Virtual Memory Simulator

Stancu Gabriel – Iulian
Technical University of Cluj – Napoca
Structure of Computer Systems – 2020/21
Teaching assistant: Neagu Mădălin Ioan

Virtual Memory Simulator

Contents

1. Introduction

1.1. Context

The goal of the project is to design, build and test for correct functionality a **virtual memory simulator**. Virtual memory is a memory management technique supported today by most of the operating systems, which makes all processes run in their own dedicated address space. This way, some problems like unintended memory overwriting between processes, concerns about memory management from the programmer's point if view, but also the limitation of the actual physical space, are all avoided and performed in the background by the operating system, using hardware and software resources.

The reason behind building such a simulator is making a benchmark-like program that tests and assesses the efficiency of these background operations performed by the operating system together with the hardware components of the computer, aspects which are abstracted from the user, in order to free the user / the program from managing the memory by itself, to increase memory security and provide more memory than it is physically available, through the concept of **pagination**: bring data from the secondary storage (the **"disk"**) to the main memory (the **"RAM"** – Random Access Memory) and "display" it in the virtual memory of the process. The data is brought in small units, called **pages**.

1.2. Specifications

The simulator can be made using Java, C, C++, C# or other such programming languages. Because of the personal level of knowledge and programming languages preferences (also because of the desire to learn new concepts in this language), the project will be written in C#. As a development IDE, Visual Studio will be used. The project will provide an interactive and attractive graphical user interface, made with WPF. The final output of the project is an executable program that performs all the tasks specified in the following part of the documentation.

1.3. Objectives

Write a graphical simulation program of the main operations performed by the virtual memory: finding an information in the virtual memory, placing a page of content from the physical memory in the virtual memory, the replacement of a page, in case the virtual memory is full and a new page needs to be loaded, case in which the operation system decides based on internal algorithms which pages from the main memory can be replaced.

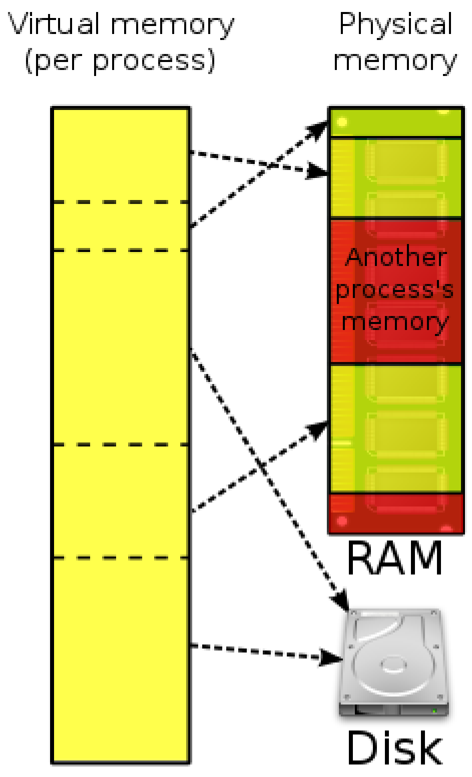
2. Bibliographic study

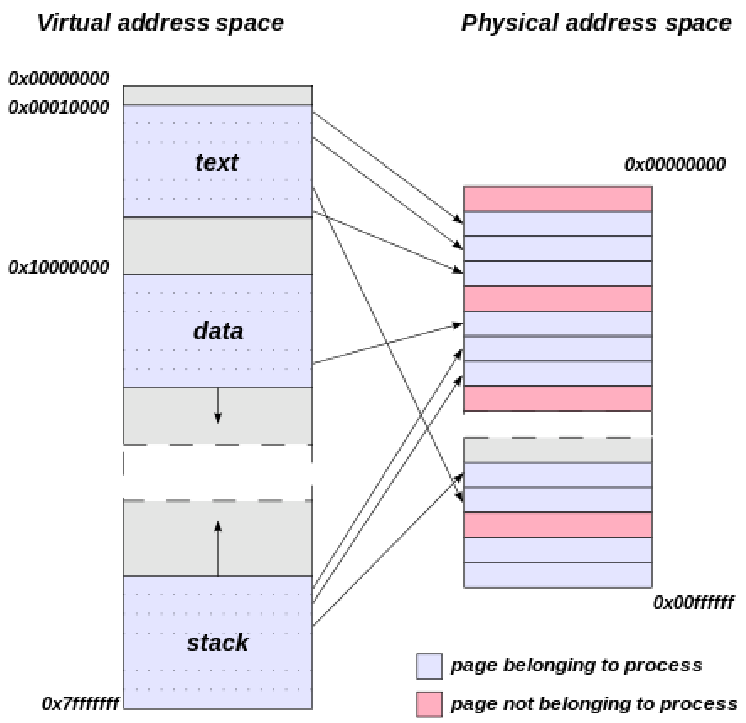
Virtual memory represents a memory management technique that the operating system uses in order to give the program / the user the false impression / the "illusion" that it can actually access a very large amount of memory, the whole memory of the computer actually. This way, all addresses the program / process accesses are called **"virtual addresses"**, which are mapped behind by the operating system to actual physical locations of the memory, which are called **"physical addresses"**. The operating system uses hardware and software resources for this purpose. To translate the virtual addresses to physical addresses, the operating system uses an address translation hardware from the CPU, called **memory management unit (MMU)**. From the software point of view, the operating system may extend the memory given to a process, even beyond the actual limit of the physical limit.

For finding the required information in the real memory, **page tables** are used. Each entry in this table contains both an address towards the location in the physical memory, but also in the virtual memory where the page of information is located. These pages are blocks of memory of different sizes, which are usually at least **4 kilobytes** in size, but their size can be increased if the total memory of the computer is large. This way, the smaller the page size, the higher the number of entries in the page table is. Each entry also contains a flag indicating whether the page is loaded or not in the virtual memory. If it's not, a **page fault exception** is raised by the hardware, which will be handled later by the operating system: if there are enough free pages for the content to be transferred, it will be transferred and the flag will indicate that the page exists in memory, otherwise page replacement is employed. If the page is already in the memory, nothing happens, as the entry in the page table points towards a valid location.

Placing the information in the main memory means that once the information is found in the secondary memory (on the **disk**), it is ready to be loaded and will be loaded in the main memory, i.e. in the **RAM**. Of course, this happens if the required pages were not already loaded.

Page replacement happens when a requested page is not in the main memory and the page fault exception was raised, and a free page cannot be allocated for the block of memory, either because there are no more free pages or the number of remaining pages is smaller than the needed number of pages for the data block. In this case, special algorithms are employed, in order to decide which pages are not needed anymore / were last used / least used, these pages being then paged out and replaced by the new content that needs to be loaded.





3. Analysis

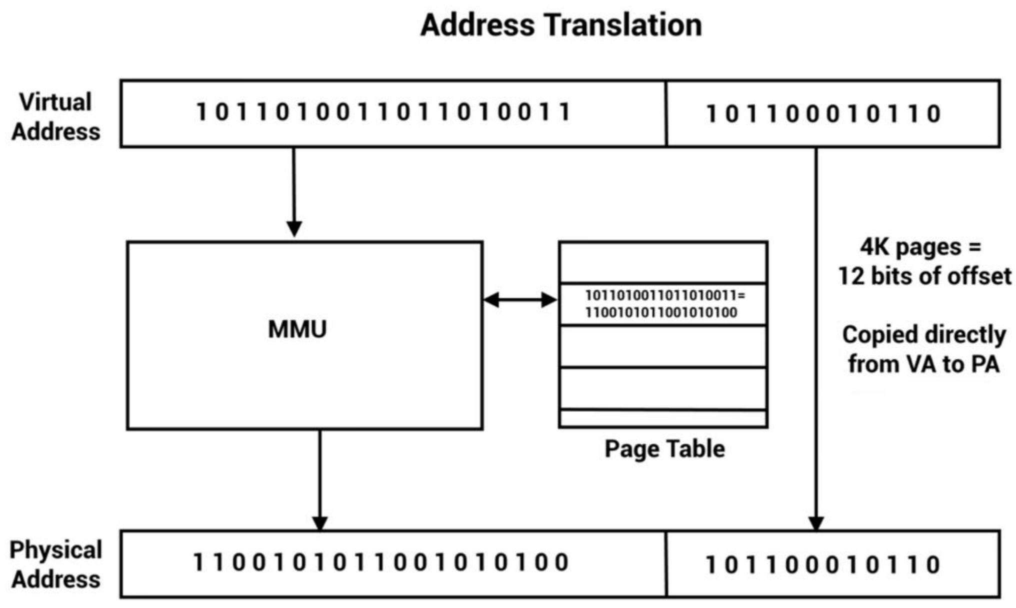
3.1. Finding the required information

The virtual memory mechanism gives each running application the illusion that it is the only running application in the system, taking the entire physical memory. That is, of course, not true. Still, each process, while running together with other processes, can access the actual content from the physical memory, without clashing with other processes, as any of them still have their own allocated memory. For this mechanism to work, the already-mentioned Memory Management Unit (MMU) is needed. It receives a virtual address from the process, it translates it to the actual / physical address, and it finally gives the process the data from the computed address. But how does this address computation work? Here is where the Page Table comes in: the main memory is divided into pages or blocks of memory. We will consider a typical system with:

- 2 ^ 32 = 4 GB of RAM
- each page containing 2^12 addresses = 4 KB of memory / page
- 2 ^ 32 / 2 ^12 = 2 ^ 20 pages

Now, the virtual memory is sending a 32-bit address to be translated to the physical memory address. The first 20 bits of the address are sent to the MMU, to be mapped to the actual page address in the physical memory. 20 bits are used for 2 ^ 20 addresses. The MMU takes these 20 bits and maps the entry corresponding to the given address to the physical address of the page from that entry, which is then returned. The remaining 12 bits are used for offsets inside the page. Because the page allocates 4 KB of memory, i.e. 2 ^ 12 addresses, 12 bits are used for the offset inside the block. These bits are not sent to the MMU but are rather combined with the resulting 20 bits from the MMU. The 20 bits from the MMU together with the 12 bits of offset give the actual address in the physical memory.

The illustration below sums up the discussed concepts:



3.2. Placing a page in the main memory

As mentioned before, the concept of pagination is used for abstracting the communication system between the main memory (the “RAM”) and the secondary memory (the “disk”). The data is stored on the disk but when needed, it is retrieved for being used in the RAM. In case the process tries to access a page / content that is not currently loaded in the RAM, a **Page Fault** is raised and the operating system performs the following steps: the required data is first looked for on the disk, then it is brought to the main memory grouped in pages. Depending on the size of the required data, one or more pages will be allocated. Then, the obtained pages will be placed in the main memory and, for being accessible to the process, their physical addresses will be placed in the Page Table, together with a virtual address from where they can be accessed by the process, through the MMU. In case the Page Table is full, a page replacement policy is implied (we will discuss some of them at 3.3). After this, the needed entries in the Page Table are free and can be used by the retrieved pages.

3.3. Replacing a page

After composing the required data from the secondary memory, i.e. the disk, in the pages to be loaded in the main memory, the following situation might be encountered: the Page Table is either full or it has less entries than the retrieved number of pages. In this case, a page replacement policy is implied, i.e. the operating system decides which page / entry from the Page Table is the most suitable to be dumped and replaced by the new content. The page replacement technique we will discuss is the "demand paging": each process starts with no entries in the Page Table, or no data loaded in the RAM. As they commit Page Faults, the Page Table is loaded, together with the main memory. A "priority queue" will be used for deciding which page was less / last used. Each page is put in the queue when loaded from the disk to the main memory, and any edit on the data contained by the page will place the page at the end of the "to be replaced pages queue". When the Page Table is full, the page from the front of the queue is the one discarded and replaced by the new one, which will, of course, be placed at the end of the queue.

4. Design

For the design of our application, we will employ object-oriented programming (OOP). The following main classes will be designed: OS, Process, MMU, PageTable, Page, RamFrame. The other secondary classes (which provide certain functionalities and the UI classes) will be discussed in the next chapter of this paper. Each of main classes will also be briefly discussed in what follows.

4.1. OS

This class will act as the operating system of the computer from our simulation. Each action that needs to be taken will be processed by this class. Each time a process needs data from the main memory, the OS is notified (by the MMU). Each time the process commits a page fault, the OS will command the retrieving of the required information from the secondary memory to the main memory. If a page fault is encountered, it will decide and it will replace the most suitable page. Also, whenever a modified page is accessed (overwriting or it needs to be swapped), the OS is notified, storing the modified page back to the Disk.

4.2. Process

This class will act as a process running in the computer memory of our simulation. For this simulation, it will not actually contain data, but it will rather have a PageTable attached to it, containing the pages (loaded or that might be loaded at some point), together with the total number of commands the process will execute during the simulation and the actual size of its PageTable.

4.3. MMU

This class will simulate the memory management unit of the computer. It will receive the list of generated commands that will be executed during the simulation from the OS and handles them one by one. First it checks whether the page is loaded or not. If not, it will notify the OS to load it. When the OS notifies back the loading of the page (or if it was already loaded), the command is executed: the read or the write request on the current page. If the page was already modified and an overwrite is required, the MMU will notify the OS to save the page first.

4.4. PageTable

This class will hold the list of the pages (either containing data or empty) of the assigned process. It will be used for displaying the state of the process and its pages in the user interface. The replacement policy will be the following: the page with the least recent access will be replaced, so a priority queue will be needed for maintaining the pages.

4.5. Page

This class will represent the storage unit used for communicating between the process and the actual memory through paging. It will also contain a value denoting whether the page is empty or dirty (edited from the point it was brought in the main memory). A field representing the request of this page to be loaded in RAM, a field stating the status of the page (loaded / not loaded) and a field storing the last access on the page (for keeping a priority queue in case a page swap is needed) are also present.

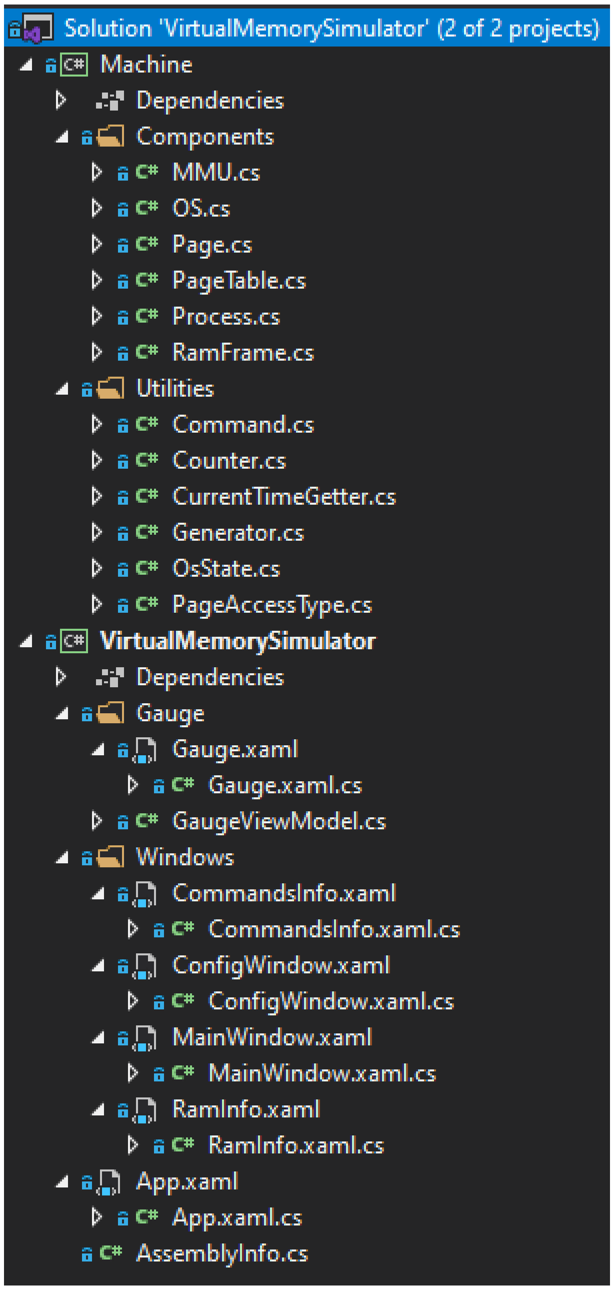
4.6. RamFrame

This is the class representing an actual frame from the RAM. A list of such RamFrames will be stored in the OS class, in order to track the actual content of the RAM, based on the states of the processes' pages from their own page tables. This class will also be used for displaying the RAM status in the user interface.

5. Implementation

As stated in the previous chapter, the object-oriented programming was employed for the development of the application. The provided solution for the application was divided in two smaller projects: **Machine** and **VirtualMemorySimulator**. This decision was taken to preserve certain functionalities between the two main components of the application: the business logic and the user interface (the UI) logic, but also for a better organization of the project. The first project provides the business logic of the application while the second one maps the required results from the first one in the user interface. So, the second project has a dependency on the first one.

Furthermore, the **Machine** project was divided in two namespaces, for better organization and for keeping classes with similar functionalities closer to each other. The two namespaces are called **Components** (here the classes representing hardware components are stored) and **Utilities** (non-hardware representing classes that provide certain functionalities needed for the correct workflow of the application.



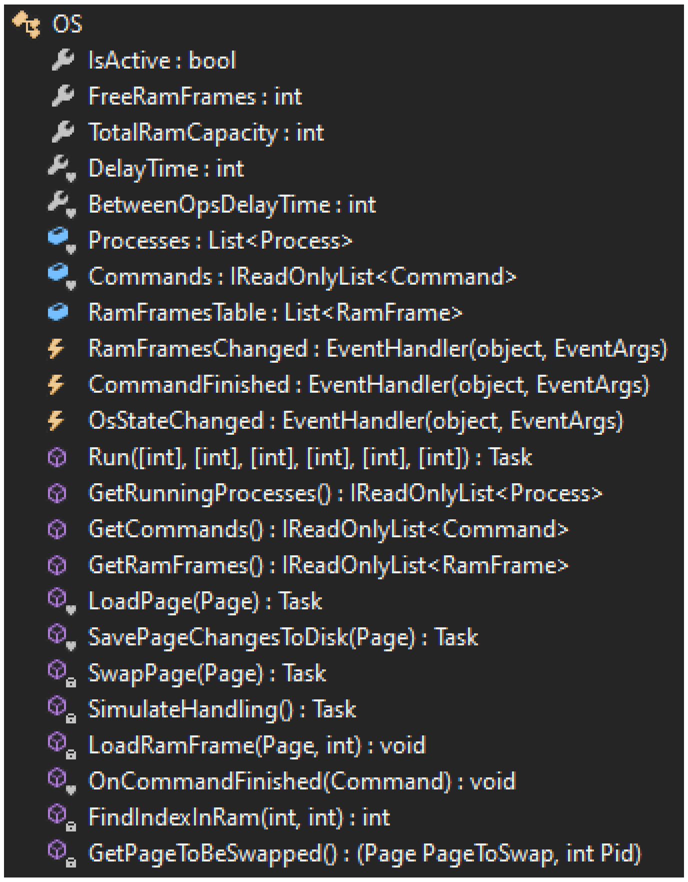
```
Solution 'VirtualMemorySimulator' (2 of 2 projects)
  Machine
    Dependencies
    Components
      MMU.cs
      OS.cs
      Page.cs
      PageTable.cs
      Process.cs
      RamFrame.cs
    Utilities
      Command.cs
      Counter.cs
      CurrentTimeGetter.cs
      Generator.cs
      OsState.cs
      PageAccessType.cs
  VirtualMemorySimulator
    Dependencies
    Gauge
      Gauge.xaml
        Gauge.xaml.cs
      GaugeViewModel.cs
    Windows
      CommandsInfo.xaml
        CommandsInfo.xaml.cs
      ConfigWindow.xaml
        ConfigWindow.xaml.cs
      MainWindow.xaml
        MainWindow.xaml.cs
      RamInfo.xaml
        RamInfo.xaml.cs
    App.xaml
      App.xaml.cs
    AssemblyInfo.cs
```

Finally the **VirtualMemorySimulator** which provides user interface logic functionality, was divided in two namespaces. The first one (**Gauge**) stores the functionality logic of a custom - made user control, which is used for displaying the fill factor of the RAM memory from the simulation. The second namespace is called **Windows**, storing the forms / windows where the user interface is displayed.

The structure of the project can be observed on the attached image. In what follows each class' implementation will be discussed. We begin with the **Components** namespace from the **Machine** project, for which all classes are written in C#, this project having the template of a class library in the .NET Core 3.1 framework.

A. The Machine project, Components namespace

1. OS



```
OS
    IsActive : bool
    FreeRamFrames : int
    TotalRamCapacity : int
    DelayTime : int
    BetweenOpsDelayTime : int
    Processes : List<Process>
    Commands : IReadOnlyList<Command>
    RamFramesTable : List<RamFrame>
    RamFramesChanged : EventHandler(object, EventArgs)
    CommandFinished : EventHandler(object, EventArgs)
    OsStateChanged : EventHandler(object, EventArgs)
    Run([int], [int], [int], [int], [int], [int]) : Task
    GetRunningProcesses() : IReadOnlyList<Process>
    GetCommands() : IReadOnlyList<Command>
    GetRamFrames() : IReadOnlyList<RamFrame>
    LoadPage(Page) : Task
    SavePageChangesToDisk(Page) : Task
    SwapPage(Page) : Task
    SimulateHandling() : Task
    LoadRamFrame(Page, int) : void
    OnCommandFinished(Command) : void
    FindIndexInRam(int, int) : int
    GetPageToBeSwapped() : (Page PageToSwap, int Pid)
```

The OS properties:

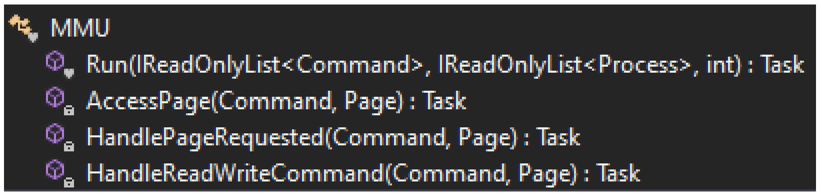
- **IsActive**: signals whether the OS is performing a task
- **FreeRamFrames**: counts the number of not loaded RAM frames
- **TotalRamCapacity**: the parameter used for stating the number of frames the RAM can be divided into
- **DelayTime**: the parameter indicating the time the program will sleep, in order to simulate the long access time needed by OS to access the Disk
- **BetweenOpsDelayTime**: delay time placed between operations; makes it easier to follow the results of each individual operation
- **Processes**: the list of running processes in the simulation
- **Commands**: the list of commands executed during the simulation
- **RamFramesTable**: the list of RAM frames and their state

Main methods:

- **Run**: called by the other project, initializes the simulation (sets the simulation parameters, creates the pages / frames, assigns a list of commands to the MMU etc.)
- **LoadPage**: whenever the MMU decides a page needs to be loaded, this method is called and the OS handles it: if there are free RAM frames, **LoadRamFrame** is called, otherwise **SwapPage** is called
- **SavePageChangesToDisk**: called whenever an edited page (dirty bit is '1') needs to be swapped or overwritten
- **SwapPage**: swaps the given page with another one that needs to be loaded from the Disk
- **SimulateHandling**: delays the flow (sleeps) for the preset amount of time
- **LoadRamFrame**: a RAM frame is loaded with the given page data
- **OnCommandFinished**: fires an event (**CommandFinished**) that signals the UI project one of the commands is finished and the interface needs to be updated

This class is the core of the Machine package. Most of the processing tasks are performed here, whether we talk about the data transfer between memories or managing the simulated hardware resources of the computer. Events are used for signaling the other project when certain occurrences take place, eventually updating the UI in an asynchronous way. Also, most of the methods are encapsulated in a Task object, which provide asynchronous execution: usually the workflow is placed by the Task on another thread, and the application remains responsive: even during the delays used for simulating the real behavior of the computer, the user interface remains active and any number of concurrent commands can be placed by the user and executed by the simulator.

2. MMU



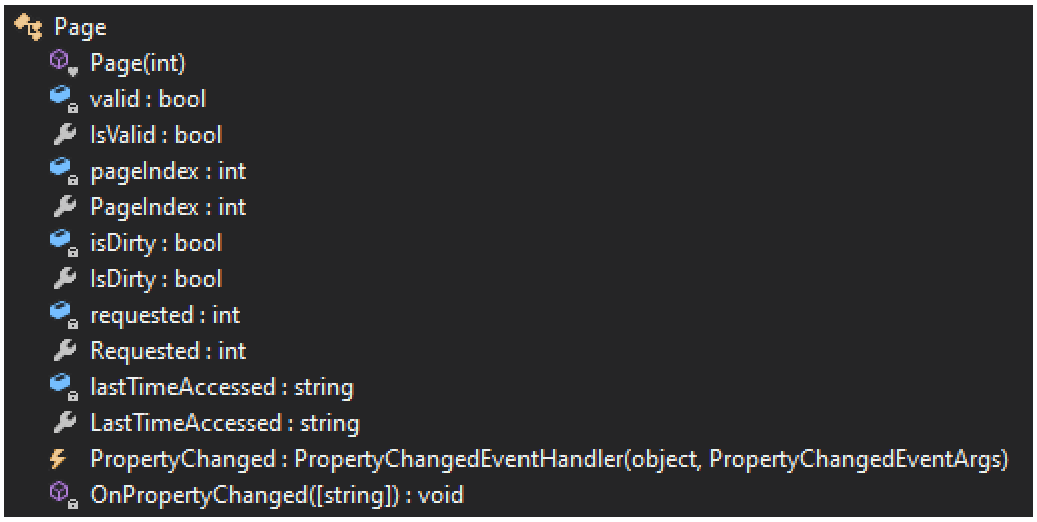
```
MMU
  Run(IReadOnlyList<Command>, IReadOnlyList<Process>, int) : Task
  AccessPage(Command, Page) : Task
  HandlePageRequested(Command, Page) : Task
  HandleReadWriteCommand(Command, Page) : Task
```

Methods:

- **Run**: receives all the commands that need to be executed, a list of the active processes and the delay that needs to be placed between two successive commands. Each command is placed on its own Task to provide asynchronous execution

- **AccessPage**: called for simulating the two ways a page can be accessed: for reading or writing. It checks whether the page is loaded in the RAM or not (checks the IsValid property of the received page). If not loaded, **HandlePageRequested** is called, otherwise **HandleReadWriteCommand** is called
- **HandlePageRequested**: calls the OS' **LoadPage** method and sets the properties of the page according to the received command
- **HandleReadWriteCommand**: simulates the reading or the writing of the page. If a writing is requested and the page is dirty, the OS is called for saving it back to the disk first

3. Page



```
Page
    Page(int)
    valid : bool
    IsValid : bool
    pageIndex : int
    PageIndex : int
    isDirty : bool
    IsDirty : bool
    requested : int
    Requested : int
    lastTimeAccessed : string
    LastTimeAccessed : string
    PropertyChanged : PropertyChangedEventHandler(object, PropertyChangedEventArgs)
    OnPropertyChanged([string]) : void
```

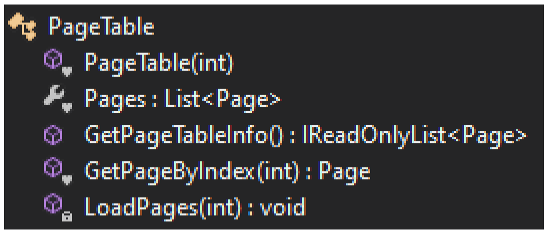
<u>Fields encapsulated by their properties:</u>

- **valid**: tells whether the page is loaded or not in the RAM
- **pageIndex**: the index of the page in the page table containing it
- **isDirty**: indicates whether the page was modified since it was loaded in the RAM
- **requested**: usually -1, it is only modified with the PID of the process requesting it to be loaded, which will be used by the OS to determine what pages need to be loaded. After being loaded, it is reset to -1
- **lastTimeAccessed**: stores the moment of the last access of this page in format "HH:mm:ss.fff"; used for maintaining a priority queue between pages

This class implements the **INotifyPropertyChanged** interface: each time a property value has changed, the **OnPropertyChanged** method is called and the **PropertyChanged** event is fired, signaling the observers.

4. PageTable

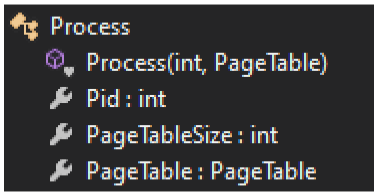
Contains a list of the pages of the assigned process (the Pages property) and methods for loading the pages with initial values, for getting the whole list of pages in read-only mode, or for getting an individual page based on its index.



```
PageTable
    PageTable(int)
    Pages : List<Page>
    GetPageTableInfo() : IReadOnlyList<Page>
    GetPageByIndex(int) : Page
    LoadPages(int) : void
```

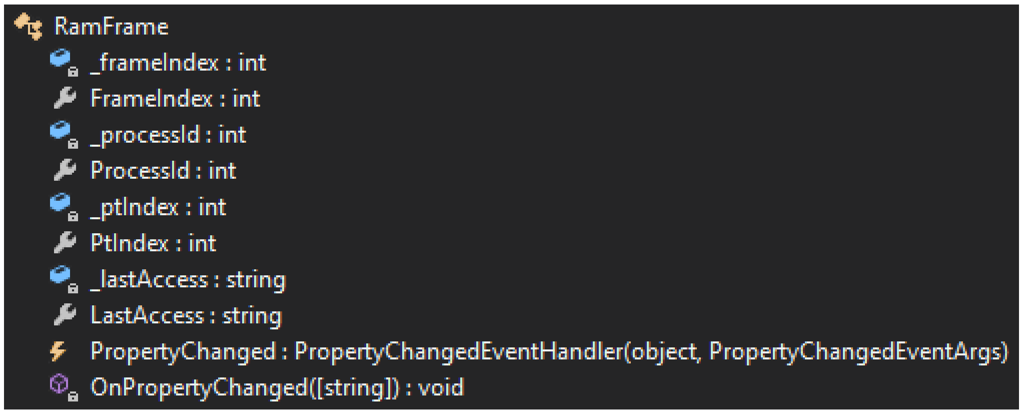
5. Process

Methodless class, only stores relevant data about each process: the process id (pid), the size of its PageTable and the PageTable itself.



```
Process
    Process(int, PageTable)
    Pid : int
    PageTableSize : int
    PageTable : PageTable
```

6. RamFrame



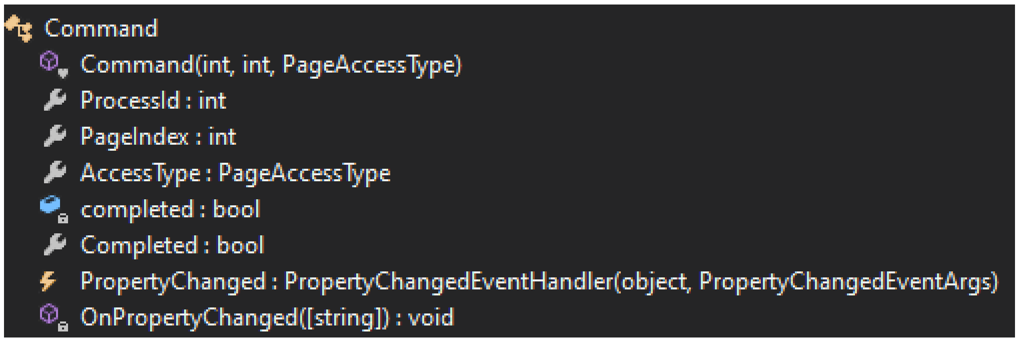
```
RamFrame
    _frameIndex : int
    FrameIndex : int
    _processId : int
    ProcessId : int
    _ptIndex : int
    PtIndex : int
    _lastAccess : string
    LastAccess : string
    PropertyChanged : PropertyChangedEventHandler(object, PropertyChangedEventArgs)
    OnPropertyChanged([string]) : void
```

Stores relevant data about the frame: the index in the RAM frames table, the pid of the process it belongs to, the index if the page in the page table mapped to this frame, the last access moment of the frame. When loaded, the value of the page is the same with the one stored in this frame.

This class implements the **INotifyPropertyChanged** interface: each time a property value has changed, the **OnPropertyChanged** method is called and the **PropertyChanged** event is fired, telling the UI project the usage rate of the RAM has changed and the UI needs to be updated.

B. The Machine project, Utilities namespace:

1. Command

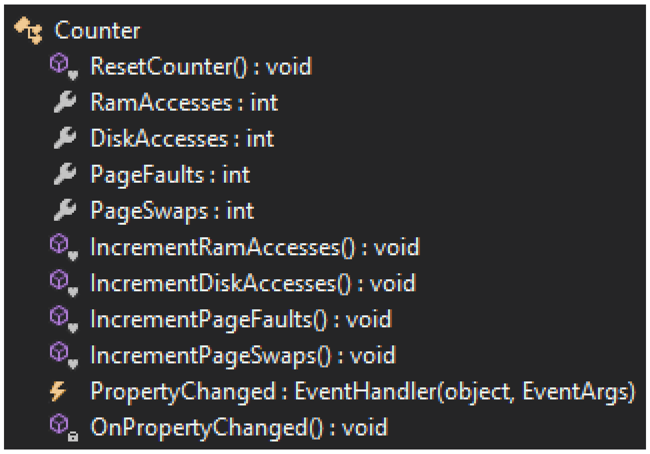


```
Command
    Command(int, int, PageAccessType)
    ProcessId : int
    PageIndex : int
    AccessType : PageAccessType
    completed : bool
    Completed : bool
    PropertyChanged : PropertyChangedEventHandler(object, PropertyChangedEventArgs)
    OnPropertyChanged([string]) : void
```

Stores data about: the process which requested this command, the page index it needs to access, the access type (read or write) and the status (completed, not completed). The **INotifyPropertyChanged** interface is implemented, when the command is complete an event is fired to signal the UI to update the commands list status.

2. Counter

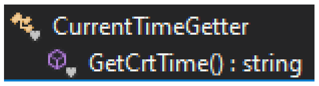
Class used for storing the accesses of the memories (the RAM and the Disk), but also the number of Page Faults and the number of page swaps that were performed. The increment methods are only called by the OS class when the appropriate event occured. The **INotifyPropertyChanged** interface is also implemented, for updating the UI asynchronously.



```
Counter
    ResetCounter() : void
    RamAccesses : int
    DiskAccesses : int
    PageFaults : int
    PageSwaps : int
    IncrementRamAccesses() : void
    IncrementDiskAccesses() : void
    IncrementPageFaults() : void
    IncrementPageSwaps() : void
    PropertyChanged : EventHandler(object, EventArgs)
    OnPropertyChanged() : void
```

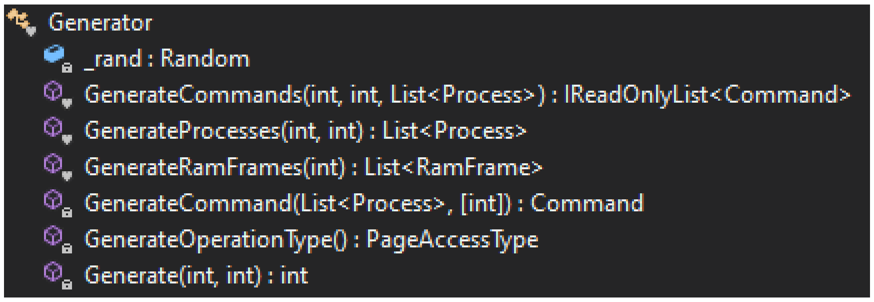
3. CurrentTimeGetter

Gets the current time which will be assigned to pages / frames.



```
CurrentTimeGetter
    GetCrtTime() : string
```

4. Generator

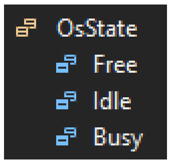


```
Generator
    _rand : Random
    GenerateCommands(int, int, List<Process>) : IReadOnlyList<Command>
    GenerateProcesses(int, int) : List<Process>
    GenerateRamFrames(int) : List<RamFrame>
    GenerateCommand(List<Process>, [int]) : Command
    GenerateOperationType() : PageAccessType
    Generate(int, int) : int
```

Class used for generating the objects required by the simulation: the commands and their properties, the processes, the RAM frames. The commands are generated randomly (their type, the process they belong to, the page they access).

5. OsState

Enum used for increasing readability of the code and storing the state of the OS:

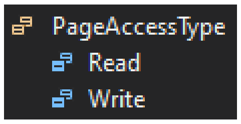


```
OsState
    Free
    Idle
    Busy
```

- Free: the simulation is either not started or finished
- Idle: the simulation is started but the OS is not required to do anything
- Busy: the OS is required to load / save / swap a page between the memories

6. PageAccessType

Enum used for increasing readability of the code and storing the two types a command can have: Read or Write.

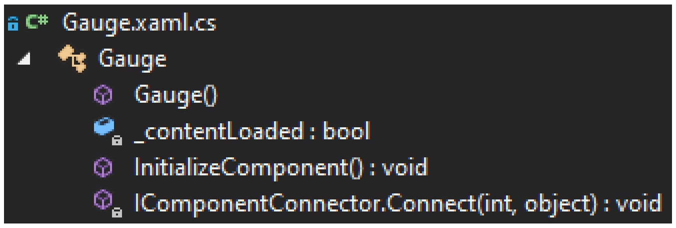


```
PageAccessType
    Read
    Write
```

C. The VirtualMemorySimulator project, Gauge namespace

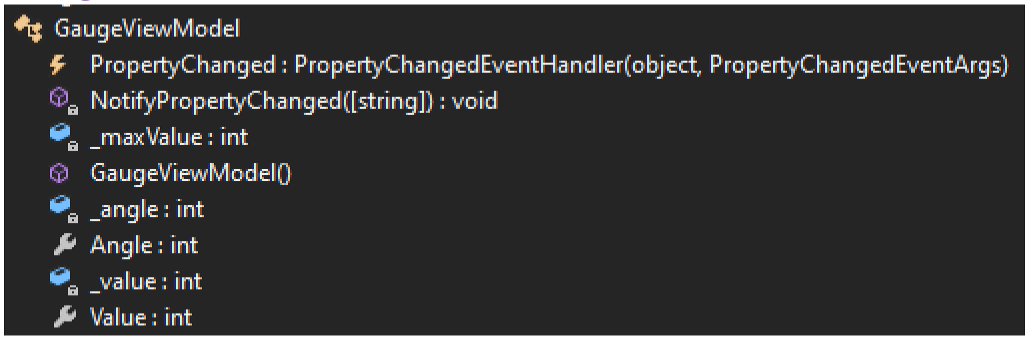
1. Gauge:

Custom-made UI control, used for displaying the usage rate of the RAM. For designing it, the WPF framework was used, as for all the visual classes in this project.



```
Gauge.xaml.cs
  Gauge
    Gauge()
    _contentLoaded : bool
    InitializeComponent() : void
    IComponentConnector.Connect(int, object) : void
```

2. GaugeViewModel



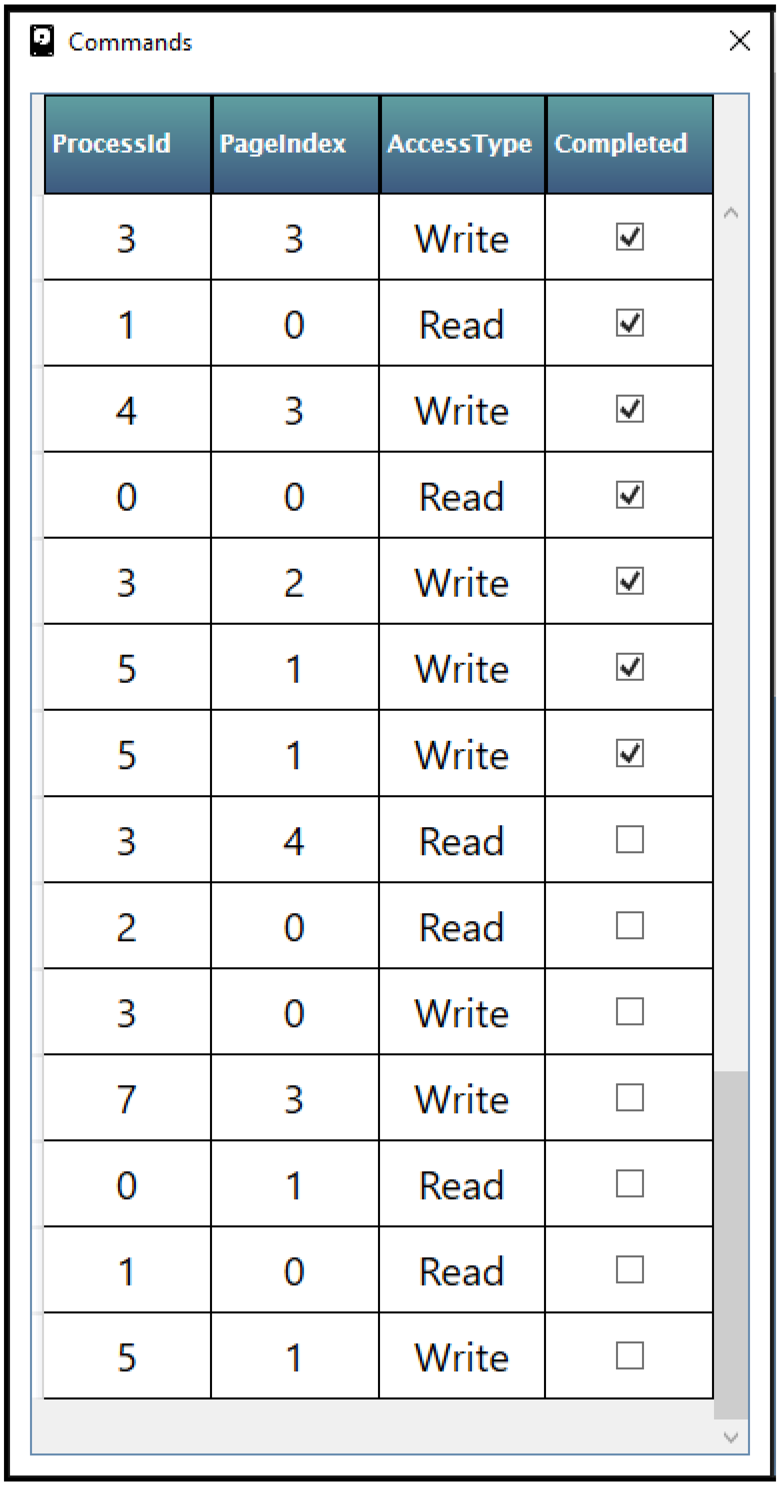
```
GaugeViewModel
  PropertyChanged : PropertyChangedEventHandler(object, PropertyChangedEventArgs)
  NotifyPropertyChanged([string]) : void
  _maxValue : int
  GaugeViewModel()
  _angle : int
  Angle : int
  _value : int
  Value : int
```

The object used for storing the data mapped to the gauge: the value to be displayed (ranging from 0 to 100%) and the angle the needle needs to rotate at, based on the value of the Value property. The **INotifyPropertyChanged** interface is implemented again: each time one of the two properties has changed the UI is updated accordingly based on the bound object.

D. The VirtualMemorySimulator, Windows namespace

1. CommandsInfo

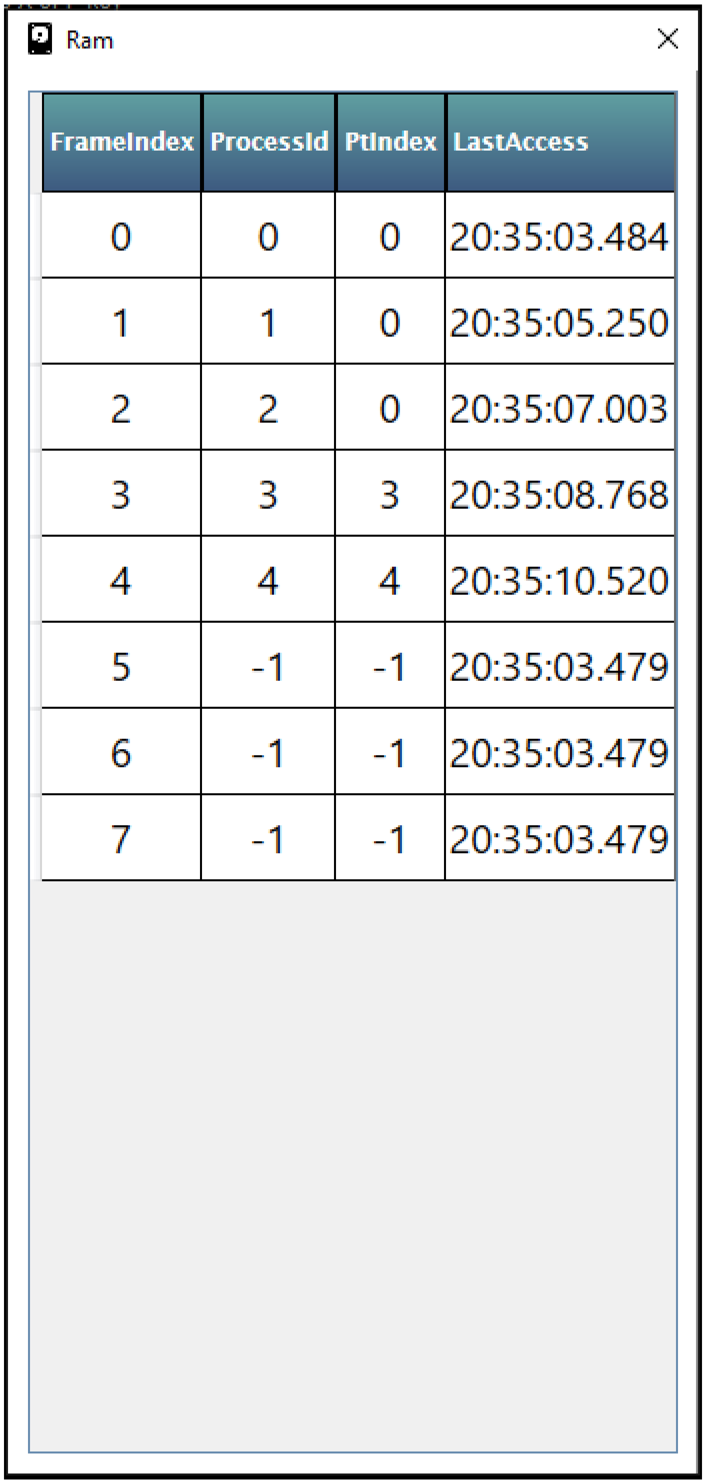
Displays the details about the commands from the simulation. The ProcessId column tells the process the command belongs to, the PageIndex columns stores the index in that process' Page Table of the page, the AccessType can be Read or Write and the Completed field is updated based on the events fired from the Machine project each time one of the commands have finished. They are executed in the order they are displayed in the grid.



ProcessId	PageIndex	AccessType	Completed
3	3	Write	✓
1	0	Read	✓
4	3	Write	✓
0	0	Read	✓
3	2	Write	✓
5	1	Write	✓
5	1	Write	✓
3	4	Read	☐
2	0	Read	☐
3	0	Write	☐
7	3	Write	☐
0	1	Read	☐
1	0	Read	☐
5	1	Write	☐

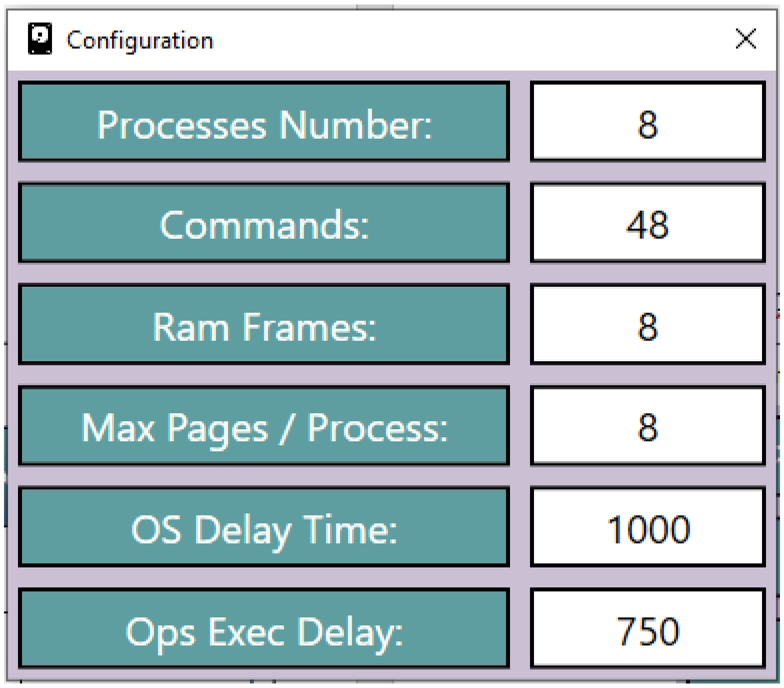
2. RamInfo

Displays the state of the RAM memory. The FrameIndex represents the index of the frame in the RAM, the ProcessId tells the pid of the process that frame belongs to, the PtIndex indicates the index of the page mapped to this process in its PageTable and the LastAccess stores the moment of the last access of the page in the "HH:mm:ss:fff" format.



FrameIndex	ProcessId	PtIndex	LastAccess
0	0	0	20:35:03.484
1	1	0	20:35:05.250
2	2	0	20:35:07.003
3	3	3	20:35:08.768
4	4	4	20:35:10.520
5	-1	-1	20:35:03.479
6	-1	-1	20:35:03.479
7	-1	-1	20:35:03.479

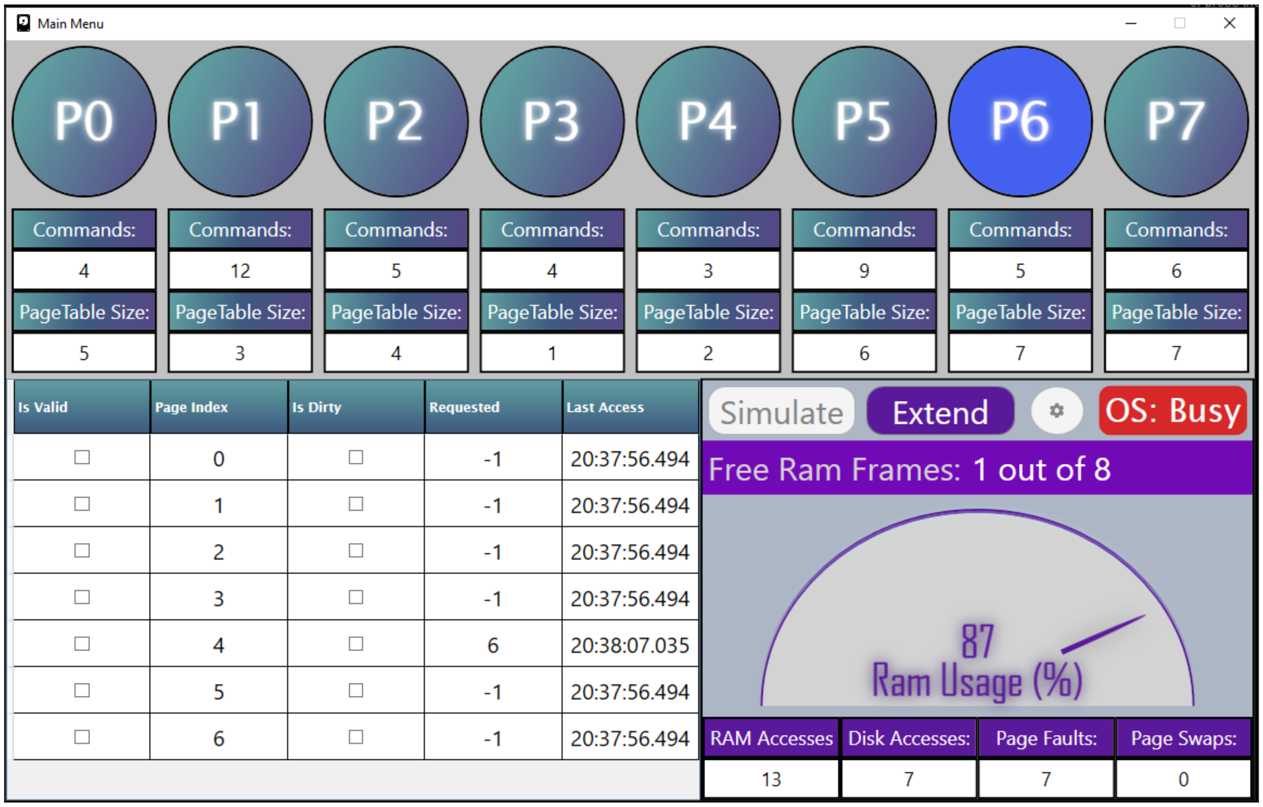
3. ConfigWindow


Configuration

Processes Number:	8
Commands:	48
Ram Frames:	8
Max Pages / Process:	8
OS Delay Time:	1000
Ops Exec Delay:	750

Displays the configurable parameters of the simulation: the process number sets the number of running processes (maximum 8). If any number provided is smaller than 8, the unused processes are disabled and greyed-out. The commands parameter represents the number of commands that will be run during the simulation. The RAM frames parameter indicates the number of frames the RAM will be divided into. Max pages / process sets a maximum number of pages each process can hold in its PageTable. The OS Delay Time is the value (in milliseconds) the execution will be delayed to simulate long times needed by the OS to access the Disk. The Ops Exec Delay is the time amount (in milliseconds) that the execution is delayed after each command. Introduced for simulating the time needed by OS to switch from a command to another and for allowing inter-commands tracking of the system. If any provided is not valid (non-numerical or over the maximum allowed value) the controls are reset to the previous values when they lose focus.

4. MainWindow



	P0	P1	P2	P3	P4	P5	P6	P7
Commands:	4	12	5	4	3	9	5	6
PageTable Size:	5	3	4	1	2	6	7	7

Is Valid	Page Index	Is Dirty	Requested	Last Access
☐	0	☐	-1	20:37:56.494
☐	1	☐	-1	20:37:56.494
☐	2	☐	-1	20:37:56.494
☐	3	☐	-1	20:37:56.494
☐	4	☐	6	20:38:07.035
☐	5	☐	-1	20:37:56.494
☐	6	☐	-1	20:37:56.494

Free Ram Frames: 1 out of 8

RAM Accesses	Disk Accesses:	Page Faults:	Page Swaps:
13	7	7	0

Displays most of the visual details about the simulation. The upper panel contains clickable circles for each active process. When clicked, the PageTable of the current process is displayed in the lower-left grid control. For each process, the number of commands it will execute and the allocated pages number are displayed. The grid contains data about the load state of the page (Is Valid), the index of the page in the PageTable of the selected process, the modified state (IsDirty), the Requested field (set to the id of the process when requested to be loaded in RAM but not loaded yet) and the last access moment of that page.

On the lower right panel, the Simulate button is active only before the simulation has started and after it finished. The Extend button toggles the visibility of the Ram and Commands Info windows, the Settings button displays the configuration window and the last label shows the status of the OS: Free when no simulation is running, Idle when not requested to access the Disk, Busy when asked to do so. We also display in numbers the free and the total number of frames from the RAM, and for a more dynamic visual effect, a gauge was used for fluctuating each time the RAM is loaded with a page. Finally, the parameters we want to count in the simulation are displayed at the bottom: the RAM accesses, the Disk accesses, the Page Faults and the page swaps.

6. Testing and Validation

For testing purposes, the following mechanisms were used:

- The OS state must be Free when no simulation is run, Idle when no action is required and the RAM state remains the same (no fluctuation on the gauge and the number of free RAM frames remains unchanged)
- All the pages loaded in the RAM and displayed in the RAM info window must have the IsValid field checked in their PageTable at any given moment
- When the gauge starts fluctuating, the IsValid field of the page that will be loaded stays set on the pid value until the gauge no longer fluctuates, the OS is idle, meaning that the page was loaded
- Because of the chosen replacement policy, at the end of each simulation the last n values displayed in the RAM info window must correlate with the last n distinct commands: the pid and the PageTable indexes must be the same, not necessarily in the same order
- At the end of each simulation, the difference between the Page Faults count and the Page swaps value must be equal to the number of RAM frames configured before the simulation
- The number of RAM accesses should be somewhere around the double value of Disk accesses: each time we load a page, we access both the Disk and the RAM, when swapping a page, the same event occurs. When reading or writing, the frame is also accessed. Depending on the number of times each page is accessed individually, the ration between the memories’ accesses may vary

Because the commands are randomly generated, the correctness of the program was ensured after running a relatively large number of simulations with different parameters (set in the configuration window). Using the above criteria, after meeting all the presented steps, the application’s correctness can be guaranteed.

7. Conclusions

The goal of the project was to simulate the Virtual Memory concept, used by nowadays computers to extend the size of the RAM memory even beyond its physical sizes, sometimes beyond the size of the Disk (the secondary memory). The simulation had to be performed and displayed to the user into an intuitive, responsive and suggestive interface.

The main reason I took this particular project was because of the lack of understanding I had around this subject, knowing that having to implement such a simulator would definitely help me understand the concepts related to this topic. Another reason for choosing this project was the possibility of writing it using C#, a programming language I personally enjoy using and learning new concepts about, which was also the case for the presented application.

The freedom of choosing what to display, how to display and actually the whole logic of the project, as long as it was related to the topic, made the development of this application as fun as it was challenging, finalizing a project I am personally proud of, due to the learnt and applied concepts, but also for all the knowledge accumulated after developing it, regarding both C# programming but also the Virtual Memory topic itself.

8. Bibliography

- https://en.wikipedia.org/wiki/Virtual_memory
- https://www.geeksforgeeks.org/virtual-memory-in-operating-system/
- https://en.wikipedia.org/wiki/Paging
- https://www.youtube.com/watch?v=2quKyPnUShQ [Great explanation of the Virtual Memory concept]
- https://www.youtube.com/watch?v=KElruOV2EfE [Gauge tutorial]