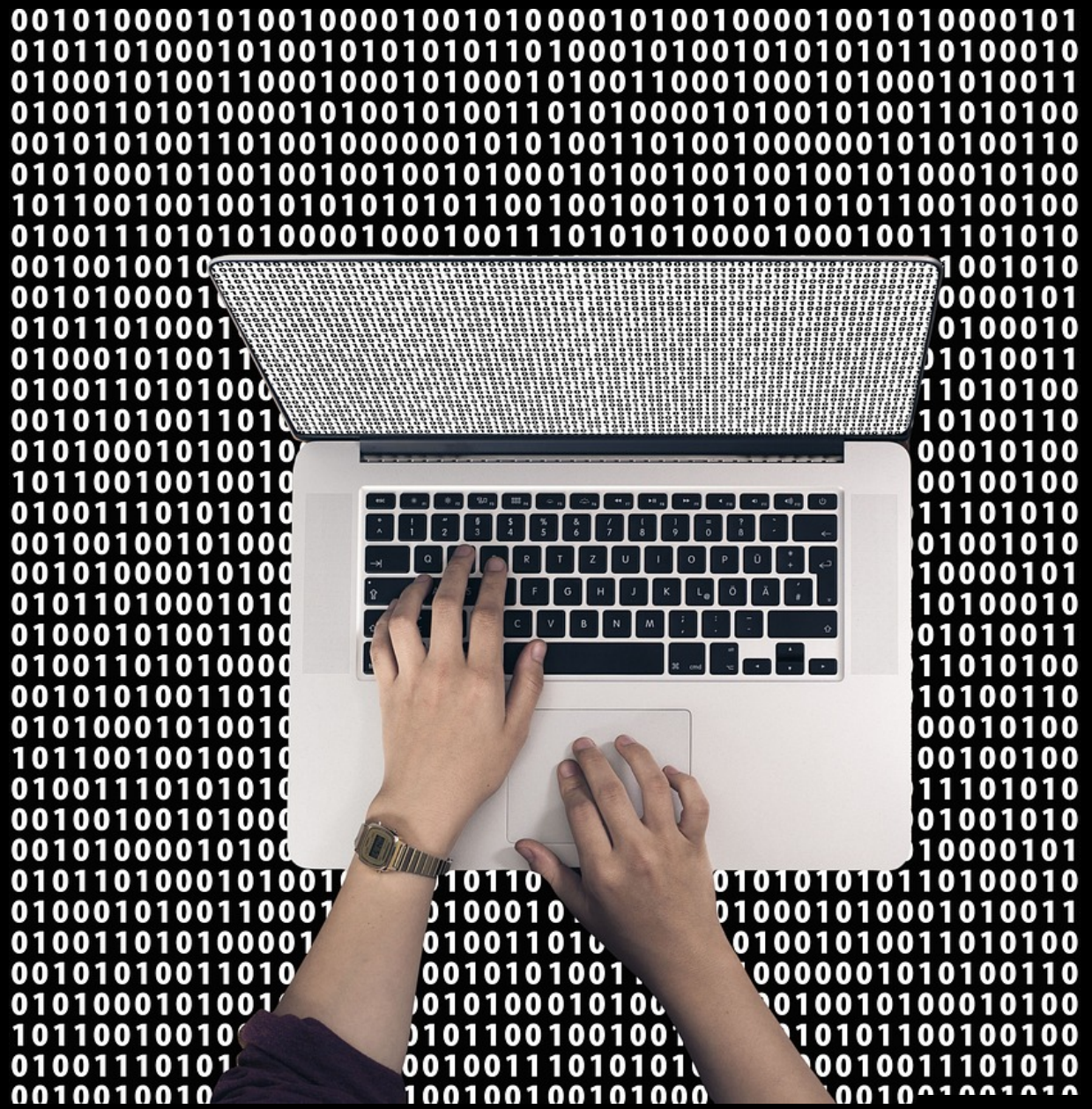
Operating System 2

Bounded Buffer



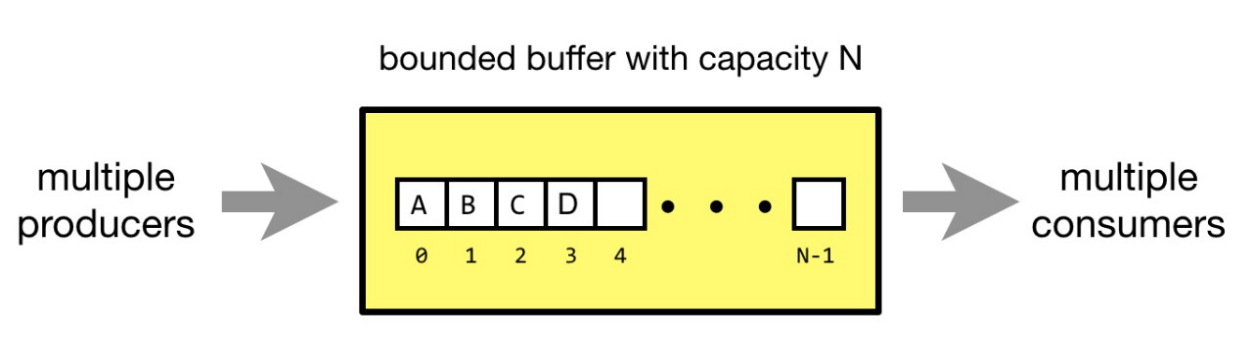
MADE WITH
beautiful.ai

Project Overview

Real World ApplicationTickets system allows user to create an account as a vendor or a customerA Vendor is able to add tickets (sell, produce) to the system and initialize its quantity.A Customer is able to buy tickets (remove , consume) from the system with any desired quantity

MADE WITH
beautiful.ai

Bounded Buffer



Bounded buffer problem, which is also called producer consumer problem, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.Bounded buffer problem, which is also called producer consumer problem, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

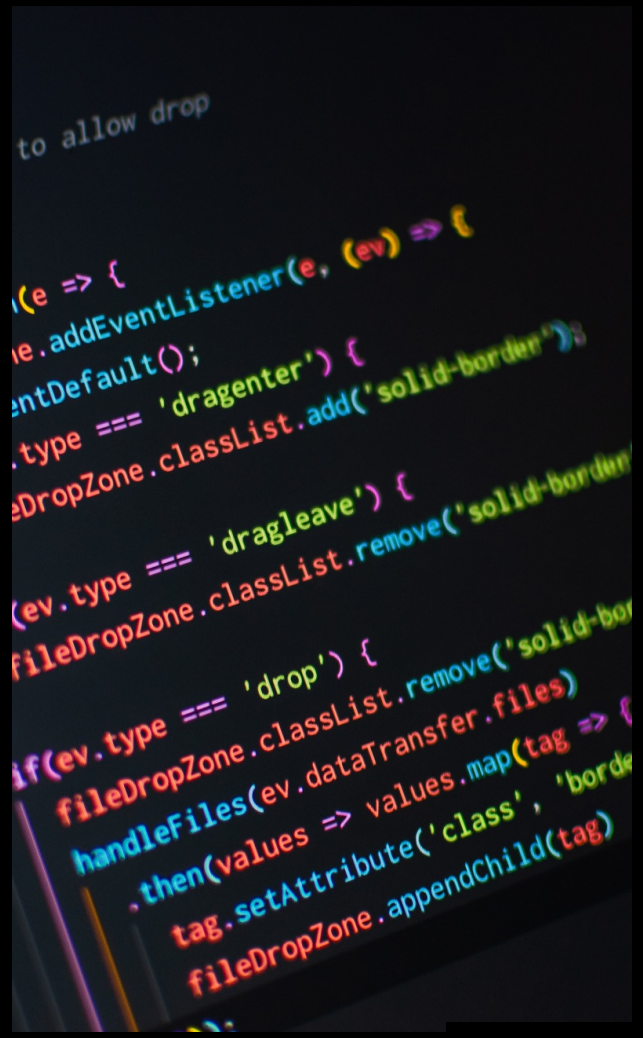
MADE WITH
beautiful.ai

Bounded Buffer Problem

Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

We need a way to make the producer and consumer work in an independent manner.



MADE WITH
beautiful.ai

Bounded Buffer Solutions

One solution of this problem is to use semaphores. The semaphores which will be used here are:
At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

- The First Solution

  Mutex ,a binary semaphore which is used to acquire and release the lock.

- The Second Solution

  Empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.

- The Third Solution

  Full, a counting semaphore whose initial value is 0

MADE WITH
beautiful.ai

Producer

Producer Operation:

The pseudocode of the producer function looks like this:

```
do {
  wait(empty); // wait until empty>0 and then decrement 'empty'
  wait(mutex); // acquire lock
  /* perform the
  insert operation in a slot */
  signal(mutex); // release lock
  signal(full);  // increment 'full'
} while(TRUE)
```

1 Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.

2 Then it decrements the empty semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.

3 Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.

4 After performing the insert operation, the lock is released and the value of full is incremented because the producer has just filled a slot in the buffer.

MADE WITH beautiful.ai

Consumer

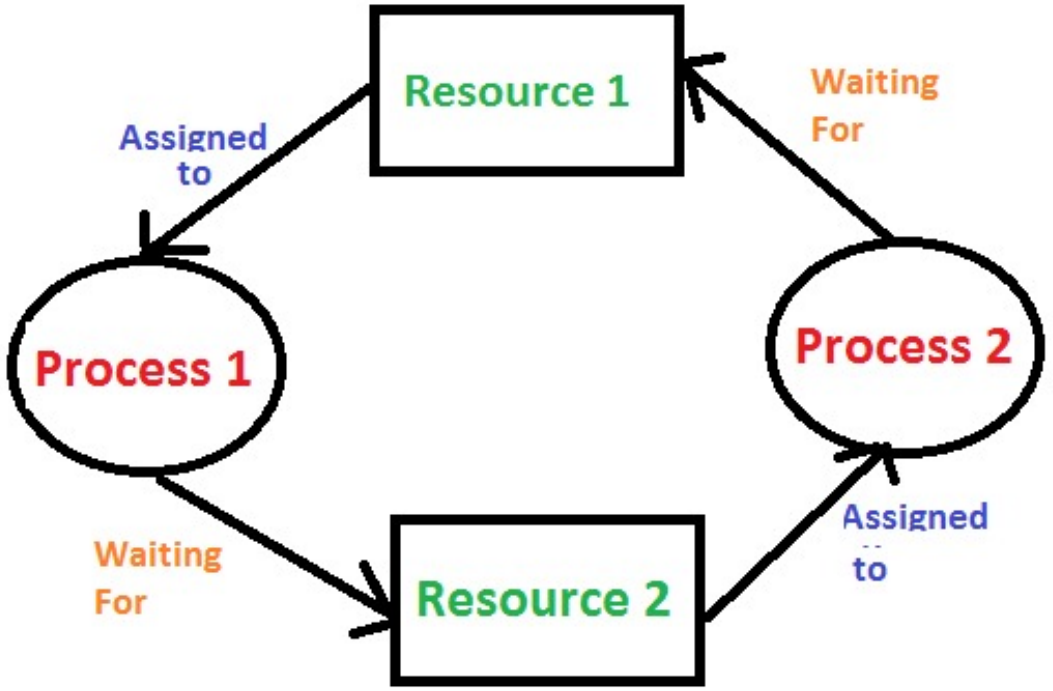
Consumer Operation:

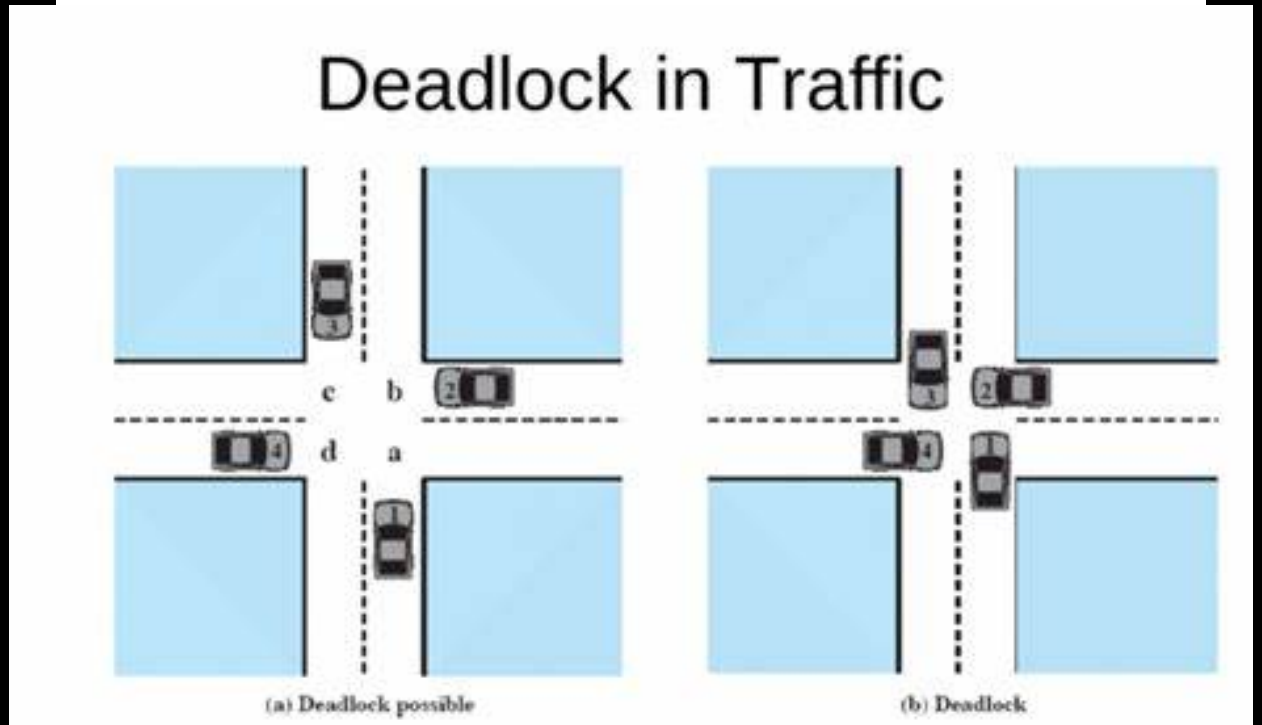
The pseudocode of the consumer function looks like this:

```
do {
  wait(full); // wait until full>0 and then decrement 'full'
  wait(mutex);  // acquire the lock
  /* perform the remove operation
   in a slot */
  signal(mutex); // release the lock
  signal(empty); // increment 'empty'
} while(TRUE);
```

1. **The consumer waits until there is atleast one full slot in the buffer.**
2. **Then it decrements the full semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.**
3. **After that, the consumer acquires lock on the buffer.**
4. **Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.**
5. **Then, the consumer releases the lock.**
6. Finally, the empty semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

MADE WITH
beautiful.ai





Deadlocks

Deadlock is a situation that occurs in OS when any process enters a waiting state because another waiting process is holding the demanded resource. Deadlock is a common problem in multi-processing where several processes share a specific type of mutually exclusive resource known as a soft lock or software.

MADE WITH
beautiful.ai

Deadlocks Examples

1 Dining Philosophers Problem

This is a classic example of a deadlock situation. It involves five philosophers seated around a circular table, each with a fork in front of them. In order to eat, a philosopher must have two forks. However, if all philosophers try to pick up the fork to their left at the same time, they will all be waiting for the fork to their right to be released, and the deadlock will occur.

2 Resource Allocation Deadlock

This type of deadlock occurs when two or more processes request a resource that is currently being used by another process, and the processes are waiting for the resource to be released. For example, if process A is using resource X and process B is using resource Y, and then process A tries to acquire resource Y and process B tries to acquire resource X, a deadlock will occur.

3 Deadlock due to Priority Inversion

This type of deadlock occurs when a low-priority process holds a resource that a high-priority process needs, and the high-priority process is waiting for the resource. The low-priority process may not release the resource until it finishes its task, causing the high-priority process to wait indefinitely.

MADE WITH
beautiful.ai

Strategies to resolve deadlocks

1 | Lock Hierarchy

One way to prevent deadlocks is to establish a lock hierarchy, where locks are always acquired in the same order. This ensures that processes do not try to acquire locks in a different order, which could lead to a deadlock.

2 | Timeouts

Another strategy is to use timeouts, where processes are allowed to hold a lock for a certain amount of time before they are required to release it. This allows other processes to acquire the lock if it is not being used, and can help to prevent deadlocks.

3 | Deadlock Detection

Some systems use algorithms to detect deadlocks and resolve them by releasing the resources held by one or more of the processes involved in the deadlock.

4 | Resource Allocation Graph

One way to visualize and prevent deadlocks is to use a resource allocation graph, which represents the resources and processes as nodes and the resource dependencies as edges. This can help to identify potential deadlock situations and resolve them before they occur.

5 | Prevention

Deadlocks can be prevented by designing processes and resource allocation strategies in a way that avoids circular waits and ensures that resources are always released in a timely manner.

MADE WITH
beautiful.ai

Deadlock Implementation

Lock Hierarchy

```
2 usages
static final Object lock1 = new Object();
2 usages
static final Object lock2 = new Object();
public static void main(String[] args) {
    Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
            synchronized (lock1) {
                System.out.println("Thread 1: Holding lock 1...");
                try {Thread.sleep( millis: 10);} catch (InterruptedException ignored) {}
                System.out.println("Thread 1: Waiting for lock 2...");
                synchronized (lock2) {System.out.println("Thread 1: Holding lock 1 & 2...");}}}});
    Thread t2 = new Thread(new Runnable() {
        @Override
        public void run() {
            synchronized (lock2) {
                System.out.println("Thread 2: Holding lock 2...");
                try {Thread.sleep( millis: 10);} catch (InterruptedException ignored) {}
                System.out.println("Thread 2: Waiting for lock 1...");
                synchronized (lock1) {
                    System.out.println("Thread 2: Holding lock 1 & 2...");
                }}}});
    t1.start();
```

One way to prevent deadlocks is to establish a lock hierarchy, where locks are always acquired in the same order. This ensures that threads do not try to acquire locks in a different order, which could lead to a deadlock.

MADE WITH
beautiful.ai

Starvation

In computer science, starvation refers to a situation in which a process or thread is unable to gain access to the resources it needs to complete its task. This can happen when a process or thread is repeatedly denied access to a resource that it needs, or when it is given a lower priority than other processes or threads.

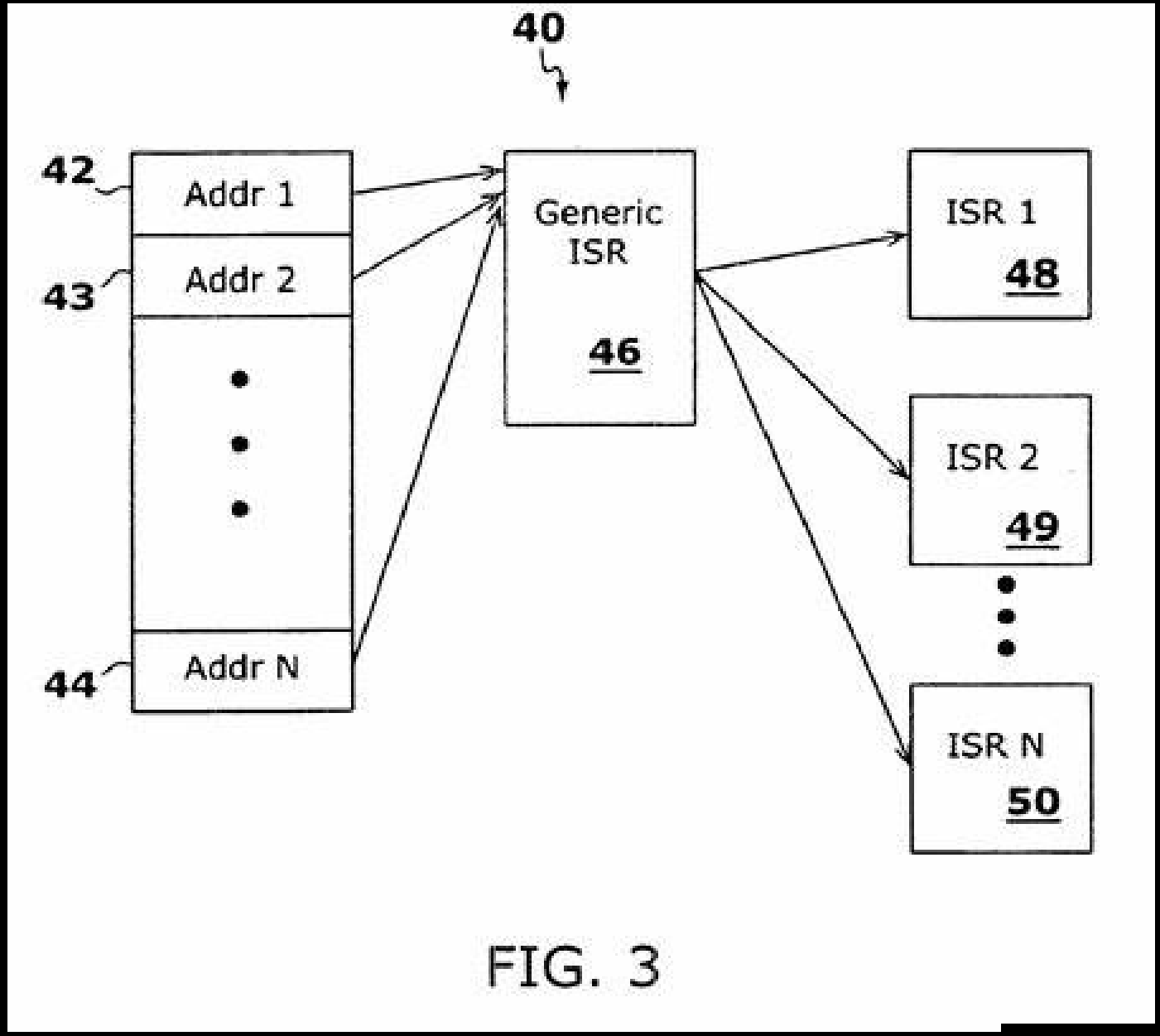


FIG. 3

MADE WITH beautiful.ai

Examples Of Starvation

- ## Priority Inversion

  This occurs when a low-priority process holds a resource that a high-priority process needs, causing the high-priority process to wait indefinitely.

- ## Resource Contention

  This occurs when multiple processes or threads are competing for the same resource, and one or more of them are consistently denied access to the resource.

- ## Unbounded Blocking

  If a process or thread is blocked indefinitely, it can cause other processes or threads to starve for resources.

- ## Fair Scheduling

  If a scheduling algorithm is not fair, it can cause some processes or threads to starve while others receive a disproportionate share of resources.

- ## Infinite Loops

  A process or thread that is stuck in an infinite loop can cause other processes or threads to starve for resources.

MADE WITH
beautiful.ai

Strategies to Resolve Starvation

```
new *
Thread t1 = new Thread(new Runnable() {
    new *
    @Override
    public void run() {
        // High-priority thread code goes here
    }
});

t1.setPriority(Thread.MAX_PRIORITY);
t1.start();
```

Priority Scheduling

One way to prevent starvation is to use a priority scheduling algorithm, where processes or threads with higher priorities are given a higher share of resources. This can be implemented using the java.lang.Thread class in Java. For example:

MADE WITH
beautiful.ai

Strategies to Resolve Starvation

- ## Resource Allocation

  Another way to prevent starvation is to ensure that resources are allocated fairly among processes or threads. This can be implemented using a resource manager or lock manager that tracks resource usage and ensures that all processes or threads have a fair opportunity to access resources.

- ## Deadlock Prevention

  Deadlocks can cause starvation, as processes or threads that are involved in a deadlock may be unable to proceed until the deadlock is resolved. Strategies such as lock hierarchy and timeouts can be used to prevent deadlocks.

- ## Time Sharing

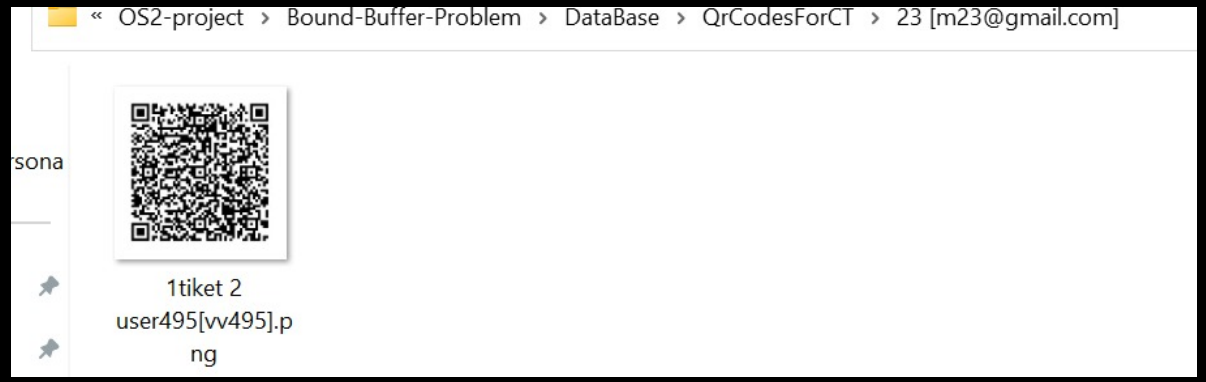
  If a process or thread is using an excessive amount of resources, it can cause other processes or threads to starve. Implementing a time sharing system, where processes or threads are given a fixed amount of time to use resources before they must yield to other processes or threads, can help to prevent this type of starvation.

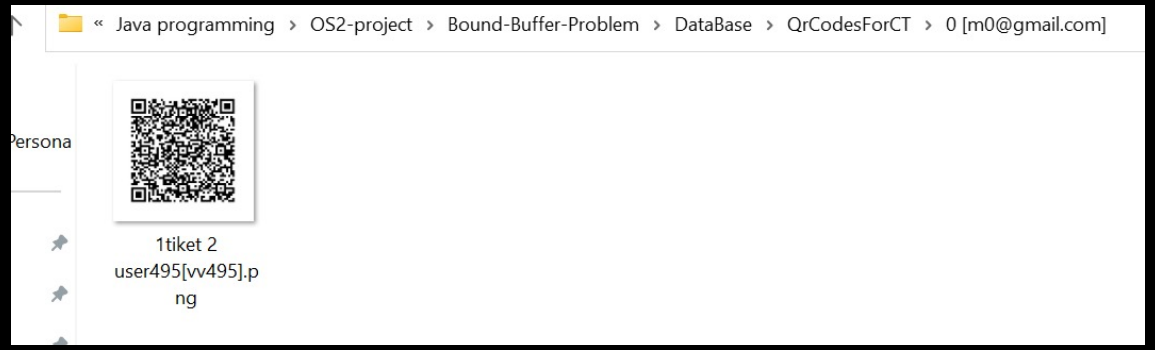
- ## Fair Queues

  If a queue is used to store tasks or requests, using a fair queue implementation can help to prevent starvation. A fair queue ensures that tasks or requests are processed in the order in which they are received, rather than allowing some tasks or requests to be processed repeatedly while others are ignored.

MADE WITH
beautiful.ai

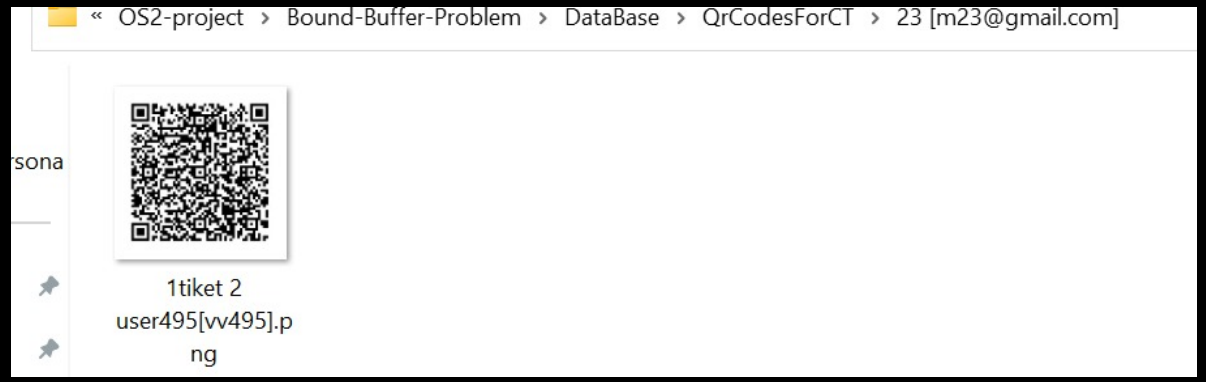
Mutual Exclusion Problem

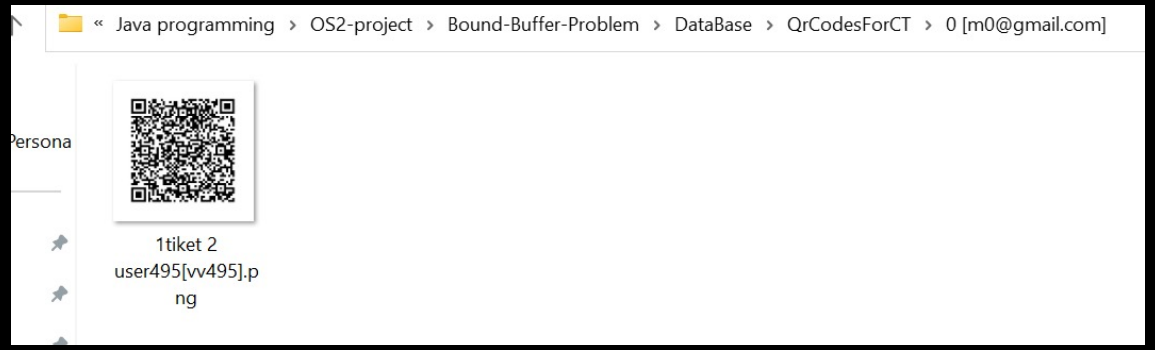




MADE WITH beautiful.ai

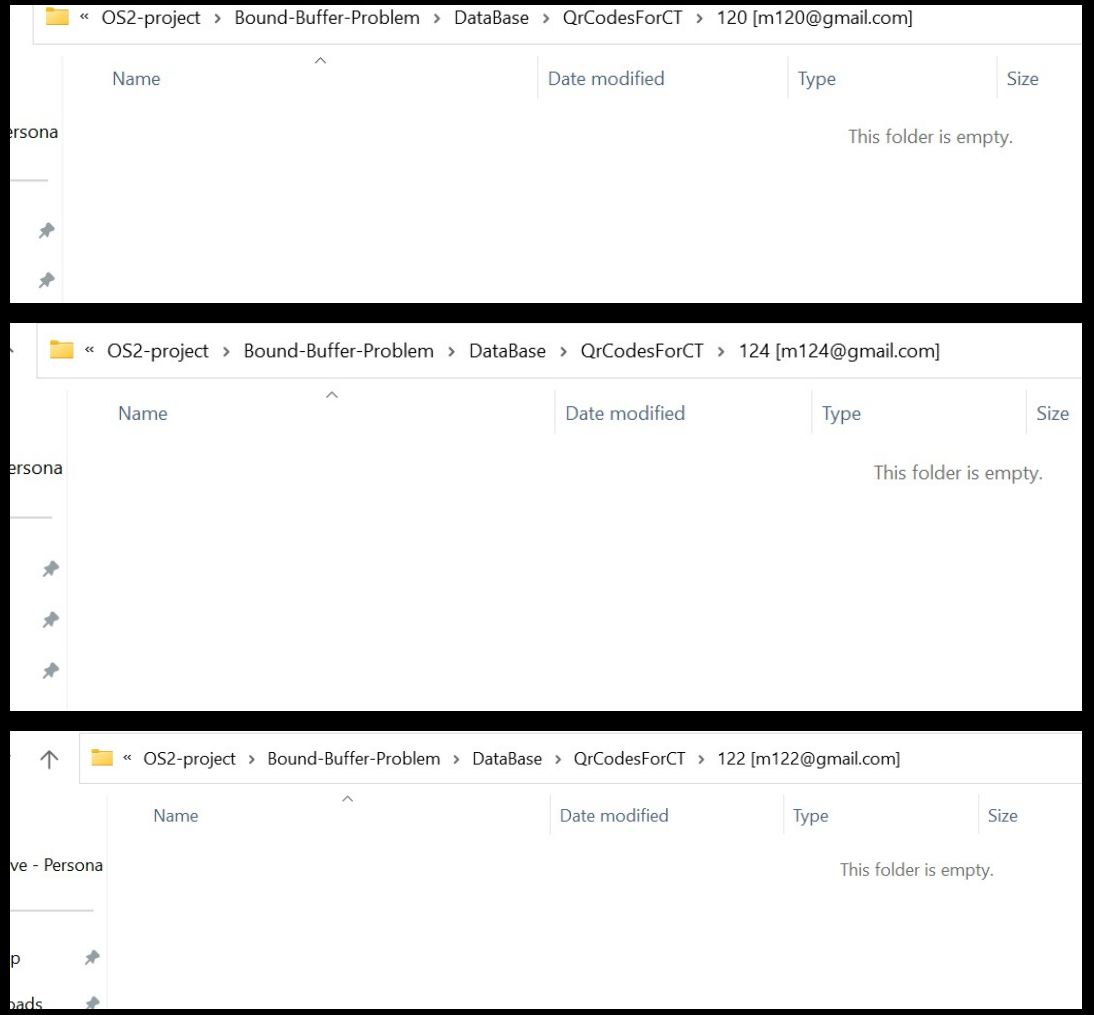
Mutual Exclusion Problem





MADE WITH beautiful.ai

Mutual Exclusion Problem



MADE WITH
beautiful.ai

Team Names

محمد زكريا كامل مسلم : 202000761

محمد اشرف خليفه صادق : 202000729

احمد اشرف طه ابراهيم احمد 202000013

محمد عماد الدين عبدالحميد : 202000806

فتحي احمد فتحي محمد: 202000645

مصطفي كمال فهمي محمد:202000907

ماهيتاب سمير حسين حسين : 202000709

طه سعيد شعبان : 202000479

MADE WITH
beautiful.ai

MADE WITH
beautiful.ai