Using semaphores to synchronize baboons across a buffer rope

Youjian Tang, Travis Tran and Tommy Le

CSE Department

University of South Florida

{tang24, travist, tle568} @usf.edu

Abstract

For the topic of the final project, we are tasked to simulate baboons crossing a single rope that spans a canyon with certain conditions that need to be upheld. The objective of the project was to create a rope buffer that would allow any number of baboons from either the right or left side to cross, but only a maximum number of 3 baboons in sequence is allowed to travel through the rope at a time. Additionally, there should be no case in which the baboons should ever have to cross each other (meaning baboons going left and right should not be in the same buffer). In the end, the baboons entering and leaving the buffer should be in FIFO order. To resolve this scenario, we used multiple semaphores to synchronize the baboon order to account for the number of baboons inside buffer, to ensure only three baboons could enter and exit the buffer at a time, and only to consider both baboon directions once the buffer is empty. After running the code, we were able to successfully able to simulate the baboons crossing under all the mentioned conditions set. The code displays the original sequence of baboons and then simulates the baboons crossing in FIFO order from the input sequence, with only a maximum of 3 baboons crossing at a time, and with no baboons from either end crossing at the same time.

1 Introduction

There is an issue in which multiple threads attempt to access the same resources simultaneously. This situation poses a significant risk of data corruption. A common solution would be using semaphores to synchronize the threads so only one thread can access a critical section at a time. Semaphores can ensure this condition with their wait() and signal() functions. When a thread wishes to enter the critical section, it must wait until the semaphore returns a value of 1. Once inside, the thread has an opportunity to access needed resources and once completed will signal the semaphore, to increase by 1. A (semaphore = 1) enables other waiting thread to access the critical section. The complexity of the issue goes beyond simply ensuring that one baboon goes to the rope at a time. The ab imposes

specific requirements, necessitating the use of a variety of semaphores which they are designed to limit the number of baboons up to a capacity on the rope and to restrict access for opposing baboons when baboons are present, adding an additional requirement during synchronization process.

2 Methodology

Given the sequence of Baboon directions, a “for loop” would record the at most three baboons, each baboon must follow the same direction, otherwise it breaks. A batch count would increase for every “equal direction”. Having “RRR” produce a batch count of 3, while having “RRL” would produce a batch count of 2 stopping at ‘L’. The purpose of batch counter was for the 2nd “for loop” to allocate memory in the rope buffer and add values the directions into “threadArgs”.

Each baboon will share a common object, which is the rope. At the beginning of the program, the rope will be initialized with its maximum capacity (set to 3 in this program), the travel time, and all the required semaphores will be initialized properly. After all the baboons finish, the rope object will be destroyed.

With the “pthread_create()”, we would pass the shared rope object and the baboon’s direction from “threadArgs[]” array and the thread runner function *ROPE_cross(), to create new threads. *ROPE_cross() requires each Baboon thread to go through ROPE_announce, ROPE_mount, and ROPE_dismount functions.

ROPE_anounce() uses a semaphore (announce = 1) to declare a direction and ensure only one baboon thread is entering a buffer at a time. At the end of the function, the “announce” semaphore should signal() to enable the next baboon to enter the rope buffer. Within ROPE_announce, a direction is declared waiting and signaling the given directional semaphores (sem_left or sem_right) and it will increase the baboon count for the given direction. When there is at least one baboon, the opposing direction semaphore will use wait(). The opposite direction baboons will not be able to enter the buffer because its semaphore would’ve been 0 and will have to wait until there is a signal from ROPE_dismount().

ROPE_mount() used a counting semaphore to ensure only a limited number of baboons (3) could enter the rope. Wait () would decrease the semaphore and would prevent new baboons from entering once the counting semaphore reaches 0. ROPE_dismount() used given direction semaphore wait and signal to decrease the number of baboons in the buffer. If there are no baboons in the buffer, the opposing direction semaphore will signal () allowing the possibility of left or right baboons to enter the rope buffer.

The main object of creating baboon threads with *ROPE_cross() was to create additional wait time when inserting or removing a baboon from buffer. The wait time is needed so the semaphore conditions are met. *ROPE_cross() creates a pause whenever it prints out the baboons entering the rope. This pause ensures at most three baboons or less will be in rope buffer and all baboons move in same direction.

After pthread_create() allocates a set of baboons from the rope, another “for loop” of the same size uses pthread_join() to wait for all the selected thread to finish executing and deallocates them. Once finished, a new batch of baboons is selected from the sequence and inserted into pthread_create() and *ROPE_cross to be allocated in the rope in a certain order. This cycle of selecting a batch of baboons to be placed in rope continues until the entire sequence is used.

3 Results

We used 2 input cases: “baboon_directions.txt, baboon_directions2.txt. Because we selected baboons in sequential order, the output should be the same as the input as the first baboons to enter, should also be the first to leave (FIFO).

```
[tle568@sclogin2 final_project]$ gcc final_project_v4.c -lpthread -lrt -std=c99
[tle568@sclogin2 final_project]$ ./a.out baboon_directions.txt 1
PTHREAD_SCOPE_SYSTEM


Start of Simulation

Original Sequence:
L L R R R R L L R R

Baboons are crossing:
L L R R R R L L R R

End of Termination

[tle568@sclogin2 final_project]$ ./a.out baboon_directions2.txt 1
PTHREAD_SCOPE_SYSTEM


Start of Simulation

Original Sequence:
L L L L L L L L L L L L L R R L L L L L R R R R R R L R L R L

Baboons are crossing:
L L L L L L L L L L L L L R R L L L L L R R R R R R L R L R L

End of Termination
```

Because each of the test cases was able to print exactly like input, the code was proven to be correct.

The project requirements were to ensure only 3 baboons max would be on the rope and all baboons on the rope must be moving in the same direction. To ensure these conditions, the code was required to print the sequence of baboons currently inside the buffer. Though the objective was to print the entire sequence in FIFO order, the order would be printed in sections and each section shows how many baboons were present and the directions of the baboons. Given the sections prints, if each sections showed the same direction and there were never more than 3

baboons in each section, and it was able to meet the first two conditions for the entire FIFO sequence, then the program was able to meet all conditions successfully.

4 Conclusion

To synchronize the baboons traveling though the rope, such that no more than three baboons were present, and no baboons should be traveling in the opposite direction, we use 4 semaphores: 1 to act as mutex, 2 to identify left or right, and 1 to count the # of baboons on the rope. The program loops through a sequence to collect a batch of baboons (3 or less). We were able to synchronize the baboons, by creating threads through “Rope_Cross” functions. Making threads go through semaphores to meet the project conditions. The results were successful as the output was able to print in exact order as the initial sequence, displaying FIFO. Additionally, each section (baboons in rope) printed while meeting project conditions