Project 1: Multithreaded Programming and Synchronization

COP 5614 – Operating Systems Principles

1. Summary

The first project is regarding several important topics on process management. But instead of developing it in kernel, we will do it in user space using a widely-used threads programming interface, POSIX Threads (Pthreads). You should implement this in Linux, which supports Pthreads as part of the GNU C library.

You should submit the required deliverable materials on Canvas by **11:55pm, September 22, 2019 (Sunday).**

2. Description

In this assignment, you will be working with the "threads"' subsystem of Linux. This is the part of Linux that supports multiple concurrent activities within the kernel. In the exercises below, you will write a simple program that creates multiple threads, you will demonstrate the problems that arise when multiple threads perform unsynchronized access to shared data, and you will rectify these problems by introducing synchronization (in the form of Pthreads mutex) into the code.

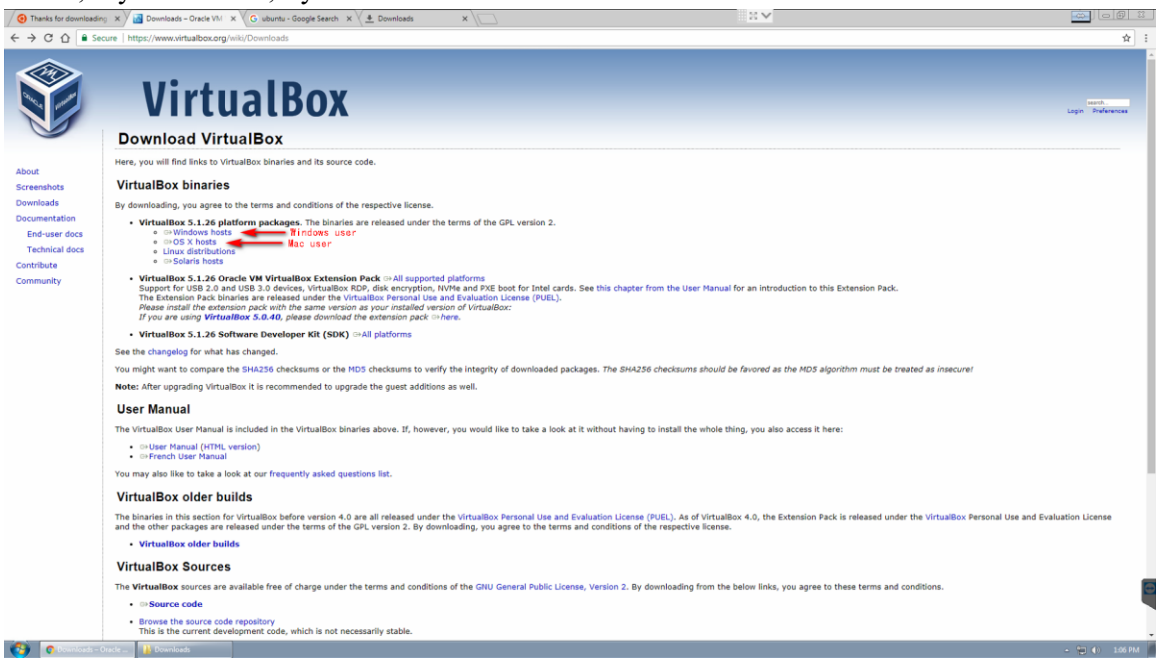
2.1 Environment Set Up

It's recommendable to use Ubuntu as operating system to accomplish this project. Ubuntu is an open source operating system software for computers. It is one of the distribution systems of Linux, and is based on the Debian architecture. For those who don't have Ubuntu installed in their computers, it's doable to use virtual machine and install the virtual Ubuntu in your Local operating system.

For virtual machine, it's recommendable to use VirtualBox:

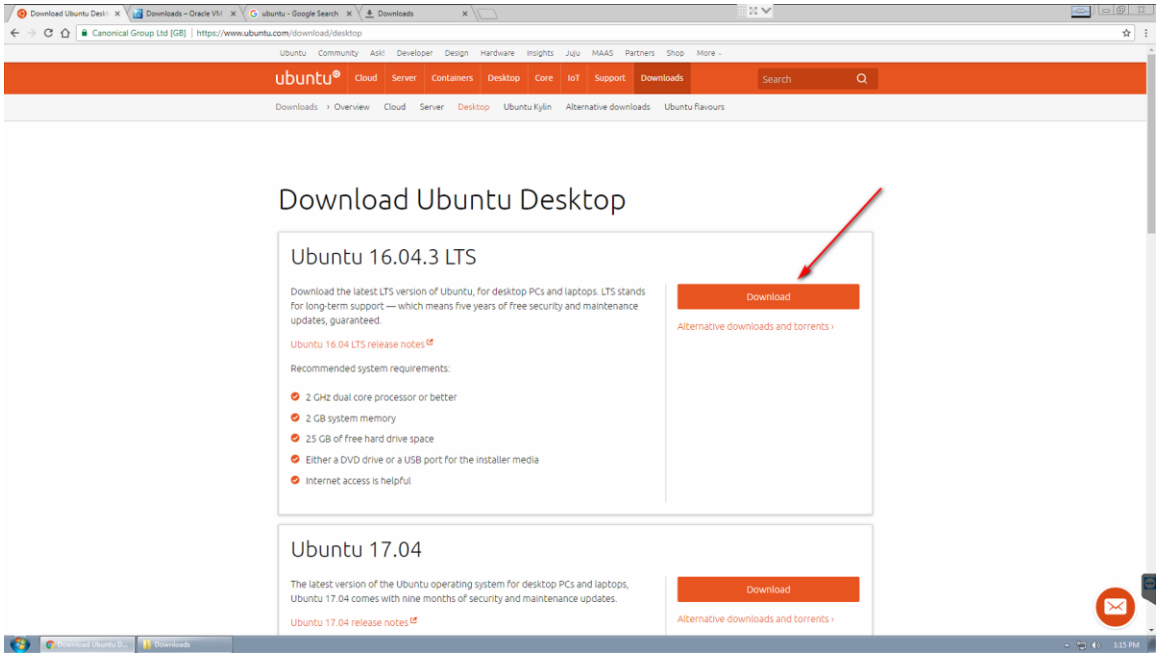
https://www.virtualbox.org/wiki/Downloads

If you have Windows operating system installed with your computer, try to download “Windows hosts”; if you use Mac, try to download “OS X hosts”.



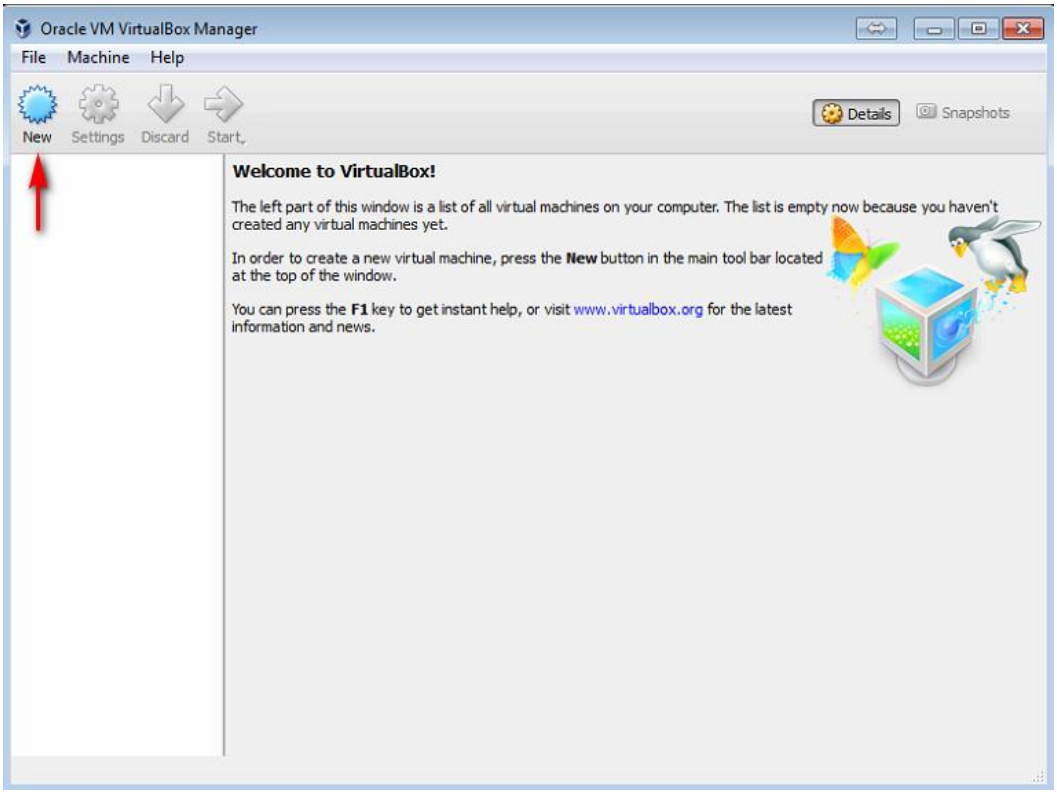
After you download and install VirtualBox in your computer, next step is to download the .iso file of Ubuntu from the official website:

https://www.ubuntu.com/download/desktop

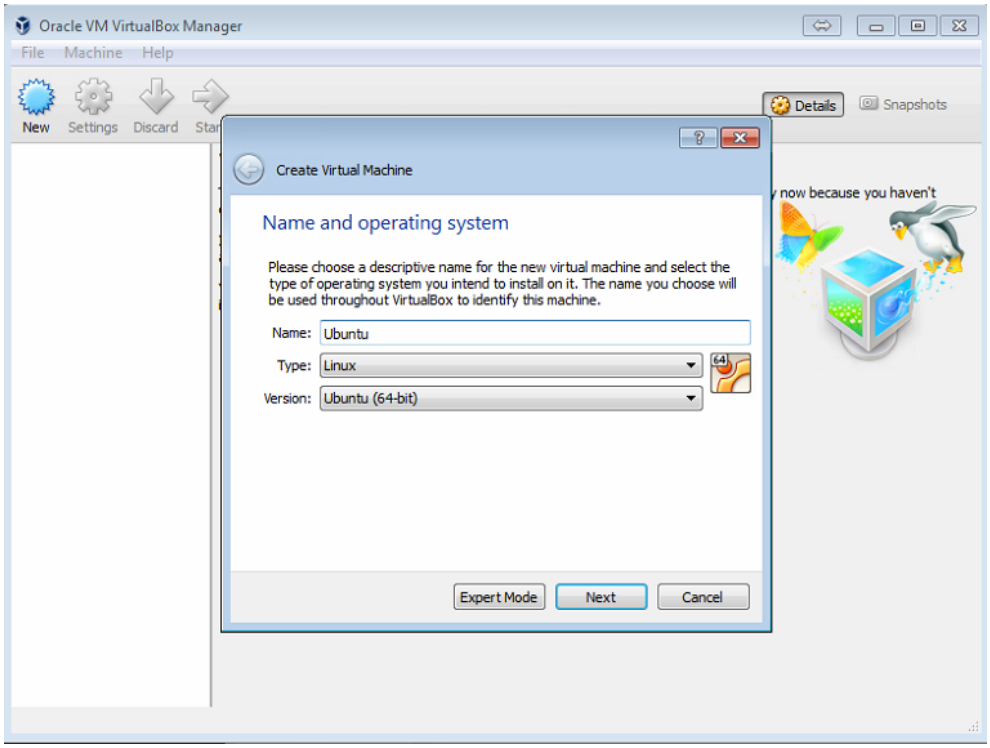


When you finish downloading the .iso file of Ubuntu, then it’s time to install a virtual Ubuntu in your virtual machine.

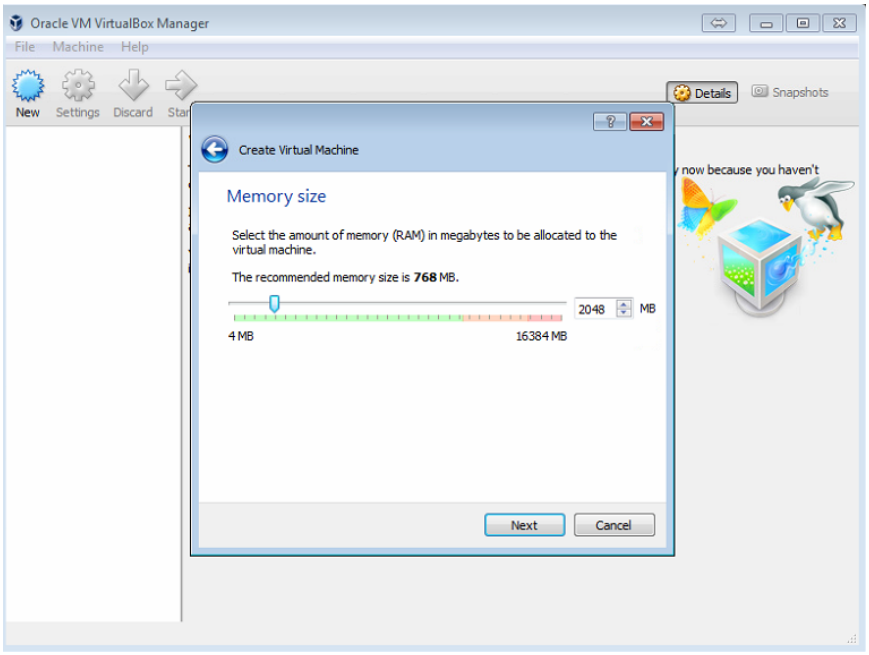
To do this, first you need to open VirtualBox, then click “New”:



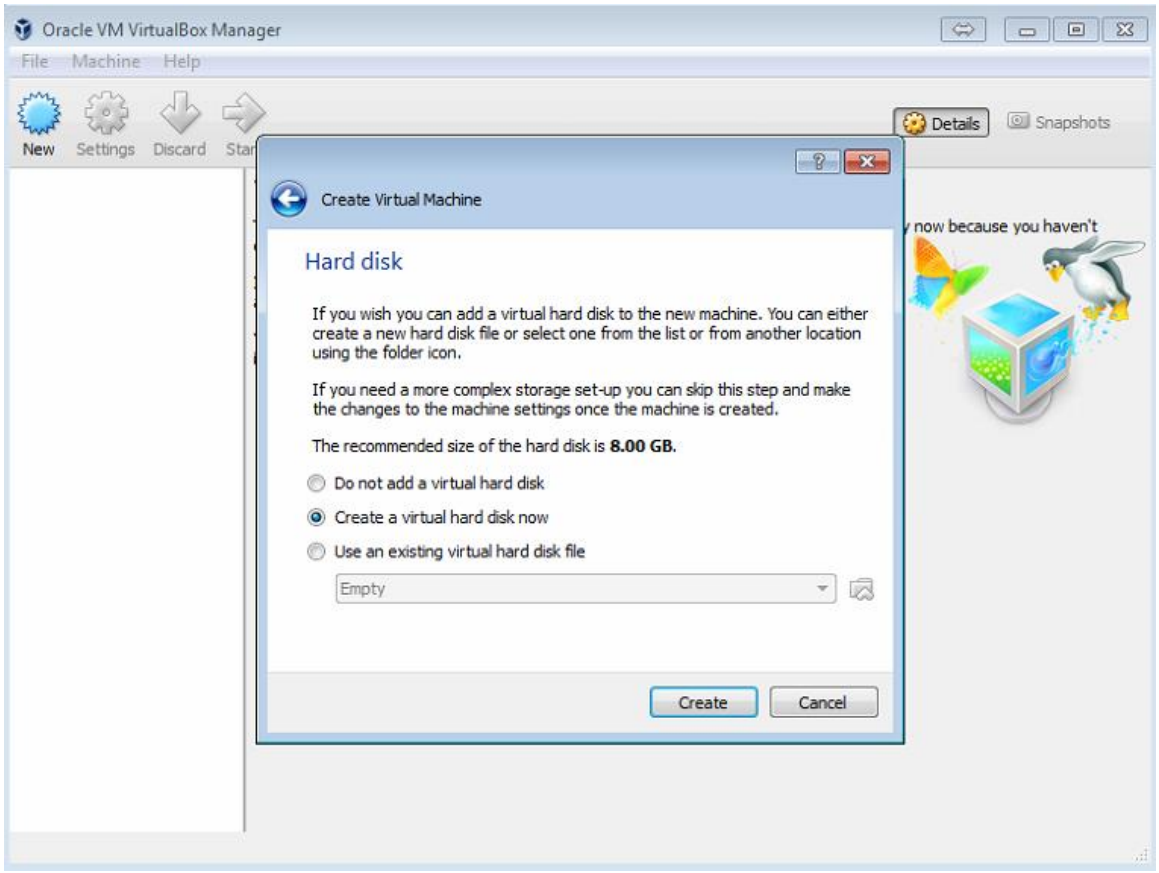
In the configuration window, you need to set a name for your virtual Ubuntu, select “Linux” for “Type”, and select “Ubuntu(64-bit)” for “Version”:



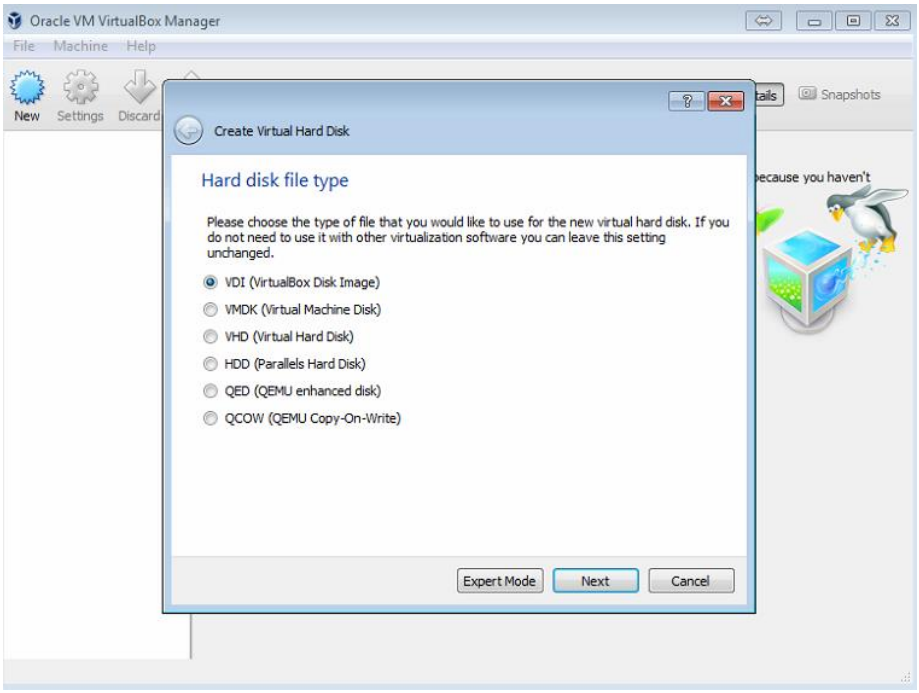
Select the amount of memory for your virtual Ubuntu. Here we recommend 2GB:



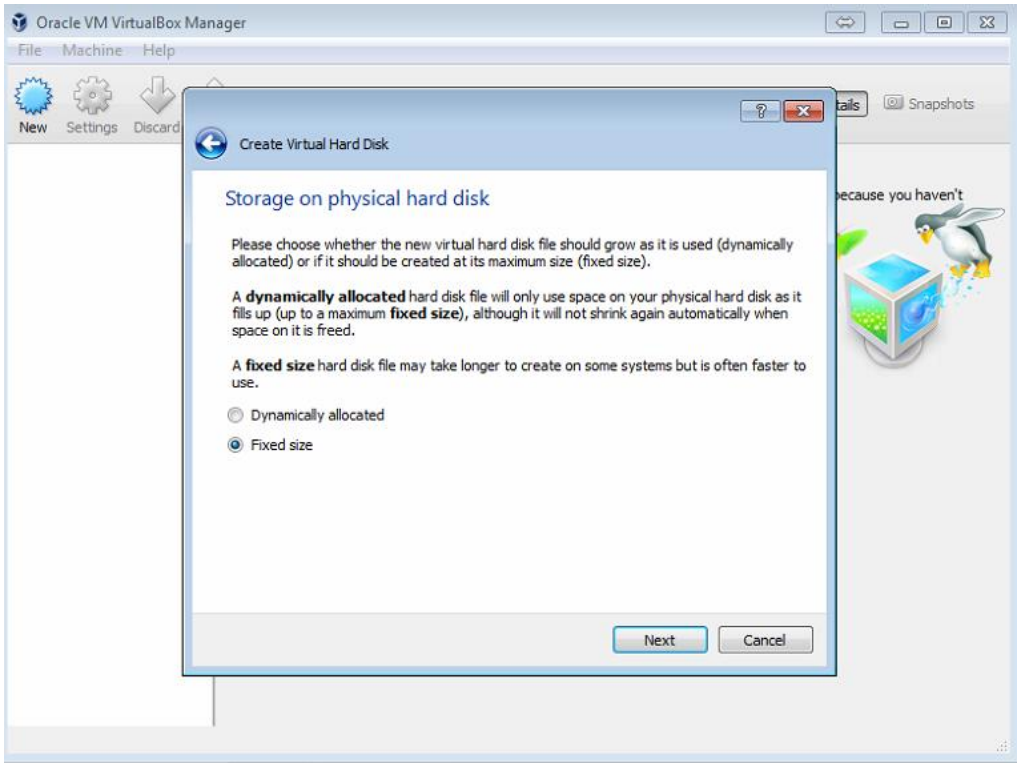
In this window, select “Create a virtual hard disk now”, then click “Create”:



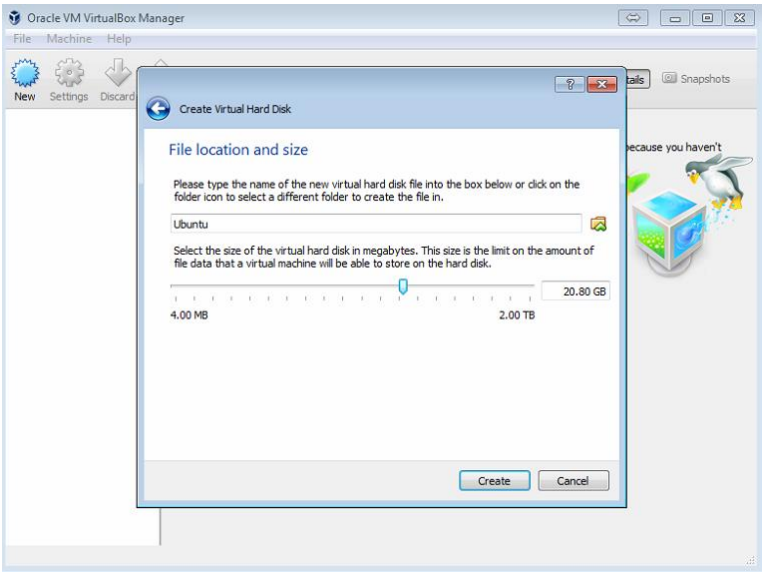
Click “Next” in this configuration window:



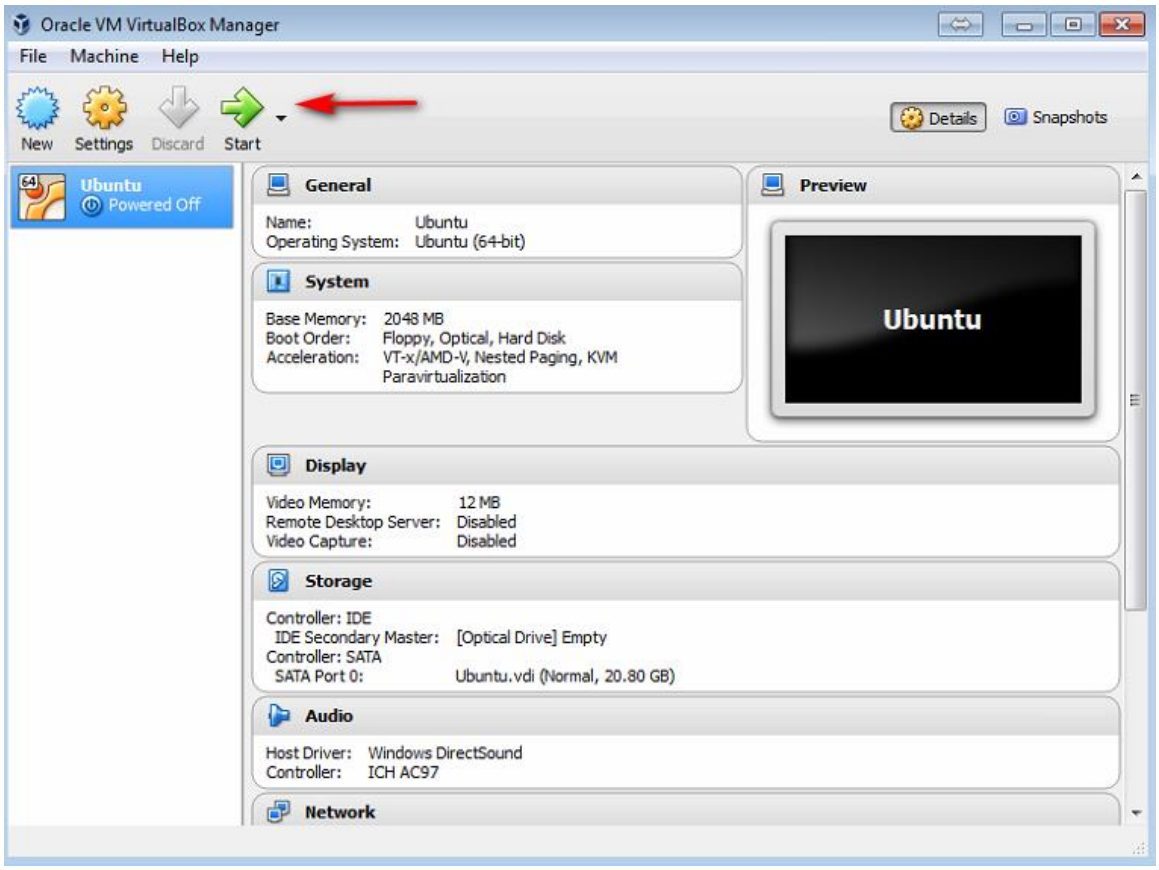
In this configuration window, you can select “Dynamically allocated”, which can make your allocated storage of the virtual Ubuntu grows if it fills up (up to a size that you will configure in the next step); or you can select “Fixed size” and then configure a size of storage for your virtual Ubuntu:



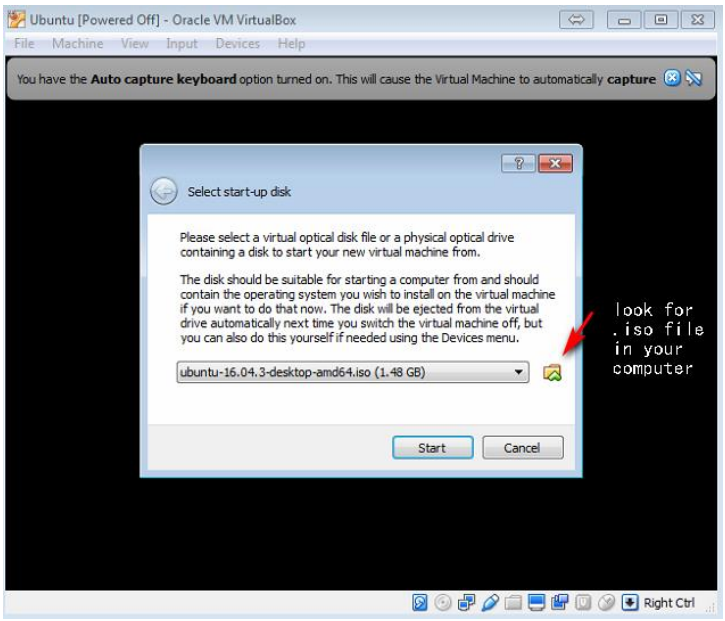
Set a size for your virtual Ubuntu. Here we set 20GB, then “Create”:



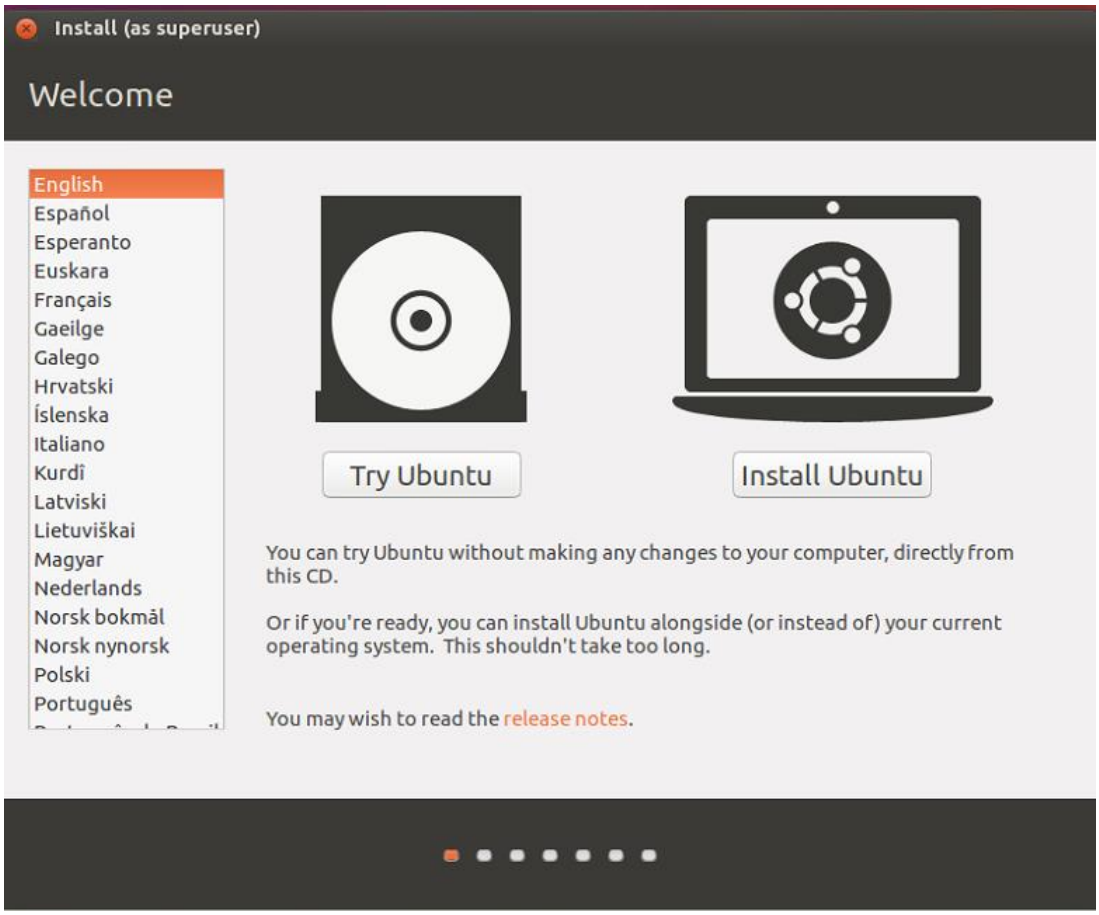
At this point, we have finished the configuration. Now you can see your virtual Ubuntu has been created in the left menu. Next you need to click “Start”:



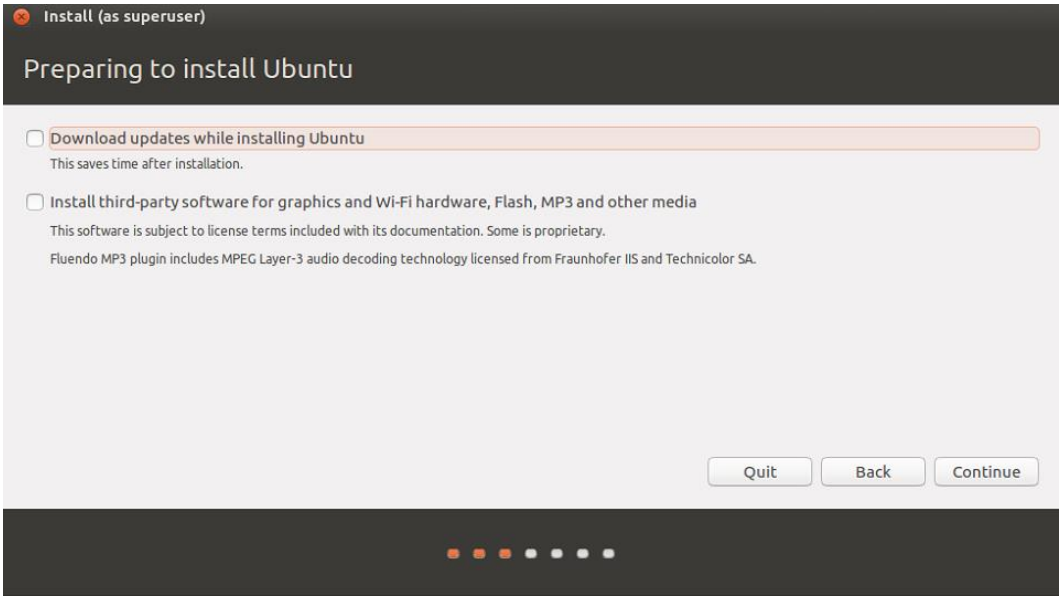
After starting, VirtualBox will ask you to “select start-up disk”, which means you need to select the .iso file of Ubuntu that you just download, then click “Start”:



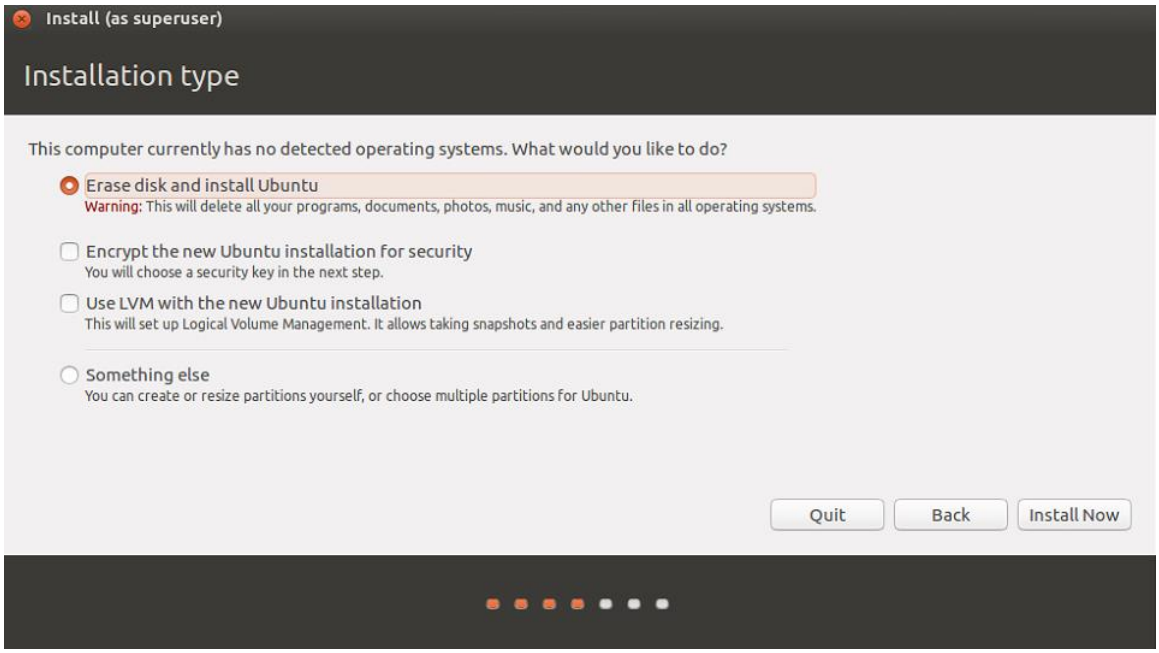
Now it’s time to install Ubuntu in VirtualBox. Click “Install Ubuntu”:

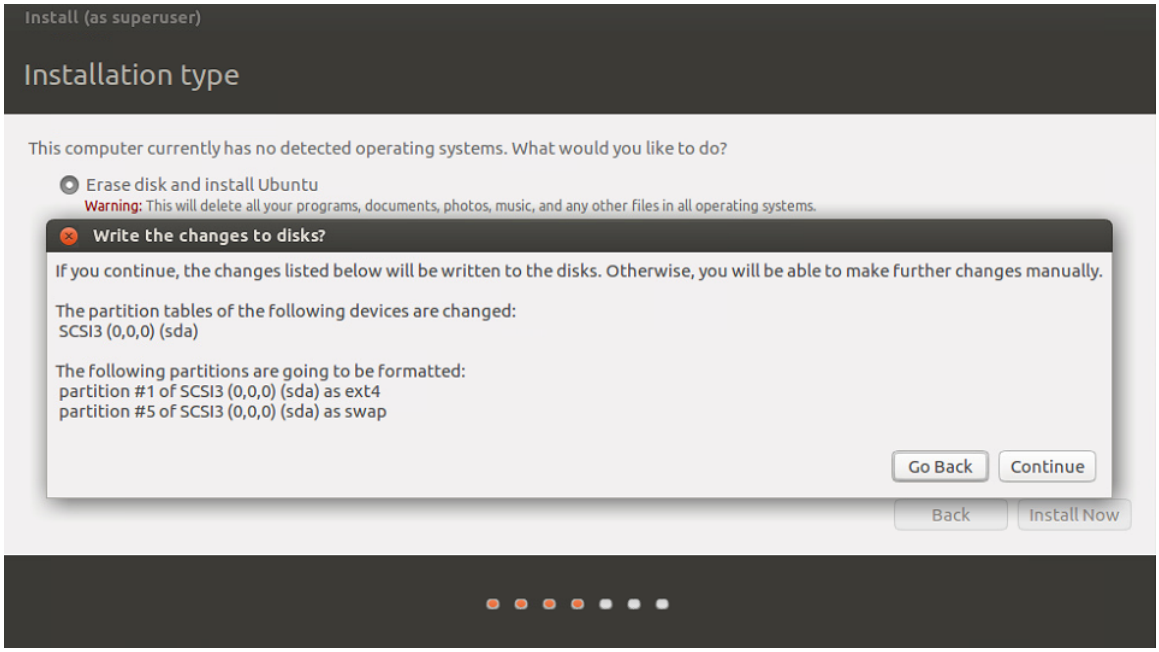


Click “Continue”:

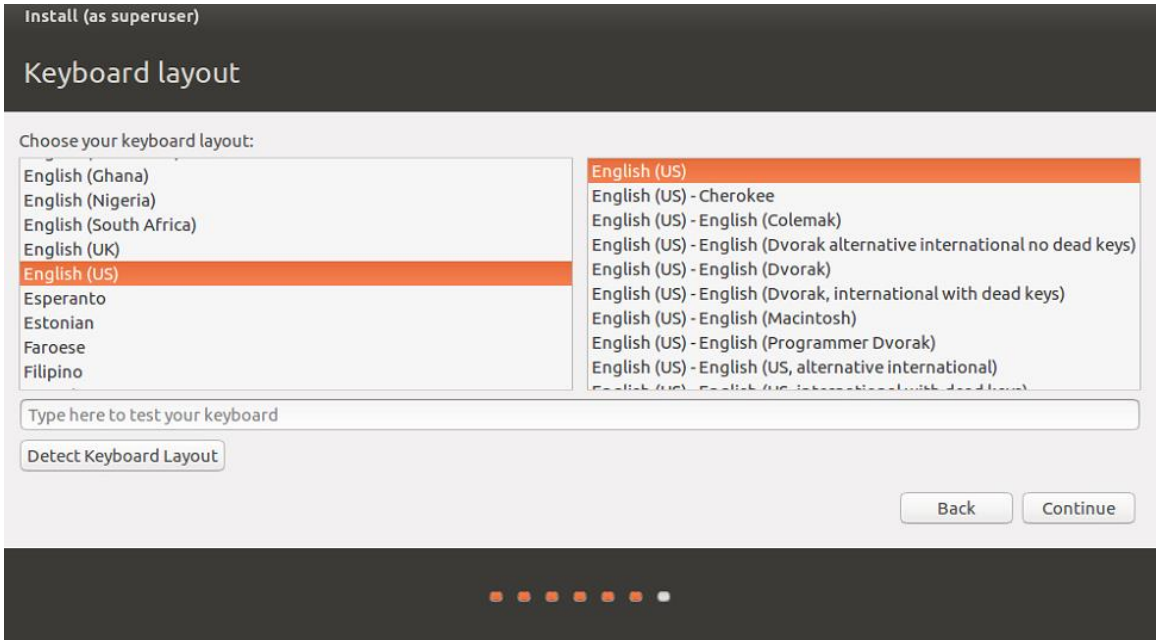


Click “Install Now” and then “Continue”:

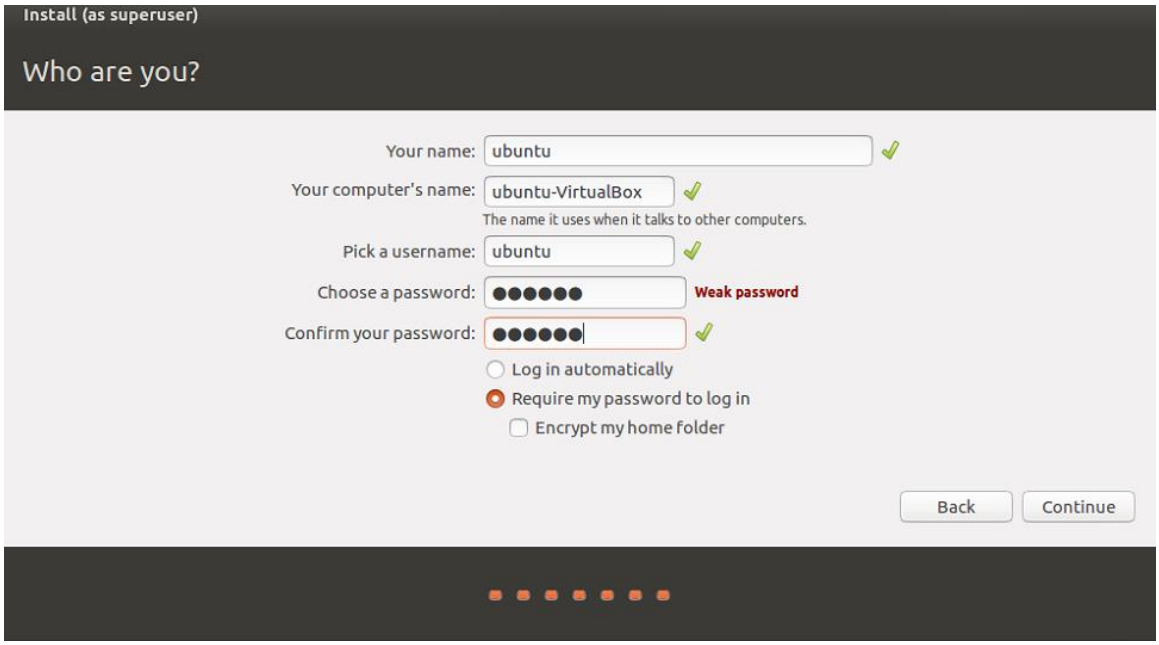




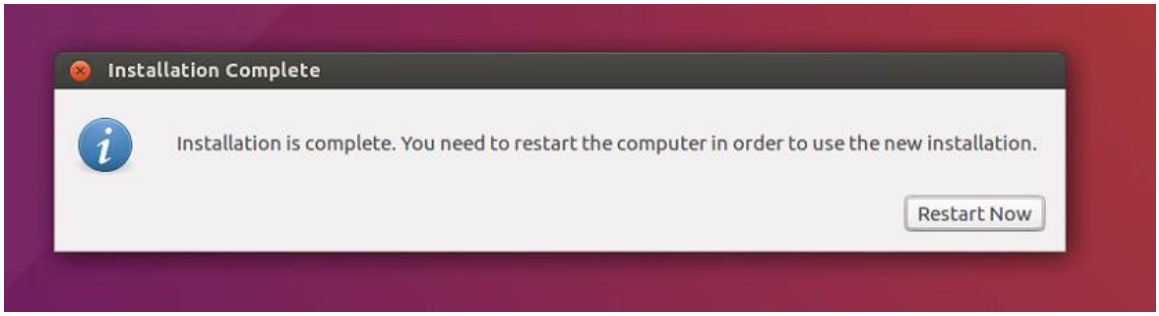
Click “Continue” for both of them:



Set up your name, computer's name, username and password for your virtual Ubuntu, then "Continue":



After finishing installation, click "Restart Now":



If you get a message “Please remove the medium and press ENTER”, just simply do so or shut down your virtual machine and start it again.

Now you get your virtual Ubuntu installed, next you can continue working on this project with it.

2.2 Simple Multi-Thread Programming

The purpose of this exercise is for you to get some experience using the threads primitives provided by Pthreads [1], and to demonstrate what happens if concurrently executing threads access shared variables without proper synchronization. Then you will use the mutex synchronization primitives in Pthreads to achieve proper synchronization.

Step 1: Simple Multi-Thread Programming without Synchronization

First, you need to write a program using the Pthreads library that forks a few threads each executes the loop in the SimpleThread function below. The number of threads is a command line parameter of your program. All the threads modify a shared variable SharedVariable and display its value within and after the loop.

```
int SharedVariable = 0;

void SimpleThread(int which) {
    int num, val;

    for(num = 0; num < 20; num++) {
        if (random() > RAND_MAX / 2)
            usleep(500);

        val = SharedVariable;
        printf("*** thread %d sees value %d\n", which, val);
        SharedVariable = val + 1;
    }
    val = SharedVariable;
    printf("Thread %d sees final value %d\n", which, val);
}
```

Your program must validate the command line parameter to make sure that it is a number, not an arbitrary string.

Your program must be able to run properly with any reasonable number of threads (e.g., 200).

Try your program with the command line parameter set to 1, 2, 3, 4, and 5. Analyze and explain the results. Put your explanation in your project report.

Step 2: Simple Multi-Thread Programming with Proper Synchronization

Modify your program by introducing Pthreads mutex variables, so that accesses to the shared variable are properly synchronized. Try your synchronized version with the command line parameter set to 1, 2, 3, 4, and 5. Accesses to the shared variables are properly synchronized if (i) **each iteration of the loop in SimpleThread() increments the variable by exactly one** and (ii) **each thread sees the same final value**. It is necessary to use a Pthreads barrier [2] in order to allow all threads to wait for the last to exit the loop.

You must surround all your synchronization-related changes with preprocessor commands, so that we can easily compile and get the version of your program developed in Step 1. E.g.,

```
for(num = 0; num < 20; num++) {
#ifdef PTHREAD _SYNC
      /* put your synchronization-related code here */
#endif
    val = SharedVariable;
    printf("*** thread %d sees value %d\n", which, val);
    SharedVariable = val + 1;
    .......
}
```

One acceptable output of your program is (assuming 4 threads):

*** thread 0 sees value 0
*** thread 0 sees value 1
*** thread 0 sees value 2
*** thread 0 sees value 3
*** thread 0 sees value 4
*** thread 1 sees value 5
*** thread 1 sees value 6
*** thread 1 sees value 7
*** thread 1 sees value 8
*** thread 1 sees value 9
*** thread 2 sees value 10
*** thread 2 sees value 11
*** thread 2 sees value 12

*** thread 3 sees value 13
*** thread 3 sees value 14
*** thread 3 sees value 15
*** thread 3 sees value 16
*** thread 3 sees value 17
*** thread 2 sees value 18
*** thread 2 sees value 19

……

Thread 0 sees final value 80
Thread 2 sees final value 80
Thread 1 sees final value 80
Thread 3 sees final value 80

Step 3: The Required Deliverable Materials

(1) A README file, which describes how we can compile and run your code. (10%)
(2) Your source code, must include a Makefile and be submitted in the required format. (45%)
(3) Your report, which discusses the output of your program without Pthreads synchronization and the one with Pthreads synchronization (must have screenshot of the outputs with your identification info (the current time or your computer ID)), as well as the reason for the difference; includes your processing of design and why to implement in this way. (45%)

3. Submission Requirements

You need to strictly follow the instructions listed below:

1) Submit a .zip file that contains all files.

2) The submission should include only your source code and project report. Do not submit your binary code.

3) Your code must **be able to compile**; otherwise, you will receive a grade of zero.

4) Your code should not produce anything else other than the required information in the output file.

5) Your code must validate command line parameters to make sure that only numbers can be accepted.

6) If you code is partially completed, also explain in the report what has been completed and the status of the missing parts.

7) Provide **sufficient comments** in your code to help the TA understand your code. This is important for you to get at least partial credit in case your submitted code does not work properly.

4. Policies

1) Late submissions will be graded based on our policy discussed in the course syllabus. 2) You must work in a group of two people on this exercise. We encourage high-level discussions among the groups to help each other understand the concepts and principles. However, code-level discussion is **prohibited**. We will use anti-plagiarism tools to detect violations of this policy.

5. Resources

The Pthreads tutorials at https://computing.llnl.gov/tutorials/pthreads and

http://pages.cs.wisc.edu/~travitch/pthreads_primer.html are good references to learn Pthreads programming.

6. References

[1] POSIX Threads Programming: https://computing.llnl.gov/tutorials/pthreads/

[2] Pthreads Primer: http://pages.cs.wisc.edu/~travitch/pthreads_primer.html

[3] POSIX thread (pthread) libraries:
http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html