COSC 439

Operating Systems

Spring 2020

**Lab Exercise #6**

*What I mean is that if you really want to understand something, the best way is to try and explain it to someone else. That forces you to sort it out in your own mind. And the more slow and dim-witted your pupil, the more you have to break things down into more and more simple ideas. And that's really the essence of programming. By the time you've sorted out a complicated idea into little steps that even a stupid machine can deal with, you've certainly learned something about it yourself.*

~ Douglas Adams

**Name: / 10**

**Goals:** The intention of this lab is to better understand synchronization and the use of semaphores in the Linux pthread library.

**Environment:** The GNU GCC C compiler on your Ubuntu virtual machine.

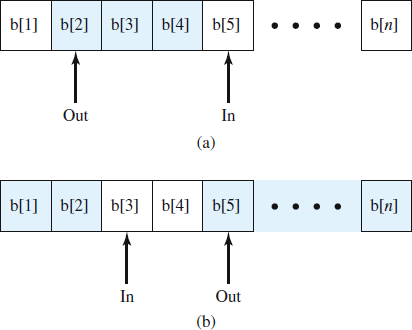
**Slack Channel:** All/any questions, problems and/or announcements for this lab should be directed to the course’s #lab6 Slack channel.

**Submission:** All files, including this document with the required screenshots, should be submitted via Slack/GitLab as indicated. ***Lab submissions not following this convention may not be graded.***

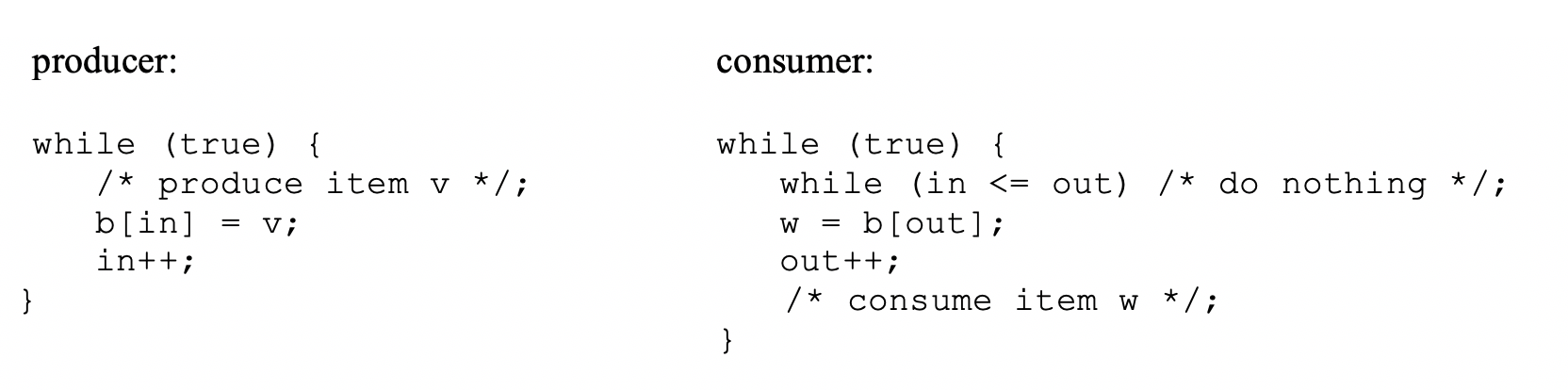
**Deadline:** Submitted via GitLab by 11:59pm on [Monday, March 23](https://www.youtube.com/watch?v=PFzns8Go9KI), 2020 with all source code, developed files and this document with screenshots checked into your GitLab repository. You should save the C programs to the same Lab 4/5 directory and repository as the previous labs.

**The Producer/Consumer Problem.**  (3 points) The Producer/Consumer problem is synchronization problem where there are one or more producers generating some type of data (e.g., records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system needs to be constrained to prevent the overlap of buffer operations. That is, only one agent (i.e., producer or consumer) may access the buffer at a time. The problem is to make sure that the producer won’t try to add data into the buffer if it’s full and that the consumer won’t try to remove data from an empty buffer.

To begin, let us assume that the buffer is infinite and consists of a linear array of elements. In abstract terms, we can define the producer and consumer functions shown in Figures 1 & 2.

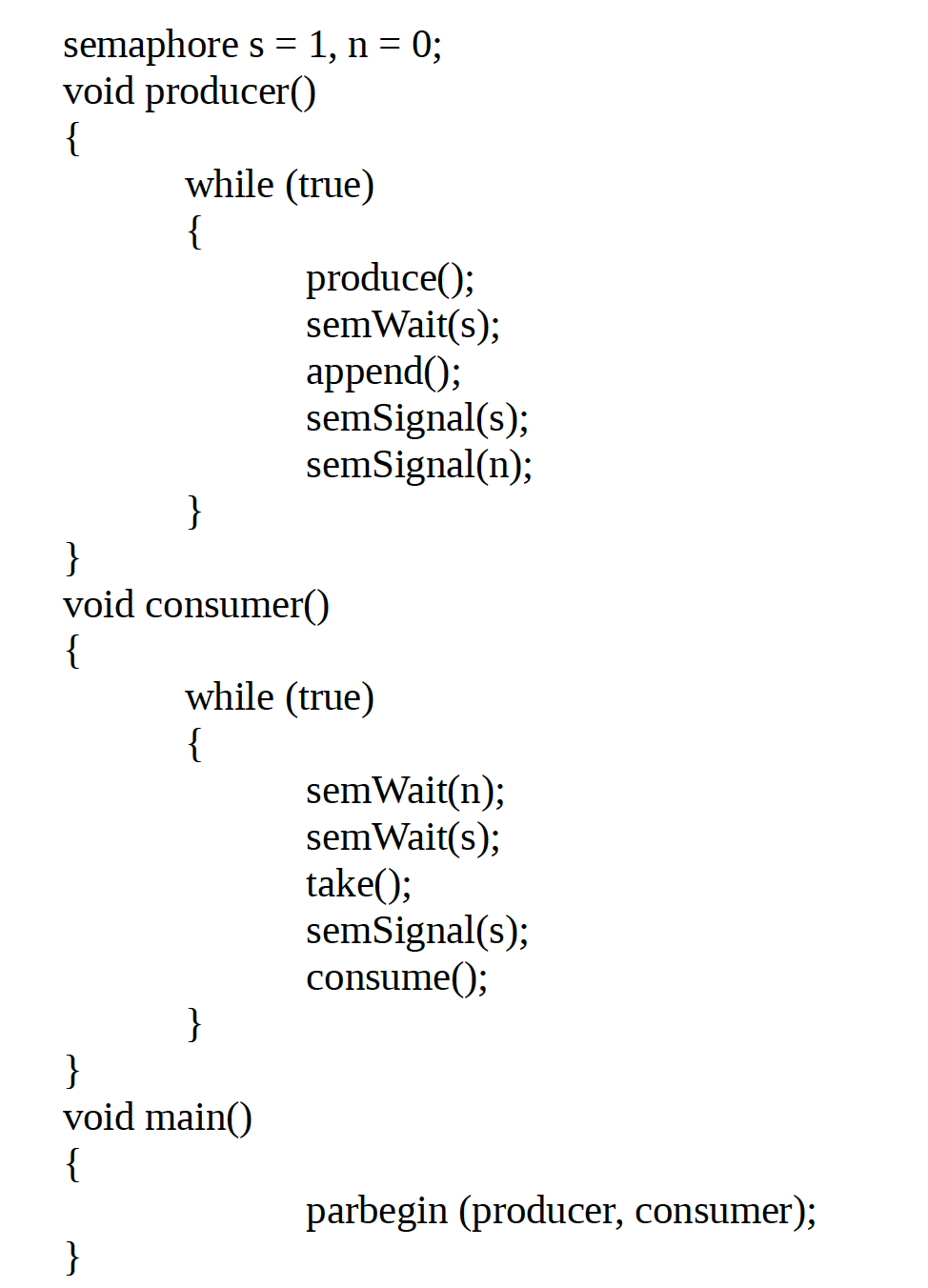
****

## **Figure 1.** Solution to bounded/circular buffer producer-consumer problem



## **Figure 2.** Solution to bounded/circular buffer producer-consumer problem

More generally, we can solve the producer/consumer problem for unbounded buffers using semaphores, as shown in Figure 3. Note that this is pseudocode, not C syntax.



## **Figure 3.** Solution to unbounded/circular buffer producer-consumer problem using semaphores

*Semaphores* are a special type of variable containing an integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are [atomic](https://www.youtube.com/watch?v=YuKMCtna19E): initialize, decrement, and increment. The decrement operation may result in the blocking of a process/thread, and the increment operation may result in the unblocking of a process/thread, also known as a counting semaphore or a general semaphore.

A semaphore may be initialized to a nonnegative integer value. The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution. The semSignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

A generalized use of a semaphore for enforcing mutual exclusion of a critical code selection is shown below and illustrated in Figure 4.

semaphore sv = 1;

loop forever

{

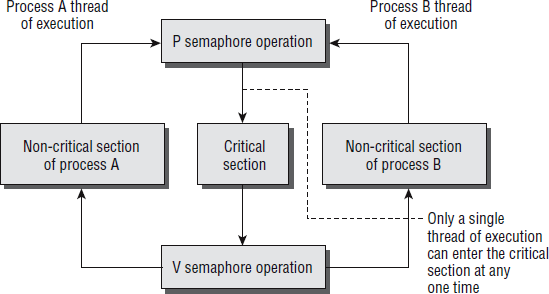
wait(sv);

// critical code section;

signal(sv);

// noncritical code section;

}



## **Figure 4.** Use of semaphores in enforcing mutual exclusion of critical code

The implementation of a binary semaphore in the Linux pthread library is as follows. A semaphore is created with the sem\_init function, which is declared as follows:

## #include <semaphore.h>

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

This function initializes a semaphore object pointed to by *sem*, sets its sharing option and gives it an initial integer value. The *pshared* parameter controls the type of semaphore. If the value of *pshared* is 0, the semaphore is local to the current process. Otherwise, the semaphore may be shared between processes. Here we are interested only in semaphores that are not shared between processes. At the time of writing, Linux doesn’t support this sharing, and passing a nonzero value for *pshared* will cause the call to fail.

The next pair of functions controls the value of the semaphore and is declared as follows:

## #include <semaphore.h>

int sem\_wait(sem\_t \* sem);

int sem\_post(sem\_t \* sem);

These both take a pointer to the semaphore object initialized by a call to sem\_init. The sem\_post function atomically increases the value of the semaphore by 1. Atomically here means that if two threads simultaneously try to increase the value of a single semaphore by 1, they do not interfere with each other, as might happen if two programs read, increment, and write a value to a file at the same time. If both programs try to increase the value by 1, the semaphore will always be correctly increased in value by 2.

The sem\_wait function atomically decreases the value of the semaphore by one, but always waits until the semaphore has a nonzero count first. Thus, if you call sem\_wait on a semaphore with a value of 2, the thread will continue executing but the semaphore will be decreased to 1. If sem\_wait is called on a semaphore with a value of 0, the function will wait until some other thread has incremented the value so that it is no longer 0. If two threads are both waiting in sem\_wait for the same semaphore to become nonzero and it is incremented once by a third process, only one of the two waiting processes will get to decrement the semaphore and continue; the other will remain waiting. This atomic “test and set” ability in a single function is what makes semaphores so valuable.

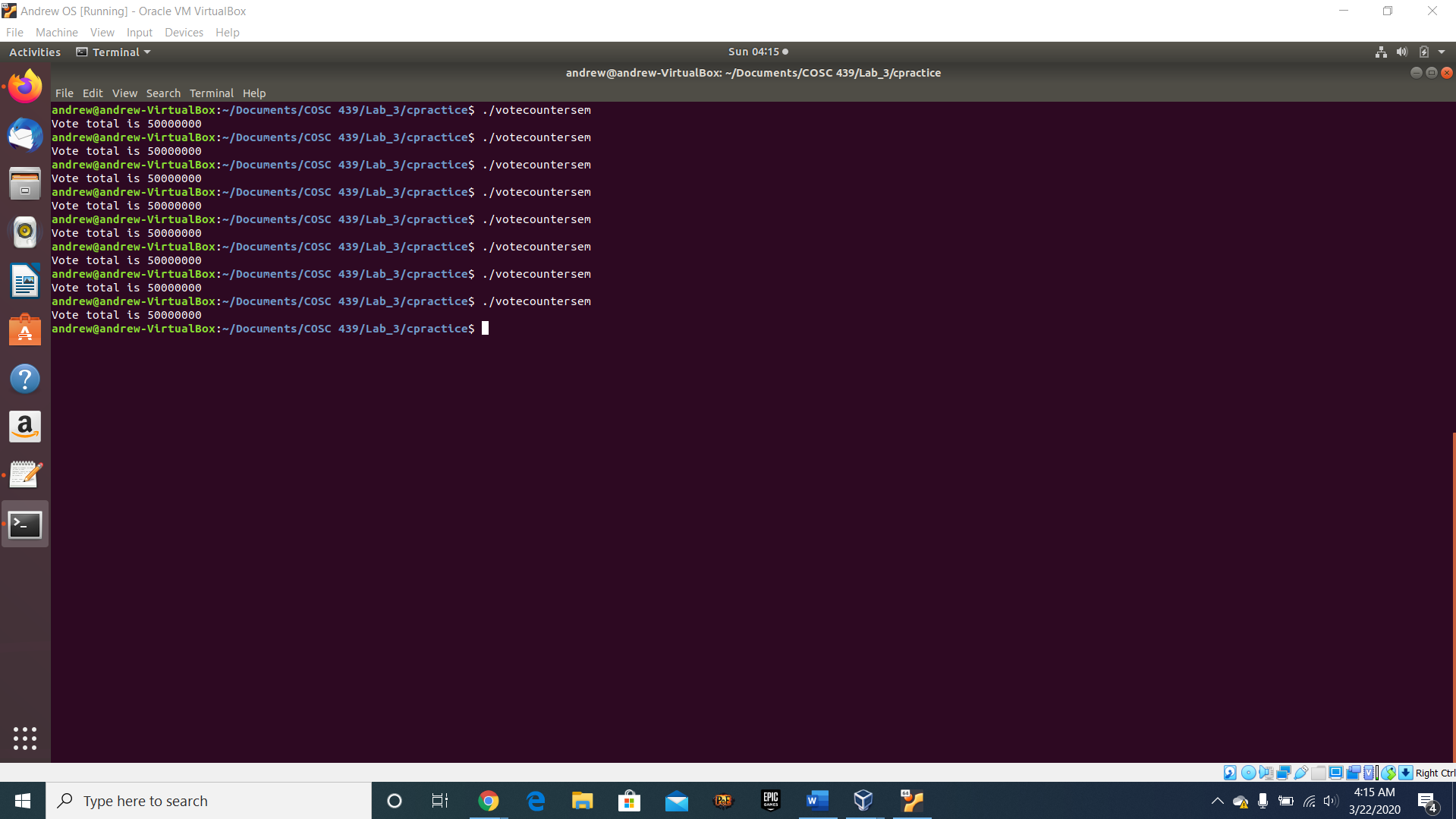
The last semaphore function is sem\_destroy. This function tidies up the semaphore when we have finished with it. It is declared as follows:

## #include <semaphore.h>

int sem\_destroy(sem\_t \* sem);

Again, this function takes a pointer to a semaphore and tidies up any resources that it may have. If we attempt to destroy a semaphore for which some thread is waiting, we will get an error. Like most Linux functions, these functions all return 0 on success.

With this, ***rename your votecounter.c solution program from Lab 5 to votecountersem.c and modify it to ensure a correct vote output by utilizing semaphores, rather than mutex locks.*** *Once you have it working,* ***include a screenshot of several runs of your output here and include the votecountersem.c program in your cpractice directory.***



**The** [**Pizza**](https://www.youtube.com/watch?v=J5wjBXjI3ds) **Studying Problem.** (7 points) ***Develop a C program, named pizzastudy.c, that solves the following scenario using either mutex or semaphores:***

Exam I is coming and you should start studying soon! To do so, you and a group of classmates get together to study, but (of course) you can only study while eating Seasons pizza. All students in your group can only study when eating pizza (slices are not shared between students). Thus, the basic algorithm that a student follows is:

while(true){

pick up a slice of pizza

study while eating pizza slice

}

If someone (you or of your study partners) finds that there are no more slices of pizza , they take a [nap](https://www.youtube.com/watch?v=JeQzivuFouM) until another pizza arrives. However, the first person that finds that there is no more pizza will call up Seasons to order another pizza before taking a nap.

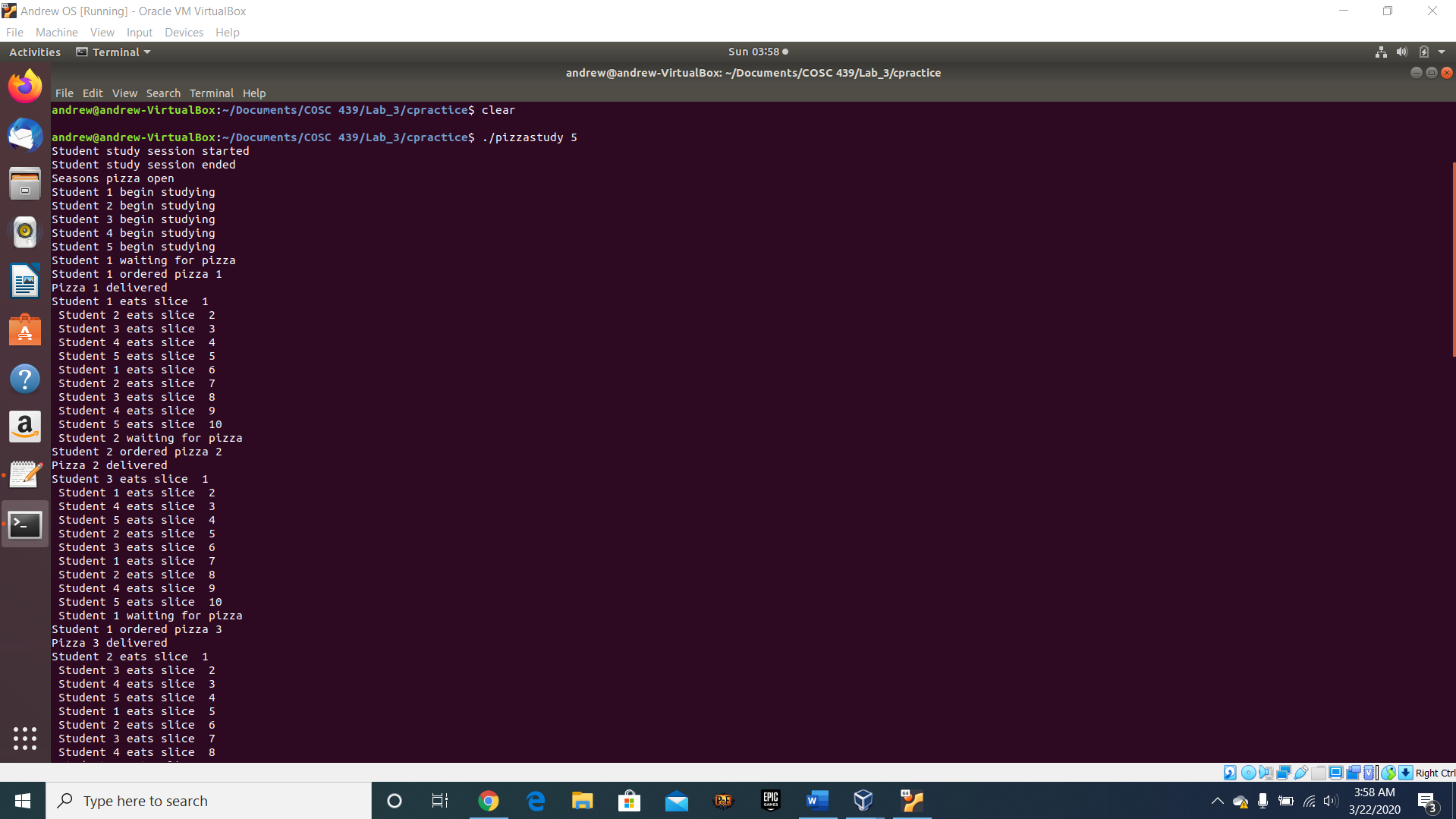
­­

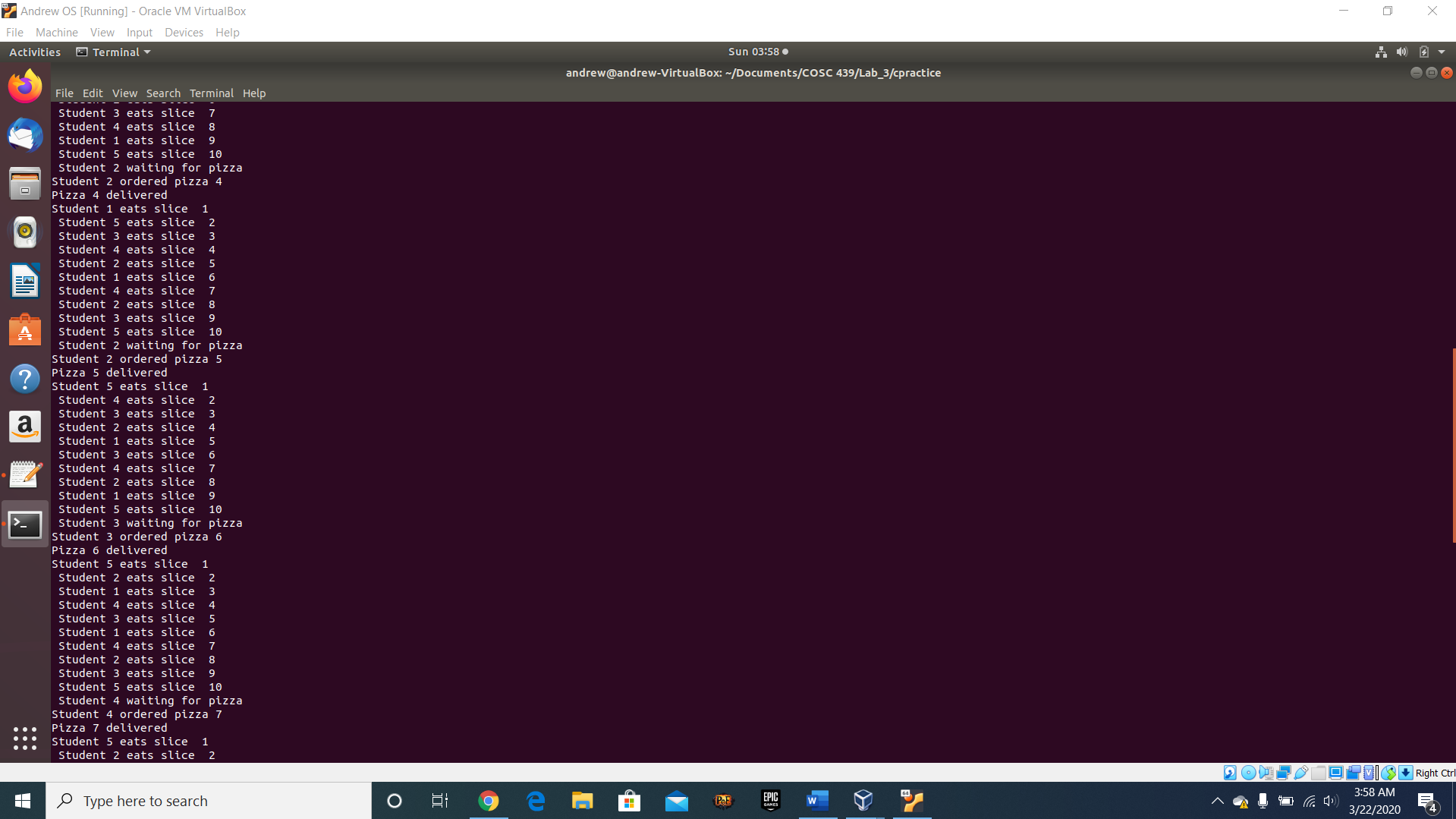
You solution *must* synchronize between the student threads (determined by the number of students in ­the study group passed in as a runtime parameter) and the pizza delivery thread. You must ensure that deadlock situations are avoided, and that Seasons is only called one time for each ordered pizza.

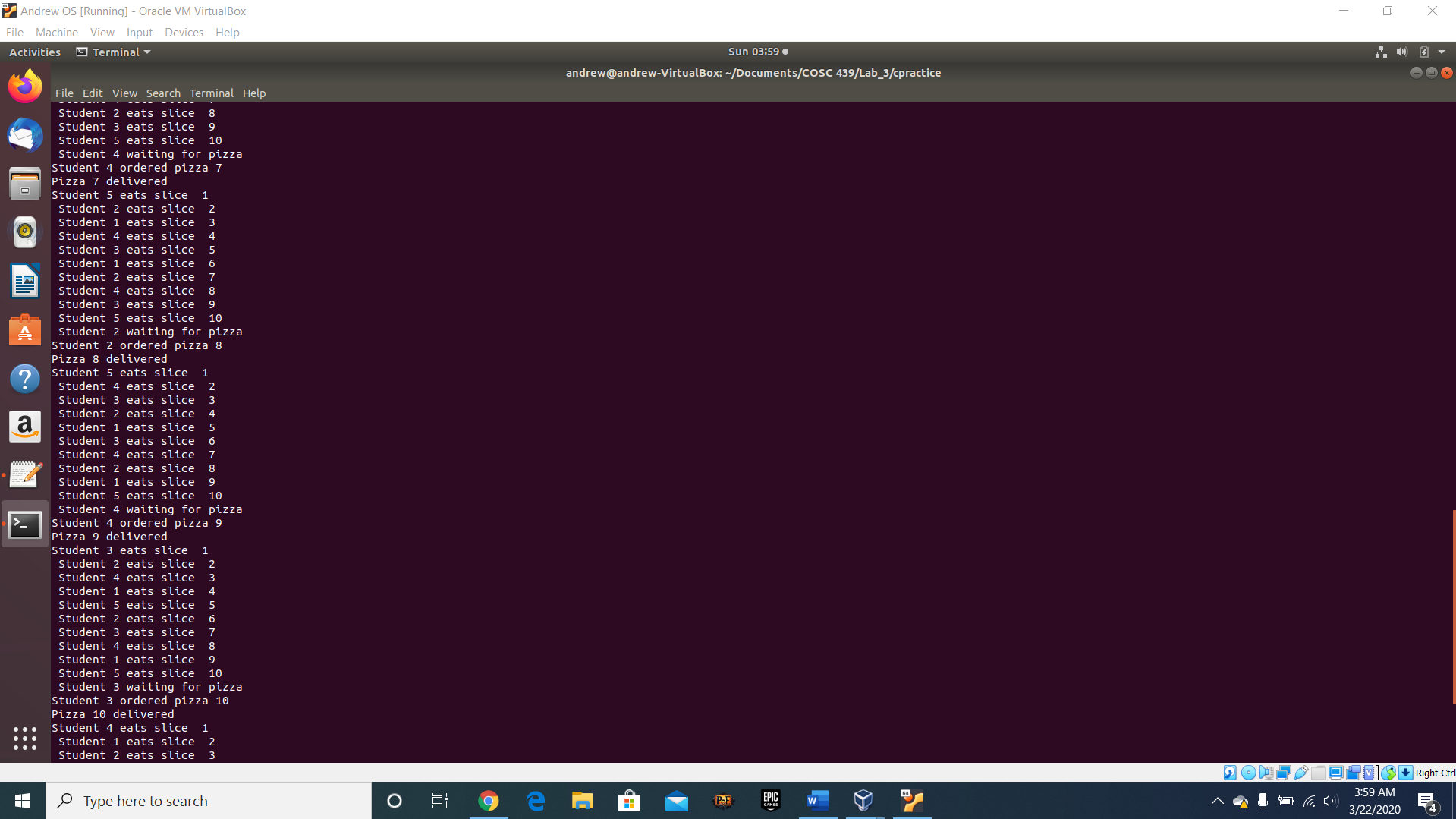
Finally, your solution should print to the console information that includes when a student eats a slice of pizza (“Student 1 eats slice 5 from pizza 3”), when a new pizza is ordered (“Student 3 orders pizza 4”), and when a pizza is delivered (“Pizza 6 is delivered”).

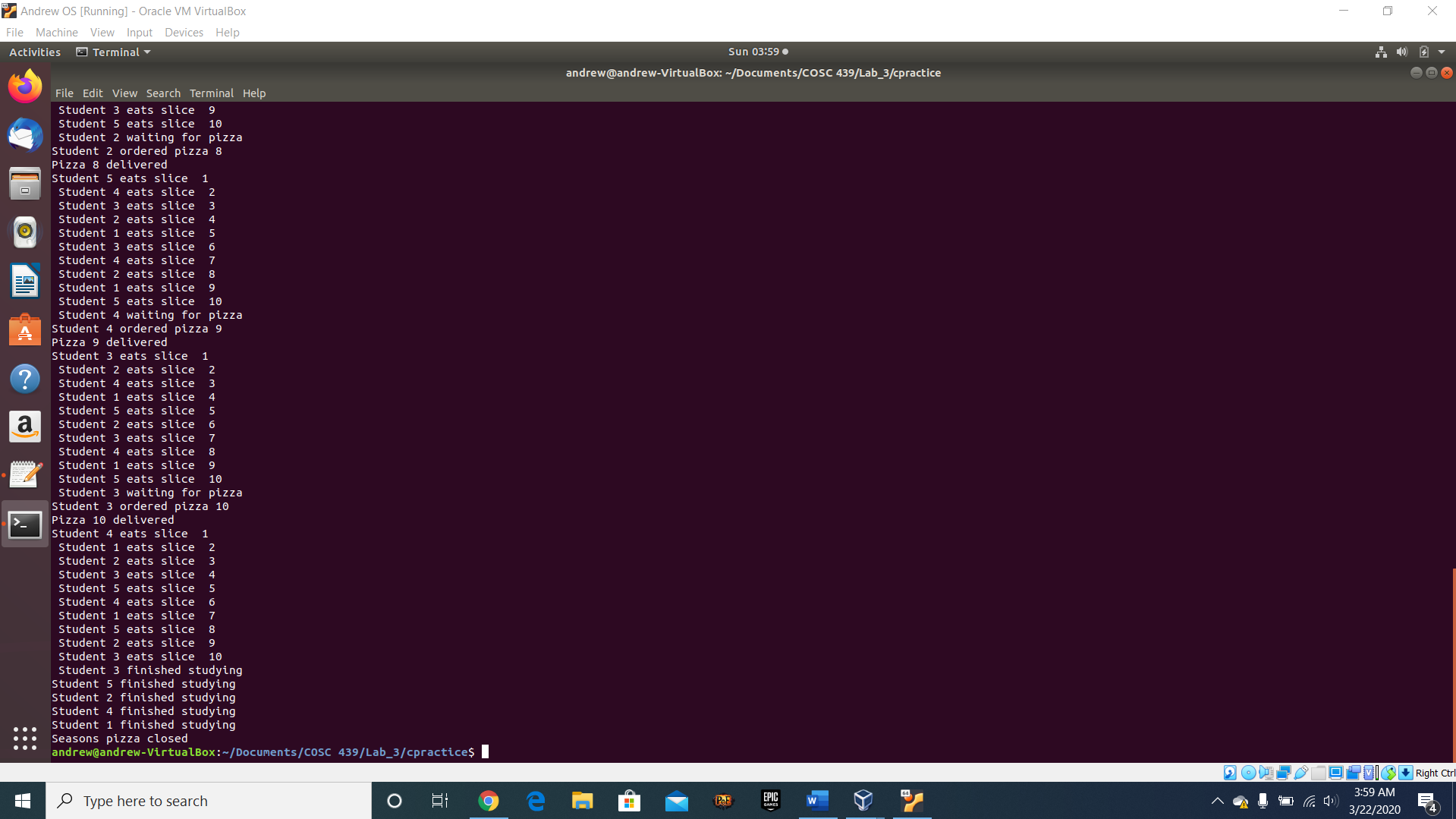
Once you have it working, ***include a screenshot of several runs of your output here and include the pizzastudy.c program in your cpractice directory.***

Screenshot with 5 students:

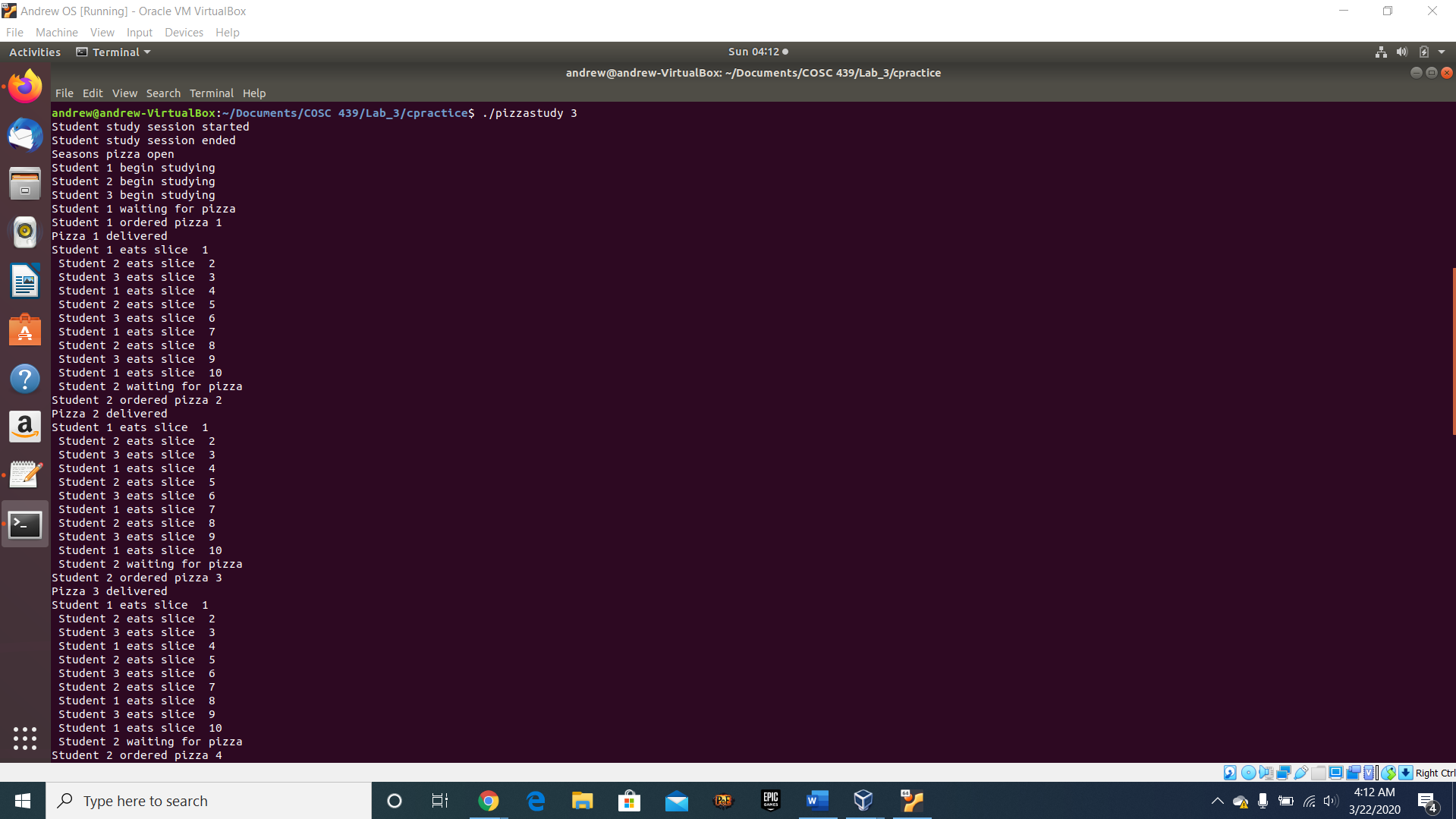


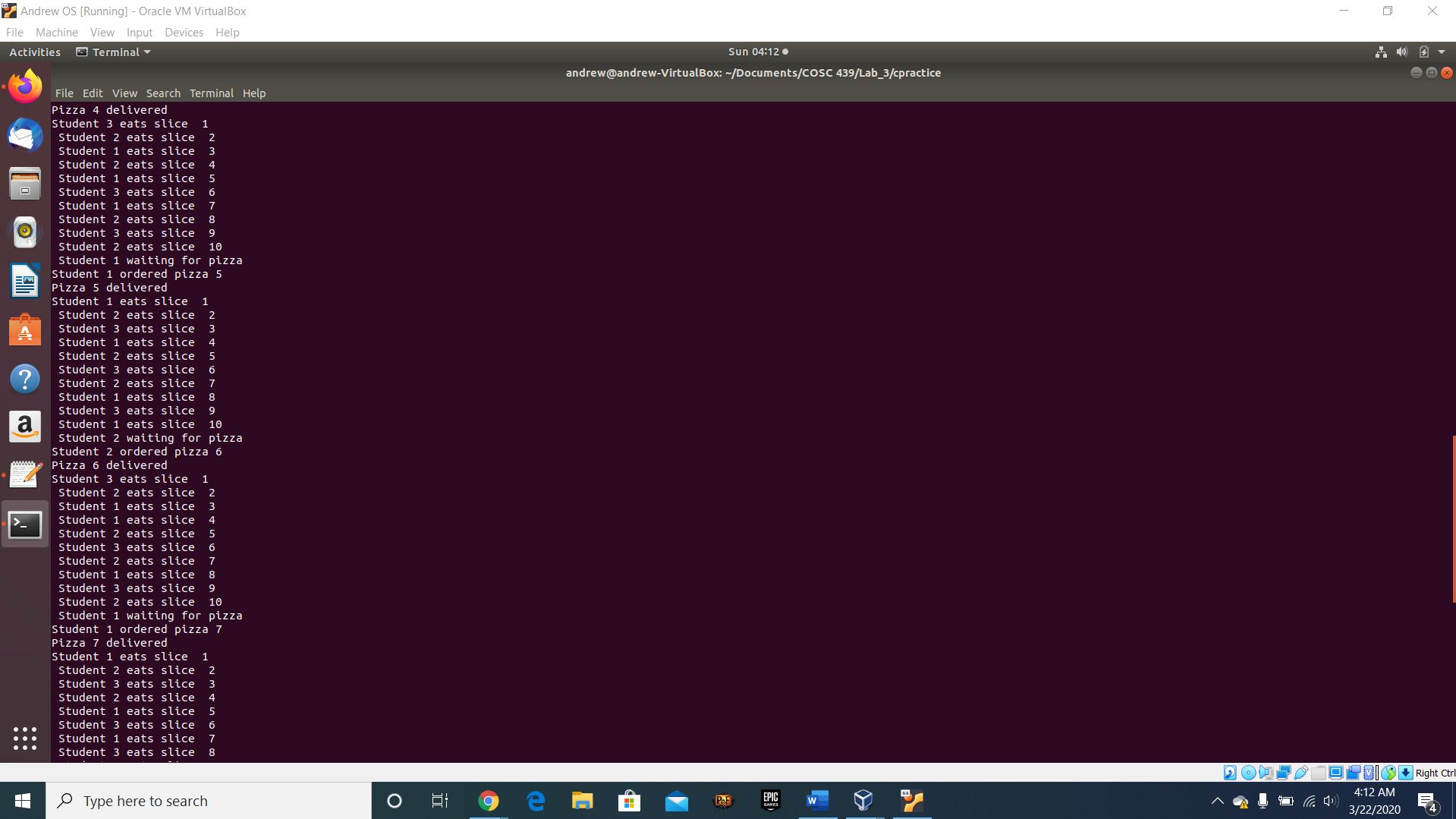


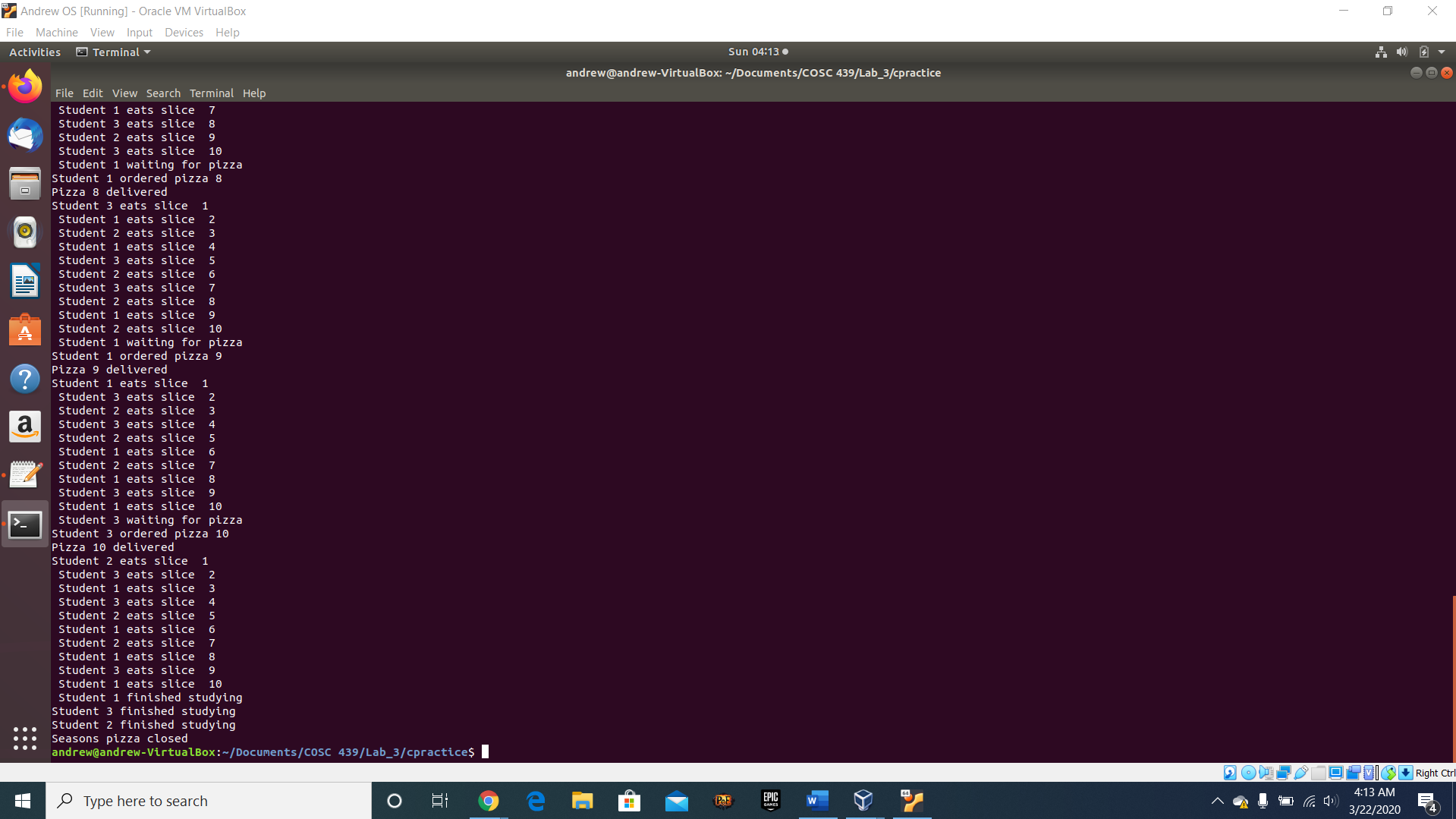




Screenshot 2 (with 3 students):







**Note:** There are at least 3 [Easter eggs](https://en.wikipedia.org/wiki/Easter_egg_(media)) (this isn’t one of them) in this lab. If you find one, Slack me and let me know!