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.

Normal Execution

Building the image

```
$ make clean

$ make
```

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
```

Note: If you are executing your code on the os cluster, you will HAVE TO use the `nox` version of commands.

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`)

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.

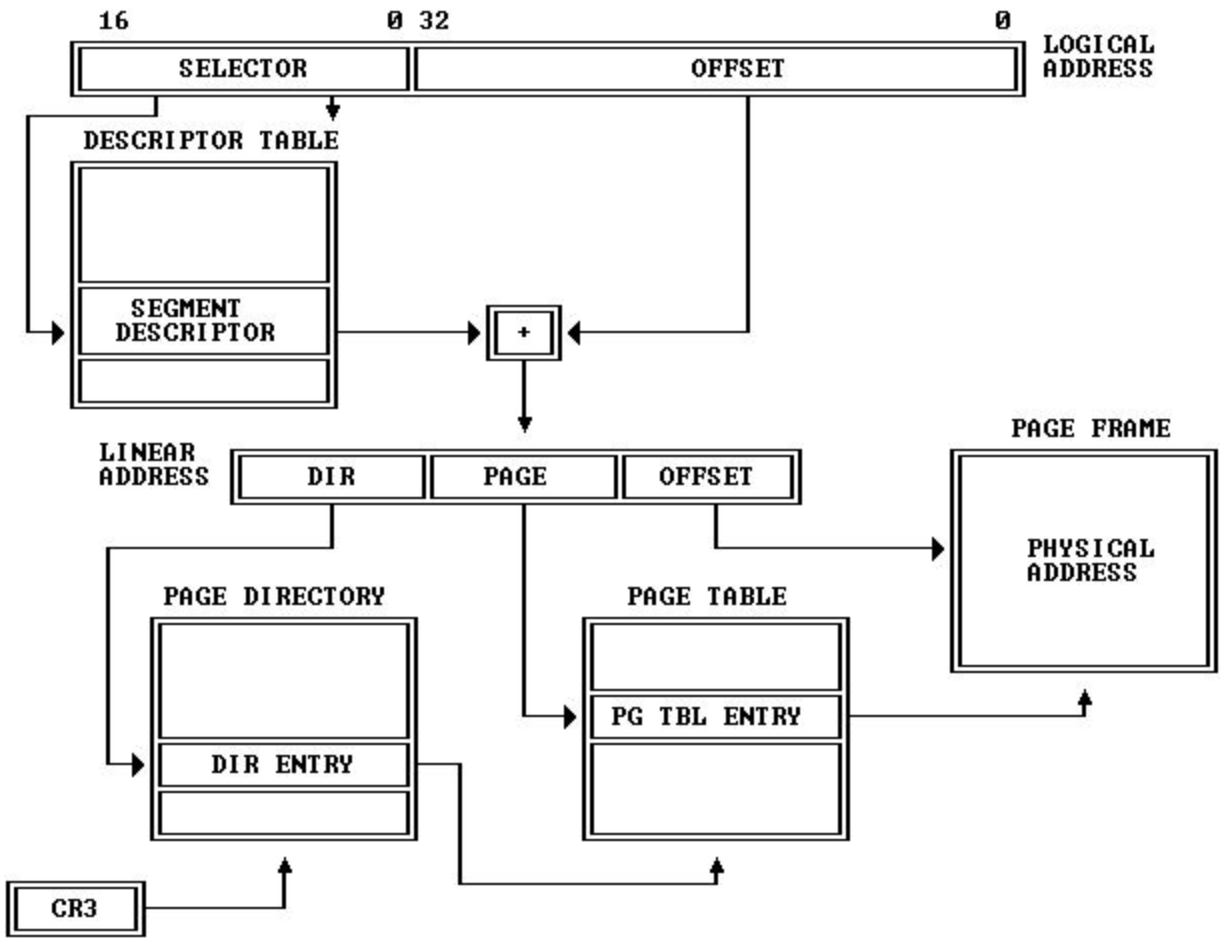

Figure 5-12 from 80386 Programmer's Manual.

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.