EURECOM
Sophia Antipolis

Exam
Operating Systems - OS

Ludovic Apvrille
ludovic.apvrille@telecom-paristech.fr

February, 6th, 2015

Authorized documents: Nothing! The grading takes into account the fact that you don't have any document with you.
A grade is provided for every question (beware: do organize your time, e.g., last question is a 4-point question). 1 additional point is given as a general appreciation, including written skills and readability.
Solutions are provided within that font/color style. You may naturally get the maximum grade with a different solution. Also, please contact me if you find any improvement to this solution.

1 Course understanding (6 points, ∼30 minutes)

a. Multi-programming enables more than one single process to apparently execute simultaneously. How can an Operating System achieve this on a mono-processor computer? For your answer, clearly state the minimal hardware support that is needed to achieve this. [2 points]

We assume that we have a mono-processor system. In that system, only one process at a time can execute. Yet, because the OS can switch from one process to another one in a fast way w.r.t. human reactivity, we have the feeling that several processes can be concurrently executed. The scheduler of the OS thus selects a process very frequently, e.g., every 10 ms. This can happen because, in preemptive systems, a timer is set by the operating system - as well as its corresponding IRQ handler - when the system boots, in kernel mode. Then, all processes are run in user mode so as to prevent them from modifying the timers, or interrupts.
Finally, the minimal hardware support in a preemptive system is a dual-mode processor (kernel/user modes), and a timer that can be modified only in kernel mode.

b. Virtual memory is always handled by OS of PCs, but its management is optional in many RTOS. First, recall what is the main purpose of virtual memory. Then, explain why it is not always useful in real-time and embedded systems. [2 points]
The main purpose of virtual memory is the following:

- *Facilitate the compilation stage of a program by generating addresses that can be used at the same time by several processes (i.e., virtual addresses)*
- *Protecting the memory allocations of the OS from being modified by a user process, and protecting those allocations between user processes, thus avoiding some malicious activities, and the fact that one process could make another process crash (process isolation).*

In embedded applications, the two above mentioned points are not necessary. First, since there is commonly only one pre-known application running, physical addresses can thus be generated at compilation stage. Second, protection between processes is not relevant since commonly only one process is executed in the system (the application). Last point to be noticed is that a complex memory management induces non deterministic behaviours.
Note: there are other advantages to MMU, e.g., making it possible to have a virtual address space that is larger than the physical memory, but this is not a capability of prime importance in my opinion.

c. You have experimented with RTAI that processes/threads/tasks could wake up in advance with regards to their expected wake up time. Explain why this can occur. [2 points]
There are plenty of operations performed by a RTOS between a timer expiration and the execution of the task that queried that timer, which induces a wake-up latency. Thus, this wake-up latency is commonly due to:

- *The execution of the timer function itself (IRQ handler)*
- *The execution of maintenance tasks in the OS*
- *The execution of more urgent tasks, i.e. tasks with a higher priority that the one that set up the timer*
- *The preemption of a lower priority task that could be running when the timer expires. Indeed, if such a task is running, and currently making a system call, that, the OS must first reach a preemption point before the lower priority task can be preempted.*

Because that latency cannot be well characterized, and because of the timer granularity, this is quite a common practice to set the timer to expire a few microseconds before the expiration date specified by the task. And so, in some circumstances, e.g., the OS is not loaded, no other task is currently executing, the task can start its execution a bit in advance.

2 Linux kernel modules (13 points, ~90 minutes)

If you want to add code to a Linux kernel, a common way to do is to add some source files to the kernel source tree, or to modify existing files, and recompile the kernel, as we have done to add a Linux system call during a lab. But you can also add code to the Linux kernel while it is running, using a Loadable Kernel Module (LKM). A RTAI task, with which you have played with during the labs, is a LKM.

a. Is it a good idead to use kernel modules to implement the following functions, or is it better to rely on user-level processes: Drivers? Protocol stacks? Scheduler? System calls? ssh server? Closely explain your answer. [1 points]

- *Drivers: generally implemented directly in the kernel, or as kernel modules. If the driver is very complex, some of its parts not related to privileged instructions or kernel-level data structures can be moved to user-level processes.*
- *Protocol stacks. Kernel-level, including modules, or user-level implementations can be used. Buffer management is commonly performed in kernel mode. Most common protocols, e.g., the TCP/IP stacks, is generally part of the kernel.*
- *Scheduler. Its part related to the preemption and start of processes on the processor must be implemented in kernel mode, sometimes as modules (see, e.g., the RTAI scheduler). Yet, the scheduling policy itself could be implemented in user-level processes, but its frequent use of kernel data makes a kernel-level implementation more adequate.*
- *System calls. They are implemented as kernel-level code since they need either to execute privileged instructions (e.g., I/O operations), or to manipulate kernel data structures.*
- *ssh server. A user-level process running as root is commonly the selected option since it can rely on user-level libraries and system calls to be implemented. No kernel data structures or privileged instructions are necessary.*

b. The crash of a kernel module can freeze the whole operating system. Explain why a user-level application should not be able to freeze the entire system, and why a LKM can easily do this. In particular, explain which hardware elements are involved in that difference between user and kernel-level applications, and how they are involved. Last, provide a code that leads the linux kernel to crash in the following *init* function of a module: [3 points]

```
#include <linux/init.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");
```

```
static int module_init(void) {
  ...
}
```

The memory of a kernel module is not isolated from the one of the Operating System (MMU). On the contrary, user-level processes (dual-mode of the processor) can not interfer with the memory of the OS. Thus, a bad memory manipulation, or a bad use of a privileged instruction, e.g., manipulation of IRQs can lead the system to freeze when executing a badly programmed kernel module.
In the init function of a module, we could simply disable the interrupts, and make an infinite loop:
disable_irq(...)
for(;;);

c. Kernel modules can also be used to intercept system calls. The basic idea is to replace in the list of pointers to system call functions (this list is an array) the default reference of the system call to intercept with a reference to the function provided by the kernel module. The following code provides a typical way to do so. Explain the purpose of the following code. Also, explain how to execute that code (this is similar to starting a task in RTAI). [3 points]

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/unistd.h>
#include <asm/arch/unistd.h>

MODULE_LICENSE("GPL");

extern void* sys_call_table[];
int (*original_fork)(struct pt_regs);

static int module_init(void) {
  printk( KERN_ALERT "[edu]   Module successfully loaded\n");
  printk( KERN_ALERT "[edu]   Intercepting fork() syscall... ");

  original_fork = sys_call_table[__NR_fork];
  sys_call_table[__NR_fork] = edu_fork;

  printk( KERN_ALERT "done/n");
  printk( KERN_ALERT "[edu]   Starting Loging system calls\n");
  return 0;
}
```

Apart from including several headers, and giving the license of this module, the code provides the function to be called (module_init) when the module is loaded into the kernel.
That function logs a few information, keeps track of the default fork system call by assigning a function pointer (original_fork) to the one stored in the system call table. Then, it modifies the function pointer saved in the system call table by a pointer to the new implementation (edu_fork). At last, it logs its final actions, and returns a non-error code (0). Also, as explained before, the module needs to be

loaded in the kernel to be executed, e.g.:
insmod edu_fork.ko

d. Implement the *edu_fork* system call that performs a *printk* each time the *fork* system call is made in the system. After the *printk*, *edu_fork* should call the default *fork* system call. [3 points]
edu_fork should first perform a printk before calling the original fork, and returning what the original fork returns.

```
int edu_fork(struct pt_regs regs) {\\
     printk("We are in edu\_fork");
     return (*original_fork)(regs);
```

Also, as shown in the code, we have to care with the arguments of the system call, i.e. the ones declared in the function pointer.

e. When the module is unloaded, the default *fork* system call should be called instead of *edu_fork*. Provide the implementation of *module_exit()* [1 point]
module_exit should put back in the system call table the original fork system call. The code is thus quite straightforward.

```
static void module_exit(void) {
  printk( KERN_ALERT "[edu]   putting back the original fork system call\n");

  sys_call_table[__NR_fork] = original_fork;

  printk( KERN_ALERT "[edu]   Stopping Loging system calls\n");
}
```

That function is called when removing the module with rmmod

f. Why isn't it a good idea to allow the interception of system calls with kernel modules? How can this be prevented in the kernel? [2 points]
If system calls can be intercepted by Linux Kernel Modules, then, each time a user performs a root logging exploit, he can intercept systems calls.
An obvious solution to circumvent this is to avoid exporting the system call table, so that it can not be used in kernel modules. Storing system calls' pointers in other kinds of data structure could help as well.