PROJEC 3 OVERVIEW

-- IMPLEMENT THE BASIC PAGING SYSTEM IN OUR KERNEL.

BY: SIDIAN WU (WITH SLIGHT MODIFICATIONS BY DILMA DA SILVA)

Two important concepts (page & frame)

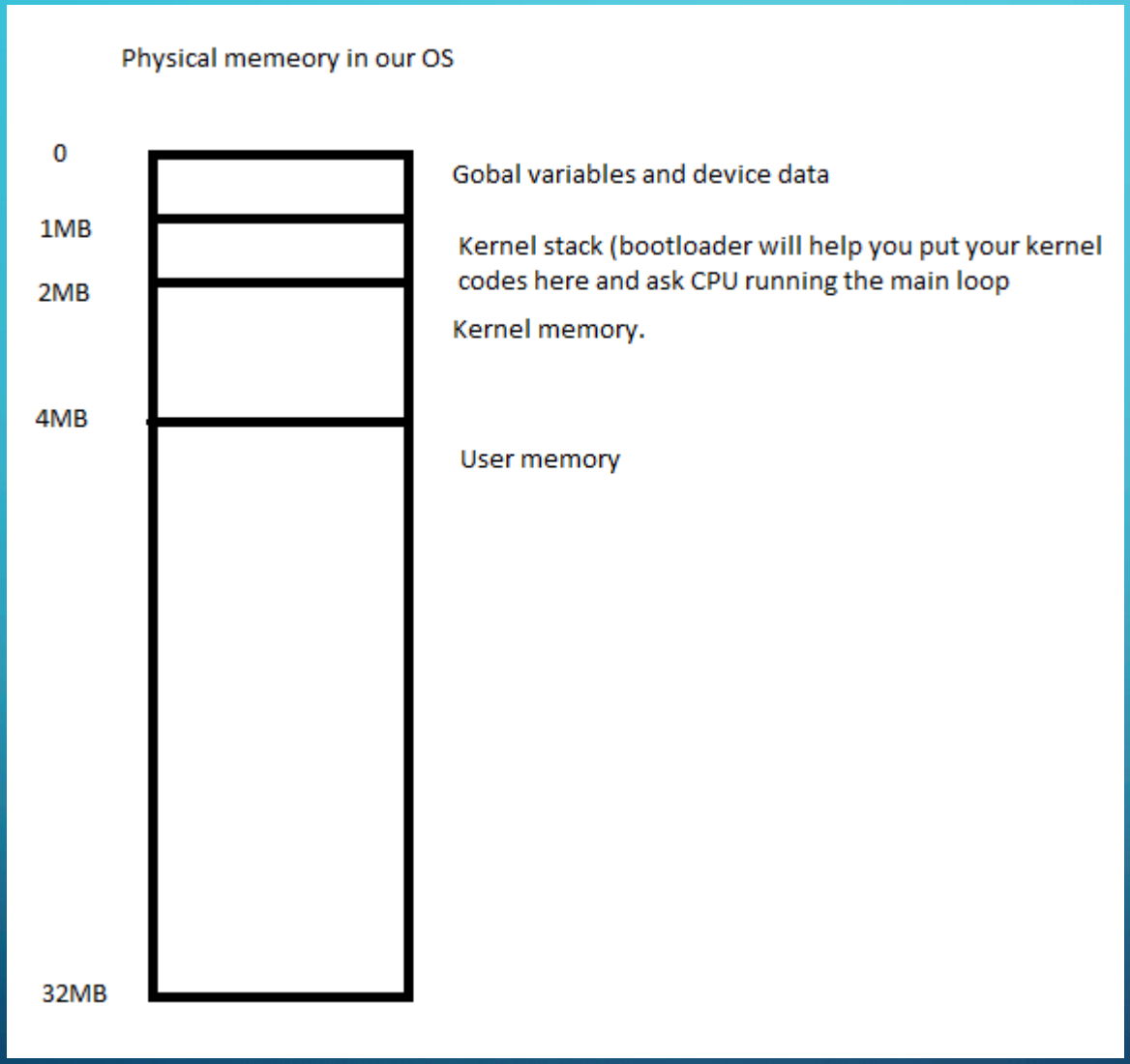
We break physical memory into fixed-sized blocks called frames and logical memory into blocks of the same size called pages.

Note that though page size vary from different OS, the size of page and frame has to be consistent in each OS. People also called frame -- page frame.

In P2, you were supposed to implement our own frame class. In P3, you are supposed to implement your own page table class to enable paging system.

The main files have to be created or modified would be:

frame_pool.H and frame_pool.C (for P2)
page_table.C (for P3; very unlikely you need to change the .H)



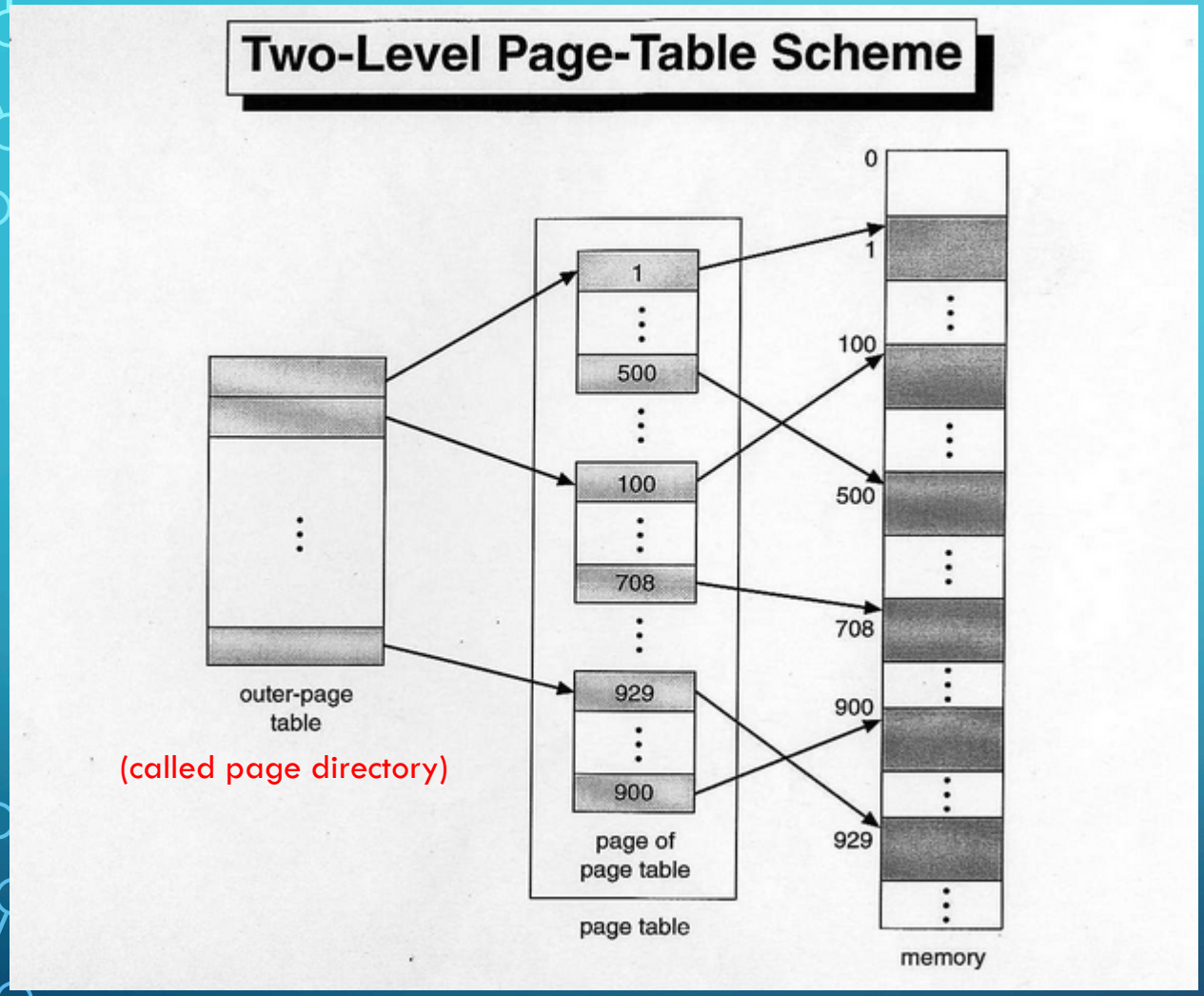
Layout of our OS memory

1. 32MB in total.
2. The first 4MB memory is for kernel. The rest is for user program (User space memory). The pages fo kernel are directly mapping to the corresponding frames while they could be different for user memory

Frame Pool that you should have at this point

Supposed we cut off our memory into 4KB pieces, and we call each piece a frame. Frame pool is the manger of all the frames.

In our project, we will created two frame pool that are in charge of kernel memory space (2MB-4MB) and user memory space (4M-32MB).
You can see these too pools being created in kernel.C



This is the best picture I could find our page table work in the same as it. People called it two-level p table. The outer-page table consi pointer to page_table. In our pro we call it **directory**. Both director and page table are 4 KB long wh means they are able to store at m 1024 entries. Therefore, a directo able to map into 1024 page tab and each page table is able to m to 1024 virtual page numbers to corresponding frames.

Since each frame is 4KB long, the total memory size a directory is a to map would be 1k * 1k * 4KB = 4GB.

Let get further into Page_table entry . Since each entry will be a frame address (not a frame number), and each frame address is always a multiple of 4KB, the last 12 bits for each frame address will always be 0. (4KB is 2^12). (In other words, offset is 0)
The first 20 bits are enough to indicate a unique frame in the memory and we can use the rest bits as flags as needed.
More information could be found at http://www.osdever.net/tutorials/view/implementing-basic-paging

31.....12	11...9	8...7	6	5	4...3	2	1	0
Page frame address	Avail	Reserved	D	A	Reserved	U/S	R/W	Present

Page frame address = Physical address of memory(either the physical address of the page, or the physical address of the page table)
Avail = Do what you want with this
Reserved = Reserved by Intel
D = Drity
A = Accessed
U/S = User or supervisor level
R/W = Read or read and write

Now let's look at our Page_table

PageTable::init_paging()

Nothing difficult here. Pass the parameters to your private variables and remember to set paging_enable to false now.

PageTable::PageTable()

This is the most difficult part in this project I think. It's not difficult to implement but it is difficult to understand what do we have to do here. Before we talk about it, let's have a overall picture of how our OS is working.

When your bochs boots, start.asm is read first from your floppy image file. It set up some global variables we don't need to care of and put kernel.C into the kernel memory stack (1MB-2MB). Then it tells CPU run the code from main loop and your OS wakes up! The paging system is not yet working right now. The CPU treats all addresses in your program as physical memory addresses.

In fact, it is not until “PageTable::enable_paging();”, does CPU understand your purpose and start to treat each memory addresses as logical address.

Why doesn’t CPU mess up when we switch the memory reading mode? Because the memory for kernel is directly mapped. In another word, the logical addresses here are the same as physical.

Now, let's look back to PageTable::PageTable()

Are we supposed to map all memory address (32MB) into our page_table as soon as we create Page_table instance?

No, for many reasons. In fact, we only deal with direct mapping here and leave other stuff to PageTable::handle_fault().

Therefore, the first thing we need to do is to initialize page_directory (we need give it a frame to store entries)
We also need a frame for the first page table in the directory since the first page table maps address from 0-4MB, this is just the size as our kernel memory address. We initialize the first page_table by filling each entries with the real physical address start from 0KB.
(Remember you also need to add flags for each entry though most of them won't be used in this project)

Done!

PageTable::load()
PageTable::enable_paging()

Those two methods are not difficult to implement. But you need some knowledge towards operations on registers. There is a file called paging_lwo.asm that has already defined the functions you need. Read the handout and try to invoke them here.

PageTable::handle_fault()

All the page fault happens when your page table is missing the entries that are supposed to tell CPU where the physical address are.

More specific this project (in Kernel.C)

```
int *foo = (int *) FAULT_ADDR;
    int i;

    for (i=0; i<NACCESS; i++) {
       foo[i] = i;
    }
```

Those statements assign addresses that have no record in our paging system (addresses larger than 4MB)

Therefore, the most important thing we have to do in handle_fault() is to make a record in our paging system and ask frames for them. We can find the fault address from CR2 register. Class Console will help you shout out some words in the screen while page fault happens as well.