Palestine Polytechnic University

College of Computer Engineering and Information Technology

Operating Systems 7505

FINAL PROJECT

Synchronization Simulator

Student: Maria Abu Samoor-201172 **Instructor:** Dr.Radwan Tahboub

Table Of Contents

Project Description

You will write a synchronization simulator to show the different results of concurrent processes / threads accessing shared data variables with and without using synchronization tools.

• Assume a critical section that changes the values of shared variables by adding or subtracting values from these variables from n processes or threads. (See water well example below)

• Your simulator should define the used processes (n1 adders and n2 subtractors) or threads (see example below) and run these processes random number of times. (n1 /adders and n2 / subtractors / removers are also random and not necessarily equal)

• The simulator should show the expected correct values of the shared variables (by calculating the number of times each adder process accesses a variable and the same for subtractors) and compare it to the real resultant values of these variables assuming no synchronization tools were used.

• The simulator should show the expected correct values of the shared variables (by calculating the number of times each adder process accesses a variable and the same for subtractors) and compare it to the real resultant values of these variables assuming synchronization tools were used.

What Is Synchronization ?

Processes Synchronization or Synchronization is the way by which processes that share the same memory space are managed in an operating system. It helps maintain the consistency of data by using variables or hardware so that only one process can make changes to the shared memory at a time. There are various solutions for the same such as semaphores, mutex locks, and synchronization hardware.

What Is Critical Section

critical section refers to a segment of code that is executed by multiple concurrent threads or processes, and which accesses shared resources. These resources may include shared memory, files, or other system resources that can only be accessed by one thread or process at a time to avoid data inconsistency or race conditions.

Codes Explanation

First Code:

In the first code I defined two functions that operate on a shared variable called Size. The variable x is used to store the initial value of Size in order to calculate the expected value later. The square function squares the Size variable after sleeping for 5 seconds. The add function adds 5 to the Size variable. Two threads are created, one running the square function and the other running the add function. The main thread waits for both threads to complete and then prints the expected and actual values of the Size variable. Since there is no synchronization mechanism used, the actual value may differ from the expected value.

In the second code I used a synchronization tool which is Mutex, the code similar to the first one, but uses a lock to synchronize access to the shared variable. The square and add functions acquire the lock before operating on the Size variable and release it afterwards. This ensures that only one thread can modify the Size variable at a time. The main thread waits for both threads to complete and then prints the expected and actual values of the Size variable, which should be the same due to the use of synchronization.

Second Code:

In the first block of code I used Python's threading module to create two threads that operate on a shared variable called size. The increment() function is defined to increment the value of size by 5, three times, with a 4-second delay between increments. The decrement() function is defined to decrement the value of size by 5, three times. Both functions print the value of size before and after the operation is performed. The two threads are started, and then the main thread waits for them to complete before printing the final value of size. Since there is no synchronization, the final value of size is expected to be 50 (the initial value of size) but the actual value may be different due to race conditions.

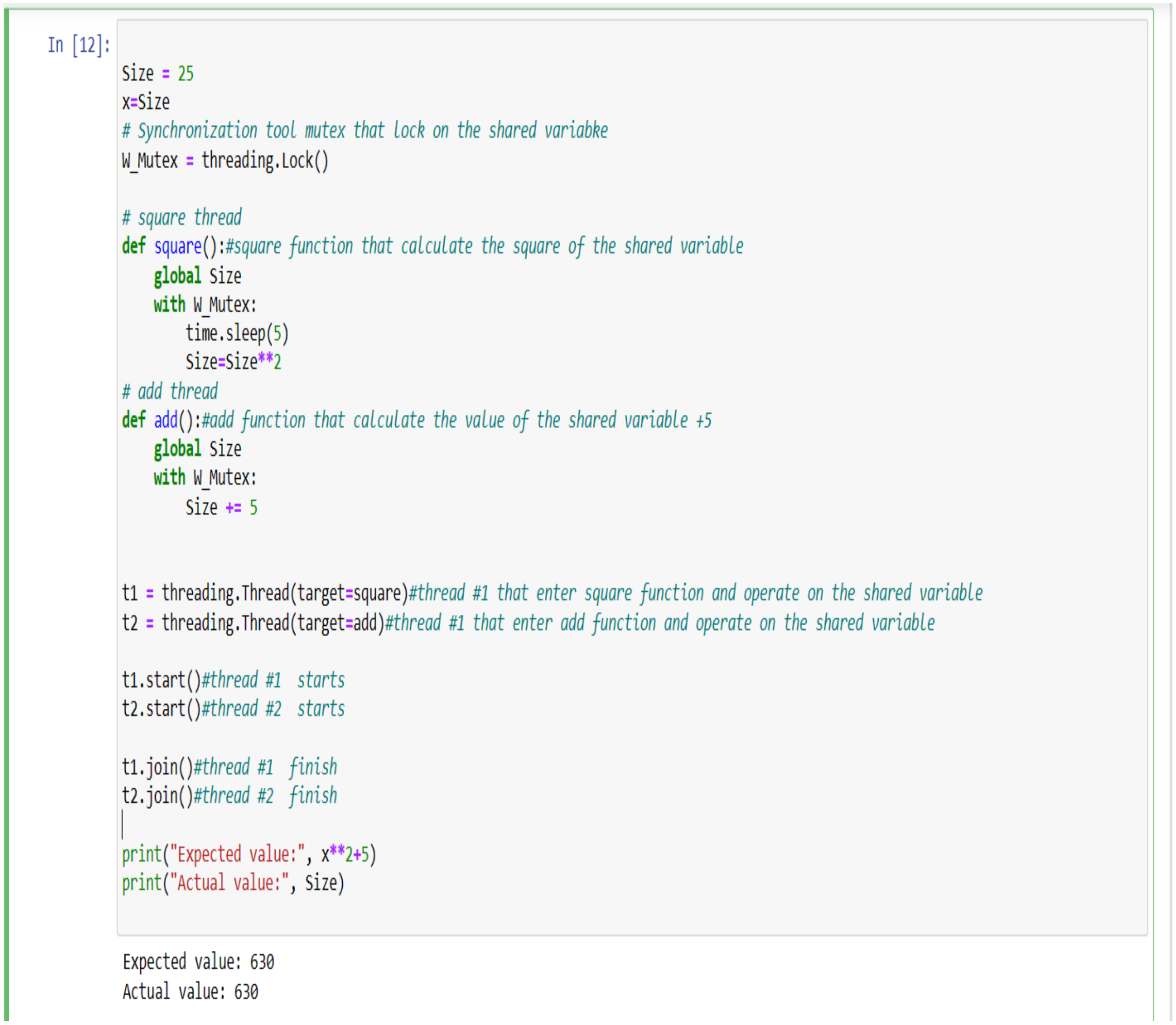
The second block of code is similar to the first, but I used a Semaphore object from the threading module for synchronization. A semaphore is a synchronization object that can be used to control access to a shared resource. The W_Mutex semaphore is initialized with a value of 1, which means only one thread can hold the semaphore at a time. The increment_sync() and decrement_sync() functions are defined to acquire the semaphore before performing their operation on size, and release the semaphore afterwards. This ensures that only one thread can modify size at a time. The two threads are started, and the main thread waits for them to complete before printing the final value of size. With the use of the semaphore, the final value of size is expected to be 50.

Execution Results

Code #1:

Code & Results:

With Synchronization Tools:

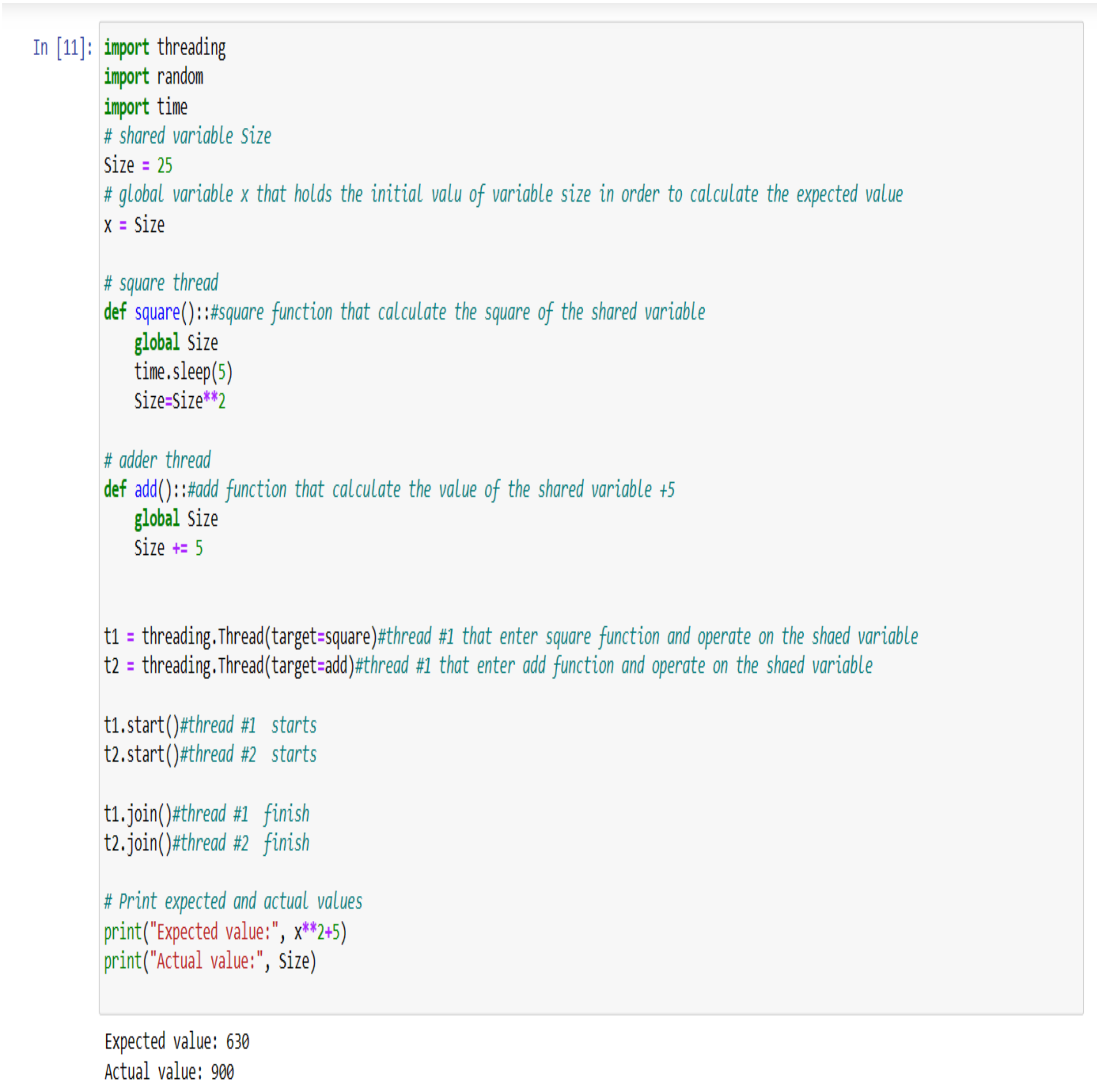


```
In [12]:
Size = 25
x=Size
# Synchronization tool mutex that lock on the shared variabke
W_Mutex = threading.Lock()

# square thread
def square():#square function that calculate the square of the shared variable
    global Size
    with W_Mutex:
        time.sleep(5)
        Size=Size**2
# add thread
def add():#add function that calculate the value of the shared variable +5
    global Size
    with W_Mutex:
        Size += 5


t1 = threading.Thread(target=square)#thread #1 that enter square function and operate on the shared variable
t2 = threading.Thread(target=add)#thread #1 that enter add function and operate on the shared variable

t1.start()#thread #1  starts
t2.start()#thread #2  starts

t1.join()#thread #1  finish
t2.join()#thread #2  finish

print("Expected value:", x**2+5)
print("Actual value:", Size)
```

```
Expected value: 630
Actual value: 630
```

Without Synchronization Tools:

Code & Results:



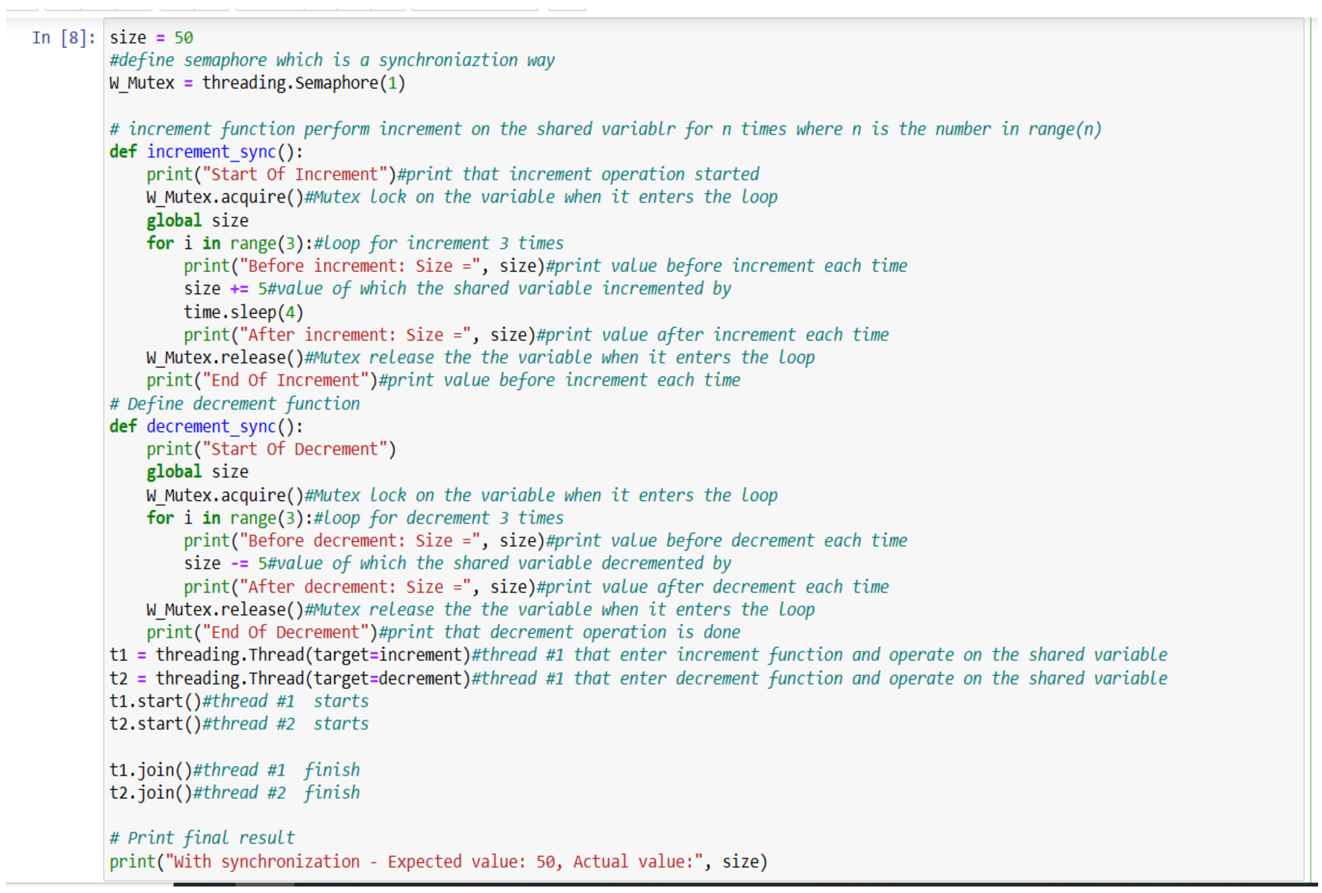
```
In [11]: import threading
         import random
         import time
         # shared variable Size
         Size = 25
         # global variable x that holds the initial valu of variable size in order to calculate the expected value
         x = Size

         # square thread
         def square()::#square function that calculate the square of the shared variable
             global Size
             time.sleep(5)
             Size=Size**2

         # adder thread
         def add()::#add function that calculate the value of the shared variable +5
             global Size
             Size += 5


         t1 = threading.Thread(target=square)#thread #1 that enter square function and operate on the shaed variable
         t2 = threading.Thread(target=add)#thread #1 that enter add function and operate on the shaed variable

         t1.start()#thread #1  starts
         t2.start()#thread #2  starts

         t1.join()#thread #1  finish
         t2.join()#thread #2  finish

         # Print expected and actual values
         print("Expected value:", x**2+5)
         print("Actual value:", Size)
```

```
Expected value: 630
Actual value: 900
```

Code #2:

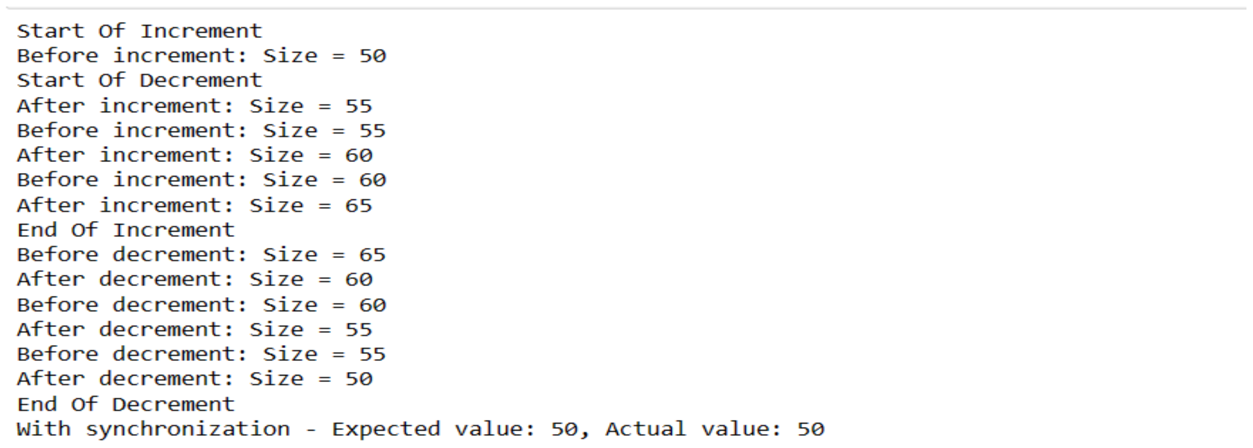
With Synchronization Tools:

Code:



```
In [8]: size = 50
        #define semaphore which is a synchroniaztion way
        W_Mutex = threading.Semaphore(1)

        # increment function perform increment on the shared variablr for n times where n is the number in range(n)
        def increment_sync():
            print("Start Of Increment")#print that increment operation started
            W_Mutex.acquire()#Mutex lock on the variable when it enters the loop
            global size
            for i in range(3):#loop for increment 3 times
                print("Before increment: Size =", size)#print value before increment each time
                size += 5#value of which the shared variable incremented by
                time.sleep(4)
                print("After increment: Size =", size)#print value after increment each time
            W_Mutex.release()#Mutex release the the variable when it enters the loop
            print("End Of Increment")#print value before increment each time
        # Define decrement function
        def decrement_sync():
            print("Start Of Decrement")
            global size
            W_Mutex.acquire()#Mutex lock on the variable when it enters the loop
            for i in range(3):#loop for decrement 3 times
                print("Before decrement: Size =", size)#print value before decrement each time
                size -= 5#value of which the shared variable decremented by
                print("After decrement: Size =", size)#print value after decrement each time
            W_Mutex.release()#Mutex release the the variable when it enters the loop
            print("End Of Decrement")#print that decrement operation is done
        t1 = threading.Thread(target=increment)#thread #1 that enter increment function and operate on the shared variable
        t2 = threading.Thread(target=decrement)#thread #1 that enter decrement function and operate on the shared variable
        t1.start()#thread #1  starts
        t2.start()#thread #2  starts

        t1.join()#thread #1  finish
        t2.join()#thread #2  finish

        # Print final result
        print("With synchronization - Expected value: 50, Actual value:", size)
```

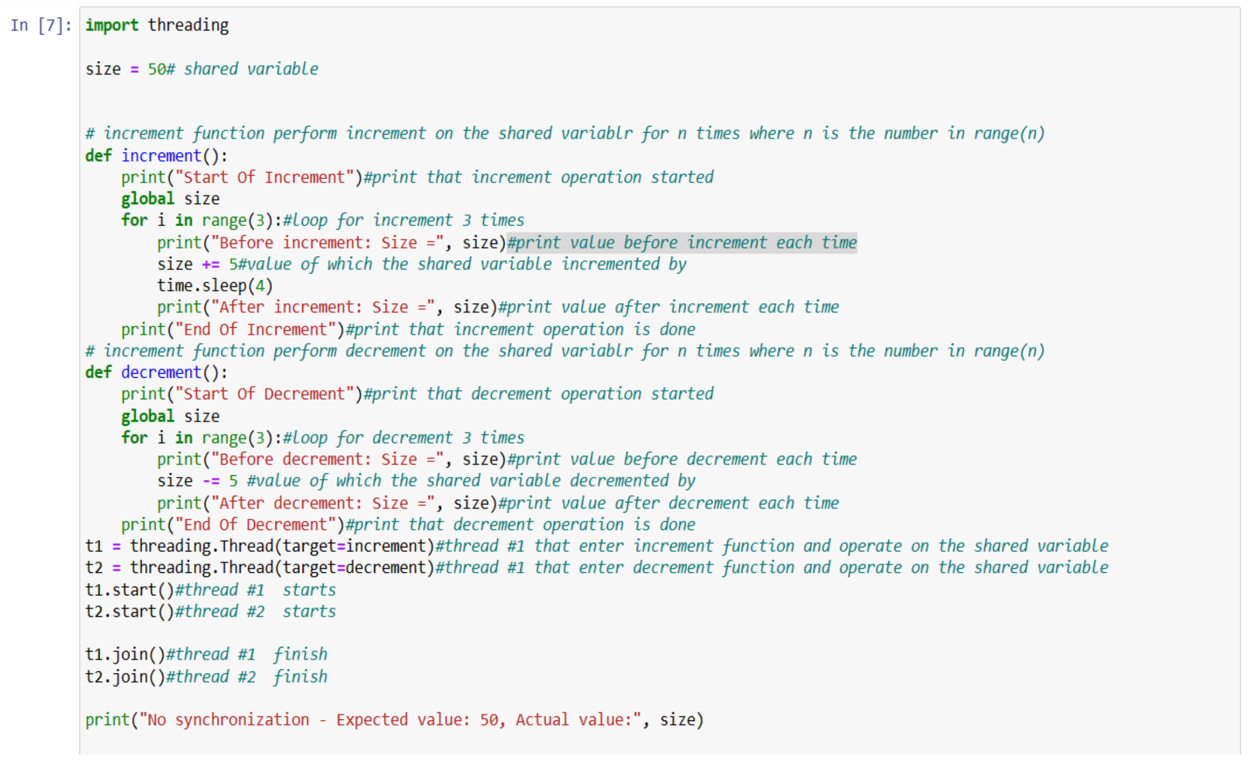
Results:



```
Start Of Increment
Before increment: Size = 50
Start Of Decrement
After increment: Size = 55
Before increment: Size = 55
After increment: Size = 60
Before increment: Size = 60
After increment: Size = 65
End Of Increment
Before decrement: Size = 65
After decrement: Size = 60
Before decrement: Size = 60
After decrement: Size = 55
Before decrement: Size = 55
After decrement: Size = 50
End Of Decrement
With synchronization - Expected value: 50, Actual value: 50
```

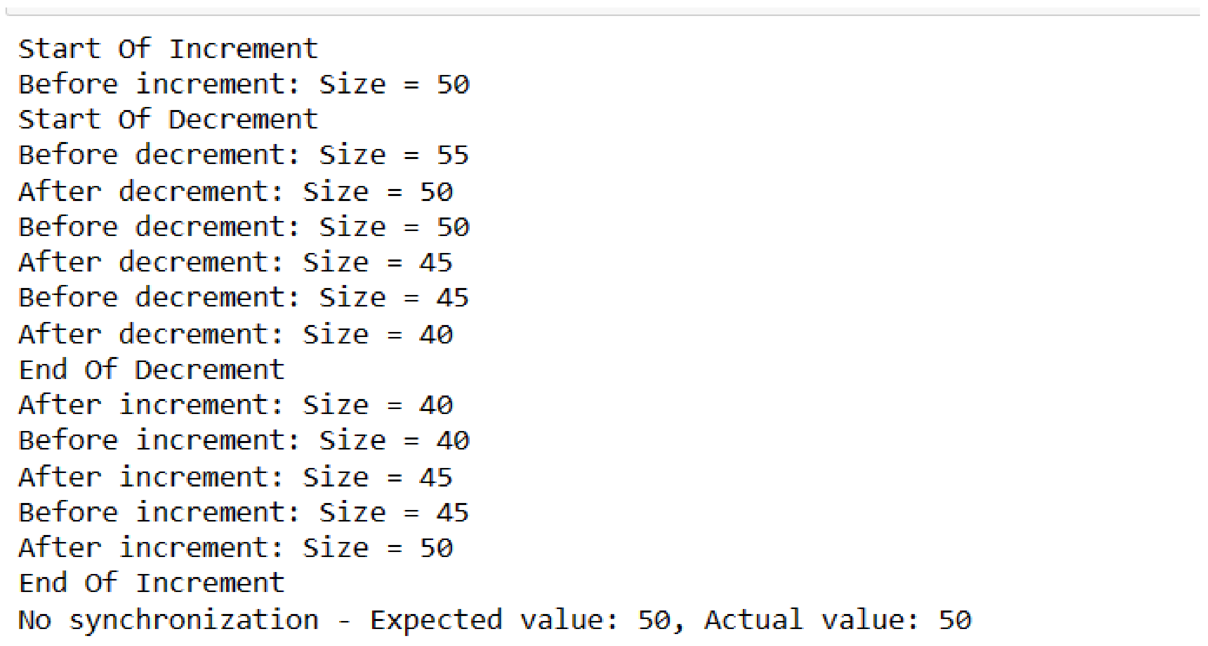
Without Synchronization Tools:

Code:



```
In [7]: import threading

        size = 50# shared variable


        # increment function perform increment on the shared variablr for n times where n is the number in range(n)
        def increment():
            print("Start Of Increment")#print that increment operation started
            global size
            for i in range(3):#loop for increment 3 times
                print("Before increment: Size =", size)#print value before increment each time
                size += 5#value of which the shared variable incremented by
                time.sleep(4)
                print("After increment: Size =", size)#print value after increment each time
            print("End Of Increment")#print that increment operation is done
        # increment function perform decrement on the shared variablr for n times where n is the number in range(n)
        def decrement():
            print("Start Of Decrement")#print that decrement operation started
            global size
            for i in range(3):#loop for decrement 3 times
                print("Before decrement: Size =", size)#print value before decrement each time
                size -= 5 #value of which the shared variable decremented by
                print("After decrement: Size =", size)#print value after decrement each time
            print("End Of Decrement")#print that decrement operation is done
        t1 = threading.Thread(target=increment)#thread #1 that enter increment function and operate on the shared variable
        t2 = threading.Thread(target=decrement)#thread #1 that enter decrement function and operate on the shared variable
        t1.start()#thread #1  starts
        t2.start()#thread #2  starts

        t1.join()#thread #1  finish
        t2.join()#thread #2  finish

        print("No synchronization - Expected value: 50, Actual value:", size)
```

Results:



```
Start Of Increment
Before increment: Size = 50
Start Of Decrement
Before decrement: Size = 55
After decrement: Size = 50
Before decrement: Size = 50
After decrement: Size = 45
Before decrement: Size = 45
After decrement: Size = 40
End Of Decrement
After increment: Size = 40
Before increment: Size = 40
After increment: Size = 45
Before increment: Size = 45
After increment: Size = 50
End Of Increment
No synchronization - Expected value: 50, Actual value: 50
```