Exam
Operating Systems - OS

Ludovic Apvrille
`ludovic.apvrille@telecom-paristech.fr`

February, 2017

Authorized documents: Nothing! The grading takes into account the fact that you don't have any documents with you.
A grade is provided for each question (beware: be sure to organize your time, e.g. you should spend 30 minutes on each exercise). 1 additional point is given for general appreciation, including writing skills and readability.

1 Course knowledge (5 points, ∼30 minutes)

a. Programmers have decided to create a brand new OS offering a better support for mathematical primitives, i.e. it offers *syscalls* such as "cosine", "FFT", "matrix inversion". Do you think this is a good idea? Justify your answer. If not, what could be done to improve the performance of mathematical computations? [2 points]

b. What is the most common scheduling policy for interactive systems, i.e. the one we find in Linux / Windows? [1 point] Why is this scheduling policy adapted to interactive systems? [1 point]

c. The Process Control Block was presented during lectures. List information that is likely to be stored in a Thread Control Block, and explain why it should be stored in this TCB. [1 point]

2 Interrupts and real time scheduling (4 points, ∼30 minutes)

A particular real-time system has three interrupt handlers. The following table shows the maximum rate at which each interrupt occurs (rate), the time taken to execute each handler (service time), and the maximum allowable interval between the interrupt and completion of the handler (deadline). In your analysis, assume that A, B, and C interrupts can arrive at any time.

Task	Rate	Computation time	Deadline
A	$1/20ms$	$10ms$	$20ms$
B	$1/80ms$	$10ms$	$80ms$
C	$1/25ms$	$5ms$	$25ms$

a. What is the percentage idle time for this system? Explain your answer. [2 points]

b. We assume the system is non-preemptive. Give the worst-case waiting time of A if the priority order of interrupts is the following: C > B > A? [1 point]

c. We still assume that the system is non-preemptive. Prove that all deadlines cannot be satisfied if the priority order of interrupts is the following: C > B > A? [1 point]

3 Spawning processes (6 points, ~30 minutes)

a. In the following code, assuming that fork() never fails, how many times is "OS is great" printed? Explain how you have obtained this result. [1 point]

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main(int nbOfArgs, char **args) {
     int i=0;
     pid_t ret;

     for(i=0; i<2; i++){
       ret = fork();
       if (ret == -1) exit(-1);
     }
     printf("OS is great!\n");
     return 1;
}
```

b. How could we modify the previous program to ensure that it prints "OS is great" twice as many times? As before, each "OS is great" should be printed by a single process only once. [1 point]

c. We now assume that fork may fail. How many times is "OS is great" printed? Is the result the same if we remove the line "if (ret == -1) exit(-1);"? [1 point]

d. We now consider the following code. Give the line number at which a new process is created in the system. [1 point]

```
01. #include <stdlib.h>
02. #include <stdio.h>
03. #include <unistd.h>
04.
```

```
05. int main(int nbOfArgs, char **args) {
06.     pid_t ret;
07.
08.     ret = fork();
09.     if (ret == -1) exit(-1);
10.
11.     if (ret == 0) {
12.       if (execl("/bin/sh", "sh", "-c", "/bin/ls", NULL) <0) {
13.         exit(127); /* error */
14.       }
15.     }
16.     return 1;
17.}
```

e. Explain the difference between "fork()" and "vfork()" [1 point]

f. If we replace "fork" by "vfork", what will change when executing the code? Now, what is the line number at which a new process is created? [1 point]

4 Linux kernel (6 points, ∼30 minutes)

We assume in the following questions that the latest stable linux kernel – kernel 4.9.6 – is located in "/tmp".

a. What do you learn from the output of the following command? [1 point]

```
$ cd /tmp/linux-4.9.6
$ du -s -m *|sort -nr
406     drivers
140     arch
38      fs
36      include
36      Documentation
33      sound
28      net
15      tools
8       kernel
7       firmware
4       scripts
4       mm
4       lib
4       crypto
3       security
...
```

b. In the "kernel" subdirectory, there is a file called "softirq.c"[1]. This file contains the following function. Explain its code. Give a typical situation where this function is used. [1.5 points]

```
static void wakeup_softirqd(void)
{
        /* Interrupts are disabled: no need to stop preemption */
        struct task_struct *tsk = __this_cpu_read(ksoftirqd);
```

[1]Originally written by Linus Torvalds. Rewritten by ANK

```
        if (tsk && tsk->state != TASK_RUNNING)
                wake_up_process(tsk);
}
```

c. In the same "kernel" subdirectory, there is another file called "signal.c"[2]. This file contains the following function. Explain its code. In particular, explain why the code refers to the word "handler". [1.5 points]

```
void ignore_signals(struct task_struct *t)
{
        int i;

        for (i = 0; i < _NSIG; ++i)
                t->sighand->action[i].sa.sa_handler = SIG_IGN;

        flush_signals(t);
}
```

d. The following code is also taken from "signal.c". Explain in which context it could be used. Also, explain what are the possible cases (e.g., on signals), and how they are handled. [2 points]

```
static void complete_signal(int sig, struct task_struct *p, int group)
{
        struct signal_struct *signal = p->signal;
        struct task_struct *t;

        /*
         * Now find a thread we can wake up to take the signal off the queue.
         *
         * If the main thread wants the signal, it gets first crack.
         * Probably the least surprising to the average bear.
         */
        if (wants_signal(sig, p))
                t = p;
        else if (!group || thread_group_empty(p))
                /*
                 * There is just one thread and it does not need to be woken.
                 * It will dequeue unblocked signals before it runs again.
                 */
                return;
        else {
                /*
                 * Otherwise try to find a suitable thread.
                 */
                t = signal->curr_target;
                while (!wants_signal(sig, t)) {
                        t = next_thread(t);
                        if (t == signal->curr_target)
                                /*
                                 * No thread needs to be woken.
                                 * Any eligible threads will see
                                 * the signal in the queue soon.
                                 */
                                return;
                }
```

[2] signal.c has been written originally by Linus Torvalds, and then improved by Richard Henderson and Jim Houston

```c
                signal->curr_target = t;
        }

        /*
         * Found a killable thread.  If the signal will be fatal,
         * then start taking the whole group down immediately.
         */
        if (sig_fatal(p, sig) &&
            !(signal->flags & (SIGNAL_UNKILLABLE | SIGNAL_GROUP_EXIT)) &&
            !sigismember(&t->real_blocked, sig) &&
            (sig == SIGKILL || !t->ptrace)) {
                /*
                 * This signal will be fatal to the whole group.
                 */
                if (!sig_kernel_coredump(sig)) {
                        /*
                         * Start a group exit and wake everybody up.
                         * This way we don't have other threads
                         * running and doing things after a slower
                         * thread has the fatal signal pending.
                         */
                        signal->flags = SIGNAL_GROUP_EXIT;
                        signal->group_exit_code = sig;
                        signal->group_stop_count = 0;
                        t = p;
                        do {
                                task_clear_jobctl_pending(t, JOBCTL_PENDING_MASK);
                                sigaddset(&t->pending.signal, SIGKILL);
                                signal_wake_up(t, 1);
                        } while_each_thread(p, t);
                        return;
                }
        }

        /*
         * The signal is already in the shared-pending queue.
         * Tell the chosen thread to wake up and dequeue it.
         */
        signal_wake_up(t, sig == SIGKILL);
        return;
}
```