TP 2

Real-time systems

The purpose of this practical work is about the POSIX API that allows to create real time tasks, by implementing many common situations.

Write your report during the session (pdf only) and, for each coding question, make a different file named `tp2_#.c` (with `#` the question number).

1 Introduction

First, you have to create a function that simulate computation during a lapse of time.

Question 1. Write a function `void do_work( unsigned int duration)` that occupies the processor during `duration` milliseconds. You will need the UNIX command `time`. Quantify the inaccuracy of measurement.

Indications :

- Calibrate this function with your processor speed simply (Do not waste time to automate this). The UNIX command `time` can be usefull, but becarefull to the start time of the program !
- Use the same optimization options for all TP because the execution speed of your function depends on it. It's best to compile without optimization here. If you compile with some optimization options, it's possible that the compiler removes some instructions of your code (your loop in this case). In all cases, use `asm volatile ("nop")` inside your loop to be sure that the compiler considers it usefull.

2 Sporadic tasks

A sporadic task performs a *job* in response to a *event*. These events are often hardware interrupts, but here we will simulate this environment by using software interrupts (UNIX signals).

Question 2. Write a program that diplays a message when the SIGUSR1 signal is sent.

The necessary functions to implement this are `signal`, `pause`, and eventually `getpid`. Reminder, the UNIX command `kill` allows to send any UNIX signal.

Question 3. Use the function `do_work()` to perform a longer job in the handler signal. Then, sends a burst of interrupts to the process. What do you notice ? What is the condition on the events sent that would avoid this problem ?

Question 4. Propose and implement a mechanism to improve the processing in the case of a burst interrupts. Is that the problem is completely resolved ?

Question 5. Propose and implement a new mechanism to verify that the previous interrupt ha been processed when it receives the next.

3 Periodic tasks

3.1 Introduction

The previous tasks work on the arrival of a event (*event-triggered*). In this section, the focus is on tasks that creates jobs at a time (*time-triggered*).

A *periodic* task is some type of time-triggered task, for which a job is created at regular intervals called *task period*.

Question 6. Implement, in two ways, a periodic task with a period equal to 1s. The task executes a work during 500ms (use the function do_work(). One program using the function `sleep()` and an other one by using the function `alarm()`. What are the problems of these methods ?

These limitations have led POSIX to get extensions for managing timers.

Question 7. Write a program that creates a periodic task using the functions `timer_create` and `timer_settime`. Don't forget the flag `-lrt` for the compilation.

3.2 Loop programming

Given a set of periodic tasks, T_2, T_3, T_4, defined by the following parameters :

- T_2 executes a `t2()` function with an execution time of 0.333s and a 2s period;
- T_3 executes a `t3()` function with an execution time of 1s and a 3s period;
- T_4 executes a `t4()` function with an execution time of 2s and a 4s period.

Question 8. Write a program that creates these tasks in a loop programming approach (i.e. one large periodic task). See Course 1 (slide 31/32) and Course 3 (slide 17).

Notes :

- Cutting treatments of `t2()`, `t3()`, `t4()` in pieces of arbitrary duration is allowed, but try to minimize the number of divisions.
- In the case of a real program, it is difficult to cut a program (or functions) into pieces with a certain duration. This is what makes the major difficulty of applying the method.

Question 9. Assume that the processor is replaced by a processor two times faster (all the duration of treatments are reduced by 50%. Is your program still working properly ? If not, propose a new one that works independantly of the processor speed execution.

Question 10. Are we in the case of a deterministic execution ?

3.3 Preemptive programming

Question 11. Write a program that executes the three previous periodic tasks, preemptively (with simply fork()).

EDF scheduling

Question 12. Show that the three previous tasks run always properly with the EDF algorithm (Earliest Deadline First). It is difficult to implement schedulers in space user (linux), in particular EDF.

Fixed priority scheduling

Ordonnancement à priorité fixe The fixed-priority scheduling run as follows:

- At any time, executes the task with the highest priority
- if a new task is ready, with an higest priority than the current task, preempt the current task and execute the new task instead.

Question 13. Can we find a distribution of priorities between the tasks T_2, T_3, T_4, such as the system becomes schedulable ? And with a faster processor ?

Question 14. Write a program with these three tasks (with the priorities that you found)

The C functions for changing a priority of a UNIX process are `sched_setparam` et `setpriority`