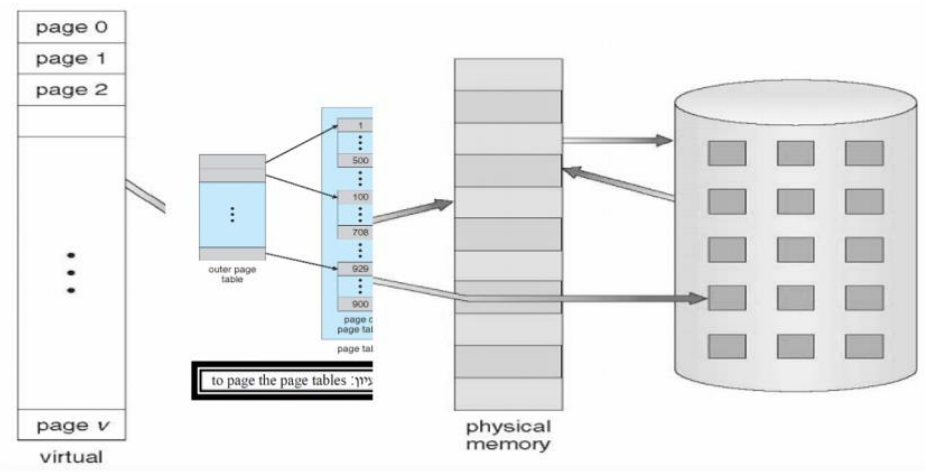
In this exercise we will implement a simulation of cpu access to memory. We will use segment page table mechanism that allows running programs where only part of them are in the memory. The program memory (also called virtual memory) is split to pages which are load to the main memory if needed. We will implement a computer virtual memory with max one program that can run in parallel. The memory mapping we will use is described here:



The main function of the exercise will be built from sequence of "load" and "store" calls (random), this functions simulate the read/write of the CPU (we will focus only on read/write operations for this simulation). We can see example for this main at the appendix below. The main part of the exercise is to implement load and store functions using a page table that maps the logical pages to physical.

You should write it in c++. Except for constructor and destructor, the main functions are:

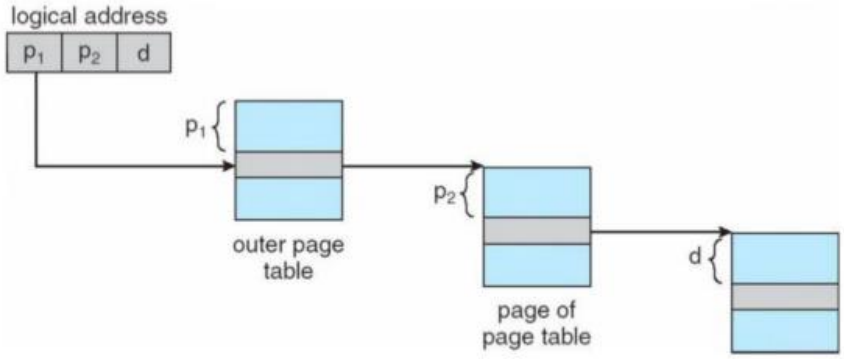
Load function:

Gets a logical address that it has to access to read data. Mainly, this function makes sure the relevant page of the wanted process is in the main memory.

Store function:

Gets an address it has to access for writing data. Similar to load, you have to make sure the page of the wanted address for the given process is in the memory.

We will use in this exercise segment page table with 2 levels. First levelswill have 4 entries for each page type (heap_stack, bss, data, text data) and second level is regular page table. The size of each type of the sub inner tables will be derivate from the sizes that will be sent to the constructor (details below)



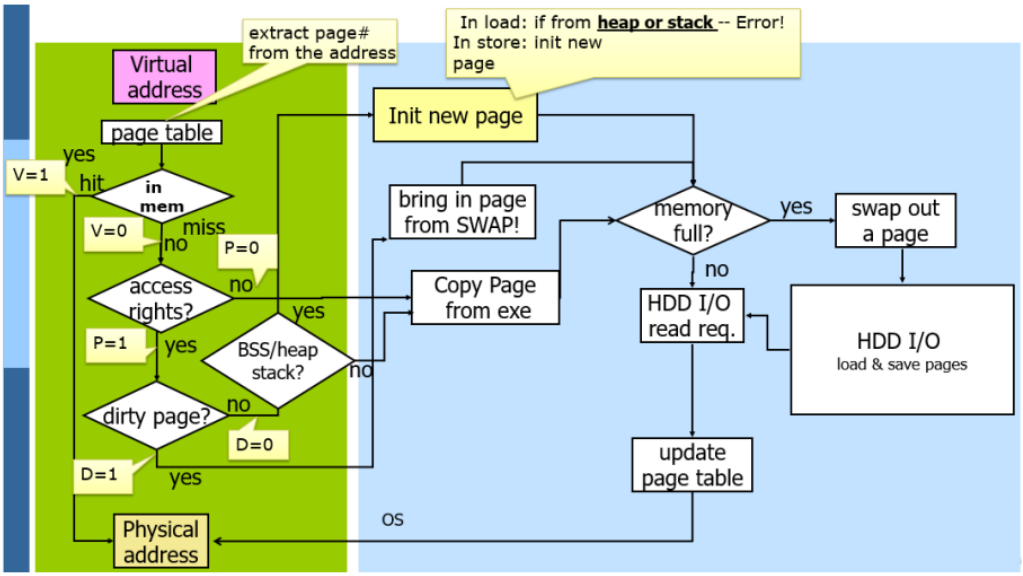


Diagram explained, the explanation is only relevant for one process:

For each virtual address that we get to store and load functions we firstly have to convert from virtual address to physical address via the two levels table:

1. Virtual address is built from the page number in the address that is represented by 12 bits, 2 bits will define the entry in the out table, and the other bits will be defined by the page size that will be sent to the constructor.
2. In case you get virtual address smaller than 12 bits, you have to pad with zeros (0) from the left.
3. Given a virtual address we would like to firstly identify to which page the address is related and what is the hist.
4. At the end, the page table will return us the page we found, and the matching frame.
5. If the page already in the main memory, we check in the page table if we can access the matching frame in the main memory and progress in it by the hist for writing or reading.
6. If the page is not in the main memory, we have to grab it from the correct place. Here there are multiple options:

1. Check if this page has read only permissions (text) or write (data+bss+stack_heap)
    1. In case there is no write permission
        1. An its load operation, then the page is in the executable file.
        2. If its store operation, the function will print matching error and return (the program continues).
    2. In case there is writing permission, it means it is a page of type (data+bss+stack_heap) so we have to check:
        - If the page is dirty – means some stuff written on in and some changes made on it (in other words, there was already a reading with store to this page):
            1. If yes, the file is in the swap file
            2. If not, you have to check if it's a page of (data+bss+stack_heap)
                1. If bss, heap_stack, we allocate new empty page – which is all 0.
                2. Else, address data will be read from the file.

There is one occurrence wher you have to differ between heap_Stack and bss in case of load:

1. If we accessed a page of type heap_stack, we cannot do on him a first time load – other wise it will be an error! In other words, when we load to a page of type heap_stack, it must be in state V=1 or D=1 – else it will be error!.
2. BSS – similar from the file same as data in the case of load on first time.

Data structures of the system:

```
#define MEMORY_SIZE 200
extern char main_memory[MEMORY_SIZE];
```

Class sim_mem: main class of the program, will be defined in your .h file of the program, contains the following fields:

1. Swap_fd – the file descriptor of the swap file, to which we will redirect pages from the main memory (we get it by using open of the file).
2. Program_fd – the file descriptor of the program file we are running in the system and the data sits in it (get by opening the file).
3. Rest of the fields explained a the constructor signature in next page.
4. Page_table – double pointer that can be used for page table with 2 levels.

```
class sim_mem {
    int swapfile_fd;                                    //swap file fd
    int program_fd;                                     //executable file fd
    int text_size;
    int data_size;
    int bss_size;

    int heap_stack_size;
    int num_of_pages;
    int page_size;
    int num_of_proc;

    page_descriptor **page_table;       //pointer to page table

    public:
                sim_mem::sim_mem(char exe_file_name, int text_size,
                        int data_size, int bss_size, int heap_stack_size,
                           int page_size)

        ~sim_mem();

        char load(int address);
        void store(int address, char value);

        void print_memory();
        void print_swap ();
        void print_page_table();
```

In this exercise we will implement segment page table with 2 levels. First level has 4 entries, one for each page type, and second level is a regular page table. As described above.

```
typedef struct page_descriptor
{
        bool valid;
        int frame;
        bool dirty;
        int swap_index;
} page_descriptor;
```

Functions you are required to implement:

1) Ctor – initialize the working space:

```
sim_mem::sim_mem(char exe_file_name[], char swap_file_name[], int text_size,  int data_size,
int bss_size, int heap_stack_size, int page_size)
```

The function gets:

1. String – an executable file name to run
2. String – a swap file name
3. The size of the text area of the program in bytes
4. The size of the data area of the program in bytes
5. The size of the bss area of the program in bytes
6. The size of the heap and stack area of the program in bytes

7. Page size and frame in the system.

The function should:

1. Initialize all data structures of the system
2. Open the exe_file_name and swap_file_name
3. Initialize the main memory with 0.
4. Allocate and initialize the page_table data structure
5. The SWAP file should be initialized with 0 by the logical memory size – with the page size of (data+bss+stack_heap)

You can assume the input sizes are valid: positive integer that their summary is less than the size of the memory segment. You cannot assume the file name is not NULL, that the file Exist, or that you have permission to it.

If the SWAP file not exists – create it.

If the executable not exists – exit the program (error + exit).

The function load:

```
char sim_mem::load(int address)
```

The goal: to make sure that the matching page (to address) of the matching process will sit on the main memory (by the diagram above) and only after the page is already in memory, only then, access the physical address in memory to return the char that sits in this address. In case of error, return \0

Note! We do not load to main memory only the wanted address (in our case single char) but a WHOLE page, meaning PAGE_SIZE chars.

The store function:

void sim_mem::store(int address , char value)

the goal: to make sure the matching page to address of the matching process (one or two processes) will really copy to the main memory. After the page copied, you have to access the physical address and store the value char in this address.

The print_memory function:

void sim_mem::print_memory();

the function prints the content of the main memory.

The print_swap function:

void sim_mem::print_swap();

the function prints the content of the swap file.

The print_page_table function

void sim_mem::print_page_table();

you get the implementation below.

Destructor:

void sim_mem::~sim_mem ();

close all open files, release the memory.

You can add help functions if you want.

Possible Main program:

```
char main_memory[MEMORY_SIZE];


int main()
{
    char val;
     sim_mem mem_sm("exec_file", "swap_file" ,16, 16, 32,32, 16, 8);
     mem_sm.store(98,'X');
    val = mem_sm.load (8);
    mem_sm.print_memory();
     mem_sm.print_swap();
}
```

Additional notes for implementation:

1. When initialize the system, do not load any of the pages from the file to memory. The page loads will be in "lazy" way, (demand paging).

2. Note that on text pages you cannot write! We will never have to backup changes in them, meaning we do not have to write them to swap, and we always can read from them again from the executable file if needed.
3. Pages written to swap by "first-fit" in the first empty space. Where a page loaded from the swap to memory. You have to delete (reset) his content in the swap files to zeros.
4. Choosing a page to free from the memory when the memory is full – using LRU algorithm.
5. You cannot change the function signature.
6. Our system runs single process. This process will be "loaded" to the system from the executable file that his name passed as argument to the program, and saved in a variable of type sim_mem
7. For ease, this type will not be standard executable file that contains machine code commands – but we will relate to him like it is.

You have to submit:

Sim_mem.h

Sim_mem.cpp

Main.cpp

RAEDME

```
/**************************************************************************************/
void sim_mem::print_memory() {
      int i;
      printf("\n Physical memory\n");
      for(i = 0; i < MEMORY_SIZE; i++) {
            printf("[%c]\n", main_memory[i]);
      }
}

/**************************************************************************************/

void sim_mem::print_swap() {
      char* str = (*char)malloc(this->page_size *sizeof(char));
      int i;
      printf("\n Swap memory\n");
      lseek(swapfile_fd, 0, SEEK_SET); // go to the start of the file
      while(read(swapfile_fd, str, this->page_size) == this->page_size) {
            for(i = 0; i < page_size; i++) {
                  printf("%d - [%c]\t", i, str[i]);
            }

            printf("\n");
      }
}
```

```
/**************************************************************************************/
void sim_mem::print_page_table() {
      int i;
      num_of_txt_pages = text_size / page_size;
      num_of_data_pages = data_size / page_size;
      num_of_bss_pages = bss_size / page_size;
      num_of_stck_heap_pages = heap_stack_size / page_size;

          printf("Valid\t Dirty\t Frame\t Swap index\n");
          for(i = 0; i < num_of_txt_pages; i++) {
            printf("[%d]\t[%d]\t[%d]\t[%d]\n",
                        page_table[0][i].V,
                        page_table[0][i].D,
                        page_table[0][i].frame ,
                        page_table[0][i].swap_index);

           }
    printf("Valid\t Dirty\t Frame\t Swap index\n");
          for(i = 0; i < num_of_data_pages; i++) {
            printf("[%d]\t[%d]\t[%d]\t[%d]\n",
                        page_table[1][i].V,
                        page_table[1][i].D,
                        page_table[1][i].frame ,
                        page_table[1][i].swap_index);

           }

    printf("Valid\t Dirty\t Frame\t Swap index\n");
          for(i = 0; i < num_of_bss_pages; i++) {
            printf("[%d]\t[%d]\t[%d]\t[%d]\n",
                        page_table[2][i].V,
                        page_table[2][i].D,
                        page_table[2][i].frame ,
                        page_table[2][i].swap_index);

           }

    printf("Valid\t Dirty\t Frame\t Swap index\n");
          for(i = 0; i < num_of_stack_heap_pages; i++) {
            printf("[%d]\t[%d]\t[%d]\t[%d]\n",
```

```
                        page_table[3][i].V,
                        page_table[3][i].D,
                        page_table[3][i].frame ,
                        page_table[3][i].swap_index);

          }


}
```