Baboon Crossing: Synchronized versus linearized approach

CS474: Operating Systems

Marco Salazar†
Computer Science Department
New Mexico State University
Las Cruces, NM USA
marcoams@nmsu.edu

Catalina Sánchez-Maes†
Computer Science Department
New Mexico State University
Las Cruces, NM USA
csanchezmaes@gmail.com

ABSTRACT

This paper attempts to implement and contrast a synchronization solution and a linear solution for the Baboon Canyon Crossing problem to determine fastest runtime. A concurrent solution based upon the ideas behind Dekker's, Peterson's, Lamport's algorithms provided a 53% speedup of the Baboon Crossing problem. Concurrent solutions are faster than linearized solutions. Some operating systems lack support for required tools when a generalized solution is met.

1 Introduction

Synchronization methodologies from Dekker's, Peterson's, and Lamport's algorithms are utilized to maintain deadlock and race condition free concurrency. The solution for the Baboon Crossing Problem preserves First-In First-Out (FIFO) through the use of semaphores and mutual exclusion concurrency principles such that the baboons are all able to safely cross the canyon in their respective directions. The semaphore implementation is varied depending on which operating system it is compiled within. A concurrent solution has a faster runtime than a linear solution for the Baboon Crossing problem due to properly implemented synchronization.

2 Problem Statement

There is a deep canyon somewhere in Kruger National Park, South Africa, and a single rope that spans the canyon. Baboons can cross the canyon by swinging hand- over-hand on the rope, but if two baboons going in opposite directions meet in the middle, they will fight and drop to their deaths. Furthermore, the rope is only strong enough to hold three baboons. If there are more baboons on the rope at the same time, it will break. Assuming that we can teach the baboons to use semaphores, we would like to design a synchronization scheme with the following properties.

Implementation:

- The solution must guarantee that once a baboon begins to cross that it reaches the other side without meeting another baboon.
- There should only be 3 Baboons on the rope at a time, and the order of them should be preserved, such that it is a first in first out queue.
- The solution should never permit them to be starvation for either side, such that there is a continuous stream of baboons going one direction and not starving the other baboons on the other side that want to travel.
- The solution shall assume that all Baboons take the same amount of time to cross the rope.
- This shall be implemented in C with the Pthreads library.
- The rope will be represented with a Critical Section, and each baboon will be represented with a thread that sleeps as long as it takes the baboon to cross the bridge.
- The input to this program will be the time that each baboon takes to cross, and a text file that contains the order of arrival for each monkey in the format "L,R,R,L,etc".

3 Methodology

Concurrency allows for a faster implementation but also comes with risks that a linear approach doesn't contain such as starvation, deadlock, and race conditions. The goal is to provide deadlock/starvation avoidance to allow for a continual movement between all Baboons to cross in both directions safely without sacrificing the speed. While Dekker's algorithm is the historically first software solution to mutual exclusion problem it is limited by its maximum of two processes, whereas highly popularized Peterson's algorithm is known for its elegance and compactness [2]. Both algorithms inherently mitigate the aforementioned concurrency risks/considerations.

Dekker and Peterson's algorithms implement the property of mutual exclusion. As defined in Peterson's paper, "Mutual exclusion means that both processes can never be in their critical sections at the same time."[1]
In the baboon solution, the principle is implemented in relation directionally and the shared resource (Rope buffer). In both of these algorithms only one process can enter the critical section at a time. Our solution allows for a maximum of three processes (baboons) to enter the critical section at a time.

Lamport's algorithm asserts that at most two processes can enter the critical section under the right circumstances and FIFO high priority processors are served first [3].
FIFO is preserved in the directional sense such that when moving baboons to the right, the left will wait. Of the crossing group, FIFO should be guaranteed because the first baboon may be suspended due to timeout to allow one of the other two baboons to run faster.

Our code guarantees that it is individual order is preserved due to a shared resource checking against this. This was implemented through an integer array 0-N indicating the order of the baboons to prevent passing. The output of the program will print out the

order that the monkeys leave in the form of L/R#: (#, #, #) Where L or R indicate whether the monkey is going left to right, or right to left respectively. Where the first number indicates which monkey it is, and the array of numbers indicates what monkeys were on the rope the moment before it left. If working correctly, the first number should be equal to the lowest nonzero number in the array next to it. Also, the array position of the numbers in the array do not indicate the position of the baboons on the rope, only the values do.
The input to this program will be the time that each baboon takes to cross, and a text file that contains the order of arrival for each monkey in the format "L,R,R,L, etc" as seen on page 5 in input_file.txt.

The concurrency solution, beginning on page 3, is implemented in C with the Pthreads library imported on line 35. The linear solution, provided on page 5, is also implemented in C. An input parameter that indicates the time required for a baboon to cross the canyon, assuming all the baboons require the same time to cross the canyon.

The creation of the shared rope buffer is contained lines 55 – 62.

The direction of the baboons is represented in threads with each semaphore controlling the number of baboons on the rope currently within the direction groups (left to right, right to left) on lines 65 and 66.
The left to right function begins on line 176 ending on line 266. This function determines the max value on the rope, and the position of the next available spot in the shared array to keep track of the baboons.

Lines 190-203 implement as follows. A Baboons ID becomes one greater than the last baboon and puts it on the rope, otherwise, if there is no current baboons on the rope it puts 1. The just placed baboon crosses the rope. Maintaining values for the other two positions holds the second and third values of the baboons, however if these values are 0 there is one less baboon.

On lines 205-266, loop to determine if the current baboon can exit, FIFO order with cases of zero, one, two, and three baboons on the rope. If there are multiple monkeys on the rope, we check the order to ensure FIFO proper exit.
The right to left function is the exact same as above except that it prints out an R.

4 Results

Concurrency Solution:

Jsers > catalinasanchez-maes > Downloads > project3_2nddraft > C project3.c > ...

```c
/*
Name: Marco Salazar and Catalina Sanchez-Maes
Date: 11/20/2020
Username for canvas: marcoams
[redacted]: catsmaes
Username for CS lab: msalazar
[redacted]: cmaes

Description:
This program will print out the order that the monkeys leave in the form of L/R#: (#, #, #)
    Where L or R indicate whether the monkey is going left to right, or right to left respectively.
    Where the first number indicates which monkey it is, and the array of numbers indicates
    what monkeys were on the rope the moment before it left. If working correctly, the first
    number should be equal to the lowest nonzero number in the array next to it.
Note: The array position of the numbers in the array do not indicate the postion of the baboons on the rope.
Only the values do.

Purpose: To Simulate the Monkey Baboon problem, making sure
1. We must guarantee that once a baboon begins to cross that it reaches the other side
without meeting another baboon.
2. There should only be 3 Baboons on the rope at a time, and the order of them should be
preserved, such that it is a first in first out queue.
3. We should never permit them to be starvation for either side, such that there is a
continuous stream of baboons going one direction and not starving the other baboons on
the other side that want to travel.
4. We shall assume that all Baboons take the same amount of time to cross the rope.
5. This shall be implemented in C with the Pthreads library.
6. The rope will be represented with a Critical Section, and each baboon will be represented
with a thread that sleeps as long as it takes the baboon to cross the bridge.
7. The input to this program will be the time that each baboon takes to cross, and a text file
that contains the order of arrival for each monkey in the format "L,R,R,L,etc".
*/

#define _REENTRANT
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <semaphore.h>

/* key number */
#define SHMKEY ((key_t) 1497)

// only one thread per critical section.
sem_t left2right;
sem_t right2left;
sem_t mutex;

// Struct to hold the rope buffer
typedef struct
{
  char array[3];
} shared_rope;


shared_rope *rope;

// function protottypes for the threads.
void *leftbaboon(void * arg);
void *rightbaboon(void * arg);

int main(int argc, char *argv[]){
  if(argc < 2){
    printf("Error, file not given.");
    exit(1);
  } else if(argc < 3){
    printf("Error, time for monkeys to cross not given.");
    exit(1);
  }
  //explanation statement
  printf("This program will print out the order that the monkeys leave in the form of L/R#: (#, #, #)

  // mutex to protect the shared rope
  sem_init(&mutex, 0, 1);

  int shmid, pid1, pid2, ID, status;
  char *shmadd;
  shmadd = (char *) 0;

  // Creates and connects to a shared memory segment
  if ((shmid = shmget(SHMKEY, sizeof(int), IPC_CREAT | 0666)) < 0)
    {
      perror ("shmget");
        exit (1);
    }
  if ((rope = (shared_rope *) shmat(shmid, shmadd, 0)) == (shared_rope *) -1) {
        perror ("shmat");
        exit (0);
    }

  // Initialize the shared memory to 0 in all places which indicates no baboons are on the rope
  for(int val = 0; val < 3; val++){
    rope->array[val] = 0;
  }

  // each semaphore controls how many monkeys are traveling on the rope currently in each group
  // eg. left to right group, and right to left group.
  sem_init(&left2right, 0, 3);
  sem_init(&right2left, 0, 3);

  pthread_t tid[1]; //process id for every baboon, we
  pthread_attr_t attr[1]; // attribute pointer array

  fflush(stdout);
  /* Required to schedule thread independently.*/
  pthread_attr_init(&attr[0]);
  pthread_attr_setscope(&attr[0], PTHREAD_SCOPE_SYSTEM);
  /* end to schedule thread independently */

  // Variable to check how many monkeys of the other side are on the rope.
  // If onRope is 3, then each group would be able to put a monkey respectively.
  // that way if no one is on the rope a group would get 3, which is the go ahead to put a monkey.
  // however, if it is any number less than three then it must wait until it is three since that means
  // there are monkeys going the opposite direction.
  int onRope = 0;

  FILE* fp;
  fp = fopen(argv[1], "r");
  char nextChar;
  // while there is characters to read
  while(fscanf(fp,"%c",&nextChar) != EOF){
    // , are not important
    if(nextChar == ',') continue;
    if(nextChar == 'R'){
      // Loop until there are no monkeys going from left to right.
      while(1){
        sem_getvalue(&left2right, &onRope);
        if(onRope == 3) break;
      }

      // start a monkey going from right to left
      sem_wait(&right2left);
      pthread_create(&tid[0], &attr[0], rightbaboon, argv[2]);
    } else {
      // Loop until there are no monkeys going from right to left.
      while(1){
        sem_getvalue(&right2left, &onRope);
        if(onRope == 3) break;
      }

      // start a monkey going from left to right
      sem_wait(&left2right);
      pthread_create(&tid[0], &attr[0], leftbaboon, argv[2]);
    }
  }

  // Wait for the last baboon to cross.
  pthread_join(tid[0], NULL);
  printf("\n");

  // Detaching the Shared Memory.
  if(shmdt(rope) == -1){
    perror("shmdt");
    exit(-1);
```

```c
    }
    shmctl(shmid, IPC_RMID, NULL);

    // Destroy the semaphores
    sem_destroy(&left2right);
    sem_destroy(&right2left);
    sem_destroy(&mutex);


    // terminate threads
    pthread_exit(NULL);

    return 0;
}

// function for a baboon going from left to right
void *leftbaboon(void * arg){
    // Put the baboon on the rope.
    sem_wait(&mutex);
    int max = 0;
    int available = 0;
    int val = 0;
    // determine the max value on the rope, and the position of the next
    // available spot in the shared array to keep track of the baboons.
    for(int i = 0; i < 3; i++){
        val = rope->array[i];
        if(val > max) max = val;
        if(val == 0) available = i;
    }
    // Baboons ID becomes one greater than the last baboon.
    max = max + 1;
    //puts the baboon on the rope with one number bigger than the last baboon that is still on the rope.
    //otherwise it puts 1
    rope->array[available] = max;
    sem_post(&mutex);

    //cross the rope.
    sleep(atoi((char *) arg));

    // values to hold the second and third values of the baboons.
    // if they are 0 there is one less baboon.
    int secondvalue = -1;
    int thirdvalue = -1;

    // Loop to find out if this baboon can exit, FIFO order.
    while(1){
        secondvalue = -1;
        thirdvalue = -1;
        sem_wait(&mutex);

        // Get the first and second value for the other baboons.
        for(int i = 0; i < 3; i++){
            if(i == available) continue;
            if(secondvalue != -1){
                thirdvalue = rope->array[i];
                break;
            }
            secondvalue = rope->array[i];
        }

        // If there are no other baboons, then this baboon can exit
        if(secondvalue == 0 && thirdvalue == 0){
            rope->array[available] = 0;
            printf("L%d:(%d %d %d)\n", max, max, secondvalue, thirdvalue);
            sem_post(&mutex);
            break;
        }

        if(secondvalue == 0){
            // If this baboon was on before the other baboon, he can exit.
            if(thirdvalue > max){
                rope->array[available] = 0;
                printf("L%d:(%d %d %d)\n", max, max, secondvalue, thirdvalue);
                sem_post(&mutex);
                break;
            }
            // otherwise he must wait.
            sem_post(&mutex);
            continue;
        }
        if(thirdvalue == 0){
            // If this baboon was on before the other baboon, he can exit.
            if(secondvalue > max){
                rope->array[available] = 0;
                printf("L%d:(%d %d %d)\n", max, max, secondvalue, thirdvalue);
                sem_post(&mutex);
                break;
            }
            // otherwise he must wait.
            sem_post(&mutex);
            continue;
        }

        // If this baboon was on before all the other baboons, he can now exit.
        if(max < secondvalue && max < thirdvalue){
            rope->array[available] = 0;
            printf("L%d:(%d %d %d)\n", max, max, secondvalue, thirdvalue);
            sem_post(&mutex);
            break;
        }
        sem_post(&mutex);
    }

    fflush(stdout);
    sem_post(&left2right);
}

// function for a baboon going from right to left
void *rightbaboon(void * arg){
    // Put the baboon on the rope.
    sem_wait(&mutex);
    int max = 0;
    int available = 0;
    int val = 0;

    // determine the max value on the rope, and the position of the next
    // available spot in the shared array to keep track of the baboons.
    for(int i = 0; i < 3; i++){
        val = rope->array[i];
        if(val > max) max = val;
        if(val == 0) available = i;
    }
    // Baboons ID becomes one greater than the last baboon.
    max = max + 1;
    //puts the baboon on the rope with one number bigger than the last baboon that is still on the rope.
    //otherwise it puts 1
    rope->array[available] = max;
    sem_post(&mutex);

    //cross the rope.
    sleep(atoi((char *) arg));

    // values to hold the second and third values of the baboons.
    // if they are 0 there is one less baboon.
    int secondvalue = -1;
    int thirdvalue = -1;

    // Loop to find out if this baboon can exit, FIFO order.
    while(1){
        secondvalue = -1;
        thirdvalue = -1;
        sem_wait(&mutex);

        // Get the first and second value for the other baboons.
        for(int i = 0; i < 3; i++){
            if(i == available) continue;
            if(secondvalue != -1){
                thirdvalue = rope->array[i];
                break;
            }
            secondvalue = rope->array[i];
        }

        // If there are no other baboons, then this baboon can exit
        if(secondvalue == 0 && thirdvalue == 0){
            rope->array[available] = 0;
            printf("R%d:(%d %d %d)\n", max, max, secondvalue, thirdvalue);
            sem_post(&mutex);
            break;
        }

        if(secondvalue == 0){
            // If this baboon was on before the other baboon, he can exit.
            if(thirdvalue > max){
                rope->array[available] = 0;
                printf("R%d:(%d %d %d)\n", max, max, secondvalue, thirdvalue);
                sem_post(&mutex);
                break;
            }
            // otherwise he must wait.
            sem_post(&mutex);
            continue;
        }
        if(thirdvalue == 0){
            // If this baboon was on before the other baboon, he can exit.
            if(secondvalue > max){
                rope->array[available] = 0;
                printf("R%d:(%d %d %d)\n", max, max, secondvalue, thirdvalue);
                sem_post(&mutex);
                break;
            }
            // otherwise he must wait.
            sem_post(&mutex);
            continue;
        }

        // If this baboon was on before all the other baboons, he can now exit.
        if(max < secondvalue && max < thirdvalue){
            rope->array[available] = 0;
            printf("R%d:(%d %d %d)\n", max, max, secondvalue, thirdvalue);
            sem_post(&mutex);
            break;
        }
        sem_post(&mutex);
    }
    fflush(stdout);
    sem_post(&right2left);

```

Linear Solution:

Users > catalinasanchez-maes > Downloads > project3_first > C linear.c > ...

```c
/*
Name: Marco Salazar and Catalina Sanchez-Maes
Date: 11/20/2020
Username for canvas: marcoams
[redacted]: catsmaes
Username for CS lab: msalazar
[redacted]: cmaes

Purpose: To see how long the linear solution takes, when only one baboon goes acr
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){
  if(argc < 2){
    printf("Error, file not given.");
    exit(1);
  } else if(argc < 3){
    printf("Error, time for monkeys to cross not given.");
    exit(1);
  }

  FILE* fp;
  fp = fopen(argv[1], "r");
  char nextChar;
  // while there is characters to read
  while(fscanf(fp,"%c",&nextChar) != EOF){
    // , are not important
    if(nextChar == ',') continue;
    sleep(atoi(argv[2]));
    printf("%c ", nextChar);
    fflush(stdout);
  }

  printf("\n");
  return 0;
}
```

Input file: bab.txt
L,R,R,L,L,L,R,R,R,R,L,L,L,L,L

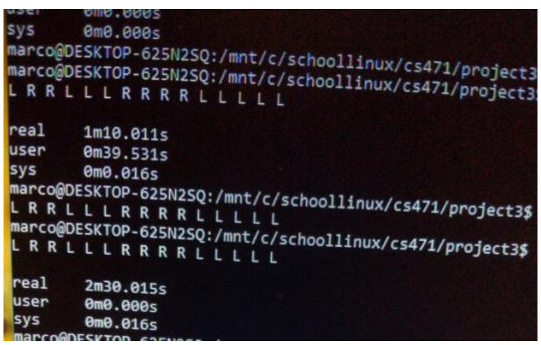

Fig. 1 Windows machine compilation "(shortened output version)"

```
marco@DESKTOP-625N2SQ:/mnt/c/schoollinux/cs471/project3$ time ./project3 bab.txt 10
This program will print out the order that the monkeys leave in the form of L#: (#, #, #)
Where the first number indicates which monkey it is, and the array of numbers indicates
What monkeys where on the rope the moment before it left. If working correctly, the first
number should be equal to the lowest nonzero number in the array next to it.
Note: The array position of the numbers in the array do not indicate the postion of the baboons on the rope. Only the va
lues do.

L1:(1 0 0)
R1:(1 0 2)
R2:(2 0 0)
L1:(1 3 2)
L2:(2 3 0)
L3:(3 0 0)
R1:(1 3 2)
R2:(2 3 0)
R3:(3 0 0)
R1:(1 0 0)
L1:(1 3 2)
L2:(2 3 0)
L3:(3 0 0)
L1:(1 0 2)
L2:(2 0 0)

real    1m10.016s
user    0m39.500s
sys     0m0.000s
```

Fig. 2 Final concurrency runtime and FIFO Proof

The finished program shown in Figure 2 utilizing principles from Dekker, Peterson, and Lamport's algorithms is 1 minute and 10 seconds which is ~53% faster than the linear algorithm with an average of 2 mins and 30 seconds. Upon early compilation while still creating the code, there was differing outputs visually for the same code.

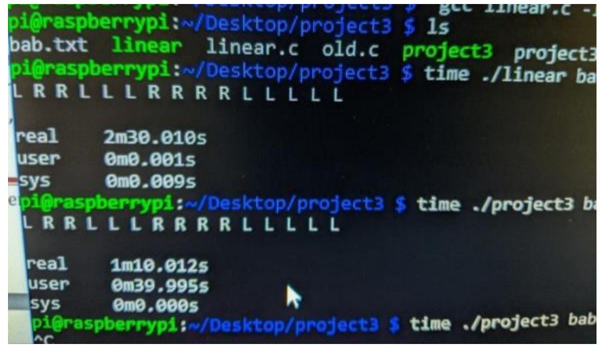

Fig. 3 Raspberry Pi machine compilation "(shortened output version)"

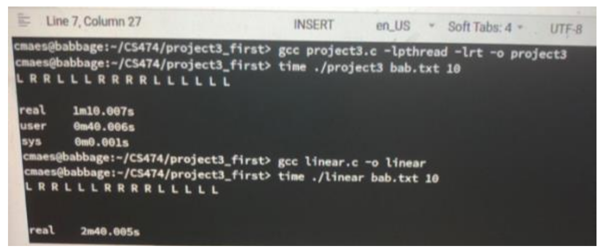

Fig. 4 Open Suze Linux SSH machine compilation from MacBook "(shortened output version)"

```
#define __deprecated     __attribute__((deprecated))
                                   ^
project3.c:95:9: warning: 'sem_getvalue' is deprecated
      [-Wdeprecated-declarations]
        sem_getvalue(&right2left, &onRope);
        ^
/Library/Developer/CommandLineTools/SDKs/MacOSX10.14.sdk/usr/include/sys/semapho
re.h:54:56: note:
      'sem_getvalue' has been explicitly marked deprecated here
int sem_getvalue(sem_t * __restrict, int * __restrict) __deprecated;
                                                       ^
/Library/Developer/CommandLineTools/SDKs/MacOSX10.14.sdk/usr/include/sys/cdefs.h
:176:40: note:
      expanded from macro '__deprecated'
#define __deprecated     __attribute__((deprecated))
                                   ^
project3.c:110:3: warning: 'sem_destroy' is deprecated
      [-Wdeprecated-declarations]
  sem_destroy(&left2right);
  ^
/Library/Developer/CommandLineTools/SDKs/MacOSX10.14.sdk/usr/include/sys/semapho
re.h:53:26: note:
      'sem_destroy' has been explicitly marked deprecated here
int sem_destroy(sem_t *) __deprecated;
                         ^
/Library/Developer/CommandLineTools/SDKs/MacOSX10.14.sdk/usr/include/sys/cdefs.h
:176:40: note:
      expanded from macro '__deprecated'
#define __deprecated     __attribute__((deprecated))
                                   ^
project3.c:111:3: warning: 'sem_destroy' is deprecated
      [-Wdeprecated-declarations]
  sem_destroy(&right2left);
  ^
/Library/Developer/CommandLineTools/SDKs/MacOSX10.14.sdk/usr/include/sys/semapho
re.h:53:26: note:
      'sem_destroy' has been explicitly marked deprecated here
int sem_destroy(sem_t *) __deprecated;
                         ^
/Library/Developer/CommandLineTools/SDKs/MacOSX10.14.sdk/usr/include/sys/cdefs.h
:176:40: note:
      expanded from macro '__deprecated'
#define __deprecated     __attribute__((deprecated))
                                   ^
project3.c:125:1: warning: control reaches end of non-void function
      [-Wreturn-type]
}
^
project3.c:133:1: warning: control reaches end of non-void function
      [-Wreturn-type]
}
^
8 warnings generated.
ld: library not found for -lrt
clang: error: linker command failed with exit code 1 (use -v to see invocation)
Catalinas-MacBook-Pro:project3_first catalinasanchez-maes$
```

Fig. 5 Error message for compilation of project3; missing tools

Figure 1 and Figure 3, run on windows and raspberry pi respectively, are visually implementing the output properly in the format of L,R,R,L,L,L,R,R,R,R,L,L,L,L,L. However, there is a contrast with an extra L on a Suze Linux SSH computer Babbage from a MacBook as shown in Figure 4. This was also tested on Euler Linux machine from the MacBook with the same results of an extra L at the end. When running the on the MacBook terminal not all tools were available and failed to compile given in Figure 5. MacOS decapitates requiring different solutions for OS when using XCode as there is not support for unnamed semaphores and instead are required to be implemented with other naming [4]. It is already known that there are segmentation fault issues when using a Mac machine. XCode updates are important to be aware of as multiple changes are added and some are unsupported.

```
marco@DESKTOP-625N2SQ:/mnt/c/schoollinux/cs471/project3$ time ./project3 allleft.txt 1
This program will print out the order that the monkeys leave in the form of L#: (#, #, #)
Where the first number indicates which monkey it is, and the array of numbers indicates
What monkeys where on the rope the moment before it left. If working correctly, the first
number should be equal to the lowest nonzero number in the array next to it.
Note: The array position of the numbers in the array do not indicate the postion of the baboons
on the rope. Only the values do.

L1:(1 3 2)
L2:(2 3 4)
L3:(3 0 4)
L4:(4 6 5)
L5:(5 6 7)
L6:(6 0 7)
L7:(7 9 8)
L8:(8 9 10)
L9:(9 0 10)
L10:(10 0 11)
L11:(11 0 0)


real    0m4.010s
user    0m0.000s
sys     0m0.000s
```

Fig. 6 Revised output to prove FIFO property

Figure 6 demonstrates the FIFO property of the individual baboons on the rope. This run is an example of when multiple monkeys get on right after another one leaves, yet they still maintain the FIFO order.

5 Conclusions

Mutual exclusion, deadlock and starvation avoidance is required to more efficiently implement concurrency with a shared resource. There are more efficient algorithms such as Dekker's and Peterson's algorithms to ensure safe synchronization in which the mentioned above are inherit in the proper implementations of the algorithms.

The use of unnamed semaphores have been deprecated on MacOS, XCode support required. The detriment of software-based concurrency solutions is that it requires constant updates of tools being supported which are not universal throughout different operating systems. A concurrent approach is significantly faster than a linearized approach. The FIFO solution derived from Dekker, Peterson and Lamport's algorithm creates a 53% speedup in runtime.

References

[1] G. L. Peterson, "Myths about the Mutual Exclusion Problem," *Information Processing Letters*, 14-Nov-2017. [Online]. Available: https://zoo.cs.yale.edu/classes/cs323/doc/Peterson.pdf. [Accessed: 22-Nov-2020].

[2] K. Alagarsamy, "Some Myths about Mutual Exclusion Algorithms," *CiteSeerX*, 05-Aug-2003. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.113.3361. [Accessed: 23-Nov-2020].

[3] L. Lamport, *A New Solution of Dijkstra's Concurrent Programming Problem*, 21-May-2002. [Online]. Available: https://lamport.azurewebsites.net/pubs/bakery.pdf. [Accessed: 22-Nov-2020].

[4] Nippysaurus, "sem_init on OS X," *Stack Overflow*, 01-Nov-1958. [Online]. Available: https://stackoverflow.com/questions/1413785/sem-init-on-os-x/1452182. [Accessed: 22-Nov-2020].