U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Mestrados Integrados em Engenharias
Física / Informática / Electrotécnica e de Computadores
SISTEMAS EMBARCADOS / SISTEMAS EMBUTIDOS E DE TEMPO REAL
Time limit: 2h00 **TEST 2019-04-03**

Answer the following questions in a concise and clear way:

(1 point) What do you understand by **Logic Control** and **Temporal Control**?
Answer: Logic control: the sequence of instructions (the code) of the application, determines the execution time; Temporal control: the sequence of executions (determines the arrival pattern – periodic, aperiodic, sporadic)

a) **(1 point)** What are the main differences between **EDF** (*Earliest Deadline First*) and **FP** (*Fixed Priority*) task scheduling under processor **overload**?
Answer: FP: overload affects the tasks with lower priority than the one causing the overload, tasks with higher priority are not affected; EDF: all tasks are affected

b) **(1 point)** It is possible to execute tasks as **hardware interrupt** service routines. Which are the **pros and cons** of such method?
Answer: Pros: low overhead, thus faster service, easiness of writing the code; Cons: scheduling and priority limited and fixed by hardware, lower robustness since any bugs in ISRs will crash the system

2. Consider a desk VoIP (Voice over IP) phone that communicates over a TCP/IP network. During a call this phone executes the following **tasks** (timings in μs; T = *mit*, $D_i=T_i$ for all i):

- τ_{00}: C_{00}=4, T_{00}=20, sporadic
  receives **packets arriving** from the network and saves in a **network_in buffer**
- τ_{01}: C_{01}=2, T_{01}=40, periodic
  samples **microphone** and saves data in the **audio_in buffer,** reads from **audio_out buffer** and writes to the **speaker**
- τ_1: C_1=3 500, T_1=10 000 periodic
  reads **audio_in buffer**, **compresses** data, **sends** 1 VoIP packet (each VoIP packet has the sound corresponding to the period T_1 that precedes the task activation)
- τ_2: C_2=3 000, T_2=10 000 periodic
  receives voice packets from **network_in buffer**, **decompresses** data, places data in the **audio_out buffer**
- τ_3: C_3=1 000, T_3=50 000 sporadic
  reads commands from **keyboard** and processes them (volume, make/finish call, …)
- τ_4: C_4=1 500, T_4=50 000 periodic
  updates the **display** (call duration, caller/callee, …)
- τ_{99}: C_{99}=50 000, no deadline aperiodic
  executes in the **background** to run other network protocols (NTP, HTTP, …)

These tasks are executed on a **preemptive** real-time operating system (RTOS) with deadline-driven scheduling (**EDF**) that also allows a background task.

a) **(1 point)** Draw a diagram with the model of this application, using circles for tasks and squares for data entities and devices.
Answer: Almost all drew it correctly but with a minor error. The Network device is the same for tasks 00 and 99. Moreover, task 3 reads the keyboards and must write in an internal state variable. Similarly, the display task 4 needs to read from an internal state variable.

b) **(1 point)** Draw the sequence of events occurring from the acquisition of sound in one phone to playing it in the other. Quantify the intervals for which you have information.
Answer: Sender side: for 10ms task 01 has to fill the audio_in buffer with audio samples.

Then task 1 reads this buffer, assembles the VoIP packet and sends it, taking between 3.5 and 10ms. Then, there is the network transmission time for which we have no information. In the receiver side: Task 00 receives the VoIP packet from the network in 20μs (can be neglected). Then task 2 reads the packet, disassembles it and places the audio samples in the audio_out buffer taking between 3 and 10ms. Finally, task 01 reads the audio samples and writes them to the speaker. End-to-end, approx. 30ms in the worst-case (plus the network delay). The actual time taken by tasks 1 and 2 could be computed, knowing the interference caused by task 01.

c) **(1 point)** Determine the schedulability of the real-time tasks.
Answer: U=4/20+2/40+3.5/10+3/10+1/50+1.5/50 = 0.95 < 1 → the real-time tasks are schedulable by EDF

d) **(1.5 points)** To improve its service, you wish to run task τ_{99} among the real-time tasks using a Constant Bandwidth Server. Which is the maximum processor bandwidth that can be assigned to the server? Suggest adequate configuration parameters.
Answer: The remaining available bandwidth without causing overload is 1-0.95 = 0.05. This is the maximum bandwidth the server should get (U_S=0.05). Then, to run task 99, the most adequate choice is to assign a C_S=C_{99}=50ms. Consequently, T_S=C_S/U_S=1s

e) **(1 point)** For higher throughput, the audio_in and audio_out buffers use a double buffer technique with switching pointers. This is the only critical region in using these buffers, thus **very** short. Which synchronization approach would be more adequate and which tasks would be affected by the corresponding blocking time?
Answer: For very short critical regions, the most adequate synchronization approach would be disabling the interrupts. All other approach would have an excessive overhead for this situation. Note that this is going to be used by task 01 running every 40μs!

f) **(1.5 points)** Given the wide difference in time scale between the inter-arrival intervals of tasks τ_{00} and τ_{01} and the remaining real-time tasks, you decide to implement these as interrupt service routines instead of ordinary software tasks. Discuss the consequences.
Answer: This is what makes sense. Running regular tasks in any multi-tasking kernel has some overhead that is not generally compatible with periods in the us range. The pros and cons are those already referred in question 1-c). Pros: lower overhead, thus faster event handling, easiness of programming; Cons: these tasks will get strictly higher priority than all other that run in EDF, Thus, the schedulability of the EDF task system must account for the time taken by these ISRs. Moreover, there will be a potential blocking in the system tick but this is irrelevant in this case because it is too small (few μs and the tick could be set to few ms).

3. The following code (see next page) in file **schedule.c** implements a small kernel using a pre-emptive priority-based scheduler. Note that this code is identical to the code used in the laboratory sessions, with the only difference being the parameter (**parm**) passed to the function executing the task (line numbers 05, 29, 46, 53).

   The code in **axis_ctrl.c** implements the algorithm to control a single axis of a 3d axis machining centre. It takes the network address of the axis to control, and uses two variables (axis_vel and axis_pos) to store the axis position and speed between two consecutive invocations of axis_ctrl().

   The **main.c** file launches **three tasks**, all executing the same function **axis_ctrl()**.

   a) **(1 point)** "The **first two lines** of file **main.c** are **incorrect**, as they reference a file containing code (.c file), instead of a header file (.h file)". Do you agree?

      If you agree with the statement, please mention which of the stages of the compilation process will generate the compilation error (pre-processing, compilation, assembly, linking), and what the error message will look like.

      If you disagree with the statement, briefly explain how these two lines will be processed during the compilation process (pre-processing, compilation, assembly, linking).

      The inclusion of c code inside other c code is legal and correct, even though it is not often used as in the general case it is not considered good practice.

      During the pre-processing phase the pre-processor will replace the pre-processor directives (e.g. #include "scheduler.c") by the contents of the referenced file (e.g. scheduler.c). The next phase, compilation, will therefore consider as input a single virtual file, equivalent to the concatenation of all .c files containing the code.

   b) **(1 point)** Identify the shortest sequence of **compilation** and **linking** commands that allows generating a single **executable file** from the above code and identify all **intermediary files** that may eventually be generated.

Compilation and linking command: gcc main.c -o main

Only the final executable file, "main", is generated.

Since the main.c file includes (#include "...") all other remaining c files, this file effectively contains all the code required to generate the executable, i.e. it does not require any further libraries. A single command to compile main.c is enough to generate the final executable file.

   c) **(2 points)** After some experimental runs, you were able to determine the **WCET** (worst case execution time) of each task and determined that the task set was schedulable. However, during run-time, the task with the **lowest priority** over-runs its WCET, and is re-scheduled for execution while its previous iteration is still running.

      Explain why the **kernel code** will **not allow** the task to interrupt itself, which would result in two simultaneous activations of the same task.

      The function `Sched_Dispatch(void)` is responsible for dispatching (i.e. calling) the function corresponding to the tasks that need to be activated. Nevertheless the Sched_Dispatch() function will only call the new task/function if it has a higher priority than the currently executing task since the for loop on line 24 will only go up to curr_taks-1 ('x<curr_task'). In case a task is re-activated while its previous execution is still running, then the priority of the re-activated task will always be equal to or lower than the currently running task (which, in the limit, may be the previous iteration of the same task, so the same priority), so the function will never be called a second time.

   d) **(1,5 points)** All three axis control tasks share the same executable code (i.e. the axis_ctrl() function). Since this is a pre-emptive kernel, it is possible to have **all three tasks active** at the same time. Explain how these three concurrent executions of the axis_ctrl() function can each store a value in the 'local_var' variable without overwriting the values stored by the other activations.

      Note that 'local_var' is declared inside axis_ctrl().

      This kernel uses a single stack for all running tasks. However, since each task activation (i.e. task starts running on the CPU) corresponds to a function call, each task/function

call will use a separate stack frame on the stack. Each task will therefore get its own copy of the local variable, stored in the respective stack frame for that task/function call. Since each task will effectively have its own instance of the variable, each task can store a different value on its instance of the variable without overwriting the other instances.

e) **(1,5 points)** The function used to implement a thread in the above kernel must have the following **prototype**:

- `void *foo(void *)`

Functions used to implement a **POSIX thread** use the exact same prototype

- `void *bar(void *)`

You decide to port the above axis control program to POSIX by **re-using** the same function axis_ctrl() to launch a POSIX thread.

This approach to port the code **will not work**. Explain why.

Although the function prototype for each task is the same for both kernels (POSIX and embedded kernel), these work with distinct semantics. The embedded kernel will call the function periodically, expecting the function call to finish execution in each period.

POSIX, however, expects a function implementing a thread to only terminate its execution when the thread is to be destroyed, and will therefore not call the thread (task) function periodically. It is up to the thread function to implement the infinite execution loop (e.g. 'while (1)') inside the task function, and to control the periodic activation of each loop (e.g., using 'clock_nanosleep()').

Unless the task functions are changed to take on this new responsibility, porting the code directly will not work.

f) **(1,5 points)** You decide to **add a mutex** (mutex_lock() and mutex_free()) implementation to the kernel, and need to choose between implementing the **priority inheritance** protocol, **priority ceiling** protocol, or **SRP** (stack resource protocol). Which would you choose? Why?

The only protocol that can work for this embedded kernel if the SRP.

As stated above, this embedded kernel uses a single stack to run all the currently active tasks. This is possible because the fixed priority execution protocol enforces a LIFO order in task execution, identical to the order in which the stack is handled.

When using PCP or PIP, the fixed priority scheduling protocol no longer enforces a LIFO order of task execution, as a higher priority task may become interrupted in order to allow a lower priority task to continue execution and therefore free up a currently locked shared resource. In these situations, during which we have priority inversion, the lower priority task may corrupt the stack frame of the higher priority (and currently interrupted) task if it decides to call another function and therefore increase its stack use.

The SRP protocol does however guarantee a LIFO order of execution of tasks, as any blocking is incurred at the start, before the task itself starts executing. In this case, once the task starts executing (and therefore reserves space for itself in the stack), it is guaranteed it will run to termination without interruption from lower priority tasks.

4. **(1,5 points)** In the context of **dependable systems**, briefly explain what is meant by **'reliability'** and **'availability'**.

Reliability: probability of the system to provide correct continuous service for a specific amount of time. Expressed as a probability function over time. From this function it is possible to determine the average/mean amount of time for which the service will be provided continuously, often expressed as the MTTF (Mean Time To Failure).

Availability: the probability of a system be available and to correctly provide the expected service when that service is required. May also be more simply expressed as the percentage of time the system is running (amount of time it is running / amount of time it is stopped).

file: schedule.c

```
01 typedef struct {
02   int period; // period in ticks
03   int delay;  // ticks to activate
04   void *(*func)(void *);// function ptr
05   void  *parm ;         // function parm
06   int exec;   // activation counter
07 } Sched_Task_t;
08 Sched_Task_t Tasks[20];

10 void Sched_Schedule(void){
11   for(int x=0; x<20; x++) {
12     if !Tasks[x].func    continue;
13     if  Tasks[x].delay
14       {Tasks[x].delay--;}
15     else {
16       Tasks[x].exec  = 1;
17       Tasks[x].delay = Tasks[x].period;
18     }
19 }}

21 int  cur_task = 20;
22 void Sched_Dispatch(void){
23  int prev_task = cur_task;
24  for(int x=0; x<cur_task; x++){
25   if Tasks[x].exec {
26     Tasks[x].exec = 0;
27     cur_task = x;
28     enable_interrupts();
29     Tasks[x].func(Tasks[x].parm);
30     disable_interrupts();
31     cur_task = prev_task;
32     /*Delete if one-shot */
33     if !Tasks[x].period
34       Tasks[x].func = 0;
35     return;
36 }}}

37 /* Int handler -> called every 10 ms */
38 void int_handler(void){
39   disable_interrupts();
40   Sched_Schedule();
41   Sched_Dispatch();
42   enable_interrupts();
43 }

45 int Sched_AddTask(
46     void *(*f)(void *), void *parm,
47     int d, int p, int pri){
48   if (Tasks[pri].func) return -1;
49   Tasks[pri].period= p;
50   Tasks[pri].delay = d;
51   Tasks[pri].exec  = 0;
52   Tasks[pri].func  = f;
53   Tasks[pri].parm  = parm;
54   return pri;
55 }

57 void Sched_Init(void) {  …  }
```

file: axis_ctrl.c

```
Double axis_vel = 0;
Double axis_pos = 0;


/* init axis with address *addr */
void axis_init(int addr) {
  // calculate axis_gain and axis_offset
  // and store in global variables
  …
}

void *axis_ctrl(void *addr) {
  int local_var;
  …
}
```

file: main.c

```
#include "scheduler.c"
#include "axis_ctrl.c"

int id1 = 1034;
int id2 = 3089;
int id3 = 7762;


int main(int argc, char **argv) {
Sched_Init();
// init axis 1, 2 and 3
axis_init(id2);
axis_init(id3);
axis_init(id4);
// launch axis_ctrl process 1, 2 and 3
Sched_AddTask(axis_ctrl, &id1, 1, 10, 1);
Sched_AddTask(axis_ctrl, &id2, 1, 10, 2);
Sched_AddTask(axis_ctrl, &id3, 1,  6, 3);

// do work (control ventilation, ...)
while(1) work();
return 0; // humour the compiler
}
```

FORMULAS

Menor majorante

$$U(n) = \Sigma^{n}_{i=1}(C_i/T_i) \leq n(2^{1/n}-1)$$

Majorante hiperbólico

$$\Pi^{n}_{i=1}(C_i/T_i+1) \leq 2$$

Response Time

$$Rwc_i\ (0) = C_i + B_i + \Sigma_{k \in hp(i)}\ C_k$$

$$Rwc_i\ (m+1) = C_i + B_i + \Sigma_{k \in hp(i)} \lceil Rwc_i(m)/T_k \rceil * C_k$$

Synchronous busy period

$$L(0) = \Sigma_i\ C \qquad L(m+1) = \Sigma_i \lceil L(m)/T_i \rceil * C_i$$

Load function

$$h(t) = \Sigma_{Di \leq t}\ (1+\lfloor (t - D_i)/T_i \rfloor) * C_i$$