# Project Proposal: Multi-Threading Ice Cream Factory

22k-4489, 22k-4415, 22k-4242

## 1. Introduction:

We propose the development of an Ice Cream Factory system utilizing multi-threading techniques. This system aims to efficiently produce ice creams with various flavours concurrently, enhancing production throughput and responsiveness. The project comprises both client and server components, communicating via inter-process communication with shared memory. Through this approach, we aim to demonstrate the effectiveness of multi-threading in a real-world scenario.

## 2. Objectives:

-Implement a multi-threaded Ice Cream Factory system.
-Utilize inter-process communication with shared memory for efficient data exchange between client and server.
-Demonstrate concurrent execution of threads to simulate the production process at different counters.
-Showcase the allocation and deallocation of shared memory segments for ice cream components.
-Implement local semaphore mechanisms to synchronize access to shared resources.
-Provide a seamless user experience through intuitive feedback on successful ice cream creation.

## 3. System Overview:

The Ice Cream Factory system consists of two main components: the client and the server. The server is responsible for managing shared memory and coordinating the production process. It allocates memory segments for each ice cream thread, assigns flavours, and oversees the production flow. On the other hand, the client represents the various counters where ice creams are made. Each counter is represented by a separate thread that produces ice creams concurrently.

## 4. Implementation Details:

**Server Component:**
-Allocates shared memory segments for ice cream threads.
-Assigns flavours to ice cream threads and coordinates the production flow.
-Detaches and deallocates shared memory segments upon completion.
**Client Component:**
-Allocates and attaches shared memory segments for ice cream components.
-Utilizes local semaphore mechanisms to synchronize access to shared resources.
-Displays feedback upon successful creation of ice creams.

## 5. Conclusion:

The proposed Ice Cream Factory system leverages multi-threading and inter-process communication to efficiently produce ice creams with various flavours. By simulating the production process across multiple counters, we aim to demonstrate the benefits of concurrent execution and shared memory management in a real-world scenario. This project aligns with the goal of showcasing advanced programming techniques in practical applications.

Algorithm: Ice Cream Factory Process

- Initialization:
  - Include necessary header files for system calls, shared memory operations, and standard I/O.
  - Declare required variables and constants.
  - Define key for shared memory and specify the size of shared memory segment.
- Server Process (Producer):
  - Allocate a shared memory segment using `shmget()` with appropriate permissions.
  - Attach the shared memory segment to the server's address space using `shmat()`.
  - Write data to the shared memory segment:
    - Define an array `next[10]` to store the lengths of strings.
    - Calculate the positions for writing each string in shared memory.
    - Use `sprintf()` to write strings into the shared memory segment.
    - Copy the `next` array and data into the shared memory segment using `memcpy()`.
  - Execute the client process using `system("./cli")`.
  - Detach the shared memory segment using `shmdt()`.
  - Deallocate the shared memory segment using `shmctl()` with `IPC_RMID` command.
- Client Process (Consumer):
  - Include necessary header files.
  - Define necessary functions and variables for multithreading and synchronization.
  - Attach the shared memory segment to the client's address space using `shmat()`.
  - Retrieve data from the shared memory segment:
    - Calculate the positions for reading each string from shared memory.
    - Extract strings from the shared memory segment.
  - Initialize a mutex semaphore for synchronization using `sem_init()`.
  - Create multiple threads (depending on user input) to process each ice cream order:
    - Each thread executes a handler function, processing a specific counter's task.
    - The handler functions simulate processing time for each order using `sleep()` to represent manufacturing and packaging.
    - Use semaphore `sem_wait()` and `sem_post()` for mutual exclusion to ensure only one thread accesses the critical region at a time.
  - After processing all threads, destroy the mutex semaphore using `sem_destroy()`.
  - Exit the program.
- Handler Functions:
  - Define handler functions corresponding to each counter.
  - Inside each handler function, simulate the processing of an ice cream order:
    - Print the arrival of the ice cream order at the counter.
    - Enter the critical region using semaphore `sem_wait()`.

        - Simulate the processing time for manufacturing or packaging using `sleep()`.
        - Print completion of the counter's task.
        - Exit the critical region using semaphore `sem_post()`.
        - Exit the thread.

- End of Algorithm