Real-Time Systems — EDA223/DIT162

Re-exam, June 12, 2019 at 14:00 – 18:00 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 15:00 and 17:00

Aids permitted during the exam:
J. Nordlander, *Programming with the TinyTimber kernel*
Chalmers-approved calculator
Electronic dictionaries may not be used.

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

Grading policy:

24–35 points ⇒ grade 3	24–43 points ⇒ grade G (GU)
36–47 points ⇒ grade 4	
48–60 points ⇒ grade 5	44–60 points ⇒ grade VG (GU)

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 available in Ladok.

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. A hand-in sheet is available at the end of the exam script. Do not forget to submit it 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) A *necessary* schedulability condition for periodic tasks in a single-processor system is that the total utilization of the tasks is not larger than 1.

b) Hard real-time guarantee cannot be provided for systems with *sporadic tasks* since the inter-arrival time of consecutive instances of the tasks is not strictly periodic.

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 RMFF scheduling approach has a utilization guarantee bound that converges towards 33% as the number of processors become very large.

e) Disabling processor interrupts is a machine-level technique that is generally used to implement *mutual exclusion* on multiprocessor systems.

f) If a given task set is known to be not schedulable, a *necessary* feasibility test will always report the answer "no" when applied to that task set.

PROBLEM 2

A fundamental prerequisite for correct concurrent execution of multiple tasks with shared resources is that the run-time system can guarantee *mutual exclusion.*

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

b) One way to achieve mutual exclusion in the run-time system is to use *semaphores.* Describe in detail the data structure and operations that define a semaphore. (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 `main` (see below) is used as part of a real-time program and that the function, when called, is allowed to take at most 90 μs to execute.

- Each declaration and assignment statement costs 1 μs to execute.
- Each function call costs 2 μs plus WCET for the function in question.
- Each evaluation of the logical condition in an `if`- or `while`-statement costs 2 μs.
- Each add and subtract operation costs 3 μs.
- Each multiply operation costs 5 μs.
- Each return statement costs 2 μs.
- All other language constructs can be assumed to take 0 μs to execute.

```
int times(int a, int b) {
    return a * b;
}

int methA(int a, int b) {
    int p;
    int i;

    p = a;
    i = 1;

    if (b == 0)
        return 1;

    while (i < b) {
        p = times(p, a);
        i = i+1;
    }

    return p;
}

int methB(int a, int b) {
    if (b == 1)
        return a;
    else
        return times(a, methB(a, b-1));
}


int main() {
    char ans;
    int x;
    int y;

    x = 2;
    y = 3;

    if (methA(x, y) > methB(x, y)) {
        ans = 'T';
        x = x + y;
    }
    else
        ans = 'F';

    return 1;
}
```

a) Derive WCET for function `main` by using Shaw's method and determine whether or not the deadline of the function (90 μs) will be met. (8 points)

b) Identify two different false paths in the program given above. (2 points)

PROBLEM 4

The TinyTimber kernel makes it possible to implement periodic activities in a C program. Consider a real-time system with four independent periodic tasks: three hard-real-time tasks (`T1`, `T2`, `T3`), and one soft-real-time background task `BG`.

The three hard-real-time tasks all arrive at $t = 0$ and have a common period of 2400 μs, but their execution is precedence constrained in the following way:

- First, task `T1` should execute for 300 μs. The relative deadline for task `T1` is 1600μs.
- Then, task `T2` should execute for 800 μs. The relative deadline for task `T2` is 1200μs.
- Finally, task `T3` should execute for 500 μs. The relative deadline for task `T3` is 2100μs.

The soft-real-time background task `BG` arrives at time $t = 0$, has a period of 1800 μs, and at each invocation executes for 700 μs. The relative deadline for the background task is equal to its period.

a) Construct a TinyTimber program with four methods `T1()`, `T2()`, `T3()`, and `BG()` that have the same timing behavior as the corresponding tasks mentioned above. On a separate sheet at the end of this exam paper you find a C-code template. Add the missing program code directly on that sheet and hand it in for grading together with the rest of your solutions. The code for the functions `Action300()`, `Action800()`, `Action500()`, and `Load700()` is assumed to already exist. (5 points)

b) Assuming that the TinyTimber kernel will be used to schedule the four tasks, is it possible to guarantee that tasks `T1`, `T2`, `T3` will meet their deadlines? Motivate your answer. Any timing overhead relating to function calls or the TinyTimber runtime system can be ignored. (3 points)

PROBLEM 5

Consider a system with four periodic tasks and a run-time system employing preemptive deadline-monotonic (DM) scheduling. The table below shows C_i (WCET), D_i (deadline) and T_i (period) for the four tasks. The initial arrival time of each task is not known. All values are given in milliseconds.

	C_i	D_i	T_i
τ_1	8	15	20
τ_2	7	30	33
τ_3	8	35	41
τ_4	5	48	50

The four tasks are not independent, but share three exclusive resources R_a, R_b, and R_c. The run-time system employs the Immediate Ceiling Priority Protocol (ICPP) to resolve resource request conflicts. The tasks use the resources in the following way:

- Task τ_1 first requests R_b and then, while using R_b, requests R_a.
- Task τ_2 first requests R_b and then, while using R_b, requests R_c; then, while using R_c (and R_b), task τ_2 requests R_a.
- Task τ_3 first requests R_c and then, while using R_c, requests R_b; then, while using R_b (and R_c), task τ_3 requests R_a.
- Task τ_4 first requests R_c and then, while using R_c, requests R_b; then, after releasing the two resources, task τ_4 requests R_a.

The table below shows $H_{i,j}$, the maximum time (in milliseconds) that task τ_i locks resource R_j during its execution.

	$H_{i,a}$	$H_{i,b}$	$H_{i,c}$
τ_1	2	3	-
τ_2	2	2	2
τ_3	3	2	1
τ_4	2	1	1

Use a suitable analysis method to determine the schedulability of the tasks in the system. Note that nested blocking is used by all tasks. This could lead to accumulated critical region blocking times in the final blocking factor. (10 points)

PROBLEM 6

Consider a real-time system with three independent periodic tasks and a run-time system that employs preemptive single-processor scheduling using the earliest-deadline-first (EDF) priority-assignment approach. The table below shows C_i (WCET), D_i (relative deadline) and T_i (period) for the three tasks. For each task τ_i it applies that $C_i \leq D_i \leq T_i$. All tasks arrive at time $t = 0$.

	C_i	D_i	T_i
τ_1	4	5	10
τ_2	2	D_2	20
τ_3	4	25	40

Apply processor-demand analysis to determine the smallest positive integer value of D_2 for which all the tasks meet their deadlines. (8 points)

PROBLEM 7

Consider a real-time system with n independent periodic tasks that should be scheduled on a multiprocessor system, using preemptive static-priority scheduling. Global scheduling is one approach for scheduling tasks on such a system.

a) Explain the meaning of *Dhall's effect* in the context of global scheduling. (2 points)

b) Describe how RM-US global scheduling is able to circumvent Dhall's effect. (2 points)

The table below shows C_i (WCET) and T_i (period) for a task set with $n = 8$ periodic tasks. The relative deadline of each periodic task is equal to its period. All tasks arrive at time $t = 0$.

	C_i	T_i
τ_1	2	8
τ_2	2	5
τ_3	1	10
τ_4	3	120
τ_5	15	30
τ_6	60	600
τ_7	100	200
τ_8	2	16

c) Determine the minimum number of processors needed to guarantee that all tasks in the task set given above are schedulable using RM-US global scheduling. (3 points)

d) Determine a static-priority ordering of the tasks in the task set given above, assuming that RM-US global scheduling is used and that the number of available processors is equal to the minimum number of processors found in sub-problem c). (3 points)

```
#include "TinyTimber.h"

typedef struct {
    Object super;
} TaskObj;

TaskObj A = { initObject() };
TaskObj B = { initObject() };

void T1(TaskObj *, int);
void T2(TaskObj *, int);
void T3(TaskObj *, int);

void T1(TaskObj *self, int u) {


    Action300(); // Do work for 300 microseconds


}

void T2(TaskObj *self, int u) {


    Action800(); // Do work for 800 microseconds


}

void T3(TaskObj *self, int u) {


    Action500(); // Do work for 500 microseconds


}

void BG(TaskObj *self, int u) {


    Load700(); // Do background work for 700 microseconds


}

void kickoff(TaskObj *self, int u) {



}

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