Assignment 2:
Peripherals and concurrency

1DT056: Programming Embedded Systems
Uppsala University

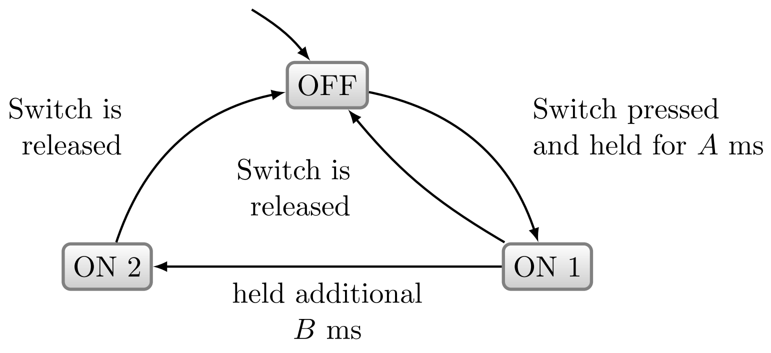
January 25th, 2012

You can achieve a maximum number of 20 points in this assignment. 12 out of the 20 points are required to pass the assignment. Assignments are to be solved by students individually.

Exercise 1 Switch (8p)

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 = 0.2s and B = 1.8s. Start from the empty uVision project `http://www.it.uu.se/edu/course/homepage/pins/vt12/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. **(5p)**

2. Write a μVision debug function that exercises your implementation. **(3p)** 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`.

Exercise 2 Concurrent datastructures (12p)

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. **(3p)**
2. A simple solution to achieve linearisability is protect the search tree using a mutex semaphore: **(4p)**

```
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? **(2p)**
4. Suppose besides function insertNode(.) you also implemented a function removeNode(.) which deletes entries from the tree. A discussion how nodes can be removed from binary search trees can be found, e.g., on `http://en.wikipedia.org/wiki/Binary_search_tree#Deletion`. **(3p)**
   (a) Show how the function removeNode(.) can be made thread-safe (so that it can be called concurrently by multiple tasks), in a similar fine-grained manner as in question 2. You do not have to give a full implementation of removeNode(.), it is enough to discuss in which order locks have to be acquired and released.
   (b) Now, imagine that two threads are executed concurrently, one of which uses your implementation of insertNode (question 2) to insert elements into the search tree, while the other task removes elements using removeNode (using your locking scheme from **(a)**). Is linearisability ensured in this scenario? Justify.

Submission

Solutions to this assignment are to be submitted
via email to othmane.rezine@it.uu.se before the exercise session on
Thursday, February 2nd, 2012, 10:15 (room 1245).

Make sure that your solution is clearly marked with your name and personal number. No solutions will be accepted after the exercise session.