

---

**OPERATING SYSTEM LAB REPORT**

**LAB 6 : Virtual memory**

---

Benjamin DAVID & Robin VAN DESSEL  
Paris, on November 25, 2022

## 1. Step 2: virtual memory

### 1. Initialization of the memory

In the `initMem()` function, we initialise a `mem` variable that has one hole the size of the memory. The function returns a `mem` object.

```
1 mem_t initMem()
2 {
3     mem_t mem;
4     std::vector<hole_t> root;
5
6     // we create a the first hole at address 0 and with size of all the memory
7     hole_t Hole;
8     Hole.adr = 0;
9     Hole.sz = SIZE;
10    root.push_back(Hole);
11
12    mem.root = root;
13
14    return mem;
15 }
```

Listing 1..1: Initialization of the memory

### 2. Memory Allocation

The second part of the lab is the allocation of the memory. We implemented 3 memory algorithms.

- **First Fit**: allocate the first fit that is big enough
- **Best Fit** : allocate the smallest hole that is big enough
- **Worst Fit**: allocate the largest hole

Let's take this example: there are two holes in the memory. One has a size of 10 and another one has a size of 50.

We want to allocate something that has a size of 9.

With the **First Fit** method, the allocation algorithm will browse for the first hole that is big enough. In this example, the hole of size 10 will be allocated

With the **Best Fit**, the allocation algorithm will browse for the first hole that has the closet size of the allocation. In this example, the hole of size 10 will be allocated.

With the **Worst Fit**, the allocation algorithm will browse for the biggest hole. In this example, the hole of size 50 will be allocated.

If the new allocation is exactly the size of the hole, the hole is removed. Also, if there is any big enough hole, the allocate function returns -1.

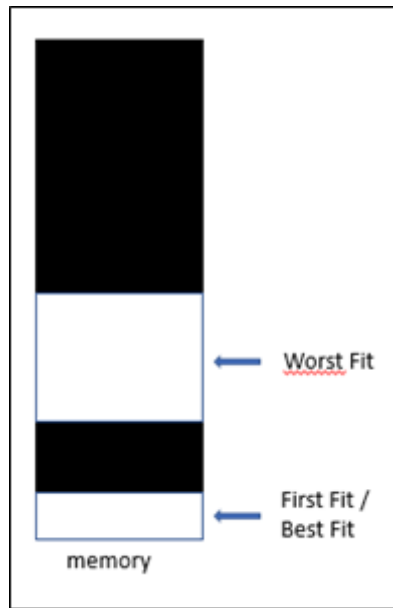


Fig. 1 : Memory allocation illustration

This is the code to allocate space :

```

1 // allocates space in bytes (byte_t) using First-Fit
2 address_t FirstFitAlloc(mem_t *mp, int sz)
3 {
4     for (int i = 0; i < mp->root.size(); i++)
5     {
6         // if the hole is big enough we decrease the size of the hole by the new size
7         if (sz < mp->root[i].sz)
8         {
9             mp->root[i].sz -= sz; // decrease the size of the hole
10            mp->root[i].adr += sz; // increase the address of the hole
11
12            return mp->root[i].adr - sz; // return the address of the alloc
13        }
14        // if the hole is exactly the size of the alloc, we remove the hole
15        else if (sz == mp->root[i].sz)
16        {
17            address_t ad = mp->root[i].adr;
18            mp->root.erase(mp->root.begin() + i); // delete the hole
19
20            return ad;
21        }
22    }
23    std::cout << "No more available space in the memory\n";
24    return -1; // return -1 if the alloc is not possible
25 };
26
27 // allocates space in bytes (byte_t) using Worst-Fit
28 address_t WorstFitAlloc(mem_t *mp, int sz)
29 {
30     int max = 0;
31     int j = 0;
32
33     // looks for the largest hole
34     for (int i = 0; i < mp->root.size(); i++)
35     {

```

```

36     if (max <= mp->root[i].sz)
37     {
38         max = mp->root[i].sz;
39         j = i;
40     }
41 }
42
43 // if the hole is big enough we decrease the size of the hole
44 if (sz < mp->root[j].sz)
45 {
46     mp->root[j].sz -= sz; // decrease the size of the hole
47     mp->root[j].adr += sz; // increase the address of the hole
48
49     return mp->root[j].adr - sz;
50 }
51 // if the hole is exactly the size of the alloc, we remove the hole
52 else if (sz == mp->root[j].sz)
53 {
54     address_t ad = mp->root[j].adr;
55
56     mp->root.erase(mp->root.begin() + j); // delete the hole
57
58     return ad;
59 }
60
61
62     std::cout << "No more available space in the memory\n";
63     return -1;
64 // return -1 if the alloc is not possible}
65 };
66
67 // allocates space in bytes (byte_t) using Best-Fit
68 address_t BestFitAlloc(mem_t *mp, int sz)
69 {
70     int min = SIZE;
71     int j = 0;
72
73     //looks for the hole that has the closest size to the allocation size
74     for (int i = 0; i < mp->root.size(); i++)
75     {
76
77         if (min >= mp->root[i].sz - sz && mp->root[i].sz - sz >= 0)
78         {
79             min = mp->root[i].sz - sz;
80             j = i;
81         }
82     }
83
84     // if the hole is big enough we decrease the size of the hole and increase the address
85     if (sz < mp->root[j].sz)
86     {
87         mp->root[j].sz -= sz;
88         mp->root[j].adr += sz;
89
90         return mp->root[j].adr - sz;
91     }
92     // if the hole is exactly the size of the alloc, we remove the hole
93     else if (sz == mp->root[j].sz)
94     {

```

```

95     address_t ad = mp->root[j].adr;
96
97     mp->root.erase(mp->root.begin() + j); // delete the hole
98
99     return ad;
100 }
101
102     std::cout << "No more available space in the memory\n";
103     return -1;
104 // return -1 if the alloc is not possible}
105 };

```

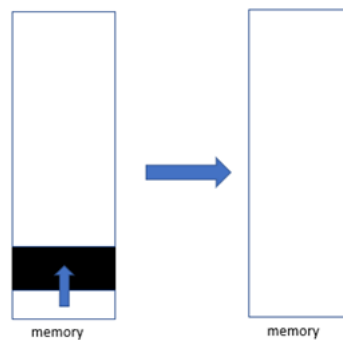
Listing 1..2: Code to allocate space

### 3. Free Memory

When we don't use an allocation anymore, we have to free the memory.

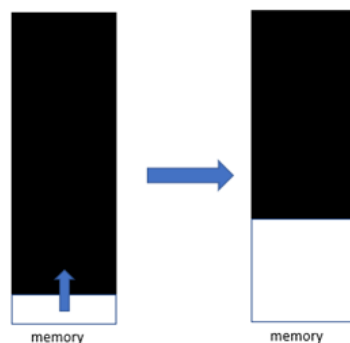
In order to free the memory, we browse for holes and check if they are followed or preceded by other holes.

If there is a hole above and below, we delete the following hole, then we increase the size of the previous hole with the size of the free allocation and the size of the following hole.



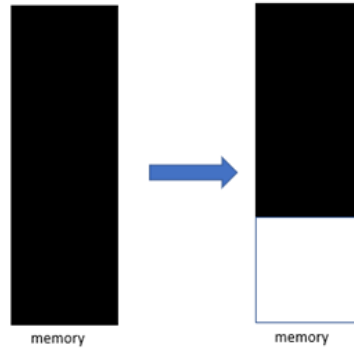
**Fig. 2 :** Free memory, a hole above and below illustration

If there is only a hole above or below, we increase the size of the hole with the size of the allocation.



**Fig. 3 :** Free memory, a hole above or below illustration

If there is no holes on both sides, we create a new hole.



**Fig. 4 :** Free memory, no holes on both sides illustration

This is the code to free allocation :

```
1 // release memory that has already been allocated previously
2 void myFree(mem_t *mp, address_t p, int sz)
3 {
4     // boolean to know if there is a hole on side
5     bool holeOnSides = false;
6
7     // browse the holes
8     for (int j = 0; j < mp->root.size(); j++)
9     {
10         // CASE 1 : There is a hole above and below
11         if (mp->root[j].adr == p + sz && (j != 0 && (mp->root[j - 1].adr + mp->root[j - 1].sz) == p))
12         {
13             std::cout << "Hole on both sides\n";
14             // we increase size of the previous hole with the size the size of the previous hole + the size of
15             mp->root[j - 1].sz = mp->root[j - 1].sz + mp->root[j].sz + sz; // new size of the hole
16             mp->root.erase(mp->root.begin() + j); // erase the next hole
17
18             holeOnSides = true;
19
20             j = mp->root.size();
21         }
22
23         // CASE 2 : There is a hole above but not below
24         else if (mp->root[j].adr == p + sz && (j == 0 || (mp->root[j - 1].adr + mp->root[j - 1].sz) != p))
25         {
26             std::cout << "Hole on the right but not on the left\n";
27             // we increase size of the previous hole with the size of the free alloc
28             mp->root[j].sz = mp->root[j].sz + sz; // new size of the hole
29             mp->root[j].adr -= sz; // new address of the hole
30
31             holeOnSides = true; // there is a hole on at least one side
32
33             j = mp->root.size();
34         }
35
36         // CASE 3 : There is a hole below but not above
37         else if (mp->root[j].adr != p + sz && (j != 0 && (mp->root[j - 1].adr + mp->root[j - 1].sz) == p))
38         {
39             std::cout << "Hole on the left but not on the right\n";
40             // we increase size of the previous hole with the size of the hole + the size of the free
41             mp->root[j].sz = mp->root[j].sz + sz; // new size of the hole
42
43             holeOnSides = true; // there is a hole on at least one side
44
45             // we end the loop
46             j = mp->root.size();
47         }
48     }
49
50     // if there is no hole on the sides
51     if (!holeOnSides)
52     {
53         // CASE 4 : There is an alloc on both sides
54         for (int j = 0; j < mp->root.size(); j++)
55         {
56             if (mp->root[j].adr != p + sz && (j == 0 || (mp->root[j - 1].adr + mp->root[j - 1].sz) != p))
57             {
```

```

58         std::cout << "Alloc on the left and on the right \n";
59         // we create a new hole at the current index
60         hole_t newHole;
61         newHole.adr = p;
62         newHole.sz = sz;
63         mp->root.insert(mp->root.begin() + j, newHole);
64         // we end the loop
65         j = mp->root.size();
66     }
67 }
68 }
69 };

```

Listing 1.3: Code to free allocation

## 4. Write and read

We write and read bytes on the mem array.

```

1 void myWrite(mem_t *mp, address_t p, byte_t val)
2 {
3     mp->mem[p] = val;
4 };
5
6 // read memory from a byte
7 byte_t myRead(mem_t *mp, address_t p)
8 {
9     return mp->mem[p];
10 };

```

Listing 1.4: Code to write and read bytes on the mem array

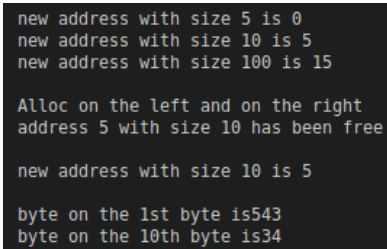


## 5. Execution

```
1 int main()
2 {
3     // initialization of the memory
4     mem_t tempMem = initMem();
5     mem_t *mem = &tempMem;
6
7     // allocation of 3 addresses
8     address_t adr1 = BestFitAlloc(mem, 5); // new address with size of 5 in the memory
9     address_t adr2 = BestFitAlloc(mem, 10); // new address with size of 10 in the memory
10    address_t adr3 = BestFitAlloc(mem, 100); // new address with size of 100 in the memory
11
12    std::cout << "new address with size 5 is " << adr1 << "\n";
13    std::cout << "new address with size 10 is " << adr2 << "\n";
14    std::cout << "new address with size 100 is " << adr3 << "\n";
15
16    myFree(mem, adr2, 10); // free address 2 with size of 10
17    // myFree(mem, adr1, 5); // free address 1 with size of 5
18
19    adr2 = BestFitAlloc(mem, 10);
20    std::cout << "new address with size 10 is " << adr2 << "\n";
21
22    myWrite(mem, adr3, 543); // write on the 1st byte
23    myWrite(mem, adr3 + 9, 34); // write on the 10th byte
24
25    byte_t val1 = myRead(mem, adr3); //
26    byte_t val2 = myRead(mem, adr3 + 9);
27 }
```

Listing 1..5: Main

Result:



```
new address with size 5 is 0
new address with size 10 is 5
new address with size 100 is 15

Alloc on the left and on the right
address 5 with size 10 has been free

new address with size 10 is 5

byte on the 1st byte is543
byte on the 10th byte is34
```

Fig. 5 : Result of the execution

## 6. Entire code

```
1 #ifndef __MMU__H__
2 #define __MMU__H__
3 #define SIZE 65536
4 #include <vector>
5 #include <iostream>
6 #include <iostream>
7 #include <stdlib.h>
8
9 typedef short byte_t;
10
11 typedef int address_t;
12
13 // structure of a hole : an address and a size
14 typedef struct hole
15 {
16     address_t adr;
17     int sz;
18 } hole_t;
19
20 // structure of a memory : a size and a vector of holes
21 typedef struct
22 {
23     byte_t mem[SIZE];
24     std::vector<hole_t> root;
25 } mem_t;
26
27 // Initialize memory
28 mem_t initMem()
29 {
30     mem_t mem;
31     std::vector<hole_t> root;
32
33     // we create a the first hole at address 0 and with size of all the memory
34     hole_t Hole;
35     Hole.adr = 0;
36     Hole.sz = SIZE;
37     root.push_back(Hole);
38
39     mem.root = root;
40
41     return mem;
42 }
43
44 // allocates space in bytes (byte_t) using First-Fit
45 address_t FirstFitAlloc(mem_t *mp, int sz)
46 {
47     for (int i = 0; i < mp->root.size(); i++)
48     {
49         // if the hole is big enough we decrease the size of the hole by the new size
50         if (sz < mp->root[i].sz)
51         {
52             mp->root[i].sz -= sz; // decrease the size of the hole
53             mp->root[i].adr += sz; // increase the address of the hole
54
55             return mp->root[i].adr - sz; // return the address of the alloc
56         }
57     }
58 }
```

```

57     }
58     // if the hole is exactly the size of the alloc, we remove the hole
59     else if (sz == mp->root[i].sz)
60     {
61         address_t ad = mp->root[i].adr; // put in a variable the address of the alloc
62
63         mp->root.erase(mp->root.begin() + i); // delete the hole
64
65         return ad; // return the address of the alloc
66     }
67 }
68 std::cout << "No more available space in the memory\n";
69 return -1; // return -1 if the alloc is not possible
70 };
71
72 // allocates space in bytes (byte_t) using Worst-Fit
73 address_t WorstFitAlloc(mem_t *mp, int sz)
74 {
75     int max = 0;
76     int j = 0;
77
78     // looks for the largest hole
79     for (int i = 0; i < mp->root.size(); i++)
80     {
81         if (max <= mp->root[i].sz)
82         {
83             max = mp->root[i].sz;
84             j = i;
85         }
86     }
87
88     // if the hole is big enough we decrease the size of the hole
89     if (sz < mp->root[j].sz)
90     {
91         mp->root[j].sz -= sz; // decrease the size of the hole
92         mp->root[j].adr += sz; // increase the address of the hole
93
94         return mp->root[j].adr - sz; // return the address of the alloc
95     }
96     // if the hole is exactly the size of the alloc, we remove the hole
97     else if (sz == mp->root[j].sz)
98     {
99         address_t ad = mp->root[j].adr; // put in a variable the address of the alloc
100
101         mp->root.erase(mp->root.begin() + j); // delete the hole
102
103         return ad; // return the address of the alloc
104     }
105
106     std::cout << "No more available space in the memory\n";
107     return -1;
108     // return -1 if the alloc is not possible}
109 };
110
111 // allocates space in bytes (byte_t) using Best-Fit
112 address_t BestFitAlloc(mem_t *mp, int sz)
113 {
114     int min = SIZE;

```

```

116     int j = 0;
117
118     //looks for the hole that has the closest size to the allocation size
119     for (int i = 0; i < mp->root.size(); i++)
120     {
121
122         if (min >= mp->root[i].sz - sz && mp->root[i].sz - sz >= 0)
123         {
124             min = mp->root[i].sz - sz;
125             j = i;
126         }
127     }
128
129     // if the hole is big enough we decrease the size of the hole and increase the address
130     if (sz < mp->root[j].sz)
131     {
132         mp->root[j].sz -= sz; // decrease the size of the hole
133         mp->root[j].adr += sz; // increase the address of the hole
134
135         return mp->root[j].adr - sz; // return the address of the alloc
136     }
137     // if the hole is exactly the size of the alloc, we remove the hole
138     else if (sz == mp->root[j].sz)
139     {
140         address_t ad = mp->root[j].adr; // put in a variable the address of the alloc
141
142         mp->root.erase(mp->root.begin() + j); // delete the hole
143
144         return ad; // return the address of the alloc
145     }
146
147     std::cout << "No more available space in the memory\n";
148     return -1;
149     // return -1 if the alloc is not possible}
150 };
151
152 // release memory that has already been allocated previously
153 void myFree(mem_t *mp, address_t p, int sz)
154 {
155     // boolean to know if there is a hole on side
156     bool holeOnSides = false;
157
158     // browse the holes
159     for (int j = 0; j < mp->root.size(); j++)
160     {
161         // CASE 1 : There is a hole above and below
162         if (mp->root[j].adr == p + sz && (j != 0 && (mp->root[j - 1].adr + mp->root[j - 1].sz) == p))
163         {
164             std::cout << "Hole on both sides\n";
165             // we increase size of the previous hole with the size the size of the previous hole + the size of
166             mp->root[j - 1].sz = mp->root[j - 1].sz + mp->root[j].sz + sz; // new size of the hole
167             mp->root.erase(mp->root.begin() + j); // erase the next hole
168
169             holeOnSides = true;
170
171             j = mp->root.size();
172         }
173
174         // CASE 2 : There is a hole above but not below

```

```

175     else if (mp->root[j].adr == p + sz && (j == 0 || (mp->root[j - 1].adr + mp->root[j - 1].sz) != p))
176     {
177         std::cout << "Hole on the right but not on the left\n";
178         // we increase size of the previous hole with the size of the free alloc
179         mp->root[j].sz = mp->root[j].sz + sz; // new size of the hole
180         mp->root[j].adr -= sz;                // new address of the hole
181
182         holeOnSides = true; // there is a hole on at least one side
183
184         j = mp->root.size();
185     }
186
187     // CASE 3 : There is a hole below but not above
188     else if (mp->root[j].adr != p + sz && (j != 0 && (mp->root[j - 1].adr + mp->root[j - 1].sz) == p))
189     {
190         std::cout << "Hole on the left but not on the right\n";
191         // we increase size of the previous hole with the size of the hole + the size of the free
192         mp->root[j].sz = mp->root[j].sz + sz; // new size of the hole
193
194         holeOnSides = true; // there is a hole on at least one side
195
196         // we end the loop
197         j = mp->root.size();
198     }
199 }
200
201 // if there is no hole on the sides
202 if (!holeOnSides)
203 {
204     // CASE 4 : There is an alloc on both sides
205     for (int j = 0; j < mp->root.size(); j++)
206     {
207         if (mp->root[j].adr != p + sz && (j == 0 || (mp->root[j - 1].adr + mp->root[j - 1].sz) != p))
208         {
209             std::cout << "Alloc on the left and on the right \n";
210             // we create a new hole at the current index
211             hole_t newHole;
212             newHole.adr = p;
213             newHole.sz = sz;
214             mp->root.insert(mp->root.begin() + j, newHole);
215             // we end the loop
216             j = mp->root.size();
217         }
218     }
219 }
220 };
221
222 // assign a value to a byte
223 void myWrite(mem_t *mp, address_t p, byte_t val)
224 {
225     mp->mem[p] = val;
226 };
227
228 // read memory from a byte
229 byte_t myRead(mem_t *mp, address_t p)
230 {
231     return mp->mem[p];
232 };
233

```

```

234 #endif
235
236 int main()
237 {
238     // initialization of the memory
239     mem_t tempMem = initMem();
240     mem_t *mem = &tempMem;
241
242     // allocation of 3 addresses
243     address_t adr1 = BestFitAlloc(mem, 5); // new address with size of 5 in the memory
244     address_t adr2 = BestFitAlloc(mem, 10); // new address with size of 10 in the memory
245     address_t adr3 = BestFitAlloc(mem, 100); // new address with size of 100 in the memory
246
247     std::cout << "new address with size 5 is " << adr1 << "\n";
248     std::cout << "new address with size 10 is " << adr2 << "\n";
249     std::cout << "new address with size 100 is " << adr3 << "\n";
250
251     myFree(mem, adr2, 10); // free address 2 with size of 10
252     // myFree(mem, adr1, 5); // free address 1 with size of 5
253
254     adr2 = BestFitAlloc(mem, 10);
255     std::cout << "new address with size 10 is " << adr2 << "\n";
256
257     myWrite(mem, adr3, 543); // write on the 1st byte
258     myWrite(mem, adr3 + 9, 34); // write on the 10th byte
259
260     byte_t val1 = myRead(mem, adr3); //
261     byte_t val2 = myRead(mem, adr3 + 9);
262 }

```

Listing 1.6: The entire code

## List of Figures

1	Memory allocation illustration . . . . .	3
2	Free memory, a hole above and below illustration . . . . .	5
3	Free memory, a hole above or below illustration . . . . .	5
4	Free memory, no holes on both sides illustration . . . . .	6
5	Result of the execution . . . . .	9

## Listings

1.1	Initialization of the memory . . . . .	2
1.2	Code to allocate space . . . . .	3
1.3	Code to free allocation . . . . .	7
1.4	Code to write and read bytes on the mem array . . . . .	8
1.5	Main . . . . .	9
1.6	The entire code . . . . .	10