EURECOM
Sophia Antipolis

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

A system call serves as an interface between a user-level application and the Operating System. This is the only way for a user-level application to execute commands for which protected instructions must be executed, typically, I/O operations. On the contrary, a library function is a way to factorize software code between applications. Library functions are executed in user mode, and thus, cannot execute protected instructions.

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]

- *Memory management error. A compiler cannot usually predict whether data written at given memory adresses are part of the allocated memory of the*

corresponding process, or not. The Memory Mamagement Unit can usually detect such errors at execution time. When such an error occurs, the MMU raises an exception that the operating system catches: the faulty process is then aborted by the Operating System, and an error is returned to the user (e.g., "segmentation fault").

- *I/O error. The program saves data into a file, but such data cannot be written to the disk because the latter is full. The driver of hard disk detects that error, and the system call used for that operation returns an error value, typically -1.*

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 code, 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".

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(;;);
}
```

The panic function is used when no other action can be taken by the operating system, that is, an error occurred and the operating system does not know how to handle it. Thus, the panic function merely returns a message to the user (the one given as argument to the function), and ends by entering into an infinite loop.

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. }
```

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

The sys_uname function provides the string values of the different fields of the uname function. The code is as follows: (line by line)

- ***Line 1**. Declaration of function. The provided argument is to be filled by the function.*
- ***Line 3 to 5**. Filling a local structure with the different values of field for uname.*
- ***Line 8**. Verifying that the provided pointer to the structure to be filled is not null. Otherwise, return an error value (-1).*
- ***Line 9**. Verifying that the allocated memory at the provided pointer to the structure is large enough.*
- ***Line 10 to 11**. Copying string from the local structure to the referenced one, step by step. Note that each string must be 8-character long.*
- ***Line 12**. Returns a success value (0).*

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];
...
};
```

The usage of those fields is not always understandable from the definition of the struct. We thus define only those for which the usage is the most obvious.

- *"state" is used to store the current scheduling state to the process.*

- *"counter" might be used to count the number of instructions used by the process during the last quantum of time, while "priority" is rather a static priority attributed to the process.*
- *"signal" probably stores the pending signals, and "sig_fn" is an array of references to functions which index in the array correspond to a signal number.*
- *"exit_code" probably references the code returned to the father process at termination.*
- *"end_code", "end_data", ... references addresses of the memory used for the allocated memory areas of the process (code, pre-declared data, dynamic allocation,stack).*
- *"pid", ... "sgid" are used to store the user/groups/etc. rights on the process.*
- *"alarm" is likely to be a timer that can be put on the process.*
- **time" are used to store timing information on the process.*
- *"tty" refers to the terminal attached to the process.*
- *"umask" is the rights that are applied to newly created files.*
- *"file" is the list of opened files.*

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]

An IO-bound process is a process that regularly waits for IO operations to be completed. Such a process spends most of its time waiting, i.e., the time it spends doing computation is negligeable with regards to the time it spends waiting. IO-bound processes are favroured in Linux to offer a better level of interactivity.

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]

The first part of the scheduler checks for terminated operations in processes, and in particular it checks for timer expirations. I would put the following comments:

- *Right after "if (*p)":*

```
/* Check for timer expiration */
```

- *Right before the last test of the first part:*

```
/* Check for sleeping processes with pending signals */
```

***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]

*The second part of the schedule function is dedicated to the selection of the process to be executed. Basically, it searches among the executable process for the process with the highest counter (in the "while (–i)" loop). It saves the index of that process, and the highest counter value in the variable "c" (that is: its dynamic priority). Then, if a process with a non zero counter was found, the loop exits ("break") and the system switches to the selected process ("switch_to(next)"). Otherwise, it means that all processes have expired their quantum: and so, a new quantum begins: the start value is put in the counter of processes (based on their static priority ((*p)->priority)), and the process selection is done again.*

```
*
 *  '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)
```

*This function is probably used to read bytes in a block device whose reference is given as argument (dev). Data must be read from a given position in the device (pos), put in a destination buffer (buf). "count" elements must be read. An important part of the function is dedicated the management of blocks of the device (e.g., block = *pos / BLOCK_SIZE, etc.)*

```
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) {
```

Main reading loop, over the number of elements to be read

```
11.                   bh = bread(dev,block);
```

Getting a pointer to the data of the current block of the device

```
12.                   if (!bh)
13.                           return read?read:-EIO;
```

If no pointer was obtained, either return already read elements, or return an error if no data could be read at all

```
14.                   chars = (count<BLOCK_SIZE) ? count : BLOCK_SIZE;
```

The number of read elements is either count, or BLOCK_SIZE, depending on the remaining data to be read in the considered block

```
15.                   p = offset + bh->b_data;
16.                   offset = 0;
17.                   block++;
18.                   *pos += chars;
19.                   //read +=
```

read += chars // the number of elements read is equal to what could be read (cf. the definition of chars)

```
20.                     //count -=
```

count -= chars

```
21.                     while (chars-->0)
22.                             put_fs_byte(*(p++),buf++);
```

Putting elements of the block into the output buffer, byte by byte

```
23.                     bh->b_dirt = 1;
24.                     brelse(bh);
25.             }
26.             return read;
```

The number of bytes read is returned

```
27.     }
```