Datorteknik
Lab 3: Memory Organization

Todor Stoyanov

August 4, 2020

1 Objectives and Lab Materials

In this lab you will learn how to store and access arrays in C.

The matrerials for this lab are:

- A PC with ThinLinc client installed.
- A ThinLinc server running Ubuntu.

Complete the following tasks. Provide all source code, including appropriate inline comments. In the report explain how the code works and describe what test you performed the validity of your code.

1.1 Preliminaries

Before starting with this lab please verify you have performed these preliminary steps (Note: same procedure as in Lab 2):

- Start a ThinLinc client. Connect to thinlinc.oru.se (or if it doesn't work try 130.243.110.30), using your EDUNET credentials.
- You are now running on one of three ThinLinc server nodes, running Ubuntu Linux. Note: to get out, simply log out from the top right corner.
- The remainder of this lab should be coded in C. You can do that either on the raspbian image, or directly on the ThinLinc server. As the raspbian emulator is slow, the suggested option is to develop, compile, and test your code directly on the ThinLinc server.
- Note: all your code should compile and link with standard `gcc` without any compilation warnings.

2 Task 1: Array Storage (15 points)

In this task you will implement storage and fetching into a 2D byte array. This is a modified version of Lab 9 from the book. The point of this lab is to use a 1D character buffer in order to emulate 2D arrays in C. You will declare two functions: `two_d_store` and `two_d_fetch` to implement storage and access from a 2D array of arbitrary type. To do that follow these steps:

- Implement a function `two_d_alloc` that allocates a `char*` to store an array of $N \times M$ elements of a given size. The function should take as arguments the number of rows, number of columns, and size of each element.
- Implement a function `two_d_dealloc` which deallocates a two dimensional array stored in a character buffer.
- Implement a main function that uses the above function to allocate and then deallocate an array.
- Implement a function `two_d_store` which stores an integer argument into a particular row and column of the array. The function should in addition take as arguments the address of the array, the size of an element, the row and column at which to store the value, as well as the numbers of rows and columns in the array. Assume row-major order.
- Implement a function `two_d_fetch` which returns the value of an integer stored in the memory array. The function should take as arguments the row and column at which the integer is stored, the numbers of rows and columns in the array, the address of the array in memory, as well as the size of an integer. Assume the array is initialized in row-major form.
- Test the functions you just wrote by allocating a 2D array, storing and then fetching a number of integer arguments. Verify that your implementation works correctly.
- Test your program for boundary conditions.
- Re-write `two_d_store` and `two_d_fetch` to use column-major format. Provide these as separate functions in your report. Test your implementations.
- Implement a version of `two_d_store` and `two_d_fetch` which operates on integer pointers, instead of integers by value. Instead of returning an integer, return the memory at which the integer is stored. You are free to choose between row-major or column-major storage.
- Implement a general version of `two_d_store` and `two_d_fetch` which use the memory address to an arbitrary data type (e.g., a double, or a struct). Again, you are free to choose between row-major and column-major storage. Test your implementations. You may want to use the `memcpy` function to copy bytes into the array.

3 Task 2: Memory Dump (5 points)

Implement a memory dump program which can read through a character array and prints out each word in hexadecimal notation. Separate words in groups of four words per line, where each word is reperesented by 8 hexadecimal numbers. Print the memory address of the first byte on every line. Test your memory dump on the character array you implemented in Task 1. Add four more columns to your output and interpret every byte as an ASCII character. If the byte does not contain a valid character, print a dot instead.

4 Task 3: Linked Lists (5 points)

Implement a struct which holds an element from a linked list. Every node in the list should contain an integer value, as well as a pointer to the next element in the list. Use the 2D array implementation from Task 1 to allocate an array of ten linked list elements. In each cell of the array store a linked list node that holds a pointer to the next element, as well as an incrementing integer value (i.e., 0, 1, 2, ...). Set the pointer of the last element to NULL. Use the memory dump program from Task 2 to examine the contents of the array. In your lab report, explain what you can see in the memory dump and identify the memory address where the fifth node in the list is located.