EOPSY - Task 4

Burak Kaymakci - 317054

Contents

1 Memory Management

Memory Management is the process of controlling and coordinating computer memory, assigning portions known as blocks to various running programs to optimize the overall performance of the system.

It is the most important function of an operating system that manages primary memory. It helps processes to move back and forward between the main memory and execution disk. It helps OS to keep track of every memory location, irrespective of whether it is allocated to some process or it remains free.

Memory management allows us to check how much memory needs to be allocated to processes that decide which processor should get memory at what time. Memory management tracks whenever inventory gets freed or unallocated. According to it will update the status. It allocates the space to application routines. It also makes sure that these applications do not interfere with each other. Helps protect different processes from each other It places the programs in memory so that memory is utilized to its full extent.

1.1 Memory Management Techniques

1.1.1 Single Contiguous Allocation

It is the easiest memory management technique. In this method, all types of computer's memory except a small portion which is reserved for the OS is available for one application. For example, MS-DOS operating system allocates memory in this way. An embedded system also runs on a single application.

1.1.2 Partitioned Allocation

It divides primary memory into various memory partitions, which is mostly contiguous areas of memory. Every partition stores all the information for a specific task or job. This method consists of allotting a partition to a job when it starts & unallocate when it ends.

1.1.3 Paged Memory Management

This method divides the computer's main memory into fixed-size units known as page frames. This hardware memory management unit maps pages into frames which should be allocated on a page basis.

1.1.4 Segmented Memory Management

Segmented memory is the only memory management method that does not provide the user's program with a linear and contiguous address space.

Segments need hardware support in the form of a segment table. It contains the physical address of the section in memory, size, and other data like access protection bits and status.

1.1.5 Swapping

Swapping is a method in which the process should be swapped temporarily from the main memory to the backing store. It will be later brought back into the memory for continue execution.

Backing store is a hard disk or some other secondary storage device that should be big enough inorder to accommodate copies of all memory images for all users. It is also capable of offering direct access to these memory images.

1.2 Paging

Paging is a storage mechanism that allows OS to retrieve processes from the secondary storage into the main memory in the form of pages. In the Paging method, the main memory is divided into small fixed-size blocks of physical memory, which is called frames. The size of a frame should be kept the same as that of a page to have maximum utilization of the main memory and to avoid external fragmentation. Paging is used for faster access to data, and it is a logical concept.

1.3 Fragmentation

Processes are stored and removed from memory, which creates free memory space, which are too small to use by other processes.

After sometimes, that processes not able to allocate to memory blocks because its small size and memory blocks always remain unused is called fragmentation. This type of problem happens during a dynamic memory allocation system when free blocks are quite small, so it is not able to fulfill any request.

Two types of Fragmentation methods are:

1. External fragmentation
2. Internal fragmentation

- External fragmentation can be reduced by rearranging memory contents to place all free memory together in a single block.
- The internal fragmentation can be reduced by assigning the smallest partition, which is still good enough to carry the entire process.

2 Memory Management Simulator

2.1 Introduction

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.

2.2 The Command 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.

2.3 Operations on Virtual Memory

Operations on Virtual Memory

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:

```
read from virtual memory address 85
write to virtual memory address 170
read from some random virtual memory address
write to some random virtual memory address
```

2.4 Sample Command File

The "commands" input file looks like this:

```
// Enter READ/WRITE commands into this file
// READ
// WRITE
READ bin 100
READ 19
WRITE hex CC32
READ bin 100000000000000
READ bin 100000000000000
WRITE bin 110000000000001
WRITE random
```

2.5 Setting Up the Virtual Memory Map

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

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

2.6 Sample Configuration File

Sample Configuration File

The "memory.conf" configuration file looks like this:

```
// 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
memset 8 8 0 0 0 0
memset 9 9 0 0 0 0
memset 10 10 0 0 0 0
memset 11 11 0 0 0 0
memset 12 12 0 0 0 0
```

```
memset 13 13 0 0 0 0
memset 14 14 0 0 0 0
memset 15 15 0 0 0 0
memset 16 16 0 0 0 0
memset 17 17 0 0 0 0
memset 18 18 0 0 0 0
memset 19 19 0 0 0 0
memset 20 20 0 0 0 0
memset 21 21 0 0 0 0
memset 22 22 0 0 0 0
memset 23 23 0 0 0 0
memset 24 24 0 0 0 0
memset 25 25 0 0 0 0
memset 26 26 0 0 0 0
memset 27 27 0 0 0 0
memset 28 28 0 0 0 0
memset 29 29 0 0 0 0
memset 30 30 0 0 0 0
memset 31 31 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
// Where  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  or <'power' num (base 2)>
pagesize 16384

// addressradix sets the radix in which numerical values are displayed
// 2 is the default value
// addressradix
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
numpages 64
```

2.7 The Output File

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.
```

2.8 Sample Output

The output "tracefile" looks something like this:

```
READ 4 ... okay
READ 13 ... okay
WRITE 3acc32 ... okay
READ 10000000 ... okay
READ 10000000 ... okay
WRITE c0001000 ... page fault
WRITE 2aeea2ef ... okay
```

3 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.

4 Memory Config File

Below you can see the contens of the `memory.conf` file.

```
// 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
```

As we were supposed to map 8 physical pages of memory to 8 virtual pages of memory. I've mapped the first 8 pages of physical memory to first 8 pages of virtual memory. Page size has been left intact. Address radix has been changed into decimal so that we can see the addresses easily without having to do the convertion from other radixes. Since we map 8 physical pages to 8 virtual pages, I've changed `numpages` to be 16 so that we are not using any other pages in the beginning.

5 Commands File

Below you can see the commands that the simulation will execute.

```
// 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 114688
READ 131072
READ 245760
READ 49151
READ 1542
READ 16384
READ bin 100
READ 20
READ 19
WRITE 147456
WRITE 148456
WRITE hex CC32
READ bin 100000000000000
READ bin 100000000000000
WRITE bin 110000000000001
WRITE random
READ 131072
READ 131073
READ 200000
```

If there is not number type (hex/bin/oct) before a number that means it will be treated as a decimal address.

6 Running the Simulation with Given Commands

Since our memory blocks are of size 16384, when calculating which address maps to which page, we need to think in multiples of 16384. For example the first page will have addresses between 0-16383 and the second page will have the addresses between 16384-32767. You can also verify this by clicking on virtual pages on the simulation. In Figure 1 you can actually verify that the addresses are between 0 and 16383 by looking at the `low` and `high` values on the right side of the pages.

6.1 Screenshots of the First 10 Steps of the Simulation

In Figures 2-11 you can see the screenshots of the first 10 steps of the simulation.

Figure 1: Addresses of the Page 0 (Low and High)

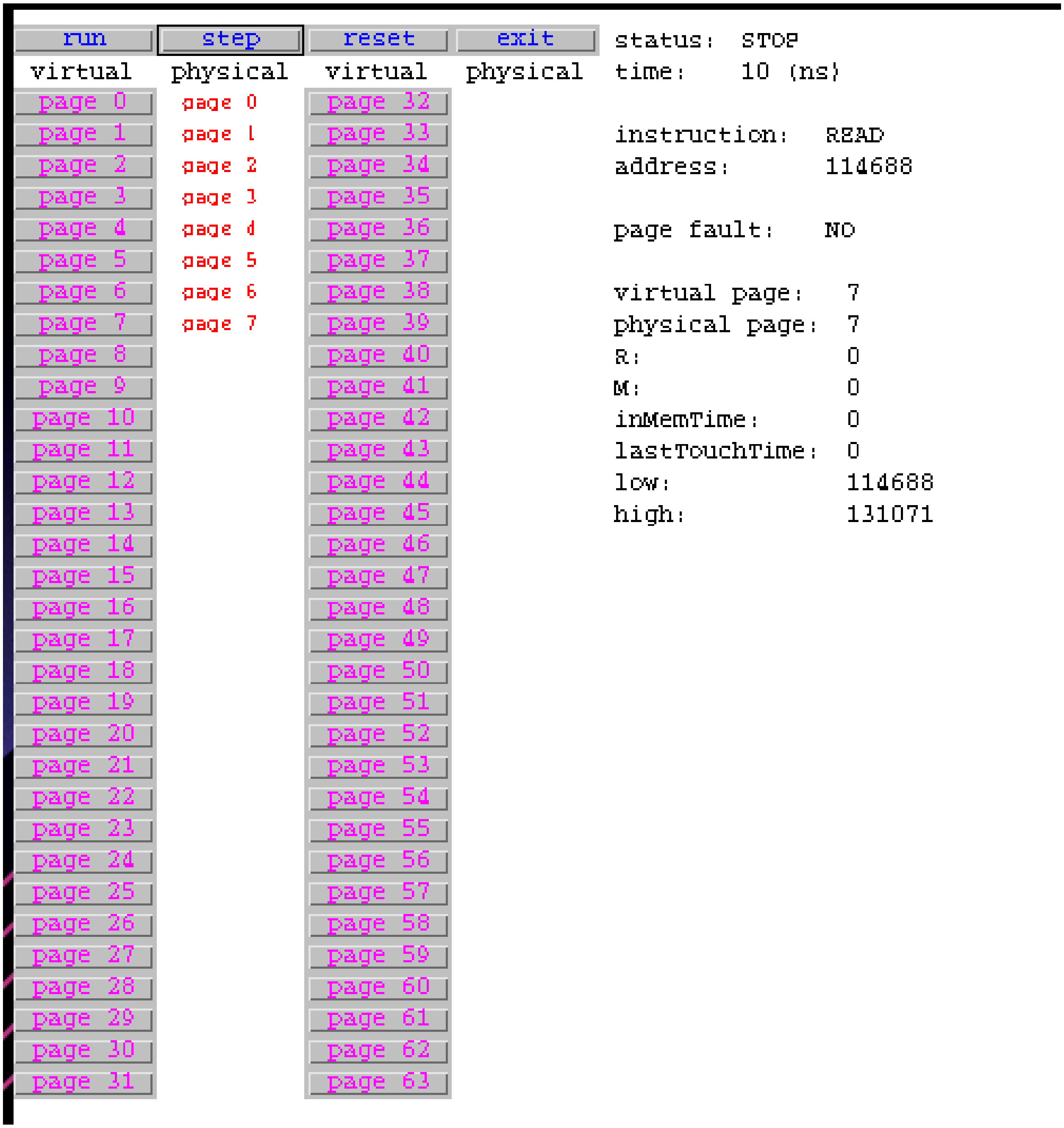

Figure 2: First Step of the Simulation

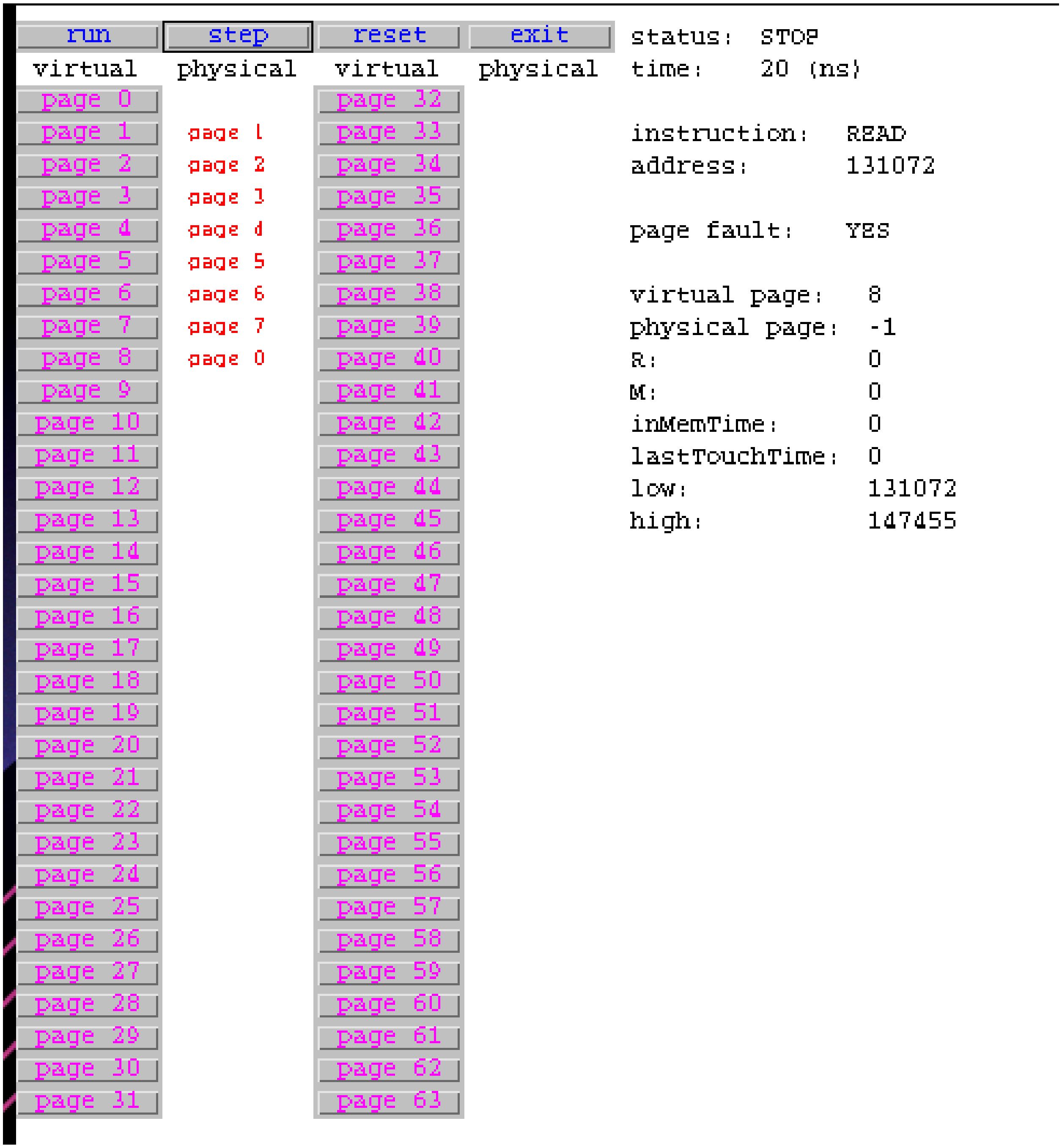

Figure 3: Second Step of the Simulation

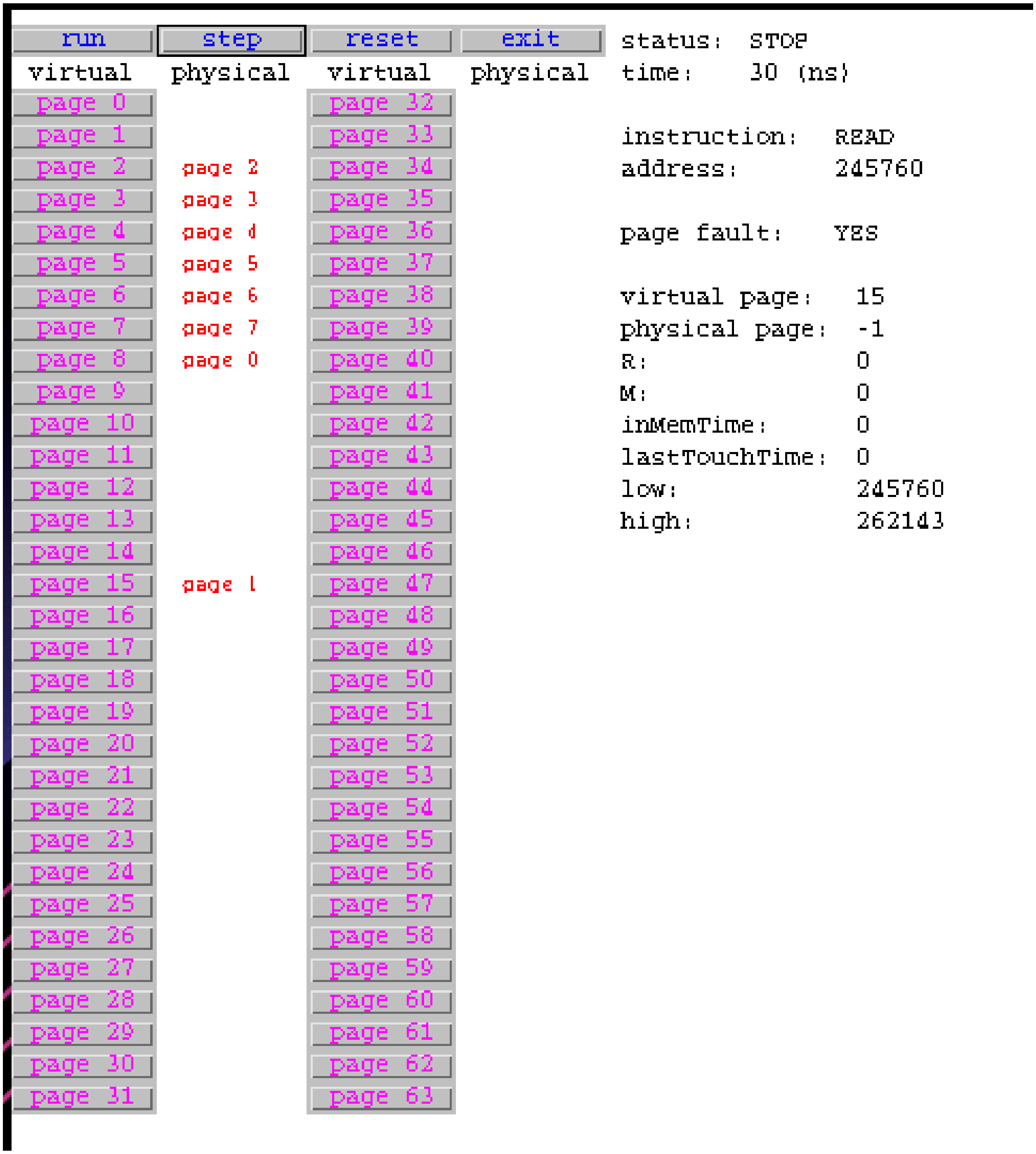

Figure 4: Third Step of the Simulation

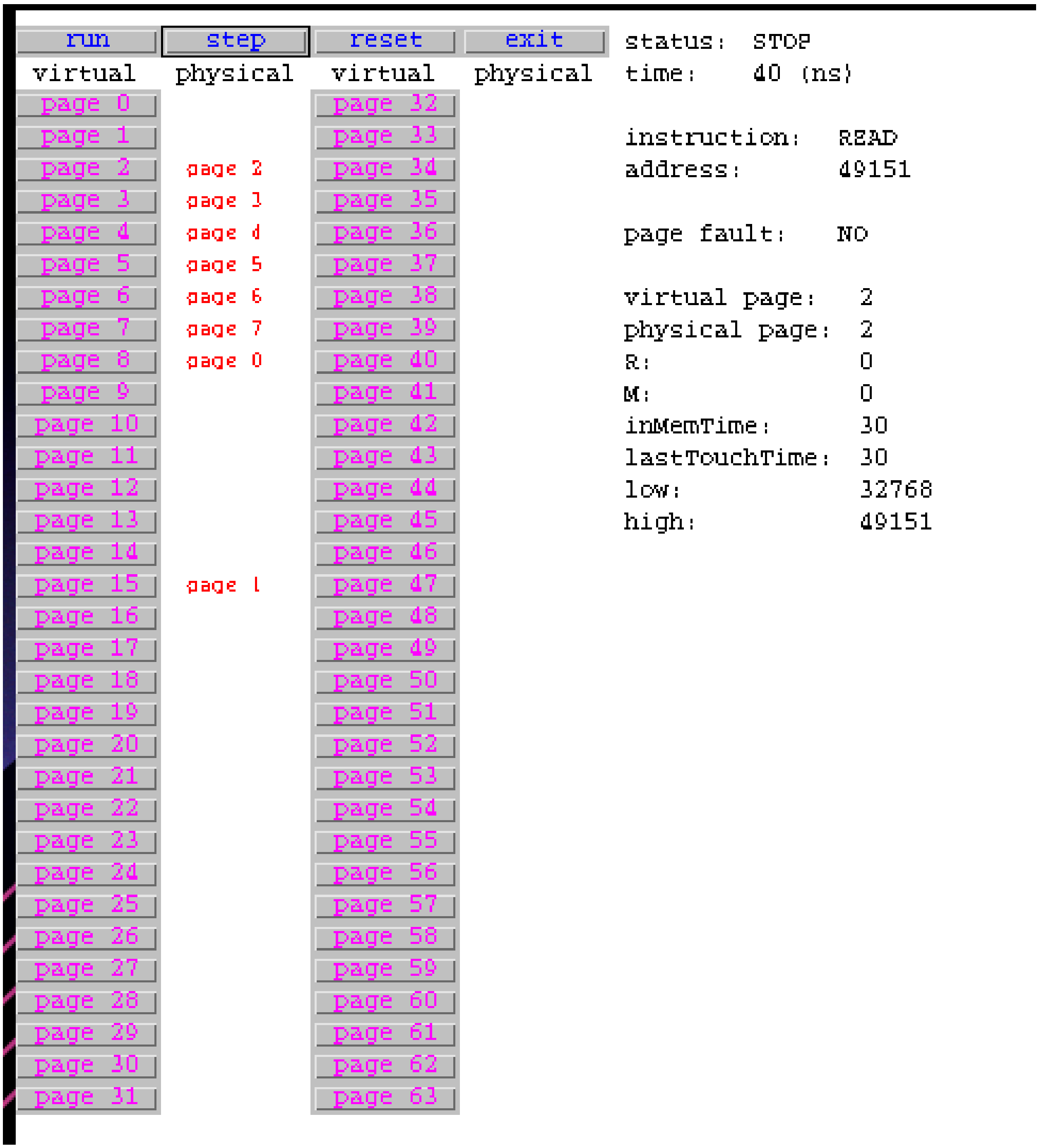

Figure 5: Fourth Step of the Simulation

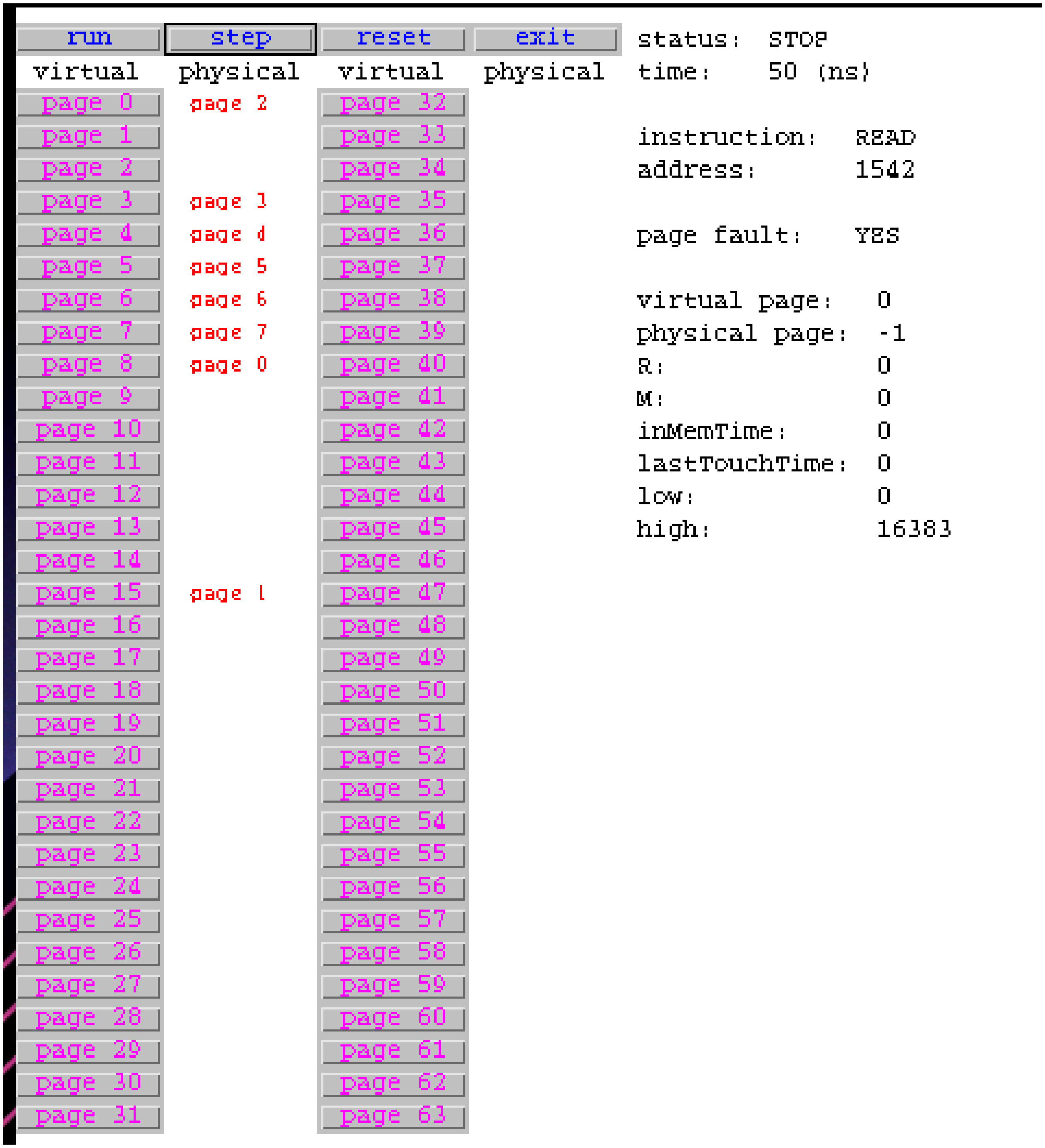

Figure 6: Fifth Step of the Simulation

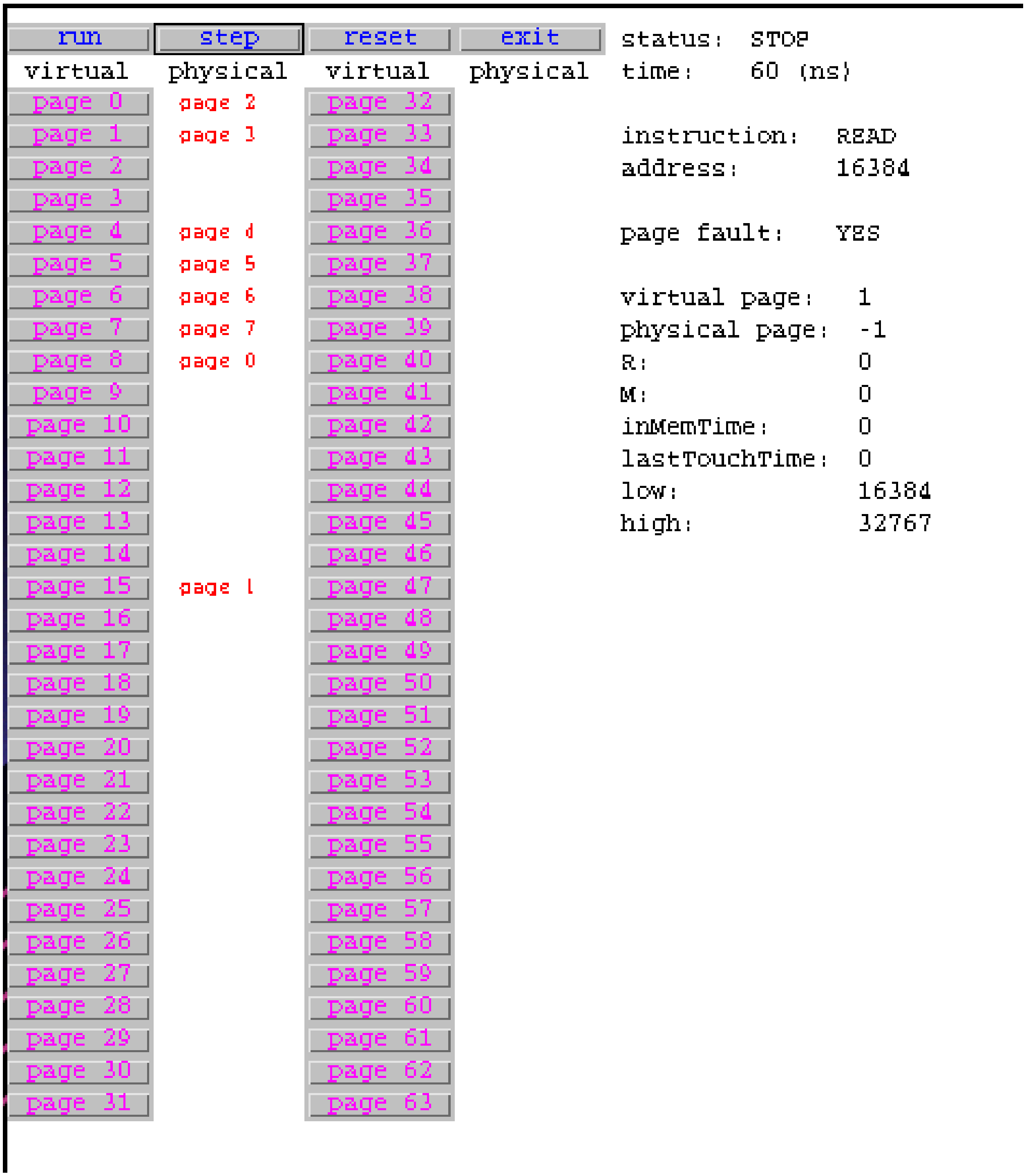

Figure 7: Sixth Step of the Simulation

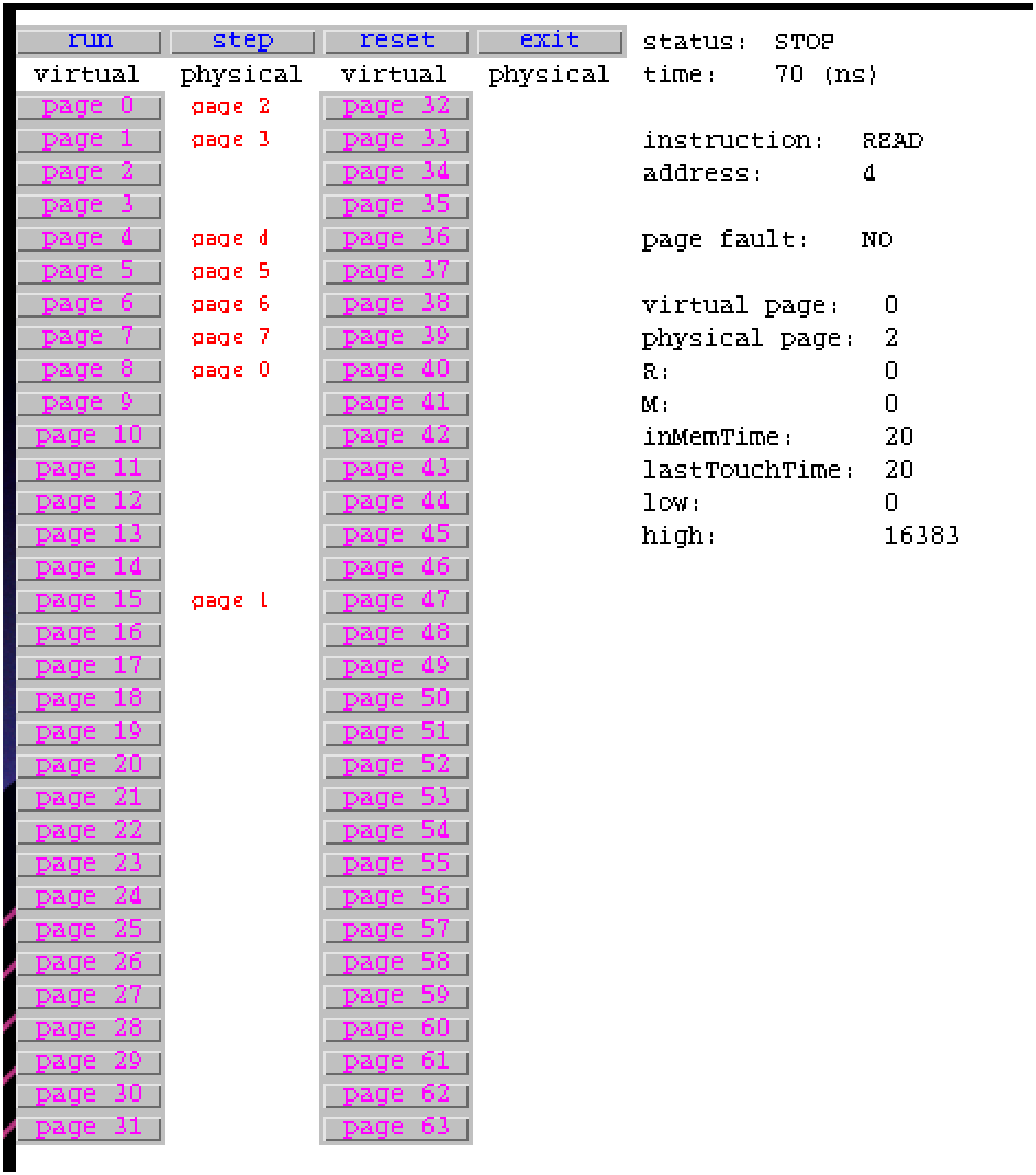

Figure 8: Seventh Step of the Simulation

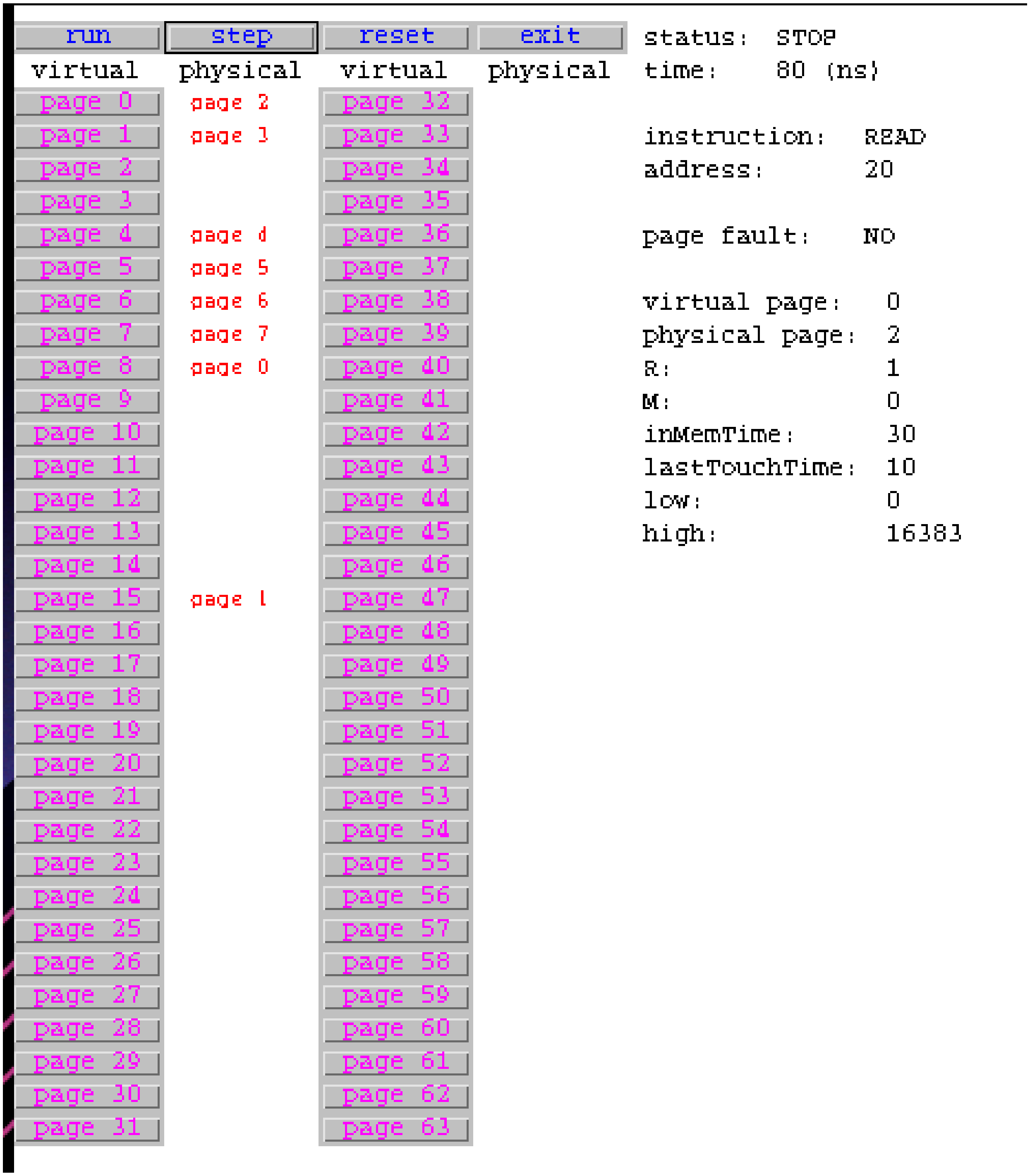

Figure 9: Eighth Step of the Simulation

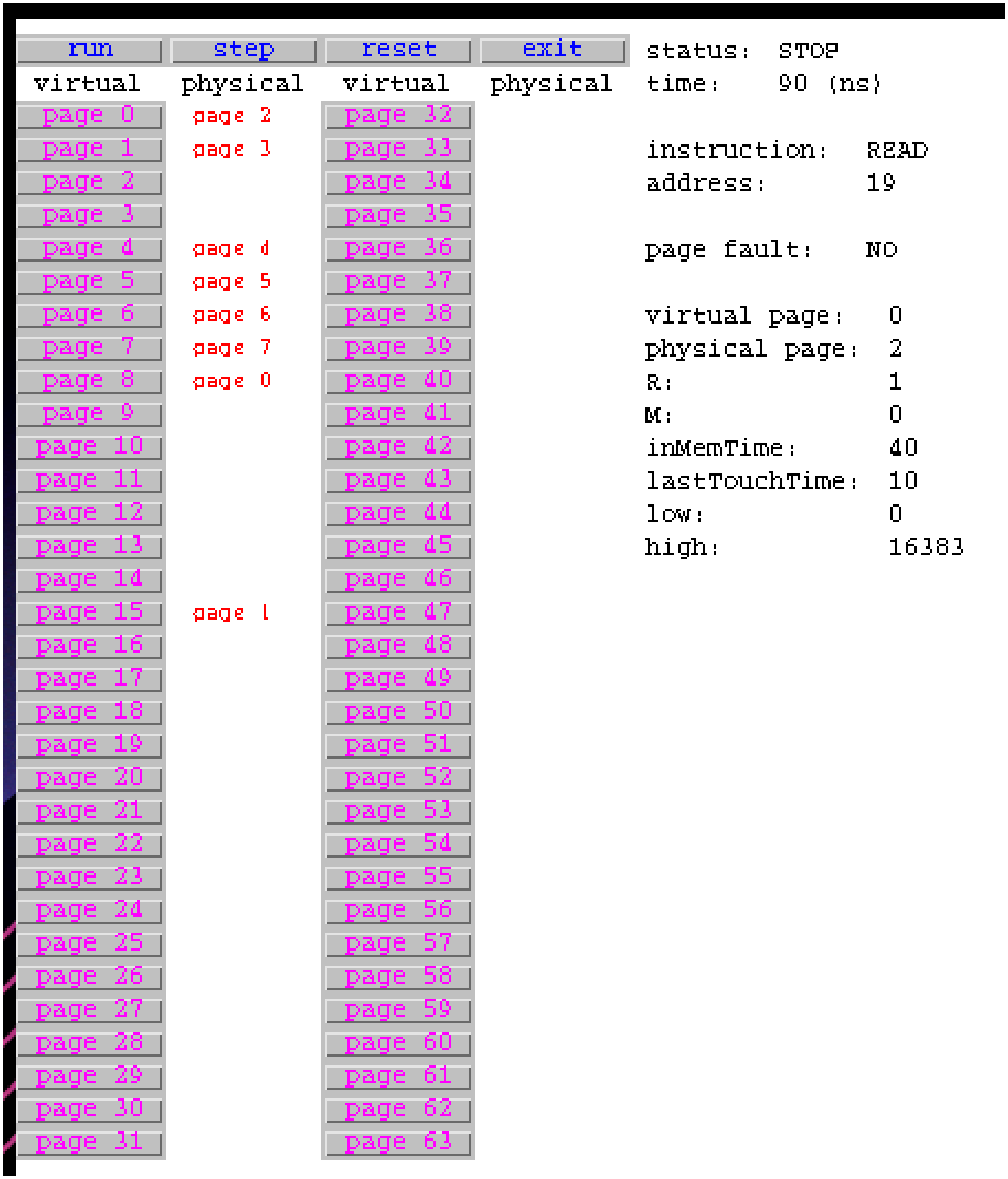

Figure 10: Ninth Step of the Simulation

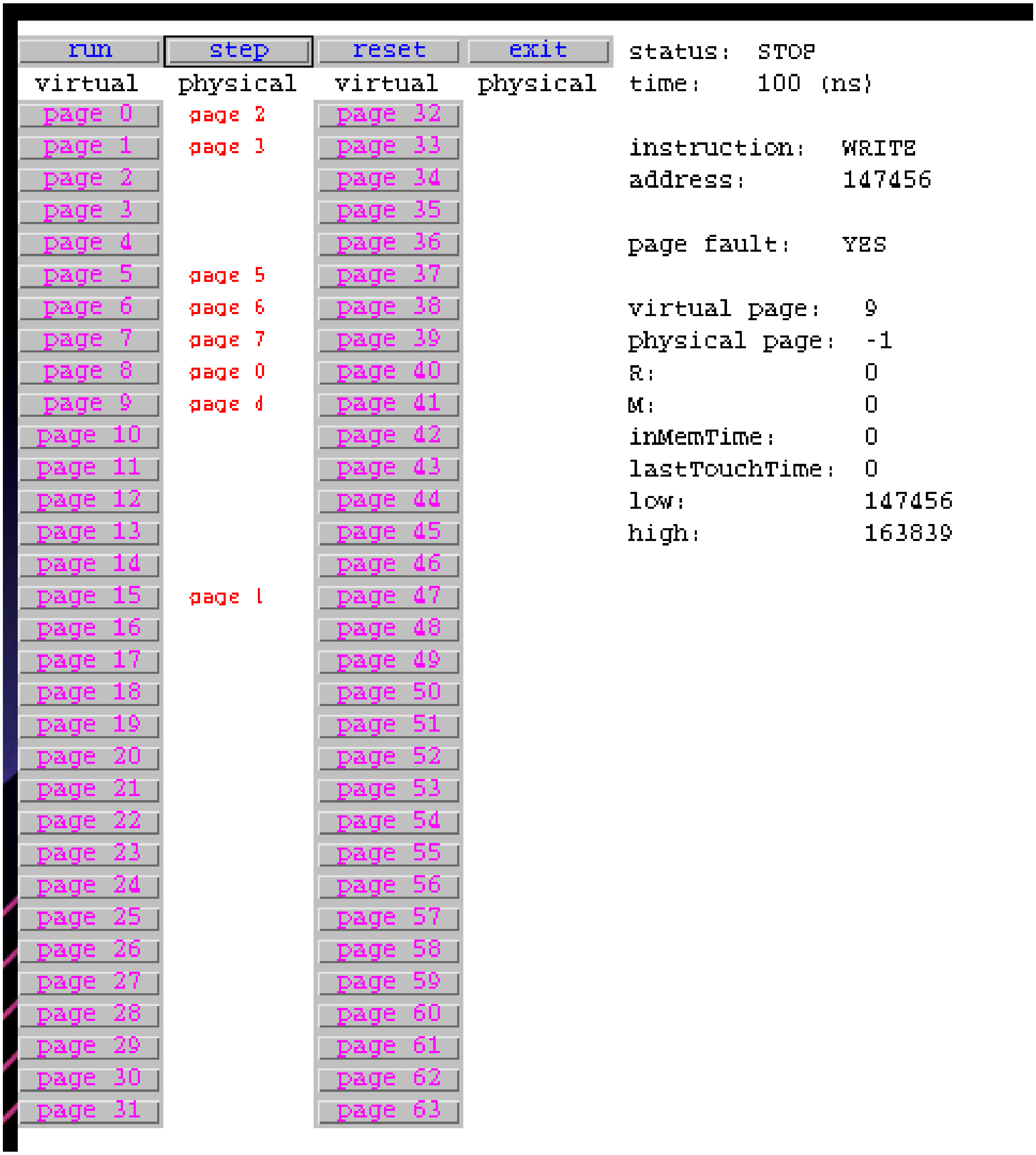

Figure 11: Tenth Step of the Simulation

7 Tracefile

Below you can see the contents of the `tracefile` after we've run through all the steps in our `commands` file.

```
READ 114688 ... okay
READ 131072 ... page fault
READ 245760 ... page fault
READ 49151 ... okay
READ 1542 ... page fault
READ 16384 ... page fault
READ 4 ... okay
READ 20 ... okay
READ 19 ... okay
WRITE 147456 ... page fault
WRITE 148456 ... okay
WRITE 52274 ... page fault
READ 16384 ... okay
READ 16384 ... okay
WRITE 24577 ... okay
WRITE 5845 ... okay
READ 131072 ... okay
READ 131073 ... okay
READ 200000 ... page fault
```

8 Conclusions

Below you can find the explanation of all the commands in the `commands` file and why there was a page fault if there was any while executing a command.

`READ 114688 ... okay`

Since the address is in the virtual page 7 (114688/16384 = 7) and since the virtual page 7 is mapped we only get `okay` result.

`READ 131072 ... page fault`

The address 131072 is in the virtual page 8 and we do not have that mapped to any physical page, hence, we get a page fault and the simulation maps the virtual page 8 to physical page 0 as you can see in Figure 3.

`READ 245760 ... page fault`

The address 245760 is in the virtual page 15 and we do not have that mapped to any physical page, hence, we get a page fault and the simulation maps the virtual page 15 to physical page 1 which was the first in the queue as you can see in Figure 4. This also gives us a hint about which algorithm the simulation uses. It is FIFO (First In First Out).

`READ 49151 ... okay`

The address 49151 is in the virtual page 2 and that page is mapped to physical page 2 so we have no problem reading from that address. See Figure 5.

`READ 1542 ... page fault`

Since the address 1542 is in virtual page 0 and we've actually re-mapped our physical page 0 from virtual page 0 to virtual page 8 above. We get a page fault trying to read from virtual page 0. Hence the algorithm again maps virtual page 0 to the next physical page in the queue which is physical page 2 as you can see in Figure 6.

`READ 16384 ... page fault`

The address 16384 is actually in virtual page 1 but since the virtual page 1 is now not mapped to any physical page, we again get a page fault. After that the algorithm maps physical page 3 to virtual page 1. See Figure 7.

`READ 4 ... okay` `READ 20 ... okay` `READ 19 ... okay`

Since the three commands above all read from the virtual page 0 and that page is actually mapped to physical page 2 we can easily read it without having any page fault. See Figures 8, 9, 10.

And basically the rest of the commands follow the same algorithm and that concludes the conclusions about this task.