Anthony Redamonti

Syracuse university

Lab 1

CIS-657 Principles of operating systems

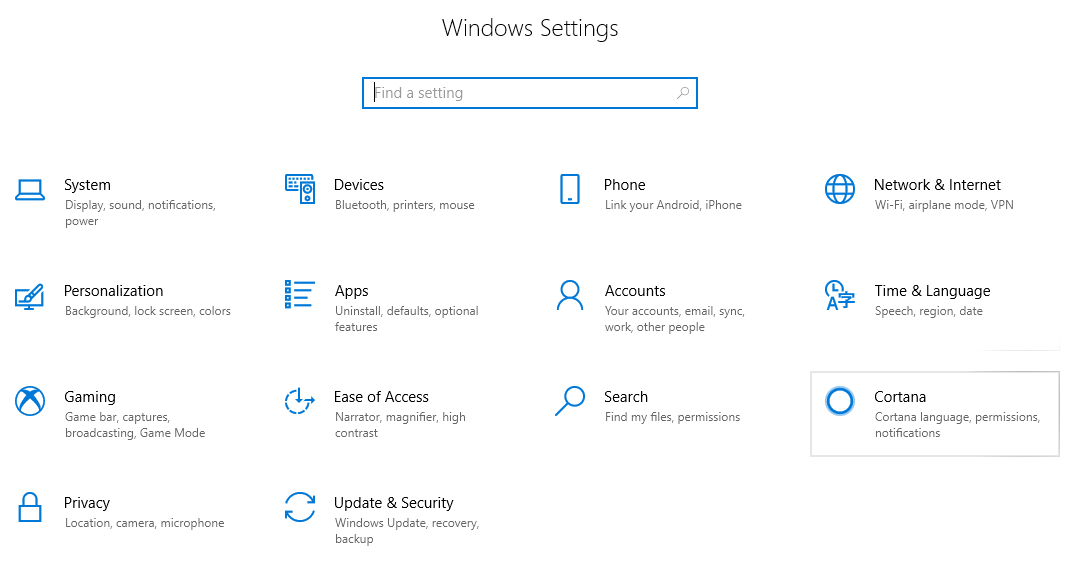
prof. mohammed abdallah

Section A: Installation and Setup

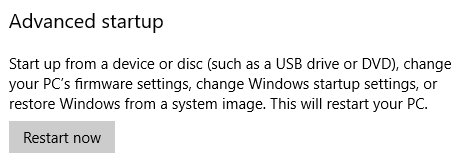
In the lab instructions, there is a list of four different options of configurations between VirtualBox and XINU versions. Option 1 was chosen, as it was the most tested and reliable option. The following setup was performed on a Windows 10 Enterprise Host OS.

Option 1: VirtualBox 5.2.28 and XINU 2019 appliances

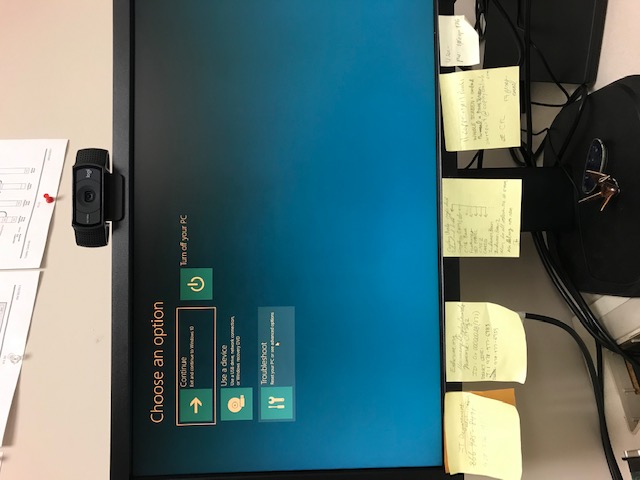
First, the Virtualization must be enabled on the host PC. Click on the home button (bottom left corner) and select “Settings”. Next, select “Update & Security” shown below.



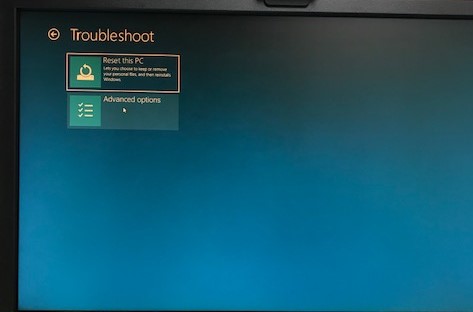
Click on “Recovery” then under Advanced Setup, select “Restart Now” shown below.



Select the “Troubleshoot” button shown below.



Select “Advanced options” shown below.



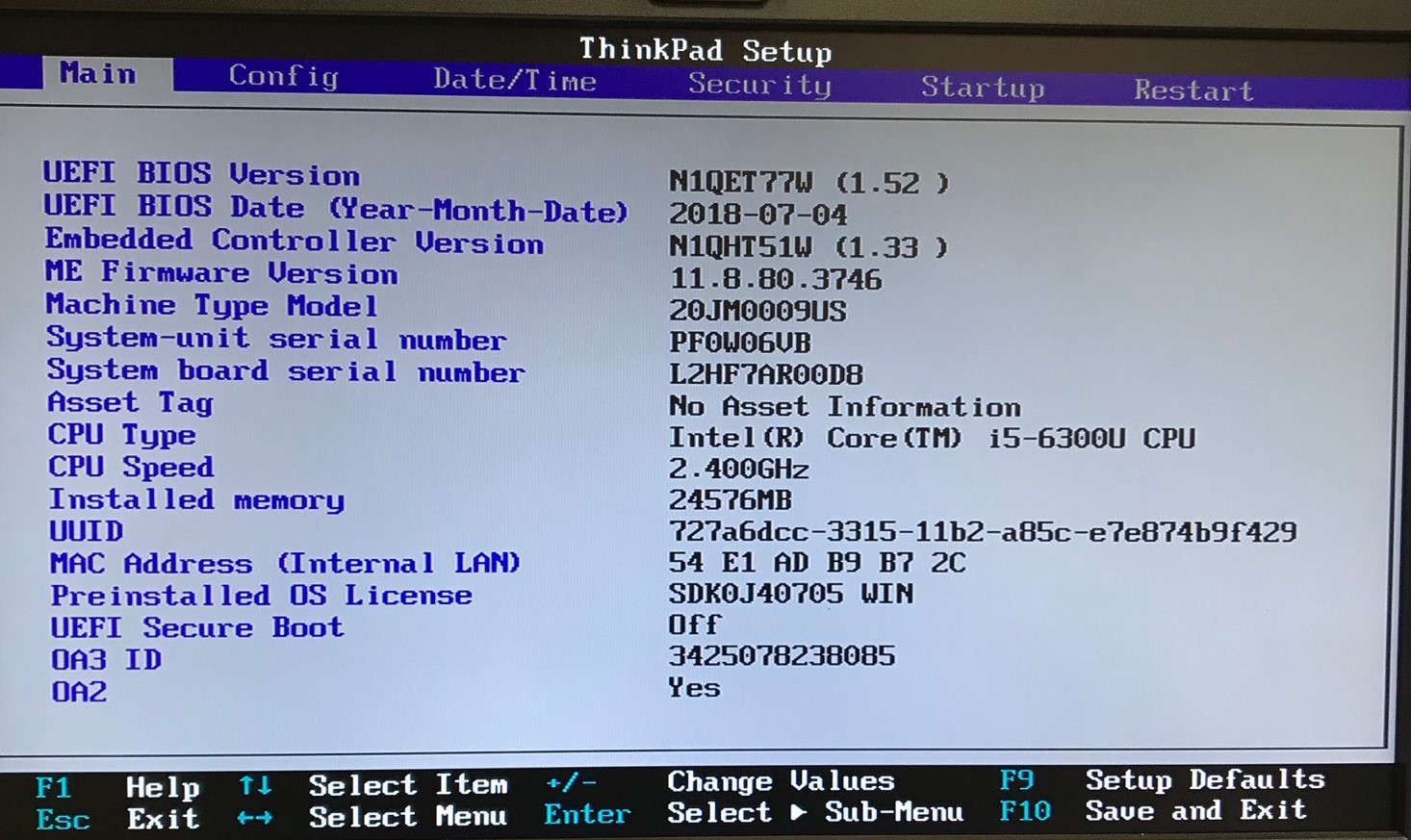
Next, select “UEFI Firmware Settings” shown below.



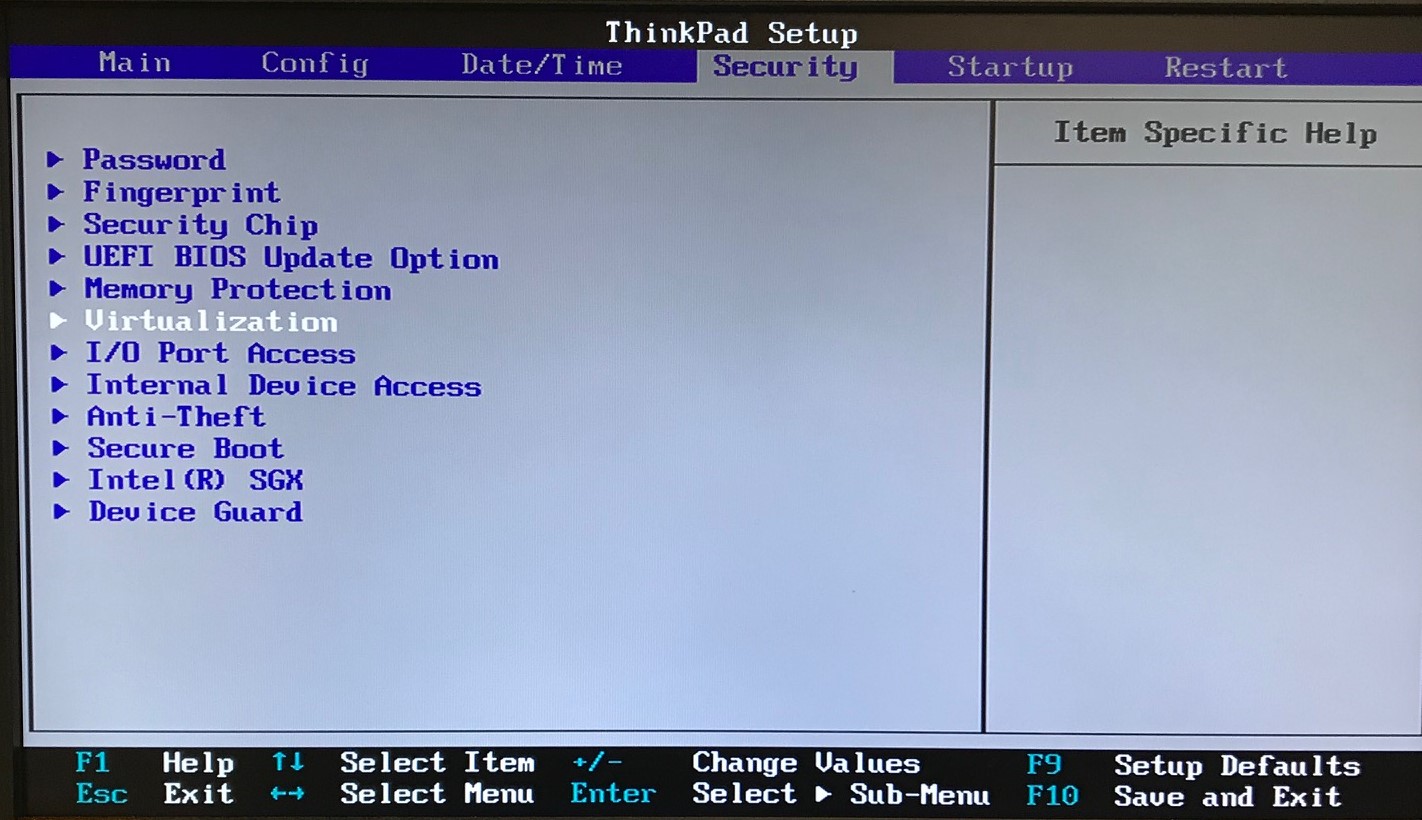
Click “restart” shown below.



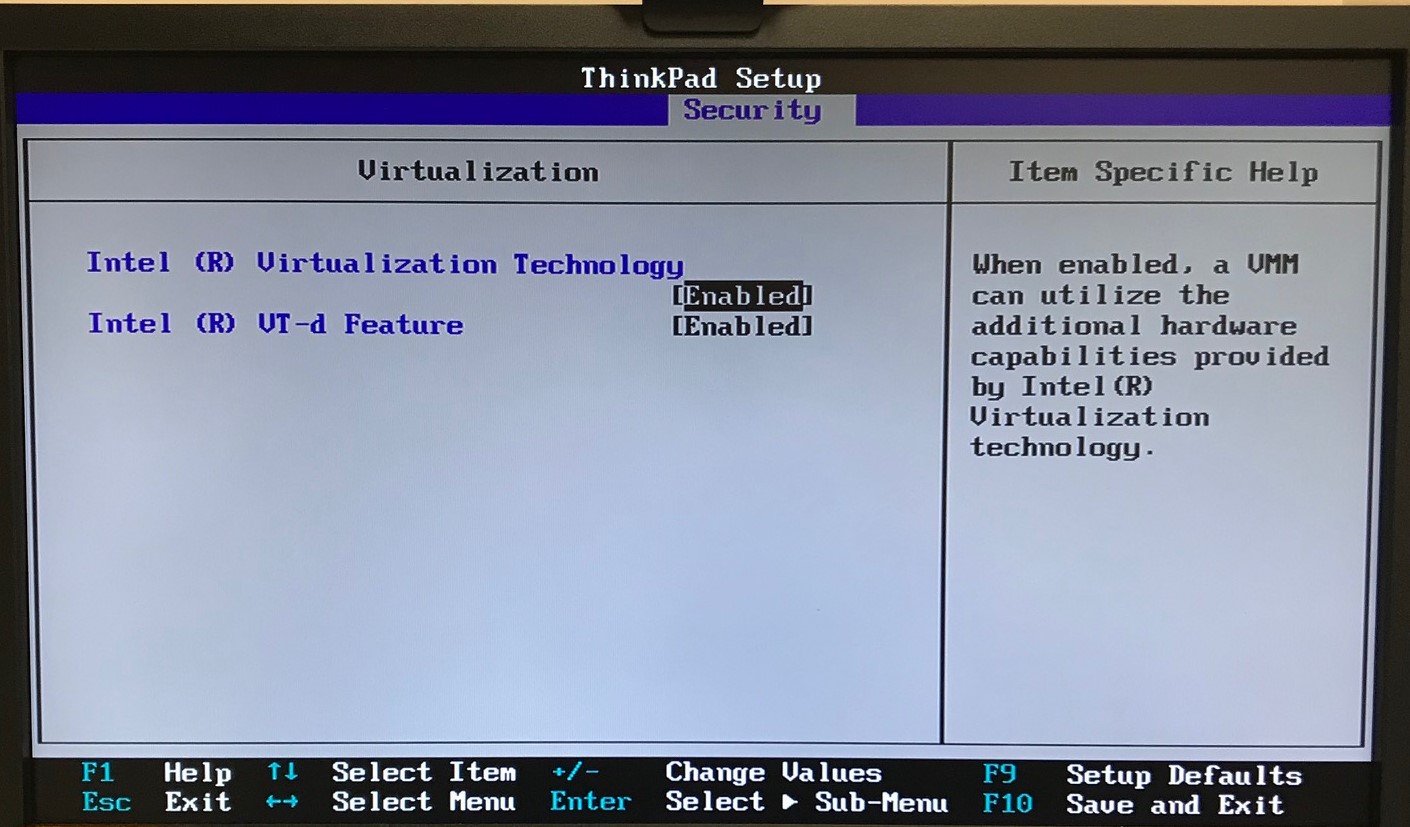
Navigate to the “Security” tab.



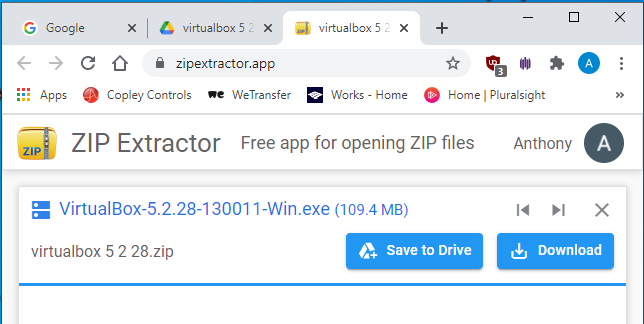
Navigate to “Virtualization”.



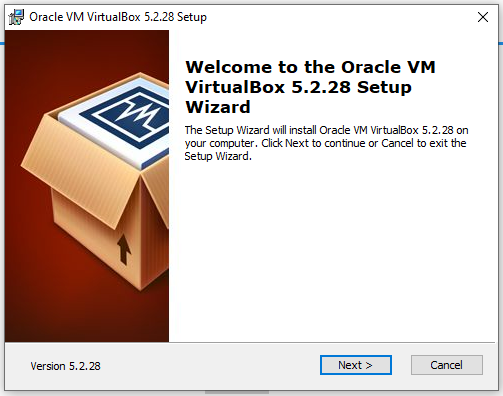
Change the virtualization setting to “Enabled”.



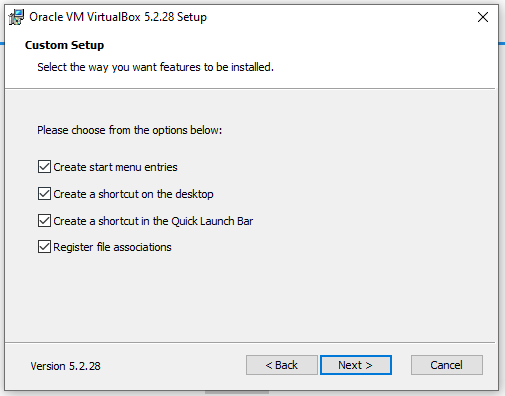
Now that the virtualization feature has been enabled, download the VirtualBox Version 5.2.28 executable from the link provided in the lab instructions.



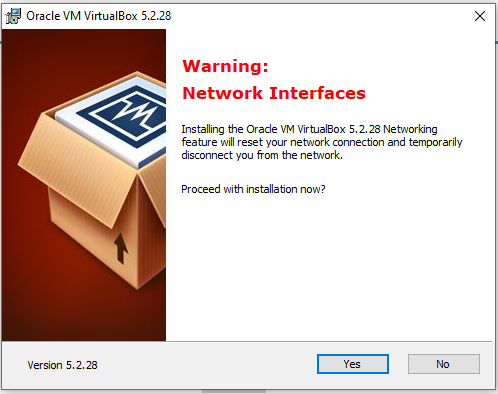
Click “Next” shown below.



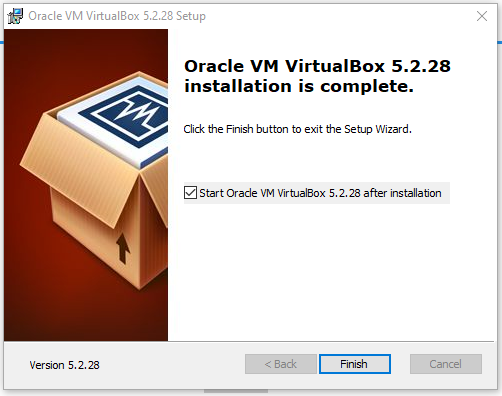
Click the “Next” button provided that all the boxes in the options menu are checked.



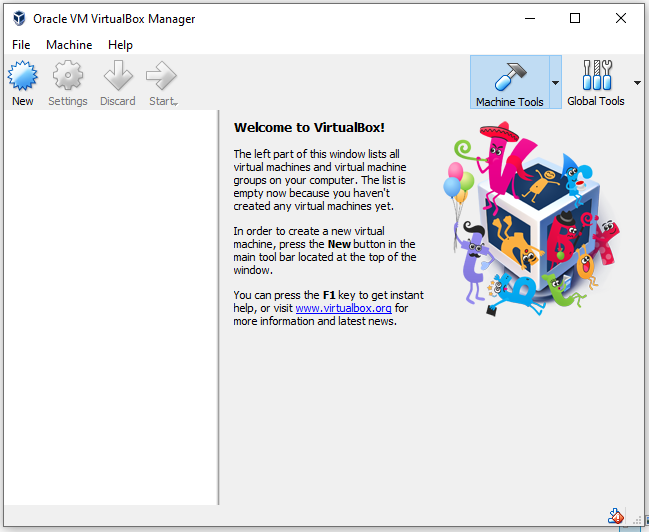
Click the “Yes” button shown below.

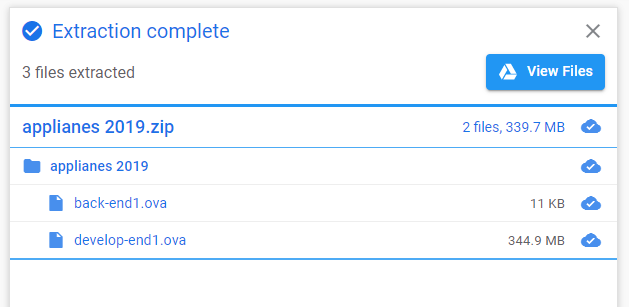


Click the “Finish” button shown below.

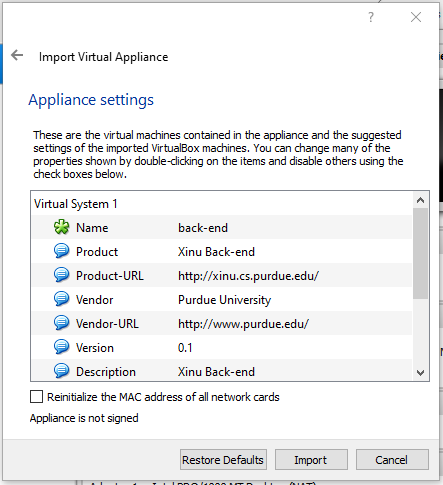


The screen below should appear.

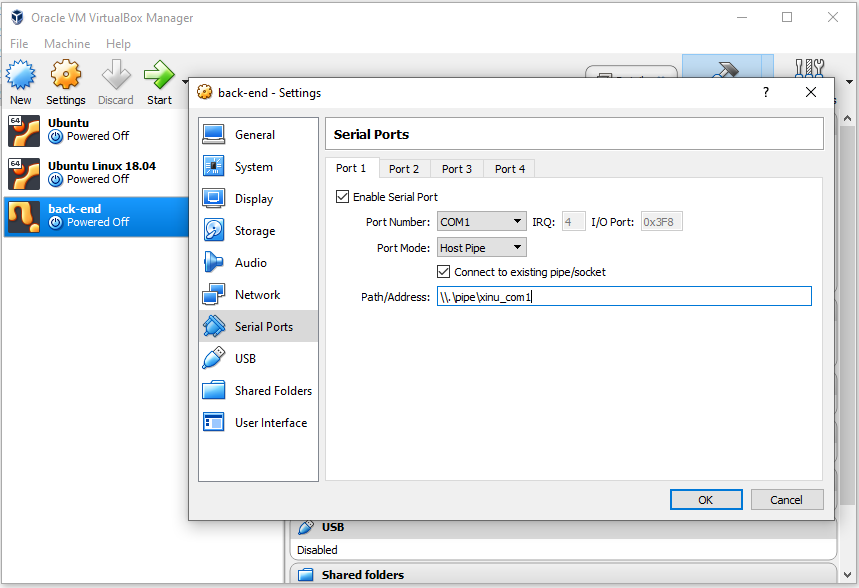


Close the VirtualBox application and download the two main files from the Xinu 2019 appliances link provided in the lab instructions.

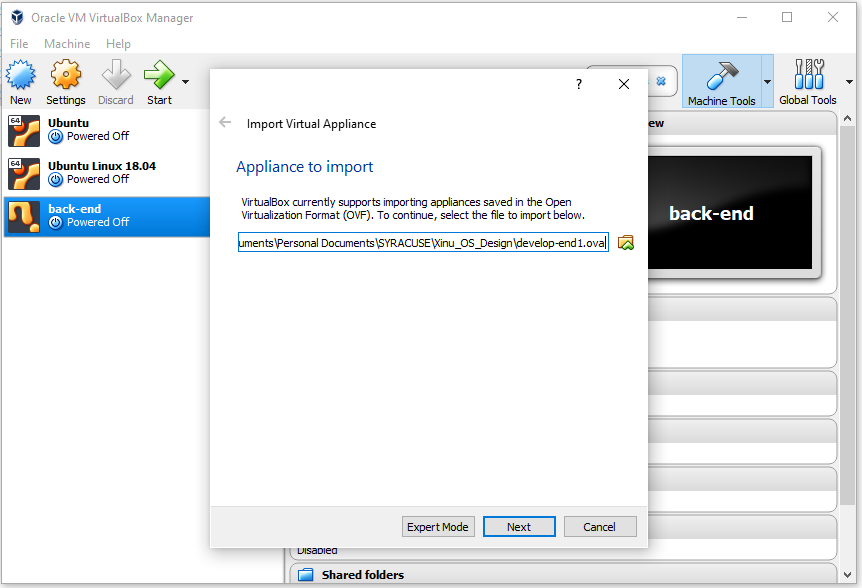
Open the VirtualBox application again and select File > Import Appliance. Click on the folder icon and select the file “back-end1.ova”. Click “Next”. Be sure to uncheck the “Reinitialize the MAC address of all network cards” checkbox. Click the “Import” button shown below.



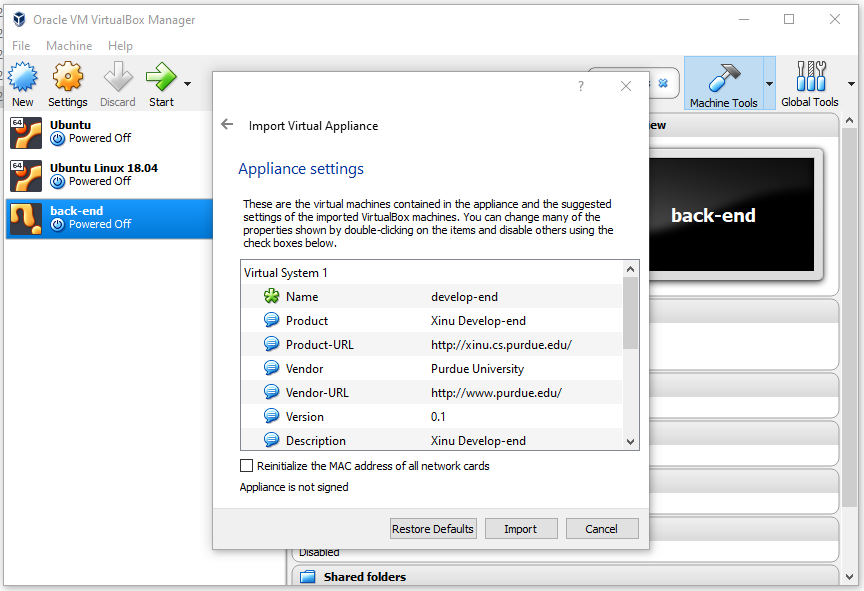
Select “back-end” from the main menu and click the “Settings” button. In the settings menu, select “Serial Ports”. Check the “Enable Serial Port” checkbox. The Port Number drop-down menu should have “COM1” selected, and the Port Mode drop-down menu should have “Host Pipe” selected. Check the “Connect to existing pipe/socket” checkbox. Edit the Path/Address to “\\.\pipe\xinu\_com1”. Click “OK”.



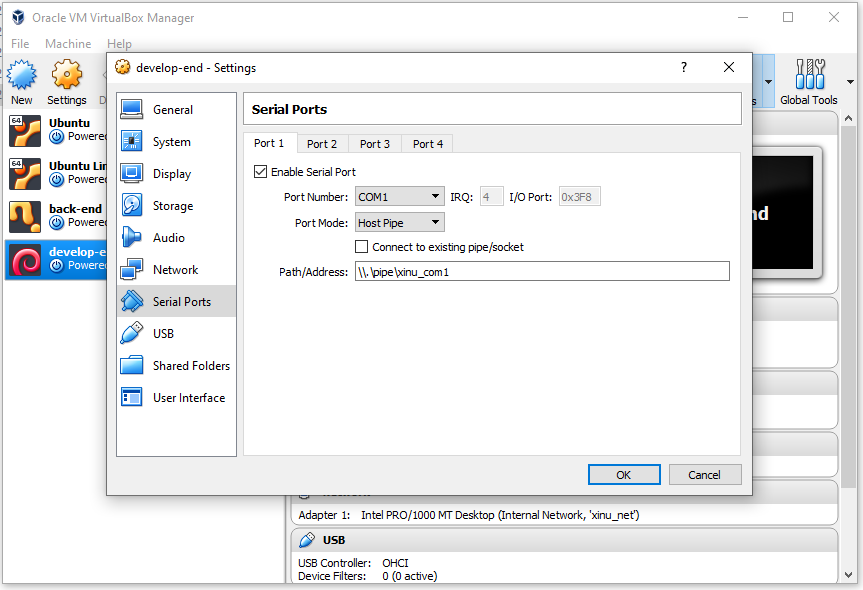
Next, import the develop-end1.ova file. Click File > Import Appliance. Click on the folder icon and select the file “develop-end1.ova”. Click “Next”.

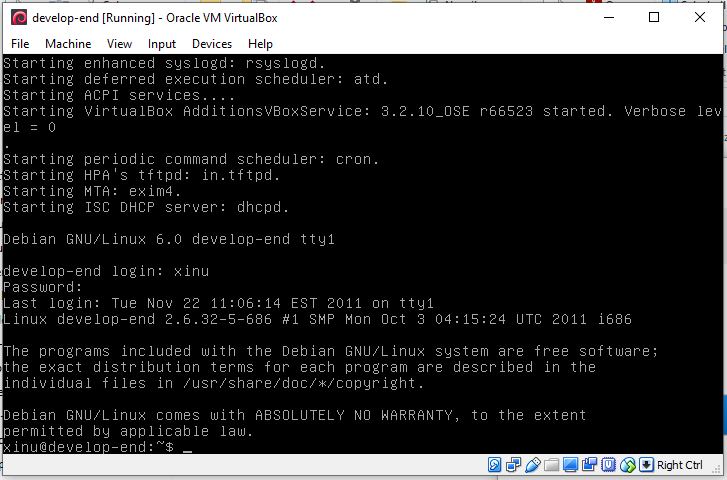


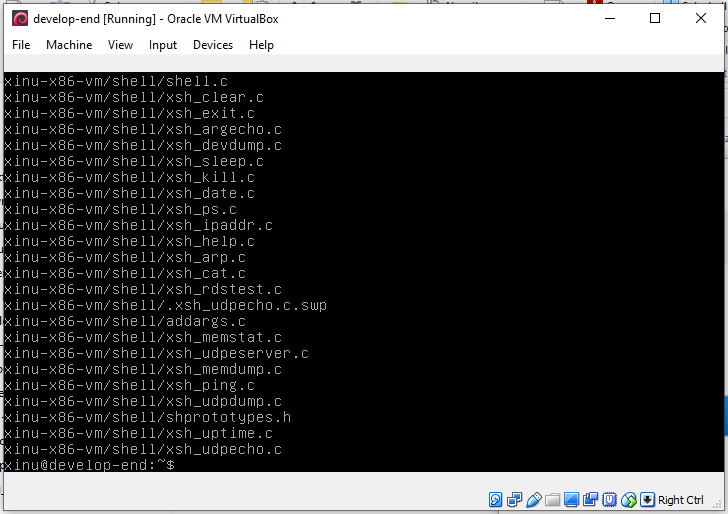
Be sure to uncheck the “Reinitialize the MAC address of all network cards” checkbox. Click the “Import” button shown below.



Select “develop-end” from the main menu, and click the “Settings” button. In the settings menu, select “Serial Ports”. Check the “Enable Serial Port” checkbox. The Port Number drop-down menu should have “COM1” selected, and the Port Mode drop-down menu should have “Host Pipe” selected. Do not check the “Connect to existing pipe/socket” checkbox. Edit the Path/Address to “\\.\pipe\xinu\_com1”. Click “OK”.



Start the develop-end and use the username “xinu” and the password “xinurocks”. The console should look like the screen below.

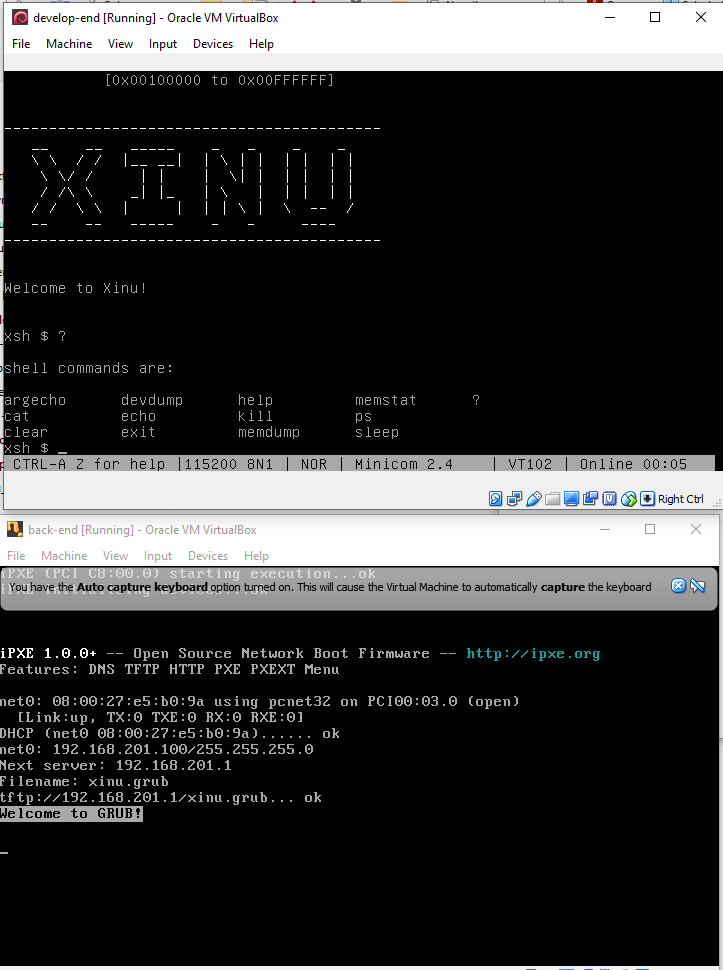
Next, navigate to the home directory and type “ls”. There should be two folders present, one of which is “xinu-x86-vm.tar.gz”. To untar this file, type “tar -zxvf xinu-x86-vm.tar.gz”.

Next, type “cd xinu-x86-vm”. Then “ls”. One of the folders listed should be “compile”. Type “cd compile”. The series of commands is shown below.

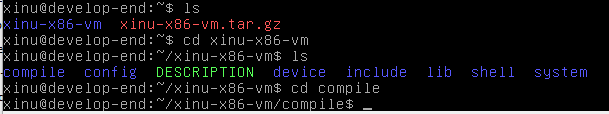


Next, type “ls” to view the files available in the compile folder.

Type “make clean;make;./upload.sh” followed by “sudo minicom”. If prompted for a password, use “xinurocks”. The screen should then display “Welcome to minicom 2.4”. Open up the back-end. The develop-end should display the image “XINU” followed by the greeting, “Welcome to Xinu!” Type the “?” character to see a list of available commands. The develop-end and back-end at this specific step are displayed on the next page.

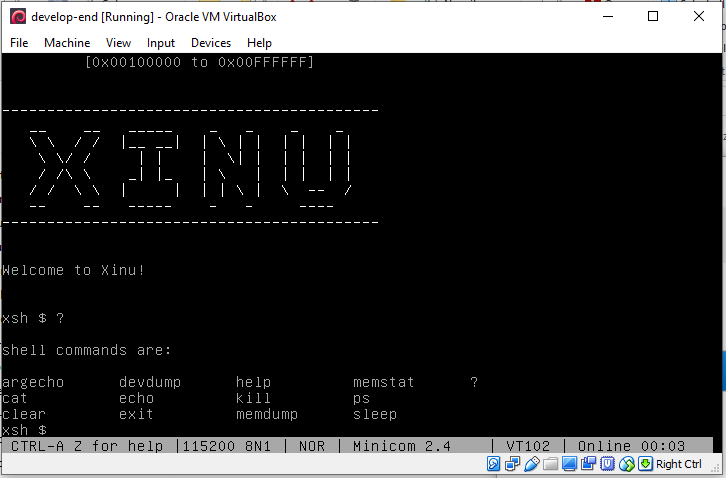


Section B: Edit and Type Your First Program

Close the develop-end and back-end. To run the compiled code again, simply navigate to the develop-end “/xinu-x86-vm/compile” folder as before and run “sudo minicom”. There is no need to recompile the project. See the commands below.

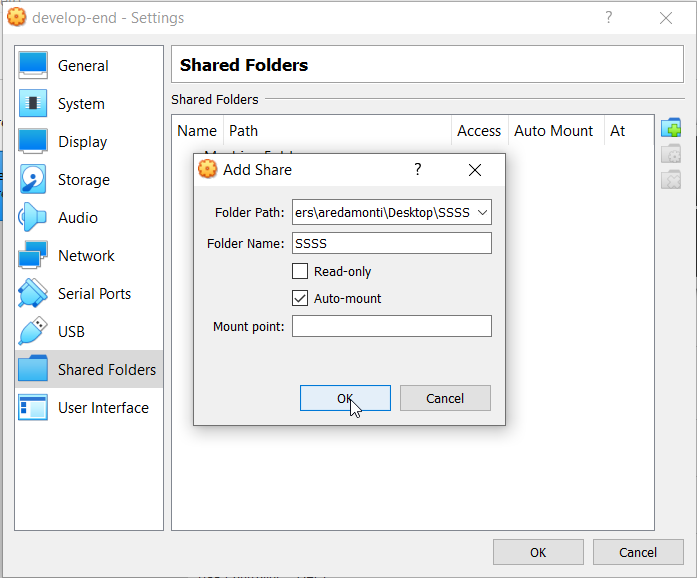


Then open the back-end. Type the “?” character for a list of available commands.



Close the develop-end and back-end. Create an empty folder on the desktop of the host OS (Windows). The name “SSSS” was arbitrarily chosen for the folder in this example. Insert a file into the folder. In VirtualBox, click on the develop-end and select Settings > Shared Folders > Add. Find and select the “SSSS” folder on the host desktop. Check the “Auto-mount” option checkbox.

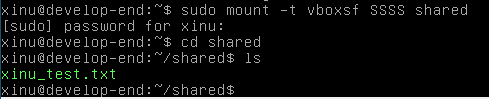
The screen below displays the correct settings for the shared folder as they were just described. Click “OK”.



Open the develop-end and type “mkdir shared” in the home directory. An “ls” command will show the newly created “shared” directory.

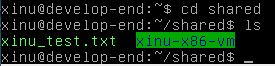


Reboot the develop-end. Before using the shared folder, be sure to mount it using the following command: “sudo mount -t vboxsf SSSS shared”. The “shared” folder will contain everything in the “SSSS” folder on the host desktop.



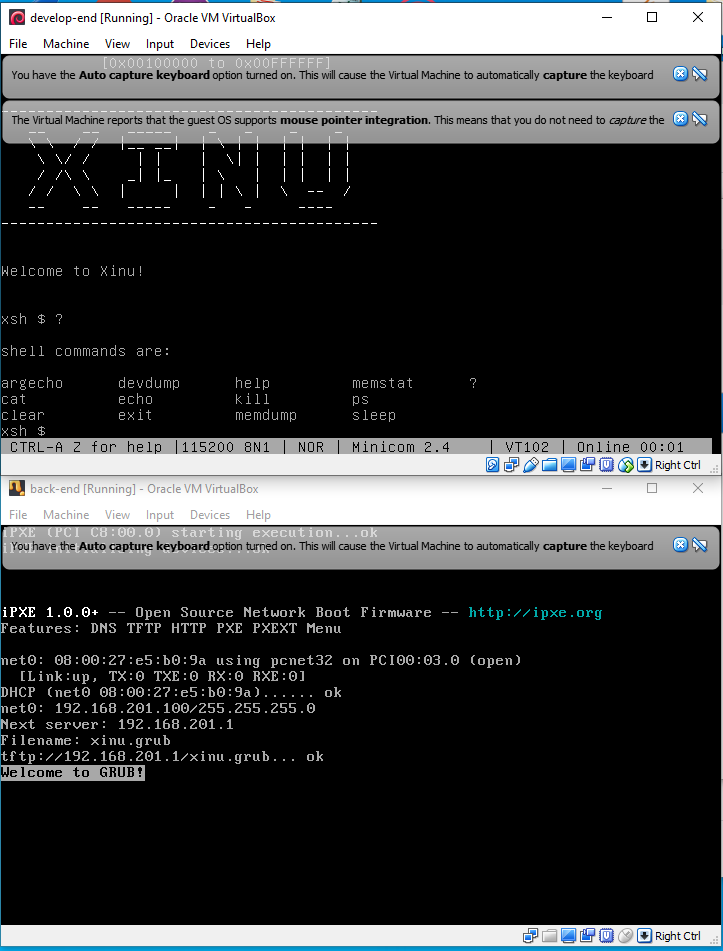
Next, copy the home directory into the shared folder. Navigate to the home directory and type “cp -R xinu-x86-vm shared”.



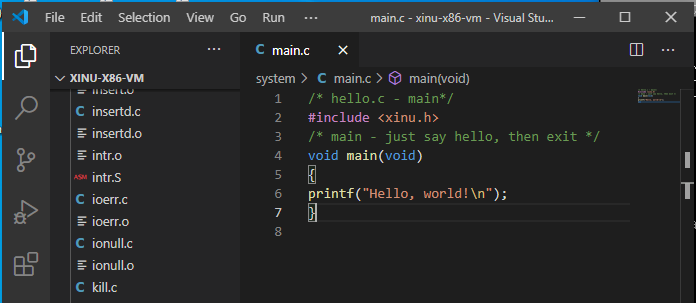


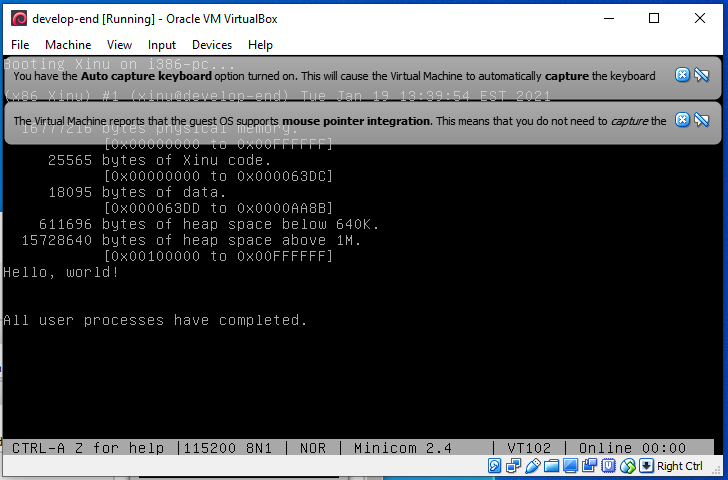
To compile in the shared folder, follow the same steps previously performed.

Starting from the beginning, first mount the “shared” folder using the command “sudo mount -t vboxsf SSSS shared”. Next, navigate to the “/shared/xinu-x86-vm/compile” folder. To compile, type “make clean;make;./upload.sh”. Then type “sudo minicom” and open the back-end. Type the “?” character in the develop-end to see a list of available commands. The develop-end and back-end screens should look like the image on the next page.



Section C: Run Your First Program

The main.c file was edited to simply print “Hello, world!” on the console. A screenshot of the program and output are below. A backup of the original “xinu-x86-vm” folder was created before any edits were made.



Section D: Type and Run A Second Program To Create Processes

Below is the program written to create two processes “sndA” and “sndB”. SndA willl continually print the character ‘A’, and sndB will continually print the character ‘B’. SndA and sndB have the same priority of 20.

/\* main.c - main\*/

#include <xinu.h>

void sndA(void), sndB(void);

/\* main - Example of creating processes in Xinu \*/

void main(void)

{

resume(create(sndA, 1024, 20, "p 1", 0));

resume(create(sndB, 1024, 20, "p 2", 0));

}

/\* sndA - Repeatedly emit 'A' on console \*/

void sndA(void)

{

while(1){

putc(CONSOLE, 'A');

}

}

/\* sndB - Repeatedly emit 'B' on console \*/

void sndB(void)

{

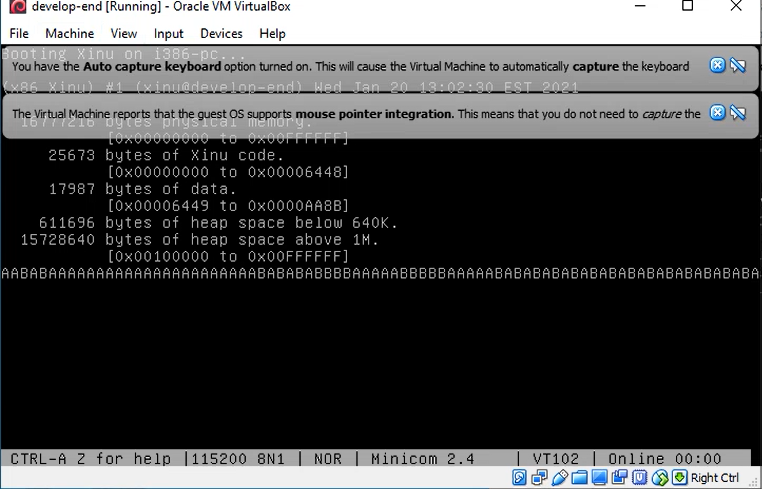
while(1){

putc(CONSOLE, 'B');

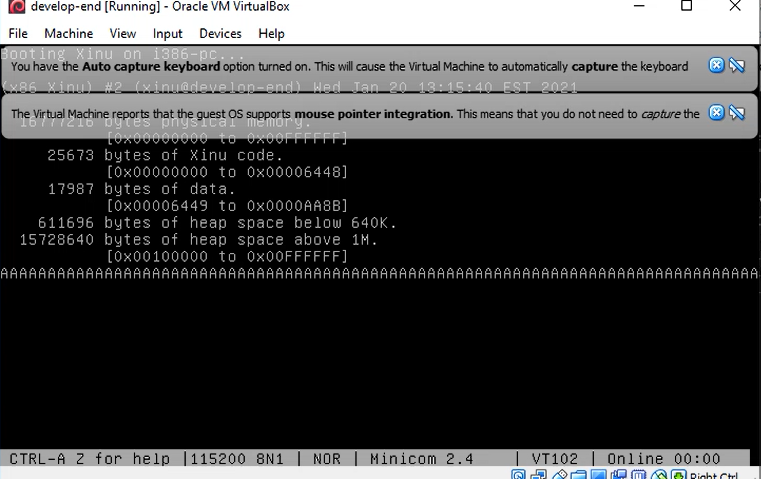
}

}

A snapshot of the output of the code is below. Note the random switching between the two processes. The pattern is unpredictable. The code is an example of two concurrent processes running the same code. There is context switching occurring between the two processes, and there are no guarantees about the order or rate of execution.



The priority of sndA was changed from 20 to 40. The output should be mostly ‘A’ characters since the priority of sndA is double that of sndB. A screenshot of the output can be found at the beginning of the next page. Indeed, the output is all ‘A’ characters. While it is still possible to have sndB be the running process, it is far less likely now that its priority is half of sndA.



The code is an example of CPU scheduling, which refers to how the OS decides how to choose which process to run from a list of ready processes located in the ready queue. There are various scheduling algorithms, but Xinu will run the highest priority process unless there are multiple highest priority processes. Xinu would use a round-robin approach for this case where each high priority process gets a Quantum time slice. To achieve round-robin scheduling, context switching is used. Context switching is when a process is stopped in the middle of execution, machine state is saved (registers, program status word, program counter, etc.) and the program counter jumps to another process, loads the new process (general purpose registers, program status word, etc.) and begins execution. Context switching gives the appearance to the user that multiple processes are executing simultaneously. On a single threaded uni-core processor, that is not possible, but because of the fast rate of context switching compared to human perception, it appears to execute multiple processes simultaneously.

Analysis of queue.c and main.c

The file “queue.c” begins by declaring the queuetab, which is an array of the qentry (queue entry) data type. Thus, the size of the array is equal to the number of queue entries (NQENT).

The definition of NQENT from queue.h is as follows: NQENT = (NPROC + 4 + NSEM + NSEM). NPROC represents the number of processes. Each process is allocated an entry slot in the queue table array. The number “4” represents two slots (head and tail) for the ready list and two slots (head and tail) for the sleep list. “NSEM + NSEM” (NSEM\*2) represents slots for the head and tail of each semaphore implemented in the OS.

Enqueue() takes two arguments: “pid” and “q”. The “pid” is the process ID of the process to insert into the queue, which is identified by the other argument “q” (the queue ID). The local variables “tail” and “prev” are declared and represent the tail and previous node indexes in the linked list (queue). The validity of the arguments is checked using the functions “isbadqid(q)” and “isbadpid(pid)”. If either argument is invalid, the enqueue function will return SYSERR. If the arguments are valid, the local variable “tail” is assigned as the queuetail(q). The queue ID is really the head of the queue, and the tail of the queue is the very next entry in the queuetab. Therefore, the queuetail is defined as q+1. The local variable “prev” is assigned to queuetab[tail].prev, which is the last queue entry before the tail. The linked list is then rearranged so that the new process “queuetab[pid]” will be inserted at the tail-end of the queue just before the tail. The function ends by returning the pid of the process that was successfully inserted into the queue.

Dequeue() takes one argument “q” which is the queue ID of the queue to perform dequeue(). If the queue ID is invalid, SYSERR is returned. Likewise, if the queue is empty, EMPTY is returned. If neither of those conditions is met then the function removes the first process from the queue at the beginning (head-end) of the queue, sets its qprev and qnext pointers to EMPTY, and returns the pid of the process.

The file “main.c” takes in command line arguments. Integer “argc” represents the number of arguments and character type “\*\*argv” is a double pointer to those arguments. Unsigned 32-bit integer “retval” (return value) is declared. The shell process is created and put onto the ready queue (resumed). The recvclr() function is called to clear out any message in the current process. A while-loop is entered where the process will wait for a message by calling “receive()”. The process will be blocked waiting on the message. The process state will be “PR\_RECV” and then it will call “resched()” to perform a context switch to another process. When the message is sent, the process will be placed back on the ready queue and eventually run. The kprintf function is called to print: “\n\n\rMain process recreating shell\n\n\r”. The “shell” process is then recreated using the “resume(create(shell, 4096, 1, ”shell”, 1, CONSOLE))” system call. Because “receive()” and “resume(create)” are in an infinite while-loop, the shell process will be continually created, blocking for a message, and calling kprintf.

Analysis of 2019 and 2020 Xinu Source Code

After viewing the two versions of the Xinu source code, the following differences were prevalent. After navigating to each version’s respective “system” folder, the 2020 version does not contain object files (.o extension), while the 2019 version does. Object files are typically created by the compiler using the source files (.c extension). The 2020 version also contains more source files (.c) such as clkhandler.c, exit.c, gettime.c, getticks.c, getutime.c, and a binary file clkdisp.S. The 2020 version also contains a folder that is not in the 2019 version called “net”. The net folder contains various source files dealing with communications over the Ethernet port such as udp.c (send/receive UDP packets) and ip.c (send/receive IP packets). Net.c manages the memory for these using the buffer-pool “netbufpool” (network buffer pool).

Reviewing the DESCRIPTION file located in each version of Xinu, there are differences between the two versions. First, the tty driver in the 2020 version uses port-mapped I/O in place of the memory-mapped I/O used on the Galileo hardware platform. Also, the 2020 version’s startup sequence (start.S) differs from the Galileo startup sequence, and the 2020 version removed the platform initialization used by the Galileo (platinit.c).