Assignment 2:
Peripherals, scheduling, concurrency

1DT056: Programming Embedded Systems
Uppsala University

January 27th, 2011

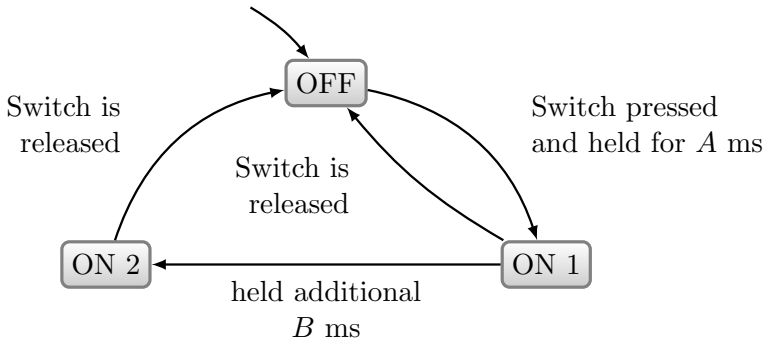
Use of the development environment

μVision is now installed centrally on the lab computers (in room 1313) in the directory `G:\Programs\Keil`. It is no longer necessary to download and install the IDE individually.

Exercise 1 Switch

In this exercise, you are supposed to implement software for a *multi-state* push-button switch. Imagine the "forward" button of a video player: pressing the button for a short time (say, less than 2s) will make the player forward slowly, while keeping the button pressed for a longer time makes the player change to a fast-forward mode.

The following state chart describes the behaviour of such a multi-state switch:



1. We assume that a push-button switch is connected to Pin 0 of GPIO C of an STM32F103 micro-controller. Implement a task function that realises the multi-state switch behaviour as described above, for $A = 0.2$s and $B = 1.8$s. Start from the empty uVision project `http://www.it.uu.se/edu/course/homepage/pins/vt11/lab_env.zip`.

The effect of "ON 1" shall be to put a 1 on Pin 1 of GPIO C, the effect of "ON 2" to put a 1 on Pin 2.

2. Write a μVision debug function that exercises your implementation. The debug function is supposed to implement a test case that takes your system through all four transitions of the switch state chart (ignore the initial transition to state "OFF"). The debug function also has to observe the values on Pins 1 and 2 of GPIO C and to assess whether the behaviour is correct.
   More information on debug functions are available at `http://www.keil.com/support/man/docs/uv4/uv4_debug_functions.htm` and `http://www.it.uu.se/edu/course/homepage/pins/vt11/lab1`.

Exercise 2 Rate-monotonic scheduling

Rate-monotonic scheduling is a strategy to choose priorities of tasks in a real-time system with preemptive multi-tasking. Suppose that a system has to execute n periodic tasks, where:

- the instances of task τ_i (for $i \in \{1, \ldots, n\}$) are regularly activated; the time T_i between two activations is called the *period* of τ_i.
- all instances of task τ_i have the worst-case execution time C_i.
- the deadline of each instance of a task τ_i is the point in time when the next instance arrives.
- the tasks do not share any resources, or have any other interdependencies.

For instance, the task described in Assignment 1, exercise 5 is a task with period $0.1s$; the worst-case execution time C_i is the maximum time needed in an iteration of the task's main loop.

In rate-monotonic scheduling, the priorities of tasks are chosen inversely related to the task's period: if $T_1 < T_2$, then the priority of τ_1 will be higher than that of τ_2. If priorities are chosen like this, then preemptive fixed-priority scheduling is guaranteed to meet the deadlines of all tasks if the following inequality holds (the system is *schedulable*):

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

The left expression is called the *utilisation factor* U of the set of tasks.

Consider the tasks running in the elevator control system described on the page `http://www.it.uu.se/edu/course/homepage/pins/vt11/lab1`; for sake of simplicity, we ignore commonly used resources of the tasks at this point.

1. Using the result about rate-monotonic scheduling, can we say anything about schedulability of the control system (or parts of it), given the fixed priorities specified on the lab page?
   Reasonable estimates of the worst-case execution times C_i are:
   - Position processor: $\leq 20\mu s$ (per iteration)
   - Button processor: $\leq 50\mu s$
   - Actuator module: $\leq 50\mu s$
   - Safety module: $\leq 30\mu s$
   - Planning module: $\leq 150\mu s$
2. Describe how such execution time estimates can practically be derived through simulation.

Exercise 3 Concurrent datastructures

Consider the following implementation of an (unbalanced) binary search tree (the source code is also available for download: `http://www.it.uu.se/edu/course/homepage/pins/vt11/search_tree_original.c`):

```
typedef struct Node {
  int data;
  struct Node *left;
  struct Node *right;
} Node;

Node nodes[64];

Node *root = NULL;

int insertNode(Node *n) {
  Node *currentNode;
  Node **currentPtr;

  assert(!n->left && !n->right);

  currentPtr = &root;

  while (*currentPtr) {
    currentNode = *currentPtr;
    if (n->data < currentNode->data) {
      currentPtr = &currentNode->left;
    } else if (n->data > currentNode->data) {
      currentPtr = &currentNode->right;
    } else {
```

```
        // already present
        return 0;
      }
    }

    *currentPtr = n;
    return 1;
  }
```

We would like to use this datastructure in a concurrent setting, i.e., have multiple tasks be able to invoke the function "insertNode" concurrently. In this context, we are interested in *linearisable* behaviour of the datastructure: if two tasks A and B are executing "insertNode" at the same time:

A:		B:
insertNode(n_1)	‖	insertNode(n_2)

the overall result shall be the same as if the function invocations had happened sequentially:

- insertNode(n_1); insertNode(n_2); or
- insertNode(n_2); insertNode(n_1);

In particular, afterwards both n_1 and n_2 are supposed to be present in the search tree (unless the values already existed in the tree before).

1. Give an example that shows that the search tree implementation, as it is given here, does not guarantee linearisable executions.
2. A simple solution to achieve linearisability is protect the search tree using a mutex semaphore:

```
  xSemaphoreHandle treeLock;

  int insertNode(Node *n) {
    Node *currentNode;
    Node **currentPtr;
    xSemaphoreTake(treeLock, portMAX_DELAY);

    // ...
          xSemaphoreGive(treeLock);
          return 0;
    // ...

    xSemaphoreGive(treeLock);
    return 1;
```

```
}

int main(void) {
  treeLock = xSemaphoreCreateMutex();
  // ...
}
```

However, this approach is rather coarse-grained and defensive, and would not scale well if the search tree were big and many tasks were to access it concurrently.
Is it possible to use a finer locking scheme, where each node is separately protected by its own semaphore (with a further semaphore protecting the root pointer)? Develop such a version of the search tree and the "insertNode" function and argue that it achieves linearisability. Keep the amount of locks held at any point in your algorithm minimal.
In such an implementation, Nodes would be defined as:

```
typedef struct Node {
  int data;
  struct Node *left;
  struct Node *right;
  xSemaphoreHandle lock;
} Node;
```

3. Compare the two solutions (the one with the global lock, and the fine-grained-lock version) in the context of embedded systems. What are the advantages and disadvantages of each?

Submission

Solutions to this assignment are to be submitted before the lecture on

Wednesday, February 3rd, 2011, 13:15, room 1245.

No solutions will be accepted after the lecture.
Make sure that you have specified your name and your personnummer on your solution.