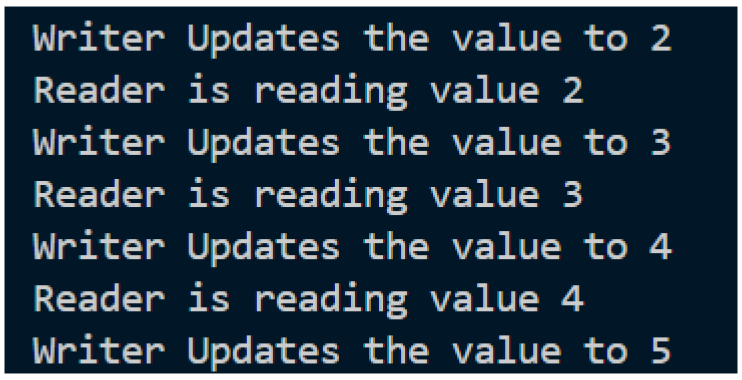
Process Synchronization

Lock Variable

Code:

```
import threading
import time

count = 1
lock = 0

def reader():

        while True :
                global lock
                global count

                while lock == 1 : pass
                lock = 1
                print(f'Reader is reading value {count}')
                lock = 0
                time.sleep(1)



def writer():

        while True:
                global lock
                global count

                while lock == 1 : pass
                lock = 1
                count+=1
                print(f'Writer Updates the value to {count}')
                lock = 0
                time.sleep(1)


reader1 = threading.Thread(target=reader)
writer1 = threading.Thread(target=writer)

writer1.start()
reader1.start()

reader1.join()
writer1.join()
```

Output :



```
Writer Updates the value to 2
Reader is reading value 2
Writer Updates the value to 3
Reader is reading value 3
Writer Updates the value to 4
Reader is reading value 4
Writer Updates the value to 5
```

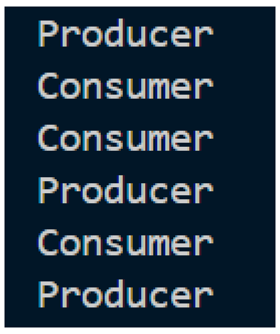
Turn Variable

Code:

```
import threading
import time

N = 2
flag = [False] * N
turn = 0


def producer():

    while True :

        global turn
        global flag

        while flag[1] and turn==1:
            pass

        print('Producer')
        time.sleep(1)

        flag[0] = False
        turn = 1
        flag[1]=True

def consumer():

    while True:

        global turn
        global flag

        while flag[0] and turn==0:
            pass

        print('Consumer')
        time.sleep(1)

        flag[1] = False
        turn = 0
        flag[0] =  True
```

```
producer_thread = threading.Thread(target=producer)
consumer_thread = threading.Thread(target=consumer)

producer_thread.start()
consumer_thread.start()

producer_thread.join()
consumer_thread.join()
```

Output:



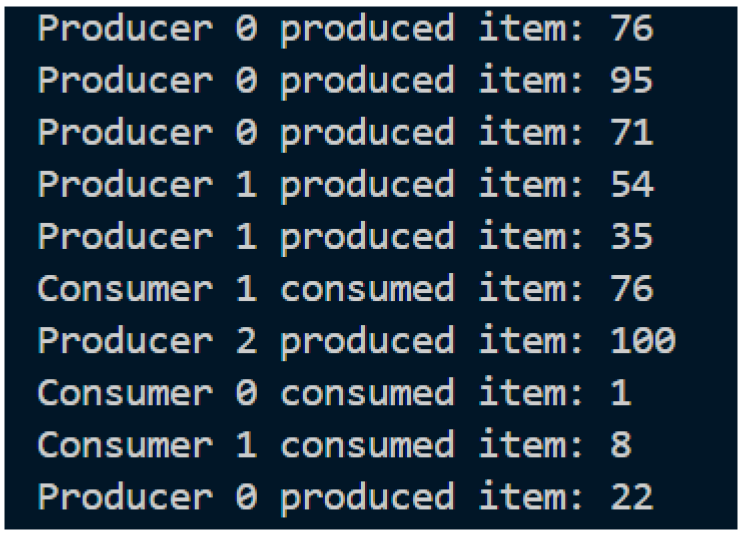
```
Producer
Consumer
Consumer
Producer
Consumer
Producer
```

Bounded Buffer

Code:

```
import queue
import threading
import time
import random

class BoundedBuffer:
    def __init__(self, max_size):
        self.buffer = queue.Queue(max_size)
        self.max_size = max_size

    def put(self, item):
        self.buffer.put(item)

    def get(self):
        return self.buffer.get()

def producer(buffer, producer_id):
    while True:
        item = random.randint(1, 100)
        buffer.put(item)
        print(f"Producer {producer_id} produced item: {item}")
        time.sleep(random.random())

def consumer(buffer, consumer_id):
    while True:
        item = buffer.get()
        print(f"Consumer {consumer_id} consumed item: {item}")
        time.sleep(random.random())

if __name__ == "__main__":

    buffer_size = 10
    buffer = BoundedBuffer(buffer_size)

    num_producers = 3
    num_consumers = 2

    producer_threads = []
    for i in range(num_producers):
        thread = threading.Thread(target=producer, args=(buffer, i))
        thread.start()
        producer_threads.append(thread)

    consumer_threads = []
    for i in range(num_consumers):
        thread = threading.Thread(target=consumer, args=(buffer, i))
        thread.start()
        consumer_threads.append(thread)

    for thread in producer_threads + consumer_threads:
        thread.join()
```

Output:



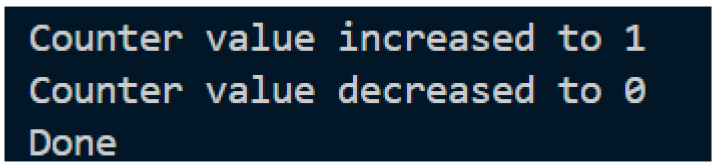
```
Producer 0 produced item: 76
Producer 0 produced item: 95
Producer 0 produced item: 71
Producer 1 produced item: 54
Producer 1 produced item: 35
Consumer 1 consumed item: 76
Producer 2 produced item: 100
Consumer 0 consumed item: 1
Consumer 1 consumed item: 8
Producer 0 produced item: 22
```

Binary Semaphore:

Code:

```
import threading
import time

class BinarySemaphore:
    def __init__(self):
        self.value = True

    def acquire(self):
        while not self.value:
            pass
        self.value = False
    def release(self):
        self.value = True


counter = 0
lock = BinarySemaphore()


def increment_counter():
    global counter
    lock.acquire()
    try:
        counter += 1
        print(f"Counter value increased to {counter}")
        time.sleep(1)
    finally:
        lock.release()


def decrement_counter():
    global counter
    lock.acquire()
    try:
        counter -=1
        print(f"Counter value decreased to {counter}")
        time.sleep(1)
    finally:
        lock.release()



threads = []
threads.append(threading.Thread(target=increment_counter))
threads.append(threading.Thread(target=decrement_counter))


for thread in threads:
    thread.start()


for thread in threads:
    thread.join()

print('Done')
```

Output:



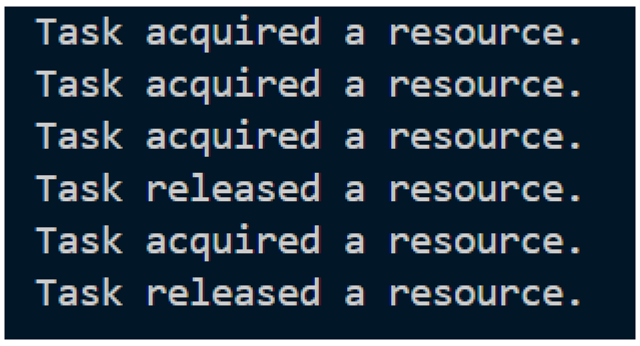
```
Counter value increased to 1
Counter value decreased to 0
Done
```

Counting Semaphore:

Code:

```
import threading
import time

class CountingSemaphore:
    def __init__(self, initial_value):
        self.value = initial_value

    def acquire(self):
        while self.value <= 0:
            pass
        self.value -= 1

    def release(self):
        self.value += 1


resource_pool_size = 3
resource_pool = CountingSemaphore(resource_pool_size)


def task():
    resource_pool.acquire()
    try:
        print("Task acquired a resource.")
        time.sleep(1)
    finally:
        resource_pool.release()
        print("Task released a resource.\n")




num_tasks = 5
threads = []
for _ in range(num_tasks):
    thread = threading.Thread(target=task)
    threads.append(thread)


for thread in threads:
    thread.start()

for thread in threads:
    thread.join()

print("All tasks finished.")
```

Output:



```
Task acquired a resource.
Task acquired a resource.
Task acquired a resource.
Task released a resource.
Task acquired a resource.
Task released a resource.
```