Politecnico di Torino

System and Device Programming

Project Report C1.2

Paolo Celada s283478

Pietro Macori s283421

Davide Perticone s281633

ACADEMIC YEAR 2020-2021

Contents

Overview

Objectives Completed:

- TLB Management
    - replacement (Round-Robin)
- Read-Only Text segments
- Pure On-Demand Page Loading
    - Hashed Inverted Page Table
- Page Replacement
    - Swap file (9 MB, modifiable)
    - Local Round-Robin victim selection
- Instrumentation
- Page allocator that keeps track of allocated/free frames with bitmap.
- System calls: read, write, exit, waitpid, getpid, fork (needed in order to use test programs, taken from course labs solutions)
- Locks and condition variables (taken from course labs solutions)

Options available for conditional compilation (in conf/conf.kern):

- paging: enables virtual memory with paging.
- list: enables usage of lists for free entries in the hash table and swapfile.
- waitpid: enables PIDs and syscalls related to PIDs.
- sync: enables locks and conditions variables.
- Fork: enables fork syscall

Kernel versions:

- PAGING: this version implements the paging virtual memory system.
- PAGING_LIST: this version implements the paging virtual memory system as well as the usage of lists in various parts of the implementation as per option "list".

Tests:

Test binaries used during development and to test the correct functioning of the part implemented:

- ctest

- huge
- matmult
- matmult modified (increased matrix dimension)
- tictac
- palin
- sort
- rdfaulter (custom version of faulter generating a VM_FAULT_READONLY)
- faulter
- zero
- parallelvm
- forktest
- forkbomb (with a lot of ram and more than 100 pids)

Kernel tests run:

- at
- at2
- bt
- tlt
- km1
- km2
- km3
- Km4

Project Implementation

Hashed Inverted Page Table (vm/pt.c)

A Hashed IPT is used as the underlying data structure of our VM system implementation. The IPT is a contiguous array of entries, one for each frame in physical memory.

Each entry is composed of two fields:

- PID
- Virtual Address

The indices of the IPT array are used to identify the physical frame each entry refers to.

The PID is a 6-bit number, while the Virtual Address is a 20-bit number. Both are stored in 32-bit integers (one for each).

It must be noticed that the Virtual Address is used to represent the page number (not the actual virtual address) and the PID is mandatory due to the possibility of multiple processes using the same pages in their own virtual space (with the problem of ambiguity when searching the IPT, if the PID is not used). Therefore, PID and Virtual Address combination represents a primary key, uniquely identifying a specific page.

Hash table for IPT (hash/st.c & hash/item.c)

In order to speed up the search of a virtual page (avoiding resorting to a linear search), a hash table with linear chaining is used. The structures stored in the hashtable are named STnode and item. STnode is a wrapper for items, allowing chaining them.

The struct item contains the following fields:

- Struct Key
- Int index

The struct Key, as the name says, stores the key of the entry (PID and virtual address). The index is the index of the IPT entry having PID and virtual address equal to the key stored in the struct.

As the key is composed of two fields, it is needed to combine them into a single one, in order to calculate the hash. Key must be unique and therefore the sum of PID and Virtual Address does not satisfy this constraint. Instead, the concatenation of them does. The PID is unique for every process and the concatenation of both of them creates a unique identifier. The concatenation is therefore used as the key of the hash table.

IPT victim selection

When an empty page is needed but all frames are occupied, a victim page must be swapped out. This operation is carried out inside *getppages()*, which contains all the logic for allocating an empty page. Before swapping out, it is necessary to select a victim. In this implementation, local replacement is used. This means that the victim must be a page belonging to the same process needing a new page. A local replacement policy has been implemented instead of a global replacement as the latter would increase the overall complexity. For example, page-locking would be needed in order to avoid that the victim's owner keeps using it while being swapped out.

A Round-Robin algorithm is used. Because of local replacement, each process can select a victim only from the pool of pages owned by him. This means that the victim selection policy must be followed independently by each process.
In order to do so, a new field, *last_victim*, is added to the proc structure. This variable has the job to remember the last victim selected so that when a further one is needed, it will be a page after it. The initialization value of last_victim is -1, as no pages have been yet swapped out.

Due to this local replacement policy, when many pages must be allocated to the kernel and there are none free, the system crashes. It is not possible for the kernel neither to swap out processes' pages or its own pages. This phenomenon is rare and may occur when the RAM size is too little and the kernel needs many pages after virtual memory bootstrap (for example, when forking many processes).

TLB (vm/vm_tlb.c)

The TLB management, in our implementation, changes significantly with respect to DUMBVM. The main changes impact the following aspects:

- TLB replacement
- TLB miss management

TLB replacement

In the DUMBVM version, the TLB did not apply any replacement policy. Once it filled up, the system crashed. In our version, a Round-Robin replacement policy is applied. This implies that the system does not crash once the TLB is full, but continues working smoothly.

When a TLB miss occurs and a new entry must be loaded, the system looks for a free entry. If it is found, it is loaded with the needed data. If it is not found, the replacement algorithm is applied, a victim is selected and replaced.

It must be noticed that with default settings (512K of ram), it is not possible to notice any TLB fault with replacement. This is because the kernel occupies at least 68 frames of the ram. Indeed, out of 128, only 60 frames are available to the processes. This implies that no more than 60 entries of the TLB are used at the same time. Therefore, the replacement of a TLB entry does not happen, as when a page is selected as a victim to be swapped out, the related entry is invalidated.

If the ram size is increased (such as 1024K), depending on the test program it is possible to notice TLB faults with replacement. At the bottom of this report, a table with test program statistics is available.

TLB miss management.

Whenever a TLB miss occurs, the event is managed by the function vm_fault.

The following steps are followed in it:

- It is determined whether the page is already in memory. To achieve this, the function *ipt_lookup* is used. It looks up in the IPT where an entry with the same page address of the faulting address is present (if yes, it is in memory).
- If it is not in memory, the following actions are performed:
  - If the page that must be loaded belongs to a code or data segment the routine is as follow:
    - A page is allocated calling the function as_prepare_load. It returns the physical address of a zero-filled frame. This frame can be taken among the free-frames or by swapping out a page of a process.
    - Before loading the needed page from the ELF file, it is necessary to look in the swap table or swap list (depending on the version of the implementation) to see if the page needed has been previously

        swapped out. In truth, this last operation can be successful only if the page belongs to a data segment (as code is never swapped out).
        - If the page is not found in the swap file, it is loaded from the EFL file by calling the function *load_page*.
        - The entry related to the requested page is inserted in the IPT (calling *ipt_add*).
        - An entry in the TLB is loaded (calling *update_tlb*). In the case of full TLB, a victim is selected using a Round-Robin policy. It is important to notice that the update of the TLB is carried out according to the segment the page belongs to; if it is a text page, it will be set as read-only, in the other case, as read-write (data page). Setting text pages as read-only prevents processes from modifying their own code.
    - If the page that must be loaded belongs to a stack segment, it means that no loading from the ELF file is needed. It is only possible either to load it from the swap file (if previously swapped out) or to allocate a new empty page. In particular, the following steps are performed:
        - Allocate a new page (calling *as_prepare load*).
        - Try to swap in the page. The page is swapped in only if it has been previously swapped out.
        - Update both IPT and TLB calling the same functions used for data or code pages.

Physical Memory Management (vm/coremap.c)

In order to create a custom virtual memory management, the team had also to implement the management of frames to allow dynamic allocation and deallocation of memory at run time. This has been implemented in a similar way to the one proposed by the professor during the course. The track of free RAM frames is done using *freeRamFrames* that is a bitmap (as suggested in lab 2) and *allocSize* which is an array of longs and indicates the numbers of contiguous allocated pages starting from a given entry. The values of *freeRamFrames* are the following:

- 0 → frame has not been freed and contains info
- 1 → frame has been freed and can be reallocated

In order to manage the bitmap, three macros have been defined: SetBit, ClearBit, and TestBit which respectively set the value of a specified bit to 1, 0, or read its value. These are used in the functions shown below.

The main functions used in coremap.c and their functionality are briefly explained below:

init_freeRamFrames() and *init_allocSize()* which are used from *vm_bootstrap()* to respectively initialize the *freeRamFrame* and *allocSize*. They both allocate kernel's memory using kmalloc and then fill the related array and bitmap with zeros. These two functions replace the original vm_bootstrap() located in the dumbvm.c file.

getfreeppages() which has not been changed with respect to the implementation seen during the course. It simply iterates over *freeRamFrame* looking for the first set of free frames that can accommodate the number of pages provided as parameter. It's important to notice that since we are implementing on-demand page loading the number of needed will always be one. This implies that the algorithm will search for the first free frame. In case an entry is found the corresponding entry in the two arrays is modified.

getppages() is a wrapper that tries different ways to get physical pages. At first, it tries to get a freed page calling *getfreeppages()*, if no previously freed frame is available then *getppages()* ask the kernel to allocate new physical memory calling *ram_stealmem()*. In case no free frame can be found using these two functions, the swap-out process is started and it will be discussed in the next section. Must be noted that before returning the physical address to the calling function, the frame is zeroed in order to remove the content of old processes.

Note: In our implementation *ram_stealmem()* is used only during *vm_bootstrap()* because the team decided to force the use of free frames. In fact, during *vm_bootstrap()* we allocate all available frames (computed as the total number of frames in memory minus the frames allocated by the kernel). Then, all these frames are immediately freed using *free_kpages()*. In this way, all the memory is deallocated and immediately visible by freeRamFrame which will work on all the frames of the RAM.

freeppages() is the function used to free the frames. It receives as input the physical address of the frame and the entry in the *freeRamFrame* array. It simply sets to '1' all the frame entries that were previously allocated. This is done by doing a lookup in *allocSize*.

On-Demand Page Loading (vm/segments.c)

In the original version of OS161, when a user program is launched, the whole content of the ELF file is loaded into memory. This procedure is often wasteful and unnecessary, as the entire program is never used altogether, causing a reduction in the degree of multiprogramming. In this project, a basic form of pure on-demand page loading has been implemented.

The procedure starts when a TLB fault is caught. The kernel checks if the faulty page is present in the IPT in order to verify its presence in memory. In case this is not verified, its presence is evaluated in the swap file (in case the page contains data). If the page contains code or if the page is not present in the swap file, it means that the faulty page has not been loaded previously, and this implies the loading of the page from the ELF file.

From *vm_fault()* the function *load_page()* is called. The goal of the function is to load a single page from the ELF file into the memory. As preliminary steps, the function has to correctly understand where and how much to read from the ELF since now it's not possible to simply load the content from the disk. This is not a trivial task as the content of the ELF file is not an exact copy of what will be loaded into the memory but instead, it contains a "compressed" version that must be correctly read to avoid errors during the execution of the program. This is done in *load_page()* by reading the address of the faulty address, the address of the page containing the faulty address, the offsets of the segment on which the page belongs, and the offset of the faulty page from the beginning of its segment.

In order to go more in detail, it's important to clearly understand how the ELF file stores information. In fact, it doesn't have the concept of page, frame, or alignment but it just stores the information and its position on the virtual memory. The process of loading the page is started by reading the segment information from the header of the ELF file. This provides the starting address of the segment, its length on the file, and its length on memory. Then a computation is made to understand the "padding" at the beginning of the segment. As stated before the ELF doesn't have the concept of a page and the information of a segment can start at any position of the file and this means that reading an entire page from the file leads to possibly reading also other segment information.

Let's go through an example to make it more clear.

The following table represents the content on the ELF file:

00000	00111	11111	11112	22222	22222

Now, let’s see what should be loaded on memory.

00000	00	11111	11111	11	22222	22222

Each cell of the table represents the content of a page while the numbers indicate the segment. Imagine that the first page of segment “1” has to be read from the ELF. If the second page is read, the content on memory won’t be the desired one but instead, the content of two different segments is loaded and this implies an error in the execution of the program. To avoid these problems some computation must be made to correctly load the needed pages.

First of all, the “padding” before the beginning of the segment is computed (e.g. counts the two ‘0’ on the second page of the example). Then three different cases are possible:

- The first page of the segment needs to be loaded
- The last page of the segment needs to be loaded
- A page in the middle of the segment needs to be loaded

For each of the three possibilities, the offset from the beginning of the segment is computed as well as the number of bytes to read from the ELF. After having set the correct parameters, the function *load_segment()* is called. Is important to notice that the dimension of the parameter memsize is always equal to PAGE_SIZE since the on-demand page loading is based on the concept of loading one page at the time.

In the beginning, the team tried to copy and paste the content of the ELF directly on the memory until we realized that the test *testbin/sort* was causing panic errors from the kernel. Strangely all other tests were passing even with our buggy solution and only launching *testbin/sort* we were able to fix the error and finally fully understand how the ELF file stores the informations.

The *load_segment()* function is the one that actually performs the read from the file. It sets the correct values to the fields of the uio structure and executes the *VOP_READ()* on the v-node related to the ELF file. In order to set the correct address in the iovec field iov_ubase, we convert the physical address of the buffer to a kernel virtual address using the function *PADDR_TO_KVADDR().* The function adds the KSEG0 value to the address. More*over,* we set the field *uio_segflg* to UIO_SYSSPACE and in this way, the read function thinks to be working on a kernel address and so the value KSEG0 is removed and the real address is obtained. Finally, it must be noticed that if the memsize > filesize then the remaining space should be zero-filled. This is not done explicitly, because our virtual memory system provides pages that don't contain old data or code but these are already initialized and zeroed.

The original file *syscall/loadelf.c* has been modified in our solution. Mainly, the *load_segment()* function has been moved to *vm/segments.c* and the *load_elf()* function modified to best fit our solution. In fact, it doesn't need to execute two loops, one for reading the ELF header and defining the region and a second one to immediately load the segments. The load is done when needed and the second loop can be deleted, keeping only the first one to define the regions of the virtual address space reading the corresponding information from the ELF header.

Virtual Memory Manager (vm/addrspace.c)

Another important step in our project implementation was to replace the current virtual memory manager (implemented in *dumbvm.c*) that only performs contiguous allocation of physical memory, without ever releasing it, with a more sophisticated solution. It's better to note that some of these functions work together with others implemented in *coremap.c*.

First of all, the central function for our VM initialization is *vm_bootstrap()*, called after *ram_bootstrap()* during boot. It calculates the total number of available frames in memory (which depends on the RAM size chosen), initializes the IPT, the instrumentation, and the 2 data structures used in *coremap.c* previously explained. Then it allocates and deallocates the entire portion of memory not taken by the kernel. The reason we decided to implement this "strange" form of initialization of ram frames is explained previously in the physical memory management section (see note).

Allocation and deallocation of kernel pages is done respectively by *alloc_kpages()* and *free_kpages()*. Both of them call the related functions in *coremap.c* and then set the IPT entry with the correct index (paddr + i * PAGE_SIZE) to -2 if the entry has been assigned to a kernel page, and to -1 if it has been freed.

Address space for a user process instead is dealt with multiple small functions called when running a user program. The main difference with the *dumbvm.c* implementation is found in the definition of the addrspace data struct in *addrspace.h*. Differently from dumbvm implementation, with the contiguous allocation of ram frames, our pages are scattered throughout the whole RAM as single independent pages, and the access is strictly dependent on the translation from virtual to physical addresses. Because of this reasoning, a direct physical address for both segments is useless. The only functions worth noticing are *as_activate()*, which interact directly with the TLB for a given process, and *as_copy()*, which copies the address space of an old process into a new one generated by the fork syscall, together with data and stack pages in the IPT and in the SWAPFILE.

Page replacement (vm/swapfile.c)

Until this moment, our VM management system didn't allow running programs or applications that would require more pages than the available number of physical frames. This problem has been addressed by implementing a simple page replacement algorithm working together with a swap file as destination and source of swapped pages. The main idea is to have a file called SWAPFILE of fixed size (initially 9MB but it could be changed) and interact with it in two distinct moments in program execution:

- When a page needs to be loaded into memory and there are no free/freed frames available, a page from memory is swapped out (written) into the swapfile if it's either in data or stack segment (pages in code segment does not need to be swapped out as they are *read-only*)
- When a page needs to be loaded from memory, after checking its absence in memory but before loading it directly from ELF, the swap file is queried and the page is returned if it has been swapped out before, performing a swap in operation (read)

The whole implementation of the interaction with the swap file is contained in the *swapfile.c* file (with the support of *swapfile.h* for constants and prototype declarations).

First of all, in both cases, the swap file is initialized in *init_swapfile()* performing a *vfs_open()* on the SWAPFILE and initializing an array of support to all -1 (all available). The support array, called *swap_table*, has 1 entry per possible page in the swap file (and so a length calculated as the total size of the swap file divided by page size), and allows us to have direct access to pages in the swap file by performing a seek operation using the index of the page taken from the support array.
Each entry of the *swap_table* array is composed of 2 fields:

- PID
- Page number

The swap-in operation is then implemented through the function *swap_in()*, called from *vm_fault()* in case of an IPT miss on a data or stack segment page, by passing the fault address as a parameter. The function looks for the corresponding entry in the *swap_table* by using the fault address together with the PID retrieved from the current process. If the entry is found, first it's invalidated by inserting a -1 value into the PID field, and then it's actually read from the SWAPFILE by calling *file_read_paddr()*. This function performs a regular *VOP_READ()* operation of the file, at our specified offset (calculated as entry index * PAGE_SIZE) with as destination the physical address passed. If the operation is successful, the number of bytes read is returned.

Similarly for the swap-out operation, which finds the first available entry in the *swap_table* in mutual exclusion and stores in it the victim's PID and page number, and then performs a *VOP_READ()* operation using the v-node of SWAPFILE. By implementing this function, we came across a problem related to the virtual to physical address translation of the victim's page: in short, the selected page can be owned by a different process with respect to the calling one, leading to a wrong translation generated by the wrong address space. A simple and direct solution is to use the physical address of the victim and map it to kernel space by adding the constant value KSEG0, and set the uio segment flag field to UIO_SYSSPACE. In this way, the translation is done as if it was a kernel address (subtracting KSEG0).
The functions used to perform it are *swap_out()* and *file_write_paddr()*. The first is called in *getppages()* as a last resort when any attempt of finding a free frame has failed, and a victim following a round-robin policy is selected.

Another important job is performed by the *duplicate_swap_pages()* function, which was implemented in order to correctly duplicate pages of the SWAPFILE. It was added late in development to make the swap process correctly deal with the fork system call. It takes an old and a new PID (of parent and child processes), it searches linearly for all entries owned by the old process and it copies them into a new free entry into the swap file (by performing in sequence *file_read_paddr()* and *file_write_paddr()*), all in mutual exclusion.

Lastly, *free_swap_table()* allows the cancellation of the current process entries when terminating a process with the exit system call (set PID values to -1) and *print_swap()* print on output the entire table content.

Optionally in the SWAPFILE management implementation, enabling the OPT_LIST option, it's possible to use instead of a simple array (the *swap_table*), 2 linked lists in order to keep track of free and occupied slots in the SWAPFILE, with the only addition of a file offset per entry, as it's not possible to use an index anymore.

Instrumentation (vm/instrumentation.c)

As the last step in our project implementation, in order to check the correctness of our solution, several statistics related to the performance of our virtual memory subsystem are calculated and shown to the user. These statistics include:

- TLB Faults
- TLB Faults with Free
- TLB Faults with Replace
- TLB Invalidations
- TLB Reloads
- Page Faults (zero-filled)
- Page Faults (disk)
- Page Faults from ELF
- Page Faults from Swapfile
- Swapfile Writes

Here are statistics taken by running some predefined user programs built with the intent to stress the VM management system on our PAGING version with 2048K RAM (found in *userland/testbin*):

	Ctest	Huge	Matmult	Palin	Sort	Parallelvm
TLB Faults	125502	6781	3754	12	19	9926
TLB Faults with Free	582	6398	3424	12	19	9926
TLB Faults with Replace	124920	383	330	0	0	0
TLB Invalidations	527	6430	773	18	22	2853
TLB Reloads	125242	3613	3371	7	12	9513
Page Faults (zero filled)	257	512	380	1	3	337
Page Faults (disk)	3	2656	3	4	4	76
Page Faults from	3	16	3	4	4	76

ELF						
Page Faults from Swapfile	0	2640	0	0	0	0
Swapfile Writes	0	2714	0	0	0	0

Workload division

From the organizational point of view, working on this project as a group of three was not easy. Even though the project is not a small one, there is a high level of coupling among all the components. This means that the workload cannot be split and carried out independently by each member. Keeping into account that the Professor advised not to work exclusively on a shared screen, we worked in the following way.

Before developing each component, the team met in order to decide how to proceed. After that, the various pieces to develop were split among the members, trying to always assign to each one components that had something to do with the others he developed. This means that each member concentrated on only 2-3 files, instead of on all of them.

Obviously, after writing the code, the team met and reviewed other member's solutions, in order to understand thoroughly how the code works and be able to debug it (a lot). This translates into all the members knowing how the system works in all its parts. Of course, the member that developed a specific piece has a deeper knowledge of it, but the others can with no problem understand the code and its functionality.

Debug sessions were divided into two: a theoretical-oriented one and a practical one. When something did not work as expected, first, the general idea of how it should work was thoroughly reviewed. This prevented the team from wasting time in debugging something that did not work because of the idea behind it. This part is the theoretical one. After the idea was double-checked, a practical debugging session was performed (with gbd and emacs). This was done sometimes in parallel and sometimes on a shared desktop, depending on the specific situation and bug.

Sharing the code throughout the entire development has been achieved through a git repository, in particular Github.

Conclusions

The development of this project has been really useful for a deeper study of what has been done during the course. During the three weeks in which we developed the project, each one of us has improved his knowledge regarding memory management, page table implementation, TLB behavior, and all other concepts we have dealt with during the development.

We had to face many theoretical and practical issues and problems generated by our own mistakes, such as the one we made implementing the loading of pages from the ELF file. These problems pushed us to study more deeply topics that have not been covered deeply during the course.

Even if our solution is still a very basic and rudimental one, we created a “smart” virtual memory, which can be compared to real solutions used in commercial operative system kernels. This really made us think about the complexity of a real O.S. and the effort needed to create a solution usable by non-expert users.