Operating Systems (EOPSY)

Laboratory no. 4

Memory management

Laura Ploch

July 2021

1. Introduction

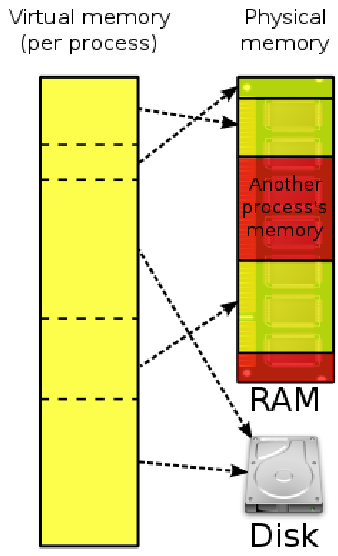


Fig. 1 Virtual memory mapping source:https://en.wikipedia.org/wiki/Virtual_memory

Memory management in operating systems is the function responsible for managing the computer's memory. It keeps track of the status of each memory location, whether it's allocated or free. It decides which process gets memory, when and how much.

Virtual memory is a memory management technique which combines the storage resources into an idealized abstraction of memory, where virtual memory addresses are mapped into physical addresses in computer memory. This allows the system to conceptually use more memory than might be physically available, using the technique of segmentation or paging.

In our case – we observe the **paging** scheme of memory management. This method divides the computer's memory into fixed-size called page frames and the virtual address space of the program into same-size blocks called pages.

Page fault occurs when a process tries to reference a page not currently present in RAM. Such invalid memory reference is treated by the processor as a page fault and then the control is transferred from the program to the operating system, which then:

- Determines the location of the data on disk
- Gets and empty page in RAM to use as a container for the data
- Loads the data into the available page frame (or a retrieved one, selected to reuse)
- Updates the page table to refer to the new page frame
- Returns control to the program

Different **algorithms** can be applied in order to solve the problem of a page fault such as: First In First Out, Last Recently Used, Second Chance, Random Page Replacement.

The simplest and most basic one is **FIFO**, where the operating system keeps track of all pages in a queue (with the oldest page on the front) and when a page fault occurs and there are no free physical pages to use, first one from the queue gets reused and is mapped to the virtual page on which the page fault occurred. In case of **LRU** (Last Recently Used) algorithm, the page that will be reused for mapping in case of a page fault is the one that has not been used for the longest amount of time.

LRU as compared to FIFO is more efficient and in case of an increase in the number of page frames, the number of page faults for a given memory access pattern doesn't increase (no Belady's anomaly). However FIFO doesn't require as much hardware support as LRU and is much easier to implement.

2. Task

The aim of this laboratory task was to configure Memory Management simulator so that it maps 8 pages of physical memory to the first 8 pages of virtual memory and then on each of the 64 virtual pages read from one virtual memory address. To do this the *memory.conf* and *commands* files were to be modified.

```
// 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 64
```

Fig. 2.1 *memory.conf* file

```
// 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                  // address to read
READ 16384              // the multiples of 16384
READ 32768
READ 49152
READ 65536
READ 81920
   (...)
READ 884736
READ 901120
READ 917504
READ 933888
READ 950272
READ 966656
READ 983040
READ 999424
READ 1015808
READ 1032192
```

Fig. 2.2 *commands* file

The *memory.conf* was edited so that the 8 pages of physical memory are mapped to the first 8 pages of virtual memory using *memset* command. The physical pages are mapped consecutively, so that the results of the simulation are clear and easier to be observed.

Additionally I changed the *addressradix* to 10 to set the radix in which the numerical values are displayed to decimal.

In *commands* file the command *READ* + address was repeated 64 times with addresses that are multiples of the size of the page – 16384, so that each time we read virtual memory address from each of the 64 virtual pages.

3. Simulation results

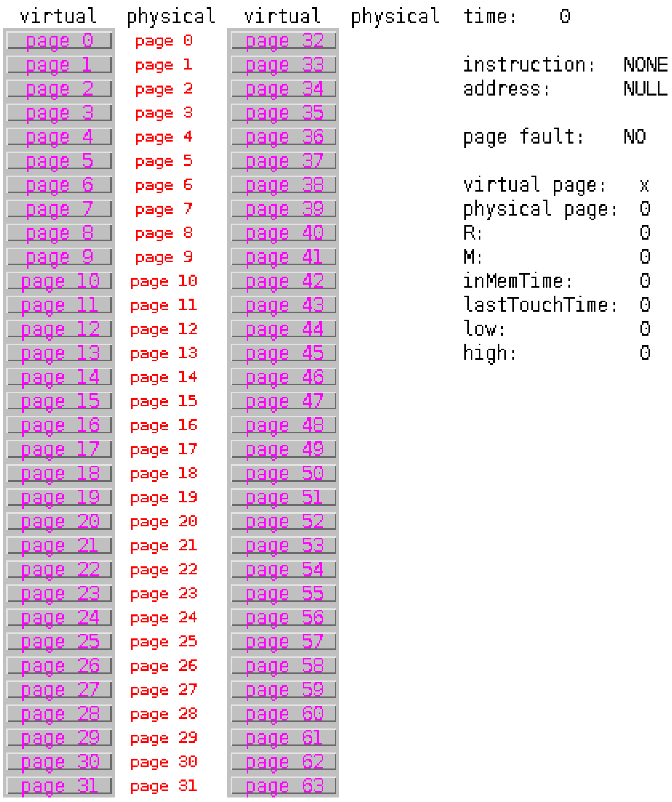
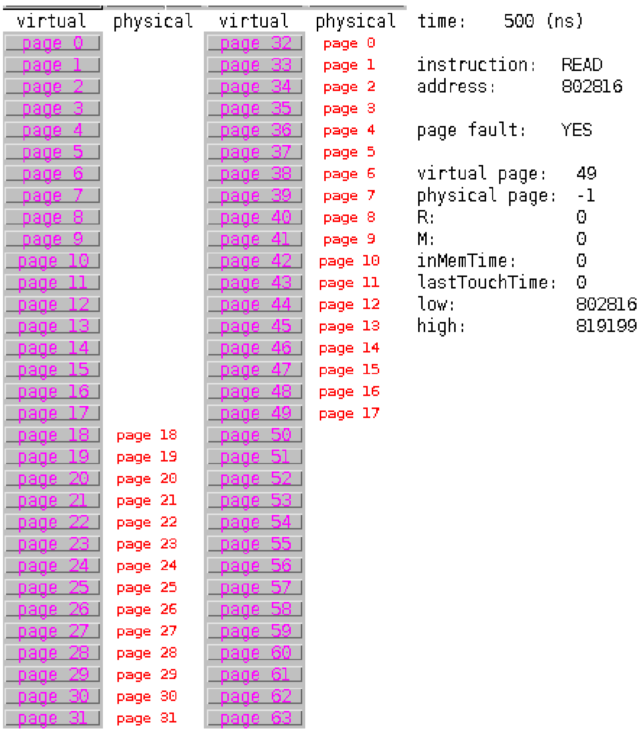


Fig. 3.1 Mapping before the start of the simulation



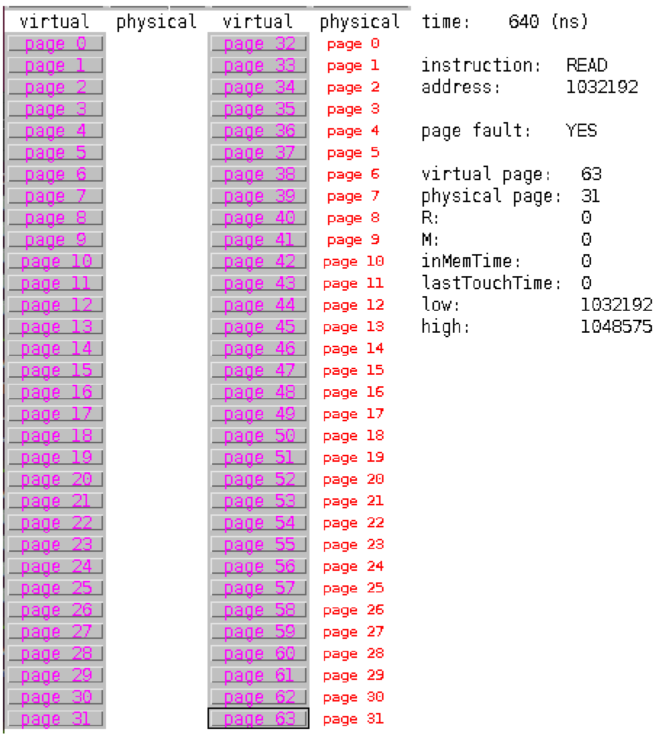


Fig. 3.2 Mapping during the simulation

Fig. 3.3 Final mapping at the end of the simulation

```
READ 0 ... okay
READ 16384 ... okay
READ 32768 ... okay
READ 49152 ... okay
READ 65536 ... okay
READ 81920 ... okay
READ 98304 ... okay
READ 114688 ... okay
READ 131072 ... okay
READ 147456 ... okay
READ 163840 ... okay
READ 180224 ... okay
READ 196608 ... okay
READ 212992 ... okay
READ 229376 ... okay
READ 245760 ... okay
READ 262144 ... okay
READ 278528 ... okay
READ 294912 ... okay
READ 311296 ... okay
READ 327680 ... okay
READ 344064 ... okay
READ 360448 ... okay
READ 376832 ... okay
READ 393216 ... okay
READ 409600 ... okay
READ 425984 ... okay
READ 442368 ... okay
READ 458752 ... okay
READ 475136 ... okay
READ 491520 ... okay
READ 507904 ... okay
READ 524288 ... page fault
READ 540672 ... page fault
READ 557056 ... page fault
READ 573440 ... page fault
READ 589824 ... page fault
READ 606208 ... page fault
READ 622592 ... page fault
READ 638976 ... page fault
READ 655360 ... page fault
READ 671744 ... page fault
READ 688128 ... page fault
READ 704512 ... page fault
READ 720896 ... page fault
READ 737280 ... page fault
READ 753664 ... page fault
READ 770048 ... page fault
READ 786432 ... page fault
READ 802816 ... page fault
READ 819200 ... page fault
READ 835584 ... page fault
READ 851968 ... page fault
READ 868352 ... page fault
READ 884736 ... page fault
READ 901120 ... page fault
READ 917504 ... page fault
READ 933888 ... page fault
READ 950272 ... page fault
READ 966656 ... page fault
READ 983040 ... page fault
READ 999424 ... page fault
READ 1015808 ... page fault
READ 1032192 ... page fault
```

Fig. 3.4 output of the simulation in the *tracefile*

```
/* It is in this file, specifically the replacePage function that will
   be called by MemoryManagement when there is a page fault.  The
   users of this program should rewrite PageFault to implement the
   page replacement algorithm.
*/

  // This PageFault file is an example of the FIFO Page Replacement
  // Algorithm as described in the Memory Management section.

import java.util.*;

public class PageFault {

  /**
   * The page replacement algorithm for the memory management sumulator.
   * This method gets called whenever a page needs to be replaced.
   * <p>
   * The page replacement algorithm included with the simulator is
   * FIFO (first-in first-out).  A while or for loop should be used
   * to search through the current memory contents for a canidate
   * replacement page.  In the case of FIFO the while loop is used
   * to find the proper page while making sure that virtPageNum is
   * not exceeded.
   * <pre>
   *   Page page = ( Page ) mem.elementAt( oldestPage )
   * </pre>
   * This line brings the contents of the Page at oldestPage (a
   * specified integer) from the mem vector into the page object.
   * Next recall the contents of the target page, replacePageNum.
   * Set the physical memory address of the page to be added equal
   * to the page to be removed.
```

Fig. 3.5 *PageFault.java* file

4. Observations

Stepping through operations during the simulation we could observe that first 8 pages that we specified in *memory.conf* file were mapped and read correctly. Moreover, the pages up to 31 (32 out of 64) also didn't produce any errors, since the mapping was probably done by default. This can be also observed in the output *tracefile* file (fig. 3.4)

However, on the 33rd page we started encountering page faults, since these page has not been mapped to any physical pages. As described in the introduction, the problem was solved by mapping consecutive virtual pages to the previously mapped physical pages (which can be observed on figures 3.1-3.3)

The algorithm of page replacement was observed to be FIFO – first in, first out – the first mapped physical page (since no free pages were available) was reused and mapped to the first virtual page on which page fault occurred and so on. This can also be confirmed in *PageFault.java* file (fig. 3.5) where it is clearly stated that the page replacement algorithm used by the simulator is FIFO (first-in first-out).

Sources:

- https://en.wikipedia.org/wiki/Memory_management_(operating_systems)
- https://www.geeksforgeeks.org/page-replacement-algorithms-in-operating-systems/
- https://www.geeksforgeeks.org/beladys-anomaly-in-page-replacement-algorithms/
- https://www.tutorialspoint.com/operating_system/os_memory_management.htm