Safari Hackathon

11 May 2025

Disclaimer: Two demonstration videos have been submitted along with this report.
(1) A regular usage video showing 3 lakes in action.
(2) A scalability demonstration video with a random number of lakes (3–5).
These videos showcase both normal functionality and the system's ability to scale under varying conditions.

1. Introduction and Overview

The system centers around an abstract `Animal` class, from which three concrete subclasses are derived: `Flamingo`, `Zebra`, and `Hippo`. Each subclass defines species-specific behaviors, such as preferred drinking conditions and mean drinking durations. Animals are instantiated at runtime and executed in their own threads, each attempting to access a shared lake resource.

Lakes are represented by the `Lake` class, which maintains an array of drinking slots, a semaphore to limit concurrent access, and logic to enforce animal-specific constraints (e.g., zebra pairs, hippo exclusivity). Mutual exclusion is enforced via a `lock` to avoid race conditions when updating lake state.

Two short demonstration videos have been submitted along with this report: one showing regular system operation with 3 lakes and a mix of animal types, and another demonstrating system scalability with a random number of lakes between 3 to 5. These highlight both correctness under typical conditions and robustness under varied system sizes.

2. Animal Creation and Execution

Each `Animal` has a unique ID assigned using an atomic operation which was used for tracking in testing:

```
this.ID = Interlocked.Increment(ref nextID);
```

This prevents race conditions during ID generation across multiple threads.

Animals are spawned in intervals determined by a Normal distribution and select a lake at random to attempt to drink from. Each animal is executed inside its own thread using a species-specific spawner. However, instead of running the drinking logic directly within the thread, the thread invokes an asynchronous method, `RunAsync()`, through a wrapper. This model is necessary due to the inability of threads to recieve functions which return Task objects (async functions). In addition to that, we can then use threads to simulate arrival and asynchronous tasks manage the drinking behavior. This allows us to enable drinking tasks to be cancelled cleanly using `CancellationToken`s, which is essential for enforcing the rule that a hippo must interrupt and clear all current drinkers before occupying the lake. Without this separation, ongoing tasks would either block threads or be difficult to interrupt safely. By using asynchronous operations for time-dependent actions and threads for independent arrivals, the system is able to balance parallelism with control while allowing the hippo instances the functionality they are required to have. This is how we implement the wrapper :

```
public void RunAsyncWrapper() {
    RunAsync().GetAwaiter().GetResult();
}
```

This wrapper bridges synchronous thread execution with async logic.

3. Lake Slot Management and Mutex Control

Each lake contains a list of slots:

```
private List<Animal> AnimalList;
private readonly object lockObj = new object();
```

We use a `lock` to guard access to the slot list, to prevent race condition as this is a writing code segment. This approach ensures thread safety while maintaining scalability as the number of animal threads grows, allowing multiple animals to interact with the lake concurrently without risking data corruption. This is how we implement it:

```
lock (lockObj) {
    // critical section
}
```

This ensures that only one thread can modify the lake's state at a time, as taught in class. This locking mechanism ensures correctness, but can become a scalability bottleneck if many animals contend for access at once, as it serializes access to the shared state.

4. Detailed Breakdown of `InsertAnimal` Method

The `InsertAnimal` method is a crucial part of the simulation that handles inserting animals into the lake in a multi-threaded environment. As the simulation runs multiple threads, each corresponding to an animal, we need to ensure that concurrent access to shared resources, such as the list of animals in the lake and the lake's available slots, is handled safely and efficiently. This is achieved using synchronization primitives like `lock`, `Semaphore`, and cancellation tokens.

The method's signature is:

```
public async Task<bool> InsertAnimal(Animal animal)
```

This method is marked as `async`, meaning it will run asynchronously and return a `Task<bool>`. The boolean return value indicates whether the animal was successfully inserted into the lake.

4.1 Concurrency Handling with Locks and Semaphores

In a multi-threaded environment, multiple animals might try to insert themselves into the lake simultaneously. To handle this, the method uses various synchronization mechanisms to prevent race conditions and ensure that the lake's state is modified safely.

4.1.1 The `lock` Statement

The `lock` statement ensures that only one thread can execute the critical section at a time, preventing concurrent modifications to shared resources. Specifically, the following line:

```
lock (lockObj)
```

protects the shared resources like the `AnimalList` (the list of animals currently occupying the lake slots) and the `hippoInLake` (a flag indicating whether a hippo is currently in the lake). This prevents multiple threads from modifying these shared resources simultaneously, which could lead to inconsistent or incorrect results.

The `lock` statement ensures that only one thread can execute this block of code at a time. If a thread is already executing within the critical section, any other thread attempting to enter it will be blocked until the lock is released.

4.1.2 Semaphore for Concurrent Access

While `lock` ensures mutual exclusion when modifying shared data, it does not manage concurrent access to limited resources such as the lake slots. To handle this, a `Semaphore` is used to control how many animals can drink from the lake at any given time. So, because semaphores allow controlled concurrent access without serializing all operations, they scale better than locks in scenarios where partial parallelism (like multiple simultaneous drinkers) is acceptable.

The `Semaphore` is initialized with the number of available slots in the lake, and animals must acquire a "slot" before they can start drinking. The following code ensures that a thread can asynchronously wait for a slot to become available:

```
await drinkingSemaphore.WaitAsync(token);
```

By using `WaitAsync`, the thread is allowed to asynchronously wait without blocking, enabling other tasks to continue running while the current task waits for an available slot. This non-blocking behavior is crucial for efficient simulation execution in a multi-threaded environment. Once the animal finishes drinking, it releases its slot:

```
drinkingSemaphore.Release();
```

This allows other animals to enter and drink in a fair manner, according to the lake's capacity.

4.2 Cancellation Logic and Handling of Special Cases

One of the key features of this simulation is the handling of interruptions, specifically when a hippo enters the lake. The hippo's presence forces all other animals to leave, and any active drinking tasks are canceled.

The method uses a `CancellationTokenSource` to manage task cancellation. When a hippo enters, it cancels all ongoing drinking tasks by calling the `Cancel` method on each `CancellationTokenSource`:

```
foreach (var kvp in cancellationTokens)
    kvp.Value.Cancel();
```

This ensures that all animals currently drinking are interrupted and forced to leave the lake. The cancellation token is passed to the `DrinkAsync` method, and if a task is canceled, it will throw an `OperationCanceledException`, which is caught to handle the task's cleanup.

4.3 Handling of Different Animal Types

Although the core logic of insertion is shared for all animal types, the method handles each animal type differently during the insertion process. Specifically:

- **Flamingos** (`"f"`) require the presence of a neighboring flamingo in the lake to begin drinking. If no adjacent place is found, only if there are no flamengos, do we insert the current flamengo. Incase there are flamengos but no adjacent open space the flamengo will be in waiting.
- **Zebras** (`"z"`) are inserted into the lake only if adjacent slots are available, either occupying one or two consecutive slots.
- **Hippos** (`"h"`) are inserted only if no other animals are present, and they force all other animals to leave the lake when they arrive.

4.4 Asynchronous Execution for Non-Blocking Behavior

The method is marked as `async` to allow asynchronous execution, which ensures that tasks like waiting for a slot or delaying for an animal to drink do not block the main thread. This allows the simulation to handle multiple animals concurrently, without blocking threads unnecessarily. For example, the `WaitAsync` call on the semaphore does not block the thread but instead frees up the thread to execute other tasks until the animal can acquire a slot.

This asynchronous behavior is essential for efficiently running a simulation with many animals, as it ensures that each animal can drink concurrently, without blocking other tasks. It also facilitates cancellation, as the asynchronous nature of the tasks allows for easy interruption by a hippo. Moreover, asynchronous execution improves scalability by avoiding thread-blocking, enabling the system to handle a large number of animals concurrently without exhausting system threads.

5. Asynchronous Drinking with Cancellation

Drinking is executed asynchronously to enable parallelized, non-blocking behavior and cancellation, especially for the hippo. Instead of blocking a thread while an animal drinks, the simulation uses the `async` keyword in the `DrinkAsync` method to start the drinking process without holding onto a thread. The thread can then be used for other tasks while waiting for the animal's drinking time to pass.

The `CancellationToken` allows mid-way interruptions of the drinking process. For example, when a hippo enters, all active drinking tasks are canceled, freeing up space for the hippo and ensuring no animals drink simultaneously with it. This mechanism avoids forcibly killing threads, which would be unsafe, and ensures responsiveness in the simulation.

Here's how the drinking process is implemented:

```
private async Task DrinkAsync(Animal animal, int index, bool isHippo
    , CancellationToken token)
{
    await drinkingSemaphore.WaitAsync(token);
    try
    {
        Logger.Log($"Lake {id}: {animal.getType()} #{animal.getId()}
            is drinking at slot {index}...");
        await Task.Delay(animal.drinkTime, token);  // Non-blocking
        Logger.Log($"Lake {id}: {animal.getType()} #{animal.getId()}
            finished drinking.");
    }
    catch (OperationCanceledException)
    {
        Logger.Log($"{animal.getType()} #{animal.getId()} was
           interrupted.");
    }
    finally
    {
        drinkingSemaphore.Release();  // Release the slot
```

```
    }
}
```

The `Task.Delay` call is also very crucial. It allows the simulation to run multiple animals drinking concurrently without blocking the system's threads. Meanwhile, the `CancellationToken` allows for task interruption when the hippo arrives. If interrupted, the task throws an `OperationCanceledException`, which safely stops the task.

When a hippo enters the lake, all ongoing drinking tasks are canceled using:

```
foreach (var kvp in cancellationTokens)
    kvp.Value.Cancel();  // Cancel all tasks
```

This ensures that the hippo has exclusive access to the lake.

Finally, the `Semaphore` ensures that no more than the allowed number of animals drink at the same time. according to their placed indexes done in the InsertAnimal method. This allows them all to be drinking at the same time while allowing them to be cancelled at any time should a hippo arrive, using:

```
await drinkingSemaphore.WaitAsync(token);  // Wait for an available
    slot
```

which waits asynchronously for a free slot in the lake, using a semaphore to limit the number of animals drinking at the same time, and supports cancellation via the provided token.

6. Zebra Cleanup

Zebras occupy two slots (or last and first slot, circular). When leaving:

```
if (index == 0 && type == "z") {
    AnimalList[slots - 1] = null;
} else if (type == "z") {
    AnimalList[index - 1] = null;
}
AnimalList[index] = null;
```

which ensures that both occupied slots are cleared.

7. Thread Spawning Logic

Each species spawns from its own thread:

```
new Thread(() => SpawnAnimals<Zebra>(...)).Start();
```

Spawned animals use Normally-distributed inter-arrival times using

```
TimeSpan delay = GetRandomNormalTime(arrivalMean);
```

Which generates random arrival times according to the distribution

8. Cancellation Token Tracking

Each drink task stores its own cancellation token, which is tied to the specific lake slot it occupies:

```
CancellationTokenSource cts = new CancellationTokenSource();
cancellationTokens[index] = cts;
```

This design allows for cancellation per drinking slot while allowing silmuntaneous drinking from each slot. By associating each drinking task with its own `CancellationTokenSource`, the system can cancel individual drinkers precisely and safely. This becomes crucial when a hippo enters the lake, which will cause all active tokens are iterated and canceled, immediately interrupting any ongoing drinking sessions.

`cancellationTokens` is a dictionary mapping slot indices to their corresponding tokens. This mapping allows the system to keep track of which animal is drinking in which slot and cancel those tasks when necessary:

```
foreach (var kvp in cancellationTokens)
    kvp.Value.Cancel();
```

Because each animal's drinking task checks its token (e.g., via `Task.Delay(..., token)`), the cancellation propagates cleanly and cooperatively, avoiding unsafe thread termination and allowing proper resource cleanup (e.g., freeing the slot, releasing the semaphore).

9. Conclusion

In the project, we have used the following learned techniques for handling threads and their dangers such as race condition and deadlocks:

Locking + Mutexes

We used `lock` and `Monitor` constructs to make sure there is mutual exclusion. This is critical in the asynchornious `InsertAnimal` method of the `Lake` class, where multiple animal threads may attempt to insert themselves simultaneously. For example, when a hippo enters the lake, the method acquires a lock to prevent other animals from modifying the lake state mid-operation. This ensures atomic updates to the `AnimalList` and the internal `hippoInLake` flag, maintaining data consistency.

Semaphores

To enforce the physical limitation on the number of concurrent drinkers by the number of slots, each lake uses a `Semaphore`. Before an animal can begin drinking, it must be allowed in via `await drinkingSemaphore.WaitAsync(token)`. For instance, in a lake with 5 available slots, only the animals that got in can be drinking at any given moment. Any additional animals will wait asynchronously until a permit is released, ensuring fair access while avoiding thread blocking, unless of course a hippo is inserted in which case we use tokenization to cancel the asynchronious tasks

Asynchronous Tasks

Drinking is implemented as an asynchronous task via the `DrinkAsync` method. This allows us to avoid dedicating an entire thread to each drinking session. For example, a flamingo starts drinking with a call to `Task.Delay(animal.drinkTime, token)`. The task yields control while it waits, allowing the system to handle dozens of simultaneous drinkers without incurring the overhead of creating or blocking threads.

Cancellation Tokens

Each animal's drinking task is associated with a unique `CancellationTokenSource`, stored by slot index in a dictionary. This allows fine-grained interruption of individual tasks. When a hippo arrives, the system iterates over all active tokens and cancels them with `cts.Cancel()`, immediately stopping any ongoing `Task.Delay()` calls. This ensures that the hippo gains exclusive access to the lake without needing to forcibly terminate threads. For example, a drinking zebra in slot 3 is interrupted by cancellation rather than being removed forcefully.

Integrated Design

Each animal species is assigned a dedicated background thread responsible for continuously spawning new animals over time, simulating independent and natural arrival patterns. After an animal is created and assigned to a lake, its behavior such as waiting to drink, drinking, or being interrupted is handled asynchronously. This separation of concerns, where threads are used for arrival and asynchronous tasks are used for interaction, ensures both realism in timing and efficient use of system resources.