Lab 3: Memory Organization

Anton Söderlund och Marcus Nyström
20181012

Task 1

We started programming in C11 on an intel processor for convenience reasons, and then transferred our code to qemu with PuTTy. This created some differences between the output, for example *char* being signed vs. unsigned. The compiler on the raspbian emulator also uses C99 and we had to make some changes there as well.

1.1

Function two_d_alloc allocates memory for an array given the three arguments row, column, element size.

```
char* two_d_alloc(int n, int m, int size) {
    return malloc(n*m*size);
}
```

1.2

Function two_d_dealloc deallocates the memory using *free*, which deallocates the memory of the pointer and effectively drops the array.

```
int two_d_dealloc(char* array) {
    printf("\nFree address = %p", array);
    free(array);
    return 0;
}
```

1.3

Function two_d_store takes the address of the array and the rest of arguments as integers. This version can only store unsigned integers (0-255) due to the array being of type *char*. We solved this problem by using *memcpy* which copies the full-length integer value to the destination at given array index.

```
void two_d_store(char* array, int size, int row, int col, int maxRow, int maxCol, int val) {
    char* newArray = array + size*(maxCol*row+col);
    memcpy(newArray, &val, size);
}
```

1.4

Function two_d_fetch returns the integer at a given row and column number. We use *memcpy* to copy the full-length integer from the array.

```
int two_d_fetch(char* array, int size, int row, int col, int maxRow, int
maxCol) {
    char* newArray = array + size*(maxCol*row+col);
    int value;
    memcpy(&value, newArray, size);
    return value;
}
```

1.5

We tested with a 10x20 matrix and with numbers 2,000,000,000 to 2,000,000,199 and after we started using memcpy it worked as intended.

1.6

Our tests showed that we can represent numbers between -2,147,483,648 and 2,147,483,647. This is a signed 32-bit integer. It makes sense because it is 4 bytes.

1.7

We implemented two additional functions that uses column-major format, the only difference is that we change maxCol*row+col to maxRow*col+row.

```
void two_d_store_colMaj(char* array, int size, int row, int col,
int maxRow, int maxCol, int val) {
    char* newArray = array + size*(maxRow*col+row);
    memcpy(newArray, &val, size);
}

int two_d_fetch_colMaj(char* array, int size, int row, int col,
int maxRow, int maxCol) {
    char* newArray = array + size*(maxRow*col+row);
    int value;
    memcpy(&value, newArray, size);
    return value;
}
```

1.8

two_d_store_mem and two_d_fetch_mem uses a pointer to the value you want to store or fetch. We then use memcpy to store the value into the array. The fetch function returns a pointer to the value in the array. This implementation lead to using memcpy in all the previous functions, as some of the instructions were unclear.

```
void two_d_store_mem(char* array, int size, int row, int col,
int maxRow, int maxCol, int* val) {
    char* newArray = array + size*(maxCol*row+col);
    memcpy(newArray, val, size);
}

int* two_d_fetch_mem(char* array, int size, int row, int col,
int maxRow, int maxCol) {
    char* newArray = array + size*(maxCol*row+col);
    int* returnValPtr = (int*)newArray;
    return returnValPtr;
}
```

1.9

gen_two_d_store and gen_two_d_fetch are generalized functions of previous store and fetch. The functions uses pointers of type void and is dependant on the size (amount of bytes).

```
void gen_two_d_store(char* array, int size, int row, int col, int maxRow,
int maxCol, void* val) {
    char* newArray = array + size*(maxCol*row+col);
    memcpy(newArray, val, size);
}

void* gen_two_d_fetch(void* array, int size, int row, int col, int maxRow,
int maxCol) {
    void* newArray = array + size*(maxCol*row+col);
    return newArray;
}
```

See full code in appendix 1.

Task 2

Our memory dump function takes two arguments, the array and the size of the array (in bytes). We divide the size by four in the loop to divide the output into four bytes per word. The memory address of the first byte is printed on every line, followed by the four words in hexadecimal notation with fixed-length of 8 characters.

```
void memDump(char* array, int arraySize) {
    int i;
    for (i = 0; i < arraySize/4; i++) {
        char* newArray = array + 4*i;
        int out;
        memcpy(&out, newArray, 4);
        if(i%4==0) {
            printf("\n%p: ",newArray);
        }
        printf("%08x ",out);
    }
}
```

Task 3

We built a struct that holds a value and a pointer, to function as a node in the linked list.

```
typedef struct node {
    int value;
    struct node* next;
} node_t;
```

We then use two_d_alloc to allocate memory for the 1x10 array and gen_two_d_store to store the integers. We verify the implementation by printing out the size of a node, the address to the allocated array, and all the numbers in the linked list, retrieved using gen_two_d_fetch.

Then we run the memory dump on our array. Since each word is 4 bytes the output of memdump corresponds perfectly to our value-pointer pairs, of which we get two per line. As can be seen in Figure 1, the last pointer on a line (except the last line which points to NULL) points to the next node which is the same address that memdump prints due to that being the first byte in that word.

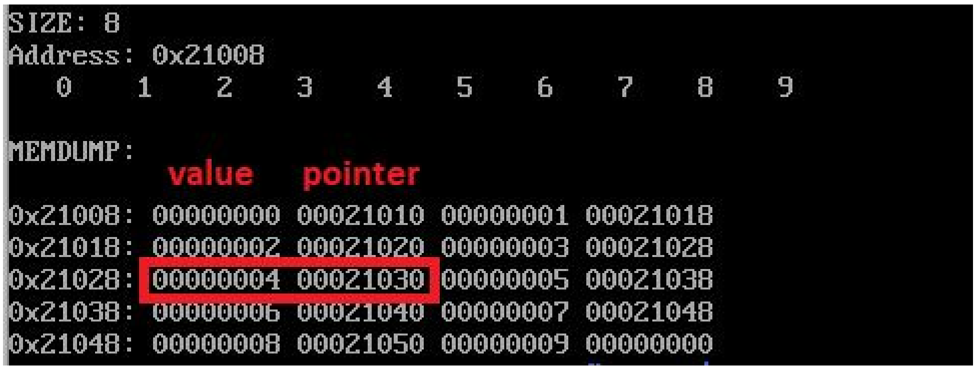

Figure 1: Memdump of the linked list with the fifth node identified.

Appendices

Appendix 1: Task 1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/**
* Allocates memory for an N by M matrix of a given size
*/
char* two_d_alloc(int n, int m, int size) {
    return malloc(n*m*size);
}
/**
* Deallocates memory for the given char* array
*/
int two_d_dealloc(char* array) {
    printf("\nFree address = %p", array);
    free(array);
    return 0;
}
/**
* Store an integer value in row-major format at a given row and column
index.
*/
void two_d_store(char* array, int size, int row, int col, int maxRow, int
maxCol, int val) {
    char* newArray = array + size*(maxCol*row+col);
    memcpy(newArray, &val, size);
}
/**
* Fetch an integer value in an array with row-major format at a given row *
* and column index.
*/
int two_d_fetch(char* array, int size, int row, int col, int maxRow, int
maxCol) {
    char* newArray = array + size*(maxCol*row+col);
    int value;
    memcpy(&value, newArray, size);
    return value;
}
/**
* Store an integer value in column-major format at a given row and index.
/*
void two_d_store_colMaj(char* array, int size, int row, int col, int maxRow,
int maxCol, int val) {
    char* newArray = array + size*(maxRow*col+row);
    memcpy(newArray, &val, size);
}

/**
```

```
* Fetch an integer value in column-major form at a given row and column
index.
*/
int two_d_fetch_colMaj(char* array, int size, int row, int col, int maxRow,
int maxCol) {
    char* newArray = array + size*(maxRow*col+row);
    int value;
    memcpy(&value, newArray, size);
    return value;
}

/**
* Store an integer value in row-major format at a given row and column *
* index.
* Takes the memory address to the value as an argument.
*/
void two_d_store_mem(char* array, int size, int row, int col, int maxRow,
int maxCol, int* val) {
    char* newArray = array + size*(maxCol*row+col);
    memcpy(newArray, val, size);
}
/**
* Fetch an integer value in an array with row-major format at a given row *
* and column index.
* The char* pointer is recast to an int pointer and returned.
*/
int* two_d_fetch_mem(char* array, int size, int row, int col, int maxRow,
int maxCol) {
    char* newArray = array + size*(maxCol*row+col);
    int* returnValPtr = (int*)newArray;
    return returnValPtr;
}
/**
* Store an  value in row-major format at a given row and column *     *
index.
* Takes an void* pointer as value to accommodate for arbitray data types
*/
void gen_two_d_store(char* array, int size, int row, int col, int maxRow,
int maxCol, void* val) {
    char* newArray = array + size*(maxCol*row+col);
    memcpy(newArray, val, size);
}
/**
* Fetch an integer value in an array with row-major format at a given row
* and column index.
* Returns a type void* to accommodate for arbitrary data types.
*/
void* gen_two_d_fetch(void* array, int size, int row, int col, int maxRow,
int maxCol) {
    void* newArray = array + size*(maxCol*row+col);
    return newArray;
```

```
}
/**
*   Main function
*/
int main(int argc, char** argv) {
    int row = 4;
    int col = 5;
    int i = 0;
    int j = 0;

    char* array = two_d_alloc(row,col,sizeof(int));
    printf("\nAddress = %p\n",array);
    /**
    *   This for-loop sets up the array with values using the first version
    *   of store and fetch.
    */
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++) {
            two_d_store(array, sizeof(int),i,j,row,col,i*col+j+2000000000);
        }
    }
    /**
    *   This for-loop prints the values of the array as a visual matrix
    */
    for (i = 0; i < row; i++) {
        for(j = 0; j < col; j++) {
            printf("%4d ",two_d_fetch(array,sizeof(int),i,j,row,col));
        }
        printf("\n");
    }
    two_d_dealloc(array);

    char* array1 = two_d_alloc(row,col,sizeof(int));
    printf("\nAddress = %p\n",array1);
    /**
    *   This for-loop sets up the array with values using the pointer
    *   version of store and fetch.
    */
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++) {
            int newVal = i*col+j+2000000000;
            two_d_store_mem(array1, sizeof(int),i,j,row,col,&newVal);
        }
    }
    /**
    *   This for-loop prints the values of the array as a visual matrix
    */
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++) {
            printf("%4d
",*(two_d_fetch_mem(array1,sizeof(int),i,j,row,col)));
```

```
        }
        printf("\n");
    }
    two_d_dealloc(array1);

    double valType;
    char* array2 = two_d_alloc(row,col,sizeof(valType));
    /**
    *   This for-loop sets up the array with values using the general
    *   version of store and fetch, in this case using type double.
    */
    for(i=0; i < row; i++) {
        for(j = 0; j < col; j++) {
            double* valPtr;
            *(valPtr) = i*col+j+2000000000;
            gen_two_d_store(array2,sizeof(valType),i,j,row,col,valPtr);
        }
    }
    printf("\nAddress: %p\n",array2);
    /**
    *   This for-loop prints the values of the array as a visual matrix
    */
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++ ) {
            printf("%4f
",*((double*)gen_two_d_fetch(array2,sizeof(valType),i,j,row,col)));
        }
        printf("\n");
    }
    two_d_dealloc(array2);
    return (EXIT_SUCCESS);
```

Appendix 2: Task 2

```
/**
* Takes an char* array and its size in bytes as arguments.
* Prints the array in 4 bytes per word, 4 words per line.
* The memory address of the first byte is printed first on every line
*/
void memDump(char* array, int arraySize) {
    int i;
    for (i = 0; i < arraySize/4; i++) {
        char* newArray = array + 4*i;
        int out;
        memcpy(&out, newArray, 4);
        if(i%4==0) {
            printf("\n%p: ",newArray);
        }
        printf("%08x ",out);
    }
}

int main(int argc, char** argv) {
    int row = 4;
    int col = 5;
    int i = 0;
    int j = 0;
    char* array1 = two_d_alloc(row,col,sizeof(int));
    printf("\nAddress = %p\n",array1);

    /**
    *   This for-loop sets up the array with values
    */
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++) {
            int newVal = i*col+j+2000000000;
            two_d_store_mem(array1, sizeof(int),i,j,row,col,&newVal);
        }
    }
    /**
    *   This for-loop prints the values of the array as a visual matrix
    */
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++) {
            printf("%4d ",
*(two_d_fetch_mem(array1,sizeof(int),i,j,row,col)));
        }
        printf("\n");
    }
    /**
    *   Memdump of array1
    */
    memDump(array1,row*col*sizeof(int));
    two_d_dealloc(array1);
```

Appendix 3: Task 3

```
typedef struct node {
    int value;
    struct node* next;
} node_t;

int main(int argc, char** argv) {
    int i = 0;
    int j = 0;

    row = 1;
    col = 10;
    node_t valType;
    printf("\nSIZE: %lu",sizeof(node_t));
    char* array2 = two_d_alloc(row,col,sizeof(valType));
    /**
    *   This for-loop sets up the array with values
    */
    for(i=0; i < row; i++) {
        for(j = 0; j < col; j++) {
            node_t element;
            if(col*i+j == col*(row-1)+(col-1)) {
                element.next = NULL;
            } else {
                element.next = (node_t*)(array2 +
sizeof(valType)*(col*i+j+1));
            }
            element.value = i*col+j;
            gen_two_d_store(array2,sizeof(valType),i,j,row,col,&element);
        }
    }
    printf("\nAddress: %p\n",array2);
    /**
    *   This for-loop prints the values of the array as a visual matrix
    */
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++ ) {
            printf("%4d ",
((node_t*)gen_two_d_fetch(array2,sizeof(valType),i,j,row,col))->value);
        }
        printf("\n");
    }

    printf("\nMEMDUMP:\n");
    /**
    *   Memdump of array2 (the linked list)
    */
    memDump(array2,row*col*sizeof(valType));
    two_d_dealloc(array2);
```