EURECOM
Sophia Antipolis

Exam
Operating Systems - OS - Fall2018

Ludovic Apvrille
ludovic.apvrille@telecom-paristech.fr

February, 2019

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 with regards to the grading policy). 1 additional point is given for general appreciation, including writing skills and readability.

1 Course knowledge (6 points, ∼40 minutes)

a. Give the example of a blocking system call, and define what "blocking" means. Would it be possible to implement a non blocking version of this syscall? Quickly sketch how you would implement this syscall or why this would not be possible. [3 points]

A blocking system call suspends the calling process until the required element (e.g. data) is available. For instance, "read(desc, buf, n)" in "section 2" is a blocking system call that suspends the calling process until the input buffer has been filled or until an error occurred.

If I were to implement a non blocking version of read, I would add a fourth argument to read() that would correspond to a callback function: this function would be called by the system when the syscall completes. Then, the calling process can continue executing other instructions while the syscall is being processed by the OS. This obvisouly implies that the calling process does not immediately need the data to be read.

b. Could you explain when "condition variables" are necessary? Do use an example to compare two different programs which have a similar behaviour, but where only one of them is able to use a condition variable. Said differently, explain how you

can replace condition variables with a similar code. [3 points]

A condition variable is a signal facility between threads. This facility supports two functions: waiting and notifying. Condition variables are used along with mutex objects. Whenever a mutex m has been locked, a thread may decide to "wait" on a condition variable c because a condition is not satisfied e.g. a buffer is empty. the call to wait on c releases m, thus allowing other threads to lock m. Whenever a given condition is solved (e.g., buffer is empty), a thread can make a notify on c if it has locked m before: this wakes up one or all threads that have made a call on c. to avoid using condition variables, we need the waiting threads to regularly awake to retest the waiting condition. We can simply use sleep() for that purpose:

With condition variable:

```
lock(m)
while (a!=0)
  wait(c, m)
a --;
unlock(m)
```

Without condition variable:

```
lock(m)
while (a!=0)
  sleep(10)
a --;
unlock(m)
```

Obvisouly, this polling technique is more resource consuming and is less reactive.

2 Real-Time Linux (13 points, ∼80 minutes)

a. Cite three differences — other than scheduling — between Operating Systems and Real-Time Operating Systems. [1.5 points]

- *RTOSs offer more deterministic communication objects e.g. rt-fifo*
- *RTOSs are commonly developed in a modular way e.g. to remove a network stack, etc.*
- *RTOSs commonly split Interrupt Service Routines into a smaller ISRs and a regular process. This features reduces blocking time for urgent application processes.*

b. The documentation of Xenomai[1] states:

To get your application threads to be really considered as real-time threads by Xenomai scheduler, you will have to get them to use the real-time scheduling policy (called SCHED_FIFO). Note, however, what the SCHED_FIFO scheduling policy

[1]Memo: Xenomai is a real time version of Linux

means: it means that the scheduler will run a thread with this scheduling policy as long as it is runnable, and no other thread of higher priority is runnable.

Describe in your own words - e.g. using a figure - how this scheduling policy works. Is it able to handle all (urgent) timing constraints such as close deadlines? If not, how could it be improved? [2 points]

According to the text, the Xenomai scheduling policy relies on a preemptive priority-based First Come First Serve policy.
Lets us consider 3 tasks : T1 arrives at 0, T2 arrives at 1 and T3 at 1, with T3 having a higher priority than T1 and T2, and T1 and T2 having the same priority. A possible execution is: T1 is executed until t=1, then T3 is executed until it completes, then T1 is given the CPU again until it completes, then T2 is executed until it completes.
Obvisouly, this policy does not consider any aspect related to timing constraints e.g. periods and deadlines. If deadlines are a critical aspect of a given application, then Earliest Deadline First seems to be a better approach than FIFO. Obviously, EDF could be combined with a priority-based approach to differentiate between urgent and less urgent tasks having the same deadline.

c. In Xenomai, there is a timer function *xntimer_get_overruns()*. Its documentation is the following one:

```
xntimer_get_overruns()

This service returns the count of pending overruns for the last tick
of a given timer, as measured by the difference between the expected
expiry date of the timer and the date now passed as argument.

Parameters
    timer The address of a valid timer descriptor.
    now current date (as xnclock_read_raw(xntimer_clock(timer)))

Returns
    the number of overruns of timer at date now
```

The documentation of POSIX for timer overruns is as follows:

Under POSIX.1b, timer expiration signals for a specific timer are not queued to the process. If multiple timers are due to expire at the same time, or a periodic timer generates an indeterminate number of signals with each timer request, a number of signals will be sent at essentially the same time. There may be instances where the requesting process can service the signals as fast as they occur, and there may be

other situations where there is an overrun of the signals.

The timer_getoverrun function helps track whether or not a signal was delivered to the calling process. DIGITAL UNIX P1003.1b timing functions keep a count of timer expiration signals for each timer created. The timer_getoverrun function returns the counter value for the specified timer ID. If a signal is sent, the overrun count is incremented, even if the signal was not delivered or if it was compressed with another signal. If the signal cannot be delivered to the calling process or if the signal is delayed for some reason, the overrun count contains the number of extra timer expirations that occurred during the delay. A signal may not be delivered if, for instance, the signal is blocked or the process was not scheduled. Use the timer_getoverrun function to track timer expiration and signal delivery as a means of determining the accuracy or reliability of your application.

If the signal is delivered, the overrun count is set to zero and remains at zero until another overrun occurs.

So, what is meant by "timer overruns"? You may use a figure to explain your points. Also, why is it important when developing a real-time application to take these overruns into account? [3 points]

"Timer overruns" is the number of time a timer has expired before being received (or acknowledged). This situation typically happens when a process can not execute because higher priority processes are running, or because the receiving process is already executing another routine.
Each timer expiration could for instance correspond to the periodic execution of a task, or to a data to be read on a sensor. In the case of a periodic task, it is common that the deadline of this task is equal to its period. Thus, if timer signals were missed, it may mean that a task has missed its deadline, which could be critical in hard real-time applications.In case of soft real-time applications, the quality of service can be degraded by missed deadlines.
In the case of a hard real time application, the latter must determine if this overrun is critical, and switch to a safe mode in case it was critical. In the case of a soft real-time task, timer overruns can be used to improve the quality of service, e.g. switch to a lower video quality in order to maintain a good frame rate.

d. Write a pseudo-C code implementing a strict periodic task using timers, e.g. a task waking up strictly every 5ms. In case you have understood overruns (see previous question), I suggest you enhance your code using *xntimer_get_overruns()* . [4 points + 2.5 points if you take into account overruns]

To be really sure to wake up on time, we must rely on a fine-grain timer, and also assume that there are no higher priority tasks. Also, in order to avoid waking up too late, a task can wake up a bit before and busy waits until the exact time. The drawback of this later solution is that it may impact overall performance. Last but

not least, we have to take into account time drift when setting the timer for the next period. Here is one possible solution (without handling timer overruns).

```
Timer timer1;

// Function to set a timer
startMyTimer(period, periodNb, startTime) {
  targetTime = period * periodNumber + startTime;
  long durationToWait = targetTime - currentTime()-1;
  durationToWait = Math.max(0, durationToWait);
  setTimer(timer1, durartionToWait); // Assumption: syscall or a lib. function
}

main() {
  int period = 30;
  int periodNb = 1;
  long startingTime = currentTime();  // Assumption: syscall or a lib. function
  setTimer();

  while(1)
    // Setting timer for the next period

    startMyTimer(period, periodNumber, startTime);

    // Task body
    ...

    // Waiting for next period
    waitForTimerExpiration();

    // Busy waiting for the correct time
    while(currentTime() <  (period * periodNb + startTime));
    periodNb++;
}
```

We now assume that we have a periodic timer. We monitor overruns and switch to a safe mode after 3 overruns.

```
Timer timer1;
int nbOfOverruns = 0;

main() {
  int period = 30
```

```
  setPeriodicTimer(timer1, period);
  while(1)
    // Waiting for next period
    waitForTimerExpiration();

    // Compute the nb of overruns
    int n = timerGetOverrun(timers1);
    nbOfOverruns += n;
    if (n>3) { safeMode() }

    // Task body
    ...
}
```