Assignment 5: Fault tolerance and Lustre

1DT056: Programming Embedded Systems
Uppsala University

February 23rd, 2011

A total number of 12 out of 20 points has to be achieved to pass the assignment. As before, the assignment as a whole will only be graded pass/-fail.

Exercise 1 Fault injection and error correction

This exercise is about testing and improving the tolerance of a linked datastructure in C w.r.t. memory faults. As in some of the previous assignments (e.g., assignment 2), we will work with an unbalanced binary search tree. The implementation of the datastructure has been changed slightly to make it more suitable for the following tasks, in particular by storing 16 bit array indexes (type NodeOffset) instead of pointers in the tree. A μVision project with the complete datastructure is available here: `http://www.it.uu.se/edu/course/homepage/pins/vt11/05_search_tree.zip`.

```
// We now store pointers within the tree in form of
// 16 bit offsets, with respect to the array "nodes"
typedef u16 NodeOffset;

// Magic offset that is used to encode absence of sub-trees
// in the search tree
#define NULL_OFFSET ((NodeOffset)0xFFFF)
#define MAX_NODE 16

typedef struct Node {
  int data;
  NodeOffset left, right;
} Node;

Node nodes[MAX_NODE];
NodeOffset root = NULL_OFFSET;

/*-------------------------------------------------------------*/
// Function to convert offsets to node pointers, and vice versa

Node* offset2Node(NodeOffset offset) {
  return nodes + offset;
}

NodeOffset node2Offset(Node *n) {
  return n - nodes;
}
```

```
int isNullOffset(NodeOffset offset) {
  return offset == NULL_OFFSET;
}

/*-------------------------------------------------------------------*/

/**
 * Insert a node into the tree, in case the represented data
 * element is not yet present in the tree
 */
int insertNode(Node *n) { /* ... */ }

/**
 * Recursively empty the tree
 */
void emptyTree(void) { /* ... */ }

/**
 * Check whether the search tree is well-formed
 */
int treeIsWellformed(void) { /* ... */ }

/*-------------------------------------------------------------------*/

void treeWorker(void *params) {
  int i, res;
  for (;;) {
    // Fill the tree with numbers
    printf("Filling tree\n");
    for (i = 0; i < MAX_NODE; ++i) {
      nodes[i].data = i * 37 % MAX_NODE;
      nodes[i].left = NULL_OFFSET; nodes[i].right = NULL_OFFSET;
      res = insertNode(nodes + i);
      assert(res);
    }
    vTaskDelay(5 / portTICK_RATE_MS);

    // Check that the tree is still wellformed
    printf("Checking tree\n");
    assert(treeIsWellformed());
    vTaskDelay(5 / portTICK_RATE_MS);

    // Empty the tree
    printf("Emptying tree\n");
    emptyTree();
    vTaskDelay(5 / portTICK_RATE_MS);
  }
}

int main( void ) {
  prvSetupHardware();
  xTaskCreate(treeWorker, "treeWorker", 300, NULL, 1, NULL);
  // ...
}
```

1. We will first assess how robust the datastructure is w.r.t. memory faults. For that purpose, add a second task to the system that injects memory faults into the datastructure, by randomly flipping bits in the left/right fields of the tree nodes. Random numbers can be generated **(2p)**

using the rand() function. The outline of the fault injecting task is:

```
void faultInjector(void *params) {
  for (;;) {
    // inject a fault at this point by flipping some
    // bit in the left/right offsets in the search tree

    vTaskDelay((rand() % 20 + 3) / portTICK_RATE_MS);
  }
}
```

Give the source code of this task and report your experiences with injecting memory faults; which kinds of failures occur?

2. We now want to increase the fault tolerance of the datastructure by introducing error-correcting codes, implemented in software. For simplicity, we use a primitive repetition code for the offset fields of the tree. This is done by replicating each of 5 databits $(d_4, d_3, d_2, d_1, d_0)$ in a 16 bit integer (since the tree nodes are taken from an array of 16 elements, 5 databits are sufficient to store indexes): **(4p)**

	d_4	d_3	d_2	d_1	d_0	d_4	d_3	d_2	d_1	d_0	d_4	d_3	d_2	d_1	d_0
15					10		8			5					0

Since every bit is now stored three times, we can correct single bit-flips occurring in any of the five data bits. E.g., if the three copies of d_0 turn out to have different values in memory, we will assume that the majority is right, and that the third bit was affected by a memory fault.
Change the implementations of the functions offset2Node, node2Offset, and isNullOffset to realise this repetition code. It is not necessary to modify the functions insertNode, emptyTree, or treeIsWellformed. Repeat your fault tolerance experiments from the first question and report the results.
Option: if you want to earn two extra points, you can also implement (more efficient) Hamming codes instead of repetition codes: `http://en.wikipedia.org/wiki/Hamming_code`

3. With error-correcting codes it is necessary to periodically check memory contents for potential bit flips (and correct them), so that accumulation of memory faults, which might no longer be correctable, is avoided. Add a *memory scrubbing* task to the system that periodically runs through the nodes array and corrects flipped bits. **(2p)**
Note that you need to protect the nodes array using a (single, global) mutex semaphore, which has to be acquired by the scrubbing task, as well as the functions insertNode, emptyTree, or treeIsWellformed when making changes to the array.

Exercise 2 Basic Lustre nodes

In this exercise, you will implement a few basic programs in the Lustre programming language. Before you submit your solutions, simulate each program using the Luke tool, in order to identify and eliminate potential bugs.

Luke binaries for various platforms and further resources are provided on the following page: `http://www.it.uu.se/edu/course/homepage/pins/vt11/lustre`.

1. In the lecture, a node Sum has been implemented that returns, at each instant, the sum of all inputs that the node has seen up to that point. Now implement SumReset, a version of Sum with an additional Boolean input Reset that makes Sum start over from 0: **(2p)**

   ```
   node SumReset(X : int; Reset : bool)
     returns (S : int);
   ```

2. Implement a node Average with the signature **(2p)**

   ```
   node Average(X : int; Reset : bool)
     returns (A : int);
   ```

   that outputs the average of the stream X since the last Reset.

3. Implement a node HasHappenedWithin with signature **(2p)**

   ```
   node HasHappenedWithin(X : bool; N : int)
     returns (Y : bool);
   ```

   that returns whether X has happened within the last N execution steps.

Exercise 3 Door control system in Lustre

We will now use Lustre to develop a simple system that controls the entrance and exit of a laboratory dealing with bio-hazardous materials. The idea is that, in order to enter or leave the laboratory, you have to go through two doors in sequence. However, these doors cannot be open at the same time, because otherwise air infected with bacteria or viruses might freely flow out of the laboratory. A biosensor reports to the door system if any dangerous organisms exist in the area between the doors.

The physical shape of the door system is as follows: The inner door to the laboratory is called door 1; the outer door is called door 2. Each door has a door sensor, a button, and a lock. The door sensors check if the corresponding door is open, producing the boolean signals `Open1` for door 1 and `Open2` for door 2. The buttons can be pushed by a human to unlock the corresponding door, producing the boolean signals `Button1` and `Button2`. The locks are controlled by the system; they listen to the boolean signals `Unlock1` and `Unlock2`. Finally, the biosensor produces a Boolean signal

`BioHazard`, that is true precisely when there are dangerous organisms in the room between the two doors.

The intended use of the system is as follows. In the beginning, both doors are closed and locked. When a person wants to leave the laboratory, she pushes the button belonging to door 1. If the outer door is not open, the inner door will be unlocked and can be pushed open by the human. Then, she waits until the door closes, before she pushes the button of the outer door. The outer door only unlocks if there is no bio-hazardous situation. If it unlocks, the human can push the door open and walk out. People can walk into the laboratory in a similar way.

The following three requirements have to hold for the system:

R1 A door should not be locked when it is open.
R2 The two doors should not be open at the same time.
R3 When there is a bio-hazardous situation, the outer door must remain closed and locked.

In order to ensure these requirements, the control system has to rely on a *environment assumption* describing the behaviour of a physical door:

E1 A door cannot be opened when it is locked.

The door controlling node has the following Lustre interface:

```
node System( Open1, Button1, Open2, Button2, BioHazard : bool )
  returns ( Unlock1, Unlock2 : bool );
```

1. Give an implementation of the node `System`. Simulate your implementation using Luke and make sure that it behaves consistently with the description given above. **(3p)**
2. We now want to verify that the `System` implementation satisfies the requirements **R1**, **R2**, **R3**. As explained in the lecture, this is done by constructing a synchronous observer containing the requirements in form of Lustre expressions. The observer will also assume that **E1** holds (for both doors) during the entire execution of the system, which is done with the help of the `Sofar` node. **(3p)**
   This observer has the form:

   ```
   node EnvironmentDoor( Open, Unlock : bool )
     returns ( E1 : bool );
   let
     E1 = ...;
   tel

   node ReqSystem( Open1, Button1, Open2, Button2,
                   BioHazard : bool )
   ```

```
  returns ( R1, R2, R3 : bool );
  var Env, Unlock1, Unlock2 : bool;
let
  (Unlock1, Unlock2) =
    System( Open1, Button1, Open2, Button2, BioHazard );
  Env =
    Sofar(     EnvironmentDoor( Open1, Unlock1 )
           and EnvironmentDoor( Open2, Unlock2 ) );

  R1 = Env => ...;
  R2 = Env => ...;
  R3 = Env => ...;
tel
```

Replace the `...` in the code with Lustre expressions faithfully formalising **E1**, **R1**, **R2**, **R3**. Verify that your `System` implementation satisfies the requirements using Luke; if verification fails, you have to correct either your implementation or your formalisation of the requirements.

Submission

Solutions to this assignment are to be submitted by

March 14th, 2011.

You can submit your solution during the lab session on March 11th (15:00 – 17:00, room 1313) or by email to othmane.rezine@it.uu.se.

Make sure that you have specified your name and your personnummer on your solution.

If you submit your solutions via email, submit in form of a *single* PDF file!