COP 6611: Operating Systems

Project 1 - Memory Management

Due Date: 02/26/2019

Objectives

This assignment asks you to implement the core functionality of a paged memory management system.

All work should be done individually.

Assignment Setup

- **DO NOT COPY-PASTE commands from the pdf.**
- Please increase the memory for your VM to 2GB.

Code Checkout

As mentioned in the previous assignment, we will be using git to distribute the assignment code. For this, you will have to add a new upstream remote repository to fetch the code. You should have received an invitation to join as collaborator to os-s19 repository. Please accept it. If you haven't let us know immediately. Please follow these instructions to fetch the code and merge your solution from project 0. If you

First, go into your working directory i.e. your project directory from project0.

```
$ cd [PATH TO YOUR PROJECT DIRECTORY]
```

From here on, all the instructions will be run in your project directory.

Adding a remote upstream repository:

```
$ git remote add upstream https://github.com/argus-classroom/os-s19.git
$ git fetch upstream
```

You can verify that a new remote repository was added by running `git remote`. It should output `origin` and `upstream`.

Checkout branch for project1:

```
$ git checkout -b project1 upstream/project1
$ git config --unset branch.project1.remote
```

This will checkout the starter code from the upstream repository and remove the upstream link so that you can work in

your own repository after fetching the code.

Merge your solution:

```
$ git merge project0
```

This will merge your solutions from project0 branch onto the project1 branch. Merging can be tricky and you might have some conflicts while merging. Please go through this git merge tutorial.

OPTIONALLY: If you do not feel comfortable with git yet, you can simply copy and paste your solution files from the previous project in the correct folder for this assignment as there are only 3 files.

Reading

Please read Chapter 8.1 through 8.3 in the textbook. Additionally, please look through chapters 5 and 6 in the Intel 80386 Programmer's Reference Manual, in particular section 5.2 on Page Address Translation and section 6.4 on the Page Table and its protection mechanisms. We recommend that you also skim the sections about segmentation; while our OS uses paging for virtual memory and protection, segment translation and segment-based protection cannot be disabled on the x86, so you will need a basic understanding of it. Please read the above document carefully and make sure that you understand how the page map is structured and how virtual memory works on the Intel x86 platforms.

Part 1: Physical Memory Management (`pmm`)

The operating system must keep track of which parts of physical RAM are free and which are currently in use. In this part, you will implement the physical memory management code in the kernel, so that it can allocate pages of memory and later free them. We will be using a page size of 4096 bytes. Our kernel manages the PC's physical memory with page granularity so that it can use the MMU to map and protect each piece of allocated memory.

Your task will be to maintain data structures that record which physical pages are reserved by BIOS/devices, which are free, and which are already allocated and in use. You will also write the routines to allocate and free pages of memory.

Project Structure

You will implement key parts of the physical memory management module strictly following the abstraction layers that we have built for you. Inside the `kern` directory, you will see a sub directory called `pmm`. This is where all the code related to physical memory management reside. The physical memory management is divided into three abstraction layers, which correspond to the three further sub directories you see under `pmm`. You will need to implement layer by layer, from bottom up, following the descriptions below. Each layer directory contains the following three files:

- `import.h`: The list of functions that are exposed to the current layer are declared and documented here. You are supposed to implement the layer functions using only the functions declared in import.h. This way, you do not have to look at the lower layers to figure out all the details. You shall not modify this file.
- `[Layer Name].c`: The list of functions in the current layer are implemented here. You are supposed to fill in the part marked as `TODO`.
- `export.h`: The declarations of the functions of the current layer that are exposed to the upper layers. You shall not modify this file.

Testing The Kernel

We will be grading your code with a bunch of test cases, part of which are given in `test.c` in each layer's sub directory. If you have already run `make` before, you have to first run `make clean` before you run `make TEST=1`.

- Make sure your code passes all the tests for the current layer before moving up to the next.
- You can write your own test cases to challenge your implementation.

Building and running the image

```
$ make clean
$ make TEST=1
```

While writing your code, use this command to run unit tests to verify the working status of your code.

ALTERNATIVE: Building and running the image with provided solutions

Although we encourage you to build on your own solutions, we have provided compiled solutions that you may use. Use the SOL flag as shown below for this.

```
$ make clean
$ make TEST=1 SOL=1
```

Normal Execution

Building the image

```
$ make clean
$ make
```

ALTERNATIVE: Building the image with provided solutions

```
$ make clean
$ make SOL=1
```

These two commands will do all the steps required to build an image file. The make clean step will delete all the compiled binaries from past runs. After the image has been created, use one of the following execution modes:

Executing WITH QEMU VGA Monitor

```
$ make qemu
```

You can use `Ctrl-a x` to exit from the qemu.

Executing WITHOUT QEMU VGA Monitor

```
$ make qemu-nox
```

Executing with GDB and QEMU VGA Monitor

```
$ make qemu-gdb
```

Executing with GDB without QEMU VGA Monitor

```
$ make qemu-nox-gdb
```

Task 1: The MATIntro Layer

MAT stands for Memory Allocation Table. In this layer, you are going to implement getter and setter functions for a data structure used to store the physical memory information. In `MATIntro.c`, the data structure `AT` is a simple array of a physical allocation table entry structure that stores the permission and allocation status of each physical page (a.k.a. frame). Please make sure you read all the comments carefully.

In the file `kern/pmm/MATIntro/MATIntro.c`, you must implement all the functions listed below:

- `at_is_norm`
- `at_set_perm`
- `at_is_allocated`
- `at_set_allocated`

Task 2: The MATInit Layer

In this layer, you will implement an initialization function that detects and sets the maximum number of pages available in the machine, and initializes the allocation table `AT` you've just implemented in the lower layer (`MATIntro`).

Recall from **Project 0** that during the startup, the bootloader loads from BIOS the information on which memory ranges are reserved by BIOS/devices and which are available to be used by OS. In the code provided to you, we have provided a thin API over that memory map for you to use in this task. Please read the comments in `kern/pmm/MATInit/import.h` very carefully on the set of functions that can be used to retrieve this part of information. More detailed documentations and hints can be found in the comments in MATInit.c.

In the file `kern/pmm/MATInit/MATInit.c`, you must correctly implement the function:

- `pmem_init`

Task 3: The MATOp Layer

In this layer, you will implement the functions to allocate and free pages of memory. Please review the list of functions you may need in import.h. In `MATOp.c`, you are asked to implement a fairly naive version of a physical page allocator and deallocator, and then come up with a slightly optimized version using the memoization. Please review the comments carefully for more details.

In the file `kern/pmm/MATOp/MATOp.c`, you must correctly implement all the functions listed below:

- `palloc`
- `pfree`

Task 4: The MContainer Layer

In this kernel, a ***container*** is an object used to keep track of the resource usage of each process, as well as the parent/child relationships between processes. It is important for the kernel to track resource usage so that buggy/malicious processes can be prevented from using up all the available resources. In our kernel, if a user process attempts to allocate all available memory (*e.g.*, by calling `malloc` in an infinite loop), the kernel will deny all allocation requests once the process has allocated its maximum allowed quota.

Additional Protections:
Note that the only resource we will track is the number of pages allocated by each process. However, we designed the container mechanism in a general way so that we can easily extend it to track other types of resources. One interesting example (and potential research project) would be extending containers to track CPU time as a resource.

To describe containers in more detail, we first need to define a way to distinguish a particular process. We do this via unique IDs. Whenever a process is spawned, it is assigned an unused ID in some range [0, `NUM_IDS`). Every ID has an associated container. ID 0 is reserved for the kernel itself.

When a new ID is created, how do we decide on the maximum quota passed to that ID? One possible solution is to fix some specific max quota for all IDs. This is quite restrictive, however, since some programs may require vastly different resource usage than others. For our kernel, we will define a parent/child relationship between IDs, and we require that parents choose resources to pass on to their children. Each ID has a single parent, and potentially multiple children. ID 0 is called the "root", as it is the root of the parent/child tree and thus the only container without a parent.

Consider any ID i. The fields of i's container are as follows:

- `quota` - the maximum number of pages that ID i is allowed to use
- `usage` - the number of pages that ID i has currently allocated for itself or distributed to children
- `parent` - the ID of the parent of i (or 0 if i = 0)
- `nchildren` - the number of children of i
- `used` - a boolean saying whether or not ID i is in use (if this boolean is false, then ID i is not in use and the values of the other fields of container i should be ignored)

During the execution, there are two situations where containers will be used:

1. Whenever a page allocation request is made (e.g., handling a page fault or handling a malloc system call request); and
2. Whenever a new ID is spawned (the parent ID must distribute some of its quota to the newly-spawned child).

To reason about the relationships between the container objects and the actual available resources, the kernel must maintain the following invariant throughout execution.

Soundness: The sum of the available quotas (i.e., quota minus usage) of all used IDs is at most the number of pages available for allocation.

When writing code to implement containers, be sure that the initialization primitive establishes this invariant, and each other primitive maintains it.

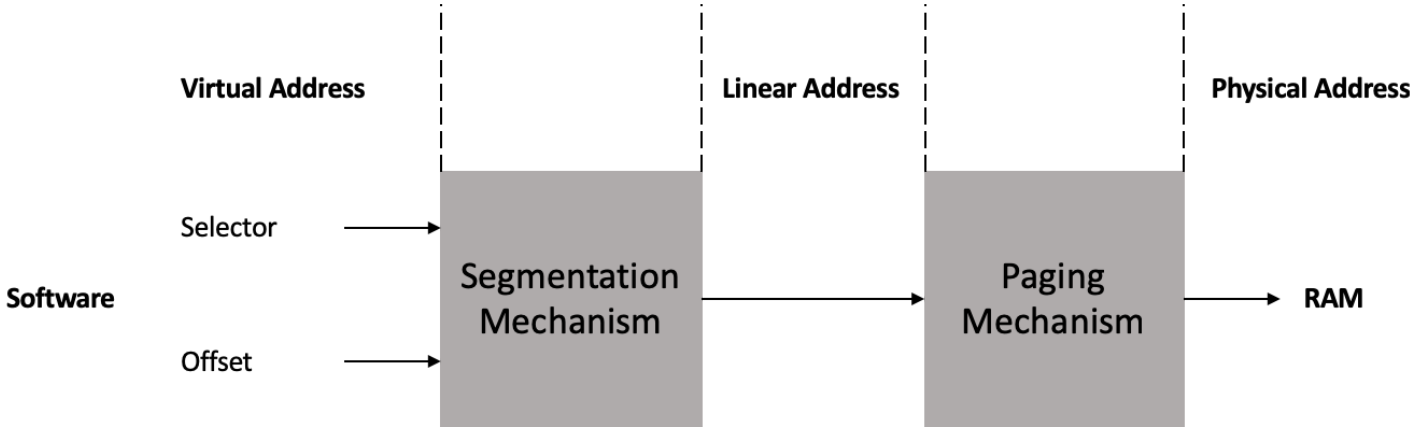
In the file `kern/pmm/MContainer/MContainer.c`, you must implement all the functions listed below:

- `container_init`
- `container_get_parent`
- `container_get_nchildren`
- `container_get_quota`
- `container_get_usage`
- `container_can_consume`
- `container_split`
- `container_alloc`
- `container_free`

Part 2: Virtual Memory Management (`vmm`)

Virtual, Linear, and Physical Addresses

In x86 terminology, a *virtual address* consists of a segment selector and an offset within the segment. A *linear address* is what you get after segment translation but before page translation. A *physical address* is what you finally get after both segment and page translation and what ultimately goes out on the hardware bus to your RAM.



A C pointer is the "offset" component of the virtual address. Here, we installed a Global Descriptor Table (GDT) that effectively disabled segment translation by setting all segment base addresses to `0` and limits to `0xffffffff`. Hence the "selector" has no effect and the linear address always equals the offset of the virtual address.

DEBUGGING:

While GDB can only access QEMU's memory by virtual address, it's often useful to be able to inspect physical memory while setting up virtual memory. Review the QEMU monitor commands from the lab tools guide, especially the xp command, which lets you inspect physical memory. To access the QEMU monitor, press Ctrl-a c in the terminal (the same binding returns to the serial console).

Use the xp command in the QEMU monitor and the x command in GDB to inspect memory at corresponding physical and virtual addresses and make sure you see the same data. You do not need to submit anything for this exercise.

From code executing on the CPU, once we're in protected mode and the paging is turned on, there's no way to directly use a linear or physical address. *All* memory references are interpreted as virtual addresses and translated by the MMU, which means all pointers in C are virtual addresses.

The kernel often needs to manipulate addresses as opaque values or as integers, without dereferencing them, for example in the physical memory allocator. The kernel also often needs to treat an integer as an address or a pointer. If you do not understand the C pointer very well, please spend some time to study it carefully before you get started on this lab.

The kernel sometimes also needs to read or modify memory for which it knows only the physical address. For example, adding a mapping to a page structure may require allocating physical memory to store a page directory and then initializing that memory. However, the kernel, like any other software, cannot bypass virtual memory translation and thus cannot directly load and store to physical addresses. In our kernel, we use separate page structure for each process, and we switch page structures when we switch among different processes. To solve the issue above, we reserve the entire page structure with index 0 for the kernel, i.e., the process 0 is always kernel process. Then we configure the entire page structure 0 as the identity map. This way, whenever the kernel needs to access a physical address, we can switch to the page structure 0, and then access whatever physical address (which is the same as the virtual address) we want. In our kernel, we need to initialize many page table mappings as the identity map, i.e., the entire page structure 0,

and the kernel portion of the memory for the rest of page structures. Instead of repeatedly allocating the same identity second level page tables, we statically allocate one and point every appropriate page directory index to the same page table entry. (see `IDPTbl` in the `MPTIntro` layer).

You will implement various parts of the virtual memory management module strictly following the abstraction layers that we have built for you. Inside the kern directory, you will see a sub directory called `vmm`. This is where all the code related to virtual memory management reside. The virtual memory management is divided into six abstraction layers, which corresponds to the further sub directories you see under `vmm`. You will need to implement layer by layer (except `MPTInit`, which is already fully implemented), from bottom up, following the instructions. Each layer directory contains the same structure as the `pmm` layers.

The MPTInit Layer

In this layer, we call a function (`set_pdir_base`) to set the `CR3` register (or the *Page Directory Base Register*) to the initial address of the page structure #0. You will be writing this function later in the `MPTIntro` layer. The `MPTInit` layer also enables paging on the CPU.

There's no task in this layer.

Task 5: The MPTIntro Layer

In this layer, you are going to implement the getter and setter functions for two data structures used to maintain the processes' page tables. Recall figure 5-12 from the 80386 Programmer's Manual. This layer implements the getter and setter functions for the Page Directory and Page Table for each process. Please make sure you read all the comments carefully.

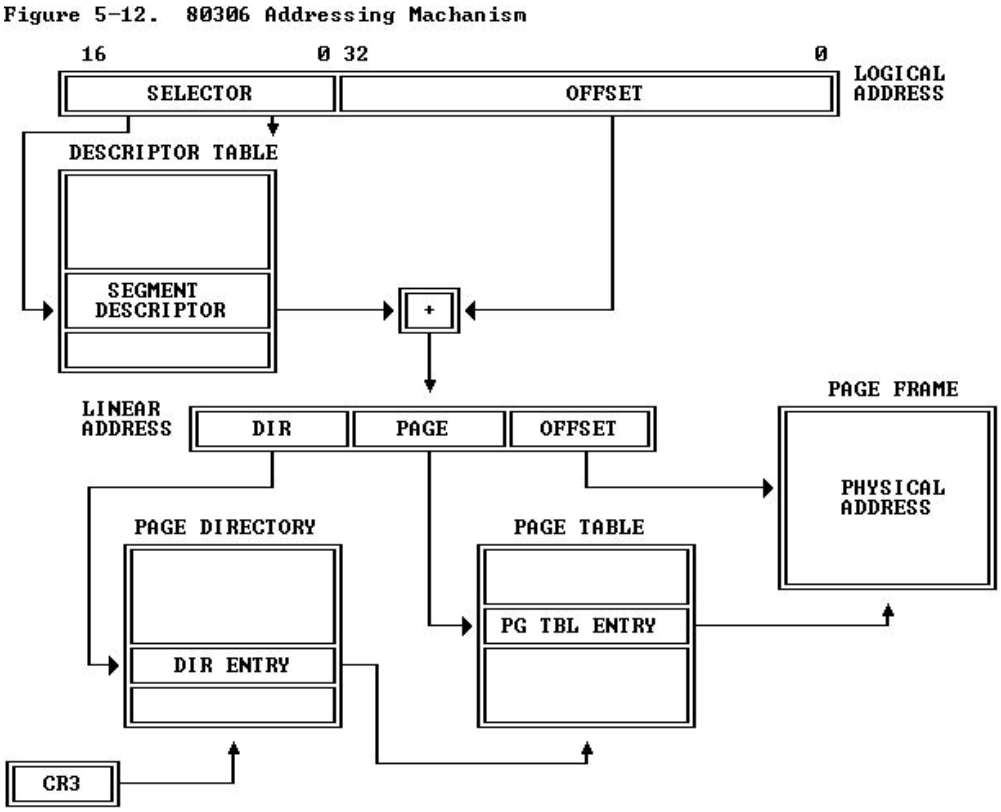


Figure 5-12 from 80386 Programmer's Manual.

In the file `kern/vmm/MPTIntro/MPTIntro.c`, you must implement all the functions listed below:

- `set_pdir_base`
- `get_pdir_entry`
- `set_pdir_entry`
- `set_pdir_entry_identity`
- `rmv_pdir_entry`
- `get_ptbl_entry`
- `set_ptbl_entry`
- `set_ptbl_entry_identity`
- `rmv_ptbl_entry`

Make sure your code passes all the tests for the `MPTIntro` layer. And write your own test cases to challenge your implementation.

Task 6: The MPTOp Layer

This layer provides a wrapper function for calling the functions in the `MPTIntro` layer by providing just the virtual address. So you will need to break up the virtual address into the corresponding components. In the file `kern/vmm/MPTOp/MPTOp.c`, you must correctly implement all the functions listed below:

- `get_pdir_entry_by_va`
- `set_pdir_entry_by_va`
- `rmv_pdir_entry_by_va`
- `get_ptbl_entry_by_va`
- `set_ptbl_entry_by_va`
- `rmv_ptbl_entry_by_va`
- `idptbl_init`

Make sure your code passes all the tests for the `MPTOp` layer. And write your own test cases to challenge your implementation.

Task 7: The MPTComm Layer

In the file `kern/vmm/MPTComm/MPTComm.c`, you must correctly implement all the functions listed below:

- `pdir_init`
- `alloc_ptbl`
- `free_ptbl`

Make sure your code passes all the tests for the `MPTComm` layer. And write your own test cases to challenge your implementation.

Task 8: The MPTKern Layer

In the file `kern/vmm/MPTKern/MPTKern.c`, you must correctly implement all the functions listed below:

- `pdir_init_kern`
- `map_page`
- `unmap_page`

Make sure your code passes all the tests for the `MPTKern` layer. And write your own test cases to challenge your implementation.

Task 9: The MPTNew Layer

In the file `kern/vmm/MPTNew/MPTNew.c`, you must correctly implement all the functions listed below:

- `alloc_page`

Make sure your code passes all the tests for the `MPTNew` layer. And write your own test cases to challenge your implementation.

Running a Process with Virtual Memory

To better illustrate the process of virtual memory and address translation, we have created an extra command in our kernel monitor called runproc. Once run, that command will start a user process defined in `user/proc/dummy/dummy.c`. The process implements a program that allows you to input an arbitrary virtual address to read from or write to. The program is rather limited, and only allows you to enter the address in decimal numbers. Feel free to replace it with whatever fancy programs that you can come up with to test the virtual memory.

A sample run of the program is pasted below:

```
1.  ****************************************
2.
3.  Welcome to the mCertiKOS kernel monitor!
4.
5.  ****************************************
6.
7.  Type 'help' for a list of commands.
8.  $> runproc
9.  Program 0x0010a004 is loaded.
10. Welcome to the user process! (Ctrl - Z to exit)
11.
12. Specify a virtual address to read from or write to.
13. Enter the address: 3489660928
14. Address entered: 3489660928
15. Specify the action: r for read, w for write.
16. w
17. Enter the value you want to write to the address.
18. Value to write (from 0 to 9): 5
19. Page fault: VA 0xd0000000, errno 0x00000002, page table # 1, EIP 0x40000255.
20. Successfully wrote the value to the virtual address. You can double check it with the read command.
21.
22. Specify a virtual address to read from or write to.
23. Enter the address: 3489660928
24. Address entered: 3489660928
25. Specify the action: r for read, w for write.
26. r
27. The value at virtual address 3489660928 is 5.
28.
29. Specify a virtual address to read from or write to.
30. Enter the address:
31. Exiting from the user process.
32. $>
```

Pay special attention to the line `#19` that is in bold italic font. That line is printed from our page fault handler (see `kern/lib/trap.c`). The page fault was triggered because it was trying to access a non-mapped virtual address (address `3489660928`, that is `0xd0000000`). When a page fault is triggered due to this reason, the page fault handler dynamically allocates a page for the corresponding virtual address and returns back to the instruction that caused the page fault (in this case, it is one of the instructions in the user process).

Note that, during the middle of your virtual memory implementation, running the above command could produce many random errors, or the machine may frequently reboot itself. Don't be panic. This is because your virtual memory management layers have not been fully implemented. Once the layers are fully implemented, you should be able to successfully start the user process as shown above.

Since we have not set up the process management code, in this assignment, we have to hack the kernel in a way such that it is under an illusion that we have the user process set up and running. However, note that the current "user process" is actually running in ring0 mode. That means you can actually access arbitrary virtual address using the program we provide. So don't be surprised that when you try to write to some address, it crashes the kernel or results in some funny behavior. In the next assignment, we will learn how to build up the process management layers to schedule and run multiple user processes in the ring 3 mode with memory protection.

Hand in Procedure

You will submit your assignment via git by creating a tag. The following instructions will explain the needed steps. To get started first change into the root directory of your repository.

Git: Adding Changes AND Git: Committing Changes

These steps are same are same as the previous project. Please `git add` and `git commit` all your changes.

Git: Push Branch: FIRST TIME ONLY

This is a checkpointing step and submits your local branch and commit history to the remote git server. Doing this means your changes will not be lost.
First time push for the new project1 branch:

```
$ git push -u origin project1
```

The additional -u flag has to be used for the first time you push the branch. This sets the proper remote tracking branch.
Subsequent push:

```
$ git push origin project1
```

Submitting: Creating a tag

```
$ git tag -a v1 -m "Submission for project 1."
```

This will create a `tag` named `v1` from the latest commit on your local.

Submitting: Pushing the tag

```
$ git push origin v1
```

This will push the `tag` named `v1` to the remote repository. Note that this step is different from “Git: Push Branch” step above.

Re-Submission

If you make changes after creating the tag, you will have to forcefully update the tag and (if already pushed) forcefully push the tag.

Follow steps for Adding Changes, Committing Changes and Push Branch as above.

Re-Submitting: Updating the tag

```
$ git tag -a -f v1 -m "Re-Submission for project 1."
```

This will forcefully update the existing tag named `v1` from the latest commit on your local.

Re-Submitting: Forcefully Pushing the tag

```
$ git push -f origin v1
```

This will forcefully push the `tag` named `v1` to the remote repository.