```
class BankersAlgorithm:
    def __init__(self, num_processes, num_resources):
        self.num_processes = num_processes
        self.num_resources = num_resources
        self.available = [0] * num_resources
        self.maximum = [[0] * num_resources for _ in range(num_processes)]
        self.allocation = [[0] * num_resources for _ in range(num_processes)]
        self.need = [[0] * num_resources for _ in range(num_processes)]

    def input_resources(self):
        print("\nEnter the available instances of each resource:")
        for i in range(self.num_resources):
            self.available[i] = int(input(f"Resource {i}: "))

        print("\nEnter the maximum demand of each process for each resource:")
        for i in range(self.num_processes):
            print(f"Process {i}:")
            for j in range(self.num_resources):
                self.maximum[i][j] = int(input(f"Maximum demand for Resource {j}: "))

        print("\nEnter the current allocation of resources to each process:")
        for i in range(self.num_processes):
            print(f"Process {i}:")
            for j in range(self.num_resources):
                self.allocation[i][j] = int(input(f"Allocated for Resource {j}: "))

        for i in range(self.num_processes):
            for j in range(self.num_resources):
                self.need[i][j] = self.maximum[i][j] - self.allocation[i][j]

    def is_safe(self):
        work = self.available[:]
        finish = [False] * self.num_processes
        safe_sequence = []

        while len(safe_sequence) < self.num_processes:
            progress_made = False
            for i in range(self.num_processes):
                if not finish[i] and all(self.need[i][j] <= work[j] for j in range(self.num_resources)):
                    for j in range(self.num_resources):
                        work[j] += self.allocation[i][j]
                    safe_sequence.append(i)
                    finish[i] = True
                    progress_made = True
                    break
            if not progress_made:
                return False, []

        return True, safe_sequence
```

```
def request_resources(self, process_id, request):
        for i in range(self.num_resources):
            if request[i] > self.need[process_id][i]:
                print(f"Error: Process {process_id} has exceeded its maximum claim.")
                return False

        for i in range(self.num_resources):
            if request[i] > self.available[i]:
                print(f"Resources not available for process {process_id}.")
                return False

        for i in range(self.num_resources):
            self.available[i] -= request[i]
            self.allocation[process_id][i] += request[i]
            self.need[process_id][i] -= request[i]

        is_safe, safe_sequence = self.is_safe()

        if is_safe:
            print(f"Resources allocated to process {process_id}. Safe sequence: {safe_sequence}")
            return True
        else:
            for i in range(self.num_resources):
                self.available[i] += request[i]
                self.allocation[process_id][i] -= request[i]
                self.need[process_id][i] += request[i]
            print("System is not in a safe state. Request denied.")
            return False


def main():
    num_processes = int(input("Enter the number of processes: "))
    num_resources = int(input("Enter the number of resource types: "))

    bankers_algo = BankersAlgorithm(num_processes, num_resources)
    bankers_algo.input_resources()

    process_id = int(input("\nEnter the process ID that is requesting resources: "))
    request = []
    print("Enter the request for resources (one number per resource type):")
    for i in range(num_resources):
        request.append(int(input(f"Resource {i}: ")))

    bankers_algo.request_resources(process_id, request)


if __name__ == "__main__":
    main()
```

```
import threading
import time
import random

class Philosopher(threading.Thread):
    def __init__(self, id, left_fork, right_fork):
        threading.Thread.__init__(self)
        self.id = id
        self.left_fork = left_fork
        self.right_fork = right_fork

    def run(self):
        while True:
            self.think()
            self.dine()

    def think(self):
        print(f"Philosopher {self.id} is thinking.")
        time.sleep(random.uniform(1, 3))

    def dine(self):
        self.pick_up_forks()
        print(f"Philosopher {self.id} is eating.")
        time.sleep(random.uniform(1, 2))
        self.put_down_forks()

    def pick_up_forks(self):
        print(f"Philosopher {self.id} is trying to pick up forks.")
        self.left_fork.acquire()
        print(f"Philosopher {self.id} picked up left fork.")
        self.right_fork.acquire()
        print(f"Philosopher {self.id} picked up right fork.")

    def put_down_forks(self):
        self.left_fork.release()
        print(f"Philosopher {self.id} put down left fork.")
        self.right_fork.release()
        print(f"Philosopher {self.id} put down right fork.")

def main():
    num_philosophers = 5
    forks = [threading.Lock() for _ in range(num_philosophers)]
    philosophers = []
    for i in range(num_philosophers):
        left_fork = forks[i]
        right_fork = forks[(i + 1) % num_philosophers]
        philosopher = Philosopher(i, left_fork, right_fork)
        philosophers.append(philosopher)
    for philosopher in philosophers:
        philosopher.start()
    time.sleep(10)
    for philosopher in philosophers:
        philosopher.join()

if __name__ == "__main__":
    main()
```

```
import threading
import time
import random

class Buffer:
    def __init__(self, size):
        self.size = size
        self.buffer = []
        self.lock = threading.Lock()
        self.empty = threading.Condition(self.lock)
        self.full = threading.Condition(self.lock)

    def add(self, item):
        with self.lock:
            while len(self.buffer) == self.size:
                self.empty.wait()
            self.buffer.append(item)
            print(f"Produced: {item}")
            self.full.notify()

    def remove(self):
        with self.lock:
            while len(self.buffer) == 0:
                self.full.wait()
            item = self.buffer.pop(0)
            print(f"Consumed: {item}")
            self.empty.notify()
            return item

class Producer(threading.Thread):
    def __init__(self, buffer):
        threading.Thread.__init__(self)
        self.buffer = buffer

    def run(self):
        while True:
            item = random.randint(1, 100)
            self.buffer.add(item)
            time.sleep(random.uniform(0.5, 1.5))

class Consumer(threading.Thread):
    def __init__(self, buffer):
        threading.Thread.__init__(self)
        self.buffer = buffer

    def run(self):
        while True:
            item = self.buffer.remove()
            time.sleep(random.uniform(1, 2))

def main():
    buffer_size = 5
    buffer = Buffer(buffer_size)

    producer = Producer(buffer)
    consumer = Consumer(buffer)

    producer.start()
    consumer.start()

    producer.join()
    consumer.join()

if __name__ == "__main__":
    main()
```