Embedded Control Systems

Exam 2017-08-17

08:30 – 12.30: M Building

Course code: SSY190

Teachers: Knut Åkesson

The teacher will visit examination halls twice to answer questions. This will be done approximately one hour after the examination started and one hour before it ends.

The exam comprises 22 credits. At least 10 credits are needed for passing the written exam. The final grade is set by the rules published at the course syllabus.

Solutions and answers should be complete, written in English and be unambiguously and well motivated. In the case of ambiguously formulated exam questions, the suggested solution with possible assumptions must be motivated. The examiner retains the right to accept or decline the rationality of assumptions and motivations.

Exam results will be reported in Ladok. *The results* are open for review 2017-09-07 12:30-13:30 at the department.

No aids are allowed on the written exam except:

- Pen and a rubber
- Standard pocket calculator (no hand computer). Erased memory.
- Essential C, Nick Parlante. No comments are allowed in the report.
- Dictionary from/to your native language to/from English

1

a) Explain the difference between soft and hard real-time systems.

(1p)

b) Describe potential problems with stability analysis of hybrid systems.

(1p)

c) Describe the general concepts of a real-time operating system, i.e. scheduler, tasks, priorities, context switches.

(2p)

d) Describe which problems bumpless-transfer solves and how it can be implemented.

(2p)

2

Consider the program on the following page that has three tasks. One task is increasing a counter by one 10 million times (1e7), the second task is decreasing the same counter by one 10 million times. The print task prints the final value of the counter when both tasks have finished. The counter is initially (before the two tasks start to run) set to zero. Since the two tasks have the sume number of increments and decrements it would be natural to expect that the final value printed would be 0. However, running this program multiple times results in different number being printed.

a) Explain how the counter can end up having a different value after both the two counter tasks have finished their work, describe the name of this phenomena and describe by refering to how a real-time operating system operates what might cause this behavior.

(2p)

b) Suggest how to modify this concurrent program to make sure the program always compute the expected output, i.e., the final value the counter should be 0. Constraints: You can add but not remove any code/task. The two increase and decrease tasks should execute with the same priority and concurrently. Note, you only have to hand in the changes to this program. You might use the line-numbers to identify where you add code.

(2p)

Comments: The data type int is large enough (64-bits) for it to not overflow/underflow. The keyword volatile is used to disable the compiler from optimizing the code. In this simple case an optimizing compiler will notice that the for-loop for increasing/decreasing the value of the counter is not really necessary, instead it might emit code that assign he final value to the counter directly. However, by adding volatile we force the compiler to emit code that will do all iterations in the for-loop.

```c
xTaskHandle incTaskHandle;
xTaskHandle decTaskHandle;
xTaskHandle printTaskHandle;

SemaphoreHandle_t incTaskDone;
SemaphoreHandle_t decTaskDone;

volatile int counter = 0;

void increaseCounterTask(void *pvParameters)
{
    int i;
    for(i = 0; i < 1e7; i++)
    {
        counter = counter + 1;
    }
    xSemaphoreGive(incTaskDone);
    vTaskDelete(incTaskHandle);
}

void decreaseCounterTask(void *pvParameters)
{
    int i;
    for(i = 0; i < 1e7; i++)
    {
        counter = counter - 1;
    }
    xSemaphoreGive(decTaskDone);
    vTaskDelete(decTaskHandle);
}

void printFinalCounterTask(void *pvParameters)
{
    xSemaphoreTake(incTaskDone, portMAX_DELAY);
    xSemaphoreTake(decTaskDone, portMAX_DELAY);
    printf("Final value: %d\n", counter);
    vTaskDelete(printTaskHandle);
}

int main(void)
{
    incTaskDone = xSemaphoreCreateBinary();
    decTaskDone = xSemaphoreCreateBinary();
    xTaskCreate(increaseCounterTask, "IncTask", configMINIMAL_STACK_SIZE, NULL, 1, &incTaskHandle);
    xTaskCreate(decreaseCounterTask, "DecTask", configMINIMAL_STACK_SIZE, NULL, 1, &decTaskHandle);
    xTaskCreate(printFinalCounterTask, "PrintTask", configMINIMAL_STACK_SIZE, NULL, 1, &printTaskHandle);
    vTaskStartScheduler();
    for( ;; );
}
```

3

Consider the C-program below.

```
#include <stdio.h>

void swap(int *a, int *b)
{
    ...
    ...
    ...
}

int main()
{
    int x = 10;
    int y = 20;

    swap(&x, &y);

    printf("After Swapping\nx = %d\ny = %d\n", x, y);

    return 0;
}
```

Write C-code that implement the swap function, the swap function should make sure that values of x and y have swapped after returning from the swap function and discuss the following for variables x, y, a, and b.

- Explain where they are allocated in memory.
- Explain when it is safe to refer to them.
- Explain what is stored in the variables.

(2p)

4

Consider the set of tasks below.

Task name	Period	Deadline	Execution time
T_A	3	3	1
T_B	5	5	2
T_C	2	2	0.5

a) Assign a priority to each task according to the Rate Monotonic Scheduling (RMS) principle, and check whether the tasks will meet their deadlines using the approximate, utilization-based schedulability condition.

(1p)

b) Check whether the task set is schedulable using response time analysis and assuming RMS priority assignment.

(2p)

c) Draw the execution schedule for the task set assuming RMS and that all tasks are released simultaneously. Draw the schedule for the full hyper-period, i.e. until the schedule will repeat itself.

(2p)

d) Is the task set schedulable using the Earliest Deadline First (EDF) scheduling algorithm?

(1p)

5

Determine for each of the following pairs of linear temporal logic expressions whether they are equivalent or not. Justify your answer with an explanation.

a)	$\mathbf{G}(p \wedge q)$	$\neg\mathbf{F}(\neg p \vee \neg q)$
b)	$\mathbf{G}p \Rightarrow \mathbf{F}q$	$p\,\mathbf{U}(q \vee \neg p)$
c)	$\mathbf{FG}p \Rightarrow \mathbf{GF}q$	$\mathbf{G}(p\,\mathbf{U}\,(q \vee \neg p))$
d)	$\mathbf{GF}p \Rightarrow \mathbf{GF}q$	$\mathbf{G}(p \Rightarrow \mathbf{F}q)$

(4p)

Good Luck!

Formula sheet

Liu-Layland schedulability test

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

Response-time analysis

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

1/

a/ See lecture slides

b/ See lecture slides

c/ See lecture slides

d/ See lecture slides

2/

a) This is a race condition problem. The two threads read and write to the same variable (counter).

Note that

counter = counter + 1

involves three operations.

1. Moving the value of counter from the main memory to a register
2. Adding one to the value in the register.
3. Storing the data from the register to the main memory.

We get the described behaviour because we might get a context switch between reading the value from the main memory and before storing the updated value back in memory.

2b/ Add mutex implemented by a binary semaphore (initialised to 1)

```
xTaskHandle incTaskHandle;
xTaskHandle decTaskHandle;
xTaskHandle printTaskHandle;

SemaphoreHandle_t incTaskDone;
SemaphoreHandle_t decTaskDone;
SemaphoreHandle_t xMutex;
volatile int counter = 0;

void increaseCounterTask(void *pvParameters)
{
    int i;
    for(i = 0; i < 1e7; i++)      ←—— xSemaphoreTake(xMutex, portMAX_DELAY);
    {
        counter = counter + 1;
    }                             ←—— xSemaphoreLeave(xMutex, portMAX_DELAY);
    xSemaphoreGive(incTaskDone);
    vTaskDelete(incTaskHandle);
}

void decreaseCounterTask(void *pvParameters)
{
    int i;
    for(i = 0; i < 1e7; i++)
    {                             ←—— xSemaphoreTake( —— " —— );
        counter = counter - 1;
    }                             ←—— xSemaphorLeave( —— " —— );
    xSemaphoreGive(decTaskDone);
    vTaskDelete(decTaskHandle);
}

void printFinalCounterTask(void *pvParameters)
{
    xSemaphoreTake(incTaskDone, portMAX_DELAY);
    xSemaphoreTake(decTaskDone, portMAX_DELAY);
    printf("Final value: %d\n", counter);
    vTaskDelete(printTaskHandle);
}

int main(void)
{                                        xMutex = xSemaphoreCreateMutex();
    incTaskDone = xSemaphoreCreateBinary();
    decTaskDone = xSemaphoreCreateBinary();  ←——
    xTaskCreate(increaseCounterTask, "IncTask", configMINIMAL_STACK_SIZE, NULL, 1, &incTaskHandle);
    xTaskCreate(decreaseCounterTask, "DecTask", configMINIMAL_STACK_SIZE, NULL, 1, &decTaskHandle);
    xTaskCreate(printFinalCounterTask, "PrintTask", configMINIMAL_STACK_SIZE, NULL, 1, &printTaskHandle);
    vTaskStartScheduler();
    for( ;; );
}
```

3/

```
Viod  swap (int *a,  int *b)
{
   int temp = *a;
   *a = *b;
   *b = temp;
 }
```

x, y are allocated on the stack, stores integers.

a, b are also allocated on the stack, stores addresses to integers.

temp is allocated on the stack.

All variables that are stored on the stack are only safe to acess while the variable is in scope. After that other data may be written at the same memory location.

4a/ RMS - shortest period will have the highest prio.

Task C highest prio
Task A medium prio
Task B lowest prio

Calculate the utilization

$$\sum_{i=1}^{3} \frac{C_i}{T_i} = \frac{1}{3} + \frac{2}{5} + \frac{0.5}{2} = 0.9833 > 3(2^{1/3}-1) \approx 0.7798$$

Thus, the utilization based criteria cannot guarantee schedulability.

b/ Calculate the response times for all task.

Check whether $R_i \leq D_i \ \forall i$

Find fix-point for

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Start with highest prio task. (C)

$R_C = 0,5.$

Then solve for medium prio task. (A)

$$R_A^1 = 1 + \left\lceil \frac{1}{2} \right\rceil \cdot 0,5 = 1,5$$

$$R_A^2 = 1 + \left\lceil \frac{1,5}{2} \right\rceil \cdot 0,5 = 1,5$$

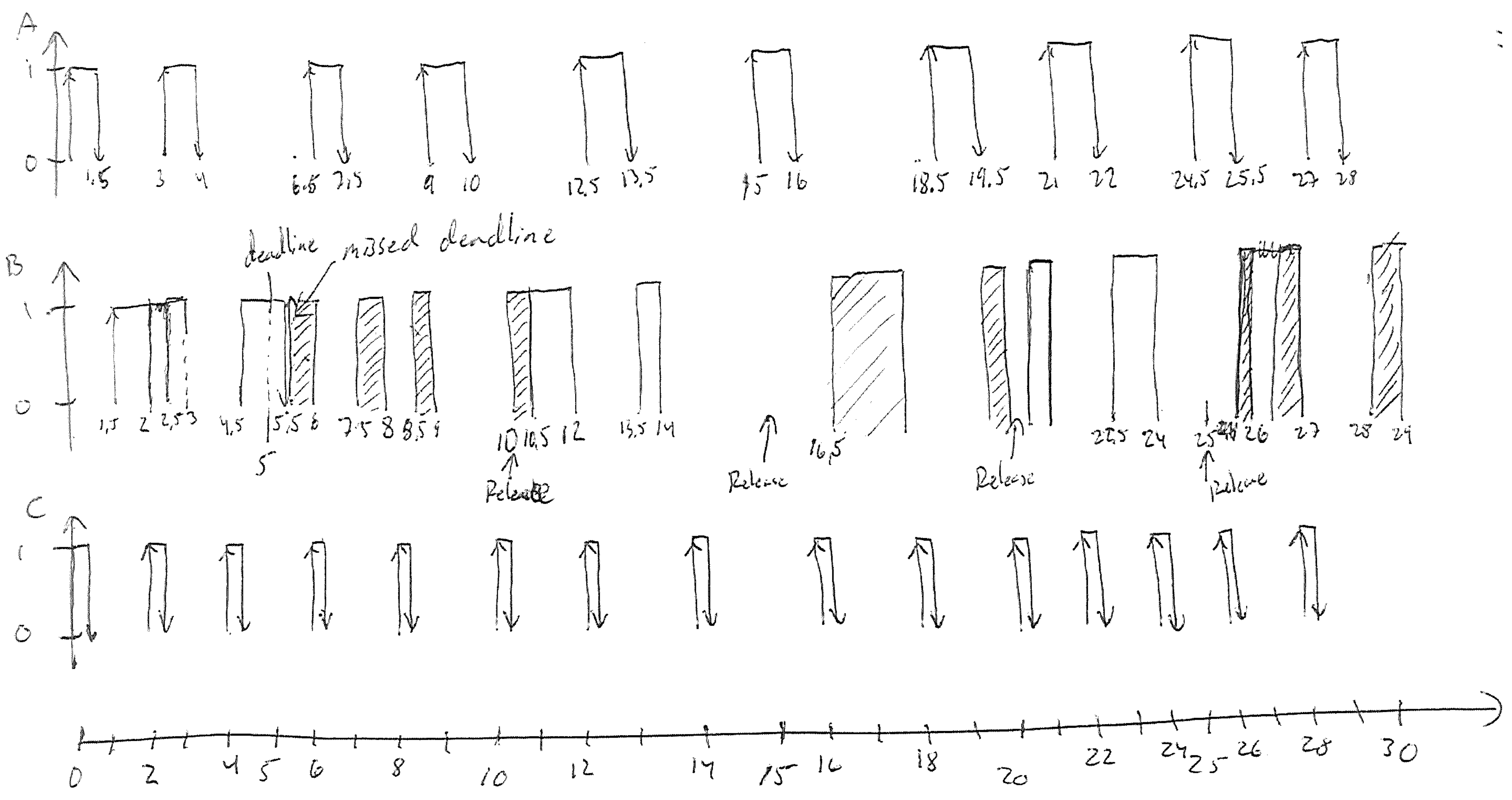
Finally solve for low prio task (B).

$$R_B^1 = 2 + \left\lceil \frac{2}{2} \right\rceil \cdot 0,5 + \left\lceil \frac{2}{3} \right\rceil \cdot 1 = 3,5$$

$$R_B^2 = 2 + \left\lceil \frac{3.5}{2} \right\rceil \cdot 0,5 + \left\lceil \frac{3.5}{3} \right\rceil \cdot 1 = 5$$

$$R_B^3 = 2 + \left\lceil \frac{5}{2} \right\rceil \cdot 0,5 + \left\lceil \frac{5,5}{3} \right\rceil \cdot 1 = 5,5$$

$$R_B^4 = 2 + \left\lceil \frac{5,5}{2} \right\rceil \cdot 0.5 = \left\lceil \frac{5,5}{3} \right\rceil \cdot 1 = \underline{\underline{5,5}}$$



A
B
C
1
0
1,5
3
4
6.5
7,5
9
10
12,5
13,5
15
16
18,5
19,5
21
22
24,5
25,5
27
28
deadline
missed deadline
1,5
2
2,5
3
4,5
5
5,5
6
7.5
8
8,5
9
10
10,5
12
13,5
14
16,5
22,5
24
25
26
27
28
29
Release
Release
Release
Release
0
2
4
5
6
8
10
12
14
15
16
18
20
22
24
25
26
28
30

4/4

5/

a) $x = p \land q \iff \neg x = \neg p \lor \neg q$

We have to show if

$G(x) \iff \neg F(\neg x)$

This is true, we show $\Rightarrow$ and $\Leftarrow$

Show $\Rightarrow$

If a trace fulfills $G(x)$ then x holds in every state. by definition. Thus $\neg x$ does not hold in any state, thus $F(\neg x)$ does not hold, thus $\neg F(\neg x)$ has to hold.

Show $\Leftarrow$

If a trace fulfills $\neg F(\neg x)$, we know that $F(\neg x)$ does not hold. Thus there does not exist a state where $\neg x$ holds, thus x has to hold in every state.

b) This is true

$Gp \Rightarrow Fq \iff p \, U \, (q \lor \neg p)$

$\iff \neg Gp \lor Fq$

$\iff F \neg p \lor Fq$

$\iff F(\neg p \lor q)$

$\iff true \, U \, (\neg p \lor q)$

$\iff p \, U \, (q \lor \neg p)$

5)

c) true

$FGp \Rightarrow GFq \iff G(p \, U (q \lor \neg p))$

$FGp \Rightarrow GFq$

$\iff \neg FGp \lor GFq$

$\iff GF\neg p \lor GFq$

$\iff G(F\neg p) \lor G(Fq)$

$\iff G(F\neg p \lor Fq)$

$\iff G(F(\neg p \lor q))$

$\iff G(p \, U (q \lor \neg p))$

d) $GFp \Rightarrow GFq \iff G(p \Rightarrow Fq)$

This is false.

Consider a trace that satisfies p only a finite number of times, and that q is never true at any state after p is true.

GFp is then false, and thus $GFp \Rightarrow GFq$ is true.

However, $G(p \Rightarrow Fq)$ is false in this situation.