CS 4323 Design and Implementation of Operating Systems I

Group Project: Full Marks 100
(Due Date and End Date: 04/20/2025, 11:59 PM CT)

This assignment is a group project that must be completed within the assigned group using either C or C++ programming language. If you are unsure about your group members, please consult the "Group Project" module located on the homepage of Canvas.

Students are encouraged to exercise their creativity and make assumptions, as long as they align with the requirements outlined in the assignment.

In this project, students will simulate a railway system where multiple trains (represented by processes) move along a track and share intersections. The goal is to manage how trains request and release intersections, ensuring that the number of trains in the intersection follows the limit at any time, while also preventing deadlock. You will simulate this system using concepts such as process forking, inter-process communication (IPC), synchronization, and deadlock detection and resolution.

Students are required to watch two videos relevant to this project. These videos can be found within the module labeled "Group Project." The first video, titled "Group Project Task Discussion and Submissions" outlines the specific tasks to be undertaken within the project and the submission guidelines. The second video, titled "Potential Challenges while working in a Group Project " addresses potential obstacles students may encounter during the project, provides guidance on project proceedings and clarifies expectations.

Project Introduction

In the Multi-Train Railway Intersection Control System project, process forking is a critical mechanism to simulate concurrent train operations. The system uses the parent-child process model, where each train is represented as an independently running child process created through fork(), allowing multiple trains to operate simultaneously. The parent process acts as the central controller (server), managing shared resources (intersections) and coordinating child processes.

The responsibilities of the parent process are as follows:

- **Initialization**: Reads the input configuration file (e.g., intersections.txt, trains.txt) to parse train routes and intersection requirements.
- **IPC Setup**: Creates shared memory segments (to track intersection availability) and message queues (for train-server communication).
- **Forking Trains**: Spawns a child process for each train using fork(). Each child inherits the parent's environment but runs independently.
- **Server Logic**: After forking, the parent transitions into the server role, handling requests from child processes and managing deadlocks.
- **Event-Driven Increment**: The server will maintain event based timing information for the logging purpose, which is the part of the output.

The responsibilities of the child process are as follows:

- **Train Behavior**: Each child process simulates a train's journey by iterating through its predefined route (e.g., acquiring and releasing intersections).
- **IPC Communication**: Uses message queues to send acquire/release requests to the server and waits for grants.
- **Progress Simulation**: Introduces delays (e.g., sleep()) to mimic traversal time between intersections.

Project Description

This section explains the project description in detail and some of the requirements that students must follow during the course of this project.

Intersection Types and Lock Mechanisms

The management of intersections relies on two distinct lock mechanisms tailored to their physical and operational constraints. For intersections that permit only one train at a time—such as narrow bridges or single-track sections—a mutex (binary semaphore) is employed. This mechanism ensures exclusive access, requiring a train to acquire the mutex lock before entering the intersection. For example, IntersectionA, designed with a capacity of 1, uses a mutex to guarantee that no two trains occupy the space simultaneously. In contrast, intersections capable of accommodating multiple trains—such as multi-track stations or wide junctions—leverage a counting semaphore. This semaphore is initialized to match the intersection's capacity, allowing concurrent access up to the defined limit. For instance, IntersectionB, with a capacity of 3, uses a counting semaphore initialized to 3, enabling up to three trains to occupy the space at once. Each time a train enters, the semaphore's count decreases, and upon exiting, the count is restored, ensuring efficient resource allocation. These lock mechanisms enable granular control over intersection access, balancing safety and throughput while reflecting real-world railway infrastructure constraints.

To simplify parsing in C and ensure clarity, the configuration file is split into two separate files:

- **File 1: intersections.txt**
  - Purpose: Define intersections and their capacities.
  - File Format: IntersectionName:Capacity
  - Example:

```
IntersectionA:1
IntersectionB:2
IntersectionC:1
IntersectionD:3
IntersectionE:1
```

  Each line specifies an intersection and its maximum concurrent trains.
  - 1 → Use a mutex (binary lock).
  - >1 → Use a semaphore with the specified capacity.

- **File 2: trains.txt**
  - Purpose: Define train names and their routes (ordered list of intersections).
  - Format: TrainName:Intersection1,Intersection2,...
  - Example:

```
Train1:IntersectionA,IntersectionB,IntersectionC
Train2:IntersectionB,IntersectionD,IntersectionE
Train3:IntersectionC,IntersectionD,IntersectionA
Train4:IntersectionE,IntersectionB,IntersectionD
```

Each line specifies a train and the sequence of intersections it must traverse. The route explanation is as follows:

- Train1: Acquires A (mutex) → B (semaphore) → C (mutex).
- Train2: Acquires B (semaphore) → D (semaphore) → E (mutex).
- Train3: Acquires C (mutex) → D (semaphore) → A (mutex).
- Train4: Acquires E (mutex) → B (semaphore) → D (semaphore).

Forking Workflow

The parent reads the input file and forks a child process for each train. Each child process then executes code to simulate its assigned train's route, while the parent becomes the server.

Inter-Process Communication (IPC)

After parsing the input text files, the parent configures shared memory segments to manage intersection resources. These segments store mutexes (implemented as pthread_mutex_t structures) for intersections with a capacity of 1, ensuring exclusive access for single-track sections like narrow bridges. For multi-track intersections (capacity > 1), the parent initializes counting semaphores (sem_t), setting their values to match the specified capacities—for example, a semaphore initialized to 3 for an intersection that can accommodate three trains simultaneously. Additionally, the shared memory includes a resource allocation table, which tracks active locks (e.g., IntersectionB: [Train2, Train5]), and synchronization metadata detailing intersection types (mutex/semaphore), capacities, and real-time lock states. This setup enables the server to enforce access rules and monitor resource usage efficiently.
The structure of the resource allocation table looks like this:

IntersectionID	Type	Capacity	Lock State	Holding Trains
IntersectionA	Mutex	1	Locked	[Train1]
IntersectionB	Semaphore	2	Locked	[Train1, Train2]
IntersectionC	Mutex	1	Locked	[Train3]
IntersectionD	Semaphore	3	Locked	[Train2, Train3]
IntersectionE	Mutex	1	Locked	[Train4]

Holding Trains:

- For the intersection that are of type mutex, only a single train ID can be accessing the intersection. (e.g., [Train3]).
- For the intersection that are of type semaphores, a list of up to capacity trains can share the intersection. (e.g., [Train2, Train5]).
-

The resource allocation table serves three critical functions in the system:

- It enables deadlock detection by allowing the server to construct a resource allocation graph, such as identifying scenarios where Train7 holds IntersectionA while requesting IntersectionB. Cycles within this graph signal deadlocks, such as interdependent resource claims that halt progress.

- The table supports conflict resolution during deadlocks. By querying the table, the server determines which trains or intersections to preempt (e.g., forcibly releasing a mutex or semaphore slot) to break the cycle and restore system flow.
- The table enforces request validation, ensuring trains cannot release intersections they do not currently hold. For instance, Train3 would be blocked from acquiring IntersectionA if the table shows it is held by Train1, preventing invalid or malicious resource claims. Together, these mechanisms ensure safe, efficient coordination of shared railway resources while maintaining system integrity.

To facilitate communication, the parent establishes message queues, which act as channels between the server and child processes. A request queue allows trains (child processes) to send ACQUIRE or RELEASE messages to the server, while a response queue enables the server to dispatch GRANT, WAIT, or DENY commands back to the trains. These queues ensure asynchronous, orderly communication, decoupling the server's decision-making logic from the trains' operational workflows.

Once the IPC infrastructure is in place, the parent forks child processes to simulate individual trains. Each child process receives its predefined route (e.g., Train3: [IntersectionC, IntersectionD, IntersectionA]) through shared memory during initialization. The child then begins executing its route, initiating an IPC workflow to navigate intersections. When approaching an intersection, the child sends an ACQUIRE request (e.g., ACQUIRE Train3 IntersectionC) via the request queue. The server evaluates the request based on the intersection type: for mutexes, it grants access if unlocked or queues the request if busy; for semaphores, it checks if the count permits entry (decrementing the count if space is available) or defers the request when full.

The child process waits on the response queue for a GRANT signal. Upon receipt, it simulates traversal by introducing a delay (e.g., sleep()), mimicking real-world travel time. After exiting the intersection, the child sends a RELEASE message, prompting the server to update the shared memory—unlocking mutexes or incrementing semaphore counts—and reassign the resource to the next queued train, if applicable. This cycle repeats until the child completes its route, ensuring synchronized, conflict-free movement across the railway network. Through this structured IPC design, the system balances concurrency with safety, replicating the complexities of real-world railway management.

Synchronization Mechanisms

In the Multi-Train Railway Intersection Control System, synchronization mechanisms ensure safe and efficient resource allocation across concurrent train operations. Mutexes (mutual exclusion locks) are employed to enforce exclusive access for intersections that can only accommodate a single train at a time. For instance, IntersectionA, designed with a capacity of 1, uses pthread_mutex_lock() to guarantee atomic access: once a train acquires the mutex, no other train can enter until it is released. This prevents collisions in critical sections like narrow bridges or single-track segments.

For intersections supporting multiple trains concurrently, semaphores manage scalable access. These counting semaphores are initialized to match the intersection's capacity, allowing a predefined number of trains to occupy the space simultaneously. For example, IntersectionD, with a capacity of 3, uses sem_wait() to decrement the semaphore count as trains enter and sem_post() to increment it upon exit. This ensures up to three trains can coexist in the intersection, ideal for multi-track stations or wide junctions.

To maintain consistency in shared memory operations—such as updating semaphore counts or mutex states—atomicity is enforced through additional safeguards. Critical operations, like modifying semaphore values or checking intersection availability, are guarded by auxiliary mutexes. This prevents race conditions where conflicting updates by concurrent processes could corrupt data. By serializing access to shared variables, the system ensures that all state changes occur indivisibly, preserving integrity across parallel train operations. Together, these mechanisms balance concurrency with safety, mirroring real-world railway management while addressing the complexities of distributed resource coordination.

Deadlock Handling

In the Multi-Train Railway Intersection Control System, deadlock handling is a critical function managed by the server to ensure uninterrupted operations. To detect deadlocks, the server periodically constructs a resource allocation graph using resource allocation table, which models the relationships between trains and intersections. In this graph, nodes represent trains and intersections, while edges define two types of dependencies: held resources (edges from a train to an intersection it currently occupies) and requested resources (edges from a train to an intersection it is waiting to acquire). A cycle in this graph may indicate a deadlock, where two or more trains are mutually blocked, each holding a resource the other needs to proceed.

Once a deadlock is identified, the server initiates resolution strategies to restore system flow. A common approach is preemption, where the server forcibly releases a resource held by one of the deadlocked trains. For example, it might revoke a mutex lock on an intersection or decrement a semaphore count to free capacity, allowing another train to proceed. This intervention may require the preempted train to temporarily backtrack or pause its route until the resource becomes available again. Alternatively, the server employs priority-based resolution, prioritizing requests based on criteria such as the age of the request (older requests are granted first) or the criticality of the intersection (e.g., high-traffic junctions receive precedence). By combining these strategies, the server ensures deadlocks are swiftly broken while minimizing disruption. This systematic approach guarantees continuous operation, balancing fairness and efficiency in a dynamic, resource-constrained environment.

Deadlock Explanation:

In the provided example, the deadlock occurred due to a circular dependency between Train1 and Train3, combined with resource contention from other trains.

Resource Allocation at Deadlock Point

Train1:

- Holds: IntersectionA (mutex, capacity 1).
- Holds: IntersectionB (semaphore, 1 of 2 slots used).
- Waits for: IntersectionC (held by Train3).

Train3:

- Holds: IntersectionC (mutex, capacity 1).
- Holds: IntersectionD (semaphore, 1 of 3 slots used).
- Waits for: IntersectionA (held by Train1).

Train2:

- Holds: IntersectionB (semaphore, 1 slot used).
- Holds: IntersectionD (semaphore, 1 slot used).
- Waits for: IntersectionE (held by Train4).

Train4:

- Holds: IntersectionE (mutex, capacity 1).
- Waits for: IntersectionB (semaphore full, 2/2 slots used).

Deadlock Conditions Met

Mutual Exclusion:

- Mutexes (A, C, E) allow only one train at a time.
- Semaphores (B, D) allow limited concurrent access.

Hold and Wait:

- Train1 holds A and B while waiting for C.
- Train3 holds C and D while waiting for A.
- Train4 holds E while waiting for B.

No Preemption:

- By default, the system does not forcibly take resources from trains unless deadlock detection triggers intervention.

Circular Wait:

- Cycle: Train1 → A → Train3 → C → Train1.
- Additional Blockage:
    - Train2 waits for E (held by Train4).
    - Train4 waits for B (full due to Train1 and Train2).

How the Deadlock Occurred

Initial Grants:

- Train1 acquires A and B.
- Train2 acquires B and D.
- Train3 acquires C and D.
- Train4 acquires E.

Deadlock Trigger:

- Train1 holds A and requests C (held by Train3).
- Train3 holds C and requests A (held by Train1).
- Neither can proceed, creating a cycle.
- Cycle: Train1 → IntersectionA → Train3 → IntersectionC → Train1

Secondary Blockage:

- Train2 and Train4 are blocked but not part of the detected cycle.

    - Train2 holds B and D, requests E (held by Train4).
    - Train4 holds E, requests B (semaphore full).
- Their dependencies add complexity but are resolved after the primary deadlock.

Deadlock Resolution

The server detected the cycle (Train1 ↔ Train3) using the resource allocation graph.

- Preemption Strategy:
    - Forcibly released IntersectionA from Train1.
    - Granted A to Train3, breaking the cycle.

Result:

- Train3 released C, allowing Train1 to proceed.
- Train1 then released A and B, freeing resources for Train4 and Train2.

Output File Specification

When all the trains have reached the destination, the program generates simulation.log file with:

- Timestamps, train actions, server responses, deadlock resolution steps.

Sample Output (simulation.log)

```
[00:00:00] SERVER: Initialized intersections:
- IntersectionA (Mutex, Capacity=1)
- IntersectionB (Semaphore, Capacity=2)
- IntersectionC (Mutex, Capacity=1)
- IntersectionD (Semaphore, Capacity=3)
- IntersectionE (Mutex, Capacity=1)

[00:00:01] TRAIN1: Sent ACQUIRE request for IntersectionA.
[00:00:01] SERVER: GRANTED IntersectionA to Train1.

[00:00:02] TRAIN1: Sent ACQUIRE request for IntersectionB.
[00:00:02] SERVER: GRANTED IntersectionB to Train1. Semaphore count: 1.

[00:00:03] TRAIN2: Sent ACQUIRE request for IntersectionB.
[00:00:03] SERVER: GRANTED IntersectionB to Train2. Semaphore count: 0.

[00:00:04] TRAIN2: Sent ACQUIRE request for IntersectionD.
[00:00:04] SERVER: GRANTED IntersectionD to Train2. Semaphore count: 2.

[00:00:05] TRAIN3: Sent ACQUIRE request for IntersectionC.
[00:00:05] SERVER: GRANTED IntersectionC to Train3.

[00:00:06] TRAIN3: Sent ACQUIRE request for IntersectionD.
[00:00:06] SERVER: GRANTED IntersectionD to Train3. Semaphore count: 1.

[00:00:07] TRAIN4: Sent ACQUIRE request for IntersectionE.
[00:00:07] SERVER: GRANTED IntersectionE to Train4.

[00:00:08] TRAIN1: Sent ACQUIRE request for IntersectionC.
[00:00:08] SERVER: IntersectionC is locked. Train1 added to wait queue.

[00:00:09] TRAIN2: Sent ACQUIRE request for IntersectionE.
[00:00:09] SERVER: IntersectionE is locked. Train2 added to wait queue.

[00:00:10] TRAIN3: Sent ACQUIRE request for IntersectionA.
[00:00:10] SERVER: IntersectionA is locked. Train3 added to wait queue.
```

```
[00:00:11] TRAIN4: Sent ACQUIRE request for IntersectionB.
[00:00:11] SERVER: IntersectionB is full. Train4 added to wait queue.

[00:00:12] SERVER: Deadlock detected! Cycle: Train1 ↔ Train3.
[00:00:12] SERVER: Preempting IntersectionA from Train1.
[00:00:12] SERVER: Train1 released IntersectionA forcibly.

[00:00:13] SERVER: GRANTED IntersectionA to Train3.
[00:00:13] TRAIN3: Acquired IntersectionA. Proceeding...

[00:00:14] TRAIN3: Released IntersectionC.
[00:00:14] SERVER: GRANTED IntersectionC to Train1.
[00:00:14] TRAIN1: Acquired IntersectionC. Proceeding...

[00:00:15] TRAIN1: Released IntersectionA and IntersectionB.
[00:00:15] SERVER: GRANTED IntersectionB to Train4. Semaphore count: 0.
[00:00:15] TRAIN4: Acquired IntersectionB. Proceeding...

[00:00:16] SIMULATION COMPLETE. All trains reached destinations.
```

Explanation of Timestamps in the Output File

The timestamps in the sample output (e.g., [00:00:01]) represent simulated time, not real-world time. The inclusion of timestamps in the output file implies the need for a simulated internal clock to log events in a sequential and logical order. The core components for this design includes:

- Shared Counter (sim_time): An integer stored in shared memory, incremented on key events.
- Synchronization Primitive: A mutex (pthread_mutex_t) in shared memory to ensure atomic updates.
- Event-Driven Increment: Time advances when events occur (e.g., requests, grants, releases).

Here is how this can be implemented:

Event-Based Increments

Every significant action in the system—such as a train sending an ACQUIRE request, the server granting access to an intersection, or a train releasing a resource—triggers an increment of the simulated time counter (sim_time). This approach ensures that each event is timestamped in the order it is processed, creating a causally consistent log. For example:

- When Train1 sends an ACQUIRE request for IntersectionA, the server increments sim_time by 1 unit and logs the event at the new timestamp.
- When the server grants the request, sim_time increments again, reflecting the progression of the simulation.

This rule guarantees that the log entries mirror the sequence of actions as they would occur in a real system, even though the processes run concurrently. By tying time increments to discrete events, the system avoids ambiguity and ensures reproducibility, which is critical for debugging and analysis.

Traversal Delay Simulation

To mimic real-world traversal times (e.g., a train moving through an intersection), the system uses simulated delays. For instance, when a train acquires an intersection, it might invoke a sleep(2)

function to represent the time taken to pass through. Instead of relying on real-time delays, the system translates these pauses into increments of sim_time. For example:

- If a train "sleeps" for 2 simulated units, sim_time advances by 2, and the log reflects this progression (e.g., [00:00:05] after a 2-unit delay starting at [00:00:03]).

This abstraction decouples the simulation from real-world clock dependencies, allowing the system to model time predictably. It also ensures that the log remains focused on logical progression rather than unpredictable execution speeds.

Server as Central Timekeeper

The server process acts as the sole authority for advancing sim_time, maintaining consistency across all processes. When handling events from message queues (e.g., requests from trains), the server:

- Locks a mutex (time_mutex) to ensure atomic updates to sim_time.
- Increments sim_time based on the event type (e.g., +1 for a request, +2 for a traversal delay).
- Logs the event with the updated timestamp.
- Releases the mutex, allowing other processes to proceed.

By centralizing timekeeping, the server prevents race conditions and ensures that all processes observe a unified timeline. This design aligns with the project's reliance on the server for resource management and deadlock resolution, reinforcing its role as the system's control hub.

Log Formatting

The simulated time counter (sim_time) is formatted into a human-readable [HH:MM:SS] timestamp for clarity. For example:

- sim_time = 125 becomes [00:02:05] (125 seconds = 2 minutes, 5 seconds).
- This formatting makes the logs intuitive, even though the time is entirely simulated. The server calculates these values using simple arithmetic:
  - Hours: sim_time / 3600
  - Minutes: (sim_time % 3600) / 60
  - Seconds: sim_time % 60

Synchronization and Determinism

To preserve event order, the server processes message queues in FIFO (First-In-First-Out) order. This ensures that even if multiple trains send requests simultaneously, their actions are logged in the sequence they were received. Combined with mutex-protected time increments, this guarantees deterministic logs—a crucial feature for replicating and diagnosing issues like deadlocks.

Project Milestones and Submissions:

This breakdown ensures students stay on track while progressively building their project, reducing last-minute panic. Each milestone has clear deliverables and submission requirements. Weekly submission will be graded based on the deliverables specified for each week.

Week 1: Project Setup and Requirement Analysis

Objectives:

- Understand the project requirements, expected outputs, grading rubrics and submission guidelines.
- Get familiar with group members and distribute responsibilities.
- Review provided videos:
  - "Group Project Task Discussion and Submissions"
  - "Potential Challenges while Working in a Group Project"
- Discuss the project plan and set internal deadlines.

Tasks:

Individual Tasks:

- Read and understand the project description, requirements, and objectives.
- Watch the provided project videos.
- Research IPC mechanisms, process forking, mutex, and semaphore concepts.

Group Tasks:

- Introduce team members and assign roles (e.g., project coordinator, coder, tester, documenter).
- Draft a high-level design document covering:
  - System architecture
  - Role of parent and child processes
  - How IPC will be implemented
- Create a shared repository (GitHub/Bitbucket/…) and set up version control.

Deliverables (Due by End of Week 01):

- Submission 1:
  - High-Level Design Document (2-3 pages, PDF format), which should include:
    - high-level breakdown of tasks for Weeks 2–4. Role of each student for weeks 2-4 should be clearly specified.
    - meeting schedule should be clearly specified in the document. There must be at least 2 meeting a week. For the meeting to be more productive it is advised to have at least a day gap in between.
    - list of questions/clarifications that needs to be resolved.
- Submission 2:
  - Weekly Peer Evaluation Form.

Week 2: System Design and Initial Implementation

Objectives:

- Design and implement the process forking.
- Set up IPC mechanisms for inter-process communication and basic train-server communication.
- Define how trains and intersections are represented in the program.

Tasks:

Group Tasks:

- Parse Configuration Files:
  - Read intersections.txt and trains.txt to extract intersection capacities and train routes.
- Shared Memory Setup:
  - Create shared memory segments to store intersection states (e.g., pthread_mutex_t for mutexes, sem_t for semaphores).
- Forking Trains:
  - Parent process forks child processes (trains) and initializes their routes.
- Basic IPC Workflow:

    - Implement message queues for ACQUIRE/RELEASE requests and server responses.
    - Simulate train movement without synchronization (e.g., ignore deadlocks).
- Implement mutex and semaphore locks based on intersection type.

Testing Tasks:

- Ensure that trains can fork successfully.
- Verify message queues for basic communication.

Deliverables (Due by End of Week 01):

- Submission 1:
    - Code Submission:
        - Basic working code that:
            - Forks processes.
            - Parses input files.
            - Sends/receives messages via queues.
- Submission 2:
    - A 2-page technical report explaining:
        - Each group member needs to clearly specify the current implementation status (what responsibility was taken and how much was it delivered by that week).
        - Problems faced by the group and how they were tackled.
        - Any changes made to the initial high-level design.
- Submission 3:
    - Weekly Peer Evaluation Form.

Week 3: Synchronization and Deadlock Detection

Objectives:

- Implement mutex/semaphore-based synchronization.
- Implement event-driven logging.
- Implement deadlock detection.

Tasks:

Group Tasks:

- Synchronization Mechanisms:
    - Use mutexes for single-capacity intersections (e.g., pthread_mutex_lock()).
    - Use semaphores for multi-capacity intersections (e.g., sem_wait()).
- Resource Allocation Table:
    - Track held intersections and waiting trains in shared memory.
- Deadlock Detection:
    - Implement a resource allocation graph to detect cycles (e.g., check for Train1 → IntersectionA → Train3 → IntersectionC → Train1).
- Simulation Logging:
    - Start logging events (e.g., TRAIN1: Sent ACQUIRE request).

Testing Tasks:

- Simulate different train schedules.
- Introduce deadlock scenarios.

Deliverables (Due by End of Week 01):

- Submission 1:
    - Code Submission:
        - Basic working code that:

- Fully functional synchronization and deadlock detection.
- deadlock handling and logging (partial simulation.log showing grant/release events).

- Submission 2:
  - A 2-page technical report covering:
    - Illustration of fully functional IPC and the synchronization mechanisms.
    - Deadlock scenarios encountered.
    - Describe 2–3 test scenarios (e.g., deadlock between two trains).
    - Testing results and example logs.
    - Each group member needs to clearly specify the current implementation status (what responsibility was taken and how much was it delivered by that week).
    - Problems faced by the group and how they were tackled.
    - Any changes made to the initial high-level design.
- Submission 3:
  - Weekly Peer Evaluation Form.

Week 4: Deadlock Resolution and Final Testing/Debugging

Objectives:

- Resolve deadlocks and finalize logging.
- Conduct extensive testing with various train schedules.
- Finalize documentation and code.

Tasks:

Group Tasks:

- Deadlock Resolution:
  - Implement preemption (e.g., forcibly release a resource) or priority-based resolution.
- Simulated Timekeeping:
  - Add timestamps to simulation.log using a shared sim_time counter.
- Edge Cases & Testing:
  - Test scenarios where trains backtrack after preemption.
  - Ensure no two trains occupy a mutex-protected intersection simultaneously.
- Prepare the final report, including:
  - System architecture.
  - Detailed implementation of IPC, synchronization, and deadlock handling.
  - Testing methodology and explain the results.
- Ensure code is well-documented.

Deliverables (Due by End of Week 01):

- Submission 1:
  - Complete and well-commented source code.
- Submission 2:
  - Final report in PDF format
- Submission 3:
  - simulation.log file from test runs.
- Submission 4:
  - Final Peer Evaluation Form.

Submission Guidelines:

All submissions must be made through Canvas. All the programs must be tested on CSX server. TA will use CSX server to test the program.

Progress Report (For Week 1-3):

- Each group must submit one progress report on Canvas under the link named "Week0X Progress Submission," where "X" represents the week number (1, 2, or 3).
- The group should designate one member responsible for submitting everything on Canvas on behalf of the entire group.

Self and Peer Evaluation Form (For Week 1-3):

- Every individual group member must submit the Self and Peer Evaluation Form on Canvas under the link named "Self and Peer Evaluation Form for Week0X)", where "X" represents the week number (1, 2, or 3).
- It's crucial to emphasize that the Self and Peer Evaluation Form will be given significant consideration during grading since the TA and instructor may not have direct visibility into the project activities. Therefore, it's imperative for each group member to provide comprehensive information in their evaluations. Insufficient information may hinder the grading process and affect fairness. It is the responsibility of the group to ensure that all relevant details are provided in the final submission of the Self and Peer Evaluation Form.

Final Submissions (For Week 4):

- Final Report:
  - Each group must submit one final report on Canvas under the link named "Week04 Final Project Submission".
  - The group should designate one member responsible for submitting everything on Canvas on behalf of the entire group.
- Self and Peer Evaluation Form:
  - Every individual group member must submit the Self and Peer Evaluation Form on Canvas under the link named "Final Self and Peer Evaluation Form".

Source Code Requirements:

1. Individual files for each student's work:
   - Each student's work must be in a separate file.
   - Each file should include header information with the following details:
     - Group number
     - Author of the source file (or the code): Must have one author for each file. If there are multiple authors in the file, the TA will consider the first author as the owner of the file and grades will be assigned accordingly.
     - Email
     - Date

- The code should include sufficient comments, with clear descriptions of each function, including input arguments and return values.

2. Code Functionality:
   - The code must function as a cohesive project when executed, as the TA will not run each individual student's work separately.

Adhering to these submission guidelines ensures clarity, organization, and completeness in the project evaluation process.

Points to remember:

- Submit Reports and Evaluation Forms Promptly:
  - Avoid waiting until the last moment to submit reports or the evaluation form.
  - The due date and end date are the same.
  - If a group or member misses the deadline, the submission link will be unavailable.
- Submission Process:
  - After submitting report/codes on Canvas, the submitting member must email the TA, cc'ing the instructor and all group members, to confirm the submission.
  - If any group member disputes the submission, they must inform the TA, cc'ing the instructor, within 24 hours.
- Use of Group Page on Canvas:
  - A group page has been created on Canvas, accessible through "People" on the left side of the Canvas page.
  - It's highly recommended that all group members use this platform for communication and file sharing.
  - In case of disputes, information shared on this page will be used for resolution.

Grading Criteria:

This group project will be assessed based on students' ability to collaborate effectively as a team and successfully complete the assigned tasks. The emphasis will be on teamwork not just individual efforts, with coordination among group members being crucial for achieving maximum points.

While there are weekly submissions, grades for these submissions will be finalized during the final project evaluation. This approach allows for an overall assessment of the project's progress and the contributions of each group member.

Grading Rubric:

1. Weekly Progress Report (6 Points)

Criteria	Points	Description
Peer Evaluation (Week 1 Submission)	6	**Each member submits a peer evaluation form describing contributions and communication efforts.**

2. Parent Process Implementation (15 Points)

Criteria	Points	Description
Reading and Parsing Input Files	3	Correct parsing of intersections.txt and trains.txt with proper error handling.
Shared Memory Setup	4	Correctly initializes shared memory segments to store intersection states and train metadata.
Message Queue Setup	4	Implements message queues for communication between server and train processes.
Train Forking (Creating Child Processes)	4	Parent process successfully forks a train process for each train defined in trains.txt.

3. Train (Child Process) Implementation (15 Points)

Criteria	Points	Description
Train Route Handling	4	Each train correctly follows its predefined route from trains.txt.
Train Requests Intersections via IPC	5	Sends requests to acquire and release intersections using message queues.
Simulated Delays for Train Movement	3	Uses appropriate delays (e.g., sleep()) to simulate train movement.
Handles Intersection Responses Properly	3	Correctly processes server responses (GRANT, WAIT, DENY) and waits when necessary.

4. Intersection Synchronization (20 Points)

Criteria	Points	Description
Mutex Implementation for Single-Train Intersections	5	Correct use of pthread_mutex_t for exclusive intersections.
Semaphore Implementation for Multi-Train Intersections	5	Proper use of sem_t to manage limited-capacity intersections.
Resource Allocation Table Management	5	Server maintains and updates the resource allocation table accurately.
Prevention of Invalid Acquisitions/Releases	5	Ensures that trains cannot release intersections they do not hold and prevents race conditions.

5. Deadlock Detection & Resolution (20 Points)

Criteria	Points	Description

Resource Allocation Graph Construction	5	Server correctly models resource allocation as a graph.
Deadlock Detection Algorithm	5	Detects circular waits and logs deadlock occurrences.
Deadlock Resolution via Preemption	5	Correctly selects a train to preempt and restores system flow.
Effectiveness of Deadlock Resolution	5	System resumes normal operation after deadlock resolution without additional issues.

6. Logging & Output Specification (10 Points)

Criteria	Points	Description
Event Logging Mechanism	4	Logs all train requests, grants, releases, and deadlock events with timestamps.
Format & Readability of simulation.log	3	Clear, structured output following timestamp format ([HH:MM:SS] style).
Correctness of Logging Information	3	Logs match expected events based on test cases.

7. Final Report & Documentation (7 Points)

Criteria	Points	Description
Final Report (5+ Pages)	5	Well-structured, covering architecture, implementation details, and testing results.
Code Documentation & Readability	2	Code contains meaningful comments and follows consistent formatting.

8. Weekly Peer Evaluations (7 Points)

Criteria	Points	Description
Submission of Peer Evaluations	7	Each student submits a form every week assessing contributions of group members.

Note:

- Each group member is responsible for ensuring the project and source code are not plagiarized.
  - Plagiarism will result in penalties as specified in the syllabus.
  - Plagiarized parts will be disregarded, and the project will be considered incomplete.
  - These guidelines are subject to the University's strict policy on plagiarism and will be strictly enforced.
- Failure of the program to compile or runtime errors on the CSX machine will result in a penalty of -20 points for each member.
- Any incomplete project will incur a penalty of -20 points for all group members.
- Issues arising from individual work not functioning correctly when merged into the group project will be considered as an incomplete project.
- Points distribution may not be equal among group members and will depend on individual responsibilities, task delivery, and contribution to the project.