Politecnico di Torino

MASTER DEGREE IN COMPUTER ENGINEERING

Report project System and Device Programming

OS Internals – Projects on OS161, A.Y. 2021/2022 (Cabodi)

Project c1: Virtual Memory with Demand Paging (PAGING)

Students

Ivana Orefice

Sergio Petrone

Concetto Antonino Privitera

Academic Year 2021-2022

Contents

Chapter 1

Introduction

written by Ivana Orefice

Before talking about the goal of this project, it is good to introduce OS161, which is an Operative System created just for didactical purposes. Indeed, it offers a kernel with a simplified scheduler, alongside a very limited virtual memory system.

1.1 Objectives

The goal of this assignment has been the one of developing a **new virtual-memory system** that relaxes some of dumbvm's limitations, as well as implementing **Demand Paging**, which will allow the programs to have address spaces larger than physical memory to run, with an **Inverted Page Table**, which is defined as IPT in the following pages, also implementing an algorithm of **Page Replacement**. This has required the **implementation of a new TLB support**, to manage it in a way that the kernel will not crash.

1.2 Preliminary Study

In order to completely get the behaviour of OS161 and better understand which were the files on which to intervene, the group started by carefully reviewing the code of OS161, particularly focusing on the /kern folder. Specifically, we studied:

- *dumbvm.c* file to get intuition on how the DUMBVM allocator and TLB management works
- *loadelf.c* to get intuition of ELF files
- *tlb.c* and *tlb.h*, to understand the flow of TLB management
- *menu.c* and *main.c*, studied to perform a smoother debugging phase.

1.3 Action strategy

After the preliminary studies were completed, we prepared an action strategy, which aim to implement the following features:

- Inverted Page Table: this feature, implemented in file *pt.c* and *pt.h*, is described in Chapter 2. The implementation in this chapter has been carried out by Ivana Orefice and Sergio Petrone. Those files report our proposal of the definition of the IPT, alongside functions which allow to implement initialization, add entry, remove entry and other IPT functionalities, needed for the implementation. Moreover, here is the place in which we implement **Page Replacement algorithm**.
- TLB Management: this part is covered in the Chapter 3 and carried out by Concetto Antonino Privitera. The implementation to manage the TLB and make it work with the IPT is located in *vm_tlb.c* file. Here, as for the page table management, is included all functions needed to initial, modify and reset the TLB entries.

- Swapfile: this feature is implemented into files named *swapfile.c* and *swapfile.h*, and it aims to implement the Management of Swapping In and Out one or more pages from the IPT. Swapfile is crucial in order to implement On-Demand Page Loading, and you can find more about it in Chapter 4 carried out by Concetto Antonino Privitera.

- *addrspace.c*: this is the most important file because it is the place in which the group managed to put every single functionality together: in fact, the *vm_fault* function is implemented right here, and it is very important because it is the "trigger" function, in the sense that when we look for a specific page and the page is missing, a **Page Fault** happens and this function manages it. The reader can find a more detailed specification in Chapter 5. The implementation has required the participation of all team members.

- Finally, to track and print the required statistics related to the performance of the virtual memory sub-system (including TLB misses and TLB replacements), the group has implemented a specific file, named *vm_stats.c*. The discussion about the statistics can be found in Chapter 6. The implementation in this chapter has been carried out by the team entirely.

We will dedicate a chapter for each of the topics we've covered, including pieces of code and where to find them into out GitHub repository, which will be linked in the follow *github repository*.

Another aspect to highlight is the usage of private *spinlock* for each feature to implement and used in case it is required mutual exclusion in the critical section. Also, we decided to mainly keep the structures used inside the file *.c* with the code in a way to make everything visible in the same file.

Chapter 2

Page Replacement and IPT

carried out by Ivana Orefice and Sergio Petrone

As said in the Introduction, the implementation choice about the Page Table we took has been the Inverted Page table, very often abbreviated into IPT.

From a theoretical point of view, the Inverted Page Table is an alternative solution, oriented to reduce space limitations of big page tables. This table is basically an array based on the following intuition: instead of using P (page number) as index and F (frame number) as content of an array entry, we do exactly the reverse, that is using P as content and F as index.

The entire implementation can found inside the dedicated files, which are pt.c and pt.h. Because we've chosen to implement an IPT, every entry in the table represents a frame inside the RAM memory, and so the implementation has been split into the implementation of the single entry and the implementation of the entire table as a collection of entries, plus some other parameters. The first structure is represented by the following structure:

```
typedef struct entry {
    vaddr_t vaddr;
    pid_t pid;
```

```
        uint32_t status;
        permission_t permission_flag;
        int position_fifo;
    } entry_t;
```

As we can see, the structure has the following fields:

1. *vaddr*, which represents the virtual address of the page
2. *pid*: this is the Process ID of the process sharing the page. We don't need to store address space pointer as we can index into process table using pid and can get the address space by accessing the thread structure.
3. *status*. This field is used to represent which is, indeed, the status of the page, so if it is a Kernel Page, a Dirty one, a Free or a Clean one.
4. *permission_flag*, which gives information about the page being a Read-Only or Read-Write one.
5. *position_fifo*: used to know when the page was added into the page table. Its purpose will be shown better when talking about the Page Replacement.

About the structure that represents the entire IPT, this will be the following:

```
typedef struct table {
    entry_t * next_entry;
    unsigned int length;
    struct spinlock table_lock;
} table_t;
```

Fields, and their meaning, are:

1. *next_ entry*, which, clearly, represents a pointer to the next entry of the IPT.
2. *lenght*, which represents how long the IPT is.
3. *table_ lock*, which is the spinlock, useful to do some operations in mutual exclusion.

2.1 Methods implemented

First of all, an init method was needed, just to initialize the IPT in its entirety, and this is done through the kmalloc function. Of course, we have allocated space for the table_t structure, but also the entries. Indeed, the function page_table_init takes, as an argument, the **lenght** parameter, used when allocating, through kmalloc, the entries that we need to have inside the IPT. Clearly, every entry's field must be initialized: we've chosen to give 0 to the parameters vaddr, position fifo and status, whereas, for pid parameter, -1 has been chosen (since pid = 0 is, indeed, a valid process).

Then, a function dedicated to the **Insertion of a new entry** has been implemented. It takes, as parameters, the PID, the Virtual and Physical Address and the status, to be able to allocate every field of the entry structure, in the way we've built is. The field position fifo is missing, because, we just have to increment it by one element from the last position fifo there was already existing.

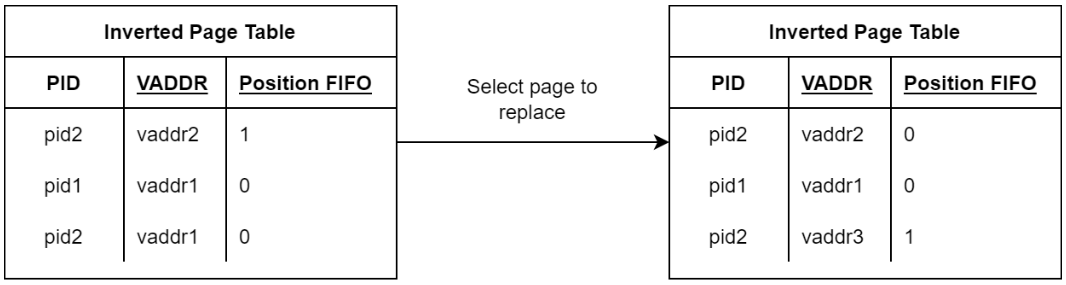
The following function is the one that has the goal of returning an entry of the IPT given the parameters *vaddr, paddr and pid*, and this function is get_paddr_entry.

We, finally, have the functions to clean up an entry given its index (this also implies that the position fifo parameter of other entries that have the same PID must be decremented), and cleaning up means to set all the parameters to 0, except for the pid field, set to -1. The other "destructive" function is the destroy one, which basically deallocates, through kfree, each entry of the IPT, and finally the entire IPT.

The most interesting function is the **Replacement** one. We decided to implement a *Local Page Table Replacement algorithm*, choosing the oldest page for a process with a PID equal to the one given as parameter. This is possible through a further property which defines the position in the FIFO of the local process.

The function works in the following way:

1. we scan every entry of the IPT, looking for an entry where the position fifo parameter is 0 and the pid parameter is the one given as parameter;
2. once we found that page, the index of that page is saved and returned;
3. This function will be largely used by the swapfile code which is in charge to reset that entry by decreasing the other FIFO positions and add the new entry (see the chapter 4)



Inverted Page Table

PID	VADDR	Position FIFO
pid2	vaddr2	1
pid1	vaddr1	0
pid2	vaddr1	0

Inverted Page Table

PID	VADDR	Position FIFO
pid2	vaddr2	0
pid1	vaddr1	0
pid2	vaddr3	1

Figure 2.1: Page replacement example

Chapter 3

TLB Management

carried out by Concetto Antonino Privitera

Another essential element that makes the operating system run smoother is the TLB management. Indeed, this helps a lot the process execution in terms of paging operations. Through this, we can get the most recent pages used in a faster way, but misses could be possible.

As for the other components, its statistics variables are computed here as well and used after by the statistics management once OS161 shuts down.

OS161 simulates MIPS R3000, so we have 64 entries in the TLB, each 64 bits, composed by different fields as shown in the figure 3.1.

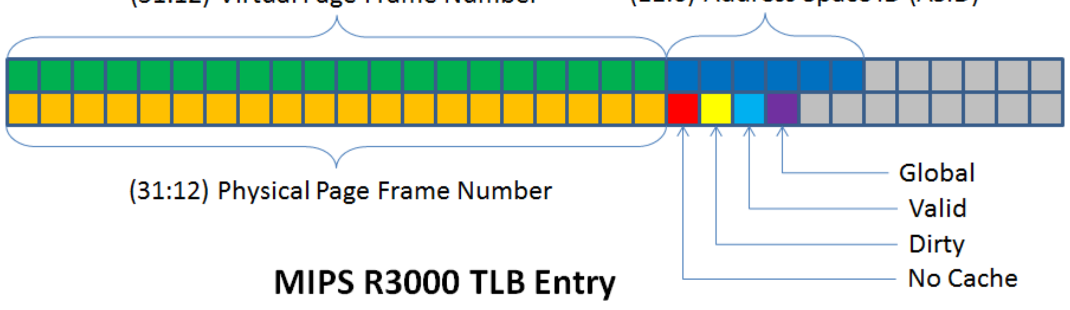

(31:12) Virtual Page Frame Number
(11:6) Address Space ID (ASID)
(31:12) Physical Page Frame Number
Global
Valid
Dirty
No Cache
MIPS R3000 TLB Entry

Figure 3.1: TLB structure [1]

As it is possible to notice, the virtual/physical page number to load has size *20* bits. In order to make it work and totally integrate it in the operating system, it was needed to create different functions that ranges from bootstrap to full reset of the tlb.

It's important to say not every field is used for this project, but only those defined in the following list:

- ASID
- Valid

The **ASID** field is used to store the PID of the process. Actually it is not needed since the TLB will only contain the current process entries. Instead, the **valid** field is used to check if the entry is still valid before loading it. So, there are few cases to check that they are explained afterwards.

The code in charge to manage the tlb is stored in the file *vm_tlb.c* with a different set of functions specified by the prototypes:

- void tlb_bootstrap(void)
- static int tlb_get_rr_victim(void)
- void write_entry(int index, uint32_t vaddr, uint32_t paddr)
- void add_entry(int *index_tlb, uint32_t vaddr, uint32_t paddr)
- int read_entry(uint32_t vaddr, uint32_t *paddr)
- void reset_one_entry_by_index(int index)
- void reset_tlb(void)
- void reset_tlb_pid_different(pid_t current_pid)

Different macro definitions are used like *NUM_TLB* and *PAGE_FRAME.* They allow to get the offset of an address by taking the page number out or just get the page number respectively. Furthermore, a bitmap is used to take trace of every entry stored into the TLB.

The code is made to work with the functions available *mips/tlb.h.* In particular the functions *tlb_write*, *tlb_read* and *tlb_probe* are strongly used to interact with the tlb.

The first function, *tlb_bootstrap*, has the goal to set the whole environment for the tlb management by setting up the bitmap. Of course, this is called once during the os161 boot.

If a vm fault happens, the system will look into the tlb at the begging by calling *read_entry* to check whether it exists. Indeed, this function is using a function called *tlb_probe* defined in *mips/tlb.h* which looks for a match with the virtual address and returns the *index.* If found, *tlb_read* is called with the aim to get the physical address. An *int* is return to know if *paddr* was correctly set.

The function *write_entry* is internally used by *add_entry* to write the virtual/physical number in a specific position of the TLB. These functions look for a free entry in the bitmap. In case it is not found, then *tlb_get_rr_victim* look for a victim to replace. Then, the entry is updated.

Finally, the functions *reset_one_entry_by_index*, *reset_tlb* and *void reset_tlb_pid_different* are in charge to reset one or more in entries in case there is a context switch respectively.

Chapter 4

On-Demand Page Loading

4.1 What is On-Demand Page Loading?

carried out by Ivana Orefice

On-Demand Page Loading is a virtual-memory management technique, thanks to which we can load one or more pages in memory but only when the page is actually requested. In its implementation, OS161 does not give support to On-Demand Page Loading: in fact, when OS/161 loads a new program into an address space using runprogram, it pre-allocates physical frames for all of the program's virtual pages, and it pre-loads all of the pages into physical memory. The goal of this assignment is to change this approach to the loading of pages, because bot physical frames and virtual pages must be loaded on demand, which means that the page should be loaded (and physical space should be allocated for it) the first time that the application tries to use (read or write) that page.

4.2 LoadELF Implementation

carried out by Sergio Petrone

This part of the assignment has started with a careful study of how the *loadelf.c* worked and what was its goal. The most important part of this file is the function called *load_ elf*, which is the one that accepts a vnode parameter, alongside the entrypoint, being able to load an ELF executable user program into the current address space, and returning the entry point (initial Program Counter) for the program in ENTRYPOINT. This function is crucial in order to perform the On-Demand Page Loading: in our approach, we didn't change the way the *loadelf.c* is implemented, but instead we preferred to change the way it is actually called. Another important step of the implementation related to the ELF file loading is represented by the function *read_ elf_page*, implemented in *addrspace.c*. This function accepts these following input parameters:

- *v_node*: it is the equivalent of a inode structure which describes the file to which it is referred in a Unix/Linux-like system. vnode does the same, but for a file in a OS161 system. By the end of the function;
- *destPhAdd*: this parameter represents the physical address to which the ELF executable will be loaded;
- *len*: this parameter represent how long is the file to be read;
- *offset*: indicates the "point" in memory from which the reading originates.

This function basically performs the reading of the file by means of two functions:

- *uio_kinit*: initializes a uio operation suitable for I/O from a kernel buffer. In this case, we understand that it is about a Reading operation because the last parameter passed to this function is the constant **UIO_READ**;
- *VOP_READ*: after having initialized the read operation, the function VOP_READ is the one that practically realizes the reading operation.

It accepts the vnode parameter and the structure ku, and has the effect to fill the vnode structure with all the information related to the file just read. It returns an integer value, called *result* in our code, which proves if the function has performed correctly or not.

Also, the result parameter will be returned by the *read_ elf_page* function to the caller. The way the *read_ elf_page* function is called and used is extensively described in Chapter 5.

Aside from the implementation of the part dedicated to the ELF file, the team has also implemented the Swap File mechanism, which is object of the following paragraph.

4.3 Swapfile Implementation

carried out by Concetto Antonino Privitera

In order to keep the on-demand mechanism possible for the whole process execution, it is needed to extend the limits of the algorithm itself by implementing the swapfile. Thanks to it, we can *swap out* any page whenever there in no memory available in the RAM and *swap in* a required page in case this one is available within the file.

Indeed, this code requires a file called **SWAPFILE** stored into *emu0* which is the container for our pages to evict from the RAM. As required, this file has to have a size of 9MB defined by the variable *FILESIZE*, whereas the number of entries is defined by *NUMBERENTRIES* which corresponds to the file size divided by the page size.

However, it is important to keep track of every entry inside the swapfile through a structure that takes into account the different properties needed to manage each one of it. These are:

- *permission_flag* used to know which the permissions related to that page
- *pid* of the process
- *vaddr* of the page itself
- *valid* to know if the entry it is actually valid

Different functions has been implemented in order to fully support the page replacement required by the page table. So, *swap_ bootstrap* is the first function called as OS161 is booting which is in charge to open the swapfile and set up every entry. During swapfile initialization, the manager is in charge create the file and reach the size required in case its size is lower than the one required. This is possible through the *uio_ kinit* and *VOP_ WRITE* functions and automatize the file creation.

Then, two main functions are the core of this mechanism: *swap_ in* and *swap_ out*.

The latter, *swap_ out*, requires the following parameters whenever this function is called:

- *pid* of the process
- *vaddr* of the page
- *permission_flag*
- *paddr*, which is the physical address where we want to load the page

This function checks the *vaddr* is not over *MIPS_ KSEG0*, so we make sure we are working in the correct area of the RAM. After that, it looks for a invalid entry to run the actual swap out. In case there is no entry available, the swap out is not possible. Otherwise, through the calls *uio_ kinit* and *VOP_ WRITE*, OS161 copies the page from the RAM to corresponding entry of the swapfile. Then, it sets the entry as valid alongside the given properties needed and reset the entry in the page table.

The other function called *swap_in* does the dual operation, moving a page from the swapfile to the RAM. In order to do this, it is required different parameters:

- *as*, which is the pointer to the address space of the process
- *pid* of the process
- *vaddr* of the page
- *paddr*, which is the physical address where we want to load the page

The function looks for the entry that has the corresponding *pid* and *vaddr* stored into the swapfile. In case it is not found, then an error is generated. If it found, then I make sure I haven't reached the maximum number pages per process because otherwise I have to run a page replacement and execute a swap out before going on with the swap in. Once every check is done, I clean the page by writing a zero region and then I move the entry from swapfile to the RAM through the calls *uio_kinit* and *VOP_READ*. Of course, the entry of the swapfile is set as invalid and the new page is added into the page table.

Finally, when OS161 is turning off, the *swap_destroy* is run to close the file and spinlock for correctness.

Chapter 5

Putting everything together: Address Space

carried out by all team members

Finally, the last important aspect to cover is the implementation of the address space located in *addrspace.c*. The goal of this file is to connect every single component made previous to each other in order to make os161 fully support virtual memory with IPT, TLB and swapfile.

As for the other components, here we have different functions to initialize and destroy the virtual memory, allocate/free pages, apart from all the functions necessary for the "pure" management of the address space, then functions like *as_ create*, *as_ copy*, *as_ destroy* and others.

1. *vm_ bootstrap*: this function has the goal of triggering the inizialization functions that we implemented in every single component (swapfile, coremap, page table and tlb), also initializing the values of statistics (see more in Chapter 6). It is called inside the *main.c* file during the bootstrap phase (method **boot**).

2. *read_ elf_ page*: this function has been already described in Chapter 4. Nevertheless, it is implemented here.

3. *vm_fault*: this is the core function of out entire implementation. It allow us implementing the on-demand page loading when a fault is triggered. Before examine the flow of execution, it is important to notice that the function accepts as parameters:

   (a) *faultaddress*, which is the virtual address in which the fault has happened.

   (b) *faulttype*: it indicates if we are manipulating a READONLY, READ or WRITE type of fault.

The flow of this important function is the following: when the function is triggered, the value of faulttype is evaluated to check whether the fault is a WRITE, READ or READONLY one. When it is a WRITE one, we count the fault that has happened by incrementing the correspondent statistical value. This done, we set the base and top position of Code, Data segments and Stack. Before starting with the On-Demand Page Loading mechanism, we should evaluate other two alternatives:

1. Look inside the page table, to see if there exists an entry in the page table that corresponds to the one we're looking for, given the PID of the current process and its virtual address, that is faultaddress. If that's the case, then there's no need to do On-Demand Page Loading.

2. Otherwise, if the previous case doesn't apply, we should evaluate the possibility of looking for the page by exploiting the swapfile situation. Indeed, we evaluate by calling the *swap_in* function, which should look for a page to swap in into the RAM. Again, f that's the case, no need to do On-Demand Page Loading.

If none of the above conditions apply, then the On-Demand Page Loading could begin. We evaluate the value of fault address, to see if it belongs to Code or Data segment or Stack. In any case, we should evaluate two cases:

1. the number of allocated pages is the maximum one
2. by calling *getppages* function, we get 0, which means can't get a free page

Both situations described above mean that there are no pages to be used, so we need to perform a page replacement. We get the index of the page that must be replaced and we perform the swap out of the page (by calling the function *swap_ out*) and we get the new address of that page exploiting the index:

```
paddr = index_page_to_replace*PAGE_SIZE;
```

This done, we should evaluate if the faultaddress is located at the begin of the first page, at the begin of the last page of in the middle, and we set the *size_ to_ read* parameter consequently. This is needed when calling the *read_ elf_ page* function. The found page must be added to the page table, by calling the *page_ table_ add_ entry* function.

Lastly, once we the page table is set, we should write inside the TLB, by performing the correspondent write call: in our project, we call the *add_ entry* function. The mechanism of this function is explained in Chapter 3.

Chapter 6

Statistics

carried out by all team members

Each file that implements a specific feature will update its internal statistics that they are finally used and printed thanks to the file *vm_stats.c*. It is important to notice that statistics must be printed only when the OS161 has been shut down by the user, i.e. when writing the **q** command on the terminal. That's why the act of printing statistics is implemented inside the **shutdown** method inside the *main.c* file.

The *vm_stats.c* file contains the initialization of the statistics, each of which corresponds to a variable whose name is self-explanatory of the statistic it represents. Each variable is incremented through a dedicated function, which also prints out the current value of the parameter. Moreover, the team has included the *vm_stats.h* file in order to use these functions in other files.

The functions defined in vm_stats in order to manage the statistics are the followings:

- *increment_tlb_faults*: this is called in *addrspace.c* in the function *vm_fault* in order to increment the number of TLB misses occurred (not including faults that cause a program crash).
- *increment_tlb_faults_free*: this is called in the function *add_entry* in

vm_ tlb.c in order to increment the number of TLB misses for which there was free space in the TLB to add the new TLB entry (so no replacement).

- *increment_ tlb_ faults_ replace*: in contrast to the one mentioned above, this function, also called in *add_ entry* of *vm_ tlb.c*, increments the number of TLB misses for which there was no free space for the new TLB entry (so replacement).
- *increment_ tlb_ invalidations*: called in *vm_ tlb.c* in the functions of tlb reset (*reset_ tlb* and *reset_ tlb_ pid_ different*), this function increments the number of times the entire TLB was invalidated .
- *increment_ tlb_ reloads*: this function is called in *addrspace.c* and increments the number of TLB misses for pages that were already in memory.
- *increment_page_faults_ zeroed*: this function counts the number of TLB misses that required a new page to be zero-filled.
- *increment_page_faults_ disk*: this increments the number of TLB misses that required a page to be loaded from disk.
- *increment_page_faults_ elf*: this function increments the number of page faults that require getting a page from the ELF file.
- *increment_page_faults_ swapin* and *increment_page_faults_ swapout*: those functions manage the increment of the number of page faults that require getting a page from the swap file.
- *increment_ swapfile_ writes*: this function counts the number of page faults that require writing a page to the swap file.

Bibliography

[1] *OS161 TLB Miss and Page Fault.* URL: http://jhshi.me/2012/04/27/os161-tlb-miss-and-page-fault/index.html#.YaPvG9DMJD8.