Injection of Memory Faults

1DT056: Programming Embedded Systems
Uppsala University

February 17, 2012

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/vt12/robust_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 using the rand() function. The outline of the fault injecting task is:

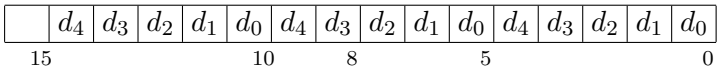
```
void faultInjector(void *params) {
  for (;;) {
```

```
    // 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):



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.

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.

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, treeIsWellformed and the treeWorker task when accessing the array.

Answer of exercise 1

1.

```
void faultInjector(void *params) {
  int n, l, bit;

  for (;;) {
    l = rand() & 1;                 // left or right pointer
    n = rand() % MAX_NODE;          // node to inject fault into
    bit = rand() % (sizeof(NodeOffset) * 8); // bit to modify

//    printf("Injecting fault ... node %d, bit %d\n",
//           n, bit + l * sizeof(NodeOffset) * 8);

    if (l)
      nodes[n].left ^= 1 << bit;
    else
      nodes[n].right ^= 1 << bit;

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

For experiments, it is useful to initialise the random number generator with different seeds in the main function ("srand(X)"), so that the effect of different faults can be observed. What usually happens during execution is the following:

- Due to the injected faults, some of the left/right offsets in the tree will modified to values outside of the range $[0, 16)$.
- Subsequent calls offset2Node(node−>left) or offset2Node(node−>right) will return addresses n to nodes that are outside of the nodes array.
- Subsequent accesses n−>left or n−>right will try to access memory that does not exist or is not accessible, leading to an error/trap during program execution.

2.

```
typedef u16 NodeOffset;

#define NULL_OFFSET ((NodeOffset)0x7FFF)
#define MAX_NODE 16

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

Node nodes[MAX_NODE];
NodeOffset root = NULL_OFFSET;

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

#define BLOCK_SIZE 5
#define OFFSET_MASK ((NodeOffset)((1 << BLOCK_SIZE) - 1))
```

```
NodeOffset encode(NodeOffset o) {
  return o | (o << BLOCK_SIZE) | (o << (2 * BLOCK_SIZE));
}

NodeOffset decode(NodeOffset o) {
  u16 block1 = o & OFFSET_MASK;
  u16 block2 = (o >> BLOCK_SIZE) & OFFSET_MASK;
  u16 block3 = (o >> (2 * BLOCK_SIZE)) & OFFSET_MASK;

  return
    ((block1 ^ (~block2)) & block1) |
    ((block2 ^ (~block3)) & block2) |
    ((block1 ^ (~block3)) & block3);

  /*
     Alternative scheme, suggested after the lecture
     (and much nicer):

     return
       (block1 & block2) | (block3 & (block1 | block2));
   */
}

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

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

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

The resulting program is very robust against the kind of faults injected by the faultInjector. A very high rate of faults is necessary to still provoke any failures, e.g., as simulated by brute-force methods such as randomly executed statements nodes[n]. left =rand().

3.

```
void memoryScrubber(void *params) {
  int n, left, right;

  for (;;) {
    // go through the array and fix corrupted offsets
    for (n = 0; n < MAX_NODE; ++n) {
      left = nodes[n].left;
      right = nodes[n].right;
      if (left != encode(decode(left)) ||
          right != encode(decode(right))) {
        xSemaphoreTake(treeLock, portMAX_DELAY);
        nodes[n].left = encode(decode(nodes[n].left));
        nodes[n].right = encode(decode(nodes[n].right));
        xSemaphoreGive(treeLock);
      }
    }

    vTaskDelay(100 / portTICK_RATE_MS);
  }
}
```