Simon Johansson & Zacharias Jöbo

Lab 3: Memory Organization

Our source code for this lab can be found at
https://drive.google.com/drive/folders/1prhKsFP8fQSO5FzhvqxvRQoZL9YD3zZY?usp=sharing

Task 1: Array Storage (15 points)

1) The function **two_d_alloc** starts at the line 11 and uses the struct **two_d_array** that can be found at line 4.
2) Function **two_d_dealloc** (line 147) frees the memory of the array buffer and sets everything to 0.
3) The test code for the functions **two_d_alloc** and **two_d_dealloc**can be found in the main body at line 157 and 167. The function is tested with a 8x2 array.
4) **two_d_store** is implemented at the line 19 **two_d_store** is in row-major order
5) **two_d_fetch** can be found at line 38 in the implementation.
6) Test code for **two_d_fetch** is at line 160-166.
7) The if-statments at line 24-25 for **two_d_store** and line 44-45 for **two_d_fetch** checks if the row and columns is inside of bounds. Therefore it only stores and reads bytes inside the buffer . 105-106 127-128
8) The column major implementation is called **two_d_store_colMajor** (line 59) and **two_d_fetch_colMajor** (line 78). Lines 64-65 and 84-85 checks the boundaries inside the buffer. The test code for the column major can be found at line 169-178.
9-10) It was very unclear what we were supposed to do in task 1.9. We interpreted it as to read The implementation of the general functions (using row major order) is called **two_d_store_generic** (line 100) and **two_d_fetch_generic** (line 119). In the fetch function it sets the return address to 0 if it attempts to read bytes outside the boundaries of the buffer. The test implementation is at line 180-209

Result of the implementation for Task 1:

```
pi@raspberrypi:~ $ ./task1

RowMajor>> Numbers are: 7, 1713, 6212, size of int: 4

ColMajor>> Numbers are: 7, 1713, 6212, size of int: 4

RowMajor>> Stored doubles: 42.360000, 13.370000, Size of double: 8

RowMajor>> Stored array: 432, 6535, 1337, 42
```

Implementation of Task 1:

```
1  . #include <stdlib.h>
2  . #include <stdio.h>
3  .
4  . struct two_d_array{
```

```
5  .    int n;
6  .    int m;
7  .    char* buffer;
8  . };
9  .
10 .
11 . struct two_d_array two_d_alloc(int n, int m){
12 .    struct two_d_array new_array;
13 .    new_array.n = n;
14 .    new_array.m = m;
15 .    new_array.buffer = malloc(sizeof(char)*n*m);
16 .    return new_array;
17 . }
18 .
19 . void two_d_store(struct two_d_array* array, int elementSize, int number, int row, int col){
20 .    int x = col;
21 .    int y = row;
22 .    int i;
23 .    for(i = 0; i < elementSize; i++){
24 .            if(x >= 0 && x < array->m){
25 .                    if(y >= 0 && y < array->n){
26 .            *(array->buffer+x+y*array->m) = (number >> (8*i));
27 .                    }
28 .            }
29 .
30 .            x++;
31 .            if(x >= array->m){
32 .                    x = 0;
33 .                    y++;
34 .            }
35 .    }
36 . }
37 .
38 . int two_d_fetch(struct two_d_array* array, int elementSize, int row, int col){
39 .    int x = col;
40 .    int y = row;
41 .    int number = 0x00000000;
42 .    int i;
43 .    for(i = 0 ; i < elementSize ; i++) {
44 .            if(x >= 0 && x < array->m) {
45 .                    if(y >= 0 && y < array->n){
46 .    number |= ((*(array->buffer+x+y*array->m)) << (8*i) & (0xff << (8*i)));
47 .                    }
48 .            }
49 .            x++;
50 .            if(x >= array->m){
51 .                    x = 0;
52 .                    y++;
53 .            }
54 .    }
55 .    return number;
56 . }
57 .
58 .
59 . void two_d_store_colMajor(struct two_d_array* array, int elementSize, int number, int row, int
col){
60 .    int x = col;
61 .    int y = row;
62 .    int i;
63 .    for(i = 0; i < elementSize; i++){
```

```
64 .            if(x >= 0 && x < array->m){
65 .                    if(y >= 0 && y < array->n){
66 .            *(array->buffer+y+x*array->n) = (number >> (8*i));
67 .                    }
68 .            }
69 .
70 .            y++;
71 .            if(y >= array->n){
72 .                    y = 0;
73 .                    x++;
74 .            }
75 .    }
76 . }
77 .
78 . int two_d_fetch_colMajor(struct two_d_array* array, int elementSize, int row, int col){
79 .    int x = col;
80 .    int y = row;
81 .    int number = 0x00000000;
82 .    int i;
83 .    for(i = 0 ; i < elementSize ; i++) {
84 .            if(x >= 0 && x < array->m) {
85 .                    if(y >= 0 && y < array->n){
86 .    number |= ((*(array->buffer+y+x*array->n)) << (8*i) & (0xff << (8*i)));
87 .                    }
88 .            }
89 .            y++;
90 .            if(y >= array->n){
91 .                    y = 0;
92 .                    x++;
93 .            }
94 .    }
95 .    return number;
96 . }
97 .
98 .
99 .
100. void two_d_store_generic(struct two_d_array* array, int elementSize, char* number, int row, int
col){
101.    int x = col;
102.    int y = row;
103.    int i;
104.    for(i = 0; i < elementSize; i++){
105.            if(x >= 0 && x < array->m){
106.                    if(y >= 0 && y < array->n){
107.            *(array->buffer+x+y*array->m) = *(number + i);
108.                    }
109.            }
110.
111.            x++;
112.            if(x >= array->m){
113.                    x = 0;
114.                    y++;
115.            }
116.    }
117. }
118.
119. char* two_d_fetch_generic(struct two_d_array* array, int elementSize, int row, int col){
120.    int x = col;
121.    int y = row;
122.    int number = 0x00000000;
```

```
123.    int i;
124.    char* address = array->buffer+col+row*array->m;
125.    for(i = 0 ; i < elementSize ; i++) {
126.
127.            if(x >= 0 && x < array->m) {
128.                    if(y >= 0 && y < array->n){
129.                    }
130.                    else {
131.                            address = 0;
132.                    }
133.            }
134.            else {
135.                    address = 0;
136.            }
137.            x++;
138.            if(x >= array->m){
139.                    x = 0;
140.                    y++;
141.            }
142.    }
143.    return address;
144. }
145.
146.
147. void two_d_dealloc(struct two_d_array* array){
148.    free(array->buffer);
149.    array->buffer = 0;
150.    array->n = 0;
151.    array->m = 0;
152. }
153.
154. int main(){
155.
156.    struct two_d_array my_array;
157.    my_array = two_d_alloc(8, 2);
158.    int size = sizeof(int);
159.
160.    two_d_store(&my_array, size, 7, 0, 0);
161.    two_d_store(&my_array, size, 1713, 2, 0);
162.    two_d_store(&my_array, size, 6212, 4, 1);
163.    int num1 = two_d_fetch(&my_array, size, 0, 0);
164.    int num2 = two_d_fetch(&my_array, size, 2, 0);
165.    int num3 = two_d_fetch(&my_array, size, 4, 1);
166.    printf("\nRowMajor>> Numbers are: %d, %d, %d, size of int: %d\n", num1, num2, num3, size);
167.    two_d_dealloc(&my_array);
168.
169.    my_array = two_d_alloc(8, 2);
170.
171.    two_d_store_colMajor(&my_array, size, 7, 0, 0);
172.    two_d_store_colMajor(&my_array, size, 1713, 4, 0);
173.    two_d_store_colMajor(&my_array, size, 6212, 1, 1);
174.    num1 = two_d_fetch_colMajor(&my_array, size, 0, 0);
175.    num2 = two_d_fetch_colMajor(&my_array, size, 4, 0);
176.    num3 = two_d_fetch_colMajor(&my_array, size, 1, 1);
177.    printf("\nColMajor>> Numbers are: %d, %d, %d, size of int: %d\n", num1, num2, num3, size);
178.    two_d_dealloc(&my_array);
179.
180.    my_array = two_d_alloc(8, 2);
181.    double number = 42.36;
182.    two_d_store_generic(&my_array, sizeof(double), (char*) &number, 0, 0);
```

```
183.     number = 13.37;
184.     two_d_store_generic(&my_array, sizeof(double), (char*) &number, 4, 0);
185.     char* address = two_d_fetch_generic(&my_array, sizeof(double), 0, 0);
186.     char* address2 = two_d_fetch_generic(&my_array, sizeof(double), 4, 0);
187.
188.     if(address != 0 && address2 != 0){
189.             double d = *((double*) address);
190.             double d2 = *((double*) address2);
191.             printf("\nRowMajor>> Stored doubles: %f, %f, Size of double: %d\n", d, d2,
sizeof(double));
192.     }
193.     two_d_dealloc(&my_array);
194.
195.     my_array = two_d_alloc(8, 2);
196.     int intArray[4] = {432, 6535, 1337, 42};
197.     //intArray[0] = 432;
198.     two_d_store_generic(&my_array, sizeof(int)*4, (char*) intArray, 0, 0);
199.     int* intArray2 = (int*)two_d_fetch_generic(&my_array, sizeof(int), 0, 0);
200.     if(intArray2 != 0) {
201.             int i1 = intArray2[0];
202.             int i2 = intArray2[1];
203.             int i3 = intArray2[2];
204.             int i4 = intArray2[3];
205.             printf("\nRowMajor>> Stored array: %d, %d, %d, %d\n", i1, i2, i3, i4);
206.     }
207.
208.     two_d_dealloc(&my_array);
209. }
210.
```

Task 2: Memory Dump (5 points)

Implementation of the memory dump is located at line 155. Function **memoryDump** arguments is a char pointer for the buffer and a integer for the number of bytes.

Result of the implementation for Task 2:

The array with the set {432, 6535, 1337, 42, 54, 2, 6, 1, 2, 5, 8712, 91} is shown below in the memory dump in hexadecimal with its corresponding address. The address increments with 16 bytes each line, which is expected since it displays four words(4 bytes) on each line.

```
pi@raspberrypi:~ $ ./task2

RowMajor>> Stored array: 432, 6535, 1337, 42, 54, 2, 6, 1, 2, 5, 8712, 91,

Memory Dump:
Address: 0x22008 >> 0x000001b0 0x00001987 0x00000539 0x0000002a
Address: 0x22018 >> 0x00000036 0x00000002 0x00000006 0x00000001
Address: 0x22028 >> 0x00000002 0x00000005 0x00002208 0x0000005b
```

Implementation of Task 2:

```
.....
155 . void memoryDump(char* buffer, int numBytes){
156 .   printf("\nMemory Dump:");
157 .   int i;
158 .   int number = 0;
159 .   for(i = 0; i < numBytes/4; i++){
160 .           number = *(((int*) buffer) + i);
161 .           int* address = (((int*) buffer) + i);
162 .           if(i % 4 == 0){
163 .                   printf("\nAddress: %p >> ", address);
164 .           }
165 .           printf("%#010x ", number);
166 .   }
167 .   printf("\n");
168 . }
169 .
170 . int main(){
171 .
172 .   struct two_d_array my_array;
173 .   my_array = two_d_alloc(24, 2);
174 .   int intArray[12] = {432, 6535, 1337, 42, 54, 2, 6, 1, 2, 5, 8712, 91};
175 .   two_d_store_generic(&my_array, sizeof(int)*12, (char*) intArray, 0, 0);
176 .   int* intArray2 = (int*)two_d_fetch_generic(&my_array, sizeof(int), 0, 0);
177 .   if(intArray2 != 0) {
178 .           printf("\nRowMajor>> Stored array:");
179 .           int i;
180 .           for(i = 0; i < 12; i++){
181 .                   printf(" %d,", intArray[i]);
182 .           }
183 .           printf("\n");
184 .   }
185 .   memoryDump(my_array.buffer, my_array.n * my_array.m);
186 .   two_d_dealloc(&my_array);
187 . }
```

Task 3: Linked Lists (5 points)

Implemented a pop and push function for the linked list since we needed to handle the memory allocation by our self. We went with the convention that elements with all ones(0xFFFFFFFF 0xFFFFFFFF) is a unused memory in the array. When we pushed we search for a free block of memory in our array and returns the address. The pop function works the same way but backwards, the memory that we free we will with all ones. Push and pop works as linked list (FIFO). Each node element consist of one integer and one node pointer. In the image below you can see that one node takes up two words. One for the integer and one for the pointer.

Result of the implementation for Task 3:

The marked (with red) is the 5th element in the linked list.

```
pi@raspberrypi:~ $ ./task3

Memory Dump:
Address: 0x22008 >> 0000000000 0x00022010 0x00000001 0x00022018
Address: 0x22018 >> 0x00000002 0x00022020 0x00000003 0x00022028
Address: 0x22028 >> 0x00000004 0x00022030 0x00000005 0x00022038
Address: 0x22038 >> 0x00000006 0x00022040 0x00000007 0x00022048
Address: 0x22048 >> 0x00000008 0x00022050 0x00000009 0000000000
```

```
pi@raspberrypi:~ $ ./task3
Pop> 0
Pop> 1
Pop> 2
Pop> 3
Pop> 4

Memory Dump:
Address: 0x22008 >> 0x0000000a 0x00022010 0x0000000b 0x00022018
Address: 0x22018 >> 0x0000000c 0x00022020 0x0000000d 0x00022028
Address: 0x22028 >> 0x0000000e 0000000000 0x00000005 0x00022038
Address: 0x22038 >> 0x00000006 0x00022040 0x00000007 0x00022048
Address: 0x22048 >> 0x00000008 0x00022050 0x00000009 0x00022008
```

Implementation of Task 3:

```
....
12  . typedef struct n{
13  .   int value;
14  .   struct n* next;
15  . }node;
16  .
....[CODE FROM PREVIOUS IMPLEMENTATION]
175 . char* nextAddress(struct two_d_array* my_array){
176 .   int x, y;
```

```
177 .   int i;
178 .   for(y = 0; y < my_array->n; y++){
179 .           for(x = 0; x < my_array->m; x++){
180 .                   char* temp = two_d_fetch_generic(my_array, sizeof(node), y, x);
181 .                   if(temp == 0)continue;
182 .                   char valid = 0;
183 .                   for(i = 0; i < sizeof(node); i++){
184 .                           if(*(temp+i) != 0xff){//check if cell is occupied(is not all ones)
185 .                                   valid = 1;
186 .                                   break;
187 .                           }
188 .                   }
189 .                   if(valid == 0){
190 .                           return temp;
191 .                   }
192 .           }
193 .   }
194 .   return 0;
195 . }
196 .
197 . void push(node* head, int value, struct two_d_array* array){
198 .   char* address = nextAddress(array);
199 .   if(address == 0)return;
200 .   //Push part here.
201 .   node* current = head;
202 .
203 .   while(current->next != 0){
204 .           current = current->next;
205 .   }
206 .
207 .   current->next = (node*) address;
208 .
209 .   //Store part here
210 .   int offset = ((address - array->buffer));
211 .   int row = offset / array->m;
212 .   int col = offset % array->m;
213 .   node newNode;
214 .   newNode.value = value;
215 .   newNode.next = 0;
216 .   two_d_store_generic(array, sizeof(node), (char*) &newNode, row, col);
217 .   //Store part done
218 .
219 .
220 . }
221 .
222 . void freeNode(char* address, struct two_d_array* array){
223 .   int offset = ((address - array->buffer));
224 .   int row = offset / array->m;
225 .   int col = offset % array->m;
226 .   node ones;
227 .   ones.value = 0xFFFFFFFF;
228 .   ones.next = (node*) 0xFFFFFFFF;
229 .   two_d_store_generic(array, sizeof(node), (char*) &ones, row, col);
230 . }
231 .
232 . int pop(node** head, struct two_d_array* array){
233 .   if(*head == 0){
234 .           return 0;
235 .   }
236 .   int retVal = (*head)->value;
```

```
237 .   node* nextNode = (*head)->next;
238 .   freeNode((char*) *head, array);
239 .   *head = nextNode;
240 .   return retVal;
241 . }
242 .
243 . void fillOnes(struct two_d_array* array){
244 .   int x, y;
245 .   int ones = 0xff;
246 .   for(y = 0; y < array->n; y++){
247 .           for(x = 0; x < array->m; x++){
248 .                   two_d_store_generic(array, 1, (char*) &ones, y, x);
249 .           }
250 .   }
251 . }
252 .
253 . int main(){
254 .
255 .   struct two_d_array my_array;
256 .   int node_size = sizeof(node);
257 .   my_array = two_d_alloc(2, 5*node_size);
258 .   //fill buffer with ones to indicate that memory is availiable.
259 .   fillOnes(&my_array);
260 .   node head;
261 .   head.next = 0;
262 .   node n0;
263 .   push(&head, 0, &my_array);
264 .   push(&head, 1, &my_array);
265 .   push(&head, 2, &my_array);
266 .   push(&head, 3, &my_array);
267 .   push(&head, 4, &my_array);
268 .   push(&head, 5, &my_array);
269 .   push(&head, 6, &my_array);
270 .   push(&head, 7, &my_array);
271 .   push(&head, 8, &my_array);
272 .   push(&head, 9, &my_array);
273 .   printf("Pop> %d\n", pop(&(head.next), &my_array));
274 .   printf("Pop> %d\n", pop(&(head.next), &my_array));
275 .   printf("Pop> %d\n", pop(&(head.next), &my_array));
276 .   printf("Pop> %d\n", pop(&(head.next), &my_array));
277 .   push(&head, 10, &my_array);
278 .   push(&head, 11, &my_array);
279 .   push(&head, 12, &my_array);
280 .   printf("Pop> %d\n", pop(&(head.next), &my_array));
281 .   push(&head, 13, &my_array);
282 .   push(&head, 14, &my_array);
283 .   push(&head, 15, &my_array);
284 .   push(&head, 16, &my_array);
285 .   push(&head, 17, &my_array);
286 .   push(&head, 18, &my_array);
287 .
288 .   memoryDump(my_array.buffer, my_array.n * my_array.m);
289 .   two_d_dealloc(&my_array);
290 . }
```