**Assignment 2: Monitors and Semaphores**

Duc Vo and Ty Gehrke

Grand Canyon University

CST-315: Operating Systems Lecture and Lab

Professor Citro

2/18/2024

**Team Members**

* Ty Gehrke
* Duc Vo

**Project Description**

In this project, we will explore and demonstrate our understanding of synchronization mechanisms in the context of processes and threads in Unix/Linux environments. Specifically, our focus will be on the difference between monitors and semaphores. Through implementation and analysis, we gain insights into the application, advantages, and limitations of each mechanism.

**Scenario**

The scenario provided involves a resource management problem where multiple processes or threads compete for access to a shared resource. For instance, we consider a scenario where a printing service is accessed by multiple users concurrently. To ensure that only one user can print at a time, a synchronization mechanism is necessary.

**Synchronization Mechanisms: Monitors vs Semaphores**

Students will design and implement two separate C programs, each utilizing a different synchronization mechanism – one employing semaphores and the other utilizing monitors. They will analyze the pros and cons of each approach in the given scenario:

## Semaphores:

* Pros: Allows for flexible synchronization strategies, such as binary semaphores for mutual exclusion.
* Cons: Requires careful handling to avoid deadlocks and race conditions, as explicit signaling is necessary.

## Monitors:

* Pros: Provides a higher level of abstraction, simplifying synchronization through built-in support for mutual exclusion and condition variables.
* Cons: Limited flexibility compared to semaphores, potentially leading to less efficient solutions in certain scenarios.

**Recommendation**

We would recommend using monitors over semaphores.

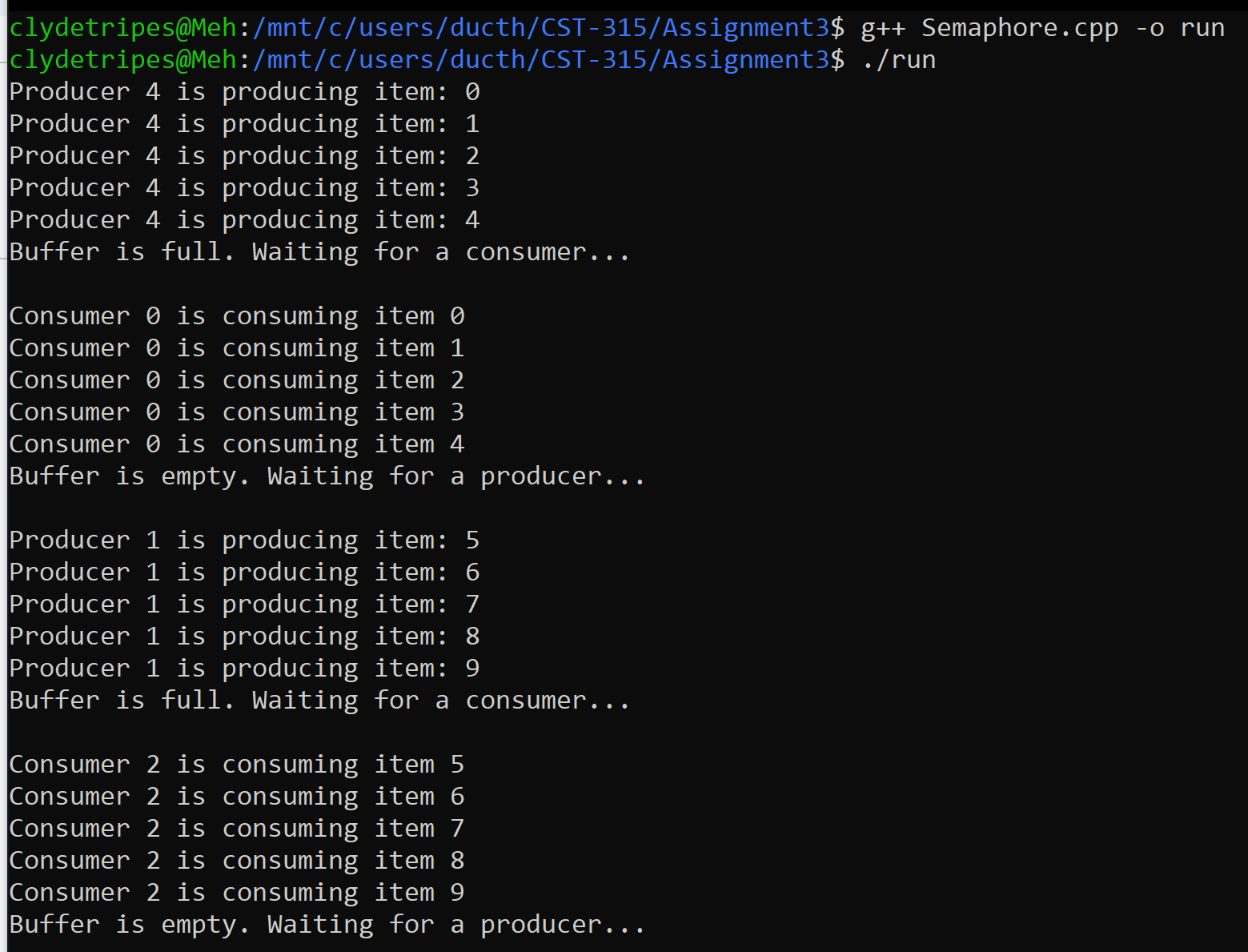
1. Simplicity and Ease of Use: Monitors provide a higher level of abstraction compared to semaphores. They encapsulate both data and synchronization primitives within a single construct, making it easier to manage concurrent access to shared resources. With monitors, the programmer doesn't need to explicitly manage synchronization primitives like in the case of semaphores, reducing the likelihood of errors such as deadlocks and race conditions.
2. Built-in Mutual Exclusion: Monitors inherently support mutual exclusion, ensuring that only one thread can execute a critical section of code at a time. This simplifies the implementation, as there's no need to manually control access to shared resources using semaphores.
3. Reduced Complexity and Potential for Errors: Using monitors can lead to simpler and more understandable code compared to semaphores. With semaphores, the programmer needs to carefully manage the signaling and waiting of threads, which can introduce complexity and increase the potential for errors.

Overall, while semaphores offer more flexibility in synchronization strategies, monitors provide a more straightforward and intuitive solution for managing access to shared resources in the given scenario. Therefore, we would recommend using monitors for this project.

**Screenshots**

## Semaphores

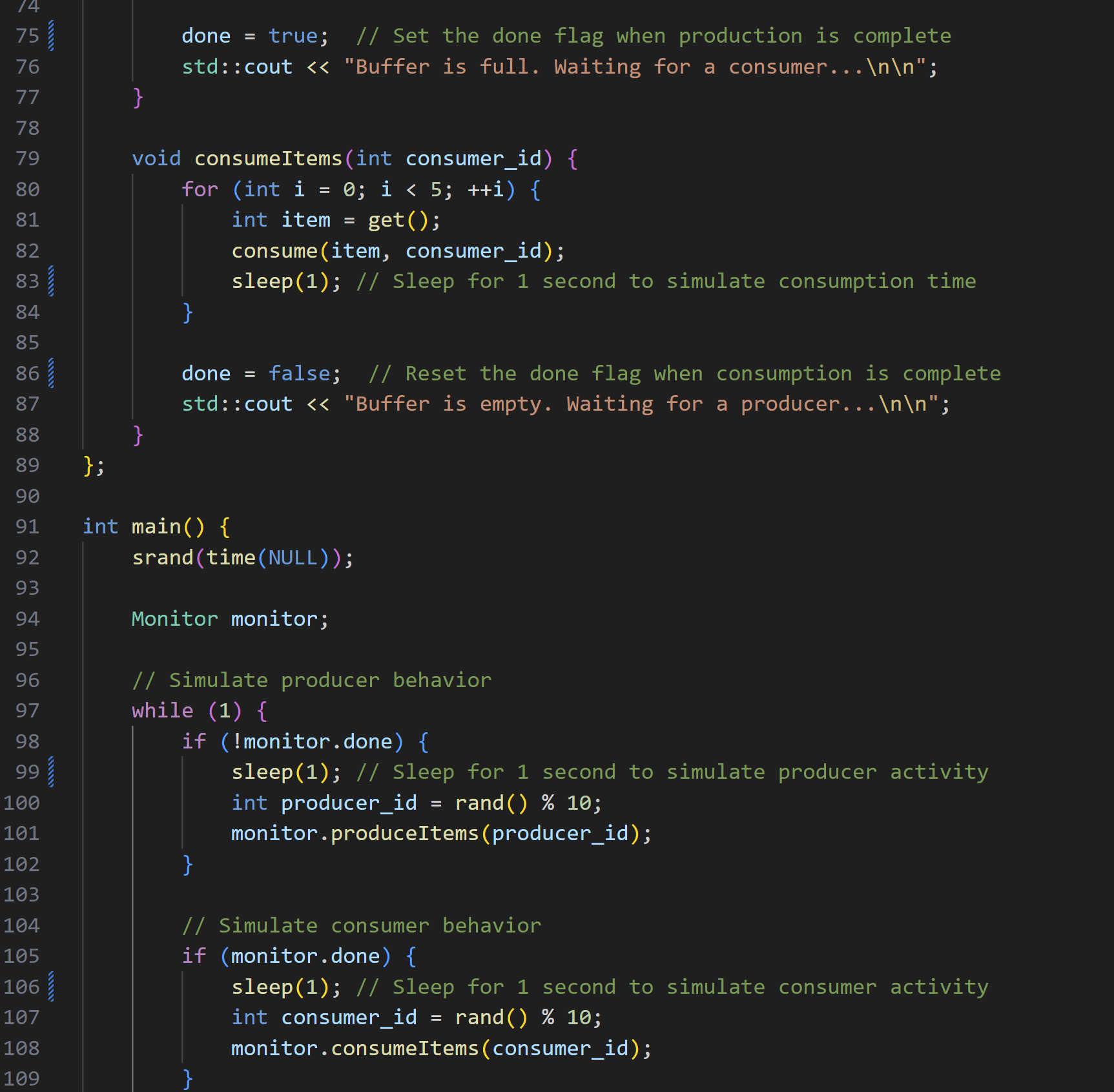
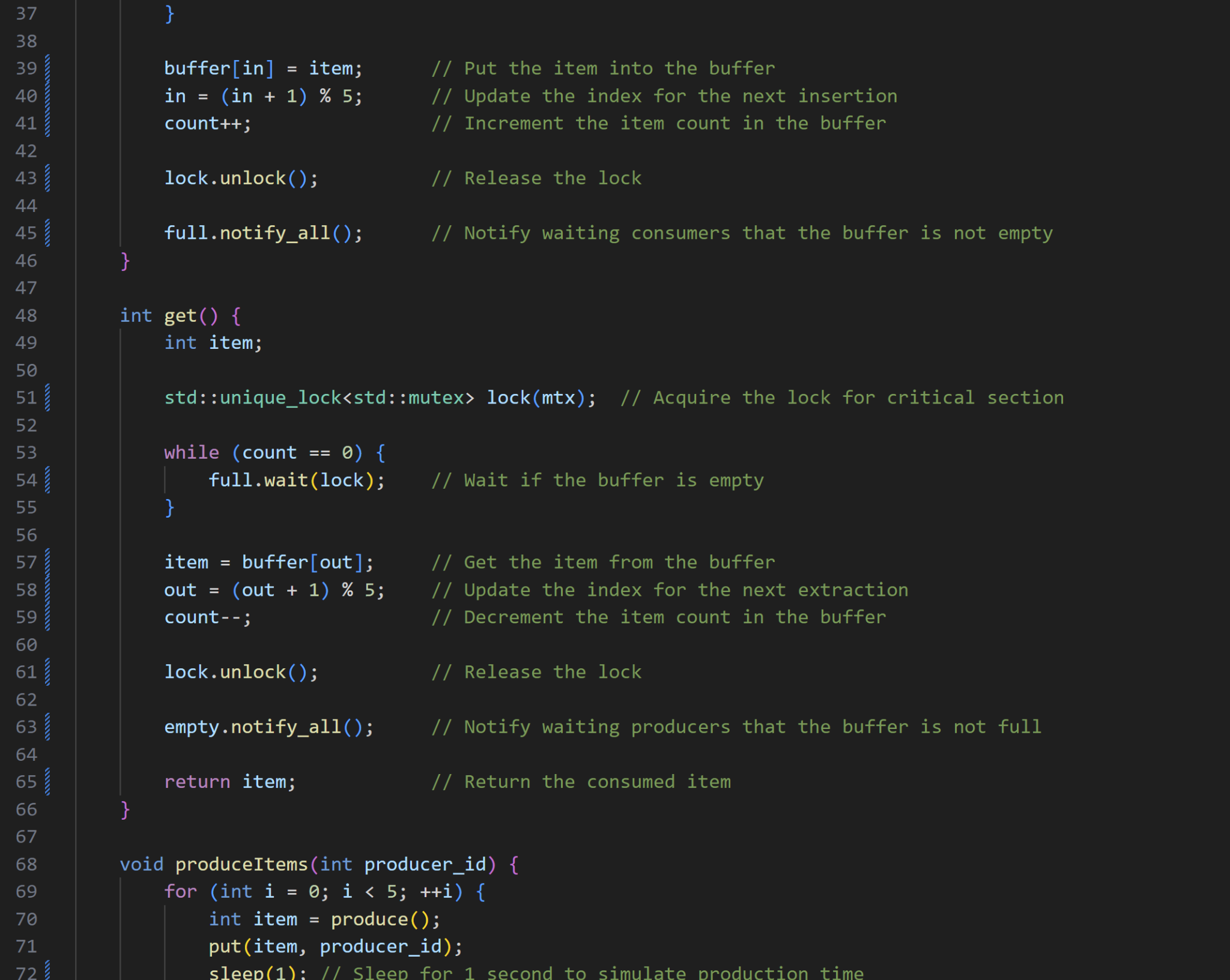
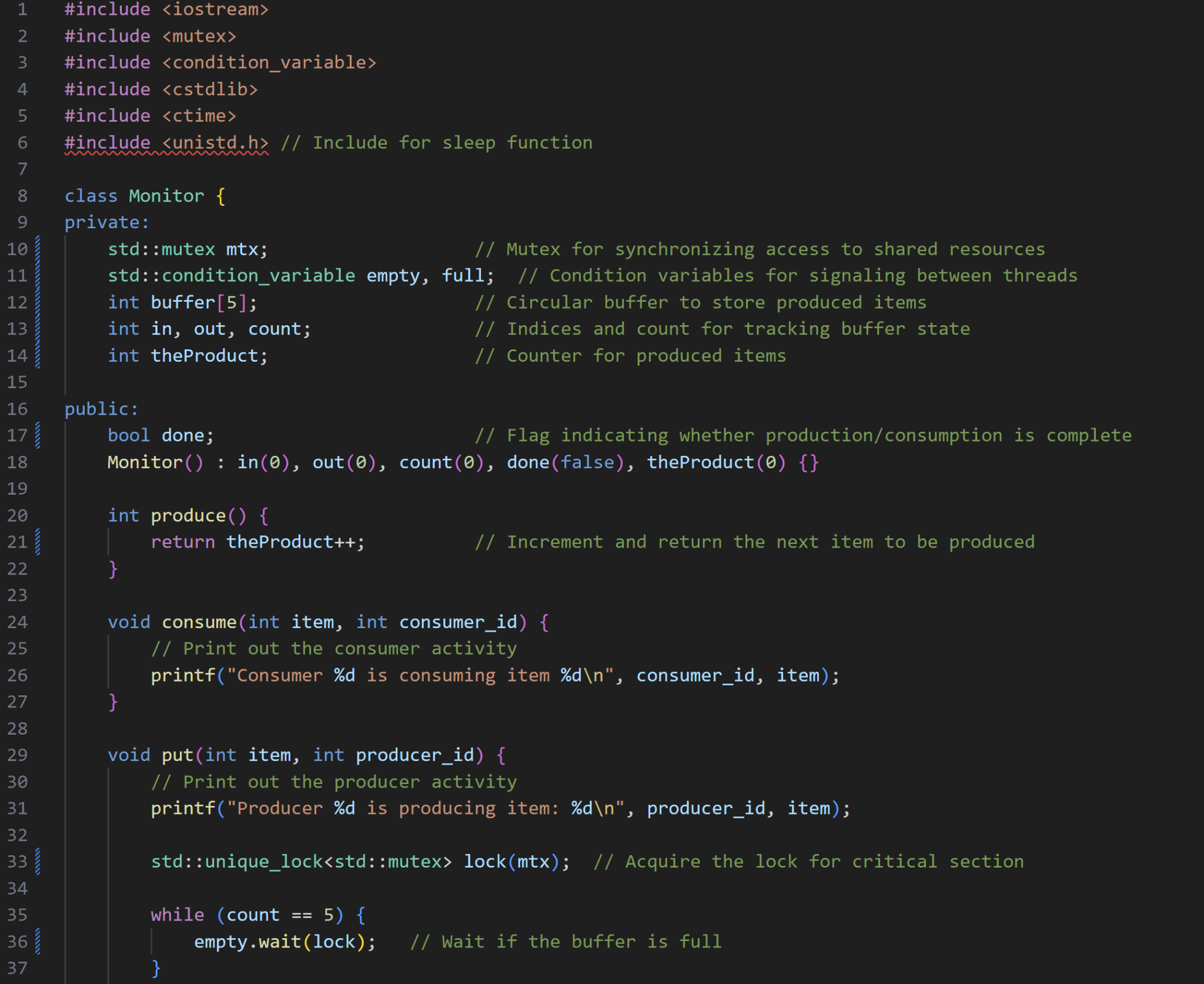
## 



## Monitor

A monitor is a higher-level synchronization concept in concurrent programming that combines data and procedures to regulate access to shared resources. While C++ lacks a direct monitor construct, the principles can be emulated using lower-level primitives. In the provided code, `std::mutex` is employed for mutual exclusion, allowing only one thread to access critical sections at a time. Combined with condition variables, this creates a controlled environment for shared resource access. The `Monitor` class encapsulates these mechanisms, providing a structured interface with methods like `put` and `get`. Through the use of mutexes, the code demonstrates how to implement monitor-like behavior in C++, emphasizing a more organized approach to concurrent programming.

To illustrate the principles of a monitor, a `Monitor` class is introduced in the provided code. This class encapsulates synchronization mechanisms using `std::mutex` and `std::condition\_variable`. The inclusion of a `produce` method for item generation, a `consume` method for item consumption, and `put` and `get` methods for managing shared resources defines the behavior of the monitor. By structuring the code within the `Monitor` class, we showcase a cleaner, higher-level abstraction for coordinating the activities of multiple threads. This design enhances code readability, modularity, and maintenance while emphasizing the implementation of monitor-like functionality using mutexes in a C++ environment. The `Monitor` class serves as a key component in demonstrating how to achieve organized and efficient concurrent programming through monitor principles.





**References**

GeeksforGeeks. (2020, March 16). *Monitor vs semaphore*. GeeksforGeeks. https://www.geeksforgeeks.org/monitor-vs-semaphore/