Lab 5: Virtual memory

This lab is based n two structures. **Hole** that have an address and a size, and **mem** that contains a byte array and a list (vector) of holes

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;

}

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.

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}

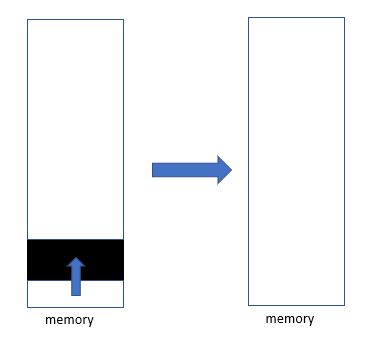
};

3. Free Memory

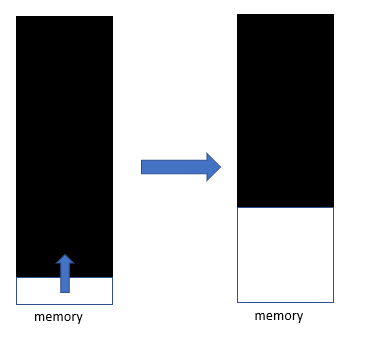
When we don’t use an allocation anymore, we have to free the memory.

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

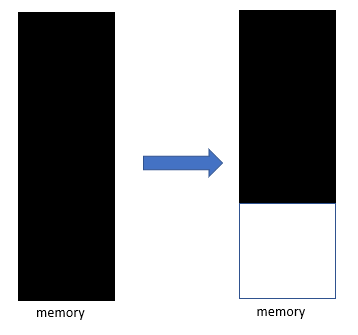
If there is a hole above and below, we delete the following hole, then we increase the size of the previous hole with the size of the free allocation and the size of the following hole.



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



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



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 the free alloc

            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();

            }

        }

    }

};

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];

};

5. 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 the free alloc

            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];

};

#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);

}

Result:

