ING 4
Group 5

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.

```
mem_t initMem()
{
    mem_t mem;
    std::vector<hole_t> root;

    // we create a the first hole at address 0 and with size of all the memory
    hole_t Hole;
    Hole.adr = 0;
    Hole.sz = SIZE;
    root.push_back(Hole);

    mem.root = root;

    return mem;
}
```

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.

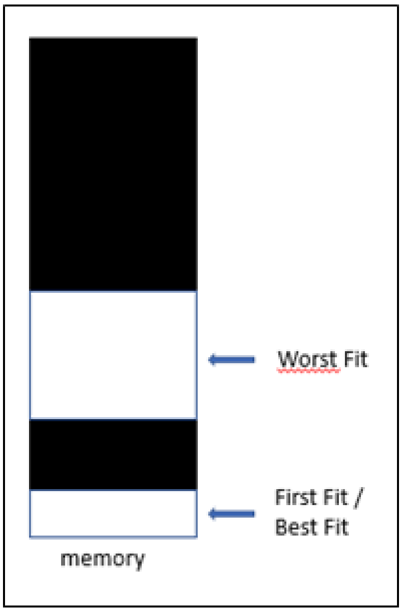

Fig. 1 : Memory allocation illustration

This is the code to allocate space :

```
// allocates space in bytes (byte_t) using First-Fit
address_t FirstFitAlloc(mem_t *mp, int sz)
{
    for (int i = 0; i < mp->root.size(); i++)
    {
        // if the hole is big enough we decrease the size of the hole by the new size
        if (sz < mp->root[i].sz)
        {
            mp->root[i].sz -= sz;  // decrease the size of the hole
            mp->root[i].adr += sz; // increase the address of the hole

            return mp->root[i].adr - sz; // return the address of the alloc
        }
        // if the hole is exactly the size of the alloc, we remove the hole
        else if (sz == mp->root[i].sz)
        {
            address_t ad = mp->root[i].adr;
            mp->root.erase(mp->root.begin() + i); // delete the hole

            return ad;
        }
    }
    std::cout << "No more available space in the memory\n";
    return -1; // return -1 if the alloc is not possible
};

// allocates space in bytes (byte_t) using Worst-Fit
address_t WorstFitAlloc(mem_t *mp, int sz)
{
    int max = 0;
    int j = 0;

    // looks for the largest hole
    for (int i = 0; i < mp->root.size(); i++)
    {
```

```
        if (max <= mp->root[i].sz)
        {
            max = mp->root[i].sz;
            j = i;
        }
    }

    // if the hole is big enough we decrease the size of the hole
    if (sz < mp->root[j].sz)
    {
        mp->root[j].sz -= sz;  // decrease the size of the hole
        mp->root[j].adr += sz; // increase the address of the hole

        return mp->root[j].adr - sz;
    }
    // if the hole is exactly the size of the alloc, we remove the hole
    else if (sz == mp->root[j].sz)
    {
        address_t ad = mp->root[j].adr;

        mp->root.erase(mp->root.begin() + j); // delete the hole

        return ad;
    }


        std::cout << "No more available space in the memory\n";
        return -1;
    // return -1 if the alloc is not possible}
};

// allocates space in bytes (byte_t) using Best-Fit
address_t BestFitAlloc(mem_t *mp, int sz)
{
    int min = SIZE;
    int j = 0;

    //looks for the hole that has the closest size to the allocation size
    for (int i = 0; i < mp->root.size(); i++)
    {

        if (min >= mp->root[i].sz - sz && mp->root[i].sz - sz >= 0)
        {
            min = mp->root[i].sz - sz;
            j = i;
        }
    }

    // if the hole is big enough we decrease the size of the hole and increase the address
    if (sz < mp->root[j].sz)
    {
        mp->root[j].sz -= sz;
        mp->root[j].adr += sz;

        return mp->root[j].adr - sz;
    }
    // if the hole is exactly the size of the alloc, we remove the hole
    else if (sz == mp->root[j].sz)
    {
```

```
        address_t ad = mp->root[j].adr;

        mp->root.erase(mp->root.begin() + j); // delete the hole

        return ad;
    }

        std::cout << "No more available space in the memory\n";
        return -1;
    // return -1 if the alloc is not possible}
};
```

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.

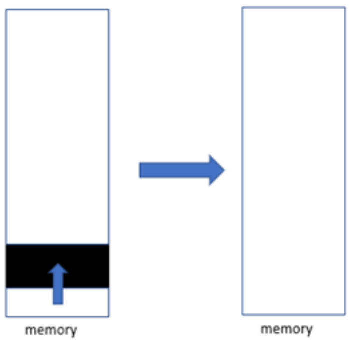

Fig. 2 : Free memory, a hole above and below illustration

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

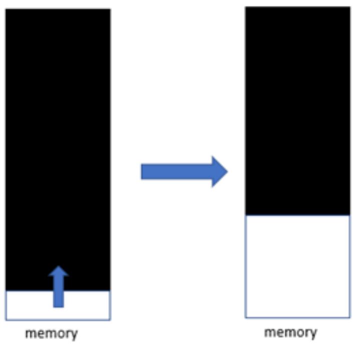

Fig. 3 : Free memory, a hole above or below illustration

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

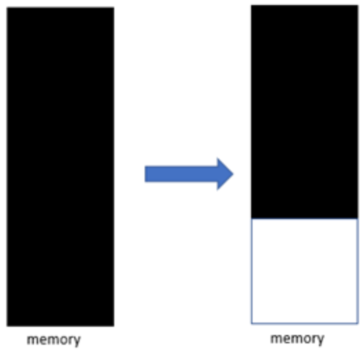

Fig. 4 : Free memory, no holes on both sides illustration

This is the code to free allocation :

```
// release memory that has already been allocated previously
void myFree(mem_t *mp, address_t p, int sz)
{
    // boolean to know if there is a hole on side
    bool holeOnSides = false;

    // browse the holes
    for (int j = 0; j < mp->root.size(); j++)
    {
        // CASE 1 : There is a hole above and below
        if (mp->root[j].adr == p + sz && (j != 0 && (mp->root[j - 1].adr + mp->root[j - 1].sz) == p))
        {
            std::cout << "Hole on both sides\n";
            // we increase size of the previous hole with the size the size of the previous hole + the size of
            mp->root[j - 1].sz = mp->root[j - 1].sz + mp->root[j].sz + sz; // new size of the hole
            mp->root.erase(mp->root.begin() + j);                          // erase the next hole

            holeOnSides = true;

            j = mp->root.size();
        }

        // CASE 2 : There is a hole above but not below
        else if (mp->root[j].adr == p + sz && (j == 0 || (mp->root[j - 1].adr + mp->root[j - 1].sz) != p))
        {
            std::cout << "Hole on the right but not on the left\n";
            // we increase size of the previous hole with the size of the free alloc
            mp->root[j].sz = mp->root[j].sz + sz; // new size of the hole
            mp->root[j].adr -= sz;                // new address of the hole

            holeOnSides = true; // there is a hole on at least one side

            j = mp->root.size();
        }

        // CASE 3 : There is a hole below but not above
        else if (mp->root[j].adr != p + sz && (j != 0 && (mp->root[j - 1].adr + mp->root[j - 1].sz) == p))
        {
            std::cout << "Hole on the left but not on the right\n";
            // we increase size of the previous hole with the size of the hole + the size of the free
            mp->root[j].sz = mp->root[j].sz + sz; // new size of the hole

            holeOnSides = true; // there is a hole on at least one side

            // we end the loop
            j = mp->root.size();
        }
    }

    // if there is no hole on the sides
    if (!holeOnSides)
    {
        // CASE 4 : There is an alloc on both sides
        for (int j = 0; j < mp->root.size(); j++)
        {
            if (mp->root[j].adr != p + sz && (j == 0 || (mp->root[j - 1].adr + mp->root[j - 1].sz) != p))
            {
```

```
                std::cout << "Alloc on the left and on the right \n";
                // we create a new hole at the current index
                hole_t newHole;
                newHole.adr = p;
                newHole.sz = sz;
                mp->root.insert(mp->root.begin() + j, newHole);
                // we end the loop
                j = mp->root.size();
            }
        }
    }
};
```

Listing 1..3: Code to free allocation

4. Write and read

We write and read bytes on the mem array.

```
void myWrite(mem_t *mp, address_t p, byte_t val)
{
    mp->mem[p] = val;
};

// read memory from a byte
byte_t myRead(mem_t *mp, address_t p)
{
    return mp->mem[p];
};
```

Listing 1..4: Code to write and read bytes on the mem array

5. Execution

```cpp
int main()
{
    // initialization of the memory
    mem_t tempMem = initMem();
    mem_t *mem = &tempMem;

    // allocation of 3 addresses
    address_t adr1 = BestFitAlloc(mem, 5);   // new address with size of 5 in the memory
    address_t adr2 = BestFitAlloc(mem, 10);  // new address with size of 10 in the memory
    address_t adr3 = BestFitAlloc(mem, 100); // new address with size of 100 in the memory

    std::cout << "new address with size 5 is " << adr1 << "\n";
    std::cout << "new address with size 10 is " << adr2 << "\n";
    std::cout << "new address with size 100 is " << adr3 << "\n";

    myFree(mem, adr2, 10); // free address 2 with size of 10
    // myFree(mem, adr1, 5);    // free address 1 with size of 5

    adr2 = BestFitAlloc(mem, 10);
    std::cout << "new address with size 10 is " << adr2 << "\n";

    myWrite(mem, adr3, 543);    // write on the 1st byte
    myWrite(mem, adr3 + 9, 34); // write on the 10th byte

    byte_t val1 = myRead(mem, adr3); //
    byte_t val2 = myRead(mem, adr3 + 9);
}
```

Listing 1..5: Main

Result:

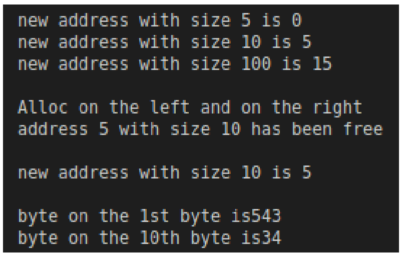

Fig. 5 : Result of the execution

6. Entire code

```
#ifndef __MMU__H__
#define __MMU__H__
#define SIZE 65536
#include <vector>
#include <iostream>
#include <iostream>
#include <stdlib.h>

typedef short byte_t;

typedef int address_t;

// structure of a hole : an address and a size
typedef struct hole
{
    address_t adr;
    int sz;

} hole_t;

// structure of a memory : a size and a vector of holes
typedef struct
{
    byte_t mem[SIZE];
    std::vector<hole_t> root;
} mem_t;

// Initialize memory
mem_t initMem()
{
    mem_t mem;
    std::vector<hole_t> root;

    // we create a the first hole at address 0 and with size of all the memory
    hole_t Hole;
    Hole.adr = 0;
    Hole.sz = SIZE;
    root.push_back(Hole);

    mem.root = root;

    return mem;
}

// allocates space in bytes (byte_t) using First-Fit
address_t FirstFitAlloc(mem_t *mp, int sz)
{
    for (int i = 0; i < mp->root.size(); i++)
    {
        // if the hole is big enough we decrease the size of the hole by the new size
        if (sz < mp->root[i].sz)
        {
            mp->root[i].sz -= sz;  // decrease the size of the hole
            mp->root[i].adr += sz; // increase the address of the hole

            return mp->root[i].adr - sz; // return the address of the alloc
```

```
        }
        // if the hole is exactly the size of the alloc, we remove the hole
        else if (sz == mp->root[i].sz)
        {
            address_t ad = mp->root[i].adr; // put in a variable the address of the alloc

            mp->root.erase(mp->root.begin() + i); // delete the hole

            return ad; // return the address of the alloc
        }
    }
    std::cout << "No more available space in the memory\n";
    return -1; // return -1 if the alloc is not possible
};

// allocates space in bytes (byte_t) using Worst-Fit
address_t WorstFitAlloc(mem_t *mp, int sz)
{
    int max = 0;
    int j = 0;

    // looks for the largest hole
    for (int i = 0; i < mp->root.size(); i++)
    {
        if (max <= mp->root[i].sz)
        {
            max = mp->root[i].sz;
            j = i;
        }
    }

    // if the hole is big enough we decrease the size of the hole
    if (sz < mp->root[j].sz)
    {
        mp->root[j].sz -= sz;  // decrease the size of the hole
        mp->root[j].adr += sz; // increase the address of the hole

        return mp->root[j].adr - sz; // return the address of the alloc
    }
    // if the hole is exactly the size of the alloc, we remove the hole
    else if (sz == mp->root[j].sz)
    {
        address_t ad = mp->root[j].adr; // put in a variable the address of the alloc

        mp->root.erase(mp->root.begin() + j); // delete the hole

        return ad; // return the address of the alloc
    }


        std::cout << "No more available space in the memory\n";
        return -1;
    // return -1 if the alloc is not possible}
};

// allocates space in bytes (byte_t) using Best-Fit
address_t BestFitAlloc(mem_t *mp, int sz)
{
    int min = SIZE;
```

```
    int j = 0;

    //looks for the hole that has the closest size to the allocation size
    for (int i = 0; i < mp->root.size(); i++)
    {

        if (min >= mp->root[i].sz - sz && mp->root[i].sz - sz >= 0)
        {
            min = mp->root[i].sz - sz;
            j = i;
        }
    }

    // if the hole is big enough we decrease the size of the hole and increase the address
    if (sz < mp->root[j].sz)
    {
        mp->root[j].sz -= sz;  // decrease the size of the hole
        mp->root[j].adr += sz; // increase the address of the hole

        return mp->root[j].adr - sz; // return the address of the alloc
    }
    // if the hole is exactly the size of the alloc, we remove the hole
    else if (sz == mp->root[j].sz)
    {
        address_t ad = mp->root[j].adr; // put in a variable the address of the alloc

        mp->root.erase(mp->root.begin() + j); // delete the hole

        return ad; // return the address of the alloc
    }

        std::cout << "No more available space in the memory\n";
        return -1;
    // return -1 if the alloc is not possible}
};

// release memory that has already been allocated previously
void myFree(mem_t *mp, address_t p, int sz)
{
    // boolean to know if there is a hole on side
    bool holeOnSides = false;

    // browse the holes
    for (int j = 0; j < mp->root.size(); j++)
    {
        // CASE 1 : There is a hole above and below
        if (mp->root[j].adr == p + sz && (j != 0 && (mp->root[j - 1].adr + mp->root[j - 1].sz) == p))
        {
            std::cout << "Hole on both sides\n";
            // we increase size of the previous hole with the size the size of the previous hole + the size of
            mp->root[j - 1].sz = mp->root[j - 1].sz + mp->root[j].sz + sz; // new size of the hole
            mp->root.erase(mp->root.begin() + j);                          // erase the next hole

            holeOnSides = true;

            j = mp->root.size();
        }

        // CASE 2 : There is a hole above but not below
```

```
        else if (mp->root[j].adr == p + sz && (j == 0 || (mp->root[j - 1].adr + mp->root[j - 1].sz) != p))
        {
            std::cout << "Hole on the right but not on the left\n";
            // we increase size of the previous hole with the size of the free alloc
            mp->root[j].sz = mp->root[j].sz + sz; // new size of the hole
            mp->root[j].adr -= sz;                // new address of the hole

            holeOnSides = true; // there is a hole on at least one side

            j = mp->root.size();
        }

        // CASE 3 : There is a hole below but not above
        else if (mp->root[j].adr != p + sz && (j != 0 && (mp->root[j - 1].adr + mp->root[j - 1].sz) == p))
        {
            std::cout << "Hole on the left but not on the right\n";
            // we increase size of the previous hole with the size of the hole + the size of the free
            mp->root[j].sz = mp->root[j].sz + sz; // new size of the hole

            holeOnSides = true; // there is a hole on at least one side

            // we end the loop
            j = mp->root.size();
        }
    }

    // if there is no hole on the sides
    if (!holeOnSides)
    {
        // CASE 4 : There is an alloc on both sides
        for (int j = 0; j < mp->root.size(); j++)
        {
            if (mp->root[j].adr != p + sz && (j == 0 || (mp->root[j - 1].adr + mp->root[j - 1].sz) != p))
            {
                std::cout << "Alloc on the left and on the right \n";
                // we create a new hole at the current index
                hole_t newHole;
                newHole.adr = p;
                newHole.sz = sz;
                mp->root.insert(mp->root.begin() + j, newHole);
                // we end the loop
                j = mp->root.size();
            }
        }
    }
};

// assign a value to a byte
void myWrite(mem_t *mp, address_t p, byte_t val)
{
    mp->mem[p] = val;
};

// read memory from a byte
byte_t myRead(mem_t *mp, address_t p)
{
    return mp->mem[p];
};
```

```cpp
#endif

int main()
{
    // initialization of the memory
    mem_t tempMem = initMem();
    mem_t *mem = &tempMem;

    // allocation of 3 addresses
    address_t adr1 = BestFitAlloc(mem, 5);   // new address with size of 5 in the memory
    address_t adr2 = BestFitAlloc(mem, 10);  // new address with size of 10 in the memory
    address_t adr3 = BestFitAlloc(mem, 100); // new address with size of 100 in the memory

    std::cout << "new address with size 5 is " << adr1 << "\n";
    std::cout << "new address with size 10 is " << adr2 << "\n";
    std::cout << "new address with size 100 is " << adr3 << "\n";

    myFree(mem, adr2, 10); // free address 2 with size of 10
    // myFree(mem, adr1, 5);    // free address 1 with size of 5

    adr2 = BestFitAlloc(mem, 10);
    std::cout << "new address with size 10 is " << adr2 << "\n";

    myWrite(mem, adr3, 543);    // write on the 1st byte
    myWrite(mem, adr3 + 9, 34); // write on the 10th byte

    byte_t val1 = myRead(mem, adr3); //
    byte_t val2 = myRead(mem, adr3 + 9);
}
```

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