Politecnico di Torino

Master Degree in Electronic Engineering

A.A. 2021/2022

Operating Systems: Virtual Memory with Demand Paging

Student:
Alaimo Davide 290161
Stabellini Valentina 287756

Contents

1 Introduction

The **OS161** operating system is a very simplified version of modern operating systems that presents a simulated CPU, a very simplified virtual memory system called **DumbVM**, a system bus, and bus devices that are realistic yet simple and easy to use. The aim of this project is improving the virtual memory management of the **OS161** by replacing the DumbVM with a system that implements the **demand paging and the swapping technique**.

2 Virtual Memory

2.1 VM fault

The `vmfault()` function is the core of virtual memory management, so it determines the correct behavior of the operating system.
When a TLB miss occurs, the base fault address is researched into page table by means of `page_int_pt()` which returns `paddr` that can be zero or not. If `paddr` is not zero means there is a matching in page table so we just update the TLB `TLB_insert_entry()` by inserting the new entry.
If `paddr` is zero means there is not the page into main memory so we have to add the new page from disk to memory. A new physical address `new_paddr` is chosen by `page_replecement` in which the base fault address will be inserted. Since the page was not found in memory, it is searched into the disk by means of `swap_in()`. If there, only updating TLB with the new address traslation, but if not present in disk it is important to determine the segment type because the requested page could be loaded from elf file only for data and text segment. So in text and data cases, let's take the requested page from elf and then update the TLB.

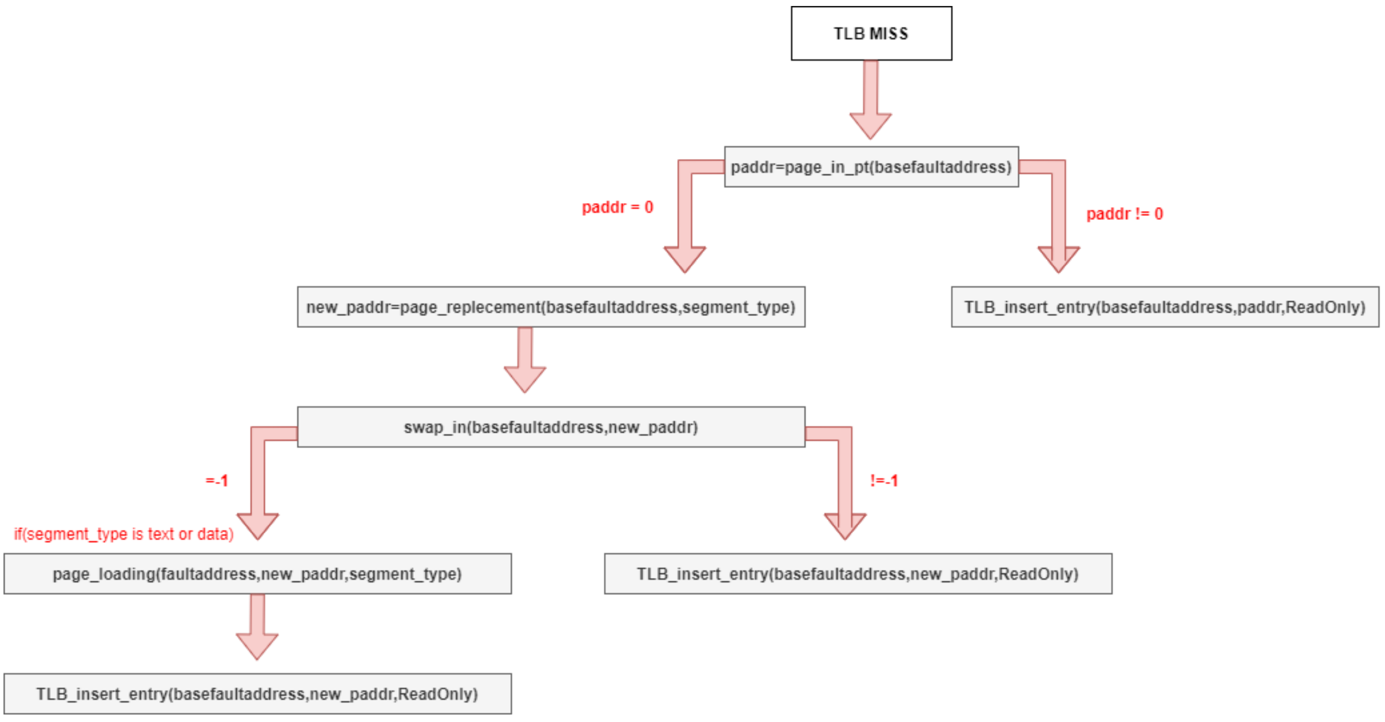

TLB MISS
paddr=page_in_pt(basefaultaddress)
paddr = 0
paddr != 0
new_paddr=page_replecement(basefaultaddress,segment_type)
TLB_insert_entry(basefaultaddress,paddr,ReadOnly)
swap_in(basefaultaddress,new_paddr)
=-1
!=-1
if(segment_type is text or data)
page_loading(faultaddress,new_paddr,segment_type)
TLB_insert_entry(basefaultaddress,new_paddr,ReadOnly)
TLB_insert_entry(basefaultaddress,new_paddr,ReadOnly)

Figure 2.1: VM fault

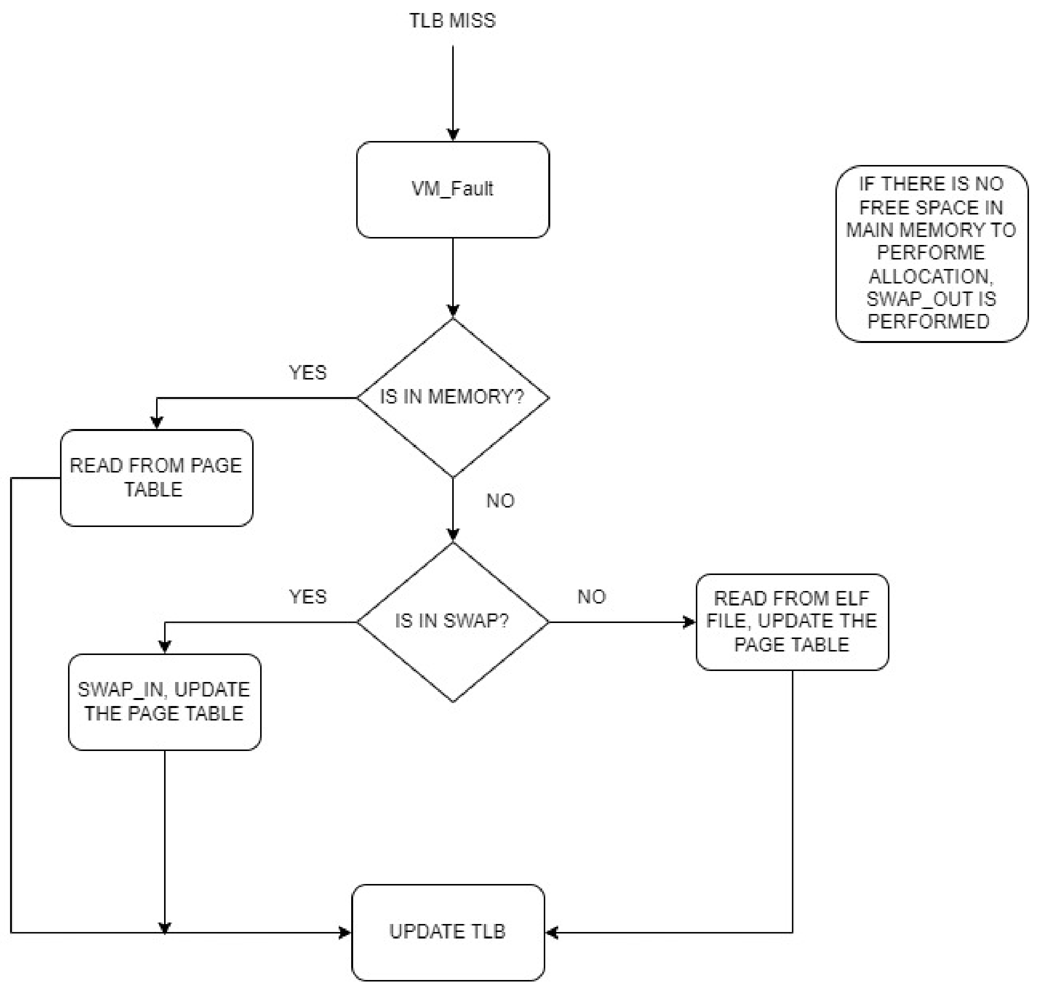

TLB MISS
VM_Fault
IS IN MEMORY?
YES
NO
READ FROM PAGE TABLE
IS IN SWAP?
YES
NO
SWAP_IN, UPDATE THE PAGE TABLE
READ FROM ELF FILE, UPDATE THE PAGE TABLE
UPDATE TLB
IF THERE IS NO FREE SPACE IN MAIN MEMORY TO PERFORME ALLOCATION, SWAP_OUT IS PERFORMED

Figure 2.2: General scheme

2.2 TLB Management

The TLB has been managed as follows:

- `static int tlb_get_rr_victim(void)`: updates the victim index when a replacement occurs.
- `int TLB_invalidate_entry(paddr_t paddr)`: removes entry in TLB by invalidating that entry. The searching is done when a matching between physical address and low part of entry, occurs.
- `int TLB_retrieve_index(vaddr_t vaddr)`: retrieves the index by providing the virtual adrress. It is a masked way to use the low-level function `tlb_probe`.
- `int TLB_invalidate_all(void)`: invalidates the whole TLB. It is called when a context switch or an activation of address space occur.

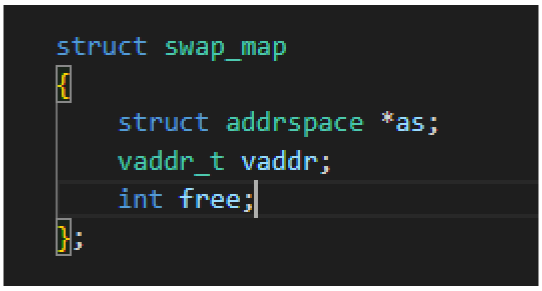
- `int TLB_insert_entry(vaddr_t faultaddress, paddr_t paddr, bool ReadOnly)`: is the main function of virtual memory management. What it does is checking if any invalid entry could be present. If so, the virtual address and the physical address are storend in TLB. On the other hand, if there are no invalid entries, the TLB replacement is performed by getting the victim index.
  When a new entry is inserted, the valid bit is set in order to indicate that entry contains a valid traslation from virtual to physical address. In addition, another parameter is considered such as the dirty bit that if is clear means that entry is a read-only one. So in the function, ReadOnly is a boolean variable which is true when dirty bit have to be clear.

2.3 Read-Only Text Segment

For this assignment, the application's text segment is read-only so the kernel should set up TLB entries properly so that any attempt by an application to modify its text section will cause a read-only memory exception VM_FAULT_READONLY.
So when a process tries to write on a read-only page, `vm_fault()` is invoked with fault type VM_FAULT_READ_ONLY and this process will be killed by `sys__exit()`.
In `vm_fault()` fault address can be compared to boundary virtual addresses' segments determining segment type. If segment type is a code, when an entry will be insert in TLB, boolean `ReadOnly` is true so dirty bit will be set to zero, dirty bit set to one otherwise.

2.4 SwapFile

The SwapFile represent the disk of our system and it is implemented by using a bootstrap function callback at the startup that open the Swapfile, that is a file created inside the `emu0:` directory and presents an editable size by changing a parameter inside the `swapfile.h`.
As requested this size is setted by default to 9MB, that are divided in slots and each of theese slots are represented by the *swap_map* structure that acts like a bitmap and is made up by:



```
struct swap_map
{
    struct addrspace *as;
    vaddr_t vaddr;
    int free;
};
```

Figure 2.3: Swap map struct

The management of the swpafile is supported by several functions:

- `swap_bootstrap()`: Is the initialization function of the Swap.It takes care of opening the file, allocating the *swap_table* in memory, using the kmalloc, and then initializing it with the `swap_free()`.

- `swap_destroy()`: it is used at the shutdown to clean the used structures.
- `swap_free(struct addrspace *as, vaddr_t vadd, int conf)`: this function presents differents configurations and it is used to clean a particular slot inside the swapfile.In particular this function is used, with the configuration number 2, in the initialization step to clean all the slot inside the swapfile.
- `int swap_tracker(vaddr_ vaddr)`: It is used to search inside the swap file a particular *vaddr*, and and if it finds it returns its position.
- `int swap_in(vaddr_t vadd, paddr_t new_paddr)`: this function moves a frame from the swap file to the main memory. First of all it checks if the virtual address searched is present on the disk, by using the `swap_tracker()`, then copy the info from the disk to the main memory without deleting the data from the swap.
- `int swap_alloc()`: finds a free position inside the *swap_table* that is used to allocate a new data inside the swap file.
- `swap_set(int index, vaddr_t vaddr,struct addrspace *as)`: Sets the different fields of the structure *swap_table* when a new data is inserted inside the *swap_file*.
- `int swap_out(paddr_t paddr,vaddr_t vaddr)`: It moves a frame from the main memory to the *swap_file*.First of all the `swap_tracker()` is used to avoid multiple copies inside the disk, then if the data is not already inside the *swap_file* find a free location with the `swap_alloc` and moves the frame there.

3 On demand paging

3.1 Read_elf instead of load_elf

Since the program doesn't have to be loaded into main memory, the `load_elf()` function has been replaced with `read_elf()` in which no segments are loaded into memory, but address space parameters are setted by means of `as_define_region()` function which allows to update the informations regarding first virtual address, last virtual address, number of pages and base virtual address of each segment.
Then, elf offset into the file and file size are inserted in address space as well which will be usefull in page loading from elf file into memory. To do notice that elf offset and file size keep being associated with each segment type.
In `read_elf()` the pointer to the file is also saved in the address space struct.

3.2 Runprogram changes

Original `runprogram()` function plans to open elf file first, create a new address space, switching to it, activating it, loading elf file, closing it and then defining the user stack in the address space.

So for our purpose, the parts related to loading elf and the next closure of it, have been replaced by simply reading of the file and the initialization of the page table. So the file elf is maintained as open for all program duration and it will be closed during `as_destroy()` called when the program is complete or interrupted.

3.3 Page_loading

The `page_loading()` function is responsible for the demand page loading, so it's called when a page is not present in main memory and disk.
To extract the correct parameters, we have to consider three possible scenarios since the first virtual address of a segment could not starting with a page-aligned address.
We will indicate as `size` the amount to read from the elf file and `offset` the starting offset to read in elf file.

- The fault page belongs to the first page of the segment:
  `size` is determined as shown in figure 3.1 while `offset` in this case will be the starting offset of the segment in elf file which is stored in address space structure.

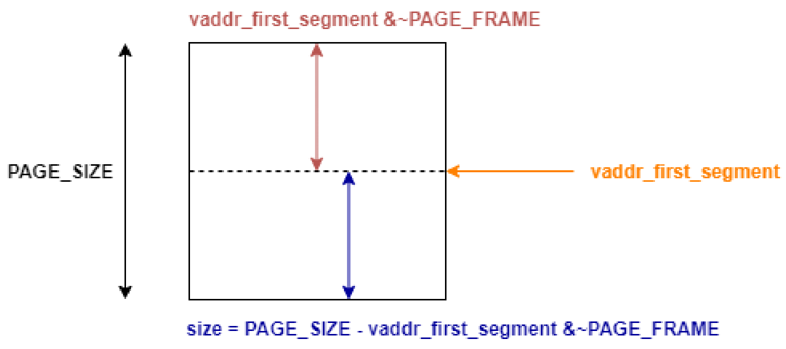

vaddr_first_segment &~PAGE_FRAME
PAGE_SIZE
vaddr_first_segment
size = PAGE_SIZE - vaddr_first_segment &~PAGE_FRAME

Figure 3.1: First page of the segment in elf file

- The fault page belongs to the last page of the segment:
  `size` is determined how figure 3.2 suggests and `offset` in figure 3.3.

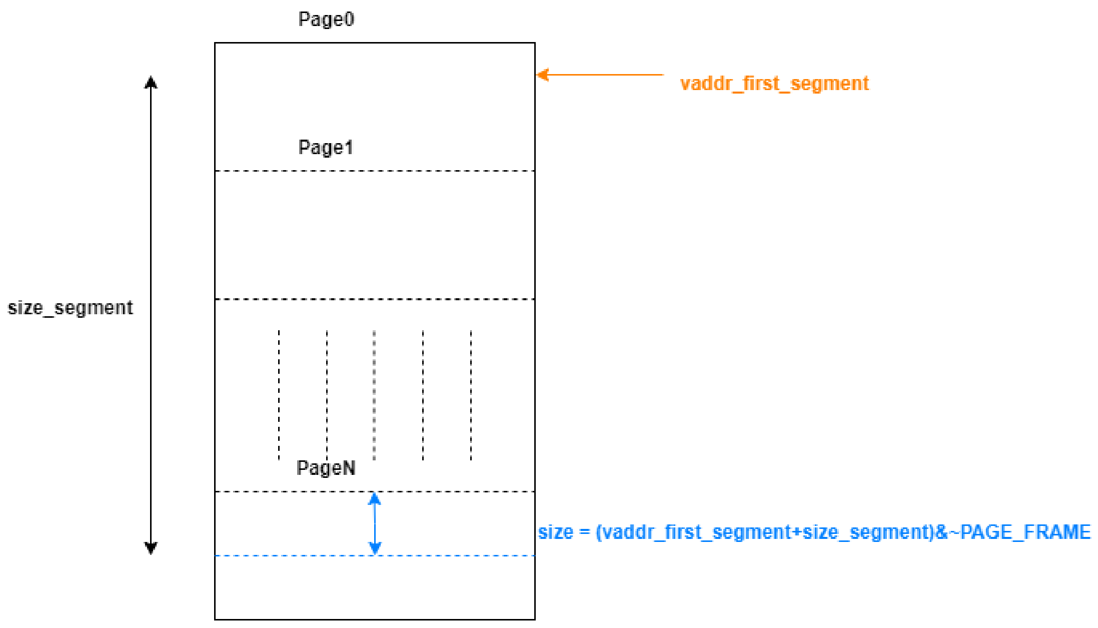


Figure 3.2: Size in last page of the segment in elf file

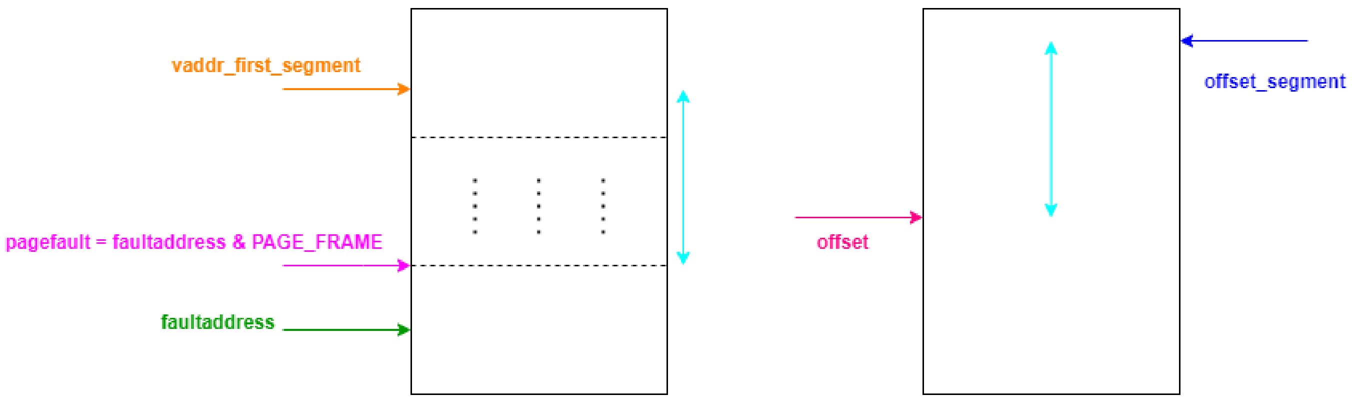


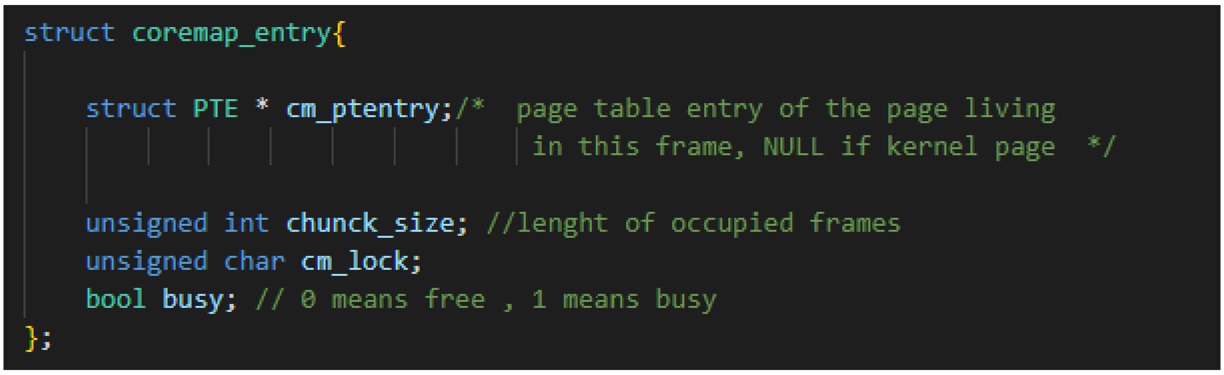
Figure 3.3: Offset in last page of the segment in elf file

- The fault page belongs to an intermediate page of the segment:
  In this case the amount to read `size` is the dimension of a page so 4KB and the starting `offset` is the same as figure 3.3 defines.

4 Coremap

The Coremap is used to manage the memory and replace the ram.c, of the *DumbVM*, in the memory bootstrap steps.
It has the purpose of tracking and mapping all the memory and in order to do that the memory

has been divided in `frame_slots` and to each of these is assigned a `coremap_entry`, struct that presents the following fields:



```
struct coremap_entry{

    struct PTE * cm_ptentry;/*  page table entry of the page living
                                 in this frame, NULL if kernel page  */

    unsigned int chunck_size; //lenght of occupied frames
    unsigned char cm_lock;
    bool busy; // 0 means free , 1 means busy
};
```

Figure 4.1: Coremap struct

The two main functions of the Coremap are the `coremap_getppages()` and the `coremap_freeppages()`, the first used in the `getppages()` for the memory allocation and the second used for the deallocation in the `freeppages()`.In this regard it is important to remember that the user allocation is one page at a time, while the kernel one allows contiguous allocation.
Analyzing more in detail the `coremap_getppages()`, it exploits the fields of the `coremap_entry()` structure to search free frames with the `coremap_find_freeframes()`, in fact the coremap track all the free and the busy frames inside the main memory.After that update the coremap structure and returns to the `getppages()` the start frame from which is possible to write.
If there is no free frame inside the memory the `coremap_swapout()` is called in orde to move one or more frames to the disk and then create the necessary space for the n requested pages.
On the other side the `coremap_freeppages()` starts from a phisical address, obtains the frame slot inside the coremap and clear the relative fields, in this way that slot is seen as free from the coremap and then can be reallocated.

5 Address Space

During `runprogram()` function, the elf file is open and `as_create()` is performed in order to return a void struct address space in which that struct is allocated in kernel space and internal parameters are initialized to useless values. Then the elf file is read, and not loaded into memory, by saving boundary virtual addresses's segments via `as_define_region()` and informations' segments about dimension and offset into the file elf.
In order to correctly implement demand paging, the elf file pointer node is not closed but kept open to allow taking page when requested during the program running.
Since address space contains also the page table pointer, the previous parameters have been initialized by reading file elf, but page table needs to be initialized as well. To do that, we implemented a dedicated function `initialize_pt()` which performs the not-contiguous user allocation by means of `getppages()` that returns one at time a physical address to store in

page table. As last step the `as_define_stack()` is called to set user stack virtual address in stack pointer.
The deletion of address space occurs when `as_destroy()` is invoked from process destruction. De-allocating user space (physical addresses kept in PT) of page table using `freeppages()`, making free the swap pages by means of `swap_free()`, closing the elf file, and de-allocating again the kernel space, previously returned from kmalloc, where the address space structure lived through `kfree()`.

5.1 Address Space structure

The address space structure stores the following field:

- `vaddr_t as_first_vaddr1`: first actual virtual address of text segment
- `vaddr_t as_last_vaddr1`: last virtual address of text segment computed as `as_first_vaddr1` + `memsize`
- `size_t as_npages1`: number of pages occupied by text segment into main memory
- `vaddr_t as_first_vaddr2`: first actual virtual address of data segment
- `vaddr_tas_last_vaddr2`: last virtual address of text segment computed as `as_first_vaddr2` + `memsize`
- `size_t as_npages2`: number of pages occupied by data segment into main memory
- `off_t offset_text_elf`: offset of starting text segment into elf file
- `size_t text_size`: size of text segment into elf file
- `off_t offset_data_elf`: offset of starting data segment into elf file
- `size_t data_size`: size of text segment into elf file
- `struct PTE *page_table[N_FRAME]`: pointer to page table
- `struct vnode *v`: pointer to elf file

To do notice an important difference between `memsize` and `filesize`: the first one indicates the dimension of the segment into memory and the second one indicates the dimension of the segment into elf file that cannot be the same. It is usefull highlighting that because are used for two different scopes.

5.2 Page Table

The page table implemented is a per-process page table with dirty bit support.
Dirty bit is a bit associated to a memory page and is set to zero when that memory page is present in swap space. It is set to one when the page is modified instead, but is not present in swap space. Modified means the processor executes write operation on it.
When a new page should be inserted in page table, if there are not free enties, `page_replecement()`

is performed. Replacement, during victim page selection, privileges pages marked as clear dirty bit (the ones which have a copy in swap space) in a way not to make a swap out. In the case any pages have not dirty bit set to zero, code pages have higher priority to be selected as victim. Once victim has been chosen, it's important to evaluate if a page is needed to be swapped out or kept. If the victim page has dirty bit set (means referred page has been modified), swap out is required because no copy exists to the disk. Otherwise, if victim page has no dirty bit set implies the page already has a copy to disk, but that copy has to be evaluated because it could not be valid yet since it could contain oldest informations. When does old copy should be considered as obsolete? When a modify on it has been performed. As said before, modify means writing on memory block and it can occurs only in data and stack segments (data one is read-only).
Consequently, swap out runs for cases in which dirty bit is one or dirty bit is zero, but the victim is not a page belonging to text segment.
In the last step in page replacement, checking if victim page in page table has a copy in TLB, if so it will be deleted e replaced with new one.

5.2.1 Page Table functions

- `int initialize_pt(struct addrspace *as)`: page table initialization called by `runprogram()`.
- `paddr_t page_in_pt(vaddr_t vaddr)`: checks if vaddr is present in page table, if so returns associated paddr, -1 if no matching.
- `static int page_victim()`: page victim selection algorithm. It simply increments counter inside page table.
- `int set_dirty_bit_pt(bool dirty, paddr_t paddr)`: sets dirty bit in page table. Called in `vm_fault()`, 1 if new page inserted comes from elf file loading, 0 if page was found in swap space.
- `bool check_if_any_dirtyzero()`: function checks if there is at least one page with dirty bit set. It return true if a page is found, false otherwise.
- `paddr_t page_replacement(vaddr_t vaddr, unsigned int segment_type)`: called in `vm_fault()` when a new entry has to be inserted. It finds some free pages first, if not found page replecement is applied.

6 Final results and tests

6.1 Statistics

- TLB Faults: The number of TLB misses that have occured (not including faults that cause a program to crash).
- TLB Faults with Free: The number of TLB misses for which there was free space in the TLB to add a new TLB entry (no replecement required).

- TLB Faults with Replace: The number of TLB misses for which there was no free space for the new TLB entry, so replecement was required.
- TLB Invalidations: The number of times the TLB was invalidated.
- TLB Reloads: The number of TLB misses for pages that were already in memory.
- Page Faults (Zeroed): The number of TLB misses taht required a new page to zero-filled.
- Page Faults (Disk): The number of TLB misses that required a page to be loaded from disk.
- Page Faults from ELF: The nnumber of page faults that require getting a page from the ELF file.
- Page Faults from Swapfile: The number of page faults that require getting a page from swap file.
- Swapfile Writes: The number of page faults that require writing a page to the swap file.

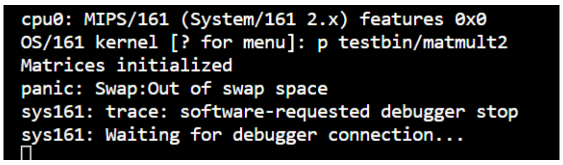
After collecting statistics, they have to respect the following equalities. This is implemented by `KASSERT` immediately upon printing.

- TLB_faults = TLB_faults_reload + page_faults_disk + page_faults_zeroed
- TLB_faults = TLB_faults_free + TLB_faults_replace
- page_faults_elf = page_faults_swapfile + page_faults_disk

6.2 User programs

	palin	huge	matmult1	matmult	matmult2	ctest
Execution time	15.43s	35.77s	18.44s	6646s	-	900s
TLB Faults	13771	6956	5025	1434673	-	201118
TLB Faults with free	13771	6742	5000	1434673	-	188875
TLB Faults with Replace	0	214	25	0	-	12243
TLB invalidations	7763	6100	3227	1150814	-	154397
TLB Reloads	13766	3944	3414	859273	-	123888
Page Faults Zeroed	1	1	1	1	-	1
Page Faults Disk	4	3011	1610	575399	-	77229
Page Faults from Elf	4	514	104	382	-	259
Page Faults from Swapfile	0	2497	1506	575017	-	76970
Swapfile Writes	0	2912	1511	575300	-	77130

In user test `matmult2` the matrix dimension is selected in order to get a panic out of swap space. The statistics are not available due to interruption but the error is demostrated in figure 6.1.



```
cpu0: MIPS/161 (System/161 2.x) features 0x0
OS/161 kernel [? for menu]: p testbin/matmult2
Matrices initialized
panic: Swap:Out of swap space
sys161: trace: software-requested debugger stop
sys161: Waiting for debugger connection...
```

Figure 6.1: matmult2 error

6.3 Kernel tests

Test name	Passed
at	True
at2	True
km1	True
km2	True
km3 10000	True