Baboon Canyon Crossing: A Synchronization Problem

Authors: Collin Gros, Preeti Maurya, Bryan Bustillos

ABSTRACT

Shared memory and synchronization are important concepts in multi-threaded programming. In our problem, we were required to use these concepts to enable baboons to cross the canyon, given the restrictions on the rope and sequence of the baboons. Semaphores control shared memory access, allowing only one baboon on each side to write data and determine if it is safe to cross. Shared memory allows the threads themselves to check whether to cross or wait.

In our program, there are four functions: main, init, left, and right. init() handles command-line arguments and reads the given file for baboon assignments. left() acts as the baboons on the left side of the canyon, while right() acts as the baboons on the right side of the canyon. main() handles the initialization of semaphores, command-line argument variables, process creation, process deletion, and cleanup.

The activity of the baboons is described through print statements to stdout, and show the requests, travel time, and status of the baboons (along with their assigned IDs and status).

From running our code, we were able to read files that contained a variety of baboon assortments, specifically ones that would prove challenging if not handled properly. Our highlighted case, for example, contains a pair of a sequence of baboons and a travel time that showcase all three requirements of the rope that stretches the canyon. Our code can handle these difficult arguments and displays the sequence of baboons along with their status in a way that does not cause deadlock and adheres to the three requirements of our problem.

INTRODUCTION

Creating threads in a program is an extremely important thing to do, as they allow you to work on multiple computations at a time. Sharing data between these threads is sometimes vital, depending on the problem at hand. Data can be shared across threads through the data and heap segments of memory. Though these data are easily shared across threads, problems arise when trying to write data, instead of just reading it.

The following defines the producer/consumer problem, an excellent example of the important of shared memory:

> "A producer process produces information that is consumed by a consumer process … One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. **The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.** " (Silberschatz et al., 2012).

The textbook stresses the importance of using a method of synchronization to allow the consumer and producer processes to work together.

We decided to solve **Problem 1,** defined in the project description. This problem directly relates to the consumer/producer problem; we require memory to be shared between threads to synchronize baboon movement across a canyon. In our program, we use semaphores (as well as shared integer variables located in the data segment) to control shared memory access, as well as prevent deadlocks.

In **Problem 1**, baboons from both sides of a canyon have access to a rope. All baboons want to cross to the other side of the canyon. There is a rope that stretches across the canyon that is accessible from both sides. However, the following restrictions take place:

1. Once a baboon has begun to cross, it is guaranteed to get to the other side without running into a baboon going the other way.
2. There are never more than three baboons on the rope. The order of the baboons crossing the rope should be preserved; i.e., the order in which they enter the rope should be the order in which they exit the rope.
3. A continuing stream of baboons crossing in one direction should not bar the baboons going the other way indefinitely (no starvation).

Our solution effectively accomplishes this task, as it uses shared memory to synchronize the baboon threads, and semaphores to protect that shared memory.

METHODOLGY

Input:

1)Text file containing the L, R, R, R, R, R, L,L,R (a sequence of alphabets L (meaning left) and R (meaning right) separated by a comma, that indicate the side of the rope a baboon is trying to cross the rope from)

2)The time (in seconds) required for a baboon to cross the canyon

Strategy:

1)Avoid Deadlock using semaphores

2)Keeping counter to limit number of Baboons that can travel at a time.

3)Avoid Starvation

We'd need 5 types semaphore variables,

```
sem_init(&rope, 0, 1); /* rope is used once at a time */

/* left and right honor mutual exclusion */

sem_init(&left_mutex, 0, 1);

sem_init(&right_mutex, 0, 1);

sem_init(&baboons_counter, 0, 3); /* 3 baboons at a time */
```

Baboon crossing from Left to Right **Baboon crossing from Right to Left**

`/* thread is created: a request is made. */printf("b%d:\tl->r?\n", *id);`	`/* thread is created: a request is made. */ printf("b%d:\tr->l?\n", *id);`

```
/* the first baboon on this side will activate this
semaphore. */
sem_wait(&left_mutex);
/* if we are the first baboon to go on our side, we are
responsible
for checking if the rope is available. once it is
available */
if (l_wait_count == 0) {
sem_wait(&rope);
}
l_wait_count++;
next_id++;
sem_post(&left_mutex);
/* after 3 are crossing in one direction,
we can't let any more on the rope */
sem_wait(&baboons_counter);
int num_baboons;
sem_getvalue(&baboons_counter, &num_baboons);
/* print that we are now crossing */
printf("b%d:\tl->r!\n", *id);
printf("b%d:\t%d seconds until crossed over...\n",
*id, travel_time);
printf("b%d:\tbaboons on rope: %d\n", *id, 3-
num_baboons);
sleep(travel_time);
printf("b%d:\t/\n", *id);
sem_post(&baboons_counter);
sem_wait(&left_mutex);
l_wait_count--;
/* only when all the baboons are done crossing we
can post the
rope */
if (l_wait_count == 0) {
sem_post(&rope);
}
sem_post(&left_mutex);
```

```
/* the first baboon on this side will activate this
semaphore. */
sem_wait(&right_mutex);
/* if we are the first baboon to go on our side, we are
responsible
for checking if the rope is available. once it is
available */
if (r_wait_count == 0) {
sem_wait(&rope);
}
r_wait_count++;
next_id++;
sem_post(&right_mutex);
/* after 3 are crossing in one direction,
we can't let any more on the rope */
sem_wait(&baboons_counter);
int num_baboons;
sem_getvalue(&baboons_counter, &num_baboons); .
/* print that we are now crossing */
printf("b%d:\tr->l!\n", *id);
printf("b%d:\t%d seconds until crossed over...\n",
*id, travel_time);
printf("b%d:\tbaboons on rope: %d\n", *id, 3-
num_baboons);
sleep(travel_time);
printf("b%d:\t/\n", *id);
sem_post(&baboons_counter);
sem_wait(&right_mutex);
r_wait_count--;
/* only when all the baboons are done crossing we
can post the rope */
if (r_wait_count == 0) {
sem_post(&rope);
}
sem_post(&right_mutex);
```

RESULTS

A note on the formatting of print statements (taken from main.c):

```
print statements:
    b0: l->r? // baboon #0 requests to cross from the left side
    b0: l->r! // baboon #0 has begun crossing
    b0: 4 seconds until crossed over... // b0's ETA
    b0: baboons on rope: 3 // number of baboons crossing from
                           // this side as well
    b0: / // baboon 0 has made it to the other side.
```

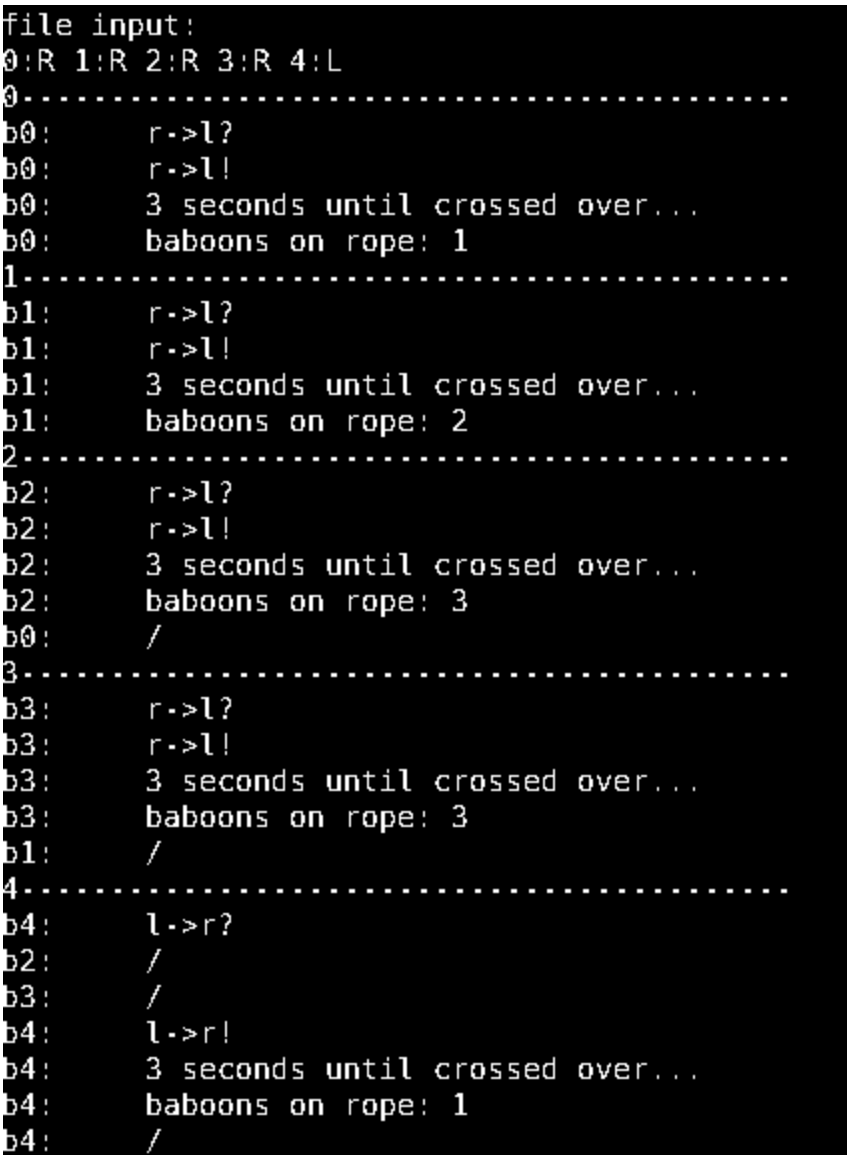
Test Cases

./run 1.txt 3

Output:

```
file input:
0:R 1:L
0----------------------------------------
b0:     r->l?
b0:     r->l!
b0:     3 seconds until crossed over...
b0:     baboons on rope: 1
1----------------------------------------
b1:     l->r?
b0:     /
b1:     l->r!
b1:     3 seconds until crossed over...
b1:     baboons on rope: 1
b1:     /
```

./run 2.txt 3


file input:
0:R 1:R 2:R 3:R 4:L
0----------
b0: r->l?
b0: r->l!
b0: 3 seconds until crossed over...
b0: baboons on rope: 1
1----------
b1: r->l?
b1: r->l!
b1: 3 seconds until crossed over...
b1: baboons on rope: 2
2----------
b2: r->l?
b2: r->l!
b2: 3 seconds until crossed over...
b2: baboons on rope: 3
b0: /
3----------
b3: r->l?
b3: r->l!
b3: 3 seconds until crossed over...
b3: baboons on rope: 3
b1: /
4----------
b4: l->r?
b2: /
b3: /
b4: l->r!
b4: 3 seconds until crossed over...
b4: baboons on rope: 1
b4: /

./run 3.txt 3

```
file input:
0:R 1:R 2:R 3:L 4:R 5:R 6:R 7:L
0----------------------------------------
b0:     r->l?
b0:     r->l!
b0:     3 seconds until crossed over...
b0:     baboons on rope: 1
1----------------------------------------
b1:     r->l?
b1:     r->l!
b1:     3 seconds until crossed over...
b1:     baboons on rope: 2
2----------------------------------------
b2:     r->l?
b2:     r->l!
b2:     3 seconds until crossed over...
b2:     baboons on rope: 3
b0:     /
3----------------------------------------
b3:     l->r?
b1:     /
4----------------------------------------
b4:     r->l?
b2:     /
b3:     l->r!
b3:     3 seconds until crossed over...
b3:     baboons on rope: 1
5----------------------------------------
b5:     r->l?
6----------------------------------------
b6:     r->l?
7----------------------------------------
b7:     l->r?
b3:     /
b4:     r->l!
b4:     3 seconds until crossed over...
b4:     baboons on rope: 1
b5:     r->l!
b5:     3 seconds until crossed over...
b5:     baboons on rope: 2
b6:     r->l!
b6:     3 seconds until crossed over...
b6:     baboons on rope: 3
b4:     /
b5:     /
b6:     /
b7:     l->r!
b7:     3 seconds until crossed over...
b7:     baboons on rope: 1
b7:     /
```

This first test case proves that this code accomplishes requirement #1.

The second test case proves that this code accomplishes requirement #2.

The third test case proves that this code accomplishes requirement #3

CONCLUSION

Many issues can arise from threads making use of and managing the same resources. There can be memory consistency issues and thread interference errors due to race conditions. In our problem, getting baboons to cross a canyon in only one direction at a time, we face these problems. More precisely, we want to avoid a deadlock where two baboons become stuck as they attempt to cross from both directions and starvation where one process waits for another. The main solution to these problems, as we've learned, is synchronization between the threads responsible for crossing baboons to the left or right of the canyon. Synchronization is accomplished by implementing semaphores and critical sections where one thread can perform its task without being interrupted by having the shared resource used by another thread. Additionally, there is a limit on the number of baboons, and thus we can check when there are no more baboons and end the program appropriately without needlessly running processes.

WORKS CITED

Silberschatz, A., Galvin, P. B., & Gagne, G. (2012). Operating System Concepts (9th ed.). Wiley.