Automatic Toll-Plaza Using Producer-Consumer Buffer

Submitted By – Rajesh Ranjan

Registration Number- 12407730

Roll Number- 63

Course Title – Operating System

Course Code – CSE316

School Of Computer Science And Engineering (B.Tech)

Batch 2024-28

Submitted To : Amandeep Kaur
Date : December 23, 2025

1. Project Overview

The main objective of this project is to design and implement an **automatic toll plaza system** based on the **Producer–Consumer problem** of Operating Systems. The system is designed to efficiently handle vehicle data such as RFID details and toll information, even when a large number of vehicles arrive at high speed.

By using a buffer and proper synchronization techniques, the system ensures smooth data transfer between the vehicle data generator and the toll processing unit. This approach helps in avoiding common issues such as data loss, delays, and system overload. The project also provides a practical understanding of core Operating System concepts by applying them to a real-world scenario.

Expected Outcomes:

- Efficient and smooth toll processing
- Prevention of system overload during heavy traffic
- Better understanding of Operating System synchronization concepts
-

Scope of the Project:

- Learning and implementing synchronization using the Producer–Consumer model
- Simulating a real-world application of Operating System concepts

2. Module-Wise Breakdown

The system is divided into three main modules to ensure clear separation of responsibilities and smooth execution of the toll plaza operations.

Module 1: Producer Module

The Producer Module is responsible for generating vehicle-related data as vehicles arrive at the toll plaza. This data may include vehicle identification details such as RFID number, vehicle number, and arrival time. Once the data is generated, it is placed into a shared buffer for further processing.

Module 2: Buffer Management Module

The Buffer Management Module acts as a temporary storage area between the producer and consumer. It maintains a buffer of limited size to store vehicle data safely. This module ensures proper handling of buffer overflow and underflow conditions and uses synchronization mechanisms to control access, preventing data inconsistency.

Module 3: Consumer Module

The Consumer Module retrieves vehicle data from the buffer and processes it to complete toll-related operations. This includes calculating the toll amount, validating vehicle details, and confirming successful processing. The consumer ensures that each vehicle’s data is processed in an orderly and reliable manner.

3. Functionalities

The automatic toll plaza system provides the following key functionalities to ensure smooth and efficient operation:

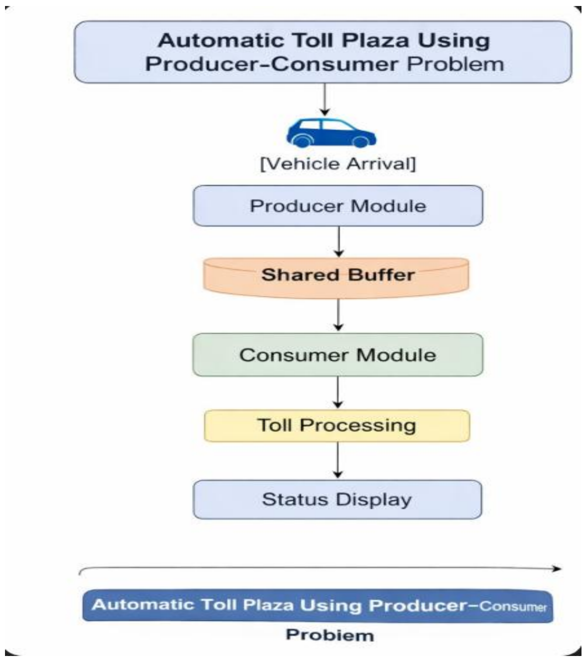
- **Vehicle Data Generation:** The system generates vehicle data whenever a vehicle arrives at the toll plaza. This data represents information such as vehicle identification and arrival details.

- **Data Insertion and Removal from Buffer:**
  Vehicle data is inserted into a shared buffer by the producer and removed by the consumer for processing. This ensures controlled data flow between different components of the system.
- **Synchronization Using Semaphores:**
  Synchronization mechanisms such as semaphores are used to manage access to the buffer. This prevents race conditions and ensures that data is neither lost nor processed incorrectly.
- **Toll Calculation:**
  The consumer processes the vehicle data and calculates the toll amount based on predefined rules, ensuring accurate and consistent toll collection.
- **Processing Status Display:**
  The system displays the current status of vehicle data processing, helping users understand whether the data is waiting, being processed, or has been successfully completed.

4. Technology Used

The following technologies and tools are used in the development of the automatic toll plaza system:

- **Programming Language:**
  C / C++ (used to implement the core logic of the Producer–Consumer problem and synchronization)
- **Operating System Concepts:**
  Producer–Consumer problem, synchronization, semaphores, and shared buffer management
- **Synchronization Tools:**
  Semaphores and mutex mechanisms are used to control access to the shared buffer and avoid race conditions.
- **Development Environment:**
  Any standard compiler such as GCC and a code editor or IDE for writing and testing the program.
- **Version Control:**
  Git and GitHub are used to track code changes and maintain different versions of the project.


Automatic Toll Plaza Using Producer–Consumer Problem
[Vehicle Arrival]
Producer Module
Shared Buffer
Consumer Module
Toll Processing
Status Display
Automatic Toll Plaza Using Producer–Consumer Probiem

Libraries and Tools

Threading libraries such as pthread or built-in threading modules are used to create producer and consumer threads. Synchronization tools like semaphores and mutex locks are applied to control access to the shared buffer and avoid race conditions during data processing.

Other Tools

GitHub is used for version control to track changes in the source code. It helps in maintaining multiple revisions of the project and ensures proper management of code updates throughout the development process.

6. Revision Tracking on GitHub

The project source code is maintained on GitHub to ensure proper version control and systematic development. A dedicated repository has been created for this project, where all updates and improvements are regularly committed with clear messages. This approach helps in tracking progress and managing different versions of the code efficiently.

Repository Name:
automatic-toll-producer-consumer
GitHub Link:
https://github.com/1RajeshRanjan

7. Conclusion and Future Scope

This project helps in gaining a clear understanding of Operating System synchronization concepts, especially the Producer–Consumer problem. By implementing an automatic toll plaza system, the project demonstrates how shared resources can be managed efficiently using buffers and synchronization techniques. It also shows the practical application of OS concepts in a real-world scenario.

Future Scope:

- The system can be extended to support multiple toll booths working simultaneously
- Real-time online payment integration can be added for faster toll collection
- AI-based traffic control and monitoring can be implemented to further reduce congestion and improve efficiency

8. References

1. Silberschatz, A., Galvin, P. B., and Gagne, G., *Operating System Concepts*, Wiley Publications. This book was referred to for understanding core Operating System concepts such as process synchronization, semaphores, and the Producer–Consumer problem.
2. Class lecture notes, assignments, and reference material provided by the course instructor, which helped in understanding the theoretical background and practical implementation of the project.
3. Official documentation and online resources related to threading, semaphores, and synchronization techniques, used for implementation guidance and clarification of concepts.

APPENDIX

A. AI-Generated Project Breakdown

This section provides a detailed breakdown of the project based on structured analysis and conceptual understanding. The project explanation focuses on the use of Operating System synchronization concepts, especially the Producer–Consumer problem, and their application in an automatic toll plaza system. The breakdown helps in clearly understanding the workflow, modules, and logic used in the implementation.

B. Problem Statement

Automatic Toll-Plaza Using Producer–Consumer Buffer

Design and implement an automatic toll plaza system that efficiently handles vehicle data using the Producer–Consumer model. The system should use a shared buffer and proper synchronization techniques to manage high traffic conditions, prevent data loss, and ensure smooth toll processing.

C. Solution / Code

The solution is implemented using appropriate programming techniques based on the Producer–Consumer problem. The complete source code of the project is provided below, which includes the implementation of producer and consumer modules, buffer management, and synchronization using semaphores or mutex locks.

(Producer–Consumer using Semaphore)

```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>


#define BUFFER_SIZE 5


int buffer[BUFFER_SIZE];

int in = 0, out = 0;


sem_t empty, full;

pthread_mutex_t mutex;


void* producer(void* arg) {

   int vehicle = 1;

    while (vehicle <= 10) {

       sem_wait(&empty);


pthread_mutex_lock(&mutex);



       buffer[in] = vehicle;

       printf("Vehicle %d entered toll
plaza\n", vehicle);

       in = (in + 1) % BUFFER_SIZE;
```

```
        pthread_mutex_unlock(&mutex);
        sem_post(&full);

        vehicle++;
        sleep(1);
    }
    return NULL;
}

void* consumer(void* arg) {
    int data;
    while (1) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        data = buffer[out];
        printf("Toll processed for vehicle
%d\n", data);
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);

        sleep(2);
    }
    return NULL;
}

int main() {
    pthread_t p, c;

     sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&p, NULL, producer,
NULL);
    pthread_create(&c, NULL, consumer,
NULL);

     pthread_join(p, NULL);
     pthread_join(c, NULL);

    return 0;
}
```

C++ (Toll Plaza Simulation – Producer Consumer)

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>
using namespace std;

queue<int> buffer;
const int SIZE = 5;
mutex mtx;
condition_variable cv;

void producer() {
    for (int i = 1; i <= 10; i++) {
        unique_lock<mutex> lock(mtx);
        cv.wait(lock, [] { return buffer.size() <
SIZE; });

        buffer.push(i);
        cout << "Vehicle " << i << " arrived at
toll\n";

        cv.notify_all();
        lock.unlock();

this_thread::sleep_for(chrono::millisecond
s(500));
    }
}

void consumer() {
    for (int i = 1; i <= 10; i++) {
        unique_lock<mutex> lock(mtx);
        cv.wait(lock, [] { return
!buffer.empty(); });

        int v = buffer.front();
        buffer.pop();
        cout << "Toll collected from vehicle "
<< v << endl;

        cv.notify_all();
        lock.unlock();

this_thread::sleep_for(chrono::millisecond
s(800));
```

```
    }
}

int main() {
    thread t1(producer);
    thread t2(consumer);

    t1.join();
    t2.join();

    return 0;
}
```

Python (Producer–Consumer Toll Plaza

```
import threading
import time
import queue

buffer = queue.Queue(5)

def producer():
    for vehicle in range(1, 11):
        buffer.put(vehicle)
        print(f"Vehicle {vehicle} entered toll
plaza")
        time.sleep(0.5)

def consumer():
    for _ in range(10):
        vehicle = buffer.get()
        print(f"Toll processed for vehicle
{vehicle}")
        time.sleep(1)

t1 = threading.Thread(target=producer)
t2 = threading.Thread(target=consumer)

t1.start()
t2.start()

t1.join()
t2.join()
```

C (Multiple Toll Booths – Advanced)

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t toll;
pthread_mutex_t lock;
int vehicle = 1;

void* booth(void* arg) {
    int id = *(int*)arg;
    while (1) {
        sem_wait(&toll);
        pthread_mutex_lock(&lock);

        if (vehicle > 10) {
            pthread_mutex_unlock(&lock);
            break;
        }

        printf("Booth %d processed vehicle
%d\n", id, vehicle);
        vehicle++;

        pthread_mutex_unlock(&lock);
        sem_post(&toll);
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t b1, b2;
    int id1 = 1, id2 = 2;

    sem_init(&toll, 0, 2);
    pthread_mutex_init(&lock, NULL);

    pthread_create(&b1, NULL, booth,
&id1);
    pthread_create(&b2, NULL, booth,
&id2);

    pthread_join(b1, NULL);
    pthread_join(b2, NULL);

    return 0;
}
```