Programming the kernel of an Operating System

Philippe Marquet, Gilles Grimaud, Samuel Hym
Florian Vanhems, Clément Ballabriga, Giuseppe Lipari

9 janvier 2024

Table des matières

1 Introduction to the programming environment

You have been writing software for a few years now, and yet many of you have never written a "first software", that is, a software that runs first, when the machine starts up. That's what we're going to do now.

1.1 A minimal kernel

In the following repository my-kernel you will find such a minimal software. The `Makefile` compiles the program in `src/main. c` and produces the iso image of a bootable disk on x86 architecture. To run this «first software» you either :

1. flash the iso image to a persistent media (flash drive, disk) ; or
2. use a virtual machine to start the iso image.

In the phase of development and testing we favour this second solution, because it is more comfortable than the first (compilation/run cycles are shorter and the debugging is easier).

To start a command line iso image, we recommend the use of qemu :

```
qemu-system-x86_64 -boot d -m 2048 -cdrom mykernel.iso -curses
```

This command is directly available in the `Makefile` with :

```
make run
```

To produce an iso image from an executable binary we use the grub utility. You can see the method that we suggest in the `Makefile`. To use it do simply :

```
make
```

The Makefile first compiles your source file into a `.o`, then it **links** your `.o` with some other elements and produces a file binary, and finally, it uses the `grub-mkrescue` and `xorriso` utilities to produce the iso file. Note that, technically speaking, the first program executed is the **bios**, which loads the `grub` software ; then `grub` displays a menu that allows you to load and launch your software (as if it were running a kernel of an operating system).

1.2 Prerequisites

Note that for this to work properly you will need the following tools :
— `gcc`
— `grub-common`
— `xorriso`
— `qemu`

These tools are installed on the machines in the training rooms. You can connect to it (via VPN) with :

```
ssh @a<#salle>p<#poste>.fil.univ-lille1.fr
```

Otherwise you can install the tools on your favourite Linux distribution. Installation on Microsoft Windows and MacOS X is not impossible, but is more delicate... Some explanations on how to set up the development environment are given in the `README.md` present in the `my-kernel` repository.

1.3 Minimal I/O library

As explained earlier, a set of `.c` files are compiled to produce the iso image. However it is important to note that the C language used here is C "bare-metal". Unlike a classic C program, it does not run "above" a operating system and you therefore have none of the libraries you are used to. So functions such as `printf`, `malloc`, `fopen`, `fork`, `exit`... which are implemented by the glibc and using services from your system are not available.

To help you in the development, the C files that we provide realize some basic functions. The first is to initialize the basic mechanisms of the Intel architecture. One of these mechanisms is the GDT, and the initialization that we proposed provides unrestricted access to all machine memory. The second mechanism is the IDT, which manages the interrupt mechanism on Intel microprocessors.

Second, we provide a minimalist implementation of basic output functions, like `putc()`, `puts()` and `puthex()`, in files `minilib.h/.c`. With these functions you can print characters on the screen, and thus display information. The screen is configured to work in text mode, with 80 columns and 25 rows. In this video mode (defined in VGA standards by IBM), the graphic controller shares a segment of its memory with the microprocessor. From the processor's point of view, the memory of the graphic controller is available starting at address `0xA0000`, but in mode text the information used by the graphics card to produce the image are accessible to the microprocessor from the address `0xb8000`. In addition, the graphic controller can handle the blinking of a cursor via specific hardware registers. The implementation of the function `putc()` programs this register to manage the hardware cursor.

To program the registers of any hardware device, Intel architectures offer two machine instructions : `in` and `out`.

The code base that you find in the repository defines two C functions, `unsigned char _inb(int port);` and a function C `void _outb(int port, unsigned char val);` in file `include/ioport. h`. The functions in `minilib.c` use these two functions to control the position of the hardware cursor.

2 Return to a context

For some applications, program execution must be recovered at a particular point. A point of execution is characterized by the current state of the call stack and processor registers ; we call this a **context**.

2.1 The standard Unix library

The `setjmp` function of the standard library stores the current context in a type variable `jmp_buf` and returns 0.

The `longjump` function reactivates a context previously saved. At the end of this reactivation, `setjmp` "returns" a non-zero value.

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

```
void longjmp(jmp_buf env, int val);
```

2.2 Exercise : Description of the `setjmp()` / `longjmp()` mechanism

1. This involves understanding the behaviour of the following program :

```
#include <setjmp.h>
#include <stdio.h>

static int i = 0;
static jmp_buf buf;

int main()
{
    int j;

    if (setjmp(buf))
        for (j=0; j<5; j++)
            i++;
    else {
        for (j=0; j<5; j++)
            i--;
        longjmp(buf,~0);
    }
   printf("%d\n", i );
}
```

2. and of the following variant

```
#include <setjmp.h>
#include <stdio.h>

static int i = 0;
static jmp_buf buf;

int main()
{
    int j = 0;

    if (setjmp(buf))
        for (; j<5; j++)
            i++;
    else {
        for (; j<5; j++)
            i--;
        longjmp(buf,~0);
    }
   printf("%d\n", i );
}
```

2.3 Exercise : Read the documentation

Explain why the following program is wrong :

```c
#include <setjmp.h>
#include <stdio.h>

static jmp_buf buf;
static int i = 0;

static int cpt()
{
    int j = 0;

    if (setjmp(buf)) {
        for (j=0; j<5; j++)
            i++;
    } else {
        for (j=0; j<5; j++)
            i--;
    }
}

int main()
{
    int np = 0 ;

    cpt();

    if (! np++)
        longjmp(buf,~0);

    printf("i = %d\n", i );
}
```

You can also enjoy the following excerpt from the manual page of `setjump(3)` :

> setjmp() and sigsetjmp make programs hard to understand and maintain. If possible, an alternative should be used.

2.4 Exercise : Return in an execution stack

Modify the following program that calculates the product of a set integers read on the standard entry, and returns in the context of the first invocation of `mul()` if a null value is encountered : the product of terms of which one is null is null.

```c
static int mul(int depth)
{
    int i;

    switch (scanf("%d", &i)) {
        case EOF :
            return 1; /* neutral element */
        case 0 :
            return mul(depth+1); /* erroneous read */
        case 1 :
            if (i)
```

```
                return i * mul(depth+1);
            else
                return 0;
    }
}

int main()
{
    int product;

    printf("A list of int, please\n");
    product = mul(0);
    printf("product = %d\n", product);
}
```

3 Execution context

To be executed, the procedures of a program written in the C language are compiled into machine code. This machine code uses, during its execution, the processor registers and an execution stack.

In the case of a program compiled for Intel x86 processors, the top of the runtime stack is pointed by the 32 bit register `esp` (*stack pointer*). Also the Intel architecture defines a register designating the base of the stack, the register `ebp` (*base pointer*).

These two registers define two addresses within the zone reserved for the execution stack, the space that separates them is the window associated with the execution of a function (*frame*) : `esp` points to the top of this area and `ebp` points to the base.

Roughly, when a function is called, the arguments of the function are put onto the stack, then the function is called (saving the return address on the stack), and then the function starts by allocating the local variables again on the stack. Note although the Intel stack is organized according to a descending order : pushing something on the stack *decreases* the value of esp. Please refer to the `archi.pdf` for more details on the way a function call is translated and how the stack is used to pass parameters and store local variables.

Saving the values of the two registers `esp` and `ebp` is enough to store a context. Restoring the values of these registers allows to *change context*. Pay attention : once these registers are restored within a function, access to automatic variables (local variables allocated in the execution stack) are no longer possible, these accesses being realized by indirection from the value of the registers.

Access to processor registers may be through the inclusion of assembly code within code C.

3.1 Assembly code

The GCC compiler allows the inclusion of assembly code within code C via the `asm()` keyword. In addition, operands may be expressed in C.

The following C code allows you to copy the contents of variable `x` in register `eax`, then transfer it in variable `y` ; the value of the register `eax` is modified by the assembly code :

```
int main()
{
    int x = 10, y;
    asm("movl %1, %%eax" "\n\t" "movl %%eax, %0"
        : "=r"(y)   /* y is the output operand */
        : "r"(x)    /* x is the input operand */
```

```
            : "%%eax"; /* eax is a clobbered register */
    }
```

Attention, this construction is highly non portable and is not standard ISO C ; therefore the `-ansi` option of `gcc` cannot be used. The general scheme is as follows :

```
asm ( "code assembleur"
      : output operands       (optional)
      : input operands        (optional)
      : list of clobbered registers (optional));
```

The assembler code can reference operands by their index (from 0) in the operand list as `%i`.

Assembly instruction. The most commonly used x86 statement in this context is `movl` which copies the second operand into the first.

Constraints. Each operand can be associated with constraints. For example, we can specify that the value must be in a register (`r`). GCC% will make the necessary for these constraints be met. The = sign indicates that the operand is output.

Refer to the online x86 assembler tutorial available at `http://www-106.ibm.com/developerworks/linux/library/l-ia.html` or GCC documentation available at `http://gcc.gnu.org/onlinedocs/`.

3.2 First practical realisation

Before starting the following exercises, we suggest that you first provide a way to display the value of the registers `esp` and `ebp` of the current function.

— Observe these values on a simple program consisting of nested function calls ;
— Compare these values with the addresses of the first and last local automatic variables declared in these functions.
— Compare these values with the addresses of the first and last parameters of these functions.
— Explain.

3.3 Exercise : try / throw

A set of primitives is defined to return to a previously stored context in a value of type `struct ctx_s`.

```
/* A function that returns an int from an int */
typedef int (func_t)(int);

int try(struct ctx_s *pctx, func_t *f, int arg);
```

This first primitive will execute the `f()` function with the `arg` parameter. If necessary, the programmer can return to the call context of the `f` function stored in `pctx`. The value returned by `try()` is the value returned by the `f()` function.

```
int throw(struct ctx_s *pctx, int r);
```

This primitive will return in a context of a call of a function previously stored in the context `pctx` by `try()`. The `r` value will then be that "returned" by invocation of the function through `try()`.

— Define the data structure `struct ctx_s`.
— The `try()` function saves a context and calls the function passed as a parameter. Give an implementation of `try()`. It is recommended to print the values of the registers `esp` and `ebp` for debug.

— Propose an implementation of the `throw()` function. The `throw()` function restores the context corresponding to that saved by the `try()` function. This `try()` function had to return a value. Instead, the value we will return is the value passed as `throw()`.

Implement the above functions and test their behavior on a variant of the exercise program 2.4.

Observe the execution of this program with a debugger ; in particular set a breakpoint in `throw()` and continue the execution step by step once this breakpoint is reached.

4 Creating an execution context

A runtime context is therefore mainly a runtime stack, the one that is manipulated by the compiled program via the `esp` and `ebp` registers.

This stack is, in itself, only a memory zone whose basic address is known. To be able to restore an execution context you must also know the value of the registers `esp` and `ebp` that identify a frame in this stack.

When a program ends, its execution context must no longer be used. Finally, a context must be able to be initialized with a function pointer and a pointer for the function arguments. This function will be the one that will be called when the context is first activated. It is assumed that the argument pointer is of type `void *`. The invoked function will be free to perform a casting of the pointed variable to the expected type.

4.1 Memory segment

The execution stack is associated with a virtual memory segment pointed by the `ss` (*stack segment*) register. `my-kernel` adopts the Linux kernel strategy : they use the same segment for the call stack and for storing the usual data (global variables and C allocation heap). This is why it is possible to allocate a memory space and use it to place an execution stack. With a different configuration of the segments, the programs we propose would be incorrect.

4.2 Exercise : structure

— Declare a new type `func_t`, used by the context to store the entry point of the function (it returns nothing and takes as parameter the argument pointer).

— Extend the data structure `struct ctx_s` of the previous exercise adding the fields necessary to describe such a context.

4.3 Exercise : initialisation

Propose a function

```
int init_ctx(struct ctx_s *ctx, func_t f, void *args);
```

which initializes the context `ctx` with a stack of `STACK_SIZE` bytes, with `STACK_SIZE` a globally declared constant (for example, equal to 4096 bytes). During its first activation this context will call the function `f` with the parameter `args`.

5 Context change

5.1 Coroutines

We're going to implement a coroutine mechanism. Coroutines are procedures that run in separate contexts. Thus a `ping` procedure can "give back the hand" to a `pong` procedure without finishing, and the `pong` procedure can do the same with the `ping` procedure. Therefore, `ping` will resume its execution in the context in which it was before passing the hand. Our table-tennis can also be played with more than two ...

```
struct ctx_s ctx_ping;
struct ctx_s ctx_pong;

void f_ping(void *arg);
void f_pong(void *arg);

void app_main()
{
    init_ctx(&ctx_ping, f_ping, NULL);
    init_ctx(&ctx_pong, f_pong, NULL);
    switch_to_ctx(&ctx_ping);
}

void f_ping(void *args)
{
    while(1) {
        putc('A') ;
        switch_to_ctx(&ctx_pong);
        putc('B') ;
        switch_to_ctx(&ctx_pong);
        putc('C') ;
        switch_to_ctx(&ctx_pong);
    }
}

void f_pong(void *args)
{
    while(1) {
        putc('1') ;
        switch_to_ctx(&ctx_ping);
        putc('2') ;
        switch_to_ctx(&ctx_ping);
    }
}
```

The execution of this program produces endlessly :

```
A1B2C1A2B1C2A1...
```

This example illustrates procedure

```
void switch_to_ctx(struct ctx_s *ctx) ;
```

which simply saves the stack pointers in the current context, then sets the context whose address is passed as a parameter as a new current context, and restores the stack registers. So when this procedure executes `return;` it "returns" in the execution context passed as a parameter.

If the context is activated for the first time, instead of returning with a `return;` function calls `f(args)` to «launch» the first execution... Be careful, after the stack registers have been initialized on a new runtime for the first time, the local variables and arguments of the `switch_to_ctx()` function are unusable (they were not saved on the execution stack).

5.2 Exercice : switch

Propose an implementation of the procedure : `switch_to_ctx()`.

6 Scheduling

The primitive `switch_to_ctx()` of the coroutine mechanism requires the programmer to explicitely specify the new context to be activated. Moreover, once the execution of the function associated with a context is complete, it is not possible for the primitive `swicth_to_ctx()` to activate another context ; no other context is known.

One of the objectives of the scheduler is to choose, during a change of context, the new context to activate. For this it is necessary to keep track of all known contexts ; for example in the form of a circular list of contexts.

We propose a new interface with which contexts are no longer directly manipulated in «user space» :

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

The `create_ctx()` primitive creates a new context for the `f=function, and inserts it into the list of "active" contexts. The primitive =yield()` allows the current context to pass the hand to another context ; the latter being determined by the scheduler.

6.1 Linked lists in the Linux kernel

The Linux kernel uses a library of data structures and associated functions to implement chained lists. This library is optimized to maximize speed and reduce memory size. A minimalist version of this library is available in the `examples/list file. h`, and an example of use in the `examples/list-example file. c`.

Online resources are available to better understand how this library works, such as :

— The Linux Kernel Documentation : `https://www.kernel.org/doc/html/v4.15/core-api/kernel-api.html`

— KernelNewbies/FAQ : Linked Lists : `https://kernelnewbies.org/FAQ/LinkedLists`

6.2 Exercise : creating context

— Extend the `struct ctx_s` data structure to create the chained list of existing contexts, using the Linux chained list library.

— Modify the primitive `init_ctx()` into a primitive `create_ctx()` to set up this chaining.

— Treat the other primitives accordingly.

6.3 Exercise : yield()

Give an implementation of `yield()`.

7 Synchronisation

A synchronization mechanism between contexts is introduced using semaphores. A semaphore is a compound data structure

— a counter ;
— a list of pending contexts on the semaphore.

The counter can take positive, negative, or null integer values. When creating a semaphore, the counter is initialized to a given value positive or null ; the queue is empty.

A semaphore is manipulated by the following two *atomic* actions :

— `sem_down()` (traditionally also called `wait()` or `P()`). This action decrements the counter associated with the semaphore. If its value is negative, the calling context is inserted in the waiting queue and the scheduler is called to change context.
— `sem_up()` (also named `signal()`, `V()`, or `post()`). This action increments the counter. If the counter is negative or zero, a context is selected in the waiting queue and becomes active ; the scheduler is called.

Two uses are made of semaphores :

— the protection of a shared resource (for example access to a variable, a data structure, a printer...). It is called a mutual exclusion semaphore ;
Typically the semaphore is initialized to the number of contexts that can simultaneously access the resource (for example 1) and each access to the resource is framed by a couple

```
sem_down(S);
<resource access>
sem_up(S);
```

— context synchronization (one context must wait for another to continue or start its execution). For example a context 2 waits for the termination of a first context to begin.) A semaphore is associated with the event, for example `findupremier`, initialized to 0 (the event did not take place) :

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

Often, the positive value of the counter can be assimilated to the number of contexts that can freely acquire the resource ; and the negative value of the counter can be assimilated to the number of blocked contexts waiting to use the resource. A classic example is given in the next section.

7.1 Producer / consumer

A solution to the problem of the consumer producer by means of semaphores is given here. The two typical uses of semaphores are illustrated.

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

struct sem_s mutex, vide, plein;

sem_init(&mutex, 1);                  /* controle d'acces au tampon */
```

```
sem_init(&vide, N);                    /* nb de places libres */
sem_init(&plein, 0);                   /* nb de places occupees */

void producteur (void)
{
  objet_t objet ;

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

void consommateur (void)
{
  objet_t objet ;

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

Convince yourself that it is not possible for the producer (resp. the consumer) to take the semaphore `mutex` before the semaphore `full` (resp. `empty`).

Test your semaphore implementation on an example like this.

Add a time loop in the producer so that the context change can take place before the buffer is full.

Try to reverse access to semaphores `mutex` and `full=/=empty` ; what do you see ?

7.2 Exercise : implementation of semaphores

7.3 Exercise : implementation of semaphores

— Provide the declaration of the data structure associated with a semaphore.

— Propose an implementation

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

— Propose implantation of both primitives

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

8 A device driver

8.1 Exercise : Read the keys entered on the keyboard

First, we will focus on the functioning of a keyboard controller typical of Intel architectures, the PS2 controller. We have not chosen to use in this subject the usb keyboard controller because the management of USB communications unnecessarily complicates the proposed exercise.

You can find a description of how the keyboard controller works on your machines at osdev.org. In particular, it can be seen that the keyboard is accessible via two registers associated with ports '0x60' and '0x64'. We learn that the first is called data port, and that it can be read or written, while the second is called status register when read, and command register when written.

In addition, the keyboard processor generates a level 1 interrupt when a key is pressed.

Question 1.1 : Query the keyboard controller.

Make a first program that simply displays the keyboard code returned by the keyboard controller when typing a key. To do this, simply make an infinite loop that reads the keyboard code produced on port 0x60 with `_inb()` and writes the number read on the screen using for example `puthex()` and `putc()`.

Question 1.3 : produce ascii codes.

Obviously, the codes produced by the keyboard controller are not ascii codes. Several codes come out for a single keystroke.

— Explain the different codes you read and

— Propose a function `char keyboard_map(unsigned char);` which returns the ascii code associated with a keyboard key when it makes sense, or zero otherwise.

Warning : with the `-curses` option, QEMU does not have direct access to your keyboard, it only receives the stream of characters transmitted by the terminal. Thus, if it receives the character `T` (capital letter T), it will produce a sequence of keyboard events that would have produced a `T`, not necessarily the one you actually performed. In particular, you will see the same events that you use **Caps Lock** or **Shift** to produce your `T`. This problem does not arise with the graphical version, because QEMU then has access to the real keyboard events.

8.2 Exercise : Keyboard Driver

The keyboard keystroke program you implemented in the previous exercice has the inconvenience of performing an "active wait" for keystrokes. On the one hand, this implies that the microprocessor spends all the time that the scheduler devotes to the input program scrutinizing the input of a key, which is not useful. On the other hand, this implies that if too many keys are typed on the keyboard, while another program is running, the keyboard controller buffer may fill, and keystrokes may be lost.

To avoid this, it is more practical to organize the architecture of your small system so that the reception of a key from the keyboard is managed by the system, and not directly by the programs.

Question 3.1 : Keyboard interruption

Develop a function associated with the keyboard interrupt (level 1 interrupt). This function reads the keyboard code, and writes, if it makes sense, the ascii code associated with the key in a queue.

Question 3.2 : Suspend contexts waiting for keyboard input

Write a function `char getc()`. This function returns a character read from the keyboard. Note that this function must be blocking. The context that calls it will be suspended (and therefore no longer elected by the scheduler) until a new key is pressed. To achieve this, propose a solution that uses the semaphores previously implemented. When a keyboard interrupt takes place, it can "unlock" a context that would wait or prevent the next call from being blocked, since a character is available.

Question 3.3 : Manage a Keyboard Input Queue

The solution proposed for the previous question only manages a context calling the function `char_getc()`. If multiple contexts call the getc() function what can go wrong ? Propose a solution to address this issue.