Operating Systems - Prof. Di Carlo

OS161- Virtual Memory with Demand Paging

by Federico Perenno and Gianfranco Sarcià

Problem Definition

The project aims at improving the memory management of the teaching operating system OS161. We must implement:

- On-demand loading of pages into physical memory
- Page replacement based on victim selection
- TLB support, in particular a replacement policy for it

How does OS161 handle a user program?

NOTE: before handling a user program the **kernel**, during its bootstrap, contiguously allocates portions of memory which must not be touched by any user program.

- After the kernel bootstrap, when the operating system receives an **ELF** (Executable and Linkable Format) file, thanks to the **RUNPROGRAM** and **LOADELF** system calls, it is able to read the executable and program **HEADERS**, as well as to identify the **ENTRY-POINT** of the executable.

- An **ADDRESS SPACE** is created based on the information found in the **HEADERS** regarding the two main segments:
  - the **TEXT** segment
  - the **DATA** segment

  Such information about each segment is stored in an **ADDRESS SPACE structure**:
    - base virtual address
    - total number of virtual pages
    - total number of bytes to load from the ELF file
    - the offset within the ELF file

ELF file example

```
Program Headers:
Type     Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
REGINFO 0x000094 0x00400094 0x00400094 0x00018 0x00018 R   0x4
LOAD    0x000000 0x00400000 0x00400000 0x002d0 0x002d0 R E 0x1000
LOAD    0x001000 0x10000000 0x10000000 0x00010 0x04030 RW  0x1000
```

Program header of an example ELF file

- The **REGINFO** section isn't used
- The first **LOAD** section corresponds to the **TEXT** segment and contains the **PROGRAM CODE** and the **READ-ONLY DATA**
- The second **LOAD** section corresponds to the **DATA** segment and contains the **INITIALIZED GLOBAL DATA** (e.g. 16 bytes) and the **UNINITIALIZED GLOBAL DATA** (e.g. 16432 bytes)

- The operating system works with 32-bit **VIRTUAL** addresses, so it needs to perform a translation between **VIRTUAL** and **PHYSICAL** address.
- In order to do so the **TRANSLATION LOOKASIDE BUFFER** (**TLB**), which has more or less the following structure, is addressed:

VIRTUAL PAGE	PHYSICAL PAGE	FLAGS
20 bits	*20 bits*	

TLB HIT example:

Virtual address: 4194912 -> 00000000 01000000 00000010 01100000
Virtual page: 1024, Virtual offset: 608, with Page size: 4kB
If in the TLB: Virtual page: 1024 -> RAM frame: 3
Physical address = RAM frame x Page size + Virtual offset
Physical address = 3 x 4KB + 608 = 12896

TLB Miss

- For a **TLB HIT** the translation is done at assembly level by reading the TLB entries
- In case of a **TLB MISS** an **EXCEPTION** is generated and needs to be handled by the **VIRTUAL MEMORY** system, whose operations are performed inside of the **VM_FAULT** function:
  - it receives the **FAULT ADDRESS**, the virtual address responsible for the TLB miss
  - the **FAULT TYPE,** indicating whether the miss occurred when trying to write into or to read from the physical memory

The **VM** is able to retrieve the information regarding the address space of the running process in order to determine which segment the virtual address belongs to:

- **TEXT**
- **DATA**
- **STACK**

This will be useful later, when needing to load a page on demand from the **ELF** file.

Where's the page?

If the entry isn't in the TLB, it doesn't necessarily mean that the page hasn't been loaded into memory before

RAM

The page is **MEMORY RESIDENT**, but either the corresponding **TLB ENTRY** has been **INVALIDATED** in the meantime due to a **CONTEXT SWITCH** or it has been **OVERWRITTEN**

SWAPFILE

The page has been loaded into memory before, but because of **PAGE REPLACEMENT**, it has been moved to a **SECONDARY STORAGE UNIT**

NOWHERE

The page has **NEVER** been loaded into memory, therefore it needs to be either retrieved from the **ELF** file or **ZERO-FILLED**

TLB RELOAD

- If the virtual page has been loaded into memory before, and is still there, we simply need to locate it and to **UPDATE** the TLB with the corresponding entries
- To do so we need a **PAGE TABLE** that keeps track of which virtual pages are loaded into memory and where
- In our case, we chose to define an **INVERTED PAGE TABLE**, which is common to all processes and maps the whole physical memory. Each entry corresponds to one physical page and is addressed by providing the **PROCESS ID (PID)** and the **VIRTUAL PAGE NUMBER**

0	*PID*	*VIRTUAL PAGE NUMBER*
1	*PID*	*VIRTUAL PAGE NUMBER*
2	*PID*	*VIRTUAL PAGE NUMBER*
3	*PID*	*VIRTUAL PAGE NUMBER*
...	...	...
N-3	*PID*	*VIRTUAL PAGE NUMBER*
N-2	*PID*	*VIRTUAL PAGE NUMBER*
N-1	*PID*	*VIRTUAL PAGE NUMBER*
N	*PID*	*VIRTUAL PAGE NUMBER*

IPT

- The page table is created during the **VIRTUAL MEMORY BOOSTRAP** and **N** entries are created based on the **RAM** and **PAGE SIZE**.
- The **PID** is initialized to **-1**, while the **VIRTUAL PAGE NUMBER** is initialized to 0.
- The resulting **PHYSICAL FRAME** is given by the **INDEX** of the corresponding entry.
- If an entry has a **PID = -1** it means that no virtual page is loaded at the corresponding physical page, but it **DOESN'T** mean that the page is free.

	PID	Virtual Page Number
0	*-1*	*0*
1	*-1*	*0*
2	*-1*	*0*
3	*-1*	*0*
...	...	...
N-3	*-1*	*0*
N-2	*-1*	*0*
N-1	*-1*	*0*
N	*-1*	*0*

IPT after virtual memory bootstrap

Page Allocation

If a virtual page isn't memory resident, we must identify where in physical memory to load it

Search for a Free Page

- Search a bitmap keeping track of free pages
- A **COREMAP** is created at bootstrap and is initialized in such a way that pages allocated by the kernel aren't touched by any user process.
- The search returns the **PHYSICAL FRAME** if successful, **0** otherwise

Page Replacement

- If the physical memory is full, we must implement a replacement algorithm
- In our case we adopted the **FIRST-IN FIRST-OUT** (**FIFO**) replacement
- Once the page to be replaced has been found, we must **SWAP OUT** the resident page and copy it from the **RAM** to the a secondary storage unit, the **SWAPFILE**
- If still in the **TLB** the old entry is **INVALIDATED**

FIFO

The implementation of the **FIFO** algorithm is quite straight forward:

- user pages are allocated in **ASCENDING** order starting from the **FIRST AVAILABLE PAGE**
- we can identify the first available page by **SEARCHING** the **IPT,** looking for the first entry which has a **PID different** from **-1**
- thus, our algorithm only needs a **VARIABLE** keeping track of the **PHYSICAL FRAME** to be replaced
- this variable is initialized to the physical frame number of the **FIRST AVAILABLE PAGE** and is **INCREMENTED** by one each time a page replacement is needed
- Once the **LAST** available page is replaced, the variable is reset to its **INITIAL** value

FIFO example

Consider the following example:

	PID	Virtual Page Number
0	-1	0
1	-1	0
2	2	1024
...	...	...
N-2	3	2391
N-1	12	4616
N	1	1182

- In this case the first available page for the user is page number 2
- Therefore the **VICTIM** variable is initialized to 2
- Each time a page replacement is needed the **VICTIM** variable is incremented by one
- When **VICTIM > N,** the variable is set once again to 2
- Since pages are allocated in ascending order from **2 to N**, replacing them starting from **2 up to N** is coherent with the **FIRST-IN-FIRST-OUT** algorithm

SWAPFILE PAGE FAULT

- Page replacement causes pages to be copied from the **RAM** to the **SWAPFILE**
- We must keep track of the **PID** and **VIRTUAL PAGE NUMBER** of the pages that have been swapped out in a **SWAPFILE TABLE**, which behaves similarly to the **IPT**
- By dividing the **SWAPFILE** in frames of 4kB, each one addressed by a **SWAPFILE TABLE ENTRY**, it becomes easy to move pages between primary and secondary storage unit

0	*PID*	*VIRTUAL PAGE NUMBER*
1	*PID*	*VIRTUAL PAGE NUMBER*
2	*PID*	*VIRTUAL PAGE NUMBER*
3	*PID*	*VIRTUAL PAGE NUMBER*
...	...	...
M-3	*PID*	*VIRTUAL PAGE NUMBER*
M-2	*PID*	*VIRTUAL PAGE NUMBER*
M-1	*PID*	*VIRTUAL PAGE NUMBER*
M	*PID*	*VIRTUAL PAGE NUMBER*

OTHER PAGE FAULTS

If the page has never been loaded into memory, we have two separate options:

Load it from the ELF file

If:

- the fault-address belongs to the **TEXT** segment
- the fault-address belongs to a page of the **DATA** segment which contains at least a portion of **INITIALIZED GLOBAL DATA**

Fill the whole page with ZEROS

If:

- the fault-address belongs to the **STACK** segment
- the fault-address belongs to a page of the **DATA** segment which corresponds entirely to **UNINITIALIZED GLOBAL DATA**

ZERO FILLING

There are two instances in which the **Virtual Memory** system may want to initialize all the bytes in a page to **ZERO.**

- we're assigning a page to the **STACK**, which needs to be initialized to zero the first time it is referenced

- the same can be said for **UNITIALIZED VARIABLES**, which belong to the **DATA** segment, but don't have to be loaded from the **ELF** file

This operation is performed by the **BZERO** function, which is already defined in OS161.

LOAD from ELF file

1. **WHERE** to start reading in the ELF file
2. **HOW MANY** bytes we need to read
3. Are virtual pages **ALIGNED** with the physical ones

These problems can be solved using the information gathered when we first read the program **HEADERS** of the **ELF** file.

In fact for each segment, the **OFFSET** within the **ELF** file is specified along with the amount of **BYTES** to be read from it.

Also the **FIRST VIRTUAL ADDRESS** of the segment is specified, from which we can determine whether the virtual address space is misaligned or not.

Type	Offset	VirtAddr	FileSiz	MemSiz
LOAD	0x000000	0x00400000	0x002d0	0x002d0
LOAD	0x001000	0x10000000	0x00010	0x04030

LOAD from ELF file

- The **ELF FILE OFFSET** is incremented by a multiple of the **PAGE SIZE** (4kB) depending on the **PAGE OFFSET** within the virtual address space and reduced according to **MISALIGNMENT**

- The **PHYSICAL ADDRESS** is given by the **PHYSICAL PAGE** found during **PAGE ALLOCATION** in all cases except one: when we're loading the **FIRST VIRTUAL PAGE** of a segment in memory and there is misalignment in which case we add to the **PHYSICAL PAGE** a **MISALIGNMENT OFFSET** so to achieve **PAGE ALIGNMENT** between virtual and physical pages

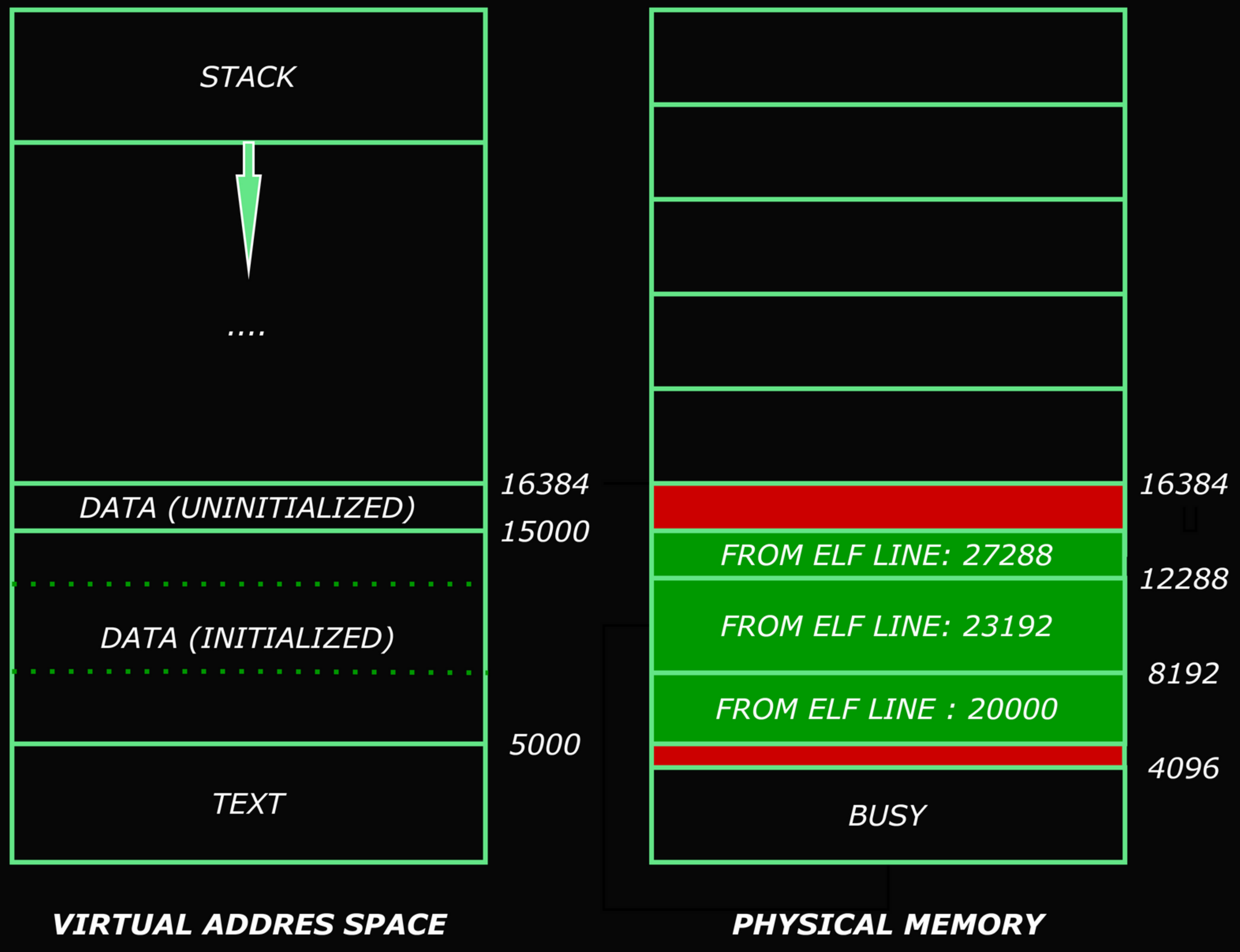
- The **AMOUNT OF BYTES** to be read depends on the **FILE SIZE** specified in the **ELF** file header and on which **VIRTUAL PAGE** within the segment we're trying to load:
  - For the **FIRST** page we check if the **FILE SIZE** minus the **MISALIGNMENT OFFSET** is bigger than the **PAGE SIZE** and if that's the case we load 4kB minus the **MISALIGNMENT OFFSET** , otherwise we load the specified amount of bytes and zero-fill the rest
  - For the **LAST** page we compute the difference between the **FILE SIZE** and **PAGE SIZE** (minus the **ALIGNMENT OFFSET**) and only take the lower 12 bits
  - For **ANY INTERMEDIATE** page we know that 4kB need to be loaded

LOAD from ELF file - EXAMPLE

Let's assume we have the following information regarding the **DATA** segment:

- vbase = 5000
- vtop = 16384
- file_size = 10000
- elf_offset = 20000

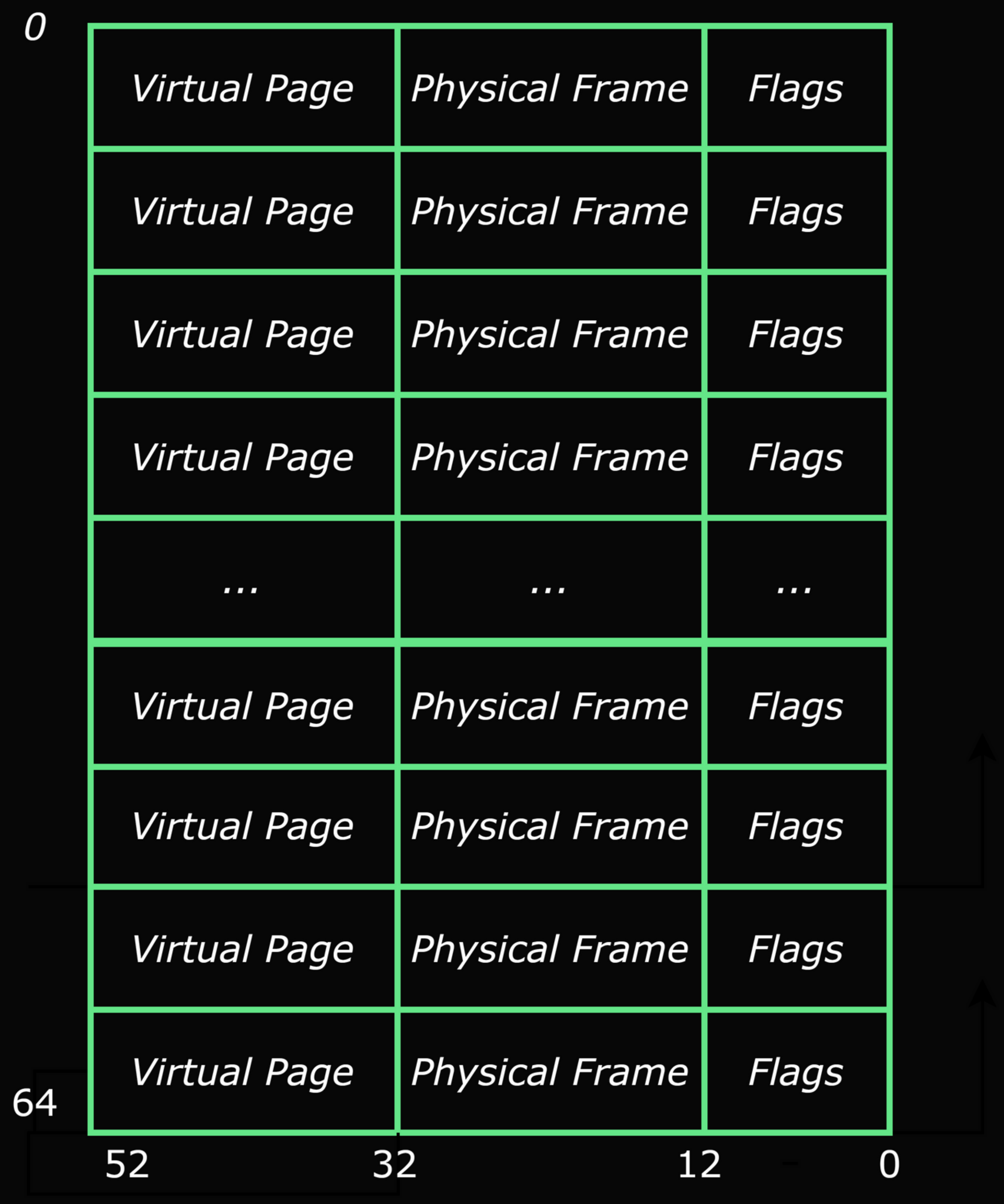
NOTE: here the virtual pages are loaded from bottom to top, but it works also if they are loaded in random order



TLB UPDATE

Once we know the **VIRTUAL PAGE**, the **PHYSICAL ADDRESS** and whether the virtual address corresponds to the **TEXT** segment or not, we can update the **TLB.**

- we exploit the **TLB_READ** function to search the **TLB** checking the **VALID** bit to find an **INVALID ENTRY**
- if **NO** invalid entry is found we replace one entry with a similar-**FIFO** algorithm, which doesn't however account for page replacements
- thanks to the **TLB_WRITE** function we are then able to write the entry in the **TLB,** setting the **VALID** bit to 1, and the **DIRTY** bit to 1, unless the corresponding virtual page belongs to the **TEXT** segment



VALIDATION

In order to validate our **VIRTUAL MEMORY** system we implemented basic **SYS_READ, SYS_WRITE**, and **SYS_EXIT** system calls in order to **SUCCESFULLY RUN** the following programs, contained in the the **TESTBIN** folder:

- **ZERO**: checks if the Virtual Memory system zeros memory like it's supposed to, by first loading both initialized and uninitialized data and by then checking if it has been loaded correctly.
- **HUGE:** creates an uninitialized 8 MB data array and manipulates it by first overwriting it and by then checking if it has been written correctly
- **SORT**: creates an array of 147456 random integers and then sorts it using quicksort
- **PALIN**: creates a long string and checks if it is palindromic
- **MATMULT**: multiplies two large matrices together and checks the result

Thank you for listening!