Course Operating Systems

Final Examination – Fall 2005

February, 2005

Duration: 2h

Ludovic.Apvrille@telecom-paris.fr

Authorized documents are limited to course's slides and to the code you produced during lab sessions. Calculators are authorized but quite useless. Answers should be as concise as possible. Also, you are free to answer either in **English** or in **French,** but please do not mix them!

Note: There are several possible solutions, this is no more than a sample. The exam was long but grading took it into account.

I. Understanding of the course (4 points)

(a) Closely explain the drawbacks of using the micro-kernel approach when implementing an Operating System?

Microkernels are characterized by their modularity: functionalities are cut into small pieces inside the kernel or moved to user space as processes. Compared to other kinds of kernels (monolithic, layered), this results in a performance drawback. Indeed, communication between kernels' entities are implemented by message passing, which is less efficient than direct function calls. In addition, communicating to an entity running in user space results in context switching.

(b) What is the main interest in using buffers for achieving input/output on devices? Also, what is double buffering? For what kind of devices is it used? Justify your answer.

Buffers are used to achieve better performance (it has been seen in labs #1). Indeed, communicating entities may produce / consume data with at very different rates. Buffers make it possible to manage this gap.
Double buffering is useful when a data consumer wants to access data while the producer is still generating new data.

(c) Describe the fundamentals of process signals under UNIX. What were the issues of applying process signals to threads? How has it been solved?

Under UNIX, signals represent an asynchronous mean for processes to communicate. To send a signal, a process performs a system call to kill(), and gives as parameter the target process identifier and the signal number. The OS forwards the signal to the destination process using in-kernel tables. The signal can either be ignored by the target process or computed using a registered handler.
Because signal sending takes into account a destination process, this communication scheme cannot be applied to threads. To apply it to threads, one could for example implement a system call handling thread identifiers, or allow whatever threads of a process to deal with signals received by the process. Note that communication between the threads of the same process is much easier because they share the same address space.

(d) How are used / implemented interrupt handlers of real-time systems to enhance the efficiency of these systems?

In real-time systems, efficiency is a matter of giving the computing resource to the task with the highest priority: interrupts interrupting such tasks should be given the computing resource during a period of time as small as possible. For this reason, interrupt handlers are commonly cut into two pieces. The first one is the interrupt handler which unlocks a process (let's call it P). The second piece is P. Thus, all processes with a higher priority than P's are blocked for a shorter time. Note also that other techniques used at interrupt handler level make it possible to enhance efficiency: interrupt masking, interrupt priority, etc.

II. Operating systems' reliability (4 points)

The following article has been extracted from LinuxWorld.

Linux Kernel 2.6 vs Windows XP: 2.6 Has Far Fewer Bugs
"Well below the industry average for commercial enterprise software," Wired reports
December 15, 2004

Four years ago at the Stanford University Computer Science Research Center an analysis project began to compare the 5.7 million lines of code in the 2.6 Linux production kernel with the 40 million lines of code in Windows XP.
According to Wired, the resulting study - which appeared yesterday - shows that, while in commercial code the industry standard is that there would be expected to be 114,000 to 171,000 bugs in 5.7 million lines of code, in Linux 2.6 there were just 985 bugs.

"This is a benefit to the Linux development community," Andrew Morton, lead Linux kernel maintainer, told Wired, "and we appreciate Coverity's efforts [Coverity is a

software engineering startup that now employs the five researchers who conducted the study] to help us improve the security and stability of Linux."

Of the 985 bugs identified, 627 were in critical parts of the kernel, Wired noted, while "another 569 could cause a system crash, 100 were security holes, and 33 of the bugs could result in less-than-optimal system performance."

(a) Provide a very short summary of this paper (2 lines).

This short article presents a study demonstrating that Linux has fewer bugs than Windows XP, with regards to the number of lines of code.

(b) Comment this paper. You should clearly emphasize which issues are raised, how they are solved, and comment whether results are relevant or not.

The purpose of this study is to compare the reliability of Linux kernel 2.6 with Windows XP's. I would not comment the results but rather the way the study was led. As a matter of fact, in my opinion, the results are not relevant at all.
Indeed, the evaluation process takes into account only one way to evaluate the number of bugs (source code analysis) and totally ignores the observed reliability of both operating systems in operation. Also, the evaluation considers all lines of code as equivalent whereas only a few lines of code of windows out of the 40 millions are really part of the operating system.

(c) What would be your approach to you compare the number of bugs in two operating systems?

I would use the proposed approach plus a benchmarking approach. I would run both operating systems on hundreds of PCs, and compare them for various operations. I would take into account the availability of these two systems per class of applications (file server, web server, end-user tools, etc.). With such a study, we might obtain exactly the results describe in the article, but these results would be much more trustable.

III. Programming with POSIX (6 points)

The goal of this exercise is for you to propose two different implementations of periodic threads using **POSIX primitives** (and not primitives specific to RT-Linux). More precisely, you should propose a C program running a thread that periodically executes a *foo*() function. Your first implementation should not use timers whereas the second one should make use of them.

(a) Your two implementations face a fundamental issue raised by real-time systems. Explain this issue and explain how you intend to solve it in your implementations.

The purpose of real-time systems is to handle time constraints. Our two implementations face the problem of handling the periodicity of the thread. More precisely, our implementation must guarantee that the periodic thread wakes up at the right moment (not before, not after) and that its periodicity does not drift.

(b) Provide your first implementation (without timer) and explain it. (For example, you may provide the implementation of a periodic thread calling the function foo() periodically with a period of 1 ms).

Note: I have simplified the management of time to focus my code on waking-up management. I assume the function gethrtime() returns the time as a long integer. Also, I have used a pseudo code to get rid of complex function arguments. Finally, I also assume that the deadline is smaller than the period.

Thread code:

```
long periodNumber = 0;
long currentTime, wakeupTime, startTime;
long period = 1000000;

startTime = gethrtime();
while(1) {
        foo();
        /* Must sleep until the next period */
        /* Danger: we must deal with spurious wake up -> we might be waked up too early */
        periodNumber ++;
        wakeupTime = periodNumer*period + startTime;
        while(currentTime=gethrtime() < wakeupTime) {
                nanosleep(wakeupTime – currentTime);
        }
}
```

main:

create the thread with pthread_create.

To be even more precise (i.e to be sure not to wake up too late), it would be better to wake-up a bit in advance, and to perform a busy waiting for a short period of time (100useconds for example).

(c) Provide your second implementation (based on timers) and explain it.

Same assumptions are taken (see previous question). In this code, I use a busy waiting to illustrate it and to be sure to wake-up right on time. Also, I assume that, when the timer expires, it calls a function thread1.

/ Timer handler */*

```
Function thread1:
        mutex_lock(&mutex);
        pthread_cond_signal(&timerExpired);
        mutex_unlock(&mutex);

Thread code:
        long periodNumber = 0;
        long currentTime, timerTime, startTime;
        long period = 1000000;

        startTime = gethrtime();
        while(1) {
                /* Must set the timer until the next period */
                periodNumber ++;
                timerTime = periodNumer*period + startTime;
                currentTime=gethrtime();

                /* Timer set for a shorter time */
                timer_settime(timer, timerTime- currentTime – 100000);

                /* Wait for timer to expire */
                mutex_lock(&mutex);
                pthread_cond_wait(&timerExpired, &mutex);
                mutex_unlock(&mutex);

                /* Busy waiting until the right time of wake-up is reached */
                currentTime=gethrtime();
                while (currentTime < timerTime) {
                        currentTime=gethrtime();
                }


                /* Call to periodic function */
                foo();
        }

main:
        create mutex (pthread_mutex_init())
        create condition variable (pthread_cond_init())
        create the timer (timer_create()). We assume that the function thread1 is called
when the timer expires
        create the thread with pthread_create().
```

IV. Scheduling algorithms (6 points)

We consider the following task set composed of 6 tasks P1, P2, P3, P4, P5, P6.
Notations: Ci and Ti designate respectively the capacity and the period of task Pi.

P1: C1 = 10, T1 = 80, the last 3 units of execution time requires exclusive access to a resource (protected by a mutex semaphore S) shared with task P4. P1 is periodic and its relative deadline is equal to 80.

P2: C2 = 20, T2 = 90. P2 is periodic and its relative deadline is equal to 90.

P3: is also periodic but (important !!) is *interrupt-driven*. It means that all processing is done by an interrupt routine and cannot be preempted. C3 = 20, T3 = 150; relative deadline is equal to 150.

P4: C4 = 20, T4 = 200; P4 executes 10 units of time, then requires S for 5 units (in order to access exclusively to the resource shared with P1) then after releasing S, P4 executes another 5 time units. P4 is periodic and its relative deadline is 200.

P5: is a periodic task whose execution is split into 2 parts. The first part is *interrupt-driven (no preemption)* and has a duration of C5a = 5 units of time. The second part is a normal task processing of C5b = 15 units of time. T5 = 250. Relative deadline is 250.

P6: C6 = 30, T6 = 300. P6 is periodic and its relative deadline is equal to 300.

We assume that the priorities are allocated according to the Rate Monotonic method.

Question 1

We consider *only* tasks P1, P2, P4 in the task set (we do not put the other tasks in service).

(a) Determine the worst case scenario of execution for the Task P1 assuming no special algorithm for priority inversion handling.
(b) Determine the worst case scenario of execution for the Task P1 assuming that Priority Inheritance Protocol is used to manage the semaphore S.

For both cases, give the termination time of task P1.

From now, we assume the protocol PIP for managing semaphore S and *all tasks* P1 to P6 are present in the system.

Question 2

For each task P1, P2, P3, P4, P5, P6, give the value of their “blocking factor” B1, B2, B3, B4, B5, B6, if any.

Question 3

Determine if the task set is schedulable under Rate Monotonic. (Use any valid method to justify your answer)

Question 4

Give the worst termination time of task P6.