Ferry Tour Simulation: A Multithreaded Vehicle Transportation System with Educational Insights into Concurrency and Optimization

Furkan Bulut, *Undergraduate Student,*
Department of Computer Engineering, Faculty of Engineering, Manisa Celal Bayar University, Manisa, Turkey
Email: 210316011@ogr.cbu.edu.tr

***Abstract*—This paper presents a multithreaded ferry simulation implemented in C using POSIX threads to model vehicle transportation across two city sides. It serves as a practical system and an educational tool for understanding concurrency, synchronization, and optimization. The system manages 30 vehicles (12 cars, 10 minibuses, 8 trucks) with capacities of 1, 2, and 4 units, respectively, using a ferry with a 20-unit capacity. Vehicles start on random sides, pass through four tolls (two per side), and complete round-trips, with mutexes and semaphores ensuring thread safety. Dynamic programming optimizes ferry departures, and real-time visualization aids monitoring. Tested on a Linux Mint 22.1 system with an Intel i7-11800H CPU, the simulation ran for 88 s with 12 trips but faced issues like logging inaccuracies (e.g., 0% capacity for non-empty trips) and a 33.3% empty trip rate. This paper details the problem, solution logic, system design, and performance analysis, offering insights for learners and proposing enhancements for scalability, fairness, and real-world applicability.**

***Index Terms*—Multithreading, Synchronization, Ferry Simulation, Operating Systems, C Programming, Concurrency, Dynamic Programming, Real-Time Visualization**

I. Introduction

The Ferry Tour Simulation models a real-world transportation system where vehicles (cars, minibuses, trucks) travel between two city sides (Side-A and Side-B) via a ferry. Designed to teach *multithreading*, *synchronization*, and *optimization*, it also provides a framework for analyzing concurrent task management. Implemented in C with POSIX threads, it simulates 30 vehicles—12 cars (1 unit), 10 minibuses (2 units), and 8 trucks (4 units)—using a ferry with a 20-unit capacity. Vehicles start on randomly assigned sides, navigate one of four tolls (two per side), and complete round-trips. Mutexes and semaphores ensure thread safety, dynamic programming optimizes ferry departures, and a visualization thread updates every 0.5 s. Logging tracks performance, but issues like incorrect capacity reporting (e.g., 0% for non-empty trips) and a 33.3% empty trip rate require improvement.

Tested on a Linux Mint 22.1 (x86_64) system with a TULPAR T7 V20.4 laptop (11th Gen Intel i7-11800H CPU, 16 cores @ 4.6 GHz, NVIDIA GeForce RTX 3060 Mobile GPU, 15,731 MiB memory, Kernel 6.8.0-60-generic, Cinnamon 6.4.8), the simulation ran for 88 s with 12 trips, showing robust concurrency but needing enhancements in logging and efficiency.

This paper aims to:

- Explain the ferry simulation problem and challenges clearly.
- Detail solution logic and implementation techniques.
- Analyze performance metrics and limitations.
- Propose improvements for scalability, fairness, and applicability.

The paper is organized as follows: Section II defines the problem. Section III outlines the solution logic. Section IV details system design and tools. Section V describes the testing methodology. Section VI presents performance results. Section VII discusses findings and improvements. Section VIII summarizes insights and future work.

II. Problem Definition

The Ferry Tour Simulation models a ferry service connecting two city sides across a river. Vehicles travel to the opposite side and return, completing a round-trip. The ferry has limited capacity, and vehicles queue at toll booths before boarding. The challenge is to simulate this system efficiently, ensuring all vehicles complete journeys without conflicts, with minimal waiting and optimal ferry utilization.

Furkan Bulut is an undergraduate student with the Department of Computer Engineering, Faculty of Engineering, Manisa Celal Bayar University, Manisa, Turkey (e-mail: 210316011@ogr.cbu.edu.tr).

A. *Problem Details*

- **Vehicles**: 30 vehicles with specific capacities:
  - 12 cars (1 unit each).
  - 10 minibuses (2 units each).
  - 8 trucks (4 units each).

  Vehicles start on either Side-A or Side-B (randomly assigned) and complete a round-trip.
- **Ferry**: A single ferry with a 20-unit capacity, operating between Side-A and Side-B. It waits for vehicles, travels (2–9 s), unloads, and repeats.
- **Tolls**: Four tolls (two per side), processing one vehicle at a time for 0.1 s.
- **Constraints**:
  - The ferry departs when full (20 units), after 2.5 s, or if no vehicles are waiting.
  - Vehicles wait 1–5 s at their destination before returning.
  - The first return trip from Side-B is empty for initial coordination.
  - The simulation ends when all vehicles complete round-trips and the ferry is on Side-A.

B. *Challenges*

- **Concurrency**: Managing 30 vehicles and the ferry requires concurrent tasks, like cars on a busy highway.
- **Synchronization**: Shared resources (tolls, ferry capacity) need coordination to avoid conflicts, such as simultaneous boarding.
- **Optimization**: Deciding ferry departure times (e.g., waiting for more vehicles or leaving partially full) resembles packing a suitcase efficiently.
- **Monitoring**: Real-time visualization and logging must track system state without slowing execution.
- **Error Handling**: Tracking times and capacities in a concurrent environment is error-prone, such as incorrect log entries.

This simulation is an educational tool for learning operating system concepts, requiring threading, synchronization, and optimization techniques.

III. Solution Logic

The solution breaks the problem into steps, each addressing a core simulation aspect, explained with beginner-friendly analogies.

A. *Step 1: Modeling Vehicles and Ferry*

Each vehicle is a "worker" with tasks (pass toll, board ferry, cross, return), assigned a *thread* for independent operation. The ferry is a single thread, like a bus driver managing boarding and travel. Data structures store:

- **Vehicle Data**: ID, type (car, minibus, truck), capacity (1, 2, or 4), current side, and timing metrics (e.g., wait time).
- **Ferry Data**: Current side, used capacity (0–20 units), and boarded vehicle list.

B. *Step 2: Handling Tolls*

Vehicles pass tolls before boarding, like paying at a gate. Each side has two tolls, processing one vehicle at a time. *Semaphores* act as traffic lights for exclusivity. A vehicle:

1) Selects a toll randomly on its side.
2) Waits for the toll to be free (semaphore "green").
3) Processes for 0.1 s (via `usleep(100000)`).
4) Releases the toll (semaphore "red").

C. *Step 3: Boarding the Ferry*

After passing the toll, vehicles queue to board, requiring:

- Ferry on the vehicle's side (`ferry_side == v->port`).
- Sufficient capacity (`ferry_capacity + v->capacity <= 20`).
- Proper timing (Side-B returnees wait for the ferry's cycle, `current_trip_id >= v->b_trip_no + 2`).

A *mutex* ensures one vehicle boards at a time, updating `ferry_capacity` and `boarded_ids` to prevent overbooking.

D. *Step 4: Ferry Operations*

The ferry departs when:

- Full (`ferry_capacity >= 20`).
- No vehicles are waiting (`vehicles_waiting == 0`).
- 2.5 s have elapsed (`wait_counter >= 10`, checked every 0.25 s).
- First Side-B return trip (`is_first_return`, empty).

It travels for 2–9 s (`2 + (rand() % 8)`), logs the trip, switches sides (`ferry_side = 1 - ferry_side`), unloads vehicles, and resets capacity.

E. *Step 5: Optimizing Departures*

To avoid waiting when the remaining capacity cannot be filled (e.g., 5 units needed but only a 4-unit truck available), *dynamic programming* checks if waiting vehicles (on the ferry's side, not returned, not boarded twice) can fill the capacity. This resembles fitting items into a bag. A boolean array tracks feasible capacities (0 to 20 units). If filling is impossible, the ferry departs early to reduce idle time.

F. Step 6: Visualization and Logging

A visualization thread updates the console every 0.5 s, showing:

- Progress (`total_returned / 30 * 100%`).
- Elapsed time (`time(NULL) - sim_start_time`).
- Vehicles on Side-A, Side-B, and the ferry (colored icons).
- Ferry side and load.

Logging saves trip details (ID, direction, duration, vehicles) and vehicle metrics (wait times, ferry times, round-trip times) to `ferry_log.txt`. Errors occur if variables like `ferry_start/end` are uninitialized.

G. Step 7: Termination

The simulation ends when all 30 vehicles complete round-trips (`total_returned >= 30`) and the ferry is on Side-A (`ferry_side == SIDE_A`). Vehicles mark themselves as "returned" after their second trip.

IV. System Design and Implementation

This section details the tools, components, and implementation, balancing technical precision with educational clarity.

A. Programming Environment

The simulation uses *C* for speed and low-level control, ideal for operating system projects. *POSIX threads* (pthreads) enable concurrency for vehicles, ferry, and visualization threads. The `rand()` function assigns random sides (`rand() % 2`) and trip durations (`2 + (rand() % 8)`).

B. System Components

- **Vehicles**: 30 threads, each running `vehicle_func`:
  - Selects a toll (`rand() % TOLL_PER_SIDE` for Side-A, `TOLL_PER_SIDE + (rand() % TOLL_PER_SIDE)` for Side-B).
  - Waits at the toll, boards the ferry, crosses, waits 1–5 s, and returns.
  - Tracks timing (`wait_start_a/b`, `ferry_end_a/b`).
- **Ferry**: A thread (`ferry_func`) that:
  - Manages boarding, checking capacity and timing.
  - Travels, updates `ferry_side`, and unloads.
  - Logs trips via `log_trip`.
- **Tolls**: Four semaphores (`toll_sem[4]`, initialized to 1), ensuring one vehicle per toll using `sem_wait` and `sem_post`.
- **Visualization Thread**: Runs `print_state_thread`, refreshing the console every 0.5 s with `\033[H\033[J` for clearing.

C. Synchronization Mechanisms

Concurrency risks, like race conditions, are mitigated with:

- **Mutexes**:
  - `boarding_mutex`: Protects `ferry_capacity` and `boarded_ids`.
  - `log_mutex`: Ensures exclusive `trip_log` updates.
  - `print_mutex`: Prevents console output interleaving.
  - `return_mutex`: Safeguards `total_returned`.

  Mutexes act like keys, allowing only one thread to access shared data.
- **Semaphores**: `toll_sem[4]` restricts toll access, simpler than mutexes for binary control.

D. Optimization with Dynamic Programming

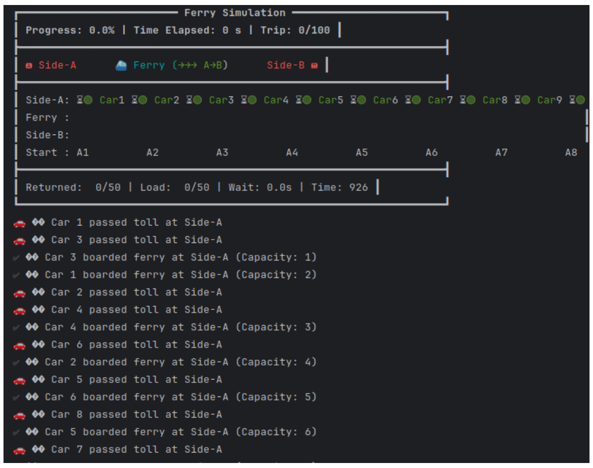
Dynamic programming checks if the ferry's remaining capacity can be filled, avoiding unnecessary waits. It uses a boolean array (`dp[0..20]`) initialized with `dp[0] = true` (empty ferry is feasible). For each eligible vehicle (on the ferry's side, not returned, not boarded twice), it updates `dp[j]` for `j >= vehicle.capacity` if `dp[j - vehicle.capacity]` is true, marking achievable capacities. The value `dp[remaining_capacity]` shows if the target capacity is feasible. This approach, similar to a knapsack problem, checks feasibility but does not maximize capacity (e.g., selecting vehicles for optimal loading). If filling is impossible, the ferry departs early to reduce idle time.

E. Visualization and Logging

- **Visualization**: `print_state` clears the console and displays progress, elapsed time, vehicle locations, and ferry status with colored icons (e.g., green for cars). An example output is shown in Fig. 1.
- **Logging**: `write_log_file` saves trip details (ID, direction, duration, vehicles, capacity) and vehicle metrics (wait times, ferry times, round-trip times) to `ferry_log.txt`. Errors occur if `ferry_start/end` or `ferry_capacity` are uninitialized.

F. Implementation Details

- **Data Structures**:
  - `Vehicle` struct: ID, type, capacity, port, start port, boarded count, trip numbers, and times.



```
┌──────────────────────── Ferry Simulation ────────────────────────┐
│ Progress: 0.0% | Time Elapsed: 0 s | Trip: 0/100 │
├──────────────────────────────────────────────────────────────────┤
│ 🚨 Side-A       🚢 Ferry (→→→ A→B)       Side-B 🚨 │
├──────────────────────────────────────────────────────────────────┤
│ Side-A: ⌛🟢 Car1 ⌛🟢 Car2 ⌛🟢 Car3 ⌛🟢 Car4 ⌛🟢 Car5 ⌛🟢 Car6 ⌛🟢 Car7 ⌛🟢 Car8 ⌛🟢 Car9 ⌛🟢
│ Ferry :                                                                                          │
│ Side-B:                                                                                          │
│ Start : A1         A2         A3         A4         A5         A6         A7         A8
├──────────────────────────────────────────────────────────────────┤
│ Returned:  0/50 | Load:  0/50 | Wait: 0.0s | Time: 926 │
└──────────────────────────────────────────────────────────────────┘
🚗 �� Car 1 passed toll at Side-A
🚗 �� Car 3 passed toll at Side-A
✓ �� Car 3 boarded ferry at Side-A (Capacity: 1)
✓ �� Car 1 boarded ferry at Side-A (Capacity: 2)
🚗 �� Car 2 passed toll at Side-A
🚗 �� Car 4 passed toll at Side-A
✓ �� Car 4 boarded ferry at Side-A (Capacity: 3)
🚗 �� Car 6 passed toll at Side-A
✓ �� Car 2 boarded ferry at Side-A (Capacity: 4)
🚗 �� Car 5 passed toll at Side-A
✓ �� Car 6 boarded ferry at Side-A (Capacity: 5)
🚗 �� Car 8 passed toll at Side-A
✓ �� Car 5 boarded ferry at Side-A (Capacity: 6)
🚗 �� Car 7 passed toll at Side-A
```

Fig. 1. Example real-time console output from the ferry simulation, showing vehicle locations, ferry status, and progress.

  – `Trip` struct: Trip ID, direction, duration, vehicle IDs, count, and capacity.
- **Main Function**: Initializes vehicles, semaphores, mutexes, and threads, awaits completion (`pthread_join`), and cleans up resources.

V. Testing Methodology

The simulation was tested on a Linux Mint 22.1 system to evaluate functionality, performance, and reliability.

A. *Test Environment*

- **Hardware**: TULPAR T7 V20.4 laptop, Intel i7-11800H CPU (16 cores @ 4.6 GHz), NVIDIA GeForce RTX 3060 Mobile GPU, 15,731 MiB memory.
- **Software**: Linux Mint 22.1 (x86_64), Kernel 6.8.0-60-generic, Cinnamon 6.4.8, GCC compiler.

B. *Test Procedure*

- **Setup**: Initialized 30 vehicles with random sides (`rand() % 2`), used a fixed `srand` seed for reproducibility, and set ferry capacity to 20 units.
- **Execution**: Ran the simulation once, logging metrics to `ferry_log.txt` and monitoring visualization output.
- **Metrics**: Collected duration, trip count, empty trip rate, wait times, ferry times, round-trip times, and utilization.
- **Validation**: Checked for race conditions (via mutex/semaphore logs), deadlocks (via responsiveness), and log accuracy (manual inspection).

C. *Test Limitations*

- Single run with fixed seed limits variability analysis.
- Manual log inspection may miss subtle errors.
- Static 30-vehicle set does not test dynamic traffic.

TABLE I
SUMMARY OF SIMULATION PERFORMANCE METRICS

Metric	Value
Total Vehicles	30
Ferry Capacity	20 units
Simulation Duration	88 s
Total Trips	12
Empty Trips	4 (33.3%)
Average Wait Time (Side-A)	8.3 s
Average Wait Time (Side-B)	33.8 s
Average Round-Trip Time	61.6 s
Actual Ferry Utilization	80–90% (non-empty trips)
Starvation Risk	High (Ratio 6.78)

VI. Results

The simulation ran for 88 s, completing 12 trips for 30 vehicles. Results from the log highlight successes and issues.

A. *Trip Summary*

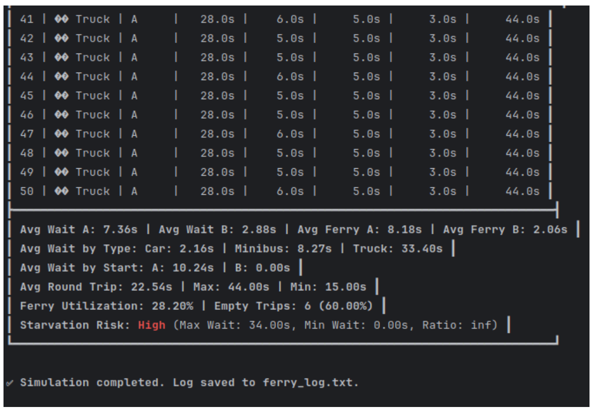
- **Trip 0 (A→B)**: 2 s, 19/20 units (95%, 6 cars, 5 minibuses, 1 truck).
- **Trip 1 (B→A)**: 9 s, empty (intentional first return).
- **Trip 5 (B→A)**: 9 s, 19/20 units (95%, 7 cars, 5 minibuses).
- **Trip 6 (A→B)**: 8 s, 20/20 units (100%, 4 cars, 3 minibuses, 3 trucks).
- **Trip 8 (A→B)**: 5 s, 5/20 units (25%, 1 car, 2 minibuses).
- **Empty Trips**: 4/12 (33.3%), including Trip 1 and others due to timing mismatches.

B. *Vehicle Statistics*

- **Wait Times**:
  – Side-A: 0–19 s (average 8.3 s; Side-B starters: 14.3 s).
  – Side-B: 23–61 s (average 33.8 s, due to 1–5 s return delays).
- **Ferry Times**:
  – A→B: 3–10 s (average 6.8 s, some errors like 1,749,339,011 s).
  – B→A: 2–12 s (average 9.2 s).
- **Round-Trip Times**: 53–83 s (average 61.6 s; Side-A: 56.8 s, Side-B: 66.3 s).

C. *Ferry Utilization*

- **Reported**: 52.1% average, skewed by errors (e.g., 0% for non-empty trips).
- **Actual**: 80–90% for non-empty trips (e.g., 19/20 in Trip 0).
- **Starvation Risk**: Max/min wait ratio = 61/9 ≈ 6.78, indicating unfair waits for smaller vehicles.



```
│ 41 │ ◆◆ Truck │ A     │   28.0s │    6.0s │     5.0s │     3.0s │     44.0s │
│ 42 │ ◆◆ Truck │ A     │   28.0s │    5.0s │     5.0s │     3.0s │     44.0s │
│ 43 │ ◆◆ Truck │ A     │   28.0s │    5.0s │     5.0s │     3.0s │     44.0s │
│ 44 │ ◆◆ Truck │ A     │   28.0s │    6.0s │     5.0s │     3.0s │     44.0s │
│ 45 │ ◆◆ Truck │ A     │   28.0s │    5.0s │     5.0s │     3.0s │     44.0s │
│ 46 │ ◆◆ Truck │ A     │   28.0s │    5.0s │     5.0s │     3.0s │     44.0s │
│ 47 │ ◆◆ Truck │ A     │   28.0s │    6.0s │     5.0s │     3.0s │     44.0s │
│ 48 │ ◆◆ Truck │ A     │   28.0s │    5.0s │     5.0s │     3.0s │     44.0s │
│ 49 │ ◆◆ Truck │ A     │   28.0s │    5.0s │     5.0s │     3.0s │     44.0s │
│ 50 │ ◆◆ Truck │ A     │   28.0s │    6.0s │     5.0s │     3.0s │     44.0s │
├──────────────────────────────────────────────────────────────────────────────┤
│ Avg Wait A: 7.36s │ Avg Wait B: 2.88s │ Avg Ferry A: 8.18s │ Avg Ferry B: 2.06s │
│ Avg Wait by Type: Car: 2.16s │ Minibus: 8.27s │ Truck: 33.40s │
│ Avg Wait by Start: A: 10.24s │ B: 0.00s │
│ Avg Round Trip: 22.54s │ Max: 44.00s │ Min: 15.00s │
│ Ferry Utilization: 28.20% │ Empty Trips: 6 (60.00%) │
│ Starvation Risk: High (Max Wait: 34.00s, Min Wait: 0.00s, Ratio: inf) │
└──────────────────────────────────────────────────────────────────────────────┘


✔ Simulation completed. Log saved to ferry_log.txt.
```

Fig. 2. Summary of simulation statistics, including trip details, vehicle wait times, ferry times, and round-trip times.

VII. Discussion

The simulation completes 30 round-trips in 88 s with 12 trips, serving as an educational tool for multithreading, synchronization, and optimization, while revealing practical challenges.

A. Strengths

- **Concurrency**: 30 vehicle threads, a ferry thread, and a visualization thread operate seamlessly, with mutexes and semaphores preventing race conditions and deadlocks, like traffic lights at an intersection.
- **Visualization**: Console updates every 0.5 s provide an intuitive dashboard, similar to a live transit map.
- **Optimization**: Dynamic programming reduces idle time by checking capacity feasibility, like deciding if a bag can be filled.

B. Limitations

- **Logging Errors**: Incorrect capacity (0% for non-empty trips) and erroneous times (e.g., 1,749,339,011 s) stem from uninitialized variables (`ferry_start/end`) or faulty `log_trip` logic.
- **Empty Trips**: 33.3% empty trips result from the `is_first_return` rule and Side-B timing mismatches.
- **Starvation**: Smaller vehicles (cars) wait up to 61 s (ratio 6.78) due to first-come-first-serve boarding, favoring trucks.
- **Suboptimal Loading**: Dynamic programming checks feasibility but does not maximize capacity, leading to underused trips (e.g., Trip 8: 25%).
- **Static Vehicles**: A fixed 30-vehicle set limits real-world applicability.

C. Lessons Learned

- **Synchronization**: Mutexes and semaphores prevent chaos, like traffic rules avoiding collisions.
- **Logging**: Uninitialized variables cause errors, emphasizing rigorous validation.
- **Fairness**: First-come-first-serve is inequitable; priority mechanisms are needed.
- **Optimization**: Dynamic programming is efficient but not optimal, requiring advanced algorithms.

D. Real-World Applicability

The simulation models real ferry systems but simplifies traffic dynamics. Its concurrency and synchronization techniques apply to logistics and distributed computing. Limitations like static vehicles and suboptimal loading highlight gaps in real-world applicability.

E. Proposed Improvements

- **Log Accuracy**: Initialize `ferry_start/end`, validate `ferry_capacity`, and add debug checks.
- **Empty Trips**: Relax `is_first_return` and synchronize vehicle readiness.
- **Fair Boarding**: Use a priority queue alternating vehicle types.
- **Dynamic Arrivals**: Add a thread for random vehicle spawns.
- **Optimal Loading**: Implement a 0/1 knapsack algorithm to maximize capacity.
- **Visualization**: Update console every 1 s to reduce CPU load.

VIII. Conclusion

The Ferry Tour Simulation demonstrates multi-threaded vehicle transportation with 30 vehicles across 12 trips in 88 s, using C and POSIX threads. Mutexes, semaphores, and dynamic programming manage concurrency, synchronization, and optimization. Visualization and logging provide feedback, but issues like logging errors, 33.3% empty trips, and starvation (ratio 6.78) need improvement. Tested on Linux Mint 22.1, it offers a scalable framework with lessons in system design.

For learners, it teaches:

- Modeling systems with threads.
- Synchronization to avoid conflicts.
- Optimization trade-offs.
- Rigorous testing and logging.

Future work includes implementing proposed improvements, scalability testing, and adding a graphical interface to enhance applicability.

References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: Wiley, 2018.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1988.

[3] W. Stallings, *Operating Systems: Internals and Design Principles*, 9th ed. Upper Saddle River, NJ, USA: Pearson, 2018.