Part a

1. Download [3000physicalview.tar.gz](https://homeostasis.scs.carleton.ca/~soma/os-2021f/code/3000physicalview.tar.gz) and unpack with the command tar xzf 3000physicalview.tar.gz. Compile 3000physicalview and 3000memview2 using the provided Makefile (i.e. by running make).

Downloaded using the link provided with wget <https://homeostasis.scs.carleton.ca/~soma/os-2021f/code/3000physicalview.tar.gz>

Then unpacked with tar xzf 3000physicalview.tar.gz.

Then compiled with make in the terminal

1. Insert 3000physicalview by running make insert. Confirm that the module is inserted using lsmod.

Ran make insert in the terminal and verified the 3000physicalview.ko has been inserted as I can see the file in the folder referring from the make file and insert command and running lsmod we can see 3000physicalview at the very top

1. Examine the call to copy\_from\_user and copy\_to\_user on lines 120 and 132 of 3000physicalview.c. Consider the following:
   * How are these functions different from put\_user that we have seen in the previous tutorial?

put\_user  is used to put single values like int,char,or long which the previous code seems to be using a single char of 1 from and to the userspace. The other put\_user uses each char from the message as well.

copy\_from\_user copies a block of data from user space into a kernel buffer, this uses a source buffer from the user space , a destination buffer in the kernel space and a length defined in bytes

copy\_to\_user copies a block of data from the kernel into the user space, this uses a pointer to the user space buffer, a pointer to the kernel buffer, and length defined in bytes

* + Why are these functions necessary? Couldn't we just access the userspace address directly? What would happen if we did?

You may have errors as youre supposed to be dealing with pointers of the userspace and kernel buffers and break some things if you tried to access the userspace directly. So attempting to access the userspace memory directly can generate a page fault which kernel code isn’t allowed to do. This would result in a death of the process that made the system call

1. 3000physicalview exposes its API to userspace in the form of an ioctl(2) call. Consider the following:
   * What is an ioctl? How is it different from a read or write system call? Hint: check man 2 ioctl.

An ioctl is a control device, it is also a system call the manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files may be controlled with ioctl requests.

Ioctl proves a way to perform miscellaneous, device specific operations. So it’s a more generic function an theoreticallt everything can be done using ioctl. But when there are operations most devices implement its better to use a more specific function like read and write. This allows the compiler to perform argument type checking which a generic function like ioctl cant check if you’ve given the right parameters for that operation so it can break stuff.

* + How does 3000physicalview implement its ioctl? What arguments does it take?

Ioctl takes a file pointer struct, unsigned int cmd, unsigned long address

It implements it by allocating kernel memory for their physicalview\_memory struct, then gets virt from userspace, uses a call helper to get physical mapping for virtual address, then gives physical mapping back to userspace. Lastly cleanups the memory at the end.

* + How does 3000memview2 call the ioctl? What arguments does it pass to the ioctl?

It calls ioctl in line 36 of 3000memview2 in the report\_memory function and takes the arguments of fd being the file descriptor, the PHYSICALVIEW\_WALK and an unsigned long of the address of the memory that’s passed into ioctl

Part b

1. With 3000physicalview inserted, run 3000memview2 and examine the output. Note that it presents virtual memory addresses on the left, and physical addresses on the right. Are these mappings consistent with what you expected?

With the 3000physicalview inserted from the earlier code of make insert, I ran 3000memview2 using ./3000memview2 and saw the output of the virtual memory addresses on the left and the physical addresses on the right.

This is what I expected as most addresses begin with 0x0000 and the virtual and physical addresses are different than each other

1. Compare 3000memview2 with 3000memview from [Tutorial 2](https://homeostasis.scs.carleton.ca/wiki/index.php/Operating_Systems_2021F:_Tutorial_2). What is similar about their code, and what is different? How similar is their output?

The two are mostly the same except the more recent 3000memview2 uses snprintf() as well as report\_memory function in the main when going through the buffer, as well as implementing report memory function instead of just using printf() to display. The outputs would be similarish I assume.

1. Do you notice a pattern in the virtual addresses of buf[i]? Is this same pattern present in the physical addresses? Why or why not?
2. Run 3000memview2 a few more times and consider the following:
   * Are the virtual addresses the same or different between runs? How about physical addresses?
   * Some physical addresses don't seem to be changing between runs. Which ones? Why do you think this might be the case?

Part c

1. As root run trace -M 100 -K 't:kmem:kmalloc printf "allocated %d bytes at address 0x%llx" args->bytes\_alloc, args->ptr' to trace the next 100 slab allocations and print the kernel stack. You may wish to pipe this output into less to read it more easily. What do you notice about the kernel's virtual addresses compared to what you have seen in userspace? Hint: Check the most significant hex digits.
2. As root run trace -M 100 -K 't:kmem:mm\_page\_alloc printf "allocted 2^%d pages at page frame number %lu" args->order, args->pfn' to trace the next 100 page allocations and print the kernel stack. You may wish to pipe this output into less to read it more easily. Based on what you can see, does page allocation seem to differ from slab allocation? How so?