Sabanci University

Operating Systems

CS 307

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

Release Date: 25 May 2024
Deadline: 3 June 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 segmentation and implementing some system calls for handling multiple processes. The existing implementation by Andrei Ciabaonu 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 thid 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 apply segmentation 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 segmentation.

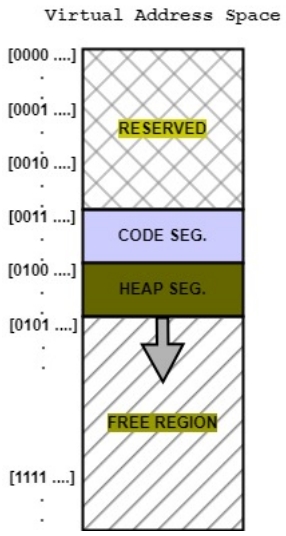

Figure 1: A sample VAS representation.

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 Program Counter (PC) of the currently running process starts from the address `0x3000`. We preserve this assumption and add another restriction saying that the size of the code segment is fixed and it is 4KB. 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 second chunk.

The second and the last segment of the VAS is the heap. Heap addresses begin with 01. Initial heap size is also 4KB. However, heap is a dynamic segment. During the execution of a process, it might grow or shrink with `brk` system call. In Figure 1, the third chunk represents the heap segment in the VAS with the initial default size.

Considering the VAS in Figure 1, any access to the addresses beginning with `0000`, `0001`, `0010`, `0101`, `0110`, `0111`, `10` and `11` must result in a segmentation fault.

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 segments to it. Reserved region on the top and the free space at the bottom are not represented in it.

We also assume a structure on the physical memory. It is represented in Figure 2. The first 4KB (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.

Three `uint16_t`'s are *curProcID*, *procCount* and *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.

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 PCB can be added to the process list. Otherwise, the new process cannot be created and the VM program terminates.

PCBs begin from the physical address 12 and each PCB consists of 6 `uint16_t`s. We have a padding between the *OSStatus* and the first PCB. Insider 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. *BSC_PCB* and *BDC_PCB* denotes the base and bound values for the code segment whereas *BSH_PCB* and *BDH_PCB* are the same values for the heap segment, respectively. Offsets of these fields from the beginning of a PCB block are defined with the same names inside the C file. Please note that all fields have the same type: `uint16_t`. Base values are physical memory addresses and bound values are number of `uint16_t`s in this segment. Aforementioned fields must appear in a PCB exactly in this order.

In Figure 2, *procCount* is 2 and *OSStatus* is `0x0000`. For the PCB starting at address 18, *PID_PCB* is 1, *BSC_PCB* is 12288, *BDH_PCB* is 4096, *BSH_PCB* is 8190 and *BDH_PCB* is 2048.

A new PCB is always allocated immediately after the last PCB. Moreover,

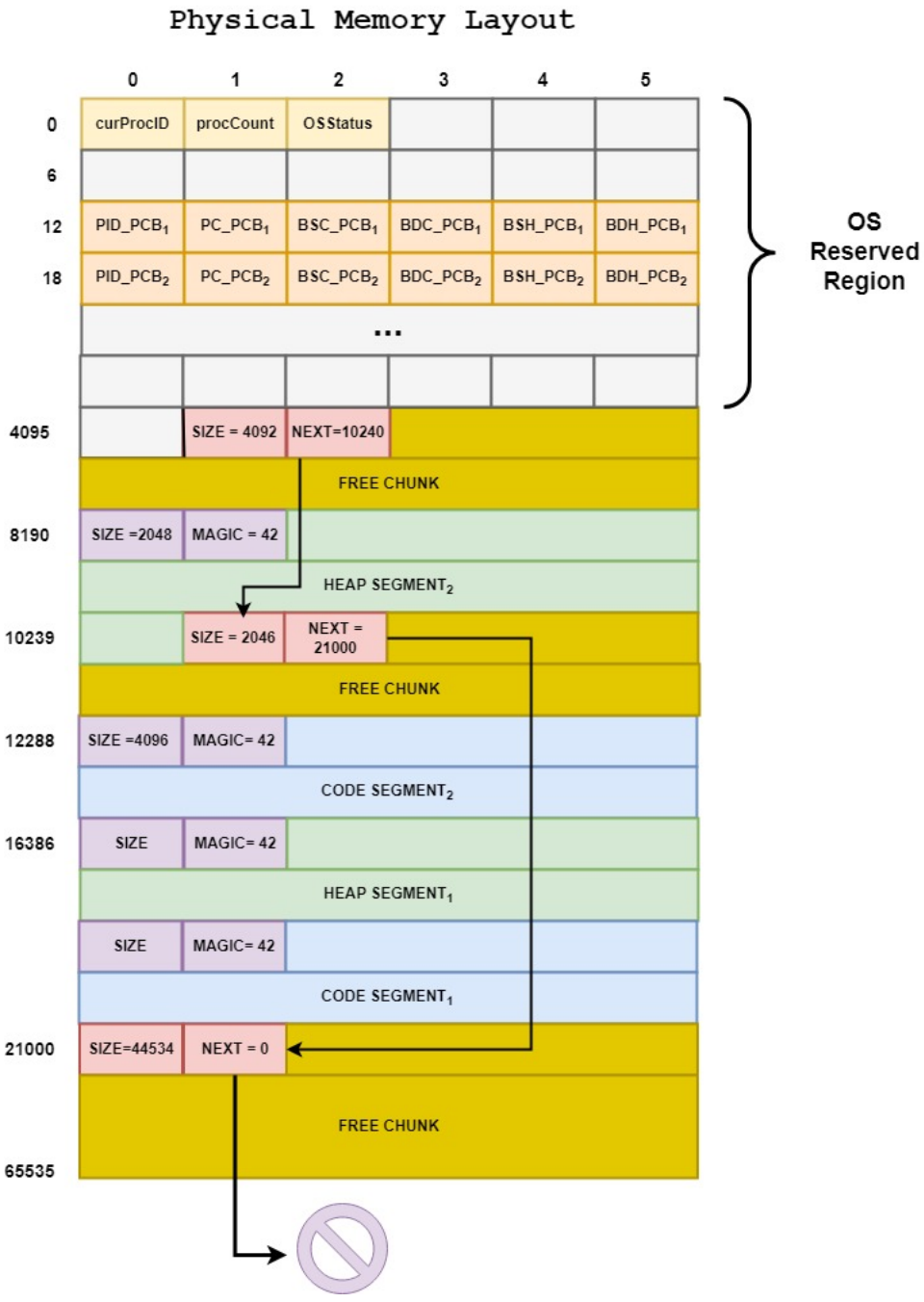

Figure 2: A snapshot of the physical memory with the OS, and two user processes. Actual layout that your program will generate might be different. Here, segments are placed arbitrarily to demonstrate how headers work and arbitrary sized chunks might exist.

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.

3.1 Free Space and Segment Size Management

Physical memory addresses after 4KB are reserved for the heap and code segments of processes. Since heap segments might have variable sizes, 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, some metadata about chunks must be kept. This is achieved by adding header regions to the beginning of both allocated and free chunks following the methodology explained in Chapter 17 of the OSTEP book. Header size is the same and consists of two `uint16_t`s. For both allocated and free chunks the first field keeps the size of the chunk excluding the header size. The second field is the magic number for allocated chunks and it is value must be 42. For the free chunks, the second field acts as the next pointer address, pointing to the beginning of the header of the next free chunk.

Free chunks constitute the free list and they are ordered with respect to their beginning addresses. Firstly, there is always a header of the free chunk in the address 4096 even though it can be empty. This chunk is always the first node in the free list. Other chunks are ordered with respect to the beginning address of their header. The last free chunk's next pointer points to the address 0. In Figure 2 free chunks, their headers and next pointers are demonstrated.

It can be useful to observe that an allocated chunk can be differentiated from a free chunk by its second field. A free chunk's next pointer cannot be 42, since it is an address inside the OS region of the physical memory.

4 New Trap Types

In the original VM implementation, behaviour 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 is executing, PC, base and bound registers are stored to the PCB of the currently running process. 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, base and bound values are restored from its PCB to the CPU registers. 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, size of the heap segment of the current process is modified according to the new size value stored in the register $R0$. If the new heap size is bigger than the old size and the new heap overlaps with an allocated region or overflows to the OS region, your implementation must exit.

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 executing 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 *RBSC*, *RBDC*, *RBSH*, *RBDH* registers, representing the base and bound registers of code and heap segments of the currently running process, respectively.

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 code segment of the new process.

Lastly, we added some macros related to OS bookkeeping and PCB fields that might be useful while implementing your procedures. These macros can be found in the `vm.c` file under the commented region called *"New OS Declarations"* with explanations.

5.2 Parts to be Modified

Memory access methods `mr` and `mw`: Given a 16-bit address, first four bits can be used for determining its segment. If it begins with `00` but not with `0011`, it is an invalid address and the method should prompt the message `"Segmentation Fault."`. Otherwise, you should compute the offset part (remaining 12-bits), compare it with the bound register of the heap segment and if the offset is bigger, prompt the invalid address with the message `"Segmentation Fault Inside Heap Segment."`. Similarly, you can perform a bound check inside the code segment.

If the given address is invalid, your program must terminate after printing the error message. Otherwise, it has to compute the correct physical memory address using the appropriate base register and the offset and perform the access.

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, the code and heap segments and the PCB of the terminating process must be freed. For freeing the segments, you can use `freeMem` method explained below. For freeing the PCB, you must only set its PCB's `PID_PCB` field to `0xffff` and do not touch the other fields in order to leave a footprint in the memory.

5.3 New Methods to be Added

The `initOS` Method: Before doing anything, this method is called for initializing the OS related parts of the physical memory. Basically, it sets the *curProcID* to `0xffff`, *procCount* to 0, and *OSSttatus* to `0x0000`.

The `allocMem` Method: It takes an `uint16_t` input called size and tries to allocate this space from one of the empty chunks. The size input denotes the number of `uint16_t`s requested. The required empty space in reality is $size + 2$ since we have to put a header to the beginning of the new chunk. If there is no free chunk satisfying this condition, this method returns 0.

Otherwise, it allocates the requested space from the end of the free chunk with enough space. It iterates over free chunks in the free list beginning from the first free chunk at the address 4096 and picks the first chunk with enough space. As a result, the allocation strategy is deterministic.

After the new chunk is allocated from the end of the free chunk, this method must initialize the header of the new chunk and return its beginning address i.e., the first address after the header.

If the free chunk has a space less than $size+2$, this chunk is not considered as suitable even though there is enough space inside a free chunk of sizes $size$ and $size + 1$ if the header region of the free chunk is taken into the consideration. This case creates complications and must be ignored as finding this chunk unsuitable.

The previous assumption entails that the first free chunk at the address 4096, never disappears and the header of this free chunk remains even though all of the space inside the chunk is allocated.

In your implementation, you should not apply compaction for obtaining a big free chunk. Even though the total free space is big enough for allocating a new region, your method implementation must return 0 if none of them big is enough itself for the request.

The `freeMem` Method: It is the dual of `allocMem` method. It takes a pointer to the beginning of an allocated region i.e., the first address after the header. If the input address is in the OS region or it is not the beginning of an allocated region[1], this method returns 0.

[1]Hint: Check the magic number in the header to detect allocated regions.

Otherwise, this chunk must be inserted to the free list not violating the beginning address order. Moreover, we expect you to perform coalescing while adding this chunk to the free list. If the next or the previous chunk in the physical memory is free as well, this chunk merges with them and they form a single free chunk. During the coalescing, headers of the disappearing chunks must be included in the total size.

This method is expected to only modify headers of chunks and not to touch tneir content. If the freed chunk 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, base and bound 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 any big enough chunk for allocating the code segment. In this case it prints `"Cannot create code segment."` and returns. Lastly, if there is not any large enough chunk for allocating the heap segment, it prints `"Cannot create heap segment."` and returns.

If none of this failure 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 a 4KB space 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. After the code segment is initialized, it sets the `BSC_PCB` and `BDC_PCB` fields of the new process' PCB.

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

The `tyld` Trap Instruction: 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 resizes the heap segment of the currently running process 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>.`"

If the new heap size is bigger than the old one, it might be the case that there is not enough free space for the new size i.e., the heap overlaps with another allocated region after the expansion. In this case, your program must prompt an error message and return.

There are two such cases. In the first one, the next chunk is also an allocated region. In this case, there is no possibility for the heap to grow and the following error message must be printed to the console: `"Cannot allocate more space for the heap of pid <curProcID> since we bumped into an allocated region."`.

In the second case, the next chunk is a free chunk. In this case, the heap can grow as much as the size of the next chunk. Note that even though it is possible for the heap to grow as much as the size of the free chunk plus its header, we do not allow it since the management of the free list becomes more difficult. If the new heap size is bigger than the old size plus the next chunk size, it is impossible to extend the heap to this size because the chunk following the free chunk must be an allocated chunk due to coalescing. In this case, the following error message must be printed to the console: `"Cannot allocate more space for the heap of pid <curProcID> since total free space size here is not enough."`.

You must also be careful when the heap shrinks. In this case, a new free chunk emerges. You have to add this chunk to the free list preserving its

sortedness or coalesce it with the adjacent free chunk.

In this method, you might need to erase some headers or modify their position. While doing this, you have to delete the data in the old header. However, you should not touch the content inside chunks.

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 03 June 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 (15 pts):** Assuming that `initOS` is correct, your program should be able to allocate certain amounts of memory and then free it using these methods. You can use the "mem-test" program and compare the output to "mem-result.txt".

- **Coalescing (10 pts):** Assuming that `allocMem` and `freeMem` are correct, when you free subsequent chunks of memory, they should merge and create one large free chunk. You can use the "coalesce-test" program and compare the output to "coalesce-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 the base and bound values of the process. You can use the "mw-mr-test" program and compare the output to "mw-mr-result.txt".

- **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 (10 pts):** Your program should be able to increase the size of the heap 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 five 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 program load:
mem[0|0x0000]= 1111 1111 1111 1111 (dec: 65535)
mem[2|0x0002]= 0000 0000 0000 1100 (dec: 12)
mem[3|0x0003]= 0000 0000 0000 1100 (dec: 12)
mem[4096|0x1000]= 1110 1111 1111 1110 (dec: 61438)
```

8.1.2 Samples:

You are also given four 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
```