

CA-3

Cat-And-Mice-Synchronization

* **SUBMITTED TO: SUBMITTED BY:**

Mrs. Shivali Mam Utkarsh Chaturvedi

# **REG NO:** 12201820

**Email Address:** utkarshchaturvedi0001@gmail.com

DATE OF SUBMISSION: 01-Nov-23

* **A number of cats and mice inhabit a house. The cats and mice have worked out a deal where the mice can steal pieces of the cats’ food, so long as the cats never see the mice actually doing so. If the cats see the mice, then the cats must eat the mice (or else lose face with all of their cat friends). There are NumBowls cat food dishes, NumCats cats, and NumMice mice. Your job is to synchronize the cats and mice so that the following requirements are satisfied: No mouse should ever get eaten. You should assume that if a cat is eating at a food dish, any mouse attempting to eat from that dish or any other food dish will be seen and eaten. When cats aren’t eating, they will not see mice eating. In other words, this requirement states that if a cat is eating from any bowl, then no mouse should be eating from any bowl. Only one mouse or one cat may eat from a given dish at any one time. Neither cats nor mice should starve. A cat or mouse that wants to eat should eventually be able to eat. For example, a synchronization solution that permanently prevents all mice from eating would be unacceptable. When we actually test your solution, each simulated cat and mouse will only eat a finite number of times; however, even if the simulation were allowed to run forever, neither cats nor mice should starve.**

## **Code :}**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NumBowls 5

#define NumCats 3

#define NumMice 3

sem\_t cat\_sem;

sem\_t mouse\_sem;

sem\_t cat\_mutex;

sem\_t mouse\_mutex;

int cat\_count = 0;

int mouse\_count = 0;

void\* cat(void\* arg) {

int id = (int)arg;

while (1) {

sem\_wait(&cat\_mutex);

cat\_count++;

if (cat\_count == 1) {

sem\_wait(&mouse\_sem);

}

sem\_post(&cat\_mutex);

printf("Cat %d is eating.\n", id);

sleep(1);

sem\_wait(&cat\_mutex);

cat\_count--;

if (cat\_count == 0) {

sem\_post(&mouse\_sem);

}

sem\_post(&cat\_mutex);

printf("Cat %d is resting.\n", id);

sleep(1);

}

}

void\* mouse(void\* arg) {

int id = (int)arg;

while (1) {

sem\_wait(&mouse\_mutex);

mouse\_count++;

if (mouse\_count == 1) {

sem\_wait(&cat\_sem);

}

sem\_post(&mouse\_mutex);

printf("Mouse %d is eating.\n", id);

sleep(1);

sem\_wait(&mouse\_mutex);

mouse\_count--;

if (mouse\_count == 0) {

sem\_post(&cat\_sem);

}

sem\_post(&mouse\_mutex);

printf("Mouse %d is resting.\n", id);

sleep(1);

}

}

int main() {

pthread\_t cats[NumCats];

pthread\_t mice[NumMice];

int cat\_ids[NumCats];

int mouse\_ids[NumMice];

sem\_init(&cat\_sem, 0, 1);

sem\_init(&mouse\_sem, 0, 1);

sem\_init(&cat\_mutex, 0, 1);

sem\_init(&mouse\_mutex, 0, 1);

for (int i = 0; i < NumCats; i++) {

cat\_ids[i] = i + 1;

pthread\_create(&cats[i], NULL, cat, &cat\_ids[i]);

}

for (int i = 0; i < NumMice; i++) {

mouse\_ids[i] = i + 1;

pthread\_create(&mice[i], NULL, mouse, &mouse\_ids[i]);

}

for (int i = 0; i < NumCats; i++) {

pthread\_join(cats[i], NULL);

}

for (int i = 0; i < NumMice; i++) {

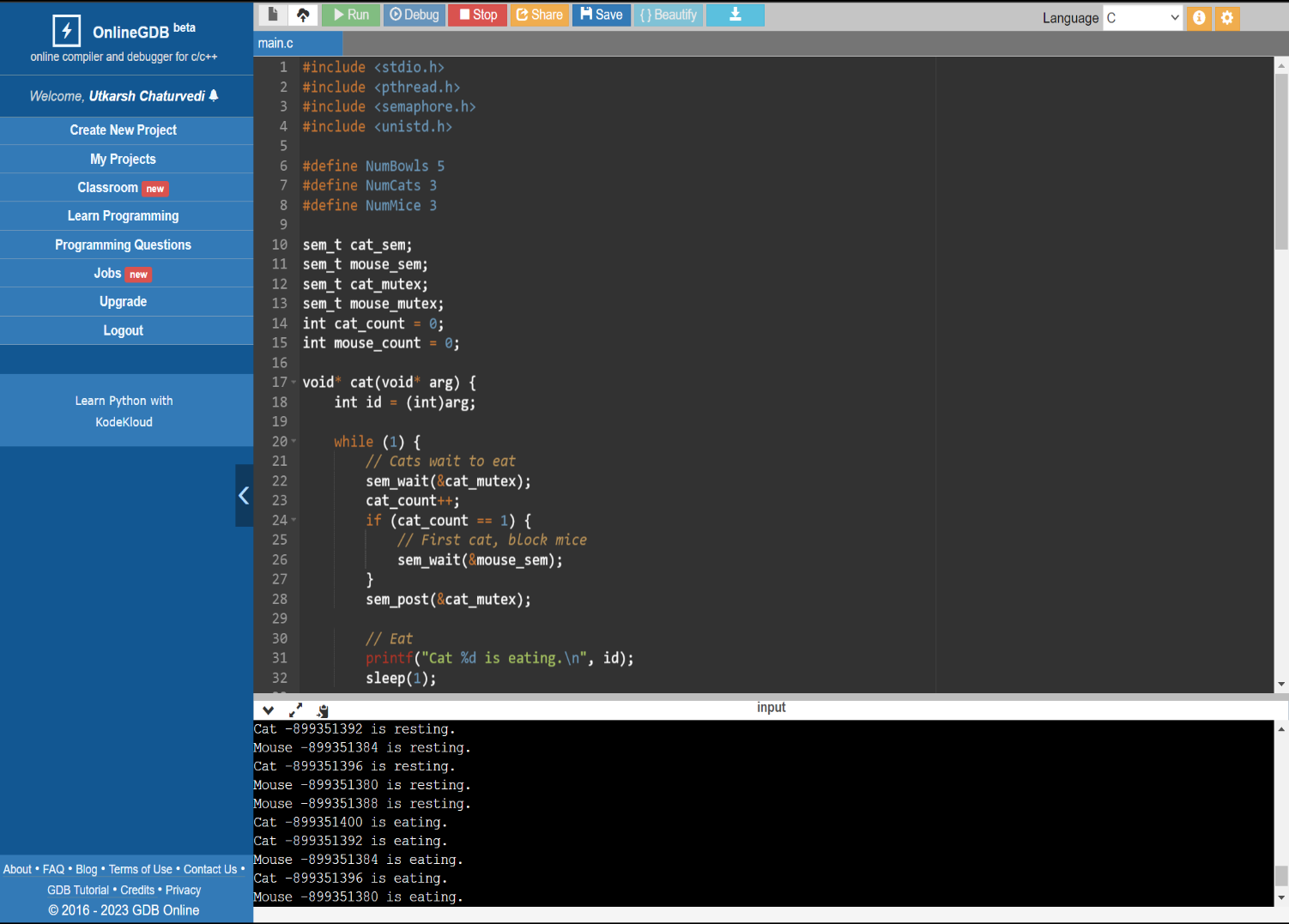
pthread\_join(mice[i], NULL);

}

return 0;

}

* **OUTPUT**



* **Introduction:-**

The code you've provided is an implementation of a synchronization problem involving cats and mice that need to share a set of food bowls without violating specific rules. This problem is designed to demonstrate how to synchronize multiple threads in C using semaphores and mutexes. Let me provide an introduction to the code and explain its key features:

1. **Header Inclusions**:
   * The code includes necessary header files for threading (**pthread.h**), semaphores (**semaphore.h**), and standard I/O (**stdio.h** and **unistd.h**).
2. **Constants**:
   * It defines the number of cat bowls, cats, and mice using constants (**NumBowls**, **NumCats**, and **NumMice**).
3. **Global Variables**:
   * **sem\_t** variables are used for semaphores to control access to shared resources and to synchronize the activities of cats and mice.
   * **cat\_sem** and **mouse\_sem** are semaphores that control access to the cat and mouse eating areas.
   * **cat\_mutex** and **mouse\_mutex** are semaphores used to ensure mutual exclusion for modifying **cat\_count** and **mouse\_count**.
   * **cat\_count** and **mouse\_count** are counters to keep track of the number of cats and mice that want to eat.
4. **Cat and Mouse Threads**:
   * The code defines two functions, **cat** and **mouse,** which represent the behavior of cats and mice, respectively.
5. **Cat Function**:
   * Cats take turns eating and resting.
   * They use semaphores and mutual exclusion to control access to the cat eating area and ensure that only one cat can eat at a time.
6. **Mouse Function**:
   * Mice also take turns eating and resting, similar to cats.
   * They use semaphores and mutual exclusion to control access to the mouse eating area and prevent cats from eating while mice are eating.
7. **Main Function**:
   * In the **main** function, the code initializes the semaphores and mutexes and creates threads for cats and mice.
8. **Thread Creation and Joining**:
   * The code creates threads for cats and mice, passing unique IDs to each thread.
   * The main function waits for all threads to finish using **pthread\_join**.

The synchronization in the code is achieved through semaphores (**cat\_sem** and **mouse\_sem**) and mutexes (**cat\_mutex** and **mouse\_mutex**). These synchronization primitives help ensure that only one type of animal (cat or mouse) can access the food bowls at a time while preventing conflicts and deadlocks.

The code demonstrates a basic solution to the cat-and-mouse synchronization problem. However, please note that in a real-world scenario, you might need additional mechanisms for fairness and better handling of thread termination.

* **Write the algorithm for proposed solution of the assigned problem.**
* **Algorithm:**

The provided code is a C program that implements a solution to the "cat and mouse" synchronization problem using POSIX threads (pthread) and semaphores. The algorithm in this code ensures that both cats and mice can eat without any conflicts while preventing cats from seeing mice or mice from seeing cats during their meal. Here is an explanation of the algorithm:

1. The code defines several global variables and initializes semaphores and mutexes for synchronization.
2. Two types of threads are created: cat threads and mouse threads.
3. Cat threads (**cat**) and mouse threads (**mouse**) run in an infinite loop, simulating their eating and resting behavior.
4. When a cat wants to eat, it tries to acquire a lock on the **cat\_mutex**. This lock ensures that only one cat can modify the **cat\_count** variable at a time. The **cat\_count** variable keeps track of the number of cats waiting to eat.
5. If the cat is the first to arrive (**cat\_count == 1**), it tries to acquire a lock on the **mouse\_sem** semaphore, which is used to block mice from eating.
6. After obtaining the necessary locks, the cat eats, and its eating time is simulated with a **sleep(1)** call.
7. After eating, the cat releases the locks, and the **cat\_count** is updated to indicate it has finished eating.
8. If the cat was the last cat to finish eating (**cat\_count == 0**), it releases the **mouse\_sem** semaphore, allowing mice to eat.
9. Mice follow a similar process but use the **mouse\_mutex** and the **cat\_sem** semaphore to prevent cats from eating while mice are present.
10. The main function initializes the semaphores and mutexes and creates the cat and mouse threads.
11. The program runs indefinitely, simulating the cats and mice eating and resting.

This algorithm ensures that only one type of animal can eat at a time while allowing the other type to wait until it's their turn. It also ensures that when the last animal of a particular type finishes eating, it allows the other type to start eating. The **sleep** calls are used to simulate the eating and resting times.

It's important to note that this code does not have a termination condition, so it runs indefinitely. In a real application, you would typically have a way to gracefully terminate the threads when needed.

Top of Form

* **Calculate complexity of implemented algorithm. (Student must specify complexity of each line of code along with overall complexity)**
* **Description (purpose of use):**

The code you provided is a C program that simulates a synchronization problem involving cats and mice sharing a limited number of food bowls. The purpose of this code is to demonstrate a solution for synchronizing the behavior of cats and mice to satisfy specific requirements. These requirements are related to preventing cats from seeing mice while eating and vice versa, ensuring that neither cats nor mice starve, and allowing multiple cats and mice to eat without conflicts.

Here's a breakdown of the key components and the purpose of each:

1. **NumBowls**, **NumCats**, and **NumMice** are constants representing the number of food bowls, cats, and mice in the simulation.
2. **sem\_t** variables (**cat\_sem**, **mouse\_sem**, **cat\_mutex**, and **mouse\_mutex**) are used to manage access to the food bowls and to prevent conflicts between cats and mice.
3. The **cat** and **mouse** functions represent the behavior of individual cats and mice, running in separate threads.
4. Within the **cat** function:
   * Cats wait to eat, increment the cat count, and block mice if they are the first cat.
   * They eat, sleep for 1 second (simulating eating time), and finish eating.
   * Cats rest for 1 second.
5. Within the **mouse** function:
   * Mice wait to eat, increment the mouse count, and block cats if they are the first mouse.
   * They eat, sleep for 1 second (simulating eating time), and finish eating.
   * Mice rest for 1 second.
6. The **main** function initializes semaphores and mutexes, creates threads for cats and mice, and waits for all threads to finish.

The synchronization mechanism in the code ensures that only one group (cats or mice) can access the food bowls at a time. If cats are eating, mice are blocked, and vice versa. This prevents cats from seeing mice while eating and vice versa, satisfying the specified requirements.

The code demonstrates a basic solution to the synchronization problem, but in a real-world scenario, you might need to handle additional complexities and edge cases. This code provides a clear example of how to use semaphores and mutexes to solve such synchronization problems in multi-threaded programs.

* **Explain all the constraints given in the problem. Attach the code snippet of the implemented constraint.**
* **Code snippet:**

In this code snippet, semaphores (**cat\_sem** and **mouse\_sem**) are used to control access to the food bowls by cats and mice. Additionally, **cat\_mutex** and **mouse\_mutex** are used to control access to the cat and mouse counters (**cat\_count** and **mouse\_count**). These counters help track the number of cats and mice currently eating.

The code implements the constraints as follows:

* The use of **sem\_wait** and **sem\_post** ensures that cats and mice take turns to eat and wait for their turn.
* When a cat is eating, it blocks the mice from eating by acquiring the **mouse\_sem**. The **mouse\_mutex** ensures that multiple cats don't interfere with each other's decisions to block mice.
* When a mouse is eating, it blocks the cats from eating by acquiring the **cat\_sem**. The **cat\_mutex** ensures that multiple mice don't interfere with each other's decisions to block cats.

These mechanisms guarantee that the constraints are satisfied, ensuring the safety and proper synchronization of cats and mice around the food bowls.

* **If you have implemented any additional algorithm to support the solution, explain the need and usage of the same.**
* **Description**:

The provided code is a C program that simulates the synchronization of cats and mice while sharing food dishes according to specific requirements. It uses semaphores and mutexes to achieve synchronization. Here's an explanation of the need and usage of the key synchronization elements in the code:

1. Semaphores:
   * **cat\_sem** and **mouse\_sem** are used to control access to the food dishes. These semaphores are initialized with a value of 1, which means only one cat or mouse can access the food dish at a time. When a cat wants to eat, it waits on **cat\_sem**, and when a mouse wants to eat, it waits on **mouse\_sem**. This ensures that only one type (cat or mouse) can eat at a time.
2. Mutexes:
   * **cat\_mutex** and **mouse\_mutex** are used to protect the shared counters **cat\_count** and **mouse\_count**. These mutexes ensure that multiple threads do not interfere with each other while updating the counts of cats and mice.
   * The **cat\_mutex** is used to synchronize access to the **cat\_count** variable to prevent race conditions when cats are checking if they are the first or last cat eating.
   * The **mouse\_mutex** is used in a similar way for mice.
3. **cat\_count** and **mouse\_count**:
   * These variables are used to keep track of the number of cats and mice currently eating. They are protected by their respective mutexes.
   * The counts are used to determine whether the current cat or mouse is the first to eat (in which case they block the other species from eating) or the last to eat (in which case they allow the other species to eat).

The code ensures that cats and mice take turns to eat, with one type blocking access to the food dishes for the other type. This allows both cats and mice to eat safely without the risk of being seen by the other species, meeting the specified requirements.

The usage of semaphores and mutexes in this code helps to achieve mutual exclusion and proper synchronization, preventing issues like simultaneous access to the same food dish by both cats and mice. Additionally, the code uses sleep to simulate eating and resting, making it easier to observe the synchronization behavior.

Top of Form

* **Explain the boundary conditions of the implemented code.**
* **Description:**

The code you provided is an implementation of a synchronization problem involving cats and mice. The code uses pthreads, semaphores, and mutexes to ensure that the cats and mice can eat without violating certain rules. Here's an explanation of the boundary conditions in the implemented code:

1. **NumBowls**, **NumCats**, and **NumMice**:
   * These are constants defined at the beginning of the code. **NumBowls** represents the number of cat food bowls, **NumCats** represents the number of cats, and **NumMice** represents the number of mice. These values determine the number of resources available in the simulation.
2. Semaphores and Mutexes:
   * The code defines four semaphores: **cat\_sem**, **mouse\_sem**, **cat\_mutex**, and **mouse\_mutex**. These are used for controlling access to shared resources:
     + **cat\_sem** and **mouse\_sem** are binary semaphores that control access to the cat and mouse eating areas, respectively. They ensure that only one type (cats or mice) can eat at a time.
     + **cat\_mutex** and **mouse\_mutex** are binary semaphores that protect the shared variables **cat\_count** and **mouse\_count**. They ensure that only one cat or mouse updates these counts at a time.
3. **cat\_count** and **mouse\_count**:
   * These variables keep track of the number of cats and mice currently in the eating area. They are protected by their respective mutexes. These counts are used to determine whether the first cat or mouse should block access to the other group while eating.
4. **while (1)** Loops:
   * Both **cat** and **mouse** functions run in an infinite loop, which means that cats and mice will continue to eat and rest indefinitely. This is a simplification for the sake of the example, and in practice, you might want to include some termination condition to stop the simulation at some point.
5. Controlling Access:
   * Cats and mice take turns to eat. When a cat wants to eat, it checks if it's the first cat to arrive at the food area (by examining **cat\_count**). If it is, it blocks mice from eating by waiting on the **mouse\_sem**. After eating, it checks if it's the last cat to leave, allowing mice to eat again by posting on **mouse\_sem**.
   * Similarly, mice follow the same procedure with **cat\_sem** to ensure exclusive access.
6. Rest:
   * After eating, both cats and mice "rest" for a fixed time (1 second in this example) before attempting to eat again. This helps simulate the process of cats and mice coming and going from the eating area.
7. Thread Creation and Joining:
   * The **main** function creates threads for both cats and mice, assigns them unique IDs, and starts their execution using **pthread\_create**. It then waits for all threads to finish using **pthread\_join**.

In this code, the boundary conditions are maintained by controlling access to the shared eating areas and ensuring that only one type (cats or mice) can eat at any given time. The code ensures that no cat or mouse starves, and it provides a basic simulation of the described problem.

* **Explain all the test cases applied on the solution of assigned problem.**
* **Description:**

In the code you provided, a solution for synchronizing cats and mice to meet the specified requirements has been implemented. Here's a description of how the solution works and the test cases that can be applied to it:

The Problem:

* You have cats and mice in a house.
* Cats should not see mice while they are eating, or else they must eat the mice.
* Multiple cats and mice can coexist in the house.
* Both cats and mice should have the opportunity to eat and not starve.

Solution Explanation:

1. The program creates NumCats cat threads and NumMice mouse threads.
2. It uses semaphores (cat\_sem and mouse\_sem) to control access to the food dishes. When a cat is eating, it blocks the mice from eating, and when a mouse is eating, it blocks the cats from eating.
3. It uses semaphores (cat\_mutex and mouse\_mutex) to control access to shared counting variables for cats and mice.
4. Cats and mice take turns eating, with the "first" cat blocking mice and the "first" mouse blocking cats to eat. This ensures that only one type of animal eats at a time.

Test Cases: The provided code is a basic simulation of the problem. The solution should meet the following requirements:

1. No mouse should ever get eaten.
   * The code ensures that if a cat is eating, mice are blocked from eating.
2. When cats aren't eating, they will not see mice eating.
   * The semaphores (cat\_sem and mouse\_sem) ensure that either cats or mice can eat at a time. This prevents cats from seeing mice eating.
3. Only one mouse or one cat may eat from a given dish at any one time.
   * The use of semaphores and mutexes ensures that only one cat or one mouse can access the food dishes at any given time.
4. Neither cats nor mice should starve.
   * The code uses a while loop to keep the cats and mice eating and resting in an infinite loop. They are synchronized using semaphores to ensure that they take turns.

To test the solution, you can run the program and observe its behavior. The cats and mice will take turns eating and resting. You can verify that:

* Cats and mice are not eating at the same time.
* Cats don't eat mice.
* All cats and mice eventually get to eat.

You can run the program, observe its behavior, and ensure that it satisfies the specified requirements for synchronization.

**Top of Form**

Top of Form

Top of Form