HW 4

ECEN 449

Jason Gilman

Objective

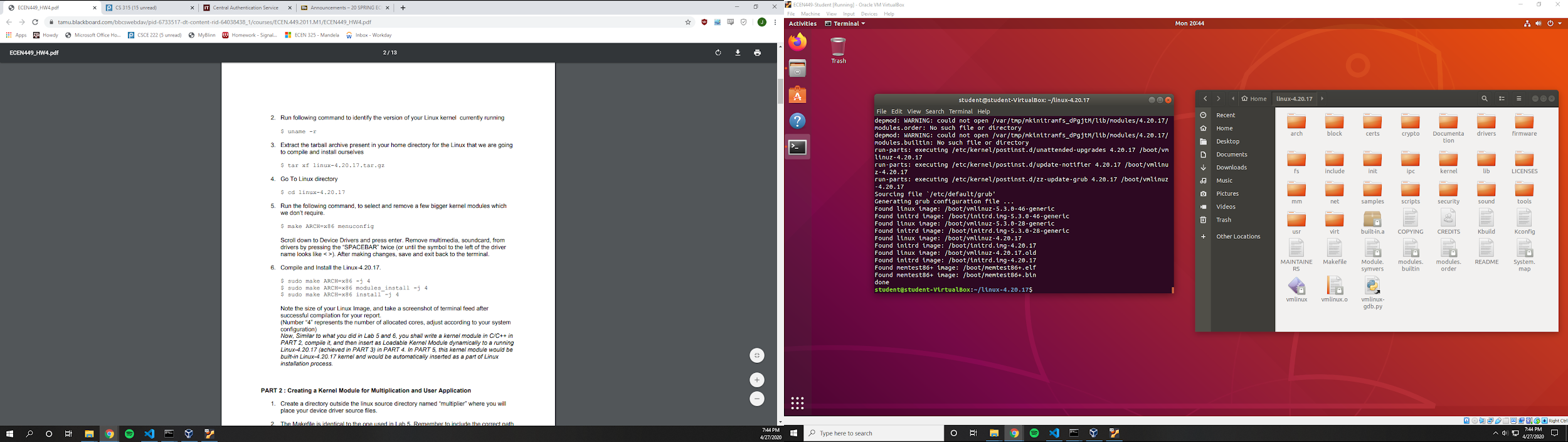
In this homework, we were tasked with designing a linux kernel module that could be used to multiply 2 numbers together. The module will be run dynamically, and then it will be built in a Linux-4.20.17 Kernel to study the differences in implementation.

Process

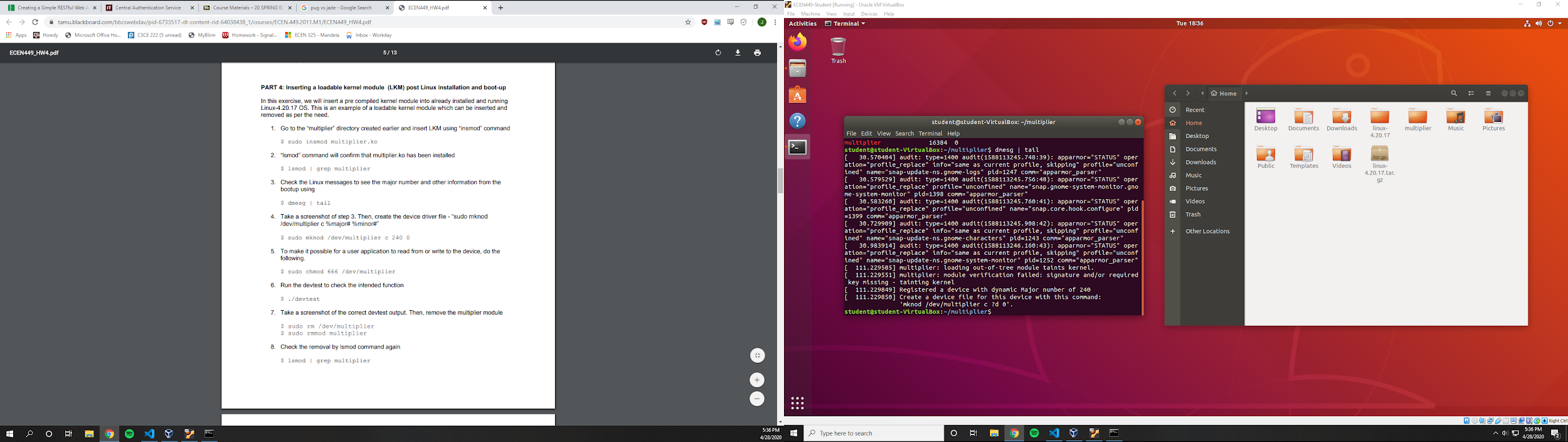
* Set up a virtual linux environment to compile c code, as well as interact with the kernel.
* Compile target linux image. This is the linux kernel that we will be designing for. Ultimately, the module will be dynamically run on this image, as well as built in as a kernel module in a later step.
* Next, the source code for the linux module was developed
  + After the module was instantiated, the device was given a Major number.
  + A local buffer was created to handle the data coming in and out of the user buffer.
  + When the module was writing, data in the form of integers was taken out of the user space by using the get\_user() command and put into our local buffer. This allowed us to compute the multiplication within our module.
  + When the module was reading, the data that had been operated on was then put back into the user buffer using the put\_user() command.
* The module was then compiled, which created an actual kernel module device file.
* Next, a program to test the module was created - devtest.
* This program was created to write numbers into the module, and get the result back to check the correctness of the modules implementation.
* Next, the target image of linux was booted up. Here, the kernel module was loaded into the image. This was to show dynamically loading and running the module.
* The module was run once again using the devtest program, but this time it was being dynamically run in the kernel.
* The module was then removed from the kernel.
* Next, the module was built into the target kernel.
  + The device driver makefile for the linux kernel was modified so that it would recognize our new module.
  + The module was also given documentation such that the compiler could recognize how the module could be used, and its requirements. This was kept in the Kconfig file.
  + The device driver file’s Kconfig was also updated such that it would recognize our modules kernel configuration file.
* With our module built into the target image’s file system, the linux image was rebuilt so that all of the modules were freshly installed. This allowed our module to be built into the kernel of the target image.
* Once again the target image was booted up. Our module was found loaded in the kernel, and it had a major number assigned to it.
* The module was tested with devtest one last time to demonstrate the built in module functionality.

Results

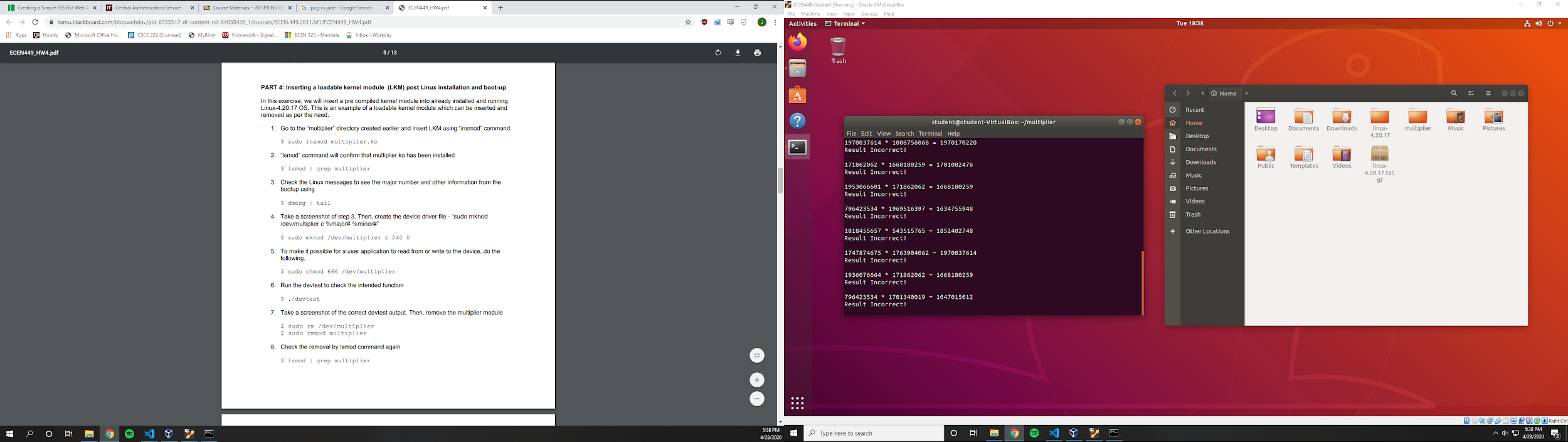
The following were screenshots taken throughout the above process:



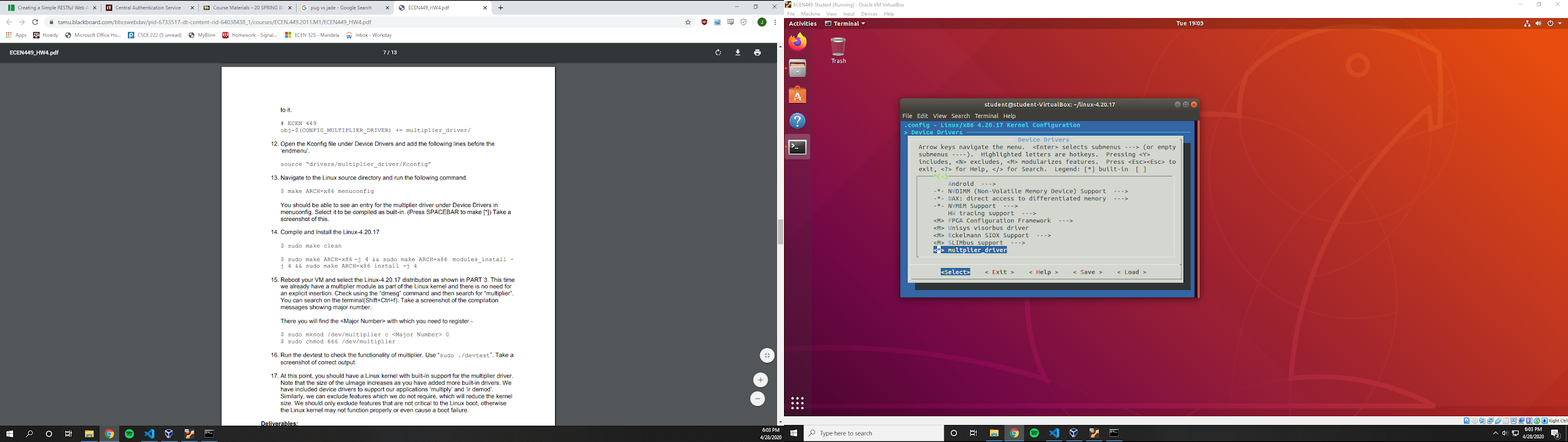
Showing that the target image of linux (Linux 4.20.17) was successfully compiled.



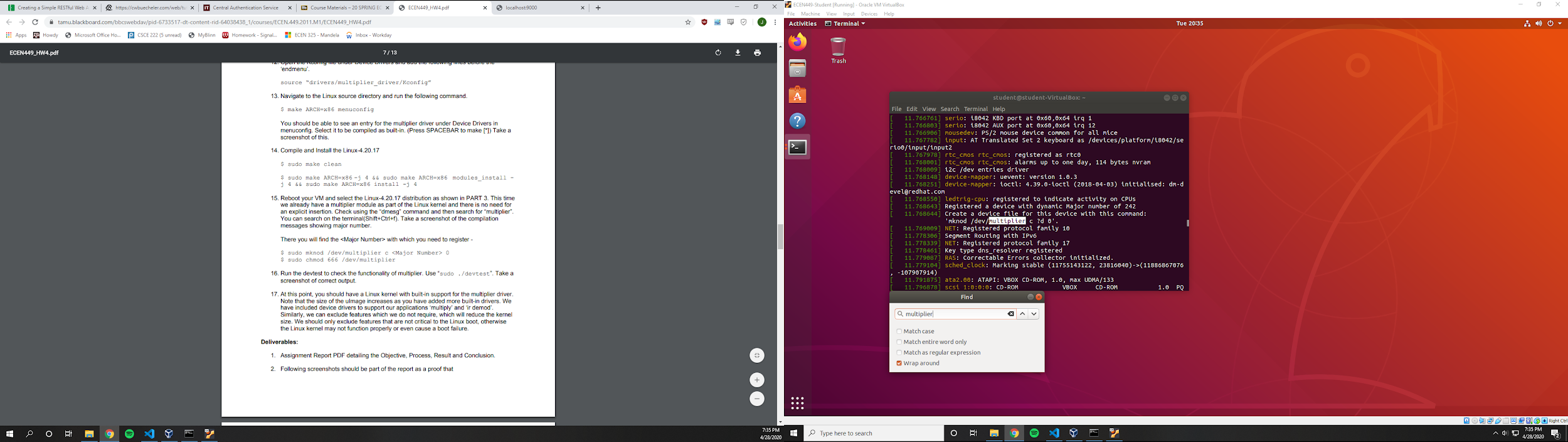
Showing the module loaded into the target kernel. Complete with the assigned major number.



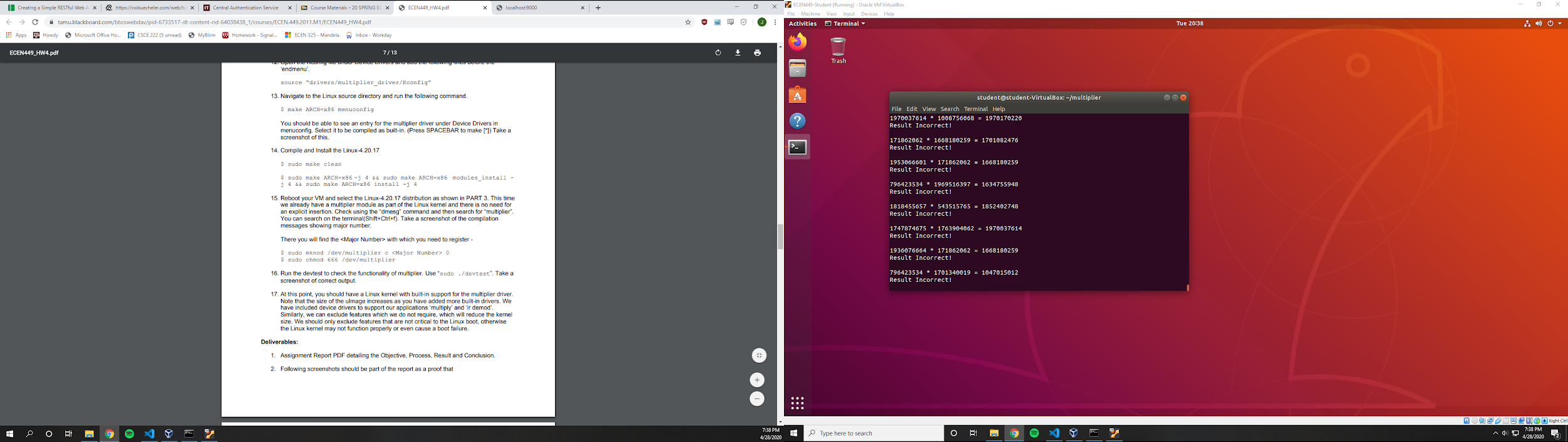
Showing an example output from the module when dynamically loaded. Clearly this is incorrect. An explanation for this could be logic errors when pulling and storing data in the user space.



Showing the menuconfig screen where the multiplier module is built into the kernel.



Showing the successful compilation of linux with the built in multiplier module. This shows the major number that can be used to access our module.



Showing the output of the multiplier module that was built into the kernel. Clearly this is incorrect. Possible explanations could be logic errors in the manipulation of data in and out of the user and local spaces in the source code for the module.

Conclusion

Kernel modules are able to interact with user software by manipulating data through a user’s buffer that is filled in their program. Usually, data is taken out of the user’s buffer and manipulated upon and then put back so that their program can receive the output of the module. Two ways of implementing a kernel module were shown in this homework, one being dynamically loading the module, while the other was building the module into the target kernel.

Questions

1. Loadable kernel modules are more dynamic, meaning that they have less dependencies on the kernel. If changes need to be made to the module, the entire kernel does not need to be built again. This is a downfall of the built in modules. Although, kernel modules are able to be more efficient with memory, because they are built in with the rest of the modules when the kernel is built. On the other side, loadable modules experience performance issues because they are not efficient memory wise - due to fragmentation. Although, loadable modules are able to save memory because they are only loaded into the kernel when they are used. Since loadable modules are loaded into the kernel after the kernel is built, modules that contain errors do not crash the kernel, while built in modules cause a system wide error.
2. I liked that the labs covered many things. Also moving from software to hardware implementations was interesting because I was able to analyze the differences, and what makes each better than the other. I wish that more of the labs were workable from home on virtual machines like this was. The lab times were short compared to how long it took to compile some of the linux files, although some were dependent on the lab computers, as the TA would not be able to help if it were done on a personal machine. Booking extra time in the labs is competitive, which hinders the students ability to get adequate time to finish the labs.