TEC | Tecnológico de Costa Rica

September 7, 2024

Narrow Bridge Problem

Computing Engineering

Operating Systems

Author:
Danilo Duque Gómez

Professor:
Eddy Ramírez Jiménez

1 Executive Summary

The Narrow Bridge Problem represents a classic synchronization challenge with significant implications for resource management in computing systems. This project delineates a comprehensive simulation focused on addressing the Narrow Bridge Problem through three distinctive methodologies: carnage mode, traffic light synchronization, and traffic officer coordination. The amalgamation of these approaches seeks not only to confront the resource allocation dilemma directly but also to yield insights into the comparative efficacy of diverse synchronization strategies.

Recognizing the inherent complexity in managing bidirectional traffic on a single-lane bridge, simplistic approaches often lead to deadlocks or unfair resource distribution. Consequently, more sophisticated synchronization mechanisms are deployed. While these strategies may not guarantee optimal throughput in all scenarios, their ability to balance fairness, utilization, and waiting times makes the obtained solutions practical for real-world applications.

Further exploration delves into the classification of this problem as a critical section problem in concurrent programming, highlighting the challenges of mutual exclusion and the prevention of race conditions. The carnage mode approach, while simple, risks inefficient bridge utilization and potential starvation. Traffic light synchronization offers a more structured solution but may lead to unnecessary waiting times. The traffic officer coordination method provides a dynamic approach, potentially optimizing bridge usage based on real-time traffic conditions.

Subsequently, the outcomes of each resolution will be scrutinized, culminating in an evaluation of the validity of each approach. A meticulous analysis of thread synchronization, mutex usage, and deadlock prevention for each case will be presented, affording a comprehensive understanding of the strengths and limitations inherent in the applied methodologies.

This project not only addresses a fundamental problem in operating systems and concurrent programming but also provides practical experience in utilizing POSIX threads and synchronization primitives, crucial skills for developing robust, multi-threaded applications.

2 Introduction

The Narrow Bridge Problem simulation has been a captivating challenge in the realm of concurrent programming and resource management for quite some time. It's akin to a practical brain teaser that has intrigued computer scientists and software engineers since the advent of multi-threaded programming. At its core, the Narrow Bridge Problem simulation poses a crucial question: How can we efficiently manage traffic flow across a bridge that's only wide enough for a single vehicle, ensuring fairness, avoiding deadlocks, and handling priority vehicles?

The concept of managing shared resources in concurrent systems has a rich history in computer science, with seminal works by Edsger Dijkstra and others in the 1960s laying the groundwork for problems like the Narrow Bridge. These ideas have since evolved into essential components of operating systems and multi-threaded application design.

In more straightforward terms, the essence of the Narrow Bridge Problem simulation lies in coordinating the movement of vehicles across a narrow bridge. This coordination must ensure that vehicles from both directions get fair access, prevent collisions, avoid deadlocks, and give priority to emergency vehicles like ambulances. The motivation behind this meticulous management stems from the need to maximize bridge utilization while maintaining safety and fairness.

Envision a bustling city divided by a river, connected only by a single, narrow bridge. Traffic controllers must devise ingenious methods to manage the flow of vehicles, balancing efficiency, fairness, and safety as they navigate this bottleneck in the city's infrastructure.

In a more formal context, the Narrow Bridge Problem simulation revolves around managing a critical section (the bridge) in a multi-threaded environment. This involves coordinating multiple threads representing vehicles, traffic lights, and traffic officers, all competing for access to the shared resource.

Let's assume we have two sets of vehicles, $E = \{e_1, e_2, \ldots, e_n\}$ approaching from the east, and $W = \{w_1, w_2, \ldots, w_m\}$ approaching from the west. The goal is to implement a system S that manages the bridge access, where S must satisfy the following conditions:

1. Mutual Exclusion: At any given time, the bridge can be occupied by vehicles traveling in only one direction.

2. Progress: If the bridge is empty and there are waiting vehicles, one must be allowed to cross.

3. Bounded Waiting: There exists a bound on the number of times other vehicles can cross the bridge before a given vehicle is allowed to cross.

4. Priority Handling: Ambulances must be given preferential treatment when they are at the front of the queue.

This classification as a critical section problem, indicating the need for careful synchronization and resource management, stems from the intricacy of coordinating multiple independent threads in a way that ensures safety, fairness, and efficiency. The challenge is further complicated by the need to handle different traffic management strategies and priority vehicles.

Furthermore, the evolution of concurrent programming techniques and synchronization primitives has catalyzed interest in practical simulations like the Narrow Bridge Problem. Techniques such as mutex locks, semaphores, and condition variables find application in addressing the challenges posed by this simulation.

The Narrow Bridge Problem simulation's significance transcends theoretical computer science. Real-world applications span traffic management systems, resource allocation in operating systems, and synchronization in distributed systems. The efficient resolution of such concurrency challenges holds practical implications for diverse fields, influencing system design and optimizing resource utilization.

In the following sections, key implementation approaches for addressing the Narrow Bridge Problem simulation are explored. The examination illuminates the use of threading, synchronization mechanisms, and different traffic

management strategies, aiming to deepen the understanding of concurrent programming challenges and contribute to the ongoing discourse regarding efficient resource management in multi-threaded environments.

3 Theoretical Framework

3.1 Concurrent Programming and Multi-threading

The Narrow Bridge Problem simulation is fundamentally a challenge in concurrent programming. Concurrent programming allows multiple computations to occur simultaneously, which is essential for simulating the independent movement of vehicles from different directions.

3.1.1 POSIX Threads (pthreads)

Our implementation utilizes POSIX Threads (pthreads), a standardized C language threads programming interface. Key concepts include:

- **Thread Creation**: Using `pthread_create()` to spawn new threads for each vehicle and traffic management entity.
- **Thread Synchronization**: Employing `pthread_join()` to wait for thread completion.
- **Thread Attributes**: Utilizing `pthread_attr_init()` and related functions to set thread properties.

3.2 Synchronization Mechanisms

To manage the critical section (the bridge) and ensure thread safety, several synchronization primitives are employed:

3.2.1 Mutex Locks

Mutex (mutual exclusion) locks prevent multiple threads from simultaneously accessing shared resources. In our simulation, mutexes protect the bridge state and shared counters. Key functions include:

- `pthread_mutex_init()`: Initialize a mutex
- `pthread_mutex_lock()`: Acquire a lock
- `pthread_mutex_unlock()`: Release a lock

3.2.2 Condition Variables

Condition variables allow threads to synchronize based on the value of data. They are used to implement the different traffic management strategies. Relevant functions include:

- `pthread_cond_init()`: Initialize a condition variable
- `pthread_cond_wait()`: Wait on a condition
- `pthread_cond_signal()`: Signal a waiting thread
- `pthread_cond_broadcast()`: Signal all waiting threads

3.3 Traffic Management Strategies

The simulation implements three distinct strategies for managing bridge access:

1. First-Come-First-Served (FCFS)
2. Traffic Light System
3. Traffic Officer Control

Each strategy is implemented using a combination of mutexes and condition variables to ensure fair and efficient bridge utilization.

3.4 Development Environment

3.4.1 C Programming Language

The project is implemented in C, chosen for its low-level control and efficiency in systems programming. Key aspects include:

- Pointer manipulation for dynamic memory management

- Struct usage for representing vehicles and simulation state
- Function pointers for implementing different traffic management strategies

3.4.2 Vim Text Editor

Vim, a highly configurable text editor, was used for code development. Relevant features include:

- Modal editing for efficient code navigation and manipulation
- Syntax highlighting for C and Makefile formats
- Integration with the compilation process for quick error checking

3.5 Randomization and Simulation

To create a realistic traffic scenario, the simulation employs randomization techniques:

- Pseudo-random number generation for vehicle arrival times
- Probabilistic determination of vehicle types (regular cars vs. ambulances)
- Random duration for bridge crossing times

3.5.1 Exponential Distribution for Vehicle Arrival Times

A key feature of the simulation is the use of an exponential distribution to model the inter-arrival times of vehicles. This choice reflects real-world traffic patterns more accurately than a uniform distribution would.

The exponential distribution is characterized by the following probability density function:

$$f(x; \lambda) = \lambda e^{-\lambda x}, \text{ for } x \geq 0$$

Where:

- x is the time between events (in our case, vehicle arrivals)

- λ (lambda) is the rate parameter, which is the reciprocal of the mean inter-arrival time

Key properties of the exponential distribution relevant to our simulation:

- **Memoryless property**: The probability of an arrival in the next time interval is independent of how long it's been since the last arrival.
- **Continuous distribution**: Arrivals can occur at any point in time, not just at discrete intervals.
- **Higher probability of shorter inter-arrival times**: This models the tendency of vehicles to arrive in clusters in real traffic scenarios.

By using this distribution, our simulation more accurately represents the unpredictable nature of traffic flow, with periods of heavy traffic interspersed with quieter periods. This adds an additional layer of complexity to the traffic management strategies, as they must be able to handle both sudden influxes of vehicles and periods of lower activity efficiently.

These elements combine to create a dynamic and unpredictable simulation environment, challenging the implemented traffic management strategies and providing a more realistic test of their effectiveness in various traffic conditions.

4 Solution Description

In this section, the implemented solutions addressing the Narrow Bridge Problem are presented. The focus lies on three distinct strategies aimed at managing traffic flow across a single-lane bridge. Each strategy ensures fair access and handles various types of vehicles, including emergency ones. The effectiveness, design, and operation of each strategy are explained in detail.

4.1 Modeling of Key Components

Before analyzing specific traffic management strategies, it is essential to understand the modeling of the core components of the simulation: the bridge

(critical zone), regular vehicles, and ambulances.

4.1.1 The Bridge (Critical Zone)

The bridge, representing the critical zone in the synchronization problem, is modeled using the `Bridge` structure:

```
typedef struct {
    int sem, sz, dir, amb_waiting, t1, t2;
    pthread_mutex_t access;
    mtx *bridge;
} Bridge;
```

This structure contains several crucial elements:

- `sem`: Semaphore controlling access to the bridge.
- `sz`: Size of the bridge, indicating how many vehicles can be on it simultaneously.
- `dir`: Current traffic direction on the bridge (-1 for west, 1 for east, 0 if empty).
- `amb_waiting`: Counter for ambulances waiting to cross.
- `t1, t2`: Time variables for traffic management strategies.
- `bridge`: Array of mutexes representing each section of the bridge.

The bridge is initialized by the `init()` function, which dynamically allocates memory and sets up initial values and necessary mutexes.

4.1.2 Regular Vehicles and Ambulances

Vehicles, both regular and ambulances, are modeled using the `Car` structure:

```
typedef struct{
    int dir;
    double v;
} Car;
```

Where:

- `dir`: Direction of the vehicle (1 for east, -1 for west).
- `v`: Speed of the vehicle, randomly generated within a specified range.

Vehicles are created through the `CreateCar()` function, which generates a new vehicle with random speed within the specified range and the assigned direction.

4.1.3 Bridge Crossing

The process of crossing the bridge is simulated by the `CrossBridge()` function, which models the vehicle's movement through each section of the bridge:

```
void CrossBridge(Car * c, int st, int end, int amb){
    for(int i = st; i!=end+c->dir; i+=c->dir){
        lock(&cz->bridge[i].scnd);
        cz->bridge[i-c->dir].frst=0;
        unlock(&cz->bridge[i-c->dir].scnd);
        cz->bridge[i].frst=amb;
        usleep(micro/c->v);
    }
}
```

This function ensures mutual exclusion in each bridge section by using locks, simulating the vehicle's movement and releasing sections once they have been crossed.

4.1.4 Ambulance Handling

Ambulances receive special handling in the simulation. The behavior of an ambulance crossing the bridge is modeled by the `EnterAmbulance()` function:

```
void* EnterAmbulance(void *arg) {
    Car* car = (Car*)arg;
    int st = start(car), end = end(car);
    lock(&cz->bridge[st-car->dir].scnd);
   lock(&bridge_mutex);
    cz->bridge[st-car->dir].frst = 2; cz->amb_waiting++;
```

```
    while((car->dir == 1)? cz->dir<0 : cz->dir>0)
   wait(&empty, &bridge_mutex);
    cz->dir += car->dir; --cz->amb_waiting;
   unlock(&bridge_mutex);
    CrossBridge(car, st, end, 2);
    lock(&bridge_mutex);
    cz->dir -= car->dir; cz->bridge[end].frst = 0;
    if (cz->dir == 0) signal(&empty);
    unlock(&bridge_mutex); unlock(&cz->bridge[end].scnd);
    free(car);
    return 0;
}
```

This function prioritizes ambulances, ensuring they can cross the bridge as soon as possible, even when other vehicles are waiting.

4.1.5 Traffic Generation

In the simulation, each traffic management mode has its own traffic generator function. These generators follow a similar structure, with variations in the vehicle crossing function they invoke. The `CarnageCarGenerator()` function serves as an example:

```
void* CarnageCarGenerator(double mu, double l, double u,
   double p, int d){
    while(1){
        usleep(-mu*log(1-prob())*micro);
        pthread_t t;
        pthread_create(&t, 0, (prob()<p)? EnterAmbulance
   : EnterCarnageCar, (void*)CreateCar(l, u, p, d));
        pthread_detach(t);
    }
}
```

Key characteristics of this generator:

- It runs continuously, generating vehicles.
- Vehicle arrival times are based on an exponential distribution.
- For each vehicle, a new thread is created and detached.

- Probability `p` determines whether the vehicle is an ambulance or a regular car.
- The `EnterCarnageCar` function is specific to the Carnage mode.

The traffic generators for other modes (Traffic Light and Traffic Officer) have similar structures, differing only in the vehicle crossing function they call:

- Carnage mode calls `EnterCarnageCar`.
- Traffic Light mode would call a function like `EnterSemaphoreCar`.
- Traffic Officer mode would call a function like `EnterTrafficCar`.

Each of these crossing functions implements the specific logic for its respective traffic management strategy. The ambulance crossing function, `EnterAmbulance`, remains consistent across all modes, ensuring priority handling for emergency vehicles.

This modular design facilitates easy implementation and comparison of different traffic management strategies, while maintaining a consistent structure for vehicle generation and handling.

These components form the basis of the simulation, providing a realistic representation of the narrow bridge problem. The following sections explore how these structures and functions are utilized in different traffic management strategies.

4.2 First-Come-First-Served (Carnage) Mode

The First-Come-First-Served strategy, also known as the Carnage mode, manages bridge access based on vehicle arrival order. This method offers a simple and fair approach to bridge traffic management but can face challenges in high-volume traffic scenarios and when prioritizing emergency vehicles.

4.2.1 Implementation Details

The core of this strategy is implemented in the `EnterCarnageCar` function:

```
void* EnterCarnageCar(void *arg){
    Car* car = (Car*)arg;
    int st = start(car), end = end(car);
    lock(&cz->bridge[st-car->dir].scnd);
    while(cz->amb_waiting || ((car->dir == 1)? cz->dir<0
   : cz->dir>0))
        wait(&empty, &cz->bridge[st-car->dir].scnd);
    cz->dir += car->dir;
    CrossBridge(car, st, end, 1);
    cz->dir -= car->dir; cz->bridge[end].frst=0;
    if (cz->dir == 0) signal(&empty);
    unlock(&cz->bridge[end].scnd);
    return 0;
}
```

Key elements of this implementation include:

- Lock Acquisition: The function begins by acquiring a lock on the bridge section where the car will enter.
- Waiting Condition: The car waits if there are ambulances waiting or if traffic is flowing in the opposite direction. This ensures that ambulances receive priority and cars only cross when the bridge is empty or traffic flows in their direction.
- Bridge Crossing: Once the conditions are met, the car updates the bridge direction and crosses using the `CrossBridge` function.
- Cleanup: After crossing, the car updates the bridge state, potentially signaling other waiting vehicles if the bridge becomes empty.

4.2.2 Conclusion

The First-Come-First-Served (Carnage) mode offers a basic yet fair approach to managing bridge traffic. Its simplicity makes it easy to implement and understand, but it may face challenges in optimizing bridge utilization and handling priority vehicles in complex traffic scenarios.

4.3 Semaphore Mode

The Semaphore mode introduces a traffic management strategy that relies on a signal-based system to control the flow of traffic across the bridge. In this mode, the semaphore alternates the right of way between eastbound and westbound traffic at regular intervals, ensuring that cars only cross the bridge when it is safe to do so. This method is particularly effective in managing congestion and ensuring fairness, especially in high-volume traffic scenarios.

4.3.1 Implementation Details

The core of the Semaphore mode is implemented using two main components: the `run_Semaphore()` function, which controls the semaphore's state, and the `EnterSemaphoreCar()` function, which handles the crossing of vehicles based on the semaphore's signals.

Semaphore Control The `run_Semaphore()` function continuously alternates the semaphore state between allowing eastbound and westbound traffic:

```
void run_Semaphore(){
    while(1){
        cz->sem = 1;
        zzz(t_si * micro);
        cz->sem = -1;
        zzz(t_sj * micro);
    }
}
```

Key elements of this implementation:

- `cz->sem` is set to `1` to allow eastbound traffic for a duration defined by `t_si`.
- After the eastbound phase, the semaphore switches to `-1`, permitting westbound traffic for a duration defined by `t_sj`.

- This alternating cycle continues indefinitely, providing a timed traffic management strategy that controls which direction has access to the bridge.

Vehicle Crossing The `EnterSemaphoreCar()` function governs how vehicles cross the bridge, based on the current semaphore state:

```
void* EnterSemaphoreCar(void *arg) {
    Car* car = (Car*)arg;
    int st = start(car), end = end(car);
    lock(&cz->bridge[st - car->dir].scnd);
    cz->bridge[st - car->dir].frst = 1;
    lock(&bridge_mutex);
    while (cz->amb_waiting || ((car->dir == 1) ? cz->dir
   < 0 : cz->dir > 0) || (car->dir != cz->sem))
   wait(&change_semaphore, &bridge_mutex);
    cz->dir += car->dir;
    unlock(&bridge_mutex);
    CrossBridge(car, st, end, 1);
    lock(&bridge_mutex);
    cz->dir -= car->dir; cz->bridge[end].frst = 0;
    if (cz->dir == 0) signal(&empty);
    unlock(&bridge_mutex); unlock(&cz->bridge[end].scnd);
    free(car);
    return 0;
}
```

Key components of this function include:

- **Lock Acquisition**: The vehicle locks the bridge section it will enter.
- **Waiting Condition**: The vehicle waits if there are ambulances waiting, if traffic is flowing in the opposite direction, or if the semaphore does not match the vehicle's direction. This ensures controlled and safe passage.
- **Bridge Crossing**: Once the conditions are satisfied, the vehicle proceeds to cross the bridge using the `CrossBridge()` function.
- **Cleanup**: After crossing, the vehicle updates the bridge state and signals other vehicles if the bridge becomes empty.

4.3.2 Traffic Generation

The Semaphore mode utilizes a specific traffic generator function, `SemaphoreCarGenerator()`, which creates vehicles at regular intervals and assigns them to either the eastbound or westbound direction based on the semaphore state:

```
void SemaphoreCarGenerator(double mu, double l, double
   u, double p, int d){
    while(1){
        zzz(-mu*log(1 - prob()) * micro);
        pthread_t t;
        pthread_create(&t, 0, (prob()<p)? EnterAmbulance
   : EnterSemaphoreCar, (void*)CreateCar(l, u, p, d));
        pthread_detach(t);
    }
}
```

This generator:

- Creates vehicles with arrival times following an exponential distribution.
- Randomly determines whether each vehicle is an ambulance or a regular car.
- Spawns a new thread for each vehicle and uses the `EnterSemaphoreCar()` function to manage its crossing based on the semaphore's state.

Additionally, the generator is started via the `run_generator_Semaphore()` function, which initializes the generator with specific parameters related to vehicle distribution and arrival rates:

```
void* run_generator_Semaphore(void* arg) {
    domain* dom = (domain*)arg;
    SemaphoreCarGenerator(dom->mu, dom->lbv, dom->ubv,
   dom->ambProb, dom->dir);
    return 0;
}
```

This function encapsulates the setup and execution of the Semaphore mode traffic generator, making it easy to start and manage within the simulation.

4.3.3 Conclusion

The Semaphore mode provides a structured approach to traffic management by introducing timed intervals for eastbound and westbound traffic. This method helps mitigate congestion and ensures that vehicles only cross the bridge when it is their turn according to the semaphore's signal. By implementing clear rules for waiting and crossing, Semaphore mode offers a reliable solution to the narrow bridge problem while also prioritizing emergency vehicles when necessary.

4.4 Traffic Office Mode

The Traffic mode introduces a system that governs vehicle passage across the bridge by allowing a specified number of cars to cross from one direction before switching to the opposite direction. This approach helps balance traffic flow and minimizes congestion by ensuring that vehicles from both sides get a fair chance to pass. The mode also prioritizes ambulances when necessary, ensuring that emergency vehicles can cross the bridge without delay.

4.4.1 Implementation Details

The core functionality of the Traffic mode is handled by the `run_Traffic()` function, which alternates control between the two directions, and the `EnterTrafficCar()` and `EnterTrafficAmbulance()` functions, which manage the crossing of regular vehicles and ambulances, respectively.

Traffic Control The `run_Traffic()` function controls the alternation of traffic flow by switching between the two directions after a certain number of cars have crossed from one side:

```
void *run_Traffic() {
    while (1) {
        lock(&bridge_mutex);
        reset(1);
        while(cz->dir) wait(&empty, &bridge_mutex);
        while (cz->t1 > 0) {
            broadcast(&pass);
            if (!cz->dir && !cz->bridge[0].frst &&
   cz->bridge[cz->sz + 1].frst) break;
            wait(&empty, &bridge_mutex);
        }
        unlock(&bridge_mutex);

        lock(&bridge_mutex);
        reset(0);
        while(cz->dir) wait(&empty, &bridge_mutex);
        while (cz->t2 > 0) {
            broadcast(&pass);
            if (!cz->dir && cz->bridge[0].frst &&
   !cz->bridge[cz->sz + 1].frst) break;
            wait(&empty, &bridge_mutex);
        }
        unlock(&bridge_mutex);
    }
}
```

Key elements of this implementation:

- `reset()` alternates the semaphore and initializes the counter for each direction.
- The function waits for the current direction's traffic to clear before switching to the opposite side.
- Traffic flow alternates between two directions based on the counters `t1` and `t2`, ensuring that only one direction is active at a time.

Vehicle Crossing The `EnterTrafficCar()` function handles the crossing of regular vehicles by checking the current traffic direction and waiting for a

turn when necessary:

```
void* EnterTrafficCar(void *arg) {
    Car* car = (Car*)arg;
    int st = start(car), end = end(car);
    lock(&cz->bridge[st - car->dir].scnd);
    cz->bridge[st - car->dir].frst = 1;
    if (!cz->dir) broadcast(&empty);
    lock(&bridge_mutex);
    while (cz->amb_waiting || ((car->dir == 1) ? cz->t1
   <= 0 || cz->dir < 0 : cz->t2 <= 0 || cz->dir > 0) ||
   (cz->sem != car->dir))
        wait(&pass, &bridge_mutex);
    cz->dir += car->dir;
    (car->dir == 1) ? --cz->t1 : --cz->t2;
    unlock(&bridge_mutex);
    CrossBridge(car, st, end, 1);
    lock(&bridge_mutex);
    cz->dir -= car->dir;
    cz->bridge[end].frst = 0;
    if (!cz->dir) broadcast(&empty);
    unlock(&bridge_mutex);
    unlock(&cz->bridge[end].scnd);
    free(car);
    return 0;
}
```

Key components of this function include:

- **Lock Acquisition**: The vehicle locks the bridge section it will enter.
- **Waiting Condition**: The vehicle waits if ambulances are waiting, if traffic is flowing in the opposite direction, or if the semaphore does not match its direction. This ensures orderly passage.
- **Bridge Crossing**: The vehicle crosses the bridge once conditions are met, using the `CrossBridge()` function.
- **Cleanup**: After crossing, the vehicle updates the bridge state and signals other vehicles if the bridge is now empty.

Ambulance Crossing The `EnterTrafficAmbulance()` function handles the crossing of ambulances, which are given priority over regular vehicles:

```
void* EnterTrafficAmbulance(void *arg) {
    Car* car = (Car*)arg;
    int st = start(car), end = end(car);
    lock(&cz->bridge[st - car->dir].scnd);
    lock(&bridge_mutex);
    cz->bridge[st - car->dir].frst = 2;
    cz->amb_waiting++;
    while ((car->dir == 1) ? cz->dir < 0 : cz->dir > 0)
   wait(&empty, &bridge_mutex);
    if (car->dir == cz->sem) (car->dir == 1) ? --cz->t1
   : --cz->t2;
    cz->dir += car->dir;
    --cz->amb_waiting;
    unlock(&bridge_mutex);
    CrossBridge(car, st, end, 2);
    lock(&bridge_mutex);
    cz->dir -= car->dir;
    cz->bridge[end].frst = 0;
    if (cz->dir == 0) broadcast(&empty);
    unlock(&bridge_mutex);
    unlock(&cz->bridge[end].scnd);
    free(car);
    return 0;
}
```

Ambulance priority ensures that traffic halts for the ambulance to cross as soon as the bridge is clear, providing a timely response to emergency situations.

4.4.2 Traffic Generation

The Traffic mode uses a traffic generator function, `TrafficCarGenerator()`, which creates vehicles at random intervals following an exponential distribution:

```
void* TrafficCarGenerator(double mu, double l, double u,
   double p, int d) {
    while (1) {
        zzz(-mu * log(1 - prob()) * micro);
        thread t;
        create(&t, (prob() < p) ? EnterTrafficAmbulance
   : EnterTrafficCar, CreateCar(l, u, p, d));
        detach(t);
    }
}
```

This generator:

- Randomly generates vehicles with arrival times following an exponential distribution.
- Decides whether a vehicle is an ambulance or a regular car.
- Spawns a new thread for each vehicle and uses the corresponding function (`EnterTrafficCar()` or `EnterTrafficAmbulance()`) to manage its crossing.

The generator is started via the `run_generator_Traffic()` function:

```
void* run_generator_Traffic(void* arg) {
    domain* dom = (domain*)arg;
    TrafficCarGenerator(dom->mu, dom->lbv, dom->ubv,
   dom->ambProb, dom->dir);
    return 0;
}
```

This function initializes the traffic generator with specific parameters, making it easy to manage within the simulation.

4.4.3 Conclusion

The Traffic mode offers a balanced approach to managing vehicle flow across the bridge by alternating the right of way between directions. The mode prioritizes ambulances when necessary and ensures that traffic is managed

fairly. Through the use of clear rules for vehicle crossing and efficient traffic generation, Traffic mode provides a robust solution for simulating realistic bridge scenarios.

5 Performance Evaluation

In this section, the strengths and weaknesses of each traffic management mode—Carnage, Semaphore, and Traffic Officer—are evaluated. The analysis focuses on the conditions in which each mode is most effective and the reasons behind these performance characteristics.

5.1 Carnage Mode

Carnage mode is optimal only under specific conditions: when the arrival time of vehicles is high, and the time vehicles spend using the bridge is also relatively short. This mode works well when there is ample spacing between vehicle arrivals, allowing the bridge to clear before another vehicle needs to cross. However, it has significant weaknesses under different conditions.

Strengths: - Carnage mode performs efficiently when the arrival time between vehicles is high and their bridge usage time is short. In such cases, it ensures that vehicles are processed quickly without significant delays, as the bridge is more likely to be free when a new vehicle arrives.

Weaknesses: - When vehicle arrival time is low, or the time spent on the bridge is high, Carnage mode may result in vehicles from one direction monopolizing the bridge. This could lead to vehicles from the opposite direction being entirely neglected, causing long delays for those vehicles. The issue becomes even more pronounced if the bridge is short, as frequent interruptions prevent smooth traffic flow.

5.2 Semaphore Mode

Semaphore mode excels in scenarios where the mean arrival time of vehicles is low, implying that vehicles appear frequently. The semaphore alternates traffic flow at regular intervals, which becomes advantageous when vehicles spend longer times on the bridge. The traffic light system effectively manages high volumes of traffic, ensuring that both directions are served, though not without its own trade-offs.

Strengths: - This mode is well-suited for situations where vehicle arrival time is low, and vehicles spend considerable time on the bridge. The regular switching of the semaphore ensures that vehicles from both directions get a chance to cross, preventing one side from monopolizing the bridge.

Weaknesses: - When the arrival time is high or the time spent on the bridge is short, the semaphore's forced alternation can lead to unnecessary delays. Vehicles might wait for the light to change even when there is no opposing traffic, leading to inefficient bridge utilization.

5.3 Traffic Officer Mode

Traffic Officer mode serves as a hybrid between the Carnage and Semaphore modes. It adapts to varying traffic conditions by dynamically adjusting the flow based on vehicle usage time. In scenarios where vehicles spend less time on the bridge, it behaves similarly to Carnage mode, optimizing for quick passage. Conversely, when vehicles require more time on the bridge, it shifts towards Semaphore-like behavior, ensuring fairness between directions.

Strengths: - This mode's adaptability makes it efficient across a broader range of scenarios. When the time spent on the bridge is short, Traffic Officer mode capitalizes on the strengths of Carnage mode, processing vehicles rapidly. When vehicles take longer to cross, it ensures fair distribution similar to Semaphore mode.

Weaknesses: - The main drawback of Traffic Officer mode is its complexity. Managing the balance between rapid passage and fairness can lead to scenarios where neither is fully optimized, resulting in performance that, while

adaptable, may not be as specialized or efficient as Carnage or Semaphore modes in their respective ideal conditions.

6 Conclusions

The Narrow Bridge Problem simulation project has provided valuable insights into the complexities of traffic management and resource allocation in concurrent systems. Through the implementation and analysis of three distinct traffic management strategies—Carnage mode, Semaphore mode, and Traffic Officer mode—we have gained a deeper understanding of the trade-offs involved in managing shared resources under varying conditions.

6.1 Key Findings

1. **Scenario-Specific Efficacy:** Each mode demonstrated unique strengths in specific traffic scenarios:
    - Carnage mode excelled in low-traffic situations with short bridge crossing times.
    - Semaphore mode proved effective in high-traffic scenarios, especially with longer bridge crossing times.
    - Traffic Officer mode showed adaptability across various traffic conditions, balancing efficiency and fairness.
2. **Importance of Adaptability:** The Traffic Officer mode's ability to adjust its strategy based on real-time traffic conditions highlighted the value of adaptive algorithms in resource management.
3. **Trade-offs in Design:** Each mode revealed trade-offs between simplicity, fairness, and efficiency. The Carnage mode's simplicity came at the cost of potential unfairness, while the Semaphore mode's fairness sometimes led to inefficiencies.
4. **Emergency Vehicle Prioritization:** All modes successfully implemented priority handling for ambulances, demonstrating the importance of considering critical cases in resource allocation problems.

5. **Concurrent Programming Challenges:** The project underscored the complexity of managing shared resources in multi-threaded environments, particularly in preventing deadlocks and ensuring fair access.

6.2 Implications and Future Work

1. **Real-world Applications:** The insights gained from this simulation have potential applications in traffic management systems, operating systems resource allocation, and other fields involving concurrent access to limited resources.

2. **Algorithm Refinement:** Future work could focus on refining the Traffic Officer mode to optimize its adaptability, potentially incorporating machine learning techniques for more intelligent decision-making.

3. **Scalability Testing:** Expanding the simulation to handle larger volumes of traffic or more complex bridge configurations could provide insights into the scalability of these approaches.

4. **Performance Metrics:** Developing more sophisticated metrics for measuring efficiency, fairness, and overall system performance could lead to more nuanced comparisons between different management strategies.

5. **Integration with Real-time Systems:** Exploring how these algorithms could be integrated with real-time traffic monitoring systems could bridge the gap between simulation and practical application.

In conclusion, this project has not only addressed a classic problem in concurrent programming but has also provided a framework for understanding and solving similar resource allocation challenges. The multi-faceted approach taken in this study demonstrates the importance of considering various strategies when dealing with complex systems, as no single solution proves optimal across all scenarios. The lessons learned here extend beyond traffic management, offering valuable insights into the broader field of concurrent system design and resource allocation.

7 Experience Description

The implementation of the Narrow Bridge Problem proved to be a challenging and, at times, frustrating experience, particularly in the realm of concurrent programming. This project provided valuable insights into the complexities of managing shared resources and the intricacies of thread synchronization.

7.1 Dealing with Deadlocks

One of the most frustrating aspects of this project was the identification and elimination of deadlocks. This process was often time-consuming and mentally taxing:

- **Elusive Nature**: Deadlocks would often occur sporadically, making them difficult to reproduce and debug consistently.
- **Complex Interactions**: The interplay between multiple threads and shared resources created scenarios where deadlocks emerged from subtle timing issues or unforeseen interactions.
- **Debugging Challenges**: Traditional debugging techniques were often insufficient, as the act of debugging could alter the timing and mask the deadlock conditions.
- **Trial and Error**: Resolving deadlocks often involved a frustrating process of trial and error, requiring patience and persistence.

7.2 Caution in Concurrent Programming

The project underscored the critical importance of extreme caution in concurrent programming. Several key lessons emerged:

- **Anticipating Edge Cases**: It was crucial to consider all possible scenarios where two or more processes might interact in unexpected ways, leading to race conditions or deadlocks.
- **Careful Resource Management**: Managing shared resources required meticulous planning and implementation to ensure proper locking and unlocking sequences.

- **Balancing Granularity**: Finding the right balance between fine-grained locking for performance and coarse-grained locking for simplicity was a constant challenge.
- **Thorough Testing**: Extensive testing under various conditions was necessary to uncover subtle concurrency issues that might not be immediately apparent.

7.3 Unexpected Behaviors

The project revealed how seemingly innocuous code changes could lead to unexpected and often perplexing behaviors:

- **Timing Sensitivities**: Minor alterations in the code could dramatically affect the timing of thread execution, leading to entirely different outcomes.
- **Non-Determinism**: The non-deterministic nature of concurrent execution meant that problems could appear or disappear across different runs with no apparent reason.
- **Cascading Effects**: Issues in one part of the system often had ripple effects, causing unexpected behaviors in seemingly unrelated components.

7.4 Learning from Frustration

Despite the frustrations, this experience provided invaluable learning opportunities:

- **Deepened Understanding**: Overcoming these challenges led to a much deeper understanding of concurrent programming principles.
- **Problem-Solving Skills**: The process honed problem-solving skills, particularly in systematic debugging and root cause analysis.
- **Appreciation for Design**: It fostered a greater appreciation for careful system design and the importance of considering concurrency from the outset.

- **Patience and Perseverance**: The experience reinforced the value of patience and perseverance in tackling complex programming challenges.

In conclusion, while the experience of implementing the Narrow Bridge Problem was often frustrating, particularly in dealing with deadlocks and the intricacies of concurrent programming, it was ultimately a highly educational journey. It highlighted the need for extreme caution, meticulous planning, and thorough testing in concurrent systems development, providing lessons that will undoubtedly prove valuable in future programming endeavors.

8 Learnings

This project has provided valuable hands-on experience in several key areas of concurrent programming and simulation. The main learnings from this project include:

8.1 Safe Usage of Mutexes and Critical Sections

Throughout the implementation of the Narrow Bridge Problem, I gained practical experience in:

- Identifying critical sections in the code where shared resources (like the bridge) are accessed.
- Implementing mutex locks to ensure mutual exclusion in these critical sections.
- Properly acquiring and releasing locks to prevent deadlocks and ensure thread safety.
- Understanding the importance of minimizing the size of critical sections to improve concurrency.

8.2 Thread Management

Working with multiple threads to simulate vehicles and traffic controllers enhanced my understanding of:

- Creating and managing threads using POSIX threads (pthreads).

- Synchronizing threads using condition variables and mutex locks.
- Handling thread creation, execution, and termination safely.
- Avoiding race conditions by carefully orchestrating thread interactions.

8.3 Exponential Distribution for Event Simulation

Implementing the traffic generation system provided insights into:

- Using the exponential distribution to model the inter-arrival times of vehicles.
- Understanding why the exponential distribution is suitable for representing random event occurrences in time.
- Implementing a pseudo-random number generator to create exponentially distributed values.
- Appreciating how this distribution creates more realistic traffic patterns compared to uniform distribution.

8.4 Practical Application of Concurrent Programming Concepts

This project served as a practical application of theoretical concepts, helping to solidify understanding of:

- The producer-consumer problem, as seen in the generation and processing of vehicles.
- The readers-writers problem, analogous to managing bi-directional traffic on the bridge.
- The importance of fairness in resource allocation to prevent starvation.
- Balancing system throughput with fairness and safety considerations.

These learnings have not only enhanced my understanding of concurrent programming and simulation techniques but have also provided valuable insights into solving complex, real-world problems involving resource management and synchronization.

References

[1] "pthread.c," Apple.com, 2024. [Online]. Available: https://opensource.apple.com/source/libpthread/libpthread-330.201.1/src/pthread.c.auto.html. [Accessed: 07-Sep-2024].

[2] "Thread functions in C/C++," GeeksforGeeks, Mar. 12, 2019. [Online]. Available: https://www.geeksforgeeks.org/thread-functions-in-c-c/.

[3] "The Open Group Base Specifications Issue 7, 2018 edition," pubs.opengroup.org. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/.

[4] "Operating System Concepts - slides," Os-book.com, 2018. [Online]. Available: https://www.os-book.com/OS10/slide-dir/. [Accessed: 07-Sep-2024].