Paging

Paging main intrest is to **isolate process memory** so it can't not access other processes memory.

With paging we can identify 2 different levels where memory is manage in different ways: - Logical Memory - Physical Memory

Logical Memory

Logical memory uses **contiguous allocation**, this memory is not physical. We manage it with holes, this tells us what memory can be allocated to the process. And to free it we use **contiguous free**. Logical memory is what we can perceive from inside a process *(eg: all **cases of a table are contiguous**)*. Eventhough it's countigous, it is **segmented in contiguous pages**. Pages is useful in isolating the process, will see how later.

Physical Memory

On the other hand Physical Memory, for one process is not contigous, it is divided in **frames**, which or not necessarily contiguous. We define the size of frames.

Paging

Paging is the **bridge between logical and physical memory**. Logical memory is segmented in pages and Physical memory is divied in frames. **Frames and Pages have the same size**. When we allocate space in the logical memory, we allocate memory in pages. When we allocate memory in one page, **we associate all of one frame to this page**. >One frame can only be associated to one process

Pages of processes will keep the corresponding index of the frame in order to be able to write data in the corresponding allocated space in physical memory. By doing this we avoid possible concurrency between processes since frames are specific to one process.

Code Explanation

We finished implementing paging, our code is composed of:

```
#define SIZE 65536
#define PAGE_SIZE 128
#define NUMBER_FRAME SIZE/PAGE_SIZE
#define NUMBER_PAGE NUMBER_FRAME

typedef struct hole
```

```
{
    address_t adr;// address on the hole at the begining
    int sz;//size of the hole
    struct hole *next;//next hole
    struct hole *prev;// previous hole
} hole_t;

typedef struct {
    hole_t* root; // holes list
    int page_table[NUMBER_PAGE];//pages in process
} mem_t; // dynamically allocates a mem_t structure and initializes its content


typedef struct ram
{
    byte_t RAM[SIZE]; //Physical memory
    int frame[NUMBER_FRAME]; //frames
} ram_t;

//Initializing memory
mem_t *initMem();
//Create a hole
hole_t* allocHole(address_t p, int sz, hole_t* prev, hole_t* next);

// allocates space in bytes (byte_t) using First-Fit, Best-Fit or Worst-Fit
hole_t* firstFit(hole_t* m_hole, int sz);
hole_t* bestFit(hole_t* m_hole, int sz);
hole_t* worstFit(hole_t* m_hole, int sz);

//Virtual memory
address_t myContAlloc(mem_t *mp, int sz); // release memory that has already been allocated
hole_t* myContFree(mem_t *mp, address_t p, int sz); // assign a value to a byte
//physical memory
address_t myAlloc(mem_t *mp, int sz); // release memory that has already been allocated pre
void myFree(mem_t *mp, address_t p, int sz); // assign a value to a byte
//write and read into physical memory
void myWrite(mem_t *mp, address_t p, byte_t val); // read memory from a byte
byte_t myRead(mem_t *mp, address_t p);
```

allocHole

Program that create a new hole and initialazing its own values and modify the `prev->next` and `next->prev`.

```
hole_t* allocHole(address_t p, int sz, hole_t* prev, hole_t* next)
{
```

```
    hole_t* new = NULL;
    new = (hole_t*)malloc(sizeof(hole_t));

    new->adr=p;
    new->sz = sz;
    new->prev = prev;
    new->next = next;

    if(prev)
    {
        prev->next = new;
    }

    if(next)
    {
        next->prev = new;
    }

    return new;
}
```

myContAlloc

Look for a hole big enough in order to allocate the needed space. If there is none exit process, if there is one bigger than what is asked modify the address and size of the hole, if same size suppress hole. return the corressponding address where memory has been reserved.

```
address_t myContAlloc(mem_t *mp, int sz)
{
    address_t address = -1;
    hole_t* m_hole = mp->root;
    hole_t* prev= NULL, *next= NULL;
    //Choose the position of the hole
    m_hole = firstFit(m_hole,sz);
    //m_hole = bestFit(m_hole, sz);
    //m_hole = worstFit(m_hole, sz);

//If none exit failure data
    if(!m_hole)
    {
        printf("Memory ERROR");
        exit(EXIT_FAILURE);
    }
    prev = m_hole->prev;
    next = m_hole->next;
```

```
//Allocating data
    address = m_hole->adr;

//Managing holes
    if(m_hole->sz == sz)//Same size -> suppress hole
    {
        if(prev != NULL) //if there is a previous
        {
            prev->next = next;
        }
        else
        {
            mp->root = next;
        }
        if(next!=NULL)//If there is a next
        {
            next->prev = prev;
        }
        free((void*)m_hole); //delete dynamic memory
    }
    else //reducing hole size
    {
        m_hole->adr += sz;
        m_hole->sz -= sz;
    }
    return address;
}
```

myContFree

If no holes create a new one to the root is. Otherwise we are placing the new space in order, looking addresses. Once an emplacement of the futur hole is find we create a new hole using allocHole. After we look if it is next to its previous and next hole. If yes we make a fusion of them. If the hole have no precedent we define it as the root hole. return new hole.

```
hole_t* myContFree(mem_t *mp, address_t p, int sz)
{
    hole_t* next = mp->root, * prev = NULL;

    if(!mp->root)
    {
        mp->root = allocHole(p,sz,NULL,NULL);
        return mp->root;
    }
```

```
    while( next != NULL && p > next->adr)//while I have a next hole and that the variable re
    {
        prev = next;
        next = next->next;
    }
    hole_t* actual = allocHole(p,sz,prev,next);// allocate space for my hole
// if actual follows the precedent hole then merge precedent in actual
    if(prev && prev->adr + prev->sz == actual->adr)
    {
        prev->sz += actual->sz;
        prev->next = actual->next;
        if(actual->next)
        {
            actual->next->prev = prev;
        }
        free(actual);
        actual = prev;
    }
// if next follows the actual hole then merge next in actual
    if(next && actual->adr + actual->sz == next->adr)
    {
        next->sz += actual->sz;
        next->adr = actual->adr;
        next->prev = actual->prev;
        if(actual->prev)
        {
            actual->prev->next = next;
        }
        free(actual);
        actual = next;
    }

// if actual have no precedent define actual as the root hole
    if(!actual->prev)
    {
        mp->root = actual;
    }
    return actual;
}
```

myAlloc (Paging)

Paging help us allocate space in the physical memory. What we do here is that we get the address from the myContAlloc, the we look the corresponding page of the first address and for the last address. For each page from first to last

including them we look if each pages is associated to a frame if not we associate a frame to the page. return the virtual address

```
address_t myAlloc(mem_t *mp, int sz){
    address_t virtual = myContAlloc(mp,sz);
    //Paging
    int first_page = virtual / PAGE_SIZE ; //define my first page use by my allocate memory
    int last_page = (virtual+sz -1) / PAGE_SIZE;//define my last page use by my allocate mem
    int free_frame = 0; //define it here, help us to not start for each page from the begin


    for( int page = first_page; page<=last_page; page++) //for each page from the first to
    {
        if(mp->page_table[page]<0)//if no frame allocated for the page
        {
            //Ask for frame
            while(ram.frame[free_frame]!=0 && free_frame < (NUMBER_FRAME))free_frame ++ ;
            //If No free Frame
            if(free_frame>=(NUMBER_FRAME))
                exit(EXIT_FAILURE);
            ram.frame[free_frame] = PROCESS_ID; //define which process runs on which frame
            mp->page_table[page]=free_frame; //add new frame to page
        }
    }

    return virtual;
}
```

myFree (Paging)

When with free space we also need to free a frame when an entire page is free. We get the page corressponding to the first address of the hole defined by myContFree and the page of the last address. Then we free all frames corresponding to inbetween pages that are not all ready free excluding first and last page and their corresponding frame. After this if the begin and end match in order to free their pages, we free the pages and corresponding frames.

```
void myFree(mem_t *mp, address_t p, int sz){
    hole_t*actual = myContFree(mp,p,sz);
    //Paging
    if(!actual)//If allocation didn't work
        exit(EXIT_FAILURE);
    address_t begin = actual->adr;//first address of my hole
    address_t end = begin + actual->sz - 1;//last address of my hole

    int first_page = begin / PAGE_SIZE ;//page where is my first address
    int last_page = end / PAGE_SIZE;//page where is my last address
```

```
    //delete all page between first and la page of my hole excluding my last and first page
    for( int page = first_page+1; page<last_page ; page++)
    {
        if(mp->page_table[page] != -1){
            ram.frame[mp->page_table[page]]=0;
            mp->page_table[page]=-1;
        }
    }
    //Suppress first page if hole take it all
    if(begin%PAGE_SIZE == 0 && (end%PAGE_SIZE==PAGE_SIZE-1 || last_page > first_page ))
    {
        ram.frame[mp->page_table[first_page]]=0;
        mp->page_table[first_page] = -1;
    }
    //Suppress last page if hole take it all
    if(end%PAGE_SIZE==PAGE_SIZE-1 && last_page > first_page)
    {
        ram.frame[mp->page_table[last_page]]=0;
        mp->page_table[last_page]=-1;
    }
}
```

Read & Write

Read & Write follow the same process we determine the adress in the ram by using paging that gives us the address of the corresponding frame's address in the physical memory from the address in the virtual memory. Then we calculate the offset which is the same in virtual and physicale memory. We have then the address in physical memory which correspond to the address in the virtual memory.

myWrite

```
void myWrite(mem_t *mp, address_t p, byte_t val)
{
    address_t address_frame = mp->page_table[p/PAGE_SIZE]* PAGE_SIZE ;
    address_t offset = p%PAGE_SIZE;
    ram.RAM[address_frame+offset] = val;
}
```

myRead

```
byte_t myRead(mem_t *mp, address_t p)
{
    address_t address_frame = mp->page_table[p/PAGE_SIZE]* PAGE_SIZE ;
```

```
    address_t offset = p%PAGE_SIZE;
    return ram.RAM[address_frame+offset];
}
```

Authors

Paul SADE & Mathis CAMARD