Sabanci University

Operating Systems

CS 307

Programming Assignment - 4: Extending a Virtual Memory Implementation with Paging

Release Date: 6 December 2024
Deadline: 16 December 2024 23.55

1 Introduction

In Programming Assignment - 4 (PA4) you are expected to extend an existing simple virtual memory by adding an address translation mechanism for paging and implementing some system calls for handling multiple processes. The existing implementation by Andrei Ciobanu can be found here.

Before proceeding any further, please read the detailed description of the VM in this blog post. It is impossible to complete PA4, before understanding the description of the registers, instructions and how a program executes as it is explained there.

The implementation assumes that the CPU is capable of executing instructions from a simple instruction set called LC-3. For your extension in PA4, you will not modify any of these instructions. Hence, you do not need to understand their implementations (method bodies). However, you have to understand what these instructions do and how they use the registers.

The first thing you should observe about the existing VM is that both the Virtual Address Space (VAS) of the running process and the physical memory are the same thing. You can see that from memory read (`mr`) and memory write (`mw`) method bodies. In reality, virtual addresses of processes are mapped to different places in the physical memory and address translation must be performed inside these methods.

Since the current implementation considers VAS and physical memory as the same objects, physical memory keeps a single address space. Therefore, system calls like `yield` which enables a context switch and `brk` that modifies the size of the heap segment are not required and not implemented.

In PA4, you will implement paging in the VAS so that the physical memory can keep multiple address spaces. Moreover, you will add `yield` and `brk` system calls to the instruction set so that processes can dynamically change their physical segment size and context switches can take place under the control of the Operating System (OS).

Your implementation will build upon the existing VM implementation inside `"vm.c"` file. Please do not use the original file in the repository linked above. Instead, use the modified version provided in the PA4 bundle that introduces some new definitions, registers and trap types. All the code you have to write must modify only this file.

In the next section, we describe the new layout of the VAS and the physical memory for the paging implementation.

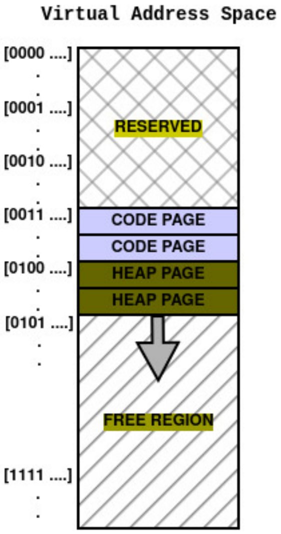

Figure 1: A sample VAS representation.

2 Virtual Address Space

In the original VM implementation, VAS consists of 16-bit addresses. VM is not byte addressed. Its elements are of type `uint16_t` (unsigned 16-bit integers) and they occupy 2 bytes. Hence, the total VAS size is 128KB.

The implementation assumes that the Program Counter (PC) of the currently running process starts from the address `0x3000`. The size of the code segments will be fixed as 8KB. In your paging implementation, every page will be 4KB in size. As such, the code segments of each process will be 2 pages. These restrictions are summarized by saying that code segment addresses start with `0011` in binary notation. In Figure 1, the code segment is represented as the two chunks that come right after the RESERVED section.

The second and the last segment of the VAS is the heap. Heap addresses begin with 01. Initial heap size is also 8KB(2 pages). However, heap size is dynamic. During the execution of a process, it might grow or shrink with `brk` system call, in which case the OS gives the process an unused page. The code pages always precede the heap pages. In Figure 1, the third and fourth chunks represent the heap segment in the VAS with the initial default configuration.

3 Physical Memory

Physical memory addresses are also 16-bit and `uint16_t` addressed. Even though the physical memory have the same size, we can fit multiple address spaces inside the physical memory since we only map occupied pages to it. Unallocated pages are not represented in it.

We also assume a structure on the physical memory. It is represented in Figure 3. The first two frames (8KB) (addresses starting with `0000`) is reserved for the OS. The OS keeps auxiliary data and metadata about the user processes in this region. These consist of three `uint16_t`'s and a process list. Process list consists of a sequence of Process Control Blocks (PCBs). For each user process, a PCB is maintained for its metadata. The third page frame (4KB) is reserved for the Page Tables.

Three `uint16_t`'s are *curProcID*, *procCount*, *OSStatus*. They keep track of the Process ID (PID) of the currently running process, total number of processes including the terminated ones and some flags about the OS, respectively. Additionally, two `uint16_t`'s represents a 32-bit page bitmap that shows the currently free page frames (0 for used, 1 for free)

PIDs start from 0. The first created user process gets PID 0 and PIDs are incremented with every newly created process. Hence, *procCount* is always 1 more than the largest PID. *OSStatus* is only used to check if the all space reserved for the OS are used or not. If the least significant bit of *OSStatus* is 0, then a new process can be created. Otherwise, either there is no space for a new PCB or a new page table, so the new process cannot be created and the VM program terminates.

PCBs begin from the physical address 12 and each PCB consists of 3 `uint16_ts`. We have a padding between the *OSStatus* and the first PCB. Inside a PCB, *PID_PCB* field keeps the unique PID of this process. *PC_PCB* shows the PC location in its VAS when this process was switched out the last time. *PTBR_PCB* which defines the pointer to the page table entry that belongs to that process. Please note that all fields have the same type: `uint16_t`.

A new PCB is always allocated immediately after the last PCB. Moreover, when a process terminates, its PCB is not removed and PCBs are never replaced. Hence, PCB address of a process can be easily calculated given its PID. When a process terminates, all the fields of its PCB except *PID_PCB* remains as it was just before its termination. *PID_PCB* is set to `0xffff` to differentiate it from running processes.

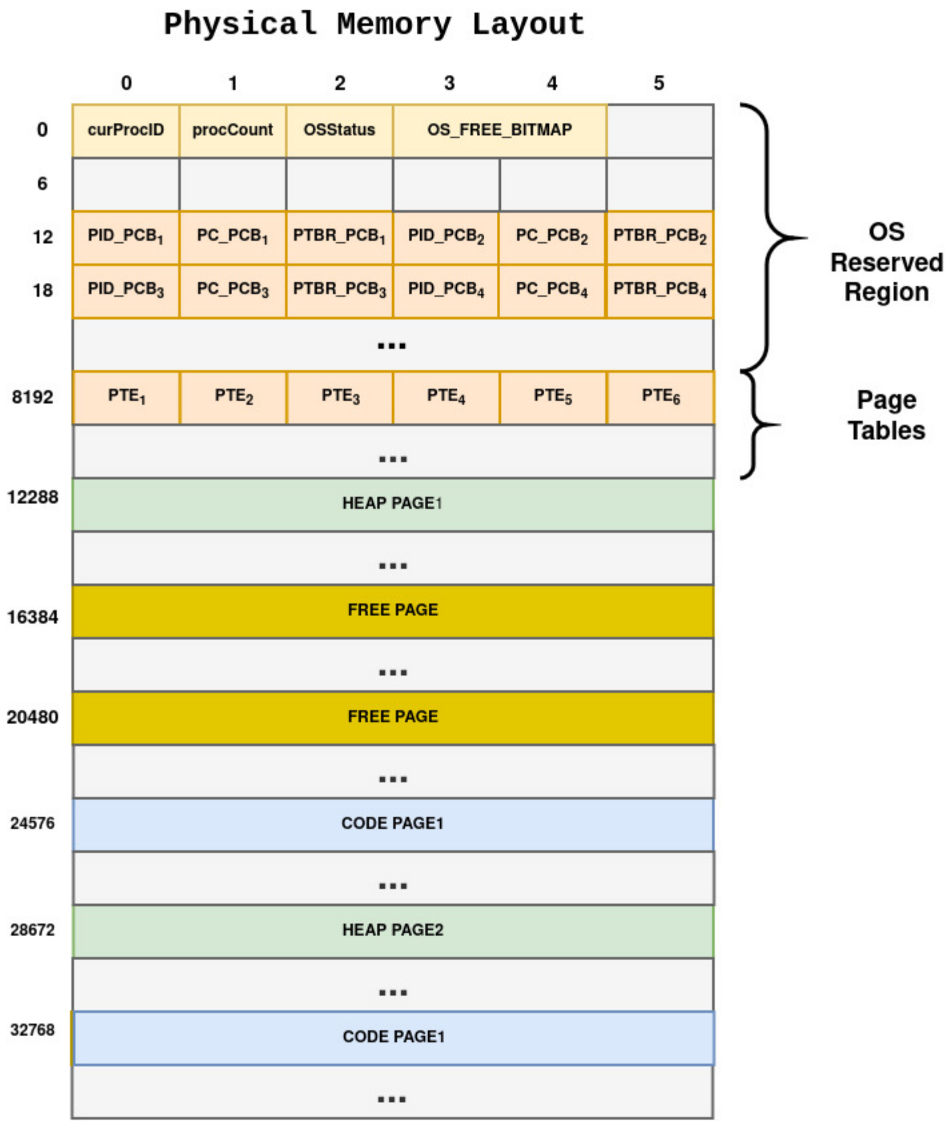

Figure 2: A snapshot of the physical memory with the OS, and one user process. Actual layout that your program will generate might be different. Here, pages are placed arbitrarily to demonstrate how VAS representation may not reflect the actual memory layout

3.1 Free Space and Page Management

Physical memory addresses after 3 pages(12KB) are reserved for the heap and code pages of processes. Since heaps might have different number of pages, the user process region of the memory might contain free and allocated chunks of various sizes.

In order to allocate and free space from this region, free page information is kept inside a bitmap inside the OS region.

As stated above, the third page frame is allocated for page tables. Each process must have a separate page table inside this region and each page table comes after the previously created process' table. We assume a linear structure to these page tables where each `VPN` directly corresponds to index of the page table with the same value.

Note that since the VAS is 128KB with 4KB pages there are 32 pages in each VAS. Therefore, five bits are enough to identify each page, so first 5-bit of a given virtual address is the `VPN` and the remaining 11-bit is the offset.

We also assume a structure on `PTE`s. Just like the VAS the physical memory is also 128KB, so there are 32 page frames in total. The first 5-bit of a `PTE` is used to indicate `PFN`. The last three bits are for write access, read access, and valid bit. Between the `PFN` and last bits there is an 8-bit padding, so a `PTE` is 2 bytes or a `uint16_t`.

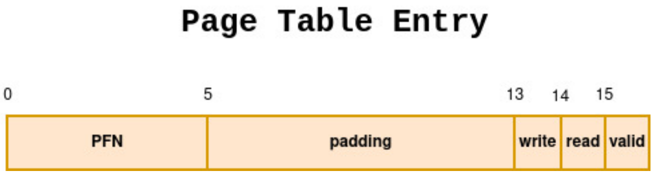

Figure 3: An example PTE

4 New Trap Types

In the original VM implementation, behavior of the trap instruction is described here. Type of the trap instruction is determined by the value inside the trap vector field. This field consists of 8 bits. Hence, there can be at most 64 different trap types. The original implementation uses 8 of them using the indices between `0x20` and `0x27`. We extend this range by adding two more trap types: `yield` and `brk`.

For the `yield` system call, trap vector index `0x28` is reserved. and it is used for voluntarily yielding the control of the CPU and allowing a context switch to happen.

When this trap instruction executes, it stores the current value of the `PC` and `PTBR` to the corresponding PCB. Then, starting from the current PID, we increment the PID until we encounter a PID that has not terminated yet. Whether a process has terminated or not can be determined by checking the *PID_PCB* field of a PCB. When a runnable process is found, its PID is set as the *CurProcID*, and its PC, and PTBR values are restored. If there is only one process running, `yield` keeps running the same process.

On the other hand, `brk` system call can be called by using the trap vector index `0x29`. When it is executed, The OS either allocates a new free page or frees one allocated by the process, changes the page table entry and marks it as used in the free page bitmap (0 for used, 1 for free). The page to be allocated or freed is determined by the value stored in register `R0`.

5 C Implementation Details

In your submission, you have to modify some of the existing methods and introduce new methods. All of them are either instructions or system calls that are supposed to be executed in the kernel mode. You should assume that only OS can execute in the kernel mode and it can directly access to the physical memory addresses without performing an address translation.

5.1 Modifications in the `"vm.c"` File

We modify the original `"vm.c"` version for the purposes of PA4. In the original implementation, there are 10 registers. We extend it by adding *PTBR* register, representing the page table base register of the currently running process.

Moreover, we extend the `trp_ex` array with `tyld` and `tbrk` trap instructions.

In addition, we modified the `ld_img` method that loads the binary program from a file to the given memory addresses. Since allocation of the code or the heap segments can be done to non-continuous pages, `ld_img` takes an array of two `uint16_t` addresses representing the two pages that are allocated.

5.2 Parts to be Modified

Memory access methods `mr` and `mw`: Given a 16-bit address, first five bits can be used for determining the `VPN`. If the page belongs to the reserved region, then `"Segmentation fault."` must be prompted. Otherwise using the `VPN` and the `PTBR` you should get the corresponding `PTE` and check the *valid* bit. If the page is not in reserved space but not valid, that is, if it is not allocated, `"Segmentation fault inside free space."` should be prompted. In both cases segmentation fault must exit the whole simulation. If the page is valid, the *protection* bits must be checked for the corresponding access type. If the process is trying to write to a read-only page, then the message `"Cannot write to a read-only page."` must be prompted.

If the address is valid compute the correct physical memory address using the `PFN` extracted from the `PTE` and the offset coming from the given virtual address (last 11-bit). Again note that this address belongs to `uint16_t` because memory is not byte addressed but it is word addressed.

Existing trap type `thalt`: In the original VM implementation, the program quits the fetch-decode-execute cycle when the single running process executes the halt trap instruction. In the extended version, there are multiple processes. Hence, the fetch-decode-execute cycle stops when all the processes execute the halt trap.

If there are other runnable processes, this method finds the next process that can run and assigns its PCB values to the CPU registers. The next process is picked by the method explained in Section 4 for the `yield` system call.

Before transferring the control of the CPU to the next process, all the pages allocated for the process including the initial code and heap segments must be freed. To free the pages, you can use the `freeMem` method explained below. To free the PCB, you must only set its PCB's `PID_PCB` field to `0xffff` and do not touch the other fields to leave a footprint in the memory.

5.3 New Methods to be Added

The `initOS` Method: Before doing anything, this method is called to initialize the OS-related parts of the physical memory. Basically, it sets *curProcID* to `0xffff`, *procCount* to 0, and *OSSttatus* to `0x0000`. Also, the bitmap for keeping the free pages must be initialized by this method, as well as marking the pages for the OS section as allocated.

The `allocMem` Method: It takes four parameters, `uint16_t` ptbr, `uint16_t` vpn, `uint16_t` read, and `uint16_t` write to allocate exactly one page frame. The method must find the first free page frame by searching the bitmap and allocate it for the given vpn. This search must be performed linearly starting from the lowest `PFN`. Therefore, the allocation is deterministic.

After a free page frame is found, a `PTE` for given vpn must be filled accordingly in the page table pointed by the ptbr parameter. The protection is determined by the read and write parameters, that is, if one parameter is given as `0xffff` (you can use `UINT16_MAX` macro), then that access is allowed. If the parameter is given as 0, then it is not allowed.

If there is no free page frame or a page frame is already allocated for the given vpn, then this method returns 0.

The `freeMem` Method: It is the dual of `allocMem` method. It takes the vpn of an allocated page and the corresponding ptbr. If the given page is not allocated, this method returns 0.

Otherwise, the allocated page frame must be freed by marking it as free in the bitmap and clearing the *valid* bit in the `PTE`.

This method is expected to only modify the bitmap and valid bit in the `PTE` and not to touch the content of the memory. If the freed page belongs to a heap segment and contains some objects or if it belongs to a code segment and contains some instructions, they should not be erased. We will have a look at the footprint of these chunks while evaluating your implementation.

The `loadProc` Method: This method takes a PID as input and loads its PC, page table base pointer values from its PCB to CPU's corresponding registers. Moreover, it sets *curProcID* to its PID.

The `createProc` method: This method takes two input strings: object file names for the code and the heap segments in this order. It returns an integer. If the process creation operation fails it returns 0. Otherwise, it returns 1.

There are three cases in which process creation fails. The first one is if the OS region of the memory is full and we cannot allocate a new PCB for the new process. This can be detected by manipulating the *OSStatus* field. In this case, this method prints: `"The OS memory region is full.  Cannot create a new PCB."` and returns.

The second case is when there is not enough free page frames for allocating the code segment. In this case it prints `"Cannot create code segment."` and returns. Lastly, if there is not enough free page frames for allocating the heap segment, it prints `"Cannot create heap segment."` and returns.

If none of this fail cases happen, this procedure assigns a unique PID to the new process which is one more than the biggest PID so far, and increments *procCount* by one. Then, based on its PID, a PCB is reserved for the new process. The method fills the `PID_PCB` value and assigns the default value `0x3000` to the `PC_PCB` field.

Then, it tries to allocate 8KB (2 pages) for the code segment, calling the `allocMem` method. If it is successful, it initializes the code segment by reading the file provided by the first input parameter. You can use `ld_img` method for this purpose.

Following the same steps, it also initializes the heap segment. Code and heap segments must be initialized in this order.

Note that if process creation fails for any reason while some page frames are allocated, they must be freed before the method returns.

The `tyld` Trap Instruction: The behavior of this instruction is described in Section 4. Whenever the `tyld` instruction is called, you should print a message as shown in the samples like: `"We are switching from process <oldProcID> to <curProcID>."`

The `tbrk` Trap Instruction: This instruction is also briefly described in Section 4. It allocates or frees pages of the currently running process according to the value stored in the register $R0$. Whenever the tbrk instruction is called, you should print a message as shown in the samples like: `"Heap increase requested by process <curProcID>.`" or `"Heap decrease requested by process <curProcID>.`" depending on the request type.

The register $R0$ is set by the process, `tbrk` instruction's job is to read and act accordingly. The first 5-bit of the $R0$ register is set to the `VPN` of the page that is being allocated or freed. The last 3-bit is set for write and read accesses, and whether this is an allocation of freeing request, respectively. For instance, the value 0101 1000 0000 0011 indicates a request for allocating a read-only page frame for the page with `VPN` 11 and the value 0111 0000 0000 0000 indicates a request to free the page with `VPN` 14. For freeing requests, values of read and write bits do not matter.

If the request is for allocation, then there are three cases. In the first one, the requested page is already allocated. In this case there is no need to take any action, thus the program must prompt `"Cannot allocate memory for page <VPN> of pid <CurProcId> since it is already allocated."` and return.

The second case is that there are no free page frames, then your program must prompt an error message `"Cannot allocate more space for pid <curProcID> since there is no free page frames."` and return.

If the first two cases do not happen, the program must allocate memory and return without any problem.

If the request is for freeing, then similarly if the page is already freed/not allocated, then the message `"Cannot free memory of page <VPN> of pid <CurProcID> since it is not allocated."` must be prompted. If not then the method must simply return after freeing the memory.

Note that during any of these operations you should not touch the content inside the page frames.

6 Submission Guidelines

For PA4, you are expected to submit two files.

- **vm.c**: A `vm.c` file is already provided to you inside the PA4 bundle. Your task is to implement or modify some methods. You can write additional helper functions if you need to. However, nothing you wrote outside of the region between `// YOUR CODE STARTS HERE` and `// YOUR CODE ENDS HERE` comments will be evaluated. Do mot change anything outside of it including the comments themselves.

- **report.pdf**: In your report, you must give a brief explanation of your implementation and explain any helper functions you wrote if any.

For the submission of PA4, you will see two different sections on SUCourse. For this assignment you are expected to submit your files **separately**. You should **NOT** zip any of your files. Please submit your **report** to “PA4 – REPORT Submission” and your **code** to “PA4 – CODE Submission”. The files that you submit should **NOT** contain your name or your id. SUCourse will **not** accept if your files are in a different format, so please be careful. If your submission does not fit the format specified above, your grade may be penalized up to 10 points. You are expected to complete your submissions until 16 December 2024, 23.55.

7 Grading

For this assignment, your work will be evaluated in a progressive manner. Each step is a precondition for the next step. If you do not get full point from a step, the next step will not be evaluated. After completing each step, please run the tests related to this step provided in the PA4 bundle. You can compile them all by running the ”make” command inside the root directory of the assignment directory. See Section 8 for details.

- **Compilation (10 pts):** Inserting your `vm.c` file to the PA4 bundle, make command should be able to generate all the test cases without any errors.
- **initOS (5 pts):** Your program should accurately fill the parts in memory dedicated for the ”OS” by implementing this method. you can use the ”initos-test” program and compare the output to ”initos-result.txt”.
- **allocMem and freeMem (20 pts):** Assuming that `initOS` is correct, your program should be able to allocate memory for pages using the correct page table and then free it using these methods. You can use the ”mem-test” and ”mem-test2” programs and compare the outputs to ”mem-result.txt” and ”mem2-result.txt”.
- **createProc and loadProc (15 pts):** Assuming that `allocMem` and `freeMem` are correct, your program should be able to read obj files and create a process and load it to the CPU accurately. You can use the ”proc-test” program and compare the output to ”proc-result.txt”.
- **Modify mr and mw (10 pts):** Assuming `createProc` and `loadProc` are correct, your program should be able to read and write values from

an address with respect to `PTBR`. You can use the ”mw-mr-test” and ”mw-mr-test2” programs to comapare your outputs.

- **tyld and thalt (15 pts):** Assuming `mr` and `mw` work correctly, your program should be able to switch to the next process when `tyld` and `thalt` instructions are used. Make sure you save the metadata of the process before loading the next one. You can test `tyld` and `thalt` by running the sample3.sh script and comparing the output to sample3-result.txt.

- **tbrk (15 pts):** Your program should be able to allocate or free a page when the brk instruction is used if there is enough free space. You can test brk by running the sample2.sh script and comparing the output to sample2-result.txt.

- **Report(10 pts):** Your report should briefly explain your implementation including any helper methods you used.(-5 pts if your report is not in pdf format).

8 Sample Runs

In this section, we provide some sample runs. Please run `"make"` command first in order to generate every test, object file and sample run.

8.1 Running the VM

The vm program takes obj files as arguments. Every input program consists of one code file and one heap file. A sample execution command is as follows:

```
$ ./vm code.obj heap.obj
```

if you want to run multiple programs, you need to give the code/heap duo for each program. Below is an example for two programs:

```
$ ./vm code1.obj heap1.obj code2.obj heap2.obj
```

8.1.1 Tests:

You are given six unit tests that check fundamental components of your implementation. The correct result for each test is given in text files under the same directory. Makes sure your implementation passes every test before moving on to sample runs.

```
$ tests/initos-test
Occupied memory after OS load:
mem[0|0x0000]= 1111 1111 1111 1111 (dec: 65535)
mem[3|0x0003]= 0001 1111 1111 1111 (dec: 8191)
mem[4|0x0004]= 1111 1111 1111 1111 (dec: 65535)
```

8.1.2 Samples:

You are also given five sample scripts along with their results as text files in the assignment folder. You can compare the output of these tests to the respective text file. Since sample output is very large, one easy way to compare them is to redirect the output of your program into a different text file and using the diff command. No output means the files are identical.

```
$ samples/sample1.sh > samples/my-sample1-result.txt
$ diff samples/sample1-result.txt samples/my-sample1-result.txt
```

Note that neither the given tests nor the samples are testing for every case. You can create your own tests using the given programs (or with new ones). You can use given tests and samples as example bases for your own tests.