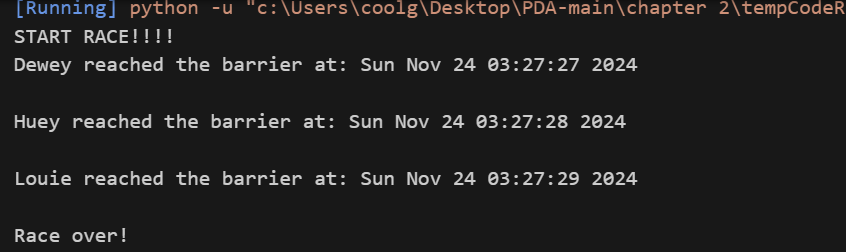
**ASSIGNMENT**

**Barrier.py:**

Race between three runners, using threads to represent each participant.

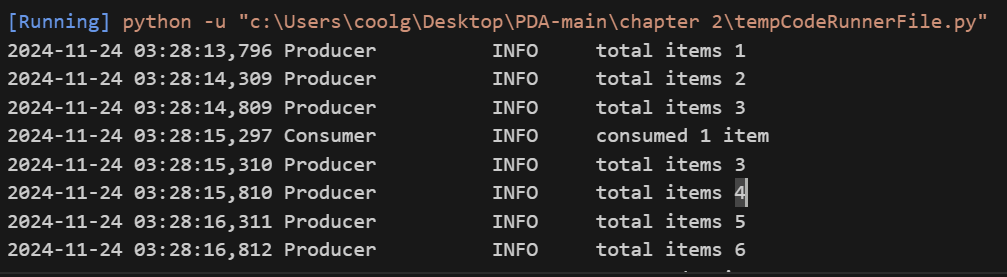
How the Race Works:

1. Three threads are made, each of which stands for a runner.   
2. Every thread begins to run, simulating varying running rates with a random sleep duration.  
3. It prints a message and waits for the other runners when a runner crosses the finish line, also known as the barrier.   
4. The barrier is released and the race is said to be ended after every runner has crossed the finish line.   
This code essentially shows how to utilize threads and synchronization techniques to mimic a parallel race in which runners run at varying speeds but finish at around the same time.



**Condition.py:**

The code uses synchronization methods and threads to implement a producer-consumer pattern. It creates a shared list to hold objects and uses a Condition object to synchronize producer and consumer thread access to the list.  
When new items become available, the producer thread notifies the customer and adds them to the list. Items are taken off the list by the consumer thread, which also alerts the producer when space opens up. By preventing race circumstances and guaranteeing effective resource use, this makes sure that the producer and consumer threads cooperate.



A screenshot of a computer

Description automatically generated

**Event.py:**

It uses threads and an Event object for synchronization, illustrating a producer-consumer pattern. The Event object is used by the producer thread to notify the consumer thread when items are created and added to a shared list. The consumer thread awaits the event that indicates an item's availability. After receiving a signal, it logs a message and consumes the item from the list. This method optimizes resource usage and prevents race situations by guaranteeing effective coordination between the two threads.

A screenshot of a computer

Description automatically generated

**MyThreadClass.py:**

Threading is used to execute tasks concurrently. It specifies a unique MyThreadClass in which every thread signals completion, sleeps for an arbitrary amount of time, then outputs its name. Nine threads are created, launched, and operated concurrently in the main function, which shortens the execution time overall. The join() method makes sure that before displaying "End" and the overall runtime, the main program waits for all threads to finish. It emphasizes the use of threads for effective, parallel job execution.

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

**MyThreadClass\_lock.py:**

It uses the concept multithreading notion with lock-based synchronization. It generates several threads, each of which has a different sleep duration. Through the use of a lock, race circumstances are avoided in which threads could interfere with one another's output. This lock guarantees that the shared resource (printing) can only be accessed by one thread at a time. A thread that has obtained the lock prints its message, releases the lock, and allows the subsequent thread to take over. Even when numerous threads are running concurrently, our method ensures that the output is organized and conflict-free.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

**MyThreadClass\_lock2.py:**

This code demonstrates using threading module to run multiple tasks concurrently. A custom thread class, MyThreadClass, is defined to execute tasks. Each thread acquires a shared lock (threadLock) before printing its details to ensure thread-safe operations, then sleeps for a random duration before completing.

In the main function, nine threads are created and started. The join() method ensures the main program waits for all threads to finish. Once done, the program prints "End" and the total execution time. This example highlights how threading allows concurrent execution while maintaining safe access to shared resources.

A screenshot of a computer

Description automatically generated

**A screen shot of a computer

Description automatically generated**

**Rlock.py:**

It uses threads to mimic concurrently inserting and removing items from a shared box. The Box class uses a reentrant lock (RLock) to provide thread-safe operations and keeps track of the total number of elements. Items are added by the adder function and removed by the remover function, both of which pause for one second in between operations. Two threads—one to add a random number of items and the other to remove a random number—are launched in the main function. The lock guarantees synchronized access to the shared resource while the threads operate concurrently. This is an example of producer-consumer thread-safe concurrency.

**A screenshot of a computer program

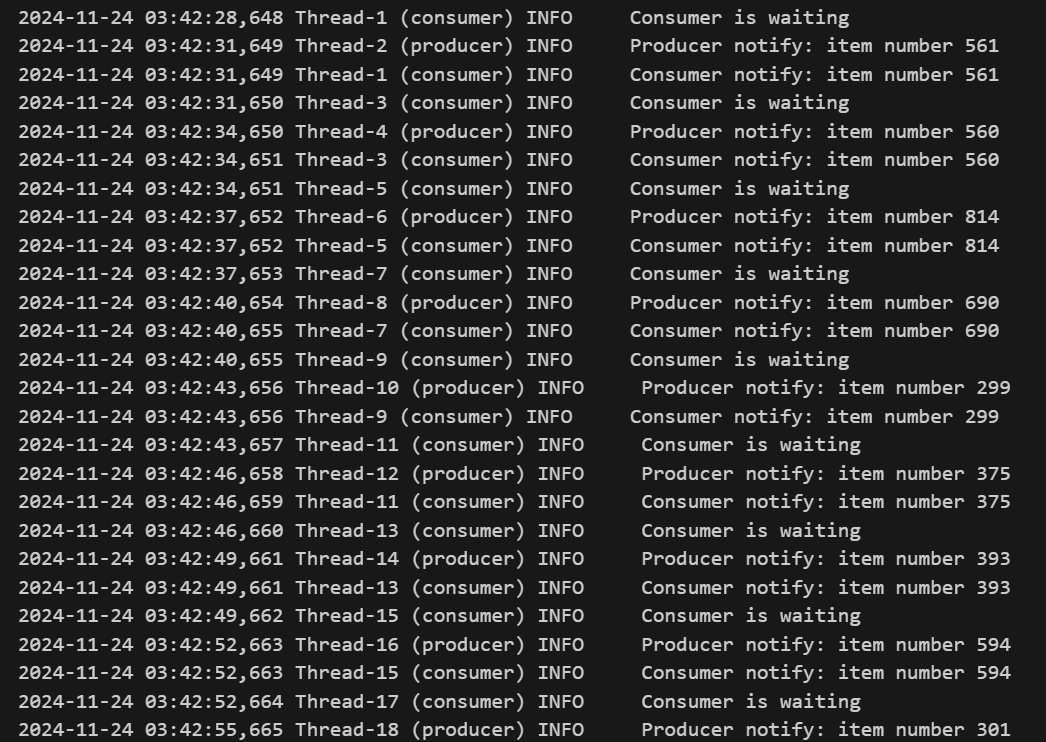
Description automatically generated**

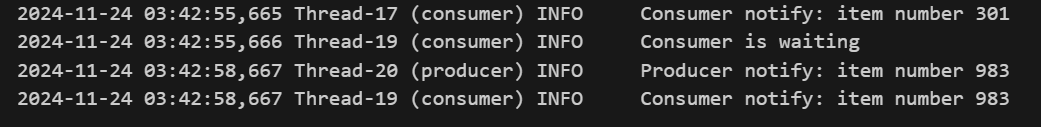
**A screenshot of a computer program

Description automatically generated**

**Semaphore.py:**

Using threads and a semaphore for synchronization, it displays a producer-consumer pattern. A random integer is created by the producer thread and saved in a shared variable. The consumer thread awaits the producer's release of the semaphore, which signals the arrival of a new item. The consumer thread reads the shared variable, processes the item (in this case, just logging it), and then releases the semaphore after acquiring it. This guarantees that the producer thread doesn't create a new item until the previous one has been eaten, and that the consumer thread only accesses the object once it has been produced. This system guarantees that the producer-consumer pattern is correctly implemented and avoids race situations.





**Thread\_defination.py:**

the creation and administration of numerous concurrent jobs through the use of threads.   
Its functionality:  
The function my\_func, defined in the code, accepts a thread number as an argument and outputs a message identifying the thread that invoked the function. Ten threads are created by the main function, and each one is given a distinct number between 0 and 9. The my\_func function is the target of each thread when it is started, and the thread number is sent as an argument. To make sure the main thread waits for all child threads to finish before continuing, use the join() method. This enables the effective completion of several activities at once.

A screen shot of a computer

Description automatically generated

**Thread\_determine.py:**

Three functions (functions A, B, and C) are executed simultaneously using a threading module. By publishing a start message, sleeping for two seconds, and then writing an exit message, each function mimics a task. Start() is used to create three threads, assign one of the functions to each, and begin the threads. The main program waits for all threads to finish before terminating thanks to the join() technique. This illustrates fundamental multithreading, which enables independent processes to execute concurrently and increases efficiency through parallel task execution.

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Thread\_name\_and\_processes.py:**

It is using the basic concept of multithreading. Two threads are created, each with the straightforward task of printing a message. To generate and control these threads, utilize the threading module. Concurrent execution allows the threads to run their code simultaneously after they are launched. The join() method makes sure that before moving forward, the main program waits for both threads to finish. This makes it possible to complete several jobs quickly and effectively, which could enhance program responsiveness and performance.

A black background with white text

Description automatically generated

**Thread\_with\_queue.py:**

It implements a producer-consumer pattern using threads and a queue. Random numbers are created by the producer thread and added to a shared queue. Several consumer threads keep an eye on the queue at all times, processing and deleting items as they become available. By serving as a buffer, the queue makes sure that the producer and customers may communicate easily. The queue can effectively manage its internal state by using the task\_done() method to signify when item processing is complete. Multiple jobs can be executed concurrently with this method, which enhances program responsiveness and performance overall.

