EOPSY

Lab 4 Michał Kurczak

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?

Background

Computer memories are made up of registers. Each register comprises of cells capable of storing one bit data, address lines used for routing the data by means of data buses. The matter of whether we want to write to the memory or read from it is handled by read and write connection lines. Those register make a one dimensional array which is used for storing data and processes. The simplest way to write them into the memory would be to organize them in a contiguous manner so right after the last bit of one program we write the first bit of another one. But processes come and go, hence when we free a memory a vacant space is left. Fortunately, a device called Memory Management Unit keeps track of it and allocates the free memory space. If we want to access the memory that has no physical address mapped then a page fault warning is returned.

But processes differ in sizes. This problem is partially handled by partitioning the memory into segments called pages.
In terms of our exercise, we are dealing with the static partitioning, where the size of each partition is pre-set. The “pagesize” variable in the memory config of our program has been set to the default 16384 and number of pages has been set to 64

```
// 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
// 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 1. Initial configurations

The task is to map the first 8 virtual addresses to the physical addresses but the application is not working properly so we'll map all the 32 pages with default settings:

```
// 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
```

Figure 2. Initial configuration of memory mapping

Using simple bash script

```
#!/bin/bash


pageSize=16384
for i in {0..63}
do
        echo READ $(( i* $pageSize )) >> commands
done
```

Figure 3. Automatic commands generation

a command file is made allowing us to read all the 64 pages status at each step

```
1 READ 0
2 READ 16384
3 READ 32768
4 READ 49152
5 READ 65536
6 READ 81920
7 READ 98304
8 READ 114688
9 READ 131072
```

Figure 4. Snipped of the commands file

After each **READ** call I'm specifying address of **n'th** page given by the formula:

page_address = n * page_size

As the reading of the corresponding pages proceeds we may predict that the first page fault will occur at the virtual page 32 as it is initially the first page with unmapped physical address.

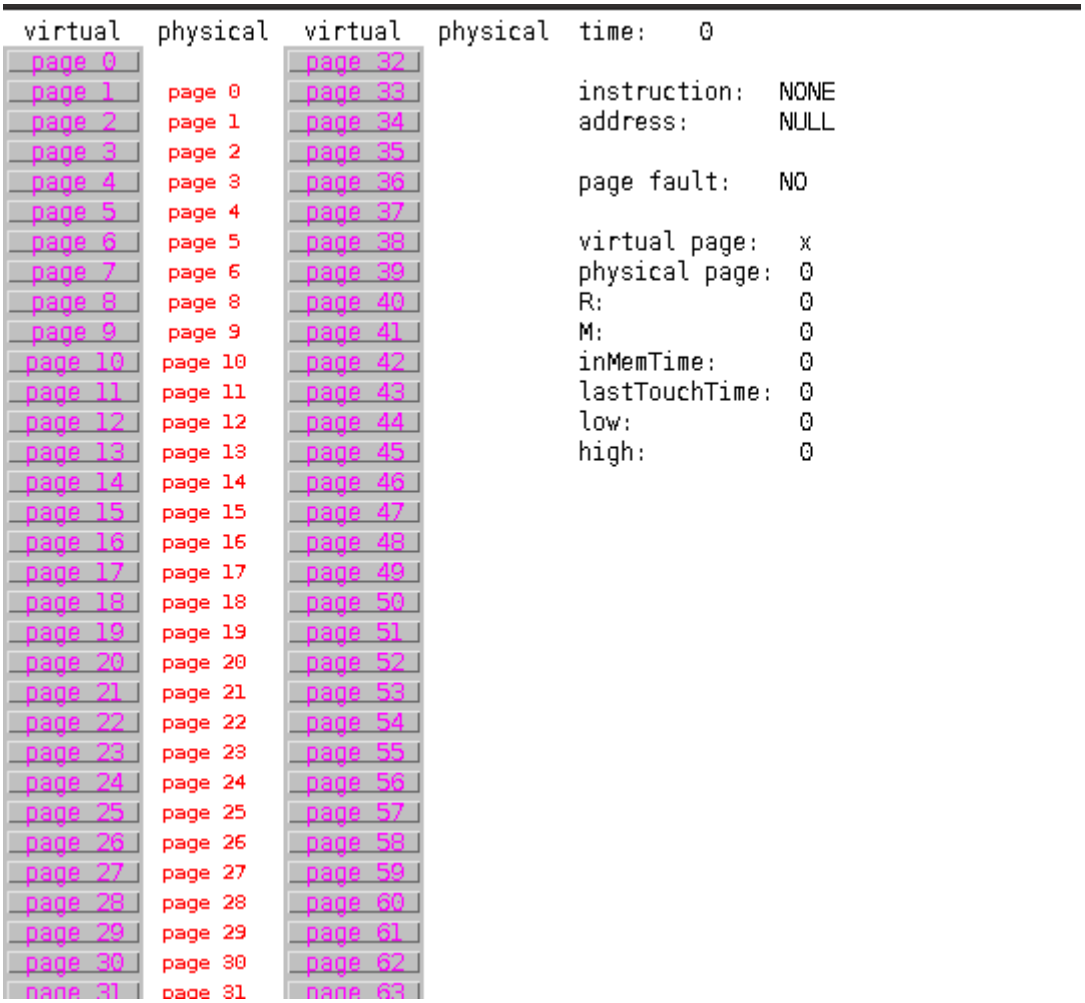

Figure 5. Initial state of the program

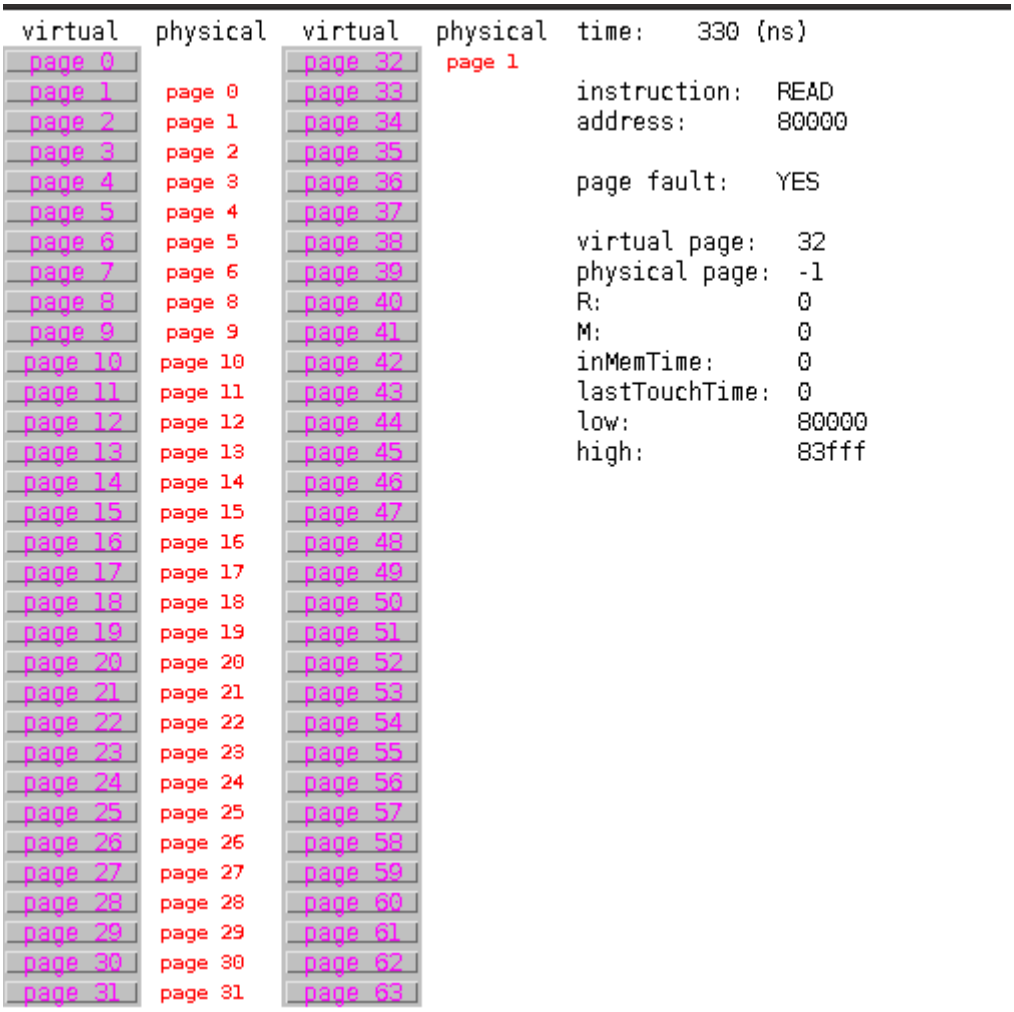

Figure 6. First page fault occurence

The page fault indeed occurred. Let's have a look at the "**tracefile"** output obtained from the configuration file

```
27 READ 68000 ... okay
28 READ 6c000 ... okay
29 READ 70000 ... okay
30 READ 74000 ... okay
31 READ 78000 ... okay
32 READ 7c000 ... okay
33 READ 80000 ... page fault
34 READ 84000 ... page fault
35 READ 88000 ... page fault
36 READ 8c000 ... page fault
37 READ 90000 ... page fault
```

Figure 7. Tracefile output

The first page fault occurred at line 33 which corresponds to the address of the virtual page with index 32. As the program executes we observe that right after we try to read the virtual page 32 it

gets the first physical address mapped. We can conclude that most likely the algorithm for memory management is a simple FIFO queue. Let's try to read from virtual page 32 right after new memory space is allocated by pasting the same line twice.

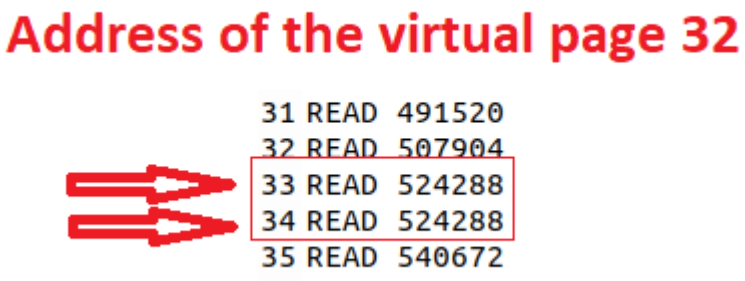

Figure 8. Reading the page after memory mapping

First page fault occurs

```
instruction:   READ
address:       80000

page fault:    YES

virtual page:   32
physical page:  -1
R:              0
M:              0
inMemTime:      0
lastTouchTime:  0
low:            80000
high:           83fff
```

Figure 9. First execution of READ

Then when we go to the next step

```
instruction:   READ
address:       80000

page fault:    NO

virtual page:   32
physical page:  1
R:              0
M:              0
inMemTime:      10
lastTouchTime:  10
low:            80000
high:           83fff
```

Figure 10. Second execution of READ

No page fault occurred since memory has been correctly mapped.

Tuning the program up

Now let's try a different number of pages and try to change the default mapping config by addressing to the first 8 virtual addresses corresponding physical addresses but incremented by 1 – no physical page 0 is present

```
1 // memset  virt page #  physical page #  R (read from)  M (modified) inMemTime (ns) lastTouchTime (ns)
2 memset 0 1 0 0 0 0
3 memset 1 2 0 0 0 0
4 memset 2 3 0 0 0 0
5 memset 3 4 0 0 0 0
6 memset 4 5 0 0 0 0
7 memset 5 6 0 0 0 0
8 memset 6 7 0 0 0 0
9 memset 7 8 0 0 0 0
```

```
// 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 32
```

Figure 11. Initial configurations

The first thing we can notice that the physical page 0 is still available even though we specified it in another way. On the other hand the physical page 7 got lost, but when we inspect a page the config seems ok

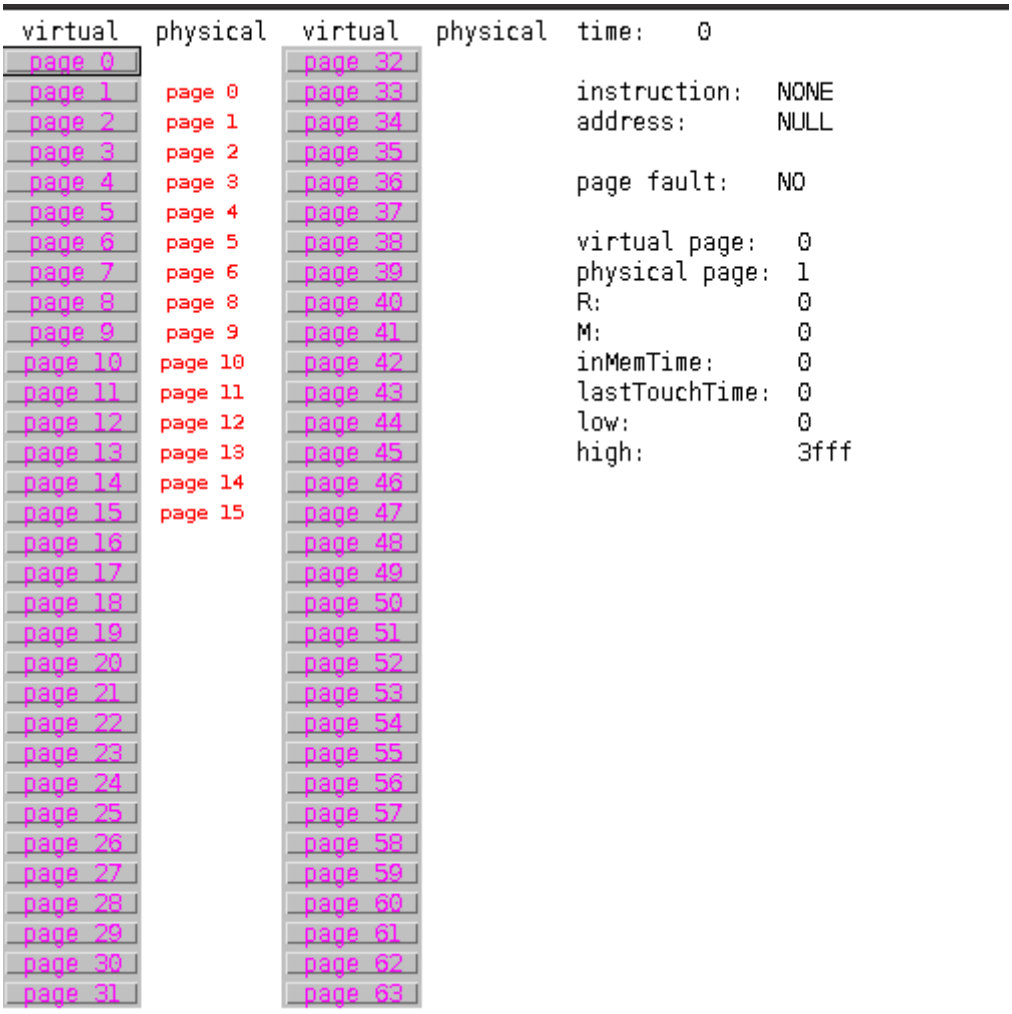

Figure 12. Initial state of the program

As we could predict, the first page fault occurred on the page 16 but now we see that two physical page 1 occurred

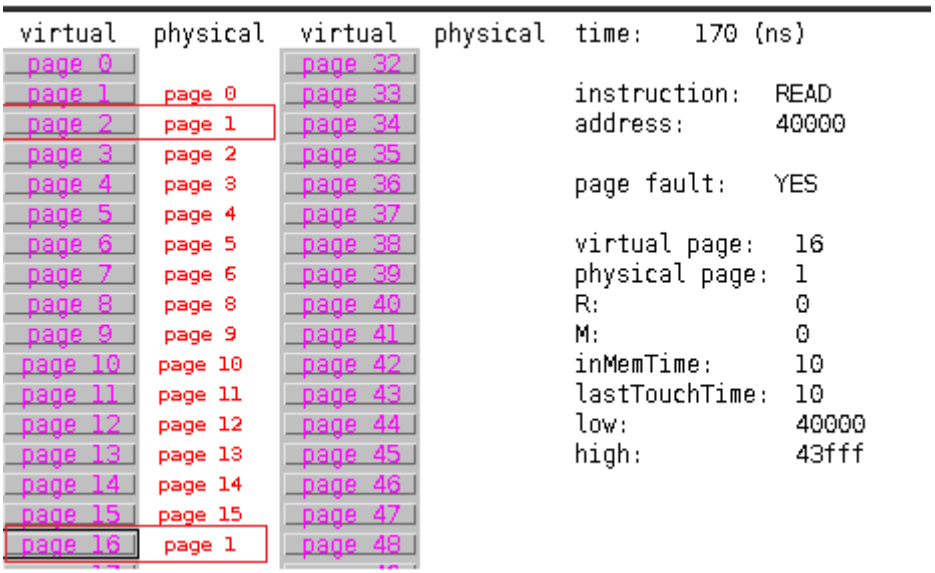

Figure 13. First page fault occurence

But when we investigate it furtherly it is just a mater of graphical user interface hardcoded values than the simulation backend code

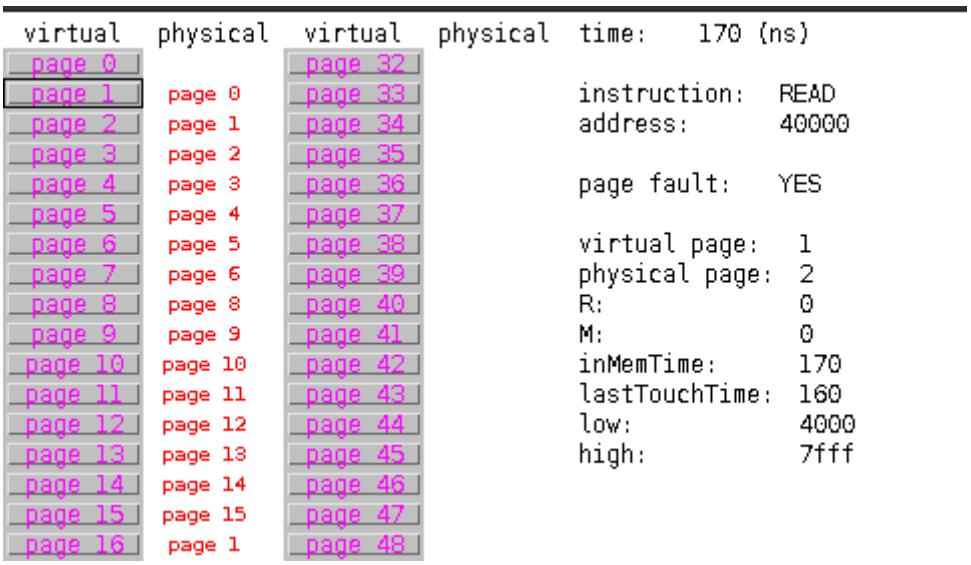

Figure 14. Mapping check

Yet, looking at the virtual page 7 and 8 we can notice that both of them share the same physical memory

```
instruction:   READ
address:       40000

page fault:    YES

virtual page:   7
physical page:  8
R:              0
M:              0
inMemTime:      170
lastTouchTime:  100
low:            1c000
high:           1ffff
```

```
instruction:   READ
address:       40000

page fault:    YES

virtual page:   8
physical page:  8
R:              0
M:              0
inMemTime:      170
lastTouchTime:  90
low:            20000
high:           23fff
```

Conclusions

Using the memory management simulation, one can possess a better understanding on how this hardware works, yet has to be careful because a lots of improvement should be done in the matters of configuration.

Resources

Geeks for geeks: https://www.geeksforgeeks.org/paging-in-operating-system/

Tomasz Kruk's EOPSY website: http://elektron.elka.pw.edu.pl/~tkruk/edu/eopsy/