As per the assignment I have created Seven python files named **Q3.py , Q4.py , Q5.py , Q6_server.py, Q6_client.py , Q7_server.py and Q7_client.py**
This report contains a detailed explanation of my code.

Description of Q3.py :

The Q3.py file contains two thead namely **EG and EC.**

- **EG** thread generates **Events** by calling **generateEvent()** method as **target**
- **EC** thread consumes **Events** by calling **consumeEvent()** method as **target**
- **q** list is used to store generated events.
- **complete** event is used as a flag to check compilation of thread generation.
- **start()** is used to start **EG and EC** thread .

generateEvent() method :

- **generateEvent()** has an internal variable **count** , which specifies how many **events** will be generated by the EG thread.
- If there is **no more event to consume by EC thread** , i.e. if **q is empty** , **a new event will be generated and stored in q to be consumed later by EC**.
- Before generating a event, it will wait for **γ** `sec , where` **γ** is uniformly sampled from {1,2,3,4,5}
- Lock variable **lock1** is used here to ensure atomicity .
- **acquire()** and **release()** is used to acquire and release lock.
- **perf_counter** is used to determine the perfect time at that program point.
- **sleep()** is used to pause the program flow for a specific amount of time.
- **set()** is used to set the event created.
- After all thread generation , the **'complete'** event will get cleared by **clear()** method .

consumeEvent() method :

- **consumeEvent()** is used to consume events generated by EG thread .
- **As soon as an event is available for consumption i.e. q has an event stored , the EC thread starts consuming it.**
- As long as **complete** is set or **q** has an element , **EC** thread **consumes events from q.**
- After all events are consumed , print the appropriate message .
- If **q** is not empty, print that the event has occurred , wait for consumption compilation .
- Every event takes a random amount of time μ as processing time . μ is uniformly sampled from {1,2,3,4,5} . hence sleep for `μ sec.`
- **pop()** the element from **q** and consume it by clearing the event .
- Lock variable **lock2** is used here to ensure atomicity .
- **acquire()** and **release()** is used to acquire and release lock.
- **perf_counter** is used to determine the perfect time at that program point.
- **sleep()** is used to pause the program flow for a specific amount of time.
- **clear()** is used to clear an event.

Test Cases outputs:

Time 0s : Event scheduled at 2s

Time 2s : Event occured
Time 3s : Event Processed
Time 3s : Event scheduled at 6s
Time 6s : Event occured
Time 9s : Event Processed
Time 10s : Event scheduled at 11s
Time 12s : Event Processed
Time 12s : Event scheduled at 13s
Time 13s : Event occured
Time 16s : Event Processed
Time 16s : Event scheduled at 20s
Time 20s : Event occured
Time 21s : Event Processed
Time 21s : Event scheduled at 24s
Time 24s : Event occured
Time 28s : Event Processed
Time 28s : Event scheduled at 29s
Time 31s : Event Processed
Time 31s : Event scheduled at 34s
Time 34s : Event occured
Time 36s : Event Processed
Time 37s : Event scheduled at 38s
Time 38s : Event occured
Time 39s : Event Processed
Time 39s : Event scheduled at 40s
Time 40s : Event occured
Time 42s : Event Processed
All events have been processed

Description of Q4.py :

The Q4.py file contains five threads each corresponding to a person visiting the mall**.**

- Five threads are created and stored in the 'arr' list with the target method as person_in_mall and arguments as name, arrival and visit .
- Start all the threads present in 'arr'.
- When a person reaches the mall, they wait for others. As soon as everyone arrives, all of them enter the mall.
- A barrier, 'barrier' is used to ensure all threads will go inside the mall at the same time.

person_in_mall() method :

- Each person reaches the mall at a random amount of time i.e. **arrival** . **arrival** is uniformly sampled from the set {1, 2, . . . , 19, 20} . each thread sleeps for **arrival** time initially to ensue this functionality.

- When a person reaches the mall, they wait for others. As soon as everyone arrives, all of them enter the mall. Before that , each thread waits at the **barrier** for every person to reach the mall
- **barrier.wait ()** is used to make all threads wait at the **barrier.**
- Subsequently, each of them spend a random amount that uniformly sampled from the set {1, 2, . . . , 9, 10}of time in the mall and then leaves the mall. Each thread sleeps for **visit** time to ensure this functionality.
- **perf_counter** is used to determine the perfect time at that program point.
- **sleep()** is used to pause the program flow for a specific amount of time.

Test Cases outputs:

Time 3s : Person 2 reached the mall
Time 5s : Person 3 reached the mall
Time 9s : Person 5 reached the mall
Time 9s : Person 1 reached the mall
Time 16s : Person 4 reached the mall
Time 16s : Person 4 enters the mall
Time 16s : Person 2 enters the mall
Time 16s : Person 1 enters the mall
Time 16s : Person 3 enters the mall
Time 16s : Person 5 enters the mall
Time 19s : Person 4 leaves the mall
Time 20s : Person 5 leaves the mall
Time 21s : Person 2 leaves the mall
Time 24s : Person 3 leaves the mall
Time 24s : Person 1 leaves the mall

Description of Q5.py :

The Q5.py file contains five threads each corresponding to a person visiting the shop**.**

- Five threads are created and stored in the ‘arr’ list with the target method as person_in_shop and arguments as name, arrival and visit .
- Start all the threads present in ‘arr’.
- No more than 2 persons can be in the shop at any time.
- As soon as a person reaches the shop they enter it. However, if there are 2 people in the shop, he/she has to wait till someone leaves the shop
- **semaphore** is used to ensure no more than 2 people can enter the shop.

person_in_shop() method :

- **acquire()** and **release()** is used to acquire and release **semaphore**.
- Each person reaches the shop at a random amount of time i.e. **arrival** . **arrival** is uniformly sampled from the set {1, 2, 3, 4, 5} . each thread sleeps for **arrival** time initially to ensue this functionality.

- As a person arrives , print that the person reached the shop along with the time .
- No more than 2 persons can be in the shop at any time.
- As soon as a person reaches the shop they enter it. However, if there are 2 people in the shop, he/she has to wait till someone leaves the shop. **semaphore** is used to ensure no more than 2 people can enter the shop.
- **semaphore.acquire()** will be executed only if there are less than 2 people inside the shop.
- After entering the shop , print that the person entered the shop along with the time .
- After entering, a person spends a random amount of time (i.e **visit)** in the shop before leaving . **visit** is uniformly sampled from the set {5, . . . , 10} . Each thread sleeps for **visit** time to ensure this functionality.
- After visiting the shop , the person leaves the shop . print that the person left the shop along with the time and release the semaphore (i.e. **semaphore.release()**) so that the person waiting outside can enter the shop.
- **perf_counter** is used to determine the perfect time at that program point.
- **sleep()** is used to pause the program flow for a specific amount of time.

Test Cases outputs:

Time 1s : Person 1 reached the shop
Time 1s : Person 1 entered the shop
Time 3s : Person 5 reached the shop
Time 3s : Person 5 entered the shop
Time 3s : Person 4 reached the shop
Time 4s : Person 3 reached the shop
Time 4s : Person 2 reached the shop
Time 8s : Person 1 left the shop
Time 8s : Person 4 entered the shop
Time 11s : Person 5 left the shop
Time 11s : Person 3 entered the shop
Time 14s : Person 4 left the shop
Time 14s : Person 2 entered the shop
Time 18s : Person 3 left the shop
Time 20s : Person 2 left the shop

Description of Q6_server.py :

The Q6_server.py represents the server code for Q6**.**

- Creates a server socket (i.e. **s**) using **socket.socket()** method .
- **Bind** the socket s with ip address as localhost and port number as 8888 , using **bind**() method
- Listen for client connection-request using the **listen**() method .
- Accept client connection using **accept**() method .
- Store the connected client socket and address in **c, addr** pair.

- Here we know that the client will send a json string object which contains 10 tuples of (a,b) where a and b are integers (an assumption)
- Receive message from client using **recv**() method with **buffer size of 1024** bytes and decode it in **utf-8** format.
- Evaluate the received message as a list object using **list(eval())**.
- Server creates the array **B = [(a, b),(amax, bmax),(amin, bmin)]** and sends it back to the client using TCP socket.
- Calculate a_bar , b_bar as average values of a's and b's , a_max ,a_min , b_max, b_min as the maximum and minimum values of a's and b's in recv_list
- create **B** list as **B = [(a_bar, b_bar),(a_max, b_max),(a_min, b_min)]**
- dump **B** as a **json** object to send to client using **json.dumps()**
- send **B** to client using **send**() method and the json object in a bytes format with utf-8 encoding i.e. **c.send(bytes(send_json,'utf-8'))**
- **Close** the connection after **B** is sent to the client to ensure proper connection termination using **close**() method.

Description of Q6_client.py :

The Q6_client.py represents the client code for Q6**.**

- Creates a client socket (i.e. **c**) using **socket.socket()** method .
- **Connect** the socket **c** with ip address as localhost and port number as 8888 , using **connect**() method .
- Form **A** as **A = [(a1, b1), . . . ,(a10, b10)]** where , value of 'a' and 'b' ranges from 1 to 100 (an assumption)
- print **A** in client side
- **Dump** A as a json object (i.e. **json_obj**) to send to client using **json.dumps()** method
- **Send** A to client as json string using **c.send(bytes(json_data,'utf-8'))**
- Receive a reply from the server using **recv**() method and decode it . rcv = c.recv(1024).decode()
- Evaluate received recv object as a list object (i.e. **recv_list**) using **list(eval()) .**
- **recv_list** contains each element as list format , make B as a list of tuple format
- Create the **B** list from **recv _list** , as **B = [(abar, bbar),(amax, bmax),(amin, bmin)]**
- Make **B** in tuple format from **recv_list**
- Print **B** in client side

Test Cases outputs:

At Server Side:

socket created
waiting for connections
connected with ('127.0.0.1', 59392)
[[50, 95], [40, 40], [63, 55], [41, 64], [57, 51], [65, 16], [70, 9], [37, 4], [44, 16], [75, 44]]
Sending B : [(54.2, 39.4), (75, 95), (37, 4)]

At scient side:

Socket created…

Socket connected…

A created : [(50, 95), (40, 40), (63, 55), (41, 64), (57, 51), (65, 16), (70, 9), (37, 4), (44, 16), (75, 44)]

A sent to server...

B received...

B : [(54.2, 39.4), (75, 95), (37, 4)]

Description of Q7_server.py :

The Q7_server.py represents the server code for Q6**.**

- Creates a server socket (i.e. **s**) using **socket.socket()** method .
- **Bind** the socket s with ip address as localhost and port number as 8888 , using **bind**() method
- Listen for client connection-request using the **listen**() method .
- Accept client connection using **accept**() method .
- Store the connected client socket and address in **c, addr** pair.
- **que1** represents the list for received messages as tuple of '(message, thread name)' form
- **que1** is rearranged in **que** in the order of sleep completion
- **que1Full** represents all messages from client threads are received or not
- Receive messages for **10** times from the client.
- Receive message from client using **recv**() method with **buffer size of 1024** bytes and decode it in **utf-8** format using **decode()** method.
- Evaluate message as list object using **list(eval())**.
- Store message as **str** and thread name ad **th**
- Append the **(str , th)** tuple to que list
- print received message in console
- when all messages received , set **que1Full** to true
- create threads for concurrent sleeping for each thread or each messages received from the client with target to **sleepingTh()**
- Check if the **que** list has any element or not , if it contains any element , that represents that for that element , sleep is completed , hence send it back to the client.
- send messages till all the 10 messages received from the client are not sent back.
- If the **que** list has element , sent back the message to client
- the message which will be sent to client , remove that from **que**
- an intentional delay is given to avoid out of order message delivery
- send a check the message to client so that client enables the proper thread to accept the message which was sent by the client earlier
- Send thread name to **activate appropriate thread in client to receive message**
- send message to client to be received by appropriate thread when the client is ready , i.e when client sends **ready** message.
- **close**() method.

sleepTh() method :

- **sleepingTh()** represents the behaviour of each thread at server side .
- start sleeping after all messages from the server are received .
- **sleep for random time** , uniformly chosen from {1,2,3,4,5} (in sec) list

- After sleep completed , acquire lock
- **pop** from **que1** and **append** to **que** to ensure que1 is rearranged in que in the order of sleep completion
- release lock , to ensure other threads work fine.

Description of Q7_client.py :

The Q7_client.py represents the server code for Q6**.**

- Creates a client socket (i.e. **c**) using **socket.socket()** method .
- **Connect** the socket **c** with ip address as localhost and port number as 8888 , using **connect**() method .
- 'arr' list stores the threads
- **'flag1arr'** is a list of all flags
- **'namearr'** is a list of all thread
- Initialize 'lock' variable which is used to ensure atomicity where needed.
- **sends()** is the target of threads which exchanges data with server with argument name
- **checker()** is the target of checkr thread which checks for which thread to activate to receive message from server.
- Create 10 threads and append them in arr list , flag1arr , namearr lists contains flag value and name of corresponding thread
- Create checker thread to check for which thread to activate to consume server reply
- Start checkr
- Start threads

sends() method :

- Each thread executes the following tasks .
- Each thread sends a tuple of message and thread name to the server using **send()** method as a json object using **json.dumps()**.
- Print messages that the message is sent to the server with timestamp.
- Each thread is **busy waiting** toreceives wakeup message for the current thread .
- As the wait completes , the thread receives a message from the server .The wait completes when the server sends back the same message sent by the thread.
- Receive message when flag1arr[index] and flag1arr[0] both are True.
- Acquire lock using **acquire()** method.
- Receive message using **recv()** method
- Prints the message received .
- set appropriate flag values (i.e flag1arr[index] and flag1arr[0]) to False to avoid confusion for other thread execution.
- Release lock using **release**() method

checker () method :

- **checker()** receives message from server and wakes up appropriate thread by setting corresponding flag value to True

- if flag1arr[0] is not True , activate thread by setting flag value of that thread from flag1arr[]
- **'activate'** represents the name of the thread to be activated for consuming server message
- Find index of the thread stored in **arr** by using **index()** method in **namearr.**
- Set flags (i.e. **flag1arr[index+1] = True, flag1arr[0] = True)**
- **send()** the ready signal by sending a **ready** message to the server .

Test Cases outputs:

At Server Side:

Hello from Thread 1 received from client thread : Thread 1
Hello from Thread 2 received from client thread : Thread 2
Hello from Thread 3 received from client thread : Thread 3
Hello from Thread 4 received from client thread : Thread 4
Hello from Thread 5 received from client thread : Thread 5
Hello from Thread 6 received from client thread : Thread 6
Hello from Thread 7 received from client thread : Thread 7
Hello from Thread 8 received from client thread : Thread 8
Hello from Thread 9 received from client thread : Thread 9
Hello from Thread 10 received from client thread : Thread 10
Sending to client : Thread 10
Sent message : Hello from Thread 10 to Thread 10
Sending to client : Thread 4
Sent message : Hello from Thread 4 to Thread 4
Sending to client : Thread 3
Sent message : Hello from Thread 3 to Thread 3
Sending to client : Thread 1
Sent message : Hello from Thread 1 to Thread 1
Sending to client : Thread 2
Sent message : Hello from Thread 2 to Thread 2
Sending to client : Thread 5
Sent message : Hello from Thread 5 to Thread 5
Sending to client : Thread 6
Sent message : Hello from Thread 6 to Thread 6
Sending to client : Thread 7
Sent message : Hello from Thread 7 to Thread 7
Sending to client : Thread 8
Sent message : Hello from Thread 8 to Thread 8
Sending to client : Thread 9
Sent message : Hello from Thread 9 to Thread 9

At scient side:

Time 0s : Thread 1 sent message to server
Msg sent by Thread 1: Hello from Thread 1
Time 0s : Thread 2 sent message to server
Time 0s : Thread 3 sent message to server
Msg sent by Thread 3: Hello from Thread 3
Msg sent by Thread 2: Hello from Thread 2
Time 0s : Thread 4 sent message to server
Msg sent by Thread 4: Hello from Thread 4
Time 0s : Thread 6 sent message to server
Time 0s : Thread 5 sent message to server
Msg sent by Thread 6: Hello from Thread 6
Msg sent by Thread 5: Hello from Thread 5
Time 0s : Thread 8 sent message to server
Msg sent by Thread 8: Hello from Thread 8
Time 0s : Thread 7 sent message to server
Msg sent by Thread 7: Hello from Thread 7
Time 1s : Thread 9 sent message to server
Msg sent by Thread 9: Hello from Thread 9
Time 2s : Thread 10 sent message to server
Msg sent by Thread 10: Hello from Thread 10
Time 3s : Thread 10 received message from server
Msg received by Thread 10: Hello from Thread 10
Time 4s : Thread 4 received message from server
Msg received by Thread 4: Hello from Thread 4
Time 5s : Thread 3 received message from server
Msg received by Thread 3: Hello from Thread 3
Time 7s : Thread 1 received message from server
Msg received by Thread 1: Hello from Thread 1
Time 7s : Thread 2 received message from server
Msg received by Thread 2: Hello from Thread 2
Time 8s : Thread 5 received message from server
Msg received by Thread 5: Hello from Thread 5
Time 9s : Thread 6 received message from server
Msg received by Thread 6: Hello from Thread 6
Time 10s : Thread 7 received message from server
Msg received by Thread 7: Hello from Thread 7
Time 11s : Thread 8 received message from server
Msg received by Thread 8: Hello from Thread 8
Time 12s : Thread 9 received message from server
Msg received by Thread 9: Hello from Thread 9