Lab 2

Ada real-time, first programs and tests

Download first the joined file `lab_2_simple_skeleton.tgz`.

1 A concurrent program of two periodic independent tasks

Four settings are required for tasks in order to ensure deterministic real-time behavior:

- **Locking the memory**, so that page faults caused by virtual memory (swap) will not undermine the deterministic real-time behavior. This can be done by calling the system call `mlockall` that prevent memory of being paged to the swap area (consult the man page for more details);
- **Pre-faulting the stack**, so that a future stack fault will not undermine the deterministic real-time behavior. This can be done by calling the system call `memset` which pre-loads each block of memory of the stack into the cache, so that no pagefaults will occur when the stack is accessed (consult the man page for more details);

  ✍ **Note about the stack:** The stack is a special region of your computer's memory that stores temporary variables created by each function call. The stack is a LIFO (last in, first out) data structure, that is managed and optimized by the CPU quite closely. Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, all of the variables pushed onto the stack by that function, are freed (that is to say, they are deleted). Once a stack variable is freed, that region of memory becomes available for other stack variables. ■

- **Analyzing the execution time** of a task, this can be done by using the system call `clock_gettime` (consult the man page for more details);
- Setting a **real-time scheduling policy** and **base priorities** for tasks.

Two C functions `lock_memory` and `stack_prefault` that respectively locks the memory of being paged to the swap area and pre-faults the stack are provided in `lock_mem.c` and `pre_fault.c` joined in `lab_2_simple_skeleton.tgz`. The first step is to fill `clk_time.c` and `simple.adb` according to the commented instructions:

- The C function `job_with_cpu_time` (in `clk_time.c`) specifies a job of with an Exact Execution Time EET in nanoseconds given as parameter. EET is used here to represent the WCET of a task. The real value in nanoseconds can be printed and compared to that given as input in a `main` procedure for example;
- The Ada procedure `Simple` performs the following:
  - It imports the three C functions `lock_memory`, `stack_prefault` and `job_with_cpu_time` resp. in three Ada functions `Lock_Memory`, `Stack_Prefault` and `Job_With_CPU_Time`;
  - It instantiates two periodic (with 150 iterations) anonymous tasks `T_1` and `T_2` for which a priority can be assigned using the pragma `Priority`: the first task has an EET C_1 close to 20 ms and the second has an EET C_2 close to 40 ms (call `Job_With_CPU_Time` to launch the current job);
  - It sets a conditional structure to check if deadlines are missed (deadlines are assumed to be implicit where $D_i = T_i$ for $i \in \{1, 2\}$).

The compilation commands are the followings:

- `arm-linux-gnueabihf-gcc -c *.c` (cross), `gcc -c *.c` (native);
- `arm-linux-gnueabihf-gnatmake simple.adb -largs clk_time.o lock_mem.o pre_fault.o` (cross), `gnatmake simple.adb -largs clk_time.o lock_mem.o pre_fault.o` (native).

The execution commands are the followings:

- `sudo ./simple` for multi-core executions;
- `sudo taskset -c 2 ./simple` for mono-core executions under the core 3 for example.

Under `htop`, by running the compiled program `./simple` (requires `sudo`), note the PIDs of `T_1` and `T_2` and check using the `chrt` command (requires `sudo`) the policy used for scheduling them.

2 First real-time tests on `voluntary` and `preempt-rt` kernels

The tests of this part has to be compared on both the `voluntary` and `preemp-rt` kernels. Argumentation should be provided for each test.

2.1 `SCHED_FIFO` tests

Steps 4, 5, 6 and 7 are to be done in parallel with a `hackbench` stress (under a separate session) with one group of 40 file descriptors (40 sender tasks). Each task will pass 1000000 messages of 100 bytes. The tasks runs at the priority level 49 under `SCHED_FIFO` (use `chrt` as done in Lab 1). The stress command should be **executed always first**.

1. The dispatching policy `FIFO_Within_Priorities` should be specified in the file `gnat.adc`.
2. What is the minimal period for `T_1` and `T_2` under `SCHED_FIFO`? Set it (with implicit deadlines) in the code.
3. Set the priorities of `T_1` and `T_2` at `System.Priority'Last` and compile.
4. As $C_1 \approx 20$ and $C_2 \approx 40$, check the tasks behavior under <u>monocore</u> executions? What do you conclude?
5. Set $C_1 = 20$ and $C_2 = 40$, check the tasks behavior under <u>multi-core</u> executions? What do you conclude?
6. Do the same checking by setting the priorities of `T_1` and `T_2` at `System.Priority'First`? What do you conclude?
7. How to emulate the scheduling algorithm RM (*Rate Monotonic*) under `SCHED_FIFO`? Redo the steps 4 and 5. What do you conclude?

2.2 `SCHED_RR` tests (bonus)

Do the same steps of Section 2.1 by replacing in `gnat.adc` the dispatching policy `FIFO_Within_Priorities` by `Round_Robin_Within_Priorities`.

3 Real-time tests on a simple LED blink Ada program (mini project)

Write an Ada concurrent program `led.adb` that blinks a led periodically. Your code should comply with the following instructions:

1. Like `simple.adb`, the main procedure should interface with `lock_memory` and `stack_prefault`;
2. Unlike `simple.adb`, your program defines only one concurrent periodic task `Blink` (for which a priority can be specified) with the main procedure that satisfies the following requirements:
   - The EET and the implicit deadline of `Blink` are respectively equal to 39 and 40 milliseconds;
   - The current job of `Blink` switch on or off the LED according to its previous state for a duration equal to EET;
   - The blink should be implemented by adapting the function `job_with_cpu_time` and scheduled using the dispatching policy `FIFO_Within_Priorities`.

In order to manipulate the GPIO pins of your RPi, follow the steps below:

1. Download, decompress and install the joined C library `bcm2835-1.50.tar.gz` in your working directory:
   - For native compilations, the commands are in the order
     `./configure`, `make`, `sudo make check` and `sudo make install`;
   - For cross-compilation, the configure command should be changed as follows:
     `./configure --host=arm-linux-gnueabihf --prefix=/usr/arm-linux-gnueabihf`.
2. Download and decompress the content of `bcm2835_ada_interfaces.tgz` in your project directory and compile the content as follows:
   - Native compilation:
     `g++ -c -fdump-ada-spec -C /usr/local/include/bcm2835.h`
     `gcc -c -gnat05 *.ads`;
   - Cross-compilation:
     `arm-linux-gnueabihf-g++-6 -c -fdump-ada-spec -C /usr/arm-linux-gnueabihf/include/bcm2835.h`
     `arm-linux-gnueabihf-gcc -c -gnat05 *.ads`.
3. Add to `led.adb` and `clk_time.c` respectively the following Ada and C entries:

```ada
-- led.adb
with bcm2835_h;
with Interfaces.C; use Interfaces.C;
```

```c
// clk_time.c
#include <bcm2835.h>
```

In order to configure GPIOs as input or output in your C code, you have some usage case studies under `bcm2835-1.50/examples`. Under `blink_ada`, get a look to `blink.adb`, an example of LED blink implemented in Ada using the `bcm2835` library.

In order to compile `clk_time.c`, don't forget to add the switch `-l bcm2835` to your `gcc` command. To (cross/native)-compile `led.adb`, add `/usr/arm-linux-gnueabihf/lib/libbcm2835.a` to the switch `-largs` of `gnatmake` (cf. the indications provided at the end of Part 1).

The blink task should be tested at the highest and lowest priorities in the presence of a `hackbench` stress on both the `voluntary` and `preempt-rt` (as described in Part 2). In your report you should interpret explicitly observations on the LED blink regularity.

Next lab: Resource sharing in Ada ...