EOPSY LABORATORY REPORT

TASK 4

PAGE REPLACEMENT EXERCISE

OMER AMAC

323796

Date : 06.05.2022

Contents

1. INTRODUCTION

A. Memory Management

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

A.1 Partitioing in Memory Management

Memory partitioning is the system by which the memory of a computer system is divided into sections for use by the resident programs. These memory divisions are known as partitions. There are different ways in which memory can be partitioned: fixed, variable, and dynamic partitioning.

A.2 Paging in Memory Management

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non – contiguous.

A.3 Segmentation in Memory Management

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

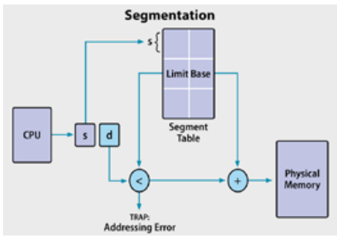


Figure 1:Segmentation Visuallzation

B. Different Ways of Memory Management[1]

B.1. Fixed Partitioning

This partitioning approach divided into a fixed number of partitions just one process can be loaded into one partition at the same time. Strengths of this approach: easy to implement it and slandered method as a partitioning solution. Weaknesses of this approach insufficient use because of the internal fragmentation, must know the maximum number of active processes can run is fixed size of the task is limited to largest partition size, degree of multiprogramming limited by the number of partitions, memory is wasted in the partition, must translate relative address to physical address.

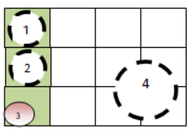


Figure 2: Fixed Partitioning Visualization

[1] International Journal of Scientific & Engineering Research, Volume 7, Issue 4, April-2016

B.2 Dynamic Partitioning

Partitions are created dynamically, each process loaded into a partition is exactly have same size as the process. Strengths of this approach are, ensure more efficient use of the main memory and no internal fragmentation. Weaknesses of this approach are inefficient use of processor because of the need for compaction and external fragmentation.

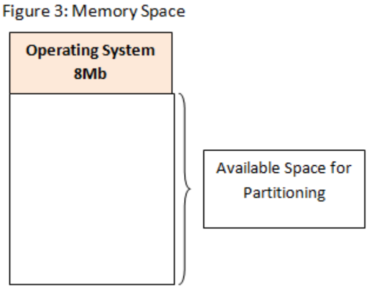


Figure 3: Dynamic Partitioning Visualization

B.3 Simple Paging

Strengths of this approach are, no need for external fragmentation, transfers between disks can be at the granularity of individual pages. Weaknesses of this approach are: maybe there is no correspondence between page protections. Settings and application data structures, requiring per process page tables, usually operating system need more storage for its internal data structures.

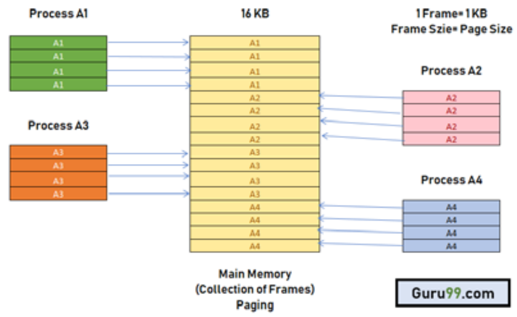


Figure 4: Paging Visualization

B.4 Simple Segmentation

Strengths of this approach is no internal fragmentation. Weaknesses of this approach are reduce the overhead compared to dynamic partitioning approach and improved the memory utilization.

B.5 Virtual-Memory Paging

Strengths of this approach are, having large virtual address space, no external fragmentation and higher degree of multi programming. Weaknesses of this approach is overhead coming from the complex of memory management.

B.6 Virtual-Memory Segmentation

Strengths of this approach are, it supports a high level of multiprogramming especially the enormous virtual address space and no internal fragmentation. Weaknesses of this approach is the overhead of complex memory management. [2]

---

[2] International Journal of Scientific & Engineering Research, Volume 7, Issue 4, April-2016

C. Page Replacement Algorithms

The page replacement algorithm decides which memory page is to be replaced. The process of replacement is sometimes called swap out or write to disk. Page replacement is done when the requested page is not found in the main memory (page fault).

There are two main aspects of virtual memory, Frame allocation and Page Replacement. It is very important to have the optimal frame allocation and page replacement algorithm. Frame allocation is all about how many frames are to be allocated to the process while the page replacement is all about determining the page number which needs to be replaced in order to make space for the requested page.[3]

FIFO Algorithm: In this algorithm, a queue is maintained. The page which is assigned the frame first will be replaced first. In other words, the page which resides at the rare end of the queue will be replaced on the every page fault.

Optimal Page Replacement Algorithm: In this algorithm, pages are replaced which would not be used for the longest duration of time in the future. Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

Least Recently Used: In this algorithm, page will be replaced which is least recently used.[4]

2. CONTENT OF EXPERIMENT

A. Task Description

Create a command file that maps any 8 pages of physical memory to the first 8 pages of virtual memory, and then reads from one virtual memory address on each of the 64 virtual pages. Step through the simulator one operation at a time and see if you can predict which virtual memory addresses cause page faults. What page replacement algorithm is being used? Locate in the sources and describe to the instructor the page replacement algorithm.

B. Environment and Required Files[5]

B.1 MOSS (Memory Management Simulator)

The memory management simulator illustrates page fault behavior in a paged virtual memory system. The program reads the initial state of the page table and a sequence of virtual memory instructions and writes a trace log indicating the effect of each instruction. It includes a graphical user interface so that students can observe page replacement algorithms at work. Students may be asked to implement a particular page replacement algorithm which the instructor can test by comparing the output from the student's algorithm to that produced by a working implementation.

The program reads a command file, optionally reads a configuration file, displays a GUI window which allows you to execute the command file, and optionally writes a trace file.

[3] https://www.javatpoint.com/os-page-replacement-algorithms
[4] https://www.geeksforgeeks.org/page-replacement-algorithms-in-operating-systems/
[5] 2001, Prentice-Hall, Inc.

The buttons:

Button	Description
`run`	runs the simulation to completion. Note that the simulation pauses and updates the screen between each step.
`step`	runs a single setup of the simulation and updates the display.
`reset`	initializes the simulator and starts from the beginning of the command file.
`exit`	exits the simulation.
`page n`	display information about this virtual page in the display area at the right.

The informational display:

Field	Description
`status:`	RUN, STEP, or STOP. This indicates whether the current run or step is completed.
`time:`	number of "ns" since the start of the simulation.
`instruction:`	READ or WRITE. The operation last performed.
`address:`	the virtual memory address of the operation last performed.
`page fault:`	whether the last operation caused a page fault to occur.
`virtual page:`	the number of the virtual page being displayed in the fields below. This is the last virtual page accessed by the simulator, or the last `page n` button pressed.
`physical page:`	the physical page for this virtual page, if any. `-1` indicates that no physical page is associated with this virtual page.
`R:`	whether this page has been read. (1=yes, 0=no)
`M:`	whether this page has been modified. (1=yes, 0=no)
`inMemTime:`	number of ns ago the physical page was allocated to this virtual page.
`lastTouchTime:`	number of ns ago the physical page was last modified.
`low:`	low virtual memory address of the virtual page.
`high:`	high virtual memory address of the virtual page.

B.2 Configuration File

The configuration file memory.conf is used to specify the the initial content of the virtual memory map (which pages of virtual memory are mapped to which pages in physical memory) and provide other configuration information, such as whether operation should be logged to a file.

The memset command is used to initialize each entry in the virtual page map. memset is followed by six integer values:

1. The virtual page # to initialize
2. The physical page # associated with this virtual page (-1 if no page assigned)
3. If the page has been read from (R) (0=no, 1=yes)
4. If the page has been modified (M) (0=no, 1=yes)
5. The amount of time the page has been in memory (in ns)
6. The last time the page has been modified (in ns)

The first two parameters define the mapping between the virtual page and a physical page, if any. The last four parameters are values that might be used by a page replacement algorithm.

For example, memset 34 23 0 0 0 0

specifies that virtual page 34 maps to physical page 23, and that the page has not been read or modified.

Note:

- Each physical page should be mapped to exactly one virtual page.
- The number of virtual pages is fixed at 64 (0..63).
- The number of physical pages cannot exceed 64 (0..63).

- If a virtual page is not specified by any memset command, it is assumed that the page is not mapped.

Keyword	Values	Description
`enable_logging`	`true` `false`	Whether logging of the operations should be enabled. If logging is enabled, then the program writes a one-line message for each READ or WRITE operation. By default, no logging is enabled. See also the `log_file` option.
`log_file`	*trace-file-name*	The name of the file to which log messages should be written. If no filename is given, then log messages are written to stdout. This option has no effect if `enable_logging` is `false` or not specified.
`pagesize`	*n* `power` *p*	The size of the page in bytes as a power of two. This can be given as a decimal number which is a power of two (1, 2, 4, 8, etc.) or as a power of two using the `power` keyword. The maximum page size is `67108864` or `power 26`. The default page size is `power 26`.
`addressradix`	*n*	The radix in which numerical values are displayed. The default radix is 2 (binary). You may prefer radix 8 (octal), 10 (decimal), or 16 (hexadecimal).

B.3 Commands File

The command file for the simulator specifies a sequence of memory instructions to be performed. Each instruction is either a memory READ or WRITE operation, and includes a virtual memory address to be read or written. Depending on whether the virtual page for the address is present in physical memory, the operation will succeed, or, if not, a page fault will occur.

There are two operations one can carry out on pages in memory: READ and WRITE. The format for each command is *operation address* or *operation* random, where *operation* is READ or WRITE, and *address* is the numeric virtual memory address, optionally preceeded by one of the radix keywords bin, oct, or hex. If no radix is supplied, the number is assumed to be decimal. The keyword random will generate a random virtual memory address (for those who want to experiment quickly) rather than having to type an address.

For example, the sequence

READ bin 01010101

WRITE bin 10101010

READ random

WRITE random

causes the virtual memory manager to:

1. read from virtual memory address 85
2. write to virtual memory address 170
3. read from some random virtual memory address
4. write to some random virtual memory address

B.4 Output File (tracefile)

The output file contains a log of the operations since the simulation started (or since the last reset). It lists the command that was attempted and what happened as a result. You can review this file after executing the simulation.

The output file contains one line per operation executed. The format of each line is:

command address ... status

where:

- *command* is READ or WRITE,
- *address* is a number corresponding to a virtual memory address, and
- *status* is okay or page fault.

3. RUNNING THE SIMULATOR 1

A. Configuration and Commands Files

A.1 Configuration File

As It was described in the Task Description part, we created the memory.conf file with 8 pages. We will observe the pages from 0 to 7 in both virtual and physical pages. Rest of the settings remain same.

```
// memset  virt page #  physical page #  R (read from)  M (modified) inMemTime (ns) lastTouchTime (ns)
memset 0 7 0 0 0 0
memset 1 6 0 0 0 0
memset 2 5 0 0 0 0
memset 3 4 0 0 0 0
memset 4 3 0 0 0 0
memset 5 2 0 0 0 0
memset 6 1 0 0 0 0
memset 7 0 0 0 0 0

// enable_logging 'true' or 'false'
// When true specify a log_file or leave blank for stdout
enable_logging true

// log_file <FILENAME>
// Where <FILENAME> is the name of the file you want output
// to be print to.
log_file tracefile

// page size, defaults to 2^14 and cannot be greater than 2^26
// pagesize <single page size (base 10)> or <'power' num (base 2)>
pagesize 16384

// addressradix sets the radix in which numerical values are displayed
// 2 is the default value
// addressradix <radix>
addressradix 16

// numpages sets the number of pages (physical and virtual)
// 64 is the default value
// numpages must be at least 2 and no more than 64
// numpages <num>
numpages 64
```

Figure 5: Configuration File

We did not map for all pages but we made pagesize as 64, it means that fort he pages till 31, it will automatically assign physical pages.

A.2 Commands File

In hexadecimal form, for all 64 pages.

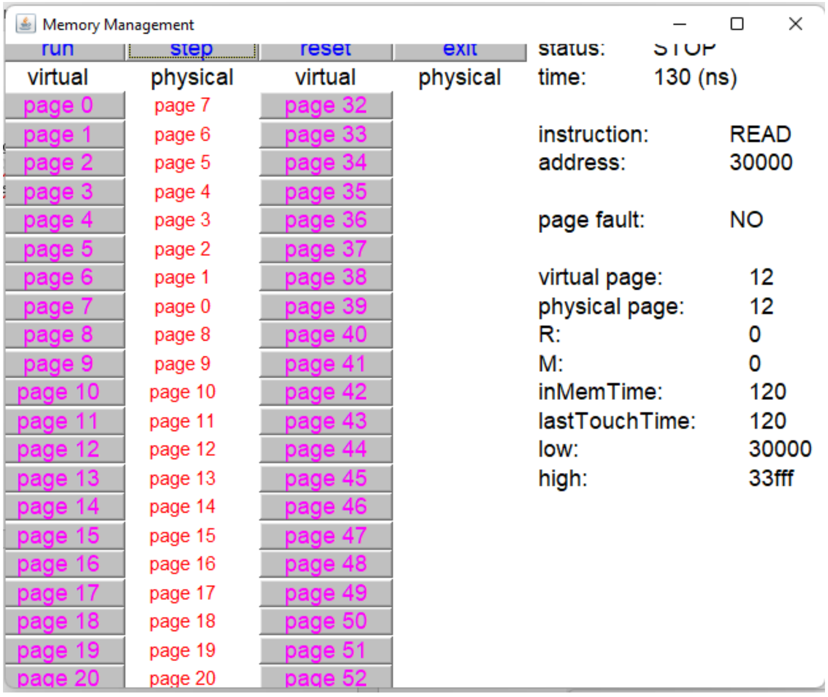
```
// Enter READ/WRITE commands into this file
// READ <OPTIONAL number type: bin/hex/oct> <virtual memory address or random>
// WRITE <OPTIONAL number type: bin/hex/oct> <virtual memory address or random>
READ hex 0000
READ hex 4000
READ hex 8000
READ hex c000
READ hex 10000
READ hex 14000
READ hex 18000
READ hex 1c000
READ hex 20000
READ hex 24000
READ hex 28000
READ hex 2c000
READ hex 30000
READ hex 34000
READ hex 38000
READ hex 3c000
READ hex 40000
READ hex 44000
READ hex 48000
READ hex 4c000
READ hex 50000
READ hex 54000
READ hex 58000
READ hex 5c000
READ hex 60000
READ hex 64000
READ hex 68000
READ hex 6c000
READ hex 70000
READ hex 74000
READ hex 78000
READ hex 7c000
READ hex 80000
READ hex 84000
READ hex 88000
READ hex 8c000
READ hex 90000
READ hex 94000
READ hex 98000
READ hex 9c000
READ hex a0000
READ hex a4000
READ hex a8000
READ hex ac000
READ hex b0000
READ hex b4000
READ hex b8000
READ hex bc000
READ hex c0000
READ hex c4000
READ hex c8000
READ hex cc000
READ hex d0000
READ hex d4000
READ hex d8000
READ hex dc000
READ hex e0000
READ hex e4000
READ hex e8000
READ hex ec000
READ hex f0000
READ hex f4000
READ hex f8000
READ hex fc000
```

Figure 6: Commands File

B. Simulation

Simulator Screen 1

At begining everything is as we set. We can see that virtual page 0 is assigned to physical page 7 as we write in the configuration file.



Simulator Screen 2

Simulation still goes on because we determined pagesize as 64 and it means that 32 virtual pages will be assigned to physical address automatically.

Simulator Screen 3

Everything still looks normal. We do not have any fault and continues to read.

Simulator Screen 4

Now we are out of mapped pages. We got an page fault here. It displayed physical page:-1, it means that page 32 is not mapped to any physical page. It started to search from virtual page 0 and saw the physical page 7 at the begining so it assigned it to there. It is because of FIFO algorithm.

Simulator Screen 5

It still gives page fault and taking the physical page numbers from the mapped ones.

Simulator Screen 6

It still gives page fault and taking the physical page numbers from the mapped ones.

Simulator Screen 7

It finished and we saw that it gave page fault for virtual pages between 32 and 63.

C. Discussion

When we check the java code we can see that FIFO algorithm runs for the page faults. It was directly considering the order of the mapping process. First mapped page was replaced for the first page fault.

FIFO algorithm did not care about the pages its own order. However, with Optimal Page replacement algorithm, we could making this replacing process depend on the duration of the last use.

Optimal Page Replacement: The idea is simple, for every reference we do following :[6]

1. If referred page is already present, increment hit count.
2. If not present, find if a page that is never referenced in future. If such a page exists, replace this page with new page. If no such page exists, find a page that is referenced farthest in future. Replace this page with new page.

[6] https://www.geeksforgeeks.org/optimal-page-replacement-algorithm/

D. Conclusion

We had chance to observe page replacement process with the FIFO algorithm. We saw that only mapped physical pages can be used by virtual pages. If there is not enoug physial pages mapped, existing physical pages are replaced and virtual pages continuie to their process. After the exercise we understood better about the page replacement process.

We also observed the some status of pages. We learned that if virtual memory did not mapped over physical memory we get page fault. In real world examples, we know that physical memory is much smaller than virtual memory, and it is obvious that we will get page faults. With this task we had an idea about how to handle with such problems.

E. Output File (tracefile)

```
READ 0 ... okay
READ 4000 ... okay
READ 8000 ... okay
READ c000 ... okay
READ 10000 ... okay
READ 14000 ... okay
READ 18000 ... okay
READ 1c000 ... okay
READ 20000 ... okay
READ 24000 ... okay
READ 28000 ... okay
READ 2c000 ... okay
READ 30000 ... okay
READ 34000 ... okay
READ 38000 ... okay
READ 3c000 ... okay
READ 40000 ... okay
READ 44000 ... okay
READ 48000 ... okay
READ 4c000 ... okay
READ 50000 ... okay
READ 54000 ... okay
READ 58000 ... okay
READ 5c000 ... okay
READ 60000 ... okay
READ 64000 ... okay
READ 68000 ... okay
READ 6c000 ... okay
READ 70000 ... okay
READ 74000 ... okay
READ 78000 ... okay
READ 7c000 ... okay
READ 80000 ... page fault
READ 84000 ... page fault
READ 88000 ... page fault
READ 8c000 ... page fault
READ 90000 ... page fault
READ 94000 ... page fault
READ 98000 ... page fault
READ 9c000 ... page fault
READ a0000 ... page fault
READ a4000 ... page fault
READ a8000 ... page fault
READ ac000 ... page fault
READ b0000 ... page fault
READ b4000 ... page fault
READ b8000 ... page fault
READ bc000 ... page fault
READ c0000 ... page fault
READ c4000 ... page fault
READ c8000 ... page fault
READ cc000 ... page fault
READ d0000 ... page fault
READ d4000 ... page fault
READ d8000 ... page fault
READ dc000 ... page fault
READ e0000 ... page fault
READ e4000 ... page fault
READ e8000 ... page fault
READ ec000 ... page fault
READ f0000 ... page fault
READ f4000 ... page fault
READ f8000 ... page fault
READ fc000 ... page fault
```

Figure 7: Output File

4. RUNNING THE SIMULATION 2

A. Configuration and Commands Files

A.1 Configuration File

As It was described in the Task Description part, we created the memory.conf file with 8 pages. We will observe the pages from 0 to 7 in both virtual and physical pages. We set addressradix to 10 to be able to observe the values in decimal, we could also observe in hexadecimal and 8 bit representation. Pagesize set to 16 because we only have 16 pages in total, we will not have mess on the simulator screen. Rest of the settings remain same.

```
// memset  virt page #  physical page #  R (read from)  M (modified) inMemTime (ns) lastTouchTime (ns)
memset 0 0 0 0 0 0
memset 1 1 0 0 0 0
memset 2 2 0 0 0 0
memset 3 3 0 0 0 0
memset 4 4 0 0 0 0
memset 5 5 0 0 0 0
memset 6 6 0 0 0 0
memset 7 7 0 0 0 0


// enable_logging 'true' or 'false'
// When true specify a log_file or leave blank for stdout
enable_logging true

// log_file <FILENAME>
// Where <FILENAME> is the name of the file you want output
// to be print to.
log_file tracefile

// page size, defaults to 2^14 and cannot be greater than 2^26
// pagesize <single page size (base 10)> or <'power' num (base 2)>
pagesize 16384

// addressradix sets the radix in which numerical values are displayed
// 2 is the default value
// addressradix <radix>
addressradix 10

// numpages sets the number of pages (physical and virtual)
// 64 is the default value
// numpages must be at least 2 and no more than 64
// numpages <num>
numpages 16
```

Figure 8: Configuration File For The Task

A.2 Commands File

We also made changes in the commands file. To have a better understanding, we will assign address numbers fort he half of the operations and fort he rest we will use random to observe *'page fault'*. It may not have fault also but we wish fort he random numbers to go beyond the mapped pages.

```
// Enter READ/WRITE commands into this file
// READ <OPTIONAL number type: bin/hex/oct> <virtual memory address or random>
// WRITE <OPTIONAL number type: bin/hex/oct> <virtual memory address or random>
READ 0
READ 1
READ 2
READ 3
READ random
READ random
READ random
READ random
WRITE 0
WRITE 1
WRITE 2
WRITE 3
WRITE random
WRITE random
WRITE random
WRITE random
```

Figure 9: Commands File For The Task

B. Simulation

Simulation will map the pages first. We know that each pagesize will be 16384. We can conclude that adresses from 0 to 131071 will be mapped. We can see the low and high address values fort he page 0 from the screenshot 1.

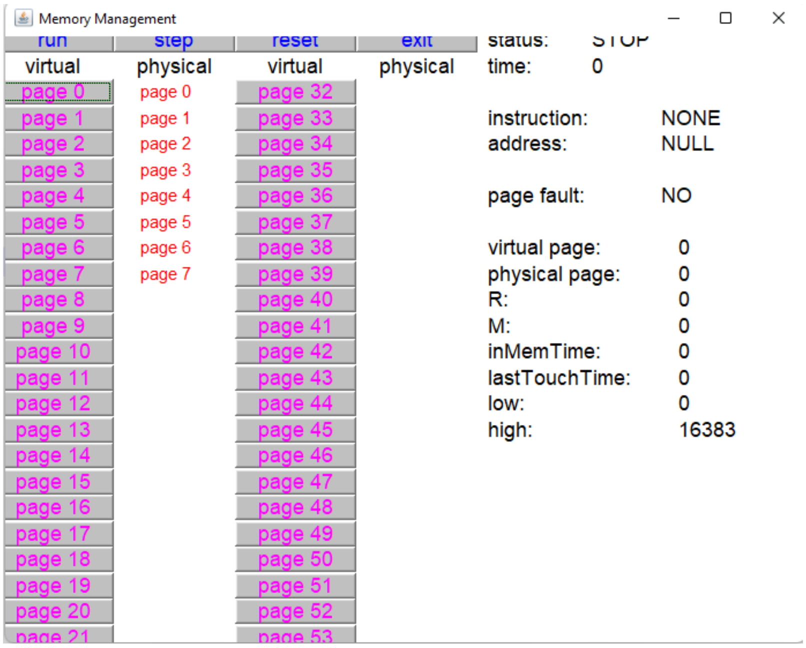
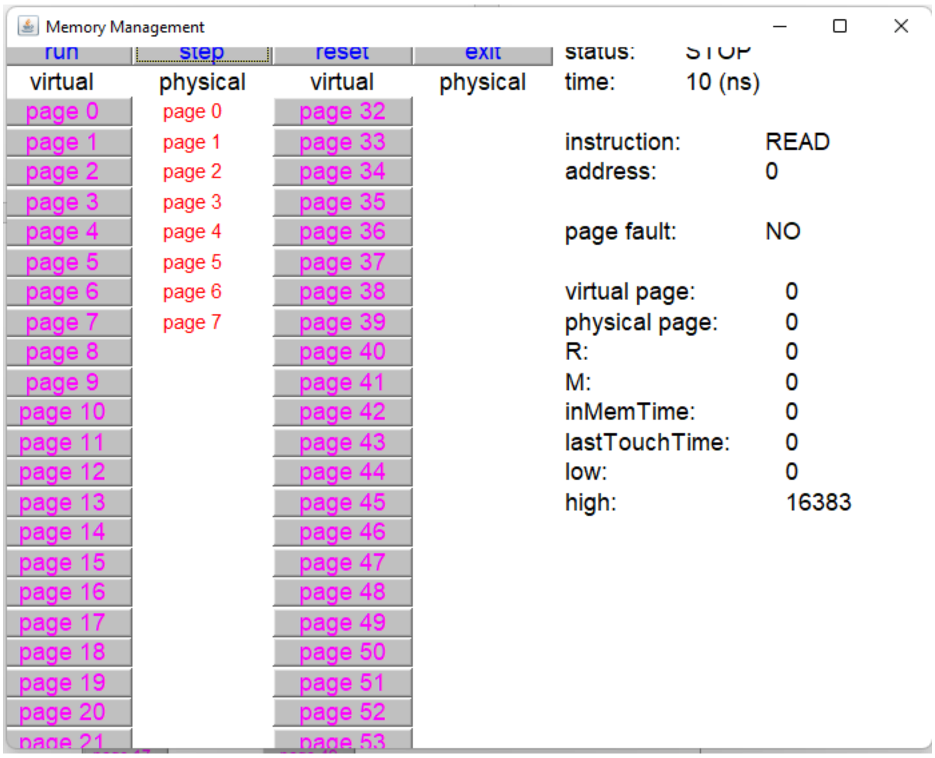
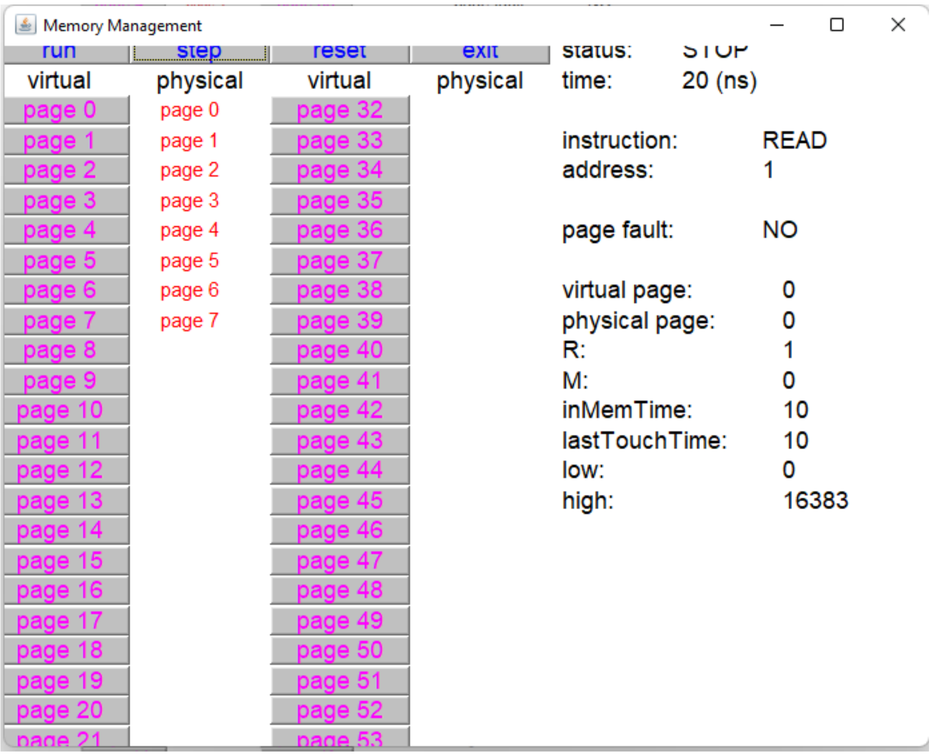


Figure 10: inital Screen of Simulator



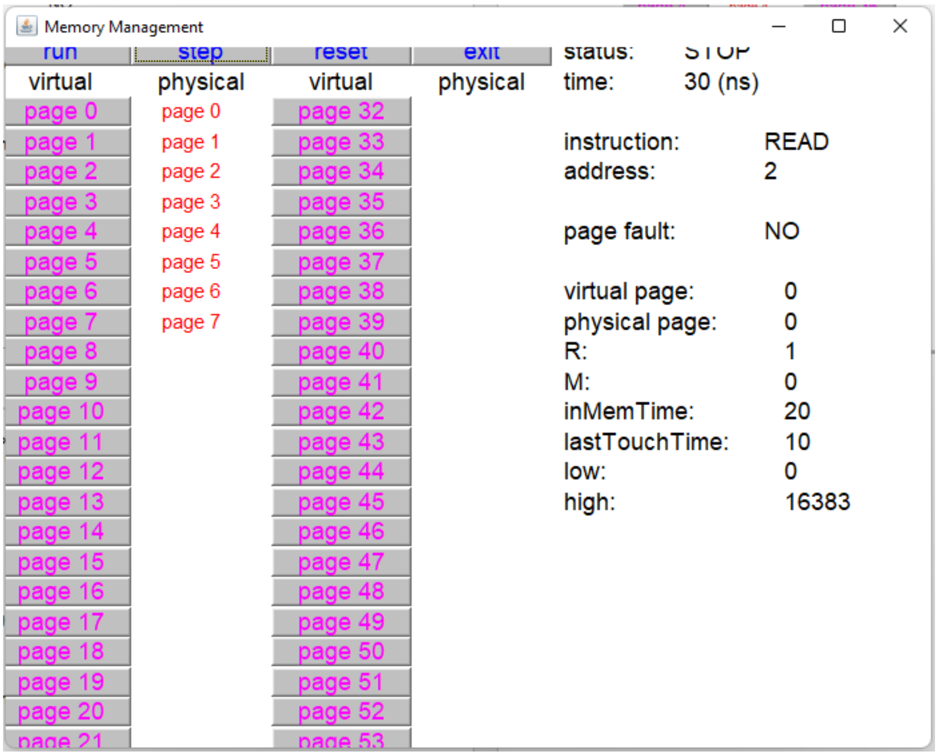
STEP 1

We can not see any page fault, It reads the 0 successfully.



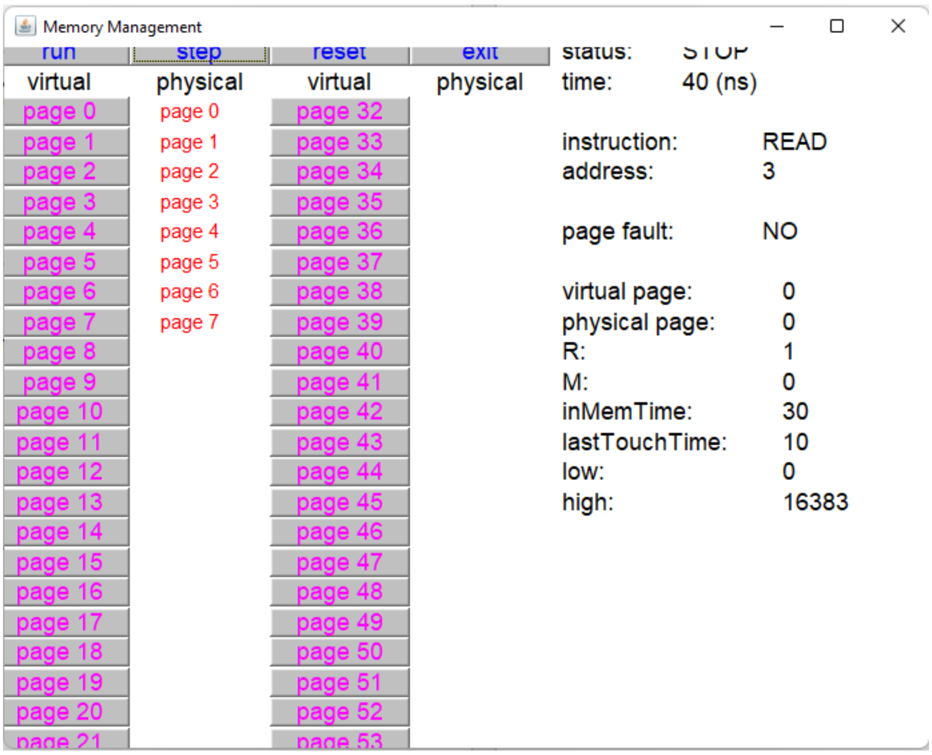
STEP 2

Again we read the page 1 and it is successful. We initialize the first 4 page in command file and because of it inMemTime is increasing linearly by the time. But last touch isreset after every step.



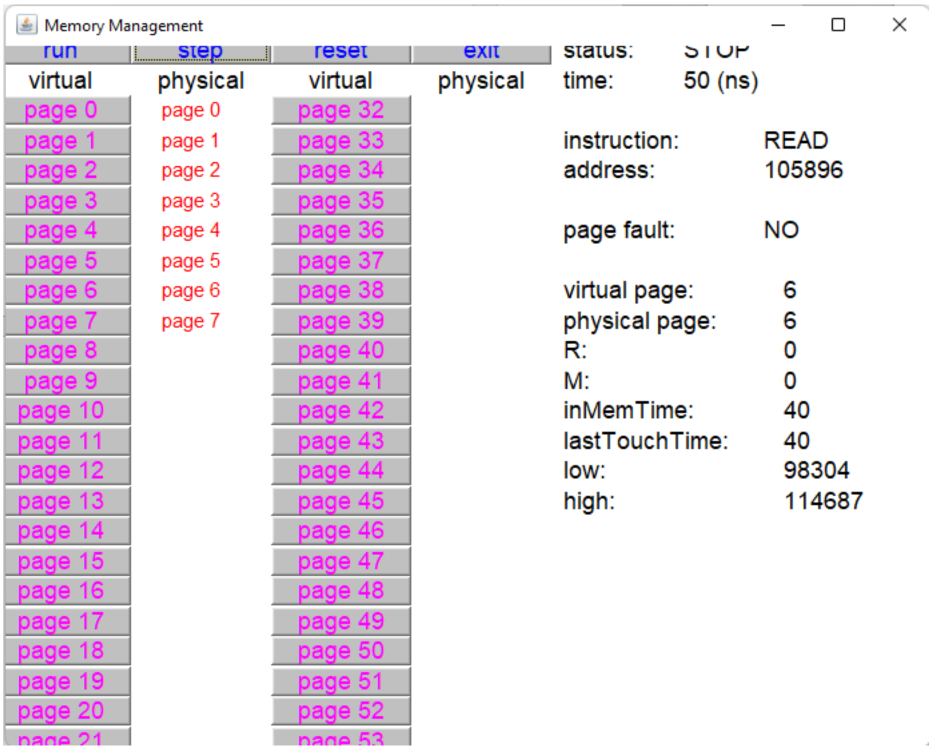
STEP 3

Again we read the page 2 and it is successful.



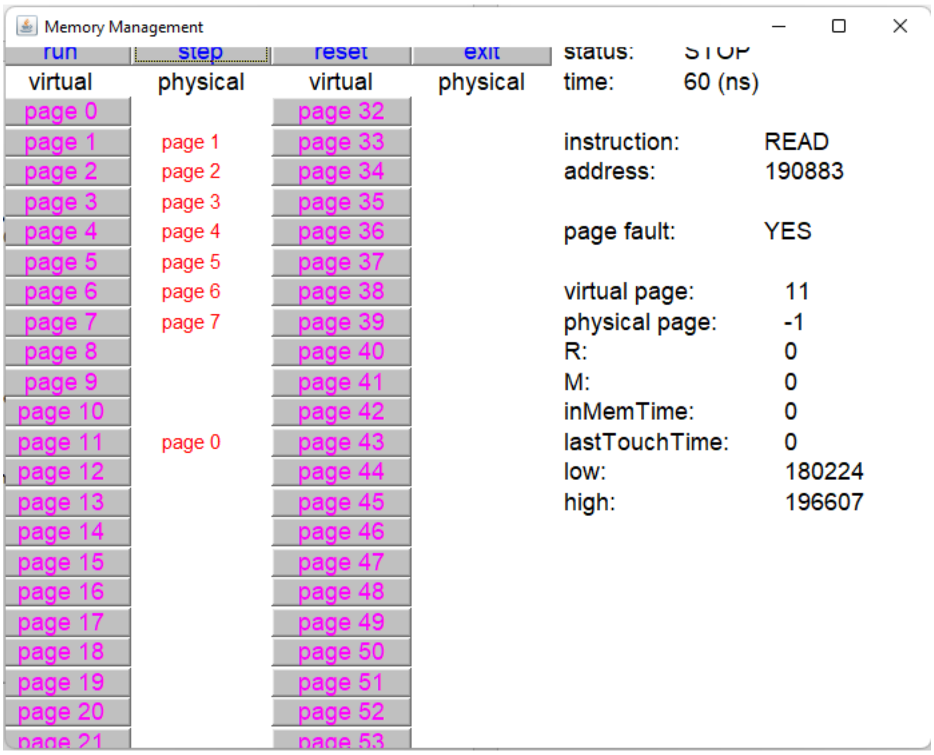
STEP 4

Again we read the page 3 and it is successful.



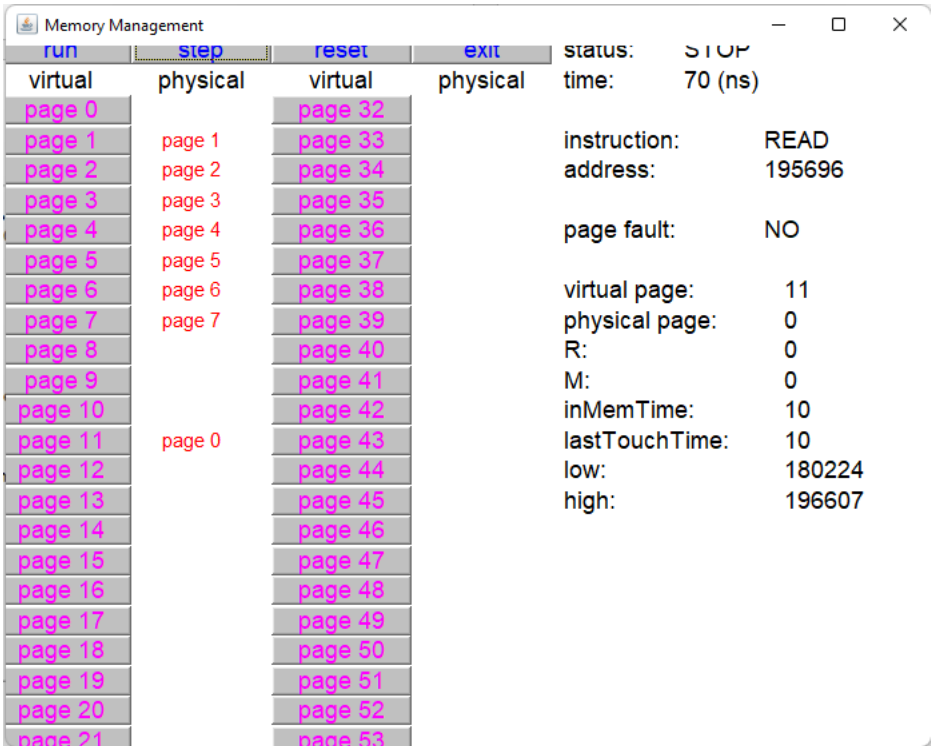
STEP 5

Remember from the commands file, we set *random* and it return page 6 as result. We were lucky that also page 6 was mapped with memory.conf file and we do not have any fault here.



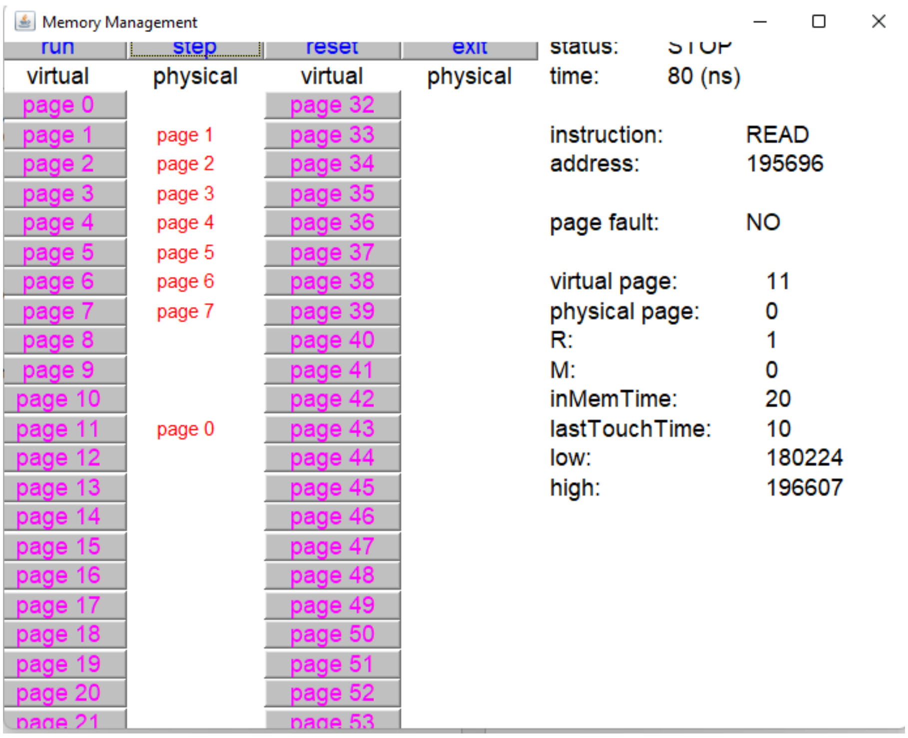
STEP 6

This time *random* returned page 11 and it gives us error because we did not map the page 11.



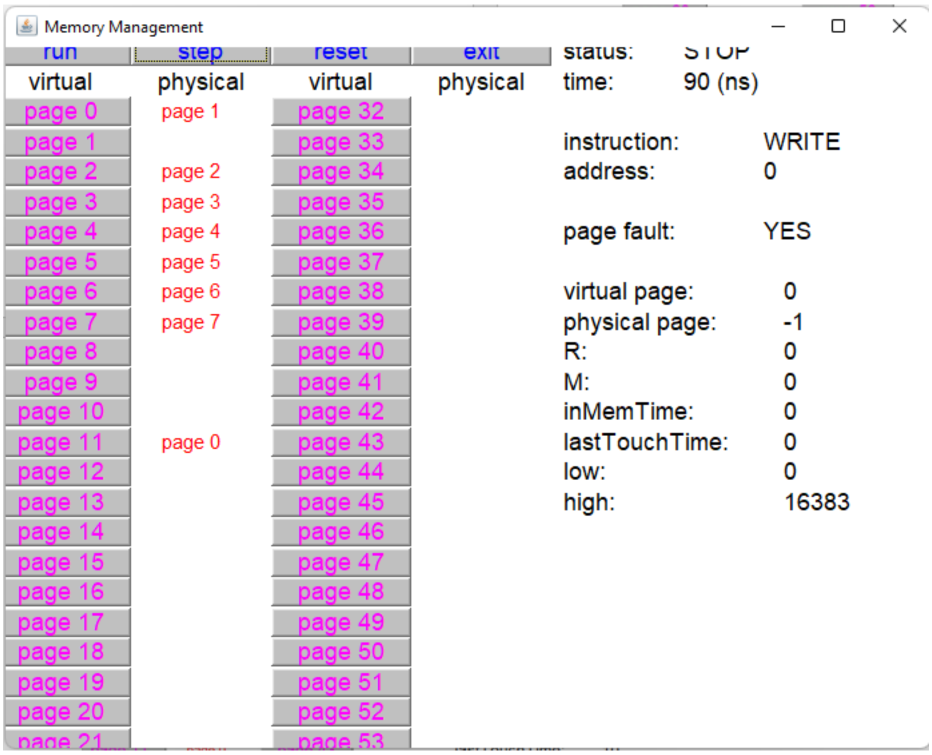
STEP 7

Now simulator start from 0 and look for a physical address to assign page 11. Page 0 was free and it used page 0 as physical address.

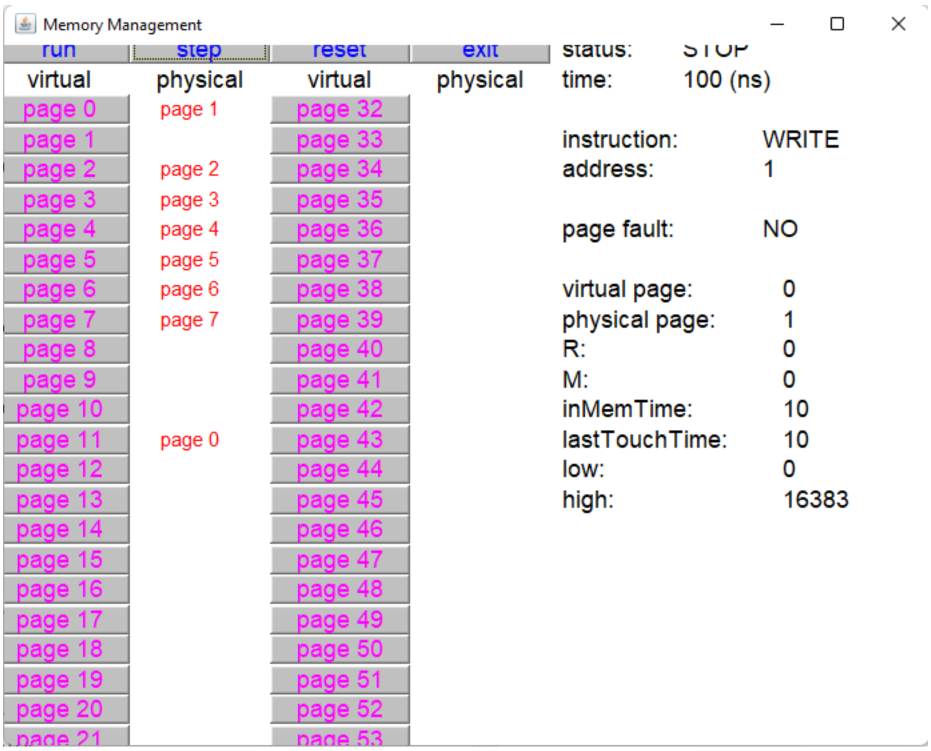


STEP 8

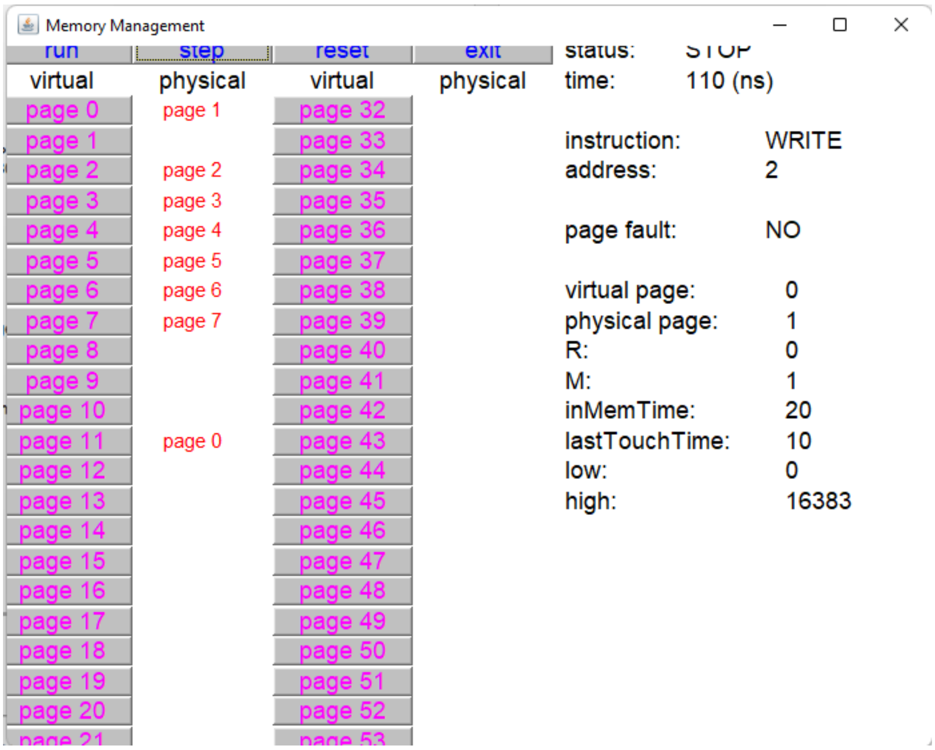
After, it was be able to read. And there is no fault. And R value which indicates the reading status become 1.



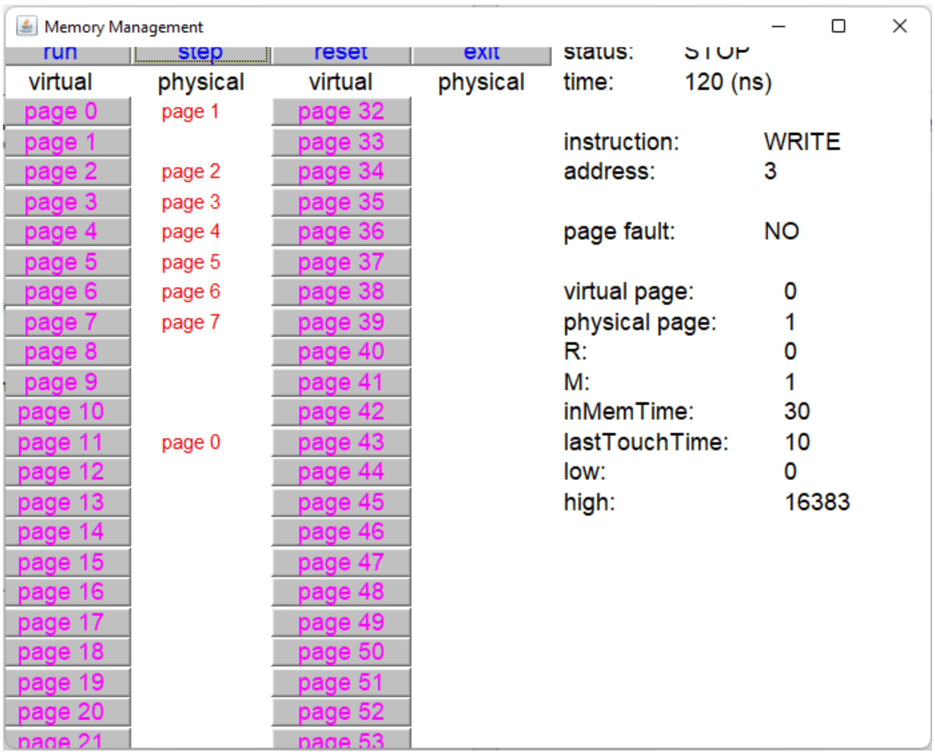
STEP 9



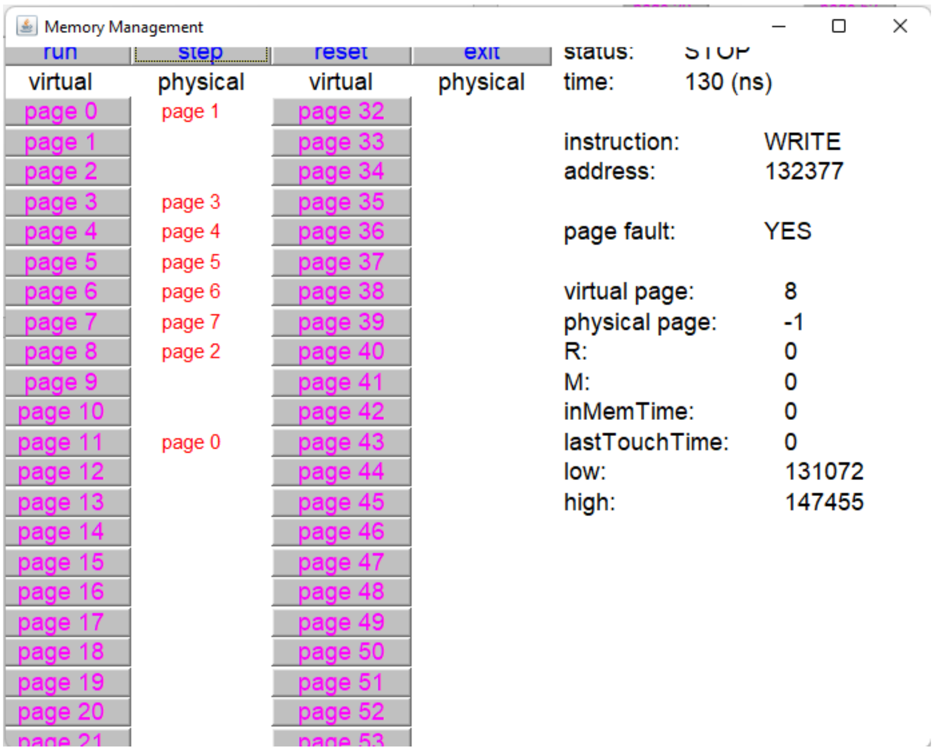
STEP 10



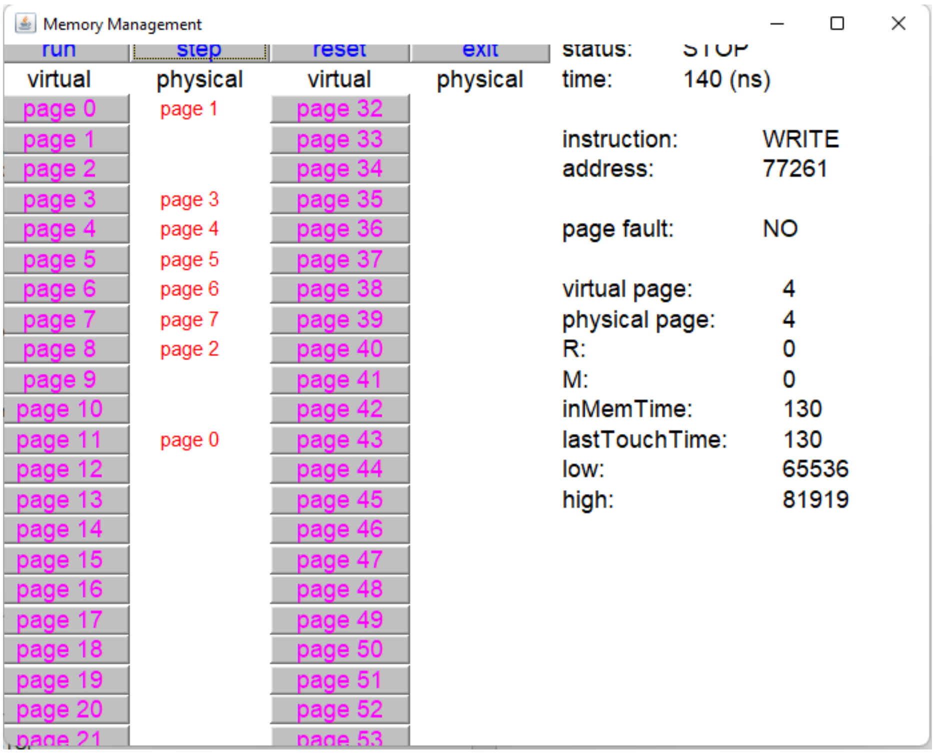
STEP 11



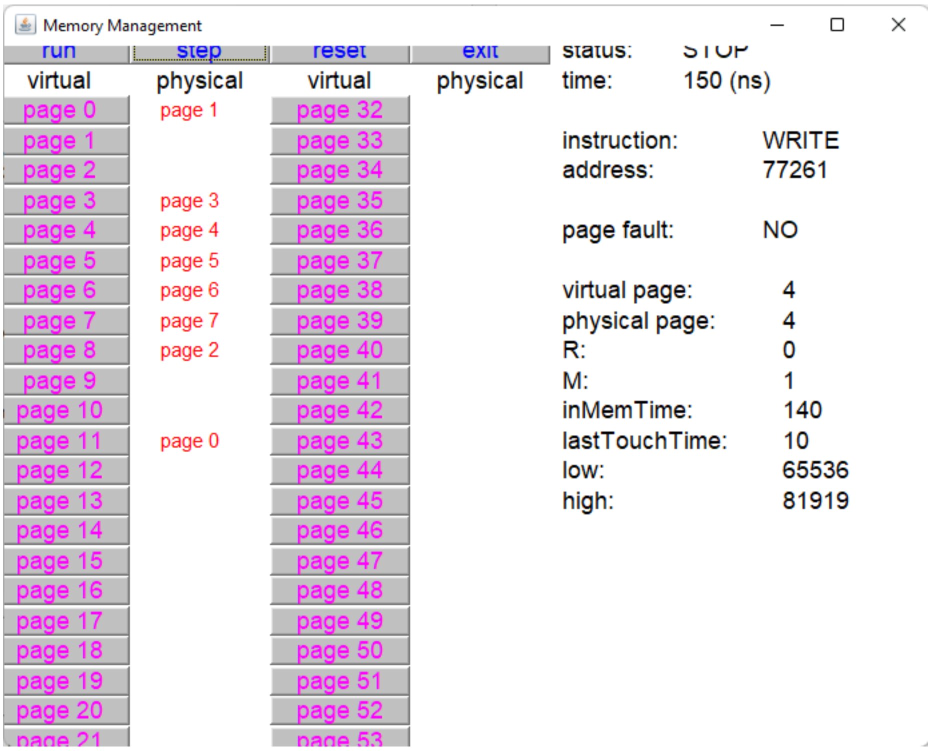
STEP 12



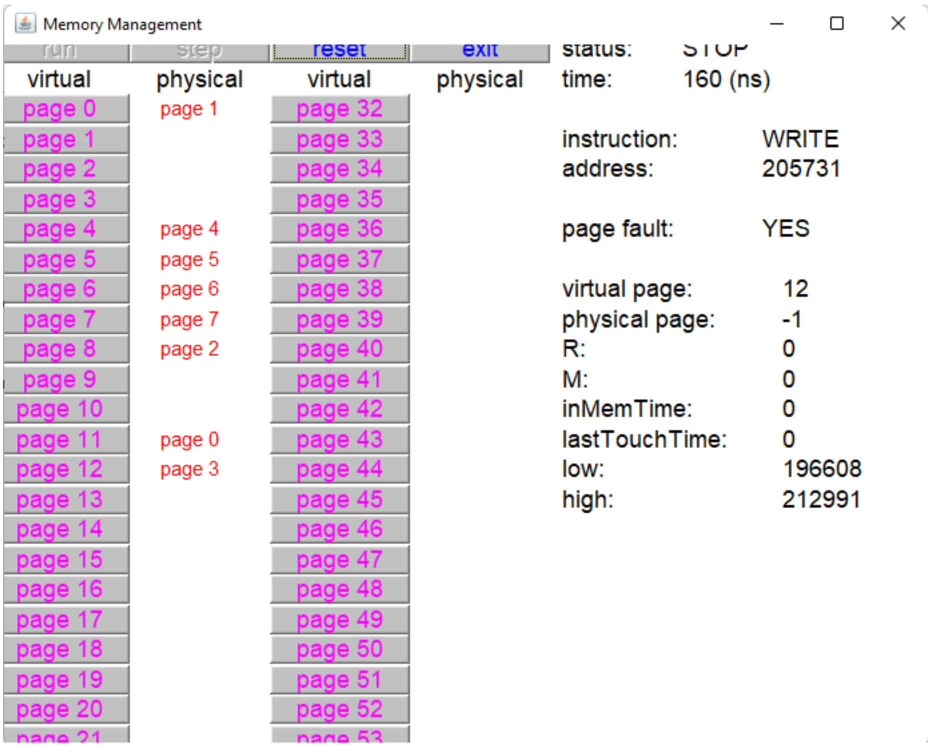
STEP 13



STEP 14



STEP 15



STEP 16

C. Discussion

FIFO algorithm acts so basic and directly takes the first physical address as new address during page fault.

We can say the same arguments as previous simulation.

D. Cocnlusion

We had a more clear view over the behaviour of the simulator. There was more clear explanation for the FIFO algorithm.

E. Output File (tracefile)

```
READ 0 ... okay
READ 1 ... okay
READ 2 ... okay
READ 3 ... okay
READ 105896 ... okay
READ 190883 ... page fault
READ 195696 ... okay
READ 195696 ... okay
WRITE 0 ... page fault
WRITE 1 ... okay
WRITE 2 ... okay
WRITE 3 ... okay
WRITE 132377 ... page fault
WRITE 77261 ... okay
WRITE 77261 ... okay
WRITE 205731 ... page fault
```

Figure 11: Output File (tracefile)

5. COMMENT

The simulator which exist in '/ Operating Systems (EOPSY) - course homepage / (pw.edu.pl)' has problems about displaying the virtual and physical pages at the beginning. Working version can be found at 'Moss | Memory Management Simulator | User Guide (ontko.com)'.