Task:

Imagine a parking lot with 5 parking spaces and a stream of 10 cars arriving randomly.  Each car needs to park in a space, stay for a random amount of time, and then leave.  If no spaces are available, the car must wait until one is free. Your goal is to write a program that:

1. Simulates the behavior of cars entering and leaving the parking lot.

2. Ensures no two cars occupy the same parking space simultaneously.

3. Prevents issues like cars waiting indefinitely.

Deliverables:

1. Program Code: o A well-documented program that simulates the parking lot scenario and includes synchronization mechanisms.

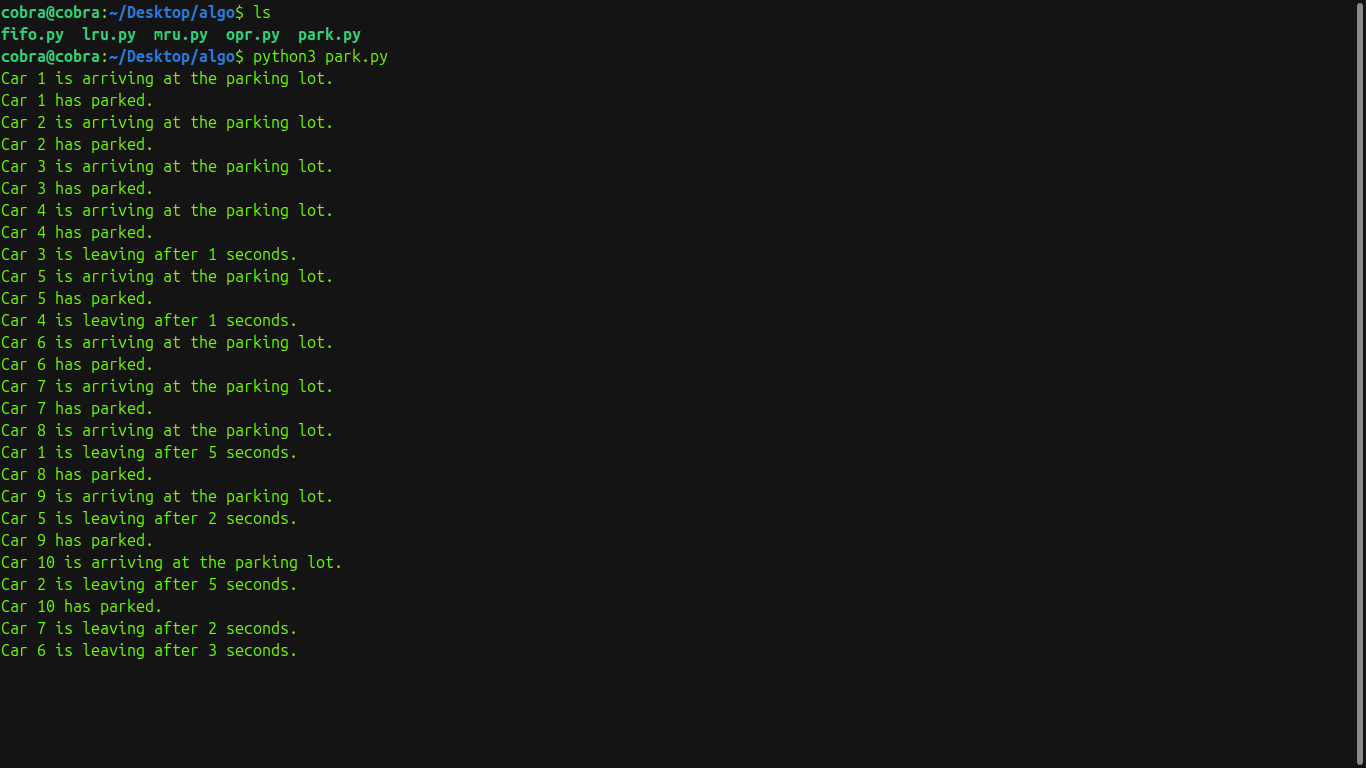
2. Short Report: o Describe the strategy you used and any variations implemented. o Highlight challenges faced and how you solved them.

o Discuss insights gained from experimenting with variations.

Park.py:

|  |
| --- |
| import threading  import time  import random  # Constants  TOTAL\_PARKING\_SPACES = 5  TOTAL\_CARS = 10  # Semaphore to represent available parking spots  parking\_lot = threading.Semaphore(TOTAL\_PARKING\_SPACES)  # Lock to prevent race conditions when accessing shared resources  print\_lock = threading.Lock()  def car\_behavior(car\_id):  with print\_lock:  print(f"Car {car\_id} is arriving at the parking lot.")  # Try to acquire a parking spot  parking\_lot.acquire()  with print\_lock:  print(f"Car {car\_id} has parked.")  # Simulate time spent parked  parking\_duration = random.randint(1, 5)  time.sleep(parking\_duration)  # Leaving the parking lot  with print\_lock:  print(f"Car {car\_id} is leaving after {parking\_duration} seconds.")  # Release the parking spot  parking\_lot.release()  def simulate\_parking\_lot():  threads = []  for i in range(1, TOTAL\_CARS + 1):  t = threading.Thread(target=car\_behavior, args=(i,))  threads.append(t)  t.start()  # Optional delay to simulate staggered car arrivals  time.sleep(random.uniform(0.2, 1.0))  for t in threads:  t.join()  if \_\_name\_\_ == "\_\_main\_\_":  simulate\_parking\_lot() |

Output:



Strategy Used:

* Threading: Each car is a thread.
* Semaphore: Manages 5 parking spaces.
* Lock: Prevents jumbled print output.
* Cars wait if no space is free, then park for a random time before leaving.

Challenges:

* Race Conditions in Output: Solved using print\_lock.
* Starvation Risk: Avoided with semaphore’s fair thread handling.

Insights:

* Semaphores simplify resource control.
* Threading simulates real concurrency.
* Random delays make the simulation realistic.
* Output synchronization is key in multithreaded programs.