Real-Time Systems — EDA222/DIT161

Final exam, March 13, 2017 at 08:30 – 12:30 in the SB building

Examiner:
Professor Jan Jonsson, Department of Computer Science and Engineering

Responsible teacher:
Jan Jonsson, phone: 031–772 5220
Visits the exam at 09:30 and 11:30

Aids permitted during the exam:
J. Nordlander, *Programming with the TinyTimber kernel*
Chalmers-approved calculator

Content:
The written exam consists of 8 pages (including cover and hand-in sheets), containing 7 problems worth a total of 60 points.

Grading policy:
24–35 ⇒ grade 3
36–47 ⇒ grade 4
48–60 ⇒ grade 5

Results:
When the grading is completed overall result statistics, and a time and location for inspection, will be announced on the course home page. Individual results will be posted in PingPong under 'Objectives & Progress'.

Language:
Your solutions should be written in English.

IMPORTANT ISSUES

1. Use separate sheets for each answered problem, and mark each sheet with the problem number.
2. Justify all answers. Lack of justification can lead to loss of credit even if the answer might be correct.
3. Explain all calculations thoroughly. If justification and method is correct then simple calculation mistakes do not necessarily lead to loss of credit.
4. If some assumptions in a problem are missing or you consider that the made assumptions are unclear, then please state explicitly which assumptions you make in order to find a solution.
5. Write clearly! If we cannot read your solution, we will assume that it is wrong.
6. Hand-in sheets are available at the end of the exam script. Do not forget to submit them together with your other solution sheets!

Good Luck!

PROBLEM 1

State whether the following propositions are TRUE or FALSE. Each correct statement will give 0.5 points; each erroneous statement will give -0.5 points; an omitted statement gives 0 points. Although a motivation for a correct answer is not required, a convincing one gives another 0.5 points, while an erroneous/weak one gives another -0.5 points. **Quality guarantee**: The total result for this problem cannot be less than 0 points. (6 points)

a) In systems using rate-monotonic (RM) scheduling, it is impossible to find a feasible schedule for a set of five tasks whose accumulated utilization is 100%.

b) For a *sporadic* task there is a maximum time interval between two, subsequent, arrivals that is guaranteed to never be exceeded.

c) For an NP-complete problem to have *pseudo-polynomial* time complexity the largest number in the problem must be bounded by the input length (size) of the problem.

d) The term *critical instant* in schedulability analysis refers to the latest point in time by which a task must have started its execution in order to meet its deadline.

e) The term *false path* in execution-time analysis refers to a program subroutine that always returns the Boolean value False when running the program.

f) If a given task set is known to be schedulable, a *necessary* feasibility test will always report the answer "yes" when applied to that task set.

PROBLEM 2

In real-time systems that employ concurrent execution of multiple tasks with shared resources there is a potential risk that *deadlock* may occur.

a) State the four conditions for deadlock to occur in such systems. (4 points)

b) In such systems where tasks are assigned static priorities it is possible to avoid deadlock by using a run-time protocol that supports *ceiling priorities*. Describe the basic idea of a priority ceiling protocol (such as ICPP). (4 points)

PROBLEM 3

Most scheduling analysis techniques assume the worst-case execution time (WCET) to model the computational demand of a real-time task. One of the earliest methods for WCET analysis was presented by Shaw in the end of the 1980s.

Assume that the function `GenPulse` (see below) is used as part of a program in a real-time system.

The system has one output port, `Outport`, located at address 0x40020C14. The two least significant bits, b_1 and b_0, of the output port each constitute a control signal to a separate actuator.

The system has two input ports, `Inport1` (at address 0x40021010) and `Inport2` (at address 0x40021011). Each input port is connected to a separate sensor that delivers 8-bit values in the range $[-128, +127]$.

The program code of `GenPulse` compares the values read from the input ports and generates pulses on the output port bits b_1 and b_0. The width, that is the time between positive flank and negative flank, of each pulse is determined by the relation between the read values (as can be seen in the program code).

The system specification contains two strict timing constraints: (1) the width of the pulse on bit b_1 on the output port cannot be less than 400 μs, and (2) the width of the pulse on bit b_0 on the output port cannot exceed 800 μs.

```
#define Inport1 (*((volatile signed char *)(0x40021010)))
#define Inport2 (*((volatile signed char *)(0x40021011)))
#define Outport (*((volatile unsigned char *)(0x40020C14)))

void GenPulse() {
   signed char d1;
   signed char d2;
   unsigned char c;

   d1 = Inport1;
   d2 = Inport2;

   if (d1 == d2)
      c = 47;
   else
      c = 31;

   Outport = 0x01;    // start pulse on bit 0 (positive flank)

   while (c > 0)
      c = c - 1;

   if (d1 < d2)
      c = 77;
   else
      c = 61;

   Outport = 0x03;    // start pulse on bit 1 (positive flank)

   while (c > 0)
      c = c - 1;

   Outport = 0x00;   // end pulses on bits 0 and 1 (negative flanks)
}
```

Use Shaw's method to solve sub-problems a), b) and c). To that end, assume the following costs:

- Each declaration and assignment statement costs 1 μs to execute.
- Each evaluation of the logical condition in an `if`-statement costs 2 μs.
- Each evaluation of the logical condition in a `while`-statement costs 2 μs.
- Each subtract operation costs 4 μs.
- All other language constructs can be assumed to take 0 μs to execute.

a) Derive the minimum width (expressed in μs) of the pulse on bit b_1 on the output port, and determine whether the minimum-width constraint of 400 μs is met or not. (3 points)

b) Derive the maximum width (expressed in μs) of the pulse on bit b_0 on the output port, and determine whether the maximum-width constraint of 800 μs is met or not. (5 points)

c) What are the widths (expressed in μs) of the pulses on bits b_1 and b_0 on the output port if the value on `Inport1` is -121 and the value on `Inport2` is 93? (2 points)

PROBLEM 4

The TinyTimber kernel makes it possible to implement periodic activities in a C program. Consider a real-time system with three periodic tasks `T1`, `T2`, and `T3` that uses a common critical region that takes 250 μs to execute.

The three tasks all arrive at $t = 0$ and have a common period of 1000 μs, but their execution is precedence constrained in the following way:

- First, task `T1` should execute the critical region. The relative deadline for task `T1` is 650 μs.
- Then, task `T2` should execute the critical region. The relative deadline for task `T2` is 950 μs.
- Finally, task `T3` should execute the critical region. The relative deadline for task `T3` is 800 μs.

You should write a TinyTimber program where the task executions are implemented using two methods, `Non_Critical` and `Critical`. These methods are common for all tasks. Which task is currently executing each method is indicated by the **`whoami`** member of the task object, which has the value '1' for task `T1`, the value '2' for task `T2` and the value '3' for task `T3`. The critical region resides in method `Critical` and consists of a call to the function `Action250()`, the code of which is assumed to already exist.

a) In this sub-problem you should implement a version of the program where the critical region is called via a `SYNC` statement in method `Non_Critical`. Show that, by initially only starting task `T1`, it is possible to get the desired execution order and timing behavior of the tasks. On a separate sheet at the end of this exam paper you find a C-code template for this sub-problem. Add the missing TinyTimber statements directly on that sheet (in the space where the 'TODO' comments are located) and hand it in for grading together with the rest of your solutions. (4 points)

b) In this sub-problem you should implement a version of the program where access to the critical region is handled by means of a set of semaphores. Below you find the code of the include file `"semaphore.h"` for the semaphore library used. Show that, by starting all three tasks concurrently and using the semaphores in an appropriate way, it is possible to get the desired execution order and timing behavior of the tasks. On a separate sheet at the end of this exam paper you find a C-code template for this sub-problem. Add the missing information for the semaphore operations directly on that sheet (on the '____' lines in the program code where the 'TODO' comments are located) and hand it in for grading together with the rest of your solutions. (4 points)

```
struct call_block;
typedef struct call_block *Caller;

typedef struct call_block {
    Caller   next;
    Object   *obj;
    Method   meth;
} CallBlock;

#define initCallBlock() { 0, 0, 0 }

typedef struct {
    Object  super;
    int     value;
    Caller  queue;
} Semaphore;

void Wait(Semaphore*, int);
void Signal(Semaphore*, int);
```

PROBLEM 5

Consider a real-time system with three periodic tasks and a run-time system that employs earliest-deadline-first (EDF) scheduling. The table below shows C_i (WCET), D_i (deadline) and T_i (period) for each task τ_i. The first instance of each task arrives at time $t = 0$.

	C_i	D_i	T_i
τ_1	3	4	4
τ_2	2	18	20
τ_3	1	3	10

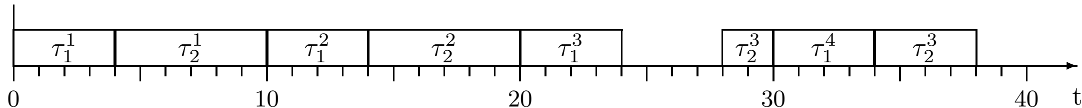
a) Explain why Liu & Layland's utilization-based test for EDF scheduling cannot be used for the given task set. (1 points)

b) Perform *processor-demand analysis* to determine the schedulability of the task set. (5 points)

c) Verify the outcome of the analysis in sub-problem b) by constructing a timing diagram for the execution of the three tasks that spans the *hyper period* of the task set. (2 points)

PROBLEM 6

Consider the timing diagram below describing the execution order of two periodic tasks when rate monotonic (RM) scheduling is used. Since RM is used, it applies for each task τ_i that its deadline D_i is equal to the period T_i. The first instance of each task arrives at time $t = 0$. The tasks are schedulable, that is, they meet their deadlines. In the timing diagram we use τ_i^k to denote instance k of the periodic task τ_i. All values are given in milliseconds. The cost for switching tasks is assumed to be negligible.



a) It is clear from the given schedule that the execution times of the tasks are $C_1 = 4$ and $C_2 = 6$, respectively. Decide the period T_i for each of the two tasks. (2 points)

b) Calculate, based on the given/derived values of C_i and T_i from sub-problem a), the utilization U for this task set. (1 point)

As was mentioned, the task set given above meets the deadlines. However, if the execution time of any of the tasks is increased ever so slightly a deadline will be missed (in this case, the deadline for the first instance of τ_2). A task set with such a property is said to *critically utilize* the processor.

c) Now add a third task to the task set as follows. Reduce the execution time of τ_2 by half (that is, let $C_2 = 3$) while keeping the period T_2 the same as before. Then, let the new task τ_3 arrive at time 0 and let its execution time $C_3 = 3$. Derive the period T_3 for τ_3 that both minimizes the utilization U of the task set and critically utilizes the processor. Also, state the resulting minimum utilization of the task set. Hint: only integer values ≤ 20 for the period T_3 need to be considered. (5 points)

d) Based on the utilizations derived in sub-problems b) and c), now imagine that you add, in a similar manner, a fourth (a fifth, a sixth etc) task to the task set. What is your prediction of the lowest utilization U for the updated task set when it critically utilizes the processor? (2 points)

PROBLEM 7

Consider the partitioned approach for scheduling tasks on a multiprocessor platform.

a) What is the Oh & Baker utilization guarantee bound for a system with m processors using the rate-monotonic first-fit (RMFF) partitioned scheduling algorithm? (1 point)

b) What is the best possible utilization guarantee bound for a system with m processors using a partitioned scheduling algorithm with static task priorities (such as RMFF)? (1 point)

c) The table below shows C_i (WCET) and T_i (period) for six periodic tasks to be scheduled on $m = 3$ processors using the RMFF algorithm. The relative deadline of each periodic task is equal to its period. The first instance of each task arrives at time $t = 0$. The values of C_i and T_i are positive integers. Determine the *maximum* value of C_2 by assigning all the six tasks on $m = 3$ processors based on the RMFF algorithm so that all the deadlines are met. Show your assignment of the tasks to the processors. (4 points)

	C_i	T_i
τ_1	10	50
τ_2	?	200
τ_3	8	20
τ_4	7	10
τ_5	5	30
τ_6	40	100

d) The table below shows C_i (WCET) and T_i (period) for six periodic tasks to be scheduled on $m = 3$ processors using the RMFF algorithm. The relative deadline of each periodic task is equal to its period. The first instance of each task arrives at time $t = 0$. The values of C_i and T_i are positive integers. Determine the *minimum* value of T_2 by assigning all the six tasks on $m = 3$ processors based on the RMFF algorithm so that all the deadlines are met. Show your assignment of the tasks to the processors. (4 points)

	C_i	T_i
τ_1	10	50
τ_2	2	?
τ_3	8	20
τ_4	7	10
τ_5	50	300
τ_6	20	100

Hand-in sheet with program code for Problem 4 a). *Anonymous code: ....*

```
#include "TinyTimber.h"

typedef struct {
    Object super;
    char whoami;
    int deadline;
} Task;

Task T1 = { initObject(), '1', 650 };
Task T2 = { initObject(), '2', 950 };
Task T3 = { initObject(), '3', 800 };

void Critical(Task*, int);

void Non_Critical(Task *self, int unused) {

    SYNC(self, Critical, 0);

    if (self->whoami == '1') {
                                                           // TODO: insert TinyTimber code


    }
    if (self->whoami == '2') {
                                                           // TODO: insert TinyTimber code


    }
    if (self->whoami == '3') {
                                                           // TODO: insert TinyTimber code


    }
}

void Critical(Task *self, int unused) {
    Action250(); // Do critical work for 250 microseconds
}

void kickoff(TaskObj *self, int u) {
    BEFORE(USEC(T1.deadline), &T1, Non_Critical, 0);
}

main() {
    return TINYTIMBER(&T1, kickoff, 0);
}
```

```
#include "TinyTimber.h"
#include "semaphore.h"

Semaphore Sem1 = { initObject(), ___, 0 }; // TODO: set initial semaphore value

Semaphore Sem2 = { initObject(), ___, 0 }; // TODO: set initial semaphore value

Semaphore Sem3 = { initObject(), ___, 0 }; // TODO: set initial semaphore value

typedef struct {
    Object super;
    char whoami;
    int deadline;
    CallBlock cb; // where call-back information is stored
} Task;

Task T1 = { initObject(), '1', 650, initCallBlock() };
Task T2 = { initObject(), '2', 950, initCallBlock() };
Task T3 = { initObject(), '3', 800, initCallBlock() };

void Critical(Task*, int);

void Non_Critical(Task *self, int unused) {
    self->cb.obj = (Object *) self; // provide call-back information
    self->cb.meth = (Method) Critical;
    if (self->whoami == '1')
        ASYNC(&________, Wait, (int) &self->cb ); // TODO: select semaphore to acquire
    if (self->whoami == '2')
        ASYNC(&________, Wait, (int) &self->cb ); // TODO: select semaphore to acquire
    if (self->whoami == '3')
        ASYNC(&________, Wait, (int) &self->cb ); // TODO: select semaphore to acquire
}

void Critical(Task *self, int unused) {
    Action250(); // Do critical work for 250 microseconds
    if (self->whoami == '1')
        SYNC(&________, Signal, 0); // TODO: select semaphore to release
    if (self->whoami == '2')
        SYNC(&________, Signal, 0); // TODO: select semaphore to release
    if (self->whoami == '3')
        SYNC(&________, Signal, 0); // TODO: select semaphore to release
    SEND(USEC(1000), USEC(self->deadline), self, Non_Critical, 0); // restart loop
}

void kickoff(TaskObj *self, int u) {
    BEFORE(USEC(T1.deadline), &T1, Non_Critical, 0);
    BEFORE(USEC(T2.deadline), &T2, Non_Critical, 0);
    BEFORE(USEC(T3.deadline), &T3, Non_Critical, 0);
}

main() {
    return TINYTIMBER(&T1, kickoff, 0);
}
```