EURECOM
Sophia Antipolis

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]
This is not a good idea because calling a syscall is less efficient than calling a library function, and more "risky" i.e. if a syscall has a bug, it could crash the whole OS. A better solution would be to optimize the mathematical library with e.g. assembly language programming or multithreading.
Note: using syscalls might be interesting if there is some mathematical support by a device (e.g., a GPU)

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]
The usual scheduling policy for interactive system is a preemptive round-robin with dynamic priorities. The dynamic management of priorities penalizes computation tasks i.e. tasks using all their time quantum. Since computation tasks are penalized, interactive tasks – i.e. the task using only a sub-part of their time quantum – have

a high priority and so are more likely to be selected quickly when they are un-blocked by an interaction e.g. a keyboard input.

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]
A TCB is likely to contain information like e.g.

- *Reference to the thread stack*
- *Reference to execution information (scheduling state, execution state)*
- *Reference to the process to which the thread belongs*

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]
If you consider a slot of $400ms$ – 400 ms is the least common multiple of the 3 tasks' periods –, A executes for $20 * 10ms$, B executes for $5 * 10ms$ and C for $16 * 5ms$. The resulting load in $400ms$ is $load = (200 + 50 + 80)/400 = 33/40$. So, the idle time is $1 - load = 1 - 33/40 = 7/40 = 17.5\%$

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]
Since the execution rate of B and C is lower than the rate of A, in the worst case, A can be delayed by one execution of C and one execution of B. So, A waits at most for the computation of C ($5ms$) and for the computation of B ($10ms$), so for $15ms$

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]
In the latter case, with all tasks released at the same time, A waits for $15ms$, so A terminates $25ms$ after its release time, so $5ms$ after its deadline. Hence, not all deadlines are satisfied.

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;
}
```

In the first loop iteration, the main process creates a child – let's call it child1 –. In the second iteration, the main process creates another child – child2 – and child1 creates a child – child11 –. Overall, we have four processes, each of them printing "OS is great" → "OS is great" is printed 4 times.

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]
To double the number of processes, we need each running process to create a new child process. We thus need all running processes to execute fork() another time. We can simply make another loop iteration: $for(i = 0; i < 3; i++)$

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]

- *In the first case (i.e. with "exit"):*
  - *If the first fork fails, no printf is executed*
  - *If the first fork succeeds, we have the main process and child1 executing*
    - *If the fork of main process and child1 fail, no printf is executed*
    - *If the fork of main process succeeds, but not the one of child1, main process and child2 will execute the printf*
    - *If the fork of child1 succeeds, but not the one of the main process, child1 and child11 will execute the printf*
    - *The two forks succeed, then we have printf executed four times*

  So, in that first case, we have 0, 2 or 4 printf.
- *in the second case, i.e., no exit.*

- *If the first fork fails, the main process will execute printf*
- *if the first fork succeeds, we have the main process and child1.*
  * *If the fork of main process and child1 fail, two printf are executed*
  * *if the fork of main process succeeds, but not the one of child1, main process, child2 and child1 will execute printf*
  * *if the fork of child1 succeeds, but not the one of the main process, main process, child1 and child11 will executeprintf*
  * *The two forks succeed, then we have printf executed four times*

So, in that second case, we have 1, 2, 3 or 4 print

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.}
```

It is "fork" creating a new process, so, the answer is "line 8".

e. Explain the difference between "fork()" and "vfork()" [1 point]
"fork" creates a new process which address space content is similar to the one of the father, apart from a few elements (e.g., the return value of fork). With vfork, the father is suspended until the child process makes a call to an exec() function, or exits. So, in the case of vfork, it is a common practice not to duplicate the address space of the father.

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]]
"vfork", just like fork, creates a new process, but the father is suspended until the child makes the call to execl (line 12). Yet, the new process is still created at "line 8".

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
...
```

The command computes the size of each subdirectory of the kernel sources ("du"), and sorts them in descending order by their size ("sort"). We observe that a large part of the kernel code is dedicated to drivers, and to support different hardware architectures.

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);

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

This function is likely to be used to wake up processes that have received a software interrupt. At first, the task – i.e., the thread or the process – targeted by this signal is selected. Then, if this task exists and is not running, it is awoken i.e. it becomes runnable.

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)
```

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

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

```
                t->sighand->action[i].sa.sa_handler = SIG_IGN;

        flush_signals(t);
}
```

This function is likely to be used to remove all the signal handlers of a given process/thread. A signal handler for signal x is a callback function called when the x signal is received. To remove all signal handlers, the function sets all signal handlers to "SIG_IGN" (IGN stands for "IGNore"). Then, all pending signals are removed using the "flush_signals" function so as to ignore these signals as well.

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;
                }
                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;
}
```

This function is used to determine the handling of a signal sent to a given thread or process. At first, the function determines if the destination thread can handle the signal. If not, another thread is searched for in the same group as the destination thread (i.e., among the threads of the same process). If no target can be found, then the function returns. Once the target is selected, the system evaluates if the considered signal indicates process termination. If so, all members of the group are terminated through a wakeup with a SIGKILL signal. Otherwise, the selected thread is awoken.