EURECOM
Sophia Antipolis

Exam
Operating Systems - OS

Ludovic Apvrille
ludovic.apvrille@telecom-paristech.fr

February, 6th, 2014

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.

1 Course understanding (6 points, ∼30 minutes)

a. Explain the difference between system calls and library functions. Why are these two facilities needed? [2 points]

b. All programming errors cannot be detected at compilation step. Mention two different programming errors that an OS can detect during a program execution that a compiler cannot detect. For both errors, closely explain the OS mechanisms that are used to detect those errors. [4 points]

2 Linux kernel code analysis (14 points, ∼90 minutes)

The code of the first real version of the Linux kernel[1] is quite readable. The purpose of this exercise is for you to explain parts of the code of this kernel. For a few lines of lines, in particular those referring to sub-functions, you will need to make assumptions on their behaviours.
We recall that the notation "z=a?x:y" means "if (a) then z=x else z=y".

[1]Kernel version 0.01, released by Linux Torvalds in 1991

a. **panic function**, of *panic.c*. Let's start with a basic yet important function provided by the kernel: *panic()*. The code is provided just below. Explain it briefly, and give in which context it is ought to be used. [1 point]

```
volatile void panic(const char * s)
{
        printk("Kernel panic: %s\n\r",s);
        for(;;);
}
```

b. **uname**, provided in *sys.c*. **Explain line by line** the code of this function, and explain its use in the system. [2 points]

```
1.  int sys_uname(struct utsname * name)
2.  {
3.    static struct utsname thisname = {
4.      "linux .0","nodename","release ","version ","machine "
5.    };
6.    int i;
7.
8.    if (!name) return -1;
9.    verify_area(name,sizeof *name);
10.  for(i=0;i<sizeof *name;i++)
11.    put_fs_byte(((char *) &thisname)[i],i+(char *) name);
12.  return (0);
13. }
```

c. **Task structure**

"task_struct" is the structure that stores information about the processes - or tasks - scheduled by the kernel. Its declaration is provided in *linux/include/sched.h*. Right below is provided a raw excerpt of this struct declaration. Your purpose is to explain various elements of the struct, that is, give an explanation on how the struct fields are probably used by the kernel. [2 points]

```
struct task_struct {
/* these are hardcoded - don't touch */
        long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
        long counter;
        long priority;
        long signal;
        fn_ptr sig_restorer;
        fn_ptr sig_fn[32];
/* various fields */
        int exit_code;
        unsigned long end_code,end_data,brk,start_stack;
        long pid,father,pgrp,session,leader;
        unsigned short uid,euid,suid;
        unsigned short gid,egid,sgid;
        long alarm;
        long utime,stime,cutime,cstime,start_time;
/* file system info */
        int tty;                /* -1 if no tty, so it must be signed */
        unsigned short umask;
        struct file * filp[NR_OPEN];
...
};
```

***d*. Main comment of schedule() function**. The code of the *schedule()* function is given in *kernel/sched.c*, and is provided just below. Let's first analyze the top comment of the function: What does Linus Torvalds mean by "IO-bound processes good response"? [1 point]

***e*. "First part" of the schedule() function**. What is the purpose of that part? Also, give the comment you would put before that part of code. [2 points]

***f*. Main part of the schedule() function**: "this is the scheduler proper:". What is the purpose of that part? What are the main elements of that code? You may explain line by line, but what I expect is rather the various steps of the scheduling algorithm. [2 points]

```
*
 *  'schedule()' is the scheduler function. This is GOOD CODE! There
 * probably won't be any reason to change this, as it should work well
 * in all circumstances (ie gives IO-bound processes good response etc).
 * The one thing you might take a look at is the signal-handler code here.
 *
 *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */
void schedule(void)
{
        int i,next,c;
        struct task_struct ** p;

/* First part */

        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                if (*p) {
                        if ((*p)->alarm && (*p)->alarm < jiffies) {
                                        (*p)->signal |= (1<<(SIGALRM-1));
                                        (*p)->alarm = 0;
                                }
                        if ((*p)->signal && (*p)->state==TASK_INTERRUPTIBLE)
                                (*p)->state=TASK_RUNNING;
                }

/* this is the scheduler proper: */

        while (1) {
                c = -1;
                next = 0;
                i = NR_TASKS;
                p = &task[NR_TASKS];
                while (--i) {
                        if (!*--p)
                                continue;
                        if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                                c = (*p)->counter, next = i;
                }
                if (c) break;
                for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                        if (*p)
                                (*p)->counter = ((*p)->counter >> 1) +
```

```
                                                (*p)->priority;
        }
        switch_to(next);
}
```

g. **block_read()**, provided in *block_dev.c*. Closely explain that code **line by line**. Also, explain what is returned by that function in various situations. And so, put what is necessary in the two lines commented with "//", and explain why you have put this. [4 points]

```
1.   int block_read(int dev, unsigned long * pos, char * buf, int count)
2.   {
3.           int block = *pos / BLOCK_SIZE;
4.           int offset = *pos % BLOCK_SIZE;
5.           int chars;
6.           int read = 0;
7.           struct buffer_head * bh;
8.           register char * p;
9.
10.          while (count>0) {
11.                  bh = bread(dev,block);
12.                  if (!bh)
13.                          return read?read:-EIO;
14.                  chars = (count<BLOCK_SIZE) ? count : BLOCK_SIZE;
15.                  p = offset + bh->b_data;
16.                  offset = 0;
17.                  block++;
18.                  *pos += chars;
19.                  //read +=
20.                  //count -=
21.                  while (chars-->0)
22.                          put_fs_byte(*(p++),buf++);
23.                  bh->b_dirt = 1;
24.                  brelse(bh);
25.          }
26.          return read;
27.  }
```