Viktor Rask
Amanda Håkansson
DT509G

Lab 3

Computer Architectures for MSc in Engineering

Viktor Rask
Amanda Håkansson
DT509G

Innehåll

Viktor Rask
Amanda Håkansson
DT509G

Introduction

The purpose of this lab was to learn about memory organization, this was done by first implementing a function in C for storage and one for fetching into a 2D byte array. The second task was to implement a memory dump that was able to read through an array and then print each word in both hexadecimal notation and as ASCII characters.

Viktor Rask
Amanda Håkansson
DT509G

Task 1

The purpose of task 1 was to create functions that was able to allocate an array, insert and gather values inside the array and then deallocate it.

Code

Main task 1:

```c
#include <stdio.h>
#include "functions.c"

int main(){
    static int rows = 4;
    static int columns = 4;
    char *array = two_d_alloc(rows, columns, sizeof(void*));
    if(array != NULL){
        int number = 0;
        for(int row = 0; row < rows; row++){
            for(int col = 0; col < columns; col++){
                void *pointer;
                pointer = &number;
                two_d_store_general(array, pointer, sizeof(void*), row, col, rows, columns);
                number++;
            }
        }
        for(int row = 0; row < rows; row++){ /* print values in array */
            for(int col = 0; col < columns; col++){
                printf("%d ", *(int*)two_d_fetch_general(array, sizeof(void*), row, col, rows, columns));
            }
            printf("\n");
        }
    }
    two_d_dealloc(array);
    printf("\n");
    return 0;
}
```

The main of task 1 tests the functions two_d_store_general and two_d_fetch_general. This is done in two nested for-loops, stepping through all rows and columns in the array and first storing all the elements in the array and then fetching them.

```c
int two_d_store_general(char* array, void* element , int size, int row_index, int col_index, int num_of_rows, int num_of_col){
    if(row_index < num_of_rows && col_index < num_of_col && num_of_rows > 0 && num_of_col > 0 && row_index >= 0 && col_index >= 0){
        int pos = size*row_index*num_of_col + size*col_index;
        memcpy(&array[pos], element, size);
        return 1;
    }
    else{
        printf("out of array bounds\n");
        return 0;
    }
}
```

The function two_d_store_general stores an arbitrary type in a 2D row major array. The element will be stored in the character array if the row and column index, that the element will be position in, is inside the array boundaries. The position is obtained by multiplying the size of the element times the row position and the number of columns and then adding the element size times the column position. Then the element is placed in the array. If the position of the row and/ or column isn't in the array boundaries,

```
68  void *two_d_fetch_general(char* array, int size, int row_index, int col_index, int num_of_rows, int num_of_col){
69      if(row_index < num_of_rows && col_index < num_of_col && num_of_rows > 0 && num_of_col > 0 && row_index >= 0 && col_index >= 0){
70          int pos = size*row_index*num_of_col + size*col_index;
71          return &array[pos];
72      }
73      else{
74          printf("out of array bounds\n");
75              return NULL;
76      }
77  }
```

```
80  int two_d_store_row(char* array, int value, int size, int row_index, int col_index, int num_of_rows, int num_of_col){
81      if(row_index < num_of_rows && col_index < num_of_col && num_of_rows > 0 && num_of_col > 0 && row_index >= 0 && col_index >= 0){
82          int pos = size*row_index*num_of_col + size*col_index;
83          memcpy(&array[pos], &value, size);
84          return 1;
85      }
86      else{
87          printf("out of array bounds\n");
88          return 0;
89      }
90  }
```

This function inserts integers into an array at a specific row and column. It uses values like num_of_rows to check that that the arguments aren't out of array bounds. If all parameters are correct the function calculates a position on row-major form and inserts the value into the array using memcpy. The function returns a 1 or a 0 depending if it is successful or not.

```
93  int two_d_fetch_row(char* array, int size, int row_index, int col_index, int num_of_rows, int num_of_col){
94      if(row_index < num_of_rows && col_index < num_of_col && num_of_rows > 0 && num_of_col > 0 && row_index >= 0 && col_index >= 0){
95          return array[size*row_index*num_of_col + size*col_index];
96      }
97      else{
98          printf("out of array bounds\n");
99          return 0;
100     }
101 }
```

This function uses the same arguments as the store function to check if the values are valid for the array. If they are valid, the array returns the value in given row and column from the arguments. The index of the value is given by the same function as in two_d_store_row. Otherwise the functions prints that it's out of bounds and returns a 0 which is not a good solution since 0 could be a value inside the array. Could have improved the function by returning a integer that would have been less likely used like INT_MAX.

Method of verification

To test that the functions worked the values in each pos in the array was fetched using the created function and then printed for every element inside the array. In the main program a 4 x 4 matrix was created and filled with the numbers 0 – 15. The printed values in the picture below shows that the values were the same. This test was done to all different versions of the store and fetch functions but with different values depending on which function.

```
viktor@thinkpad:~/Documents/Datorteknik/datorteknik-lab-3$ ./task_1
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
```

Task 2

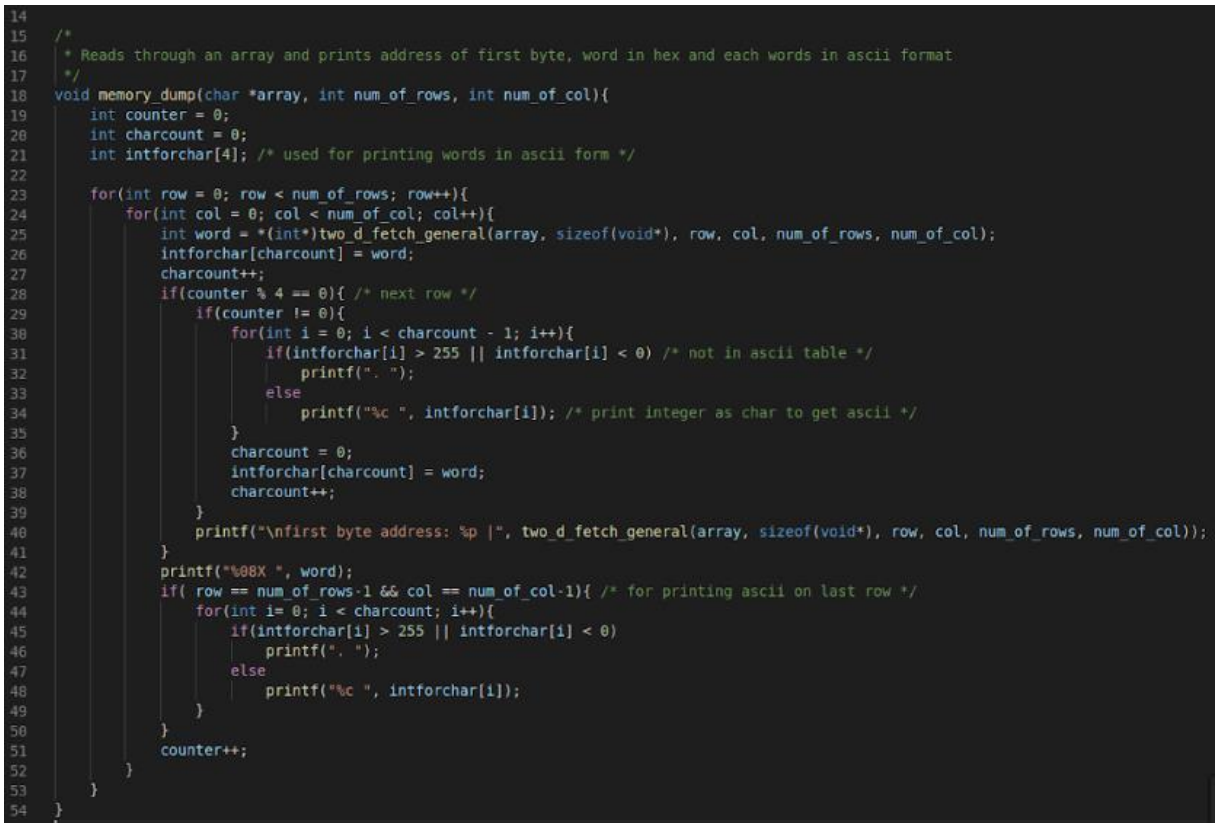
The objective of task 2 was to create a function that could go through the content of an array from task_1 and print various information in different formats.

Code

```c
#include <stdio.h>
#include "functions.c"

int main(){
    static int rows = 4;
    static int columns = 4;
    char *array = two_d_alloc(rows, columns, sizeof(void*));
    if(array != NULL){
        int counter = 65;
        for(int row = 0; row < rows; row++){
            for(int col = 0; col < columns; col++){
                if(row == 2 && col == 0){
                    int temp = 300;
                    void *pointer;
                    pointer = &temp;
                    two_d_store_general(array, pointer, sizeof(void*), row, col, rows, columns);
                }
                else{
                    void *pointer;
                    pointer = &counter;
                    two_d_store_general(array, pointer, sizeof(void*), row, col, rows, columns);
                    counter += 3;
                }
            }
        }
        memory_dump(array, rows, columns);
    }
    two_d_dealloc(array);
    printf("\n");
    return 0;
}
```

The main part of the program designed to test the memory dump function inserts integers into an array which is allocated and then deallocated at the end using functions from task_1. The function goes throug each pos inside the 2d array to store values. The function two_d_store_general was used to store the integers into the array. At row 3 column 1 the integer 300 is inserted to test that the function could handle the ascii part when there is a number outside of 0 – 255.



```
14
15  /*
16   * Reads through an array and prints address of first byte, word in hex and each words in ascii format
17   */
18  void memory_dump(char *array, int num_of_rows, int num_of_col){
19      int counter = 0;
20      int charcount = 0;
21      int intforchar[4]; /* used for printing words in ascii form */
22
23      for(int row = 0; row < num_of_rows; row++){
24          for(int col = 0; col < num_of_col; col++){
25              int word = *(int*)two_d_fetch_general(array, sizeof(void*), row, col, num_of_rows, num_of_col);
26              intforchar[charcount] = word;
27              charcount++;
28              if(counter % 4 == 0){ /* next row */
29                  if(counter != 0){
30                      for(int i = 0; i < charcount - 1; i++){
31                          if(intforchar[i] > 255 || intforchar[i] < 0) /* not in ascii table */
32                              printf(". ");
33                          else
34                              printf("%c ", intforchar[i]); /* print integer as char to get ascii */
35                      }
36                      charcount = 0;
37                      intforchar[charcount] = word;
38                      charcount++;
39                  }
40                  printf("\nfirst byte address: %p |", two_d_fetch_general(array, sizeof(void*), row, col, num_of_rows, num_of_col));
41              }
42              printf("%08X ", word);
43              if( row == num_of_rows-1 && col == num_of_col-1){ /* for printing ascii on last row */
44                  for(int i= 0; i < charcount; i++){
45                      if(intforchar[i] > 255 || intforchar[i] < 0)
46                          printf(". ");
47                      else
48                          printf("%c ", intforchar[i]);
49                  }
50              }
51              counter++;
52          }
53      }
54  }
```

The memory_dump function takes in 3 arguments, an array, the number of columns and the number of rows. When it starts to go through each row and column, the first thing it does is to use two_d_fetch_general from task 1 to get the value at current row and column. To make sure that 4 words are printed per row the code on row 28 checks if a number is divisible by 4, and if it is a new row is printed with the address to the first byte. Every 4 words are saved inside a int array to be used when each word is printed in ascii format. The extra code at row 43 is used to print the ascii part on the last row.

Method of verification

The test that was done to the memory dump was to make it read integers starting at 65 (Hexadecimal: 00000041. ASCII: A) and adding three to every number after that. One of the numbers was changed to 300, to make sure that the ASCII part behaved as expected and could handle numbers outside of 0 - 255.