**Virtual memory graphic simulator**

1. **Introduction**

Virtual memory is a memory management technique through which the operating system is using secondary storage to simulate the idea of more physical memory.

There were two main reasons for which the idea of virtual memory was developed: to lift off the programmers’ burden of constantly taking care that their programs’ size was smaller than main memory’s dimension and for protection in case of memory sharing between multiple programs.

Virtual memory was developed at a time when physical memory was expensive and in short supply. As the programs became bigger and more complex, it was up to the programmer to make them fit inside the physical memory. Initially, one solution for making this possible was to divide the programs into parts which were mutually exclusive, called overlays, and load or unload them depending on the situation. The programmer had to ensure that the program never tried to access an overlay that was not loaded and that the overlays loaded never exceeded the total size of the memory. This responsibility was quite a burden on the programmers therefore virtual memory came as a relief since it automatically manages the two levels of memory represented by main memory and secondary storage.

Computers nowadays run more than one process at the same time which means that all these processes are simultaneously in the physical memory. This may generate some problems such as a program accessing the memory allocated for another program if not dealt with. The second motivation for creating the concept of virtual memory listed above was to solve this type of problems. Virtual memory provides memory protection by giving each program a virtual address space in which it can use as much memory as it wants. However, only a portion of that virtual address space is located in physical memory and programs can access only those pages that are mapped in their page table. In this way, a program cannot access another one’s physical pages because they are not mapped in its page table.

A chronological short history depicting the evolution that led to creating the virtual memory is showed below.

* **1959**: The Atlas Team at University of Manchester produced the first working prototype computer with a one-level storage system(virtual memory).
* **1960’s**: Popular commercial operating systems used virtual memory. Included were such computers as the IBM 360/67, CDC 7600, GE 645, RCA Spectra/70, Burroughs 6500.
* **1965**: In 1965, Maurice Wilkes invented the idea of fast "slave memory" to act as a buffer between the memory and the processor. It can hold a small number of recently used "data". Slave memory also included the hardware for address translation. Now called cache memory, it allows the system to run within a few percent of full processor speed. Cache memory is now standard in every computer today.
* **1966**: Peter Denning came up with an idea to solve thrashing by observing the locality of a program. He called this the working set.
* **1967-1975**: Experiments were performed to determine the nature of virtual memory and perfect it for stable use.
* **1969**: David Sayre's IBM research team proved that the operating system's set of automatic overlay strategies could outperform programmer's manual overlay strategies. This meant that virtual-memory controlled multiprogramming systems would be more efficient than manual-controlled systems.
* **1985**: In 1985 Intel offered virtual memory and cache in the 386 microprocessor and Microsoft offered multiprogramming in Windows 3.1
* **1995**: Microsoft included virtual memory in Windows 95.
  1. **Physical and Virtual Addressing**

The main memory of a computer is organized as an array of n contiguous byte-sized cells, each of them having a unique physical address. Consequently, physical addressing is the method through which the CPU is accessing memory. (Example diagram 1!)

This approach on accessing memory locations was used by early PCs, however nowadays computers use virtual addressing. (Example diagram 2!) With this new form of addressing, the CPU generates a virtual address which is converted to its equivalent physical address and sent further to main memory. This process of translating virtual addresses to physical addresses is performed by dedicated hardware, called Memory Management Unit (MMU), which uses a look-up table stored in main memory and whose contents are managed by the operating system.

Just like physical memory, virtual memory is also organized in byte-sized cells but stored on disk. Both, data on disk and on memory is partitioned into blocks that serve as the transfer units between the two level of memory. In virtual memory these blocks are called virtual pages, while the blocks from main memory are called physical pages, or page frames. (Example diagram 3?)

* 1. **Address Translation**

Address translation is a mapping between the elements of an n-element virtual address space and an m-element physical space. The n-bit virtual address has two components: a p-bit virtual page offset (VPO) and an (n-p)-bit virtual page number (VPN). The MMU uses the VPN to select the appropriate page table entry. The corresponding physical address is the concatenation of the physical page number from the page table entry and the virtual page offset from the virtual address.

* 1. **Page Table**

A page table is a data structure stored in physical memory that allows the CPU to determine which physical page is cached in. In case of a miss, the system scans the table to find out where on the disk is the referenced virtual page, it chooses a victim page in the memory and replaces it with a copy of the virtual page.

These actions can be executed through the combined capabilities of operating system software, MMU and the page table. The address translation hardware reads the page table each time it converts a virtual address to a physical address. The operating system is responsible for maintaining the contents of the page table and transferring pages between disk and RAM.

Each virtual page in the virtual address space maps one page table entry located at a fixed offset in the page table. The page table entry is composed of a valid bit and an n-bit address field. The valid bit indicates if the page is in RAM. If it is set, then the page is in the physical memory, otherwise it is on disk. A null in the address field means that the virtual page has not been allocated yet. (Example diagram 4)

* + 1. **Page Hit**

The address translation hardware uses the virtual address to locate a specific page table entry and read it from memory. If it is a page hit then the valid bit is set on 1, meaning that the page is cached in. The physical address is constructed and the word from that location is being read. (Example diagram 5)

* + 1. **Page Fault**

A page fault happens when the word referenced by the CPU is not being cached in RAM. From the valid bit, which is set on 0, the address translation hardware infers that the virtual page it tries to access is not in memory and it triggers a page fault exception. This exception invokes a pace fault exception handler in the kernel, which selects a victim page and copies from disk, in its place, the virtual page that the program needs. If the victim page has been modified before, the kernel copied it back to disk. To establish if the victim page was modified or not a dirty bit is used.

After the page from disk is copied in the victim page’s place, the handler returns and restarts the faulting instruction, the difference this time being the fact that the page needed by the program is cached in. (Example diagram 6)

* + 1. **Working Set**

“Although the total number of distinct pages that programs reference during an entire run might exceed the total size of physical memory, the principle of locality promises that at any point in time they will tend to work on a smaller set of active pages known as the working set or resident set. After an initial overhead where the working set is paged into memory, subsequent references to the working set result in hits, with no additional disk traffic.”

* + 1. **Thrashing**

If the program does not have good temporal locality and the working set size exceeds physical memory’s size then thrashing can appear, meaning that pages are swapped in and out continuously.

* + 1. **Swap**

Swapping (or paging) is the activity of transferring a page between disk and memory.

* + 1. **Demand Paging**

Demand paging is a strategy in which it is waited until the last moment to swap in a page.

* 1. **Translation Lookaside Buffer**

Every time CPU generates a virtual address, the MMU must refer to a page table entry to translate the virtual address in a physical address. In the worst case, this requires an additional fetch from the disk, when the virtual page searched for is not cached in. In order to reduce the costs a small cache for page table entries, called translation lookaside buffer, is included in the MMU.

1. **Objectives**

The main objective of the project is developing an application which will visually simulate how the virtual memory works by implementing some of its main actions, such as finding information, storing a page inside the main memory, replacing a page and so on. This process will imply a series of steps to be followed.

Step 1: Some initial settings are to be done in order to advance to the following steps. These settings include the storage capacity of both main memory and virtual memory, the size of a page, how many offset bits are to be used and the number of entries in the Translation Lookaside Buffer (TLB).

Step 2: An instruction is generated randomly. In order to fetch it from the physical memory the instruction is breakdown into the bits which form the virtual page number and the bits used for offset.

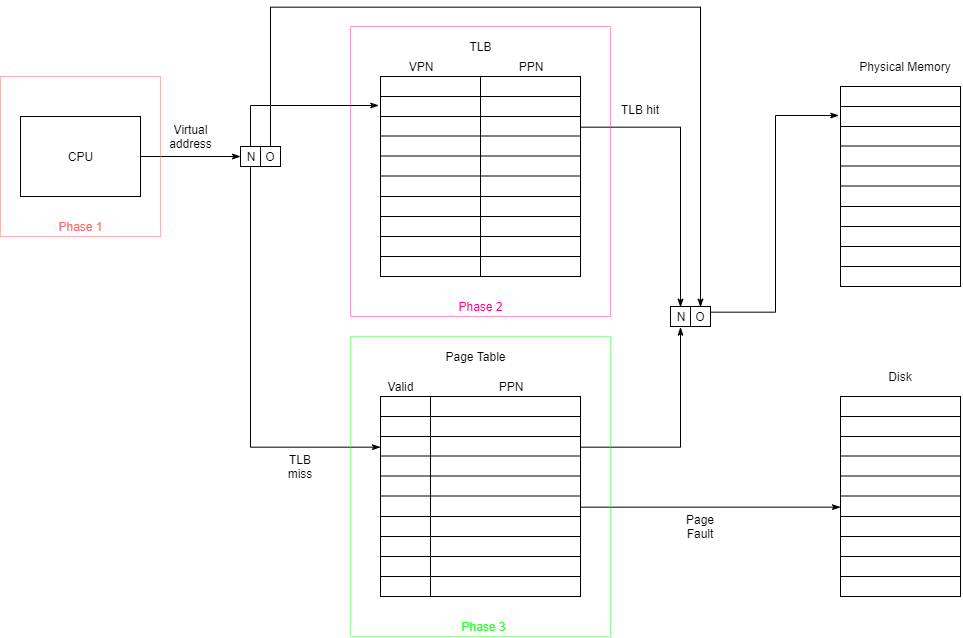
Step 3: The page is initially searched for inside the Translation Lookaside Table, in case it was used before. If the page is there the physical memory address is computed and used to access the physical page we need, otherwise the search is continued in the page table inside the main memory.

Step 4: If the page is inside the page table the TLB is simply updated and the last accessed location inside it is replaced by the new accessed location. However, if the page is not in the page table, which is called a page fault, that means that the CPU has to read the page from the disk and transfer it to main memory. Afterwards, we jump back at step 3 and redo the process.

For the purpose of creating such a virtual representation we will use Java as a programming language.

1. **Theoretical aspects**

For a better visualization of the system to be implemented, a scheme, depicting the architecture of the program, is provided and all the following explanations are based on it.



As it can be seen in the diagram, the main components of the system are the CPU, which generates the virtual address where the needed data resides, the translation lookaside buffer, the page table and the physical memory. These components are the main abstract data types to be used in simulating the virtual memory workings.

In the **first phase** of the project, a generator creates a random address, equivalent to the virtual address the CPU has at hand for locating data in main memory. This generator takes into consideration some constraints, such as the number of offset bits and the number of bits for the virtual and physical page number. With only these three details it is possible to deduce the rest of needed information.

The number of offset bits determines the page size. For example, with 12 bits allocated for offset, the program can jump from an address X to X + 2^12 – 1. All this space in which the program can jump starting from a base address is called a page. In the example above, a 12 bits offset implies a 4KB page of memory.

The number of bits designated for representing the page number for both memories determine their size. For a 20 bit page number, the size of the memory is 2^20, meaning 1MB. These computations are important since using them the number of pages that can be stored inside the virtual memory or physical memory can be determined easily by dividing the respective memory size with the size of a page. In the case above 2^20 / 2^12 (256) pages can be placed inside the memory with 20 bits for page numbers and 12 bits of offset. (Relevant for when we determine the max number of pages in each memory).

The **second phase** of the program revolves around the Translation Lookaside Buffer and the two cases that can be met while searching it. By using the most significant n bits of the virtual address that comes from the CPU, the TLB is scanned in order to determine if the virtual page number is stored inside the buffer. The two cases encountered are:

1. There is a TLB hit, meaning that those n most significant bits, which represent the virtual page number, have been found inside the table. The physical page number mapped onto the virtual page number is taken and concatenated together with the virtual address’ offset bits, resulting in the physical address from where the CPU is expecting data. Previously, it was mentioned that the offset bits in the virtual address are the same as those in the physical address and this happens since the pages from both memories have the same size.
2. The virtual page number that is being searched for in the TLB is not present, resulting in a TLB miss. What happens further is described in the next phase.

In the **third phase** the virtual page number is used as index to determine at which position in the page table the needed physical page number is found. However, there are two possible outcomes in this case, too.

1. The valid bit inside the page table is set to 1, meaning that the physical page number is mapped in the table and it can be used to get the physical address where the data needed by the CPU is found. This outcome implies some changes to be made inside the TLB, namely if there is room inside it, the physical page number found inside the page table is mapped to its corresponding virtual page number, otherwise one mapping inside the table has to be overriden. To decide how to replace the entries inside the TLB when it is full, the FIFO method is used.

In order to implement the FIFO mechanism a **global clock** is used and each element inside the TLB has to remember when it was stored inside the table. When the TLB is full, the entry with the smallest value for the clock value is being replaced with a new entry.

1. The valid bit inside the page table is 0, which means that the referenced virtual page is not inside the physical memory, but on disk. Consequently, it is simulated the operating system fetches the virtual page from disk, puts it into the main memory using a certain strategy and updates the page table with the physical page number and valid bit set on 1. After the page fault handling routine ends, the control returns to the faulted instruction.

It was mentioned above about a strategy that is used to decide which page is overriden when the physical memory is full and a virtual page has to be cached in. Two kinds of replacement strategies are being implemented: the **first in first out** strategy and the **least recent used** strategy.

In the first one, the page having the smallest clock value, is being replaced, while the second strategy suggests that each virtual page inside the physical memory registers the time it is modified or accessed and when the time comes for a new virtual page to be added to the full main memory, the page with the smallest clock value is replaced. In the second strategy the smallest clock value implies the page that has stayed unmodified the longest in contrast with the first strategy where the minimum clock value

When deleting a page from memory in order to bring a new page from disk it is needed to check the **dirty bit** to see if the page has been modified or not. In case it has been changed while it was in memory, it has to be copied on the disk and only afterwards deleted. Otherwise, if it has not been modified, then it can be simply deleted since the information in it is the same as the information on the disk.

**4. Implementation**

The project implements the MVC architecture, hence it is structured in three parts: model, view and controller.

1. The **model** part contains all the basic objects that describe the logic behind the project.
   1. The first part of the model package is the class AddressGenerator, which, just as its name says, generates an address based on the input offered by the user through the graphic interface. Since only a single object of type AddressGenerator is needed per program, this class has been designed to implement the singleton pattern. This design pattern ensures that the class which implements it that only a single instance object is instantiated thorough the whole execution of the program. Singleton design pattern is implemented by making the constructor private, declaring all attributes and methods static and being able to generate an instance of the class only once through method getInstance which checks first if an instantiated object already exists. In that case it returns the reference to that object. If there is no object instantiated then it calls the constructor.

In order to generate an address the number of bits for the address is needed. However, it would be more useful for further implementations if the user could provide the number of bits per virtual page number and the number of bits per offset. In this case, the class has to compute when it is instantiated on how many bits is the address which is generating (by summing the before mentioned parameters offered by the user).

After finding out the number of bits used by the address, the next step is to compute the maximum value that the address can have so that we can generate randomly an address. The idea is to keep shifting to the left and or-ing with one an int which is initially 1. For example, the maximum value for an address on three bits is composed like this:

Step 1: max = 001

Step 2: max = 010 → max = 011

Step 3: max = 110 → max = 111

Therefore, the maximum value that an address on three bits can have is 7. So, in order to generate a random address we use this maximum value and the class Random.

* 1. Class VirtualAddress is created by calling AddressGenerator and keeps track of some important information such as: virtual address, the number of bits used for virtual page number, the number of bits used for offset and the values for virtual page number and offset. The last two values are extracted from the address using the bit-related information received in constructor. The extraction of the offset is done exactly like the computation of an address’ maximum value. However, to extract the page number is a bit different in the fact that this time we work with the MSB and shift to right.
  2. VirtualPageNumber is not a complex class but it is the basic unit that the program is working with since it is used in TLB and physical memory, too. hashCode and equals methods have been overriden in order to be able to implement a hashtable based on VirtualPageNumber.

The clock variable has been declared for the time when the Translation Lookaside Buffer is full and a page has to be replaced. Using the First-In-First-Out technique and based on the clock the page that has been the longest in the TLB is replaced.

* 1. The PhysicalPageNumber class is mainly designed to provide objects which serve as entries in the page table because it stores the physical page number value and the valid bit.

Just as in the case of the VirtualPageNumber class, the equals and hashCode methods have been implemented in order to implement a hashmap of PhysicalPageNumber objects and PhysicalMemoryPage objects.

* 1. TranslationLookasideBuffer’s core element is the hashmap of virtual page numbers and physical page numbers. This element simulates as close as possible the reality in which the physical page number are mapped to virtual page numbers.

In order to keep the size of the hashtable the same as the one received as parameter in the constructor the load factor is chosen to be 1. This means that only when the hashtable receives its size+1 entry it will expand.

To decide which entry already in the table is overriden when the TLB is full and another entry needs to be added, the FIFO strategy is used. To implement this strategy the VirtualPageNumber objects need to store a variable called clock which remembers at which time they were added in the TLB. The FIFO strategy is implemented using a method which determines the VirtualPageNumber with the minimum clock, meaning the least recent page to be added in the TLB and also the page which is overriden.

* 1. The PageTableEntry is used to implements the PageTable hashmap. It contains the PhysicalPageNumber which describes where in the physical memory is the location described by the generated address and the valid bit which indicates if the location is still in the physical memory or it has been moved back to hard disk.
  2. Class PageTable is implemented using a hashMap with VirtualPageNumber as key and PageTableEntry as value. Methods created include isHit, which tests if the virtual page is inside the page table, getPhysicalPageNumber, which returns the physical page number corresponding to the given virtual page number, addPageTableEntry, which is more like an update of the TLB because it adds a new entry if it is not present in the page table or it updates a present one if the value given as parameters already exists in the page table. The last method, computeSize, computes the size of the page table using the virtual page number bits given as argument.
  3. PhysicalMemoryPage, just as PageTableEntry, has been introduced in order to implement the physical memory as a hashtable. All the information needed from a page inside the physical memory is stored in this class, namely the virtual page number, the dirty bit and the clock.

It is noticeably that there is a clock variable inside two different classes, VirtualPageNumber and PhysicalMemoryPage. The reason behind this is the fact that the Tranlsation Lookaside Buffer is implemented using the FIFO technique and a clock indicating which virtual page has been the longest inside the TLB is necessary inside the respective class. On the other side, the PhysicalMemoryPage needs a clock inside too, since the physical memory has a replacing technique on its own – FIFO or LRU. There is the need to keep track of how long the pages have been inside the memory or how long it has been since they were last accessed. The concept of dirty bit is used for the last aspect mentioned before – keeping track if the page has been modified while in memory or not. A dirty bit set on true requests that the page is first copied inside the hard disk and only after that erased from memory.

* 1. Just like the TLB and the page table, physical memory is also built on the concept of a hashtable using PhysicalPageNumber as key and PhysicalMemoryPage as value. The main method inside this class is updatePhysicalMemory which basically goes through the following steps:

1. Verifies if the virtual page is already inside the physical memory. If it is true, based on what strategy is selected by the user to describe the overriding method of the physical memory, it modifies either the dirty bit of the entry or both the dirty bit and the clock.
2. In case the above condition is evaluated to false, it is deduced that the virtual page has to be added inside the physical memory because it does not contain it. In this case there are two possible scenarios: the memory is full and the memory has enough space to store another virtual page.
3. When the memory is full the strategy chosen for overriding has to be taken into consideration. However, when implementing the functionality for this part of the method there are not two cases since in both cases the virtual page with the smallest clock value has to be replaced. The only difference between strategies is when the clock is modified, when the page is added inside the memory or even when the page is accessed.
4. The other case is when there is enough space for adding another page.

Methods containsVpn, getVpnSpecificPmp, getVpnSpecificPpn and getDirtyBitStatus are implemented based on the same principle – iterating over the keys of the hashtable and checking if the value specific to that entry has some qualities that are needed or some values specific to the value are simply returned.

ComputeSize does exactly what its name suggests, it computes the size of the main memory based on the physical page number bits and the findMinClock chooses the physical page number with the lowest clock value.

* 1. The class which is responsible of manipulating all the logic components mentioned above into doing what a virtual memory does is the Manager class. It supervises all needed information like the number of bits of specific components, the tables and other functionalities.

Each method inside the Manager class corresponds to a state or helps implementing one of the states. Most methods are just one layer closer to the controller layer from the logic layer. One method which is not in this category is verify. This method looks at the page table and at the physical memories and check is there are differences, meaning that it checks if some pages are not in the memory anymore and it updates the page table by setting the valid bit to false.

1. The **view** component implements the part with which the user is interacting. From this stage all the information provided by the user has to go to the logic unit. This is possible by means of the last component.
2. The **controller** part is the element which links the view with the logic. Any request that comes from the user is sent to the logic, processed and then sent back to the user as an answer.

The Controller class is mostly comprised of initialization of action listeners for buttons and validating the input from the user. For the settings submit button initializer it takes the input from the user and converts it in objects or values that can be understood inside the logic layer.

Following, an address is generated or specified by the user. The submit button has to be pressed for the application to work. When the address is submitted the internal clock is incremented too.

Initializing the next button is structured in a set of states:

1. **case** ***TLB\_SEARCH***:

It is verified if the page is already in the TLB. If it is the next state is ***TLB\_HIT***, otherwise it is ***TLB\_MISS***.

1. **case** ***TLB\_HIT***:

The physical memory is updated – the dirty bit is modified and in case the strategy selected by the user is LRU then the clock which establishes when was the page last accessed is updated too.

Following, the physical memory table is updated.

1. **case** ***TLB\_MISS***:

The next state is ***PAGE\_TABLE\_SEARCH***.

1. **case** ***PAGE\_TABLE\_SEARCH***:

There it is established if the page is inside the page table or not. Depending on the result the next state can be ***PAGE\_TABlE\_HIT*** or ***PAGE\_TABLE\_MISS***.

!!!!Contains + valid bit.

1. **case** ***PAGE\_TABLE\_HIT***:

The physical memory and the TLB are updated. In physical memory the dirty bit and the clock – depending on the strategy – are modified, while inside the TLB the update consists of adding the page.

1. **case** ***PAGE\_TABLE\_MISS***:

The next state is ***UPDATE\_PHYSICAL\_MEMORY*** which will imply updating the page table and the TLB in order.

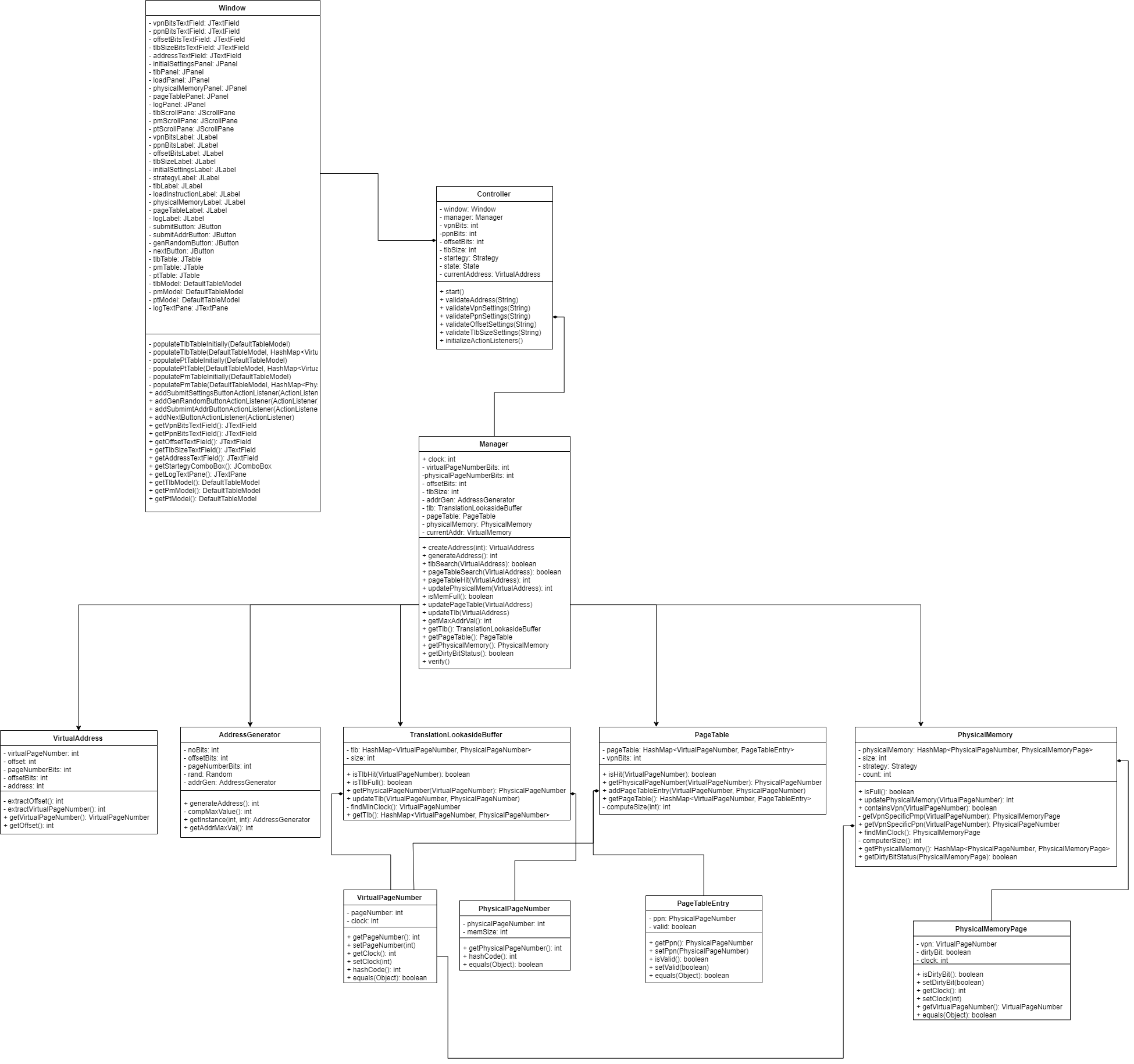
1. **case** ***UPDATE\_TLB***:
2. **case** ***UPDATE\_PAGE\_TABLE***:
3. **case** ***UPDATE\_PHYSICAL\_MEMORY***:

Besides updating the physical memory and establishing the next state, inside this state the status of the dirty bit is determined only if the memory is full, because only then overriding and moving pages back to hard disk is necessary.

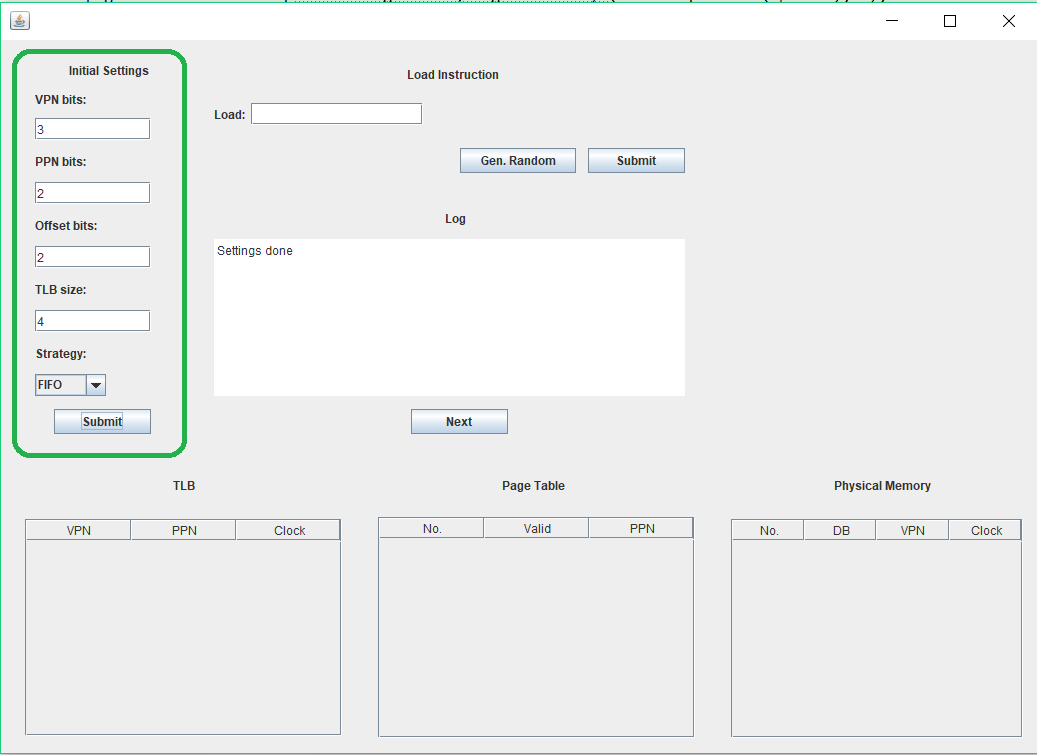
1. **case** ***END***:

This state is just a marker that the sequence of states for this address has ended.

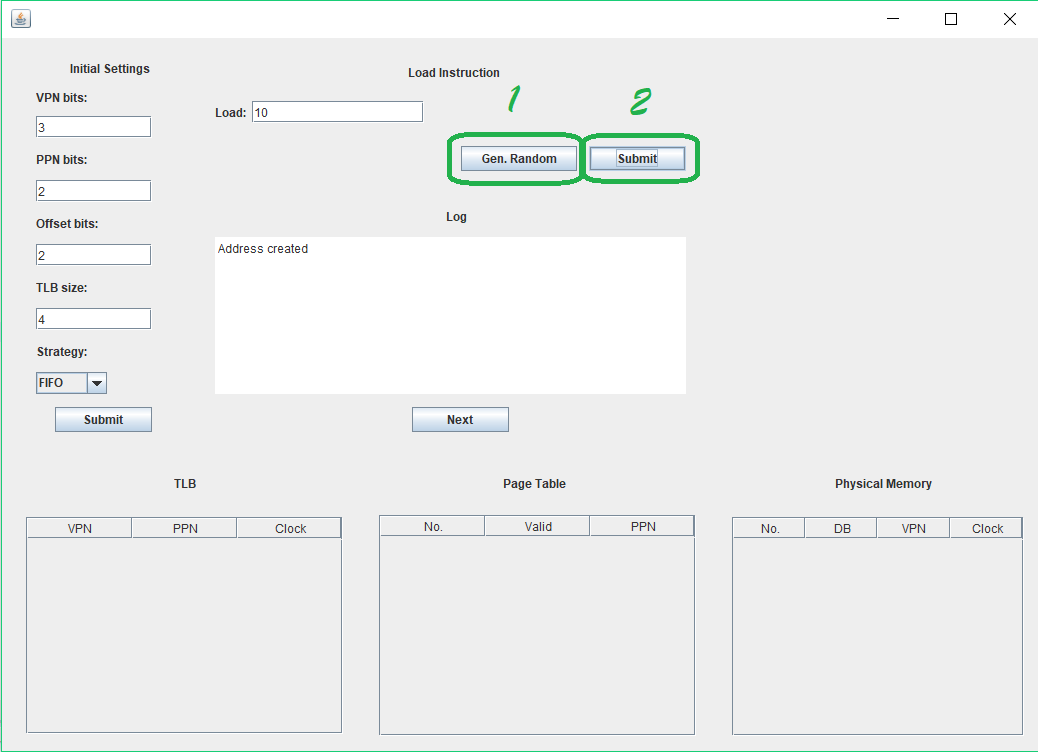
**UML diagram**



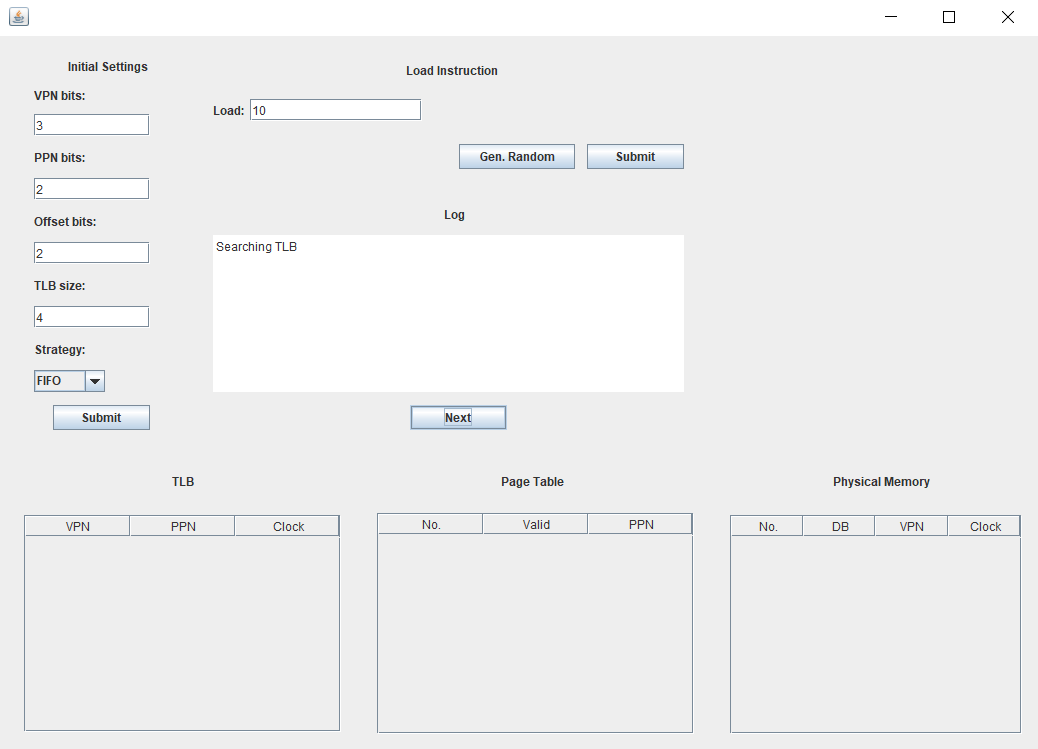
**5.Testing**



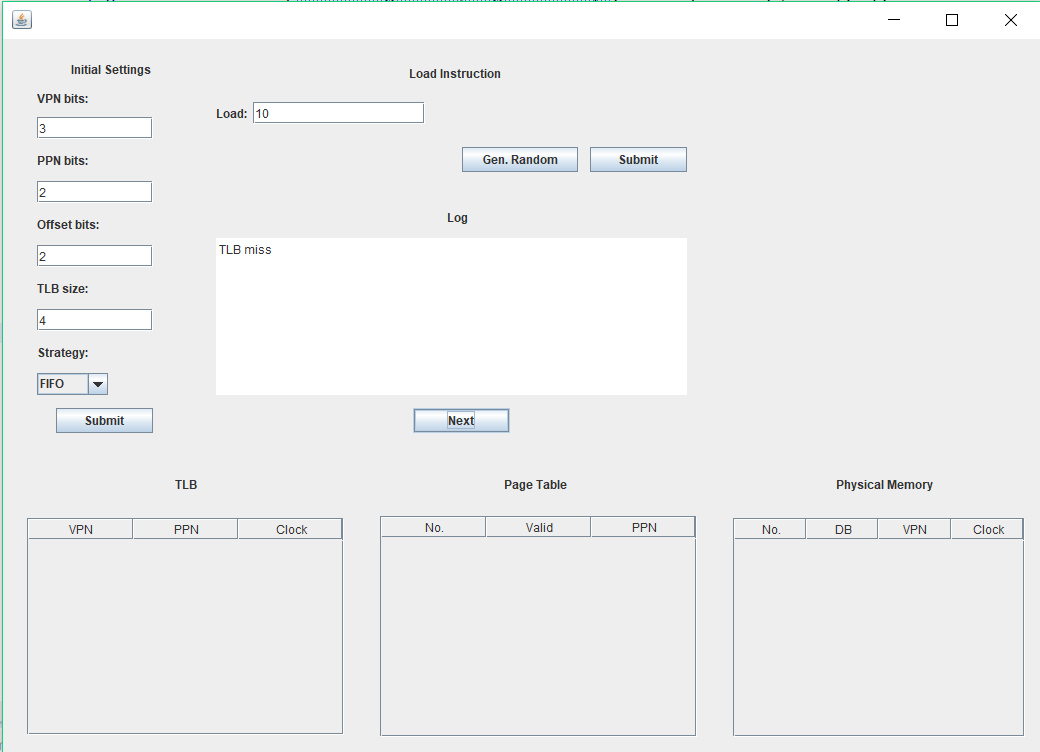
The first thing that has to be done to be able to use the virtual memory graphic simulator is to introduce the specifications for the virtual memory. Without doing this before going any further the application throws an error. The settings introduced in this section determine the dimensions of the components (TLB, page table, physical memory).



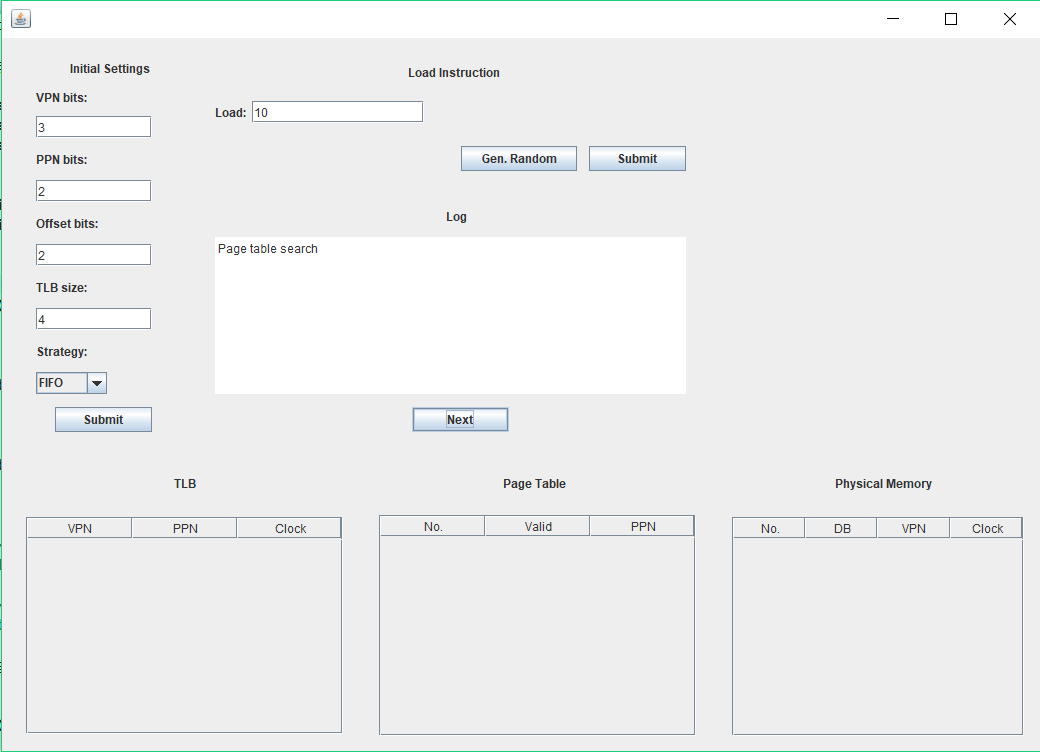
The user has the choice of getting a randomly generated value or to introduce one. A message is displayed on the log console in order to give an update of the state of the workflow.



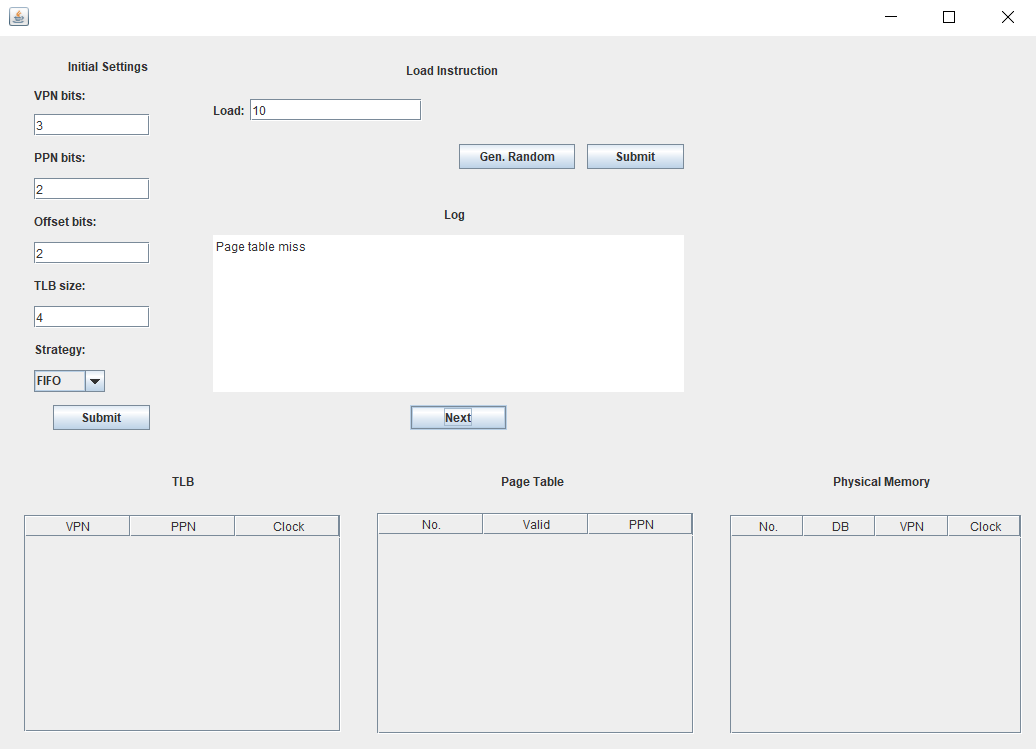
First, the program seraches inside the TLB in order to see if the virtual page is there.



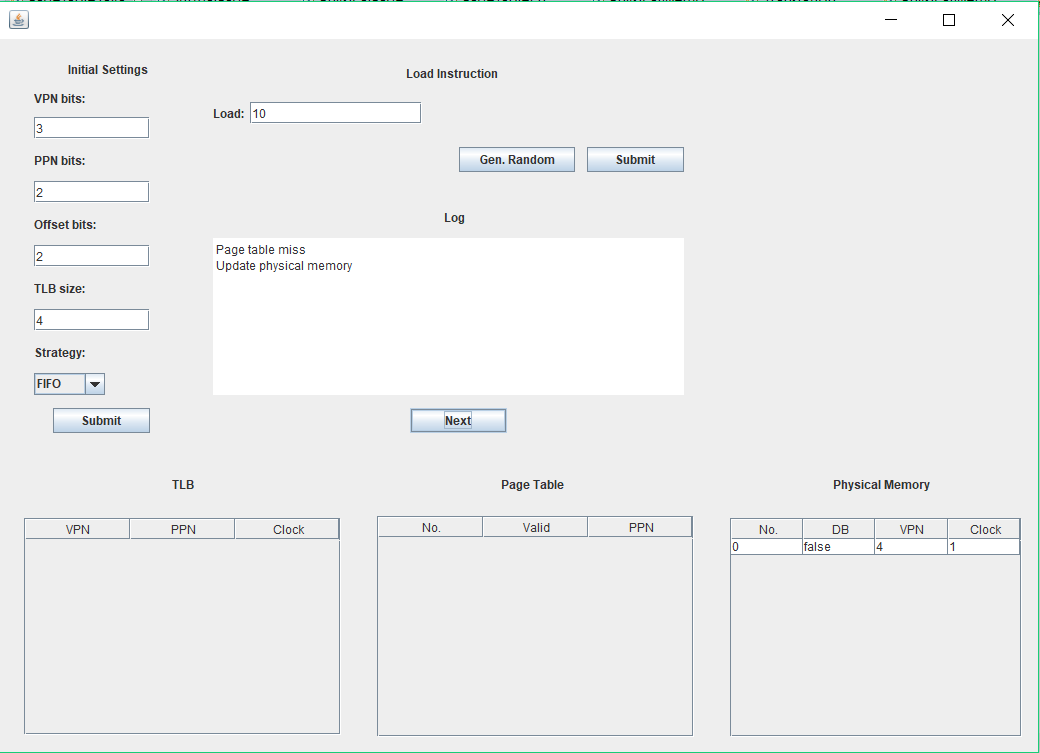
In the case exemplified, the virtual page is not inside the TLB, hence the „TLB miss” message inside the log console.



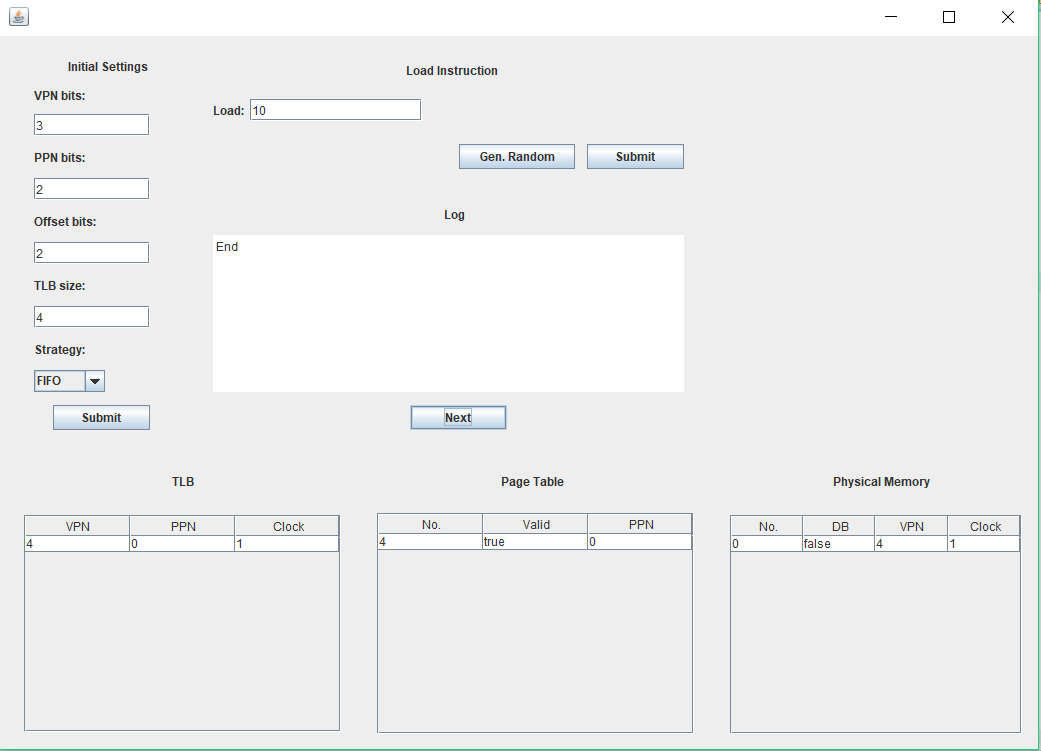
After a TLB miss there is a Page Table search which results in miss.



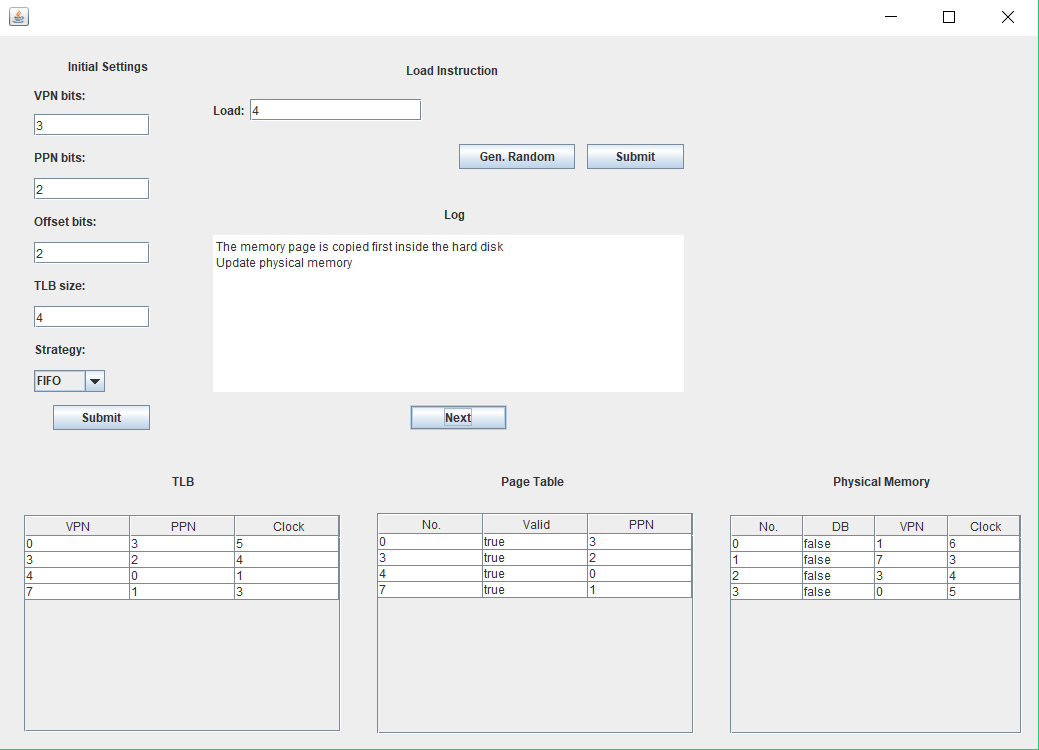
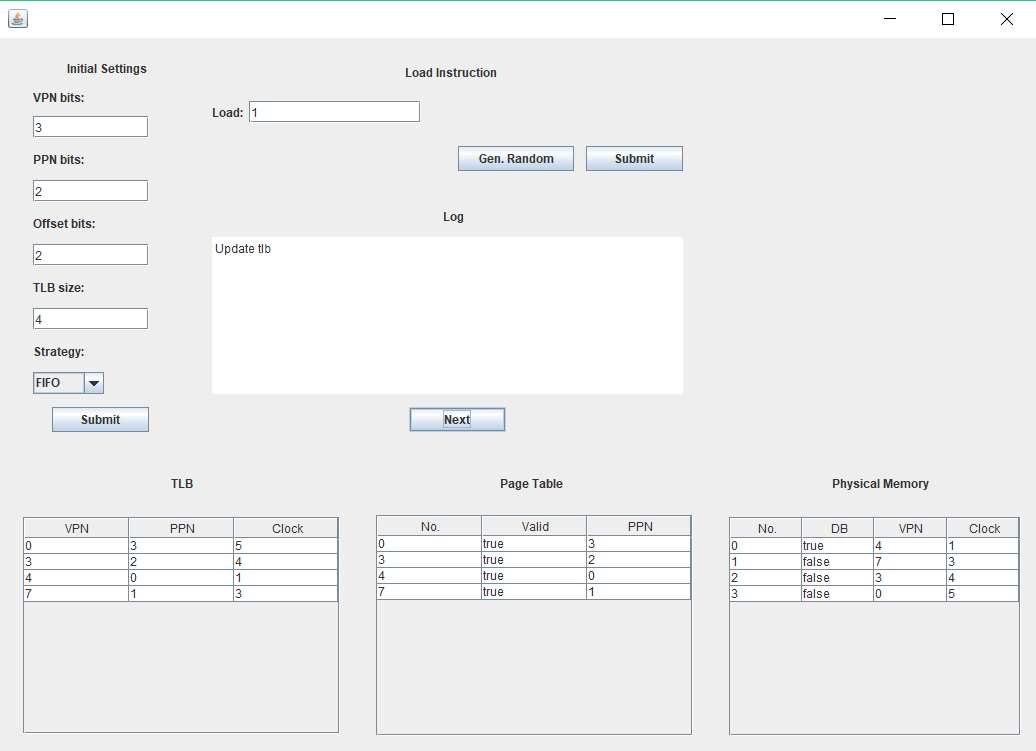
Afterwards the physical memory is checked. The virtual page is not there either, so it is brought inside the main memory from the disk.



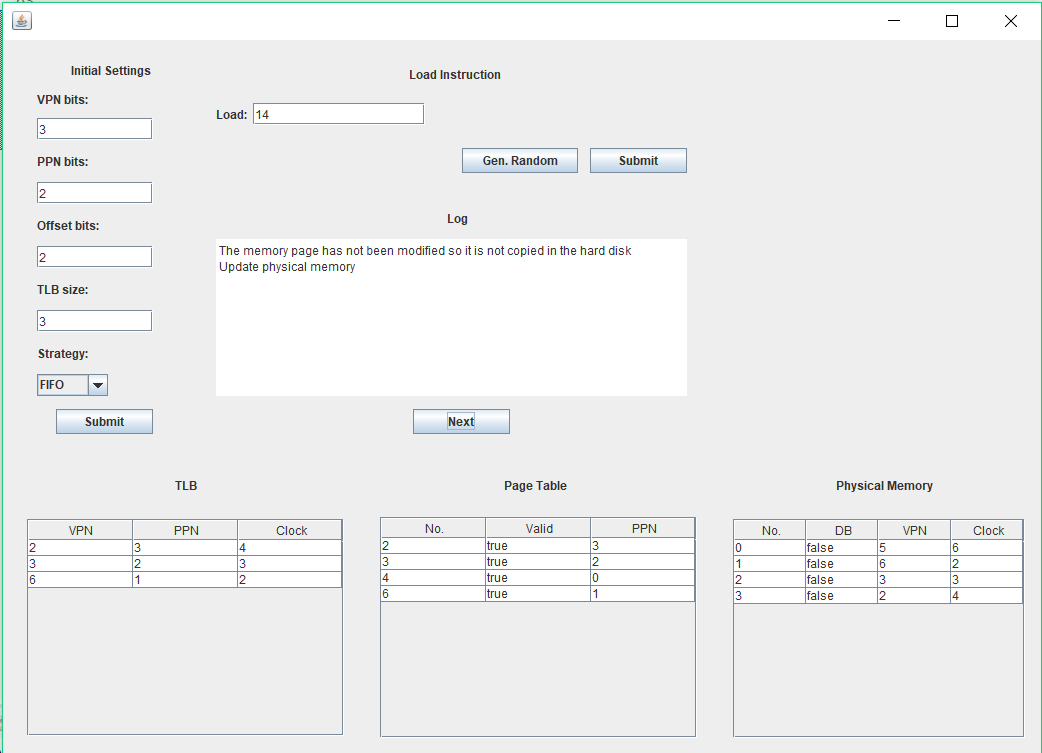
When the main memory is updated all the other tables are updated, too.



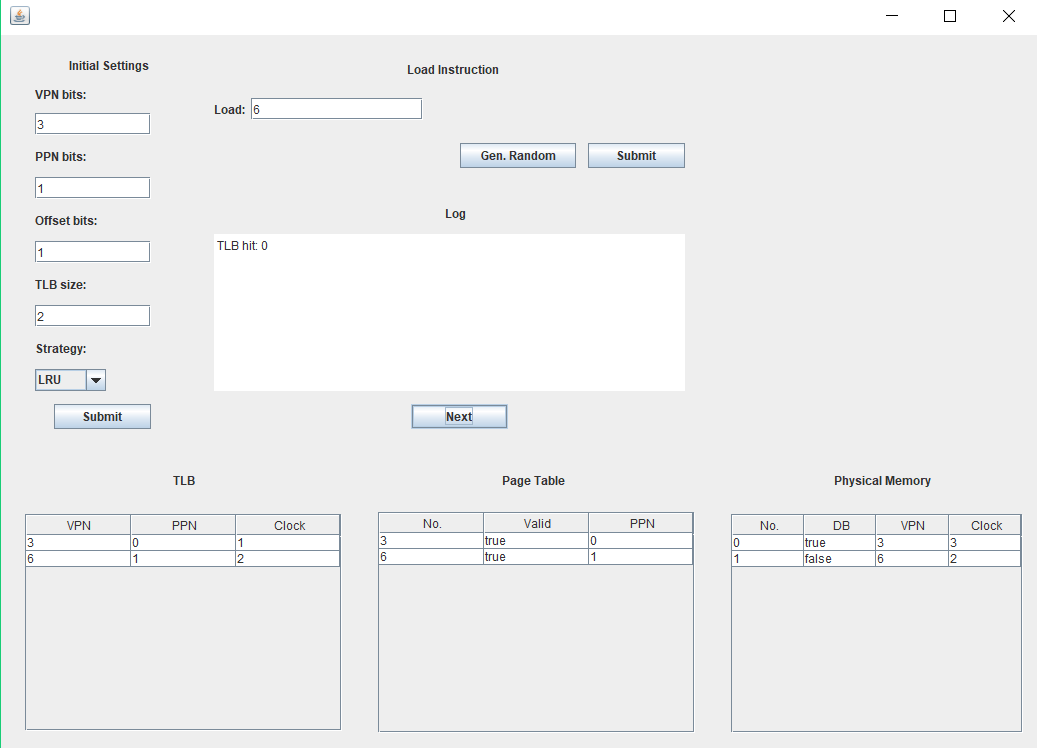
When a hit happens, depending on the startegy selected in the Initial Settings, different outcomes happen. In the case above, in which the FIFO strategy is selected, only the dirty bit is modified when a hit occurs.



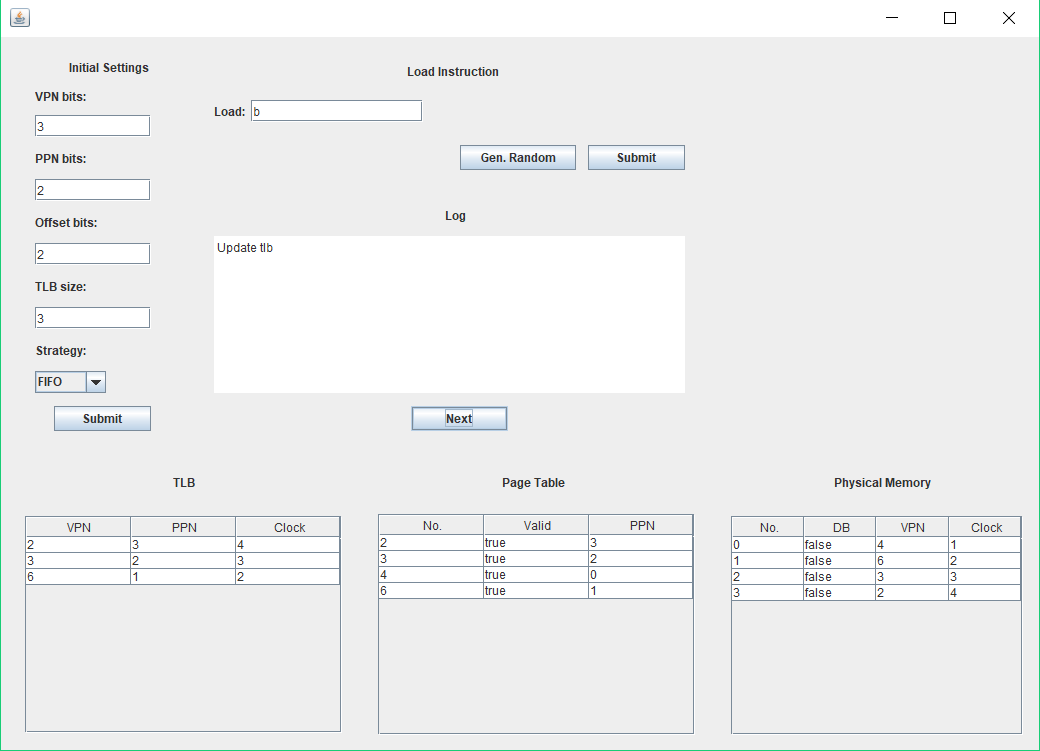
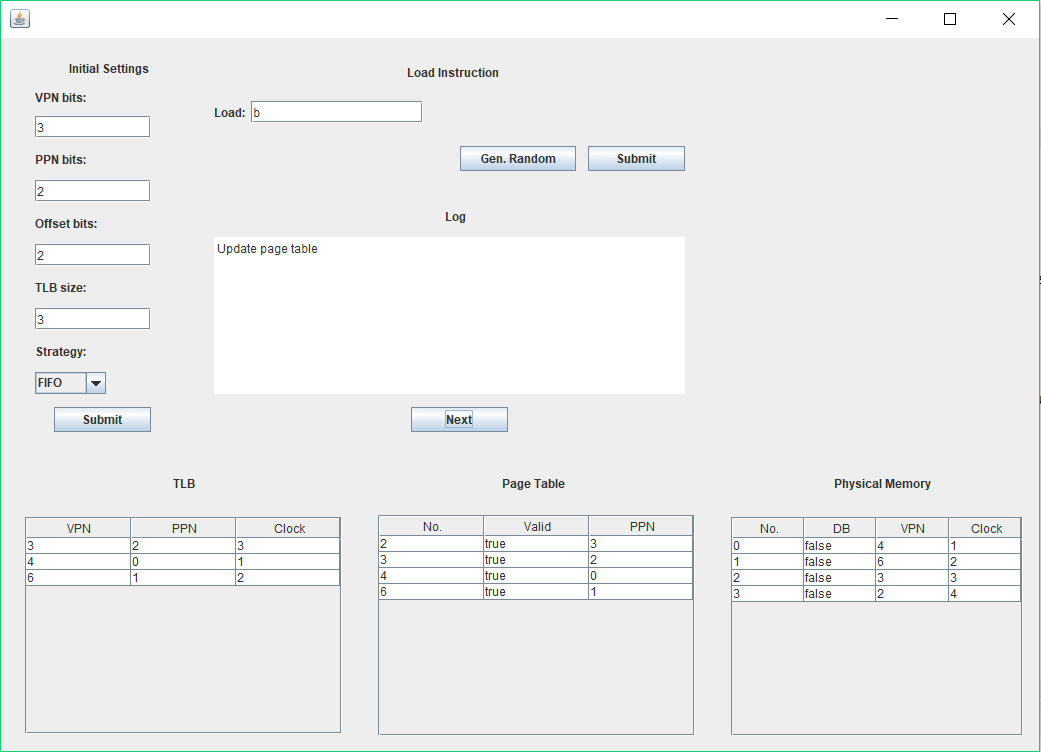
Depending on the dirty bit value of a page that is meant to be replaced the CPU decides if the page inside the main memory is first copied inside the hard disk or not. If the page has been modified, therefore having its dirty bit set to true, the message „The memory page is first copied inside the hard disk” is displayed inside the log console.



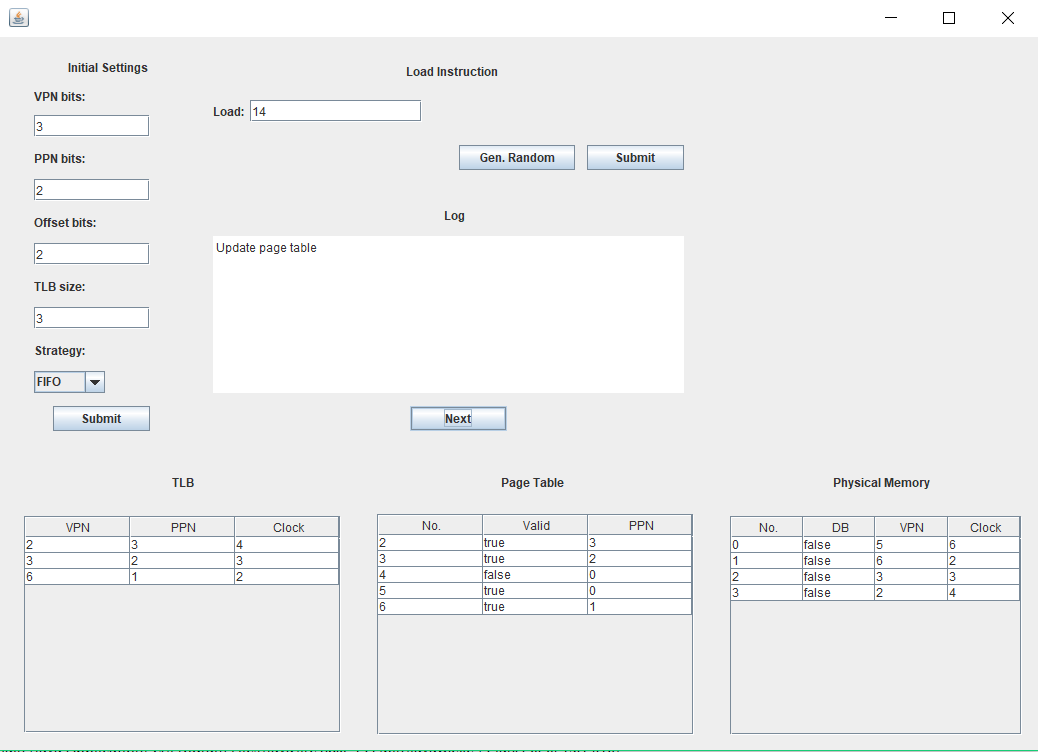
The other case is when the page has not been modified so there is no need to copy it inside the hard disk just as it is indicated by the log console message – „The memory page has not been modified so it is not copied in the hard disk”.



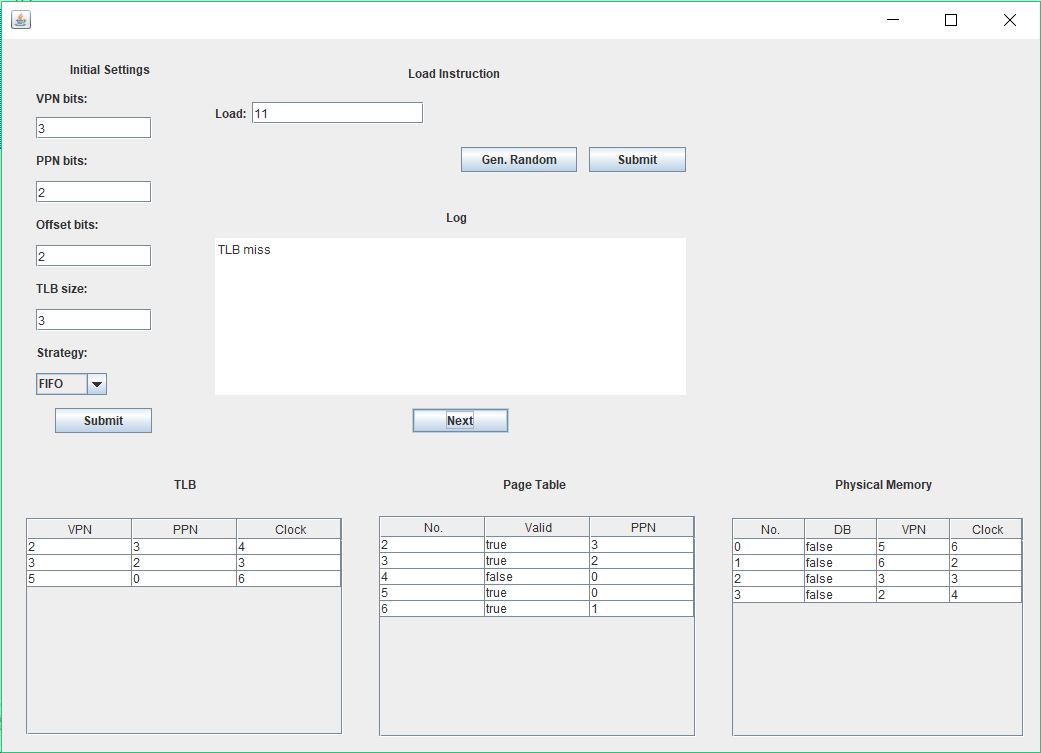
When the other replacement strategy, Least Recently Used, is selected, when a hit occurs, not only the dirty bit is modified, but also the clock, which in this case keeps track of the chronological order in which the pages have been accessed.



The TLB also has a replacement strategy, but that is by default FIFO.



In case a page is taken out of the main memory, the page table still keeps a reference to that certain virtual page. However, in order to be able to distinguish between pages in memory and those that are not, it sets the valid bit to false for those that are no longer in memory. This situation can be seen in the image above.



Address 11H is inside the virtual page that has been taken out from the memory. When the CPU needs data from this address it results in a miss followed by an update of the main memory, the page table - where the valid bit is set to true and the physical page updated to where the address can be found inside the memory - and the TLB.

**6.References**

1. Computer Organization and Architecture by Wiliam Stalings
2. Digital Design and Computer Architecture by David Harris and Sarah Harris
3. <https://www3.ntu.edu.sg/home/smitha/ParaCache/Paracache/vm.html>