Integrated Master in

Electrical and Computer Engineering / Informatics and Computer Engineering

EMBEDDED (AND REAL_TIME) SYSTEMS

Time limit: 1h30 **TEST 2020-05-06**

1.

An industrial robot arm has 4 axes and it is used to move parts from an input buffer to a machine tool. The robot arm is controlled by a single processor that runs a set of sporadic tasks on a real-time operating system (RTOS). Tasks τ_1 to τ_4 control the 4 axes independently. Task τ_5 dynamically adjusts the setpoints of the 4 axes to produce a desired arm trajectory. Finally, task τ_6 is triggered by an emergency push button and it replaces task τ_5 to provide setpoints to tasks τ_1 to τ_4 that force a controlled shutdown of the robot arm. The tasks timing properties (execution time, deadline and minimum inter-arrival time) are the following (unit *ms*):

$\tau_1 \rightarrow$	$C_1=2,\ D_1=T_1=10$	(feedback loop of axis 1)
$\tau_2 \rightarrow$	$C_2=2,\ D_2=T_2=10$	(feedback loop of axis 2)
$\tau_3 \rightarrow$	$C_3=2,\ D_3=T_3=10$	(feedback loop of axis 3)
$\tau_4 \rightarrow$	$C_4=2,\ D_4=T_4=10$	(feedback loop of axis 4)
$\tau_5 \rightarrow$	$C_5=10,\ D_5=50,\ T_5=200$	(arm trajectory control)
$\tau_6 \rightarrow$	$C_6=10,\ D_6=50,\ T_6=100$	(emergency push button)

a) **(2 points)** Consider the RTOS uses **fixed priorities** and that tasks priorities are assigned sequentially from τ_1 highest to τ_6 lowest. Is this an adequate **priority assignment** in terms of task set schedulability? Can you determine the task set schedulability with **utilization** or **density**-based tests? Justify.

b) **(2 points)** Determine the **worst-case response time** of the tasks.

c) **(2 points)** Once the emergency button is pressed, task τ_6 needs to be started. If τ_5 is idle, task τ_6 becomes ready immediately. If τ_5 is ready or running, τ_6 must wait for τ_5 to finish and then it becomes ready. Which is the maximum time that τ_6 may have to wait to become ready after the emergency button is pressed? Draw a **Gantt chart** of this situation, from the pressing of the button to when τ_6 becomes ready.

d) **(1 point)** For the sake of simplicity, you try running this system **without preemption**. Is this an adequate option? Justify.

e) **(1.5 points)** The controller uses an **8-bit processor**. The setpoints used to communicate between the tasks are **8-bit variables**. In this case, is it necessary to use a **synchronization mechanism** (e.g., interrupts disabling or mutexes) to ensure consistency of the variables? Justify.

f) **(1.5 points)** Consider you want to **redesign** your system using a **static cyclic-table scheduling** approach. Explain how you would do it.

FORMULAS

Liu & Layland Least Upper Bound	$U(n) = \Sigma^{n}_{i=1}(C_i/T_i) \leq n(2^{1/n}-1)$
Bini & Buttazzo Hyperbloic Bound	$\Pi^{n}_{i=1}(C_i/T_i+1) \leq 2$
Worst-Case 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	$L(0) = \Sigma_i\, C_i$ $\quad L(m+1) = \Sigma_i \lceil L(m)/T_i \rceil * C_i$
Load Function	$h(t) = \Sigma_{D_i \leq t}\, (1+\lfloor (t - D_i)/T_i \rfloor) * C_i$

2.

```
1  #include <time.h>    // required for clock_gettime(), clock_nanosleep()
2  #include <pthread.h> // required for pthread_create()
3
4  void f1(void); // declaration
5  void f2(void); // declaration
6  void f3(void); // declaration
7
8  void set_sched_parameter(int policy, int priority); // declaration
9  struct timespec addtimespec(struct timespec t1, struct timespec t2);
10
11 /***************************************************/
12
13 typedef struct {
14    struct timespec period;
15    void (*func_ptr)(void);
16    int    sched_policy;
17    int    sched_priority;
18 } ThreadParm;
19
20 void *PeriodicThread(void *arg) {
21   ThreadParm *tp = (ThreadParm *)arg;
22   set_sched_parameter(tp→sched_policy, tp->sched_priority);
23   struct timespec next_start;
24   clock_gettime(CLOCK_MONOTONIC, next_start);
25   while (1) {
26     next_start = addtimespec(next_start , tp->period);
27     clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
28                     next_start, NULL);
29     tp->func_ptr(); // call function => do work
30   }
31 }
32
33 /***************************************************/
34 #define T1  {0/*s*/, 50*1000000/*50 ms in ns*/}
35 #define T2  {0/*s*/, 50*1000000/*50 ms in ns*/}
36 #define T3  {0/*s*/, 12*1000000/*12 ms in ns*/}
37
38 void main(void) {
39   ThreadParm T1prm = {T1, f1, SCHED_RR, 100};
40   ThreadParm T2prm = {T2, f2, SCHED_RR, 101};
41   ThreadParm T3prm = {T3, f3, SCHED_RR, 102};
42
43   pthread_create(NULL, NULL, PeriodicThread, &T1prm);
44   pthread_create(NULL, NULL, PeriodicThread, &T2prm);
45   pthread_create(NULL, NULL, PeriodicThread, &T3prm);
46 }
```

Consider the above program stored in file **main.c**, written to run on a POSIX compatible Real-Time operating system. It launches 3 threads, each running the same **`PeriodicThread()`** function (lines 44-46). The **`PeriodicThread()`** function configures the scheduling policy and priority by calling **`set_sched_parameter()`** (line 23). It then sets up a periodic invocation of one of the three functions doing work, namely f1, f2, f3 (lines 25-31). All functions have a deadline equal to their period (50 ms for f1 and f2, 12 ms for f3). All the parameters (priority, policy, function, and period) used by **`PeriodicThread()`** to configure and run a task are stored in a **`ThreadParm`** struct.

Notes: The implementation of function **`set_sched_parameter()`** is not shown, but you can consider that it correctly calls the appropriate POSIX functions to configure the scheduling policy and priority. The function **`addtimespec()`** adds two times in **`struct timespec`** format.

a) **(2 points)** The f1(), f2() and f3() functions have each been defined in an independent file f1.c, f2.c and f3.c. Additionaly, the functions **`PeriodicThread()`** and **`set_sched_parameter()`** have both been defined in file util.c.

Explain the sequence of steps required to transform the mentioned files into an executable program. Describe the work done by the compiler in each of the steps. Which steps would need to be repeated if file f1.c were changed to fix a bug?

b) **(2 point)** The program uses some POSIX functions (e.g. `clock_gettime()`), which requires the program to be linked to the POSIX libraries using command line options: "gcc …. -lpthread -lrt"

By default, the compiler will use dynamically linked libraries, but you may also opt for the static version of those same libraries. Explain whether this choice between static and dynamic libraries will have any impact on the timely execution of your program, both on its startup, as well as on its capacity of executing the periodic functions within their deadlines.

c) **(2 points)** Consider that the functions f1() and f2() both need to atomically access the same shared resource. Thinking things through a little further, you realise that f1 and f2 both have the same period and deadline (50 ms). Instead of using a mutex to protect the access to the shared resource, you opt instead to configure both threads (f1 and f2) with the same priority (e.g. priority = 100) and keep using the `SCHED_RR` scheduling policy. Explain whether this is enough to guarantee that these threads will never pre-empt each other's execution, therefore making it safe to do without the use of the mutex.

d) **(2 points)** Functions f1, f2 and f3 are called once for each task activation, and always run to completion (i.e. the function reaches the end and returns) before the next periodic activation. Explain why any state that the task needs to maintain between task activation needs to be stored in either a global variable, a local static variable, or in the heap. Give examples of how you could implement each of these options.

3. **(2 points)** In the context of **dependable systems** briefly explain the methods of achieving software dependability, and how MISRA-C fits/maps onto these methods.