Session 6 Linked List

6.1 Introduction

In this document we will provide you with a specification of a Linked List interface. The goal of this (optional) assignment is to implement this specification in both C and RISC-V. You will start working on this implementation in the current exercise session. However, to fully complete the assignment will take a lot more time than just one session. You can, if you choose, work on this interface for the rest of the semester, at your own pace. Questions can be asked in the current session, in the last session of the semester (this will be a revision session) and on the discussion forum.

Implementing the full interface is a great way to prepare yourself for the RISC-V programming part of the exam. C programming will not be evaluated on the exam. However, you will need to translate C code to RISC-V code. Thus, it is smart to also implement the C part. Furthermore, implementing the interface in RISC-V will be a lot easier when you have C code that you can translate, especially if you wrote that C code yourself.

6.2 Preparation

To prepare for the session, download `linked_list.zip` from Toledo. Inside, you will find the files below. Only files followed by an asterix should be edited.

`c/linked-list.h` C header file containing the definition of the interface functions.

`c/linked-list.c*` The template in which you need to implement all function definitions.

`c/linked-list-tests.c` The C test suite. This file contains the `main` function that executes all tests in the suite. Look at the implementation of the different functions to get an idea of how to use the linked list interface.

`c/Makefile` File that is used to build the C project. Use the command `make` inside the `C` folder to generate the executable `linked-list`. Afterwards, use the command `./linked-list` to run the test suite. You can try this out immediately after downloading. It should simply fail all tests.

`asm/linked-list-tests.asm` The RISC-V test suite. Keep this file open while writing your RISC-V functions. Whenever the test suite fails a certain test, the failing assertion will be highlighted in the simulator. This can be used to quickly discover bugs in your implementation.

`asm/malloc.asm` Our own malloc implementation, which is our solution to the complex allocator exercise from session 4. In your implementation you are expected to use malloc and free to allocate memory, as you would do in C.

`asm/main.asm*` The main function that executes the test suite. It's possible to write your own tests in this main function, before executing the test suite, if you want.

`asm/list-*.asm*` A seperate file for each interface function that you need to implement. We use seperate files per functions because RISC-V implementations can become long and working in a big file can be quite an annoying experience in RARS.

6.3 RARS set-up

It's very important to enable the setting *Assemble all files in directory.* This will allow you to easily work with the multiple files. Of course, don't forget to enable *Initialize program counter to global main if defined.*

6.4 Test suites

Both the C and RISC-V are delivered with an extensive test suite. All interface functions (except the printing function) are extensively tested for many different edge cases.

The C suite should be relatively straightforward to use. Whenever you execute your program, the test suite will test each of the interface functions. Whenever a certain part of a test fails, you will get an assertion error together with a line number in the test suite. Check out the error in the suite to figure out what your implementation did wrong.

The RISC-V suite is a little bit more complicated. It follows exactly the same structure as the C suite, thus, if you get lost, take a look at the C implementation as well. Since there is no direct way of using assertions in RISC-V, we hacked our own version in the simulator. We use macro's to execute the same assertions as the ones in C. When the assertion fails, however, we throw an exception by executing `lw t0, 0xdeadbeef`. The exception you will see in RARS whenever an assertion fails is of the following structure:

```
line 131: Runtime exception at 0x004002ac:
Load address not aligned to word boundary 0xdeadbeef
```

While this error message means nothing for your program, the line number exactly points you to the assertion in `linked-list-tests.asm` that failed for your program. By double clicking the message, RARS will highlight this line of code in the source file. Thus, this system makes it very easy to see where the test suite fails and why.

Whenever you have an exception, make sure to check both the `messages` and the `Run I/O` tab. For errors that occur often, our exception handler prints messages with hints to `Run I/O` tab. This will hopefully speed up your debugging process.

6.5 General Tips

- Always check both `Messages` and `Run I/O` in case of errors
- Implement the interface functions in the given order. Later function implementations might need to call earlier functions.
- If you located a piece of RISC-V code that is potentially the cause of your problem, use the `ebreak` instruction to have the debugger automatically pause at that location.
- A good order in which to solve this assignment is to implement a function first in C, then translate this to RISC-V, then move on to the next C function and so forth. Improving at C will help your understanding of assembly and the opposite is true as well. Having the function implementation fresh in your memory will also help your assembly implementation.
- Follow calling conventions. Our test suite actively tries to detect violations of the conventions.
- Before translating a complex function to RISC-V it might be a good idea to explicitly write down (in comments) which registers you will allocate to which variables This makes it much easier to keep track of sometimes very hard to read RISC-V code. It might make your life a lot easier if you have to fix a bug at a later time.

- `malloc` can return `0`, for example when there is no more memory available. Our interface defines an explicit return code for this situation. Make sure you don't assume `malloc` returns a valid address. Our RISC-V suite will actively try to crash your program by calling your functions while simulating an out-of-memory situation.

6.6 The interface

6.6.1 Error codes

All functions, except `list_create`, return a status code describing wether or not the function was executed successfully. These are the possible codes:

```
typedef enum
{
    OK = 1,
    UNINITIALIZED_LIST = -1,
    OUT_OF_MEMORY = -2,
    INDEX_OUT_OF_BOUNDS = -3,
    UNINITIALIZED_RETVAL = -4,
} status;
```

`UNINITIALIZED_LIST` is returned whenever a function is called with the value `NULL` filled in as the list parameter. `OUT_OF_MEMORY` occurs whenever `malloc` returns 0, and thus the function failed to execute. `INDEX_OUT_OF_BOUNDS` occurs in those functions that ask an `index` parameter. If this `index` is an impossible value, return this status code. `UNINITIALIZED_RETVAL` is used whenever a function provides a return value via a parameter. Thus, the function is supplied a pointer to which to write this return value. If that pointer points to `NULL`, return this error code.

6.6.2 Data structures

These are the C data structures used to represent our linked list:

```
struct ListNode
 {
    int value;
    struct ListNode *next;
 };

 struct List
 {
    struct ListNode *first;
 };
```

Thus, a list is represented using the `struct List`, which contains a pointer to the first element of the list. A node in the list is represented using the `struct ListNode`, which contains a value and a pointer to the next node in the list.

6.6.3 Functions

A brief explanation of the expected operation of each function is described below. Consult the test suite for an in-depth look into the workings of this API.

```
struct List *list_create();
```

Creates a new list by allocating a `struct List` and returning the address of this list.

```
status list_append(struct List *list, int value);
```

Appends a value to the end of an existing list by allocating a new `struct ListElement` and adding it to the chain. Returns either `OK`, `OUT_OF_MEMORY` or `UNINITIALIZED_LIST`.

```
status list_get(struct List *list, int index, int *value);
```

Get the value at position `index` in an existing list. Store this value at the address stored in the `int *value` pointer. Returns either `OK`, `INDEX_OUT_OF_BOUNDS`, `UNINITIALIZED_LIST` or `UNINITIALIZED_RETVAL`.

```
status list_print(struct List *list);
```

Prints all elements of the list to the console. Returns either `OK` or `UNINITIALIZED_LIST`. This function is not tested by the test suites.

```
status list_remove_item(struct List *list, int index);
```

Remove the item with the provided index from the list. Returns either `OK`, `INDEX_OUT_OF_BOUNDS` or `UNINITIALIZED_LIST`.

```
status list_delete(struct List *list);
```

Delete the provided list. Free up all memory used by both the individual elements and the enclosing list structure. In general it's not possible to test whether or not memory was freed correctly. Our test suite might miss mistakes here. Returns either `OK` or `UNINITIALIZED_LIST`.

```
status list_insert(struct List *list, int index, int value);
```

Insert an element wtih the provided value at the provided index in the list. Thus, executing with index 0 should insert the new element in the front of the list, and so forth. Returns either `OK`, `INDEX_OUT_OF_BOUNDS`, `UNINITIALIZED_LIST` or `OUT_OF_MEMORY`.