reason.

The solution is to allocate an execution stack to each proces, and save/restore also the stack pointer during context-switching.

Question 5-9 We will now give each process one PCB but also an execution stack. In order not to repeat code, you will first add to `sched.c` a function `create_process()` which will dynamically allocate (see the insert 9) a `struct pcb_s`, initialise it, and return it to the caller.

More precisely, the function prototype will be `struct pcb_s* create_process(func_t entry)`, taking as parameter a particular function pointer and returning a pointer to `pcb`. For user processes, you will define in `sched.h` your type `func_t` in the following way: `typedef int (func_t) (void);`. In other words, a user process is a function without parameter, and returning an integer.

Yau can now remove your two variables `pcb1` and `pcb2`, and re-write correctly your function `kmain()` as shown below. Remark: for the moment, your processes return nothing, and cannot even terminate... You must explicitly type cast so that the compiler accepts our program. We could also have declared `func_t` in an other way, and save the cast. However, in the next chepter, you will implement process termination, and return of a value, so let's keep `func_t` in its final form.

```
void kmain( void )
{
    sched_init();

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

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

    sys_yieldto(p1);

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

Frame 9: Memory management

From now on, you will need to allocate memory dynamically in order to store all your data structures (execution stacks, PCB...). However, you don't have implemented the memory manager of your OS yet... We provide you with a very simple allocator that manages physical memory blocks. In order to allocate memory, use function

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

This function allocates `size` bytes of data, and gives back a pointer to the start of the allocated zone. To free this zone, use

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

These two functions are declared in `kheap.h` and defined in `kheap.c`. Before you can use them, you have to initialize the memory manager by calling `kheap_init()`.

Question 5-10 Read the insert 9 then initialise the memory manager in your function `sched_init()` (it is important that it is here because tests suppose that).

Question 5-11 Allocate to each process one execution stack of 10Kb, this will be enough for this project¹. Add to your PCB a field `sp` in order to store the stack pointer of a process. Initially, the stack of each process contains no data. On which address must you make this field `sp` point to ?

Question 5-12 As for `LR_user`, you have to save/restore the register `SP_user` to/from the good `PCB->sp` during context-switches.

Your program must now execute correctly. Check step by step that you jump indefinitely from one process to another, and that their local variables are preserved after each context-switch: during execution, `v1` is increasing, and `v2` is more and more negative.

Question 5-13 Strictly speaking, one last detail remains to be adjusted. During each context-switch, you have to save the state register of the process, i.e. `CPSR_user`, which has been copied for you, by the CPU, in `SPSR_svc` when the instruction SWI was executed. Add to your context-switch mechanism the saving and the restoring of this register.

Question 5-14 Check with `run-test.sh` and program `kmain-yieldto.c` that your kernel successfully pass the following tests:

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

Question 5-15 In a classical OS, the kernel doesn't allow for an application to choose to which one it gives access to the CPU. One usually find a system call `yield()` without parameter, but no system call `yield_to(dest)`. Why ? Give at least two good reasons not to offer this feature.

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

Chapter 6

Collaborative scheduling (round-robin)

If numerous processes are executing concurrently, it wouldn't be serious to let them choose which process they let access the CPU. It is one of the main role of the operating system to choose intelligently, during each context-switch, which process will execute next. The component that is in charge of this decision is the **scheduler**. In this chapter, you will implement a simple scheduler.

6.1 Ce que vous allez apprendre dans ce chapitre

OS / Concurrency : Concepts.

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

OS / Concurrency : Skills.

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

OS / Scheduling and Dispatch : Concepts.

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

SF / Resource Allocation and Scheduling : Skills.

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

6.2 Scheduling

As we saw precedently, the information needed for the OS to be able to context-switch between two processes are stored in the PCB of each process. But in the general case, an arbitrary number of processes can execute concurrently. The role of the scheduler is to:

- maintain one or several data structure containing the PCBs. Several data structures can be used for this purpose. You will, in this chapter, use a linked list of PCBs.
- choose, when it is needed, the process to be executed. This choice is denoted the **election** of the process. Many, many algorithms exist, each having different properties (performance, starvation possibility, etc.). In this chapter, you will implement a quite simple **round-robin** scheduler : the elected process will be the process following, in the linked list, the one that was executing just before.

6.3 Exercise: round-robin scheduling

The goal of this exercise is to implement the list of PCB and the election function of the scheduler. Concretely, at the end of the exercise, your kernel will be able to execute the program `test/kmain-yield.c`, visible on page 34.

Question 6-1 A simple solution to organise the PCBs consist in chaining them directly between them, without defining additional data structures. Propose a modification of the structure `struct pcb_s` for this purpose.

Question 6-2 Implement the scheduler round-robin. Your job includes (but is not restricted to):

- modify `create_process()` which must now add the new PCB to the linked list. Function `kmain()` now cannot have access to the PCBs.
- implement a function `void elect()`, which chooses the next process and makes global variable `current_process` point to its PCB.
- add a new system call `sys_yield()` without parameter, allowing an application to yield..
- think of chaining `kmain_process` with other processes, or not...

Question 6-3 At this stage, your OS must be capable of executing correctly the kind of program given just below. Check with `gdb` that you go through each process at its turn, and that the different variables increase at the right pace.

```

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

#define NB_PROCESS 5

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

void kmain( void )
{
    sched_init();

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

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

    while(1)
    {
        sys_yield();
    }
}

```

Question 6-4 Check with `avec run-test.sh` and the program `kmain-yield.c` that your kernel successfully pass the following tests:

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

6.4 Exercise : process termination

When a program terminates, its execution context must not be used anymore, and useless data structures must be deallocated.

A simple solution consists, in order for a process to indicate its termination to the OS, to make a system call, in the user code. You will implement the following system call :

Name	Number	Role
<code>void sys_exit(int status)</code>	7	Ends the current process with the return code <code>status</code> .

Some subtleties force us to guide you in its kernel side implementation (function `do_sys_exit()` if you have not forgotten the nomenclature rules), via the following few questions above. It's not that we are kind, it's just that don't want to explain the same things 15 times.

Question 6-5 How can you, in `do_sys_exit()`, remove from the linked list the PCB of the process that has just terminated ? What do you think of the effectiveness of this operation in the case where a huge number of processes are executing ? Is it a problem ?

Question 6-6 Both following solutions work, feel free to implement the one you prefer :

- Mark the process as `TERMINATED` and handle its termination in function `select()`. In this case, you have to add the notion of *state* in the PCBs, and modify `select()` in order to remove a potential `TERMINATED` process met when walking through the linked list;
- Double-chain the linked list of PCBs. It does not prevent you from adding the process state in the PCB structure.

Question 6-7 Modify your `kmain.c` so that USER processes call `sys_exit()` at some point. Check, step by step, that your OS behave correctly.

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

    sys_exit();
}
```

Question 6-8 I bet you have forgotten that `sys_exit()` must return an error code. Modify the PCB structure in order to store this return code. To get this code on the kernel side, remember the system call `sys_gettime()`, that returns a value.

Question 6-9 When the scheduler have not process to execute anymore, your OS calls `terminate_kernel()`. Indeed, for the moment, nothing can lead to the creation of a process except a user process itself.

6.5 Optional exercise : non explicit termination of a process

Forcing the application programmer to call `sys_exit()` is a problem : in case of oversight, the whole system becomes unusable. Ok, I agree that the wrong behavior of a proces can anyway make the whole system unusable beacause your scheduler is collaborative for the moment. But this remark is also valid for your future preemptive scheduler.

The launch of a process must not be done with a mere jump to its main function `pcb->entry()`, but to a new function `start_current_process()`. This function statrts by calling function `pcb->entry()`, then, when this one, performs the system call `sys_exit()`, in a transparent way for the user process.

Chapter 7

Preemptive scheduling

The scheduler that you have implemented until now allow you to share the processor in a *collaborative* way, that is to say that your kernel waits for processes to let it executing. Even if some OS use this approach, most rather implement a *preemptive* scheduling, that is to say that the kernel *force* a process to pause at some point. You will implement such a scheduler in this chapter.

7.1 What you will learn in this chapter

OS / Concurrency : Concepts.

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

OS / Concurrency : Skills.

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

OS / Scheduling and Dispatch : Concepts.

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

OS / Scheduling and Dispatch : Skills.

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

SF / Resource Allocation and Scheduling : Skills.

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

7.2 Scheduling on interruption

Your scheduler will be able, at the end of this chapter, to pause a process executing and execute another process. In this goal, you will use a hardware *timer* and make it send to the CPU some **interrupt requests** regularly. An interrupt request is an event quite similar to a SWI: it leads the CPU to changing its execution mode and jumping to a given kernel function (an **interrupt handler**) in which you will make a context-switch. For more details on interrupts, read the insert 10, and report to the ARM reference manual §2.6.8 page 62.

In order to save you some time, and avoid useless redundancies with other courses you may have followed, we provide you with the few primitives below in order to interact with the hardware layer (go and see them more closely in `hw.c/hw.h`). Beware: they only work correctly si the processor is in a privileged execution mode !

```
void timer_init();
void clear_and_set_timer();
void DISABLE_IRQ();
void ENABLE_IRQ();
```

Function `timer_init()`, mmm... initialises the timer. It sets the delay after which the timer will expire and allows the timer to trigger an IRQ at this moment. But it does not enable the *processor* to take into account these interrupts ! Each time you want to re-arm the timer (after a timer interrupt has occurred), you have to call the primitive `clear_and_set_timer()`. This one configures the timer expiration 1s later. The interrupt vector is configured to jump at label `asm_irq_handler` in the file `init.s`. As for SWI, you will have to modify the assembly code so that it jumps to a function `C_irq_handler()` that you will write in C.

The two primitives `DISABLE_IRQ()` and `ENABLE_IRQ()` configure the behavior of the CPU when it receives an IRQ (thanks to the bit I of the status register¹). Note that when the system boots, interrupts are deactivated.

¹cf manuel de référence ARM §2.5 page 49

Frame 10: Interruption matérielle

Small architecture reminder : the processor executes fetch-decode-execute cycles, checking before each instruction if an IRQ has not raised. If an IRQ has raised, the CPU jumps to the appropriate **interrupt handler**. In our case, the behavior of processor ARM related to hardware interrupts is *grosso modo* the same that what we saw concerning *software interrupts* (insert 5 page 20). When it receives an IRQ, the CPU does, by itself, several things :

1. save CPSR by copying it in SPSR_irq
2. change to mode IRQ ;
3. deactivate interrupts (by setting to zéro the bit I of CPSR)
4. save PC in LR_irq (see below).
5. load to PC the appropriate interrupt vector.

In ARM CPUs, interrupt vector table is not made of addresses but of instructions. That is to say, if IRQ nb. n is pending, then the n-th instruction of the interrupt vector table is executed. We have filled the interrupt vector table so that this instruction is a jump to a given address when the timer sets an IRQ. You will have to change this address in order to execute your code but beware ! When your **interrupt handler** will execute, the pipeline of the processor makes that PC already contains the address of the next instruction. You must therefore decrease LR_irq by 4 to get the «good» return address, or we will «miss» one instruction each time.

For more détails, see the ARM reference manual ARM §A2.6.8 page 62.

7.3 Exercise: handling interruptions

Question 7-1 Modify your code in order to enable interrupts in general, and timer interrupts in particular.

Question 7-2 On timer interrupts the processor must jump to a function `C_irq_handler()`, which does nothing for the moment.

Question 7-3 Implement `C_irq_handler()` so that it re-arms the timer, and returns correctly. Check step by step that your code comes back to the right place, and that the execution mode is restored correctly.

To help you in the implementation, add the following parameters to the command line that invokes Qemu: `-singlestep -d exec,int -icount 0`. The execution will slow down and the emulator may become unstable, but it will print each instruction executed, which can be helpful. Beware though, this feature is not always well supported in Qemu: one instruction may be printed several times and some instructions may be correctly executed but not printed. It is not a foolproof tool but combined with GDB, it should help you.

Some explanations:

- `-singlestep` tells Qemu to translate each instruction (rather than an entire basic block);
- `-d exec,int` forces printing each executed instruction and received interrupt;
- `-icount 0` slows down the emulation

Some explanations about instructions not correctly printed:

<https://lists.gnu.org/archive/html/qemu-discuss/2015-01/msg00014.html>

Exercise: preemptive scheduling

Question 7-4 Which registers have a hardware version different in the IRQ executino mode than in the SVC mode ? Use Figure 3.3.

Question 7-5 With this information in mind, modify your `C_irq_handler()` so that it does not simply returns to the interrupted process also triggers a context-switch. In order to help your sense of autonomy to grow, we propose you two means one more time :

1. The simplest way is to proceed as in `C_swi_handler()` for saving/restoring registers, and as in `yield()` for replacing the contexts on the stack. The stack pointer will be the one of the IRQ execution mode but that doesn't change anything to the mechanism. For this solution, program two functions having the same role that `void context_save_to_pcb()` and `void context_load_from_pcb()` but compatible with the IRQ mode rather than SVC.
2. You can also manage to avoid copy-pasting and invoking your existing code (`do_sys_yield()`). This requires to place the CPU in the right execution mode, and to correctly save/restore all registers of the user program.

At this stage, your OS must be able of executing correctly the following program. Check with `gdb` that the control flow goes through the 3 user processes turn by turn and indefinitely. In this goal, you can setup breakpoints in each of the three infinite loop, then use command `continue N` to ignore the *N* next stop on this breakpoint.

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

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

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

void kmain( void )
{
    sched_init();

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

    timer_init();
    ENABLE_IRQ();

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

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

Question 7-6 Check with `avec run-test.sh` and the program `kmain_preemptive.c` that your kernel successfully pass the following test:

- `round-robin-fairness-preemptive.gdb`

Chapter 8

Virtual memory management: paging

This chapter guide you in the implementation of a physical memory manager providing paging.

You will see later in chapter 10, how to isolate processes between them with the use of pages, but beware ! These two concepts are orthogonal :

- processes can be isolated without using memory paging ;
- memory can be managed through paging without isoler les processus entre eux.

The first sections of this chapter give details on the paging mechanism and how to handle this on the Raspberry Pi processor. So, you have a bit of reading before starting implementing. Don't worry, keep calm, it's not that painful...

8.1 Virtual memory

Figure 8.1 illustrates how, in lots of computing systems, one address manipulated by the CPU is translated to access at the right place in the physical memory where the information is really stored. réellement stocké l'information recherchée.

Addresses seen by the processor are called virtual address. It is translated thanks to the MMU (*Memory Management Unit*) in a physical address. The MMU looks up for the translation in the page table, located in RAM. In order not to spend too much time looking in the page table, the MMU use a translation cache: the TLB (*Translation Lookaside Buffer*). This component is located inside the MMU (so not in RAM) , and is composed essentially of an associative memory.

8.2 The co-processor

The microprocessor ARM1176JZF-S inside the Raspberry Pi has a *co-processor*, denoted CP15 (for *Control Processor*), allowing to control some hardware components: the cache system, the DMA, some performance management as well what we are now interested in : the MMU.

Controlling these components is performed by writing in the registers of the coprocessor, named *c0* to *c15*. But beware, it exists several hardware versions of each of these registers. In the following, we refer to a given hardware version of a given register thanks to this nomenclature: `<register number>/<register name>`, e.g. `c2/TTBRO`.

These registers are accessible via these two instructions:

- `MCR{cond} P15,<Opcode_1>,<Rd>,<CRn>,<CRm>,<Opcode_2>` Write in a register
- `MRC{cond} P15,<Opcode_1>,<Rd>,<CRn>,<CRm>,<Opcode_2>` Read a register

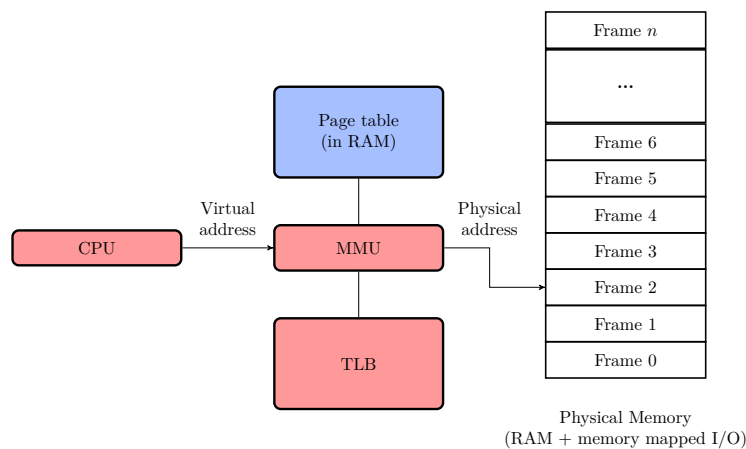


Figure 8.1: Simple illustration of the translation of a virtual address in a physical address. The page table is in RAM (so in some *frames*...); it is filled by the OS, but the look-up inside it is a hardware task.

The field **Rd** specifies the register of the processor (**r0** to **r12**) involved in the transfert. The field **CRn** specifies the *number* of the register of the coprocesseur to which we want access to, for instance **c0**. Fields **CRm** and **Opcode_2** indicate the operations we want to perform as well as the hardware register. For instance, the instruction `mcr p15, 0, r0, c2, c0, 0` specifies to write the value contained in **r0** to the register **c2/TTBR0**.

Configuring and monitoring the MMU is done through the one 32-bits, read-only register, one 32-bits write-only register, and 22 32-bits r/w registers. The coprocessor registers that you will need are detailed in the remainder of the document.

8.3 The MMU

8.3.1 Activating the MMU

The necessary instructions for configuring and activating the MMU are given in the Figure 8.2. This code is explained above.

The activation and the de-activation of the MMU are performed via setting one bit (the **M** bit, number 0) of the coprocessor control register (**c1/Control register**). However, in order to activate the MMU, the ARM documentation indicate that it is necessary to respect the following sequence :

1. Configure coprocessor registers.
2. Configure page table descriptors (cf. 8.3.2)
3. De-activate and invalidate the instruction cache. Then, it is possible to re-activate it when activating the MMU. Here is the code for this purpose : `mcr p15, 0, r0, c7, c7, 0` (you can see this code in file `init.s`).
4. Activating the MMU by setting the **M** bit of register **c1/Control Register**.

8.3.2 Configuring the page table descriptors

The coprocessor contains two registers pointing to 2 page tables: **TTBR0** and **TTBR1**. Also, it contains one control register: **TTBCTR**. The address space is divided in 2 regions :

- addresses between 0 and $1 \ll (32 - N)$ are translated by the page table pointed to by **TTBR0** ;
- addresses between $1 \ll (32 - N)$ and 4GB are translated by the page table pointed to by **TTBR1** ;

```

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: The two functions allowing to configure and activate the MMU

N is configured via c2/TTBCR. If N=0 (will be your case !), then the address space is handled via c2/TTBR0. Else, the address space reserved for the OS and the I/Os is located at the top of the address space (via c2/TTBR1 done).

In case of *TLB miss*, heavy bits of the logical address are used to decide whether TTBR0 or TTBR1 is used to translate this address.

8.4 Paging

Address translation can be done, on our processor, in ARMv5 or ARMv6 mode. The implementation that is provided to you is in ARMv6 mode. This is configured via the bit XP (23) of register c1/control register.

In the ARM processor of the Raspberry Pi, the MMU is responsible for filling the TLB with pairs <logical address, physical address>. Contrary to an Intel processor, the OS cannot fill directly the TLB with the desired pairs. When a logical address is not found in the TLB, the MMU walks through the page table in order to find the corresponding physical address. This page table is located in memory, it is filled by the operating

system, and allows to translate a logical address into a physical address regarding to the mechanism described in Figure 8.3. Beware ! The implementation doesn't manage sections nor supersections, only pages (of size 4KB).

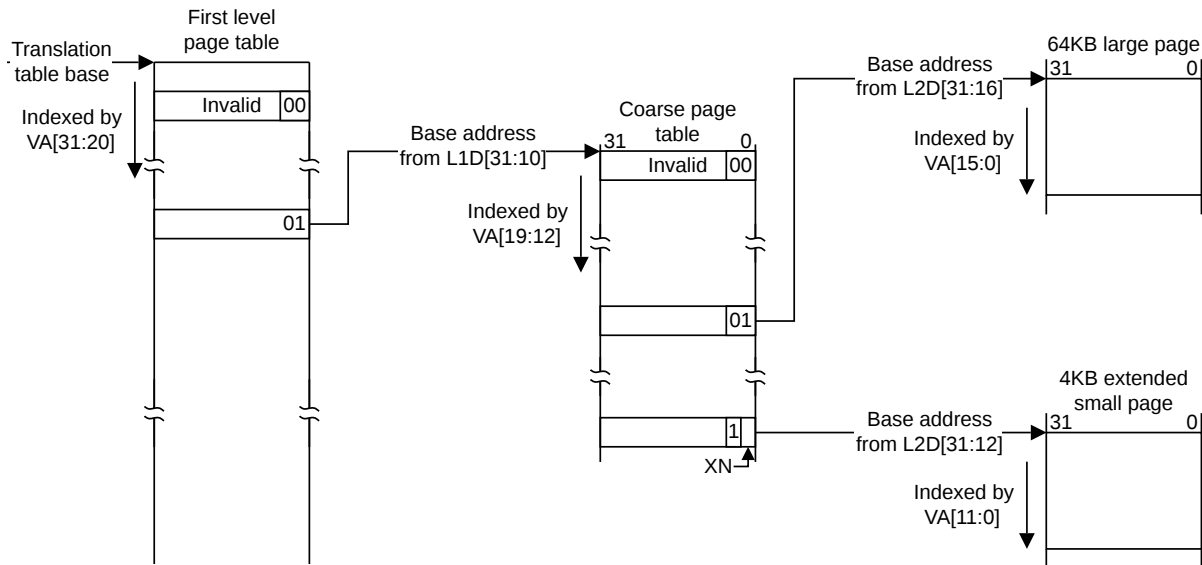


Figure 8.3: Vue générale de la translation d'adresse en mode ARMv6

The address translation mechanism is illustrated quite precisely, by Figure 8.4.

8.4.1 Level 1 page table

An entry in the level 1 page table is called a *page table entry*, or PTE. The format of a PTE is detailed in Figure 8.5. As a function of the two least significant bits, the entry looks like one of the two first lines (in your implementation, only those 2 lines are valid, we don't handles sections and supersections).

The bits of the second line have the following role :

P bit not supported by our processor processeur).

Domain the MMU performs a coarse grain control access: these bits indicate a *domain* to which belong the memory zone. Registre **c3** of the coprocessor associate some rights to each of these domains. Initialise them to '0' for all pages to belong to the same domain. More information in Section 6.5.1 of the processor documentation.

SBZ "Should Be Zero" (for ascendent compatibility).

NS Bit related to the *TrustZone* extension of ARMv6. Leave to '0' ("Secure"). More information in Section 6.6.3 of the processor documentation.

8.4.2 Level 2 page tables

The format of an entry in a 2nd level table is detailed in Figure 8.7. Once again, with respect to the two least significant bits, the entry must be read according to one of the line. But we don't manage pages of size 64kB. The bits of the only meaningful line have the following role :

nG (Not-Global) determines if this entry is globale, that is to say valid for all processues, or if it is valid for only one process. Initialise it to '0' as long as you do not deal with isolation between processes.

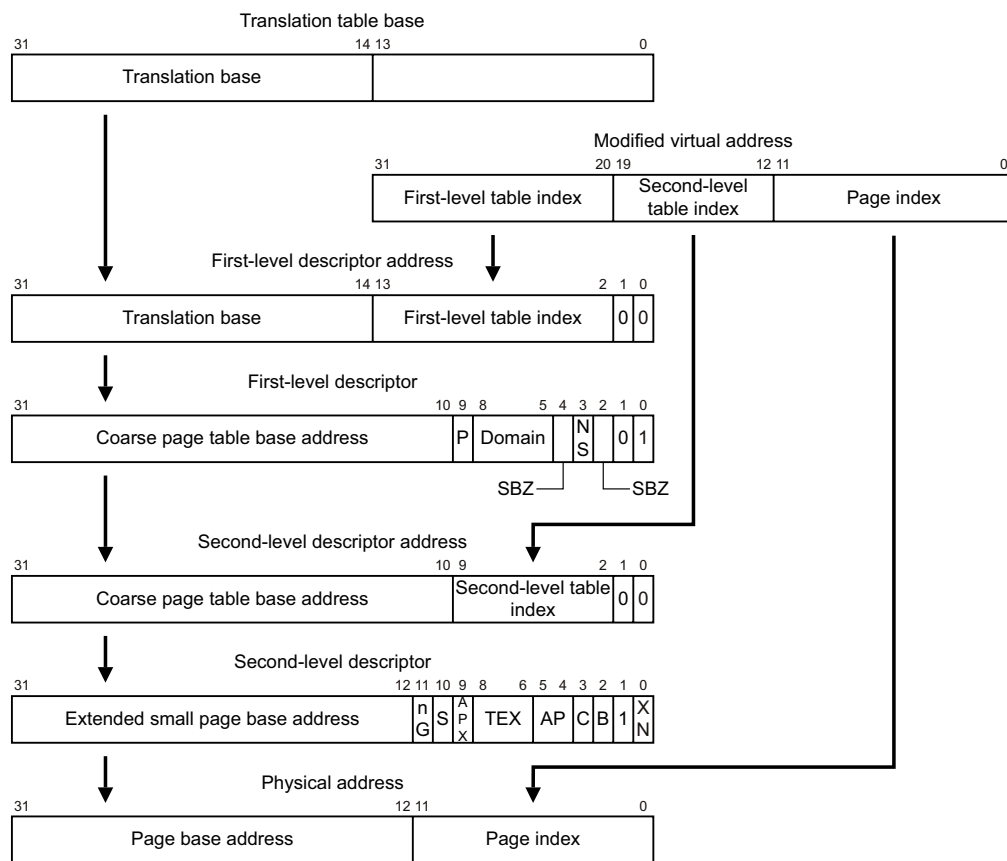


Figure 8.4: Address translation in ARMv6 mode, 4KB pages

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

Figure 8.5: An entry of the level 1 page table. Only lines 1 et 2 are useful because we won't use *Section* nor *Supersection* in this project. So, if bits 1 and 0 are equal to '00' and it means that no translation exist for the virtual address, or they are equal to '01', and line 2 (Coarse page table) describes the format of the entry.

S (Shared) détermine if the entry concerns a shared page ('1') or not ('0').

XN (eXecute-Never) determines if the page contains executable code ('0') or not ('1').

TEX, C et B Concerns cache policy and region type :

- initialise to TEX=000, C=0, B=1 if the page correspond to a peripheral mapped in memory ;
- else, initialise to TEX=001, C=0, B=0 (shareable memory, no caching)

More information in Section 6.6.1 of the doc on the processor.

APX If ForceAP=1, the OS can use this bit to optimise the replacement of pages. But, in your implementation, ForceAP=0 so the meaning of the APX bit depends on AP, cf. below ; initialise for instance to '0'.

AP With APX, the meaning of these bits is given by Table 8.6. Initialise for instance to '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 8.6: Access permission

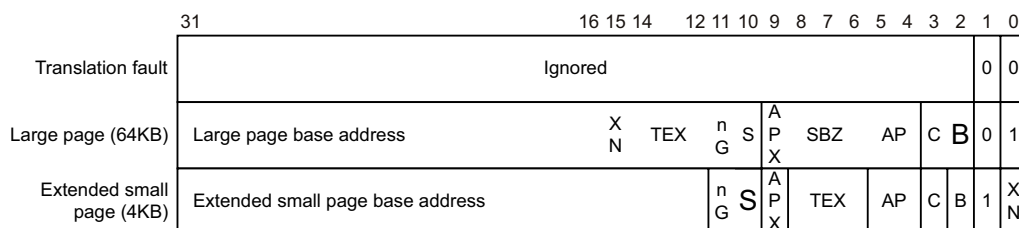


Figure 8.7: An entry in a 2nd level page table. If bits 1 and 0 contains '00', it is a translation fault. If they contain '01', the entry corresponds to a Large page, 64 Ko, not useful in this project. If the bit 0 is set ('1'), then the entry corresponds to a 4kB page (that we use in this project), and line 3 describes the format of the entry.

8.5 Exercise : paging without dynamic allocation

In order to implement your paging mechanism, you need to:

- initialise the page table: fill the 1st-level et necessary 2nd-level tables,
- initialise the **frame table**: each bit b (or more conveniently but less efficiently, byte) of this data structure indicates if the b -th frame is in use or not,
- activate the MMU,
- provide allocating/liberation primitives of pages.

The sequence of questions below guides you in the implementation of this mechanism. Your code should be put mainly in a file `vmem.c` (and the associated `.h` file).

Question 8-1 Use the explanations of previous sections, in particular Figure 8.4, to perform those tasks:

- Find the size of a page and define the macro `PAGE_SIZE` that is worth this value,
- How many entries contains one 2nd-level page table ? Définie the macro `SECON_LVL_TT_COUNT` that is worth this value.
- Define the macro `SECON_LVL_TT_SIZE` that is worth the size in bytes on one entire 2nd-level page table,
- How many entries contains one 1st-level page table ? Define the macro `FIRST_LVL_TT_COUNT` that is worth this value,
- Define the macro `FIRST_LVL_TT_SIZE` that is worth the size in bytes of the 1st-level page table.

Question 8-2 Using the previous explanations, in particular Figure 8.7, write on paper the bitfield representing one entry of a 2nd-level page table, considering that this page is allowed to be read and written by everyone.

Question 8-3 Give the line of code that initializes correctly the variable `uint32_t device_flags` that is worth the value for entries of 2nd-level page tables pointing to addresses between `0x20000000` et `0x20FFFFFF`, that is to say the region where peripherals are mapped.

Question 8-4 To allocate pages tables (1st and et 2nd-levels), you have to use the primitive `uint8_t* kalloc_aligned(unsigned int size, unsigned int pwr_of_2)` of file `kheap.c`, that allocates `size` bytes at an address aligned to $2^{pwr_of_2}$. Why ?

Question 8-5 You now have to implement the function `unsigned int init_kern_translation_table(void)` that allocates then initialises the whole page table. Given that you don't relocate code and static data, your page table must translate a logical address by itself for all logical addresses where kernel data structures are located. For instance, the physical address corresponding to logical address `0x8000` is worth: `0x8000`. All other frames are empty at the beginning of the kernel execution. You may translate in this way:

- logical address = physical address for any physical address between `0x0` and `__kernel_heap_end__` (have a new look at Figure 3.4 if needed) ;
- logical address = physical adress for any physical address between `0x20000000` and `0x20FFFFFF` ;
- else, translating fault (for the moment).

Question 8-6 In a function `void vmem_init()`, initialise the physical memory, configure the MMU, activate interrupts as well as *data aborts*, and finally activate the MMU. Use the functions given in Figure 8.2. The call to `vmem_init()` is done directly in `sched_init()`, in this way :

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

Question 8-7 Test your paging mechanism. Help yourself with function `vmem_translate()` given in Figure 8.8, that translates one virtual address in a physical address in the same manner as the MMU.


```

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

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

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

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

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

    table_base = table_base & 0xFFFFC000;

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

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

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

    /* Second level descriptor */
    second_level_table = first_level_descriptor & 0xFFFFFC00;
    second_level_descriptor_address = (uint32_t*) (second_level_table | (second_level_index << 2));
    second_level_descriptor = *((uint32_t*) second_level_descriptor_address);

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

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

    return pa;
}

```

Figure 8.8: This function translates the given virtual address in a physical address by looking into the page table of the process given as parameter, exactly as would do the MMU. Use it for debug purpose.

Chapter 9

Dynamic allocation

Ce chapitre vous guide dans l'implémentation des deux appels système suivants :

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

9.1 Exercice : compréhension

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

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

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

Question 9-1 En regardant la figure 3.4, on voit que l'espace d'adressage physique accessible par l'ARM va de 0 à 0x20FFFFFF. Quelle est la taille de l'espace d'adressage physique ? Et quelle est la taille de l'espace d'adressage logique ?

Question 9-2 Du coup, quelle taille fait la table d'occupation des frames ?

Question 9-3 Allouez puis initialisez la table d'occupation des frames, dans `vmem_init()`. À ce stade, certaines frames doivent être marquées occupées car correspondent au code, données, structures noyaux etc.

Question 9-4 Implémentez la fonction `uint8_t* vmem_alloc_for_userland(pcb_t* process)` qui alloue une page dans l'espace d'adressage du processus donné en paramètre.

Question 9-5 On veut maintenant pouvoir allouer plus d'une page à un processus. Les pages allouées doivent-elles être situées consécutivement dans l'espace d'adressage virtuelle de ce processus ? Et les frames ?

9.2 Exercice : implémentation

Question 9-6 Ajoutez un paramètre `unsigned int size` à la fonction `vmem_alloc_for_userland(...)`. Elle alloue maintenant `size` octets dans l'espace d'adressage du processus donné en paramètre. Contentez-vous d'arrondir au nombre de page supérieur nécessaire.

Question 9-7 Implémentez la fonction `void vmem_free(uint8_t* vAddress, pcb_t* process, unsigned int size)` qui libère les pages allouées par `vmem_alloc_for_userland(...)`.

Question 9-8 Mettez en place l'appel système `sys_mmap()`. La fonction `do_sys_mmap()` (côté noyau donc) se sert bien sûr de `vmem_alloc_for_userland(...)`.

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

Question 9-10 Testez vos appels système `sys_mmap()` et `sys_munmap`.

Chapter 10

Process isolation

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.

10.1 Exercice : gérer une faute de traduction

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 10-1 Donnez une adresse à laquelle le processeur ne peut pas accéder une fois les processus créés et la mémoire initialisée.

Question 10-2 Déclarez et définissez une fonction `data_handler` et faites en sorte que celle-ci soit exécutée sur une interruption *data abort*. Vérifiez cela en tentant d'accéder à l'adresse que vous avez donné précédemment

Question 10-3 Dans ce traitant, identifiez la cause de l'interruption et terminez l'exécution de votre noyau. Cette fonction vous sera utile pour déboguer la suite.

10.2 Exercice : compréhension

Une solution simple pour isoler les processus entre eux est que chaque processus ait sa propre table des pages. Quelques précisions :

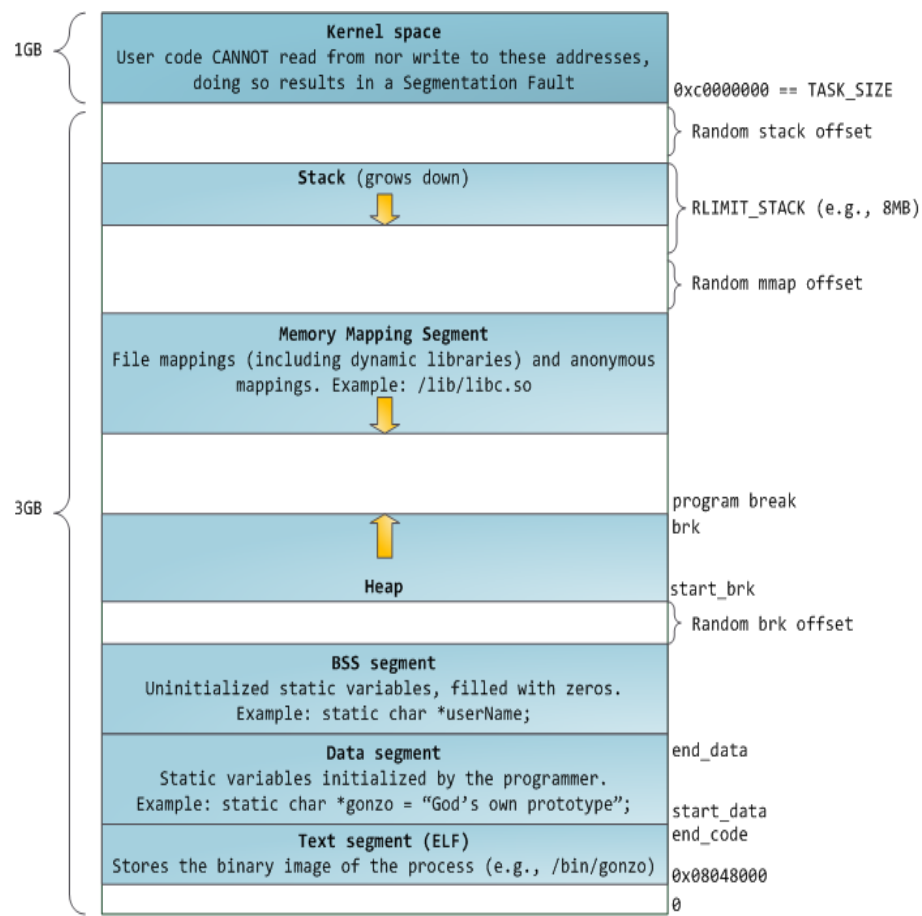


Figure 10.1: Organisation of memory space in Linux

- Lors d'un context-switch, l'OS doit invalider toutes les entrées de la TLB et fait pointer `c2/TTBR0` sur la table des pages du processus élu.
- Le pointeur vers la table des pages est stocké dans le PCB des processus. Le pointeur de table des pages de l'OS est stocké dans une variable globale.
- Pour invalider les entrées de la TLB, l'instruction suivante suffit : `MCR p15,0,<Rd>,c8, c6,0, peu importe <Rd>`.
- Si un processus tente d'accéder à une page qui n'existe pas dans l'espace d'adressage du processus, la MMU va générer une *Translation fault*.

Question 10-4 À des fins d'isolation, le PCB de chaque processus doit-il être alloué dans l'espace d'adressage de chaque processus, ou continuer à résider uniquement dans l'espace d'adressage du noyau ? Pourquoi ?

Question 10-5 À des fins d'isolation, la pile d'exécution de chaque processus doit-elle être allouée dans l'espace d'adressage de chaque processus, ou continuer à résider uniquement dans l'espace d'adressage du noyau ?

Question 10-6 Lorsqu'un processus est créé, une table des pages pour ce processus doit être créée et initialisée. Quelles traductions doit-elles contenir ?

10.3 Exercice : implémentation

Question 10-7 Modifiez la fonction `create_process()` de manière à allouer pile et PCB comme il faut.

Question 10-8 Modifiez la fonction `create_process()` de manière à allouer et initialiser la table des pages de chaque processus.

Question 10-9 Montrez que pour certaines adresses logiques, la traduction qui est faite donne deux résultats différents pour deux processus différents, ainsi que pour l'OS.

Question 10-10 Votre traitant d'interruption `data_handler` doit maintenant gérer les fautes d'accès. Par exemple en terminant le processus fautif. C'est la fameuse *segmentation fault* !

Question 10-11 Avec ce que vous savez maintenant du fonctionnement de la MMU et du remplissage de la TLB : pourrait-on isoler les processus entre eux sans paginer sur le processeur du Raspberry Pi ?

Chapter 11

Shared memory

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.

Chapter 12

Process synchronization

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 : <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 11.

```
#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 */
    }
}
```

Frame 11: 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 12-1 Donnez la déclaration de la structure de donnée associée à un sémaphore.

Question 12-2 Proposez une implantation de la primitive

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

Question 12-3 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 12-4 Proposez une implantation des deux primitives

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

12.1 Prévention des interblocages

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

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

L'interface de manipulation des verrous est la suivante :

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

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

Exercice: Le dîner des philosophes

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

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

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

Question 12-5 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 12-6 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);
```

Chapter 13

Allocation dynamique de mémoire

13.1 Une première bibliothèque standard

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

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

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

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

13.1.1 Principe

Lors de la création d'un processus, un espace mémoire lui est alloué, contenant la pile d'exécution de ce processus, le code de celui-ci, les variables globales, ainsi que le tas, dans lequel les allocations dynamiques effectuées au travers `gmalloc()` sont effectuées. Ce tas mémoire est accessible le champs `heap` de la structure associé au processus (voir le chapitre 5). Au début de l'exécution du processus, ce tas ne contient aucune donnée. Il va se remplir et se vider au gré 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 pointerait sur le premier.

13.1.2 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.

13.1.3 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.

13.2 Optimisations de la bibliothèque

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

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

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

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

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

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

Part I

Annexes

Appendix A

Interruptions

Une *interruption matérielle* est un signal (un '1' sur un fil, rien de plus) positionné par un périphérique, indiquant au processeur qu'un évènement a eu lieu nécessitant l'intervention du système d'exploitation. De multiples interruptions peuvent avoir lieu dans un système informatique tel que le Raspberry Pi, et à chacune est associé un numéro : 0 pour *reset*, ...

La *table des vecteurs d'interruption* est une structure de données, qui associe à chaque numéro d'interruption, un moyen de traiter l'interruption. Ce moyen peut, dans le cas général, être de deux types :

L'adresse du *traitant d'interruption*, ou *routine d'interruption* ou *ISR* (Interrupt Service Routine. Lorsque le processeur détecte qu'une interruption a été levée (il effectue cette vérification à chaque cycle fetch-decode-execute), il branche à cette adresse.

Une instruction qui est exécutée par le processeur lorsque l'interruption est détectée.

La table des vecteurs d'interruption est placée à une adresse fixe `it_vec_table`, de sorte que le CPU trouve l'information (adresse ou instruction) à l'adresse `it_vec_table + num` lorsqu'une interruption survient, pour peu que cette table ait été initialisée.

Sur les systèmes ARM, c'est la deuxième solution qui est implémentée. Pour ce TP, nous avons initialisée la table pour vous. 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 CPU boucle sur la même instruction, située au label `irq` du fichier `init.s`.

Appendix B

Gestionnaire de mémoire physique simple

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

Appendix C

Installation des outils nécessaires à la réalisation du projet

C.1 Installation de Qemu pour Raspberry Pi

C'est assez facile maintenant, le code d'émulation a été intégré à la branche principale de Qemu. Donc téléchargez et compilez simplement les sources de Qemu dans une version >2.6. Si vous voulez juste compiler Qemu pour l'émulation de la Raspberry Pi (et pas pour pouvoir émuler x86 et plein d'autres cibles), vous pouvez l'indiquer lors de la configuration :

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

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

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

C.2 Installation de GCC pour ARM

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

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

C.3 Installation de GDB pour ARM

Linux Attention la version 7.8 de GDB contient des bugs qui ne vous autoriseront pas à utiliser le système de test que nous vous fournissons. Il vous faut dans ce cas recompiler gdb. C'est ultra simple :

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

- Ajoutez `<install_dir>/bin` à votre PATH

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

Appendix D

FAQ

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

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

The assembly code seems sub-optimal, why ?

Have you checked the optimisation level ? -O0 ?

Appendix E

Appendix about virtual memory

E.1 Detail concerning register *c1/Control register*

Explanation of the value of the *c1/Control* register. First, let's see the binary value:
 $75864189 = 0 \times 485987d = (00000100100001011001010001111101)_2$

Then, role of each bit:

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

- 3 W : 1 \rightarrow “Not implemented”
- 2 C : 1 \rightarrow Cache de données niveau 1 activé
- 1 A : 0 \rightarrow Détection de non-alignement non activé
- 0 M : 1 \rightarrow **Activation de la MMU**

E.2 Isolation in ARM’s processors

E.2.1 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.

Appendix F

Aide-mémoire gdb

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

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

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

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

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

Essential Commands

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

Starting GDB

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

Stopping GDB

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

Getting Help

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

Executing your Program

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

Shell Commands

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

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

Breakpoints and Watchpoints

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

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

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

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

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

Program Stack

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

Execution Control

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

Display

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

Automatic Display

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

Expressions

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

show values [<i>n</i>]	show last 10 values [or surrounding <i>\$n</i>]
show conv	display all convenience variables

Symbol Table

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

GDB Scripts

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

Signals

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

Debugging Targets

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

Controlling GDB

set <i>param value</i>	set one of GDB's internal parameters
show <i>param</i>	display current setting of parameter

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

Working Files

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

Source Files

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

GDB under GNU Emacs

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

GDB License

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

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

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

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

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

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

Glossary

attribut

Information donné au compilateur pour l'aider à optimiser sa génération de code, ou lui indiquer de compiler le code d'une certaine manière. Il existe des attributs de fonction, de variable, et de type. [13](#)

context switch

TODO [28](#)

coroutine

TODO [28](#)

cross-compiler

TODO [6](#)

dispatcher

The part of the operating system responsible for saving and restoring execution contexts [27](#)

election

TODO [33](#)

epilogue

Quelques instructions assembleur générées par le compilateur à la toute fin de chaque fonction, afin de restituer la frame de la fonction appelante. [13](#)

exception

TODO [16](#)

execution context

Une photo des registres CPU prise à un certain point du programme. Recharger un contexte dans le CPU permet de reprendre l'exécution du programme à partir de ce point (puisque le registre PC fait partie du contexte) [28](#)

execution mode

TODO [74](#)

frame table

A simple data structure (*e.g.* a bitfield) used by the kernel to know which physical pages are in use [47](#)

interrupt handler

TODO [13](#)

interrupt handler, or ISR (*Interrupt Service Routine*)

procedure invoked automatically by the CPU when it receives an [interrupt request](#). See also the insert [10](#) page [38](#). [37](#), [38](#)

interrupt request, or IRQ

Hardware signal sent to the CPU by a peripheral to notify an (hardware) event. [37](#), [74](#)

microcontroller

TODO [10](#)

notion

Une notion, en philosophie, est une connaissance élémentaire, souvent tirée d'observations empiriques. [4](#)

PCB

Process Control Block [28](#)

privilege

TODO. Et dire que ça inclut le [execution mode](#), mais pas seulement. [15](#), [19](#)

process

Programme en cours d'exécution [27](#)

process state

TODO [28](#)

prologue

Quelques instructions assembleur générées par le compilateur au tout début de chaque fonction, afin de préparer l'espace dédié à l'exécution de la fonction sur la pile d'exécution. [13](#)

round-robin

Politique d'ordonnancement consistant à exécuter les processus chacun leur tour [33](#)

scheduler

TODO [32](#)

system call

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. [19](#)