# CS474
# Operating Systems I
# Project 3: Synchronization

Jason Miller, Ne'kko Montoya, Navarre Brown

***Abstract*—This project was completed for the CS472: Operating Systems I course. The project demonstrated the concept of process synchronization. It allowed me to experiment with process synchronization mechanisms.**

## I. Introduction

The goal of Project 3 is to showcase what has been learned in the past semester. To Demonstrate this our group will be doing Synchronization. The goal is to use semaphores to protect a limited size resource, such as baboons using a rope to cross a canyon. A **semaphore** *(S)* is an integer variable that apart from initialization it can only be accessed through two standard atomic operations: *wait()* and *signal()*. All operations which modify the integer value of the semaphore must be done atomically. Two processes cannot modify the semaphore simultaneously.

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

Fig. 1. The wait() Atomic Operation

```
signal(S) {
    S++;
}
```

Fig. 2. The signal() Atomic Operation

The project Consists of 5 semaphores. **rope** which is used to ensure the baboons go in only one direction at a time. **mutexL** a mutex semaphore for the left side of the canyon. **mutexR** a mutex semaphore for the right side of the canyon. **Dp**, a deadlock prevention semaphore. **maxOnRope** semaphore for making sure only three baboons enter the rope.

A *file* is read in with a sequence to indicate which side of the rope the baboon is attempting to cross the rope from.

Baboons are separated into their respective threads with initialized semaphores for processing so they can safely cross the canyon. Destroying the semaphores.

## II. Source Code

This project was written using the C programming language and was designed for the Linux Operating System. Therefore, all tests were conducted on laboratory computers using SSH.

### A. Included Libraries

I included a total of 10 different libraries. Once again, this project was designed for the Linux operating system, therefore some of them may be operating system dependent:

- *pthread.h*
- *stdio.h*
- *stdlib.h*
- *unistd.h*
- *sys/types.h*
- *sys/ipc.h*
- *sys/shm.h*
- *sys/wait.h*
- *fcntl.h*
- *semaphore.h*

### B. Defining Constants

A few constants were needed for this project to utilize thread safety, define crossings for baboons L and.

- *goingRight and goingLeft* - Allows program to process if another direction is on the rope, to wait until all baboons are off rope.
- *maxOnRope* - Allowed baboons on rope semaphore
- *timeToCross* - Sleep for amount of time it takes to cross the rope

### C. Necessities

A few other key things were needed before writing the remainder of the program: function prototypes,semaphores, a file reader, and a couple of flags.

```
//initialize vairbales
char c;                                 //temp char variable to hold the characters read from the file
int numBaboons=0;                       //total number of baboons
char temp[100];                         //temp char array for input from file
int baboonsL=0;                         //number of baboons on the left side of the canyon
int baboonsR=0;                         //number of baboons on the right side of the canyon
int numbaboonsL=0;                      //tracking the number of threads made for left side of canyon
int numbaboonsR=0;                      //tracking the number of threads made for right side of canyon
int leftID[baboonsL];                   //tracking id for each baboon om east side of canyon
int rightID[baboonsR];                  //tracking id for each baboon om west side of canyon
pthread_t left[baboonsL],right[baboonsR];   //threads for baboons on the left side and right side if the canyon
```

Fig. 3. Additional Necessities threads

### D. input.txt file

There is an additional file which holds the characters that the producer will read to then place into the shared buffer. This file was called *input.txt* and holds the characters L and R. L being shown 3 times and R being shown 6 times. This is shown below:


Fig. 4. input.txt file

### E. Main Function

Writing the program, we started with reading in the file contents from *input.txt* in main with the specified file pointer. We Then started initializing the variables necessary for getting the baboons across the canyon. numBaboons for total number of baboons on processes left and right. BaboonsL corresponds to the left and vice versa for Right. LeftID for each baboon in east and rightID for the number of baboons in the west. Then using guidelines given based on previous assignments we initialize threads *left* and *right* for our baboonsL and baboonsR. These two threads correspond to thread1 - the consumer and thread2 - the producer. we also declared the attribute pointer array as done in previous assignments which was used to attach a shared memory segment.

```c
/* Create shared memory and pthreads as instructed on assignment*/
int i;
int shmid;
pthread_t tid1[1]; // pthread process id
pthread_t tid2[1];
pthread_attr_t attr; //attribute pointer array

char *shmadd;
shmadd = (char *) 0;

/* Check as done in previous sassignments*/
if ((shmid = shmget (SHMKEY, sizeof(int), IPC_CREAT | 0666)) < 0) {
    perror("shmget");
    return 1;

}
if ((buffer = (sbuffer *) shmat (shmid, shmadd, 0)) == (sbuffer *) - 1) {
    perror("shmat");
    return 0;
}
```

Fig. 5. Declaring Variables and Initializing Shared Memory for Use

```c
//declare semaphores
sem_t rope;
sem_t mutexL;
sem_t mutexR;
sem_t dp;
sem_t maxOnRope;

//declare global variables
int goingRight = 0;
int goingLeft = 0;
int timeToCross;
```

Fig. 6. Semaphores, and global variables

### F. Left Thread

Next we had declared two functions, *leftside* and *rightside*. Both functions are identical but only work for a direction specified in main for the baboons. These functions have and id pointer for the baboon id number that is read. To keep track of the remaining spots on the rope for baboons a *remainingSpots* variable was initialized. Then we built a critical section, where sem_wait(dp); and sem_wait(mutexL); was defined for deadlock prevention. We then check for the first instance of that direction based on if the baboon from the left is going right, and if that is so, the process should wait for all the baboons are off the rope. Then the section for tracking the number of baboons was created. We begin by establishing a basis to print the baboons to the terminal. sem_wait(maxOnRope) was initialized to decrement the *maxOnRope* value by one. Then sem_getValue() pointing to both *maOnRope* and *remainingSpots* this saves the current value of the max on the rope so we know how many remaining spots are left. Now we get some input and notify the user on the baboon that is allowed to start crossing left to right. after execution we call sem_getValue() again to save the current amount left on rope. Then we notify the user that the baboons have finished crossing.

```c
void* leftSide(void*arg)          {
    int id = *(int*)arg;//variable for baboon id number
    int remainingSpots = 0;//variable for number of remaining spots on the rope

    //critical section
    sem_wait(&dp);
    sem_wait(&mutexL);
    goingRight++;
    if (goingRight == 1){//first instance of that direction
        sem_wait(&rope);//if another direction is on rope, wait until all baboons off rope
        printf("Baboon %d: waiting\n", id);
    }//end if
    sem_post(&mutexL);
    sem_post(&dp);

    //section for tracking number of baboons on rope and printing to terminal
    sem_wait(&maxOnRope);//decriment the maxOnRope value by one
    sem_getvalue(&maxOnRope, &remainingSpots);//save current value of maxOnRope value into remainingSpots variable
    printf("Baboon %d: Allowed to start crossing left to right (Number of baboons on rope: %d)\n", id, 3-remainingSpots);//
    sleep(timeToCross);//sleep for amount of time it takes to cross the rope
    sem_getvalue(&maxOnRope, &remainingSpots);//save current value of maxOnRope value into remainingSpots variable
    printf("Baboon %d: Finished crossing left to right rope (Number of baboons on rope: %d)\n", id, 2-remainingSpots);
    sem_post(&maxOnRope);//incriment the maxOnRope value by one

    //critical section
    sem_wait(&mutexL);
    goingRight--;
    if (goingRight == 0){
        sem_post(&rope);//all baboons of this direction off rope, allow for other direction to enter rope
    }//end if
    sem_post(&mutexL);
}//end leftSide
```

Fig. 7. LeftSide process

### G. Right thread

The right thread is responsible for the baboons assigned to the right. and functions the same as the left thread.

### H. Compilation

Compilation is very simple, using gcc:

$$gccprob1.c - o - lpthread - lrt$$

### I. Conclusion

In order to accurately test my code, I had to SSH into a school laptop. What was expected was the producer to produce a few characters and then the consumer to consume them. This is exactly what was seen. Process synchronization is important when we have to processes running concurrently. It was very important for the two processes to share the data without interference. This was seen at the output as the processes ran

```
//thread handling west to east travel
void* rightSide(void*arg){
    int id = *(int*)arg;//variable for baboon id number
    int remainingSpots = 0;//variable for number of remaining spots on the rope

    //critical section
    sem_wait(&dp);
    sem_wait(&mutexR);
    goingLeft++;
    if (goingLeft == 1){//first instance of that direction
        sem_wait(&rope);//if another direction is on rope, wait until all baboons off rope
        printf("Baboon %d: waiting\n", id);
    }//end if
    sem_post(&mutexR);
    sem_post(&dp);

    //section for tracking number of baboons on rope and printing to terminal
    sem_wait(&maxOnRope);//decriment the maxOnRope value by one
    sem_getvalue(&maxOnRope, &remainingSpots);//save current value of maxOnRope value into remainingSpots variable
    printf("Baboon %d: Allowed to start crossing right to left (Number of baboons on rope: %d)\n", id, 3-remainingSpots);
    sleep(timeToCross);//sleep for amount of time it takes to cross the rope
    sem_getvalue(&maxOnRope, &remainingSpots);//save current value of maxOnRope value into remainingSpots variable
    printf("Baboon %d: Finished crossing right to left (Number of baboons on rope: %d)\n", id, 2-remainingSpots);
    sem_post(&maxOnRope);//incriment the maxOnRope value by one

    //critical section
    sem_wait(&mutexR);
    goingLeft--;
    if (goingLeft == 0){
        sem_post(&rope);//all baboons of this direction off rope, allow for other direction to enter rope
    }//end if
    sem_post(&mutexR);
}//end rightSide
```

Fig. 8. Rightside Process

correctly without interference. The test result demonstrated the producer producing its 9 characters, then the consumer consuming them. This test result is seen on the next page.

## REFERENCES

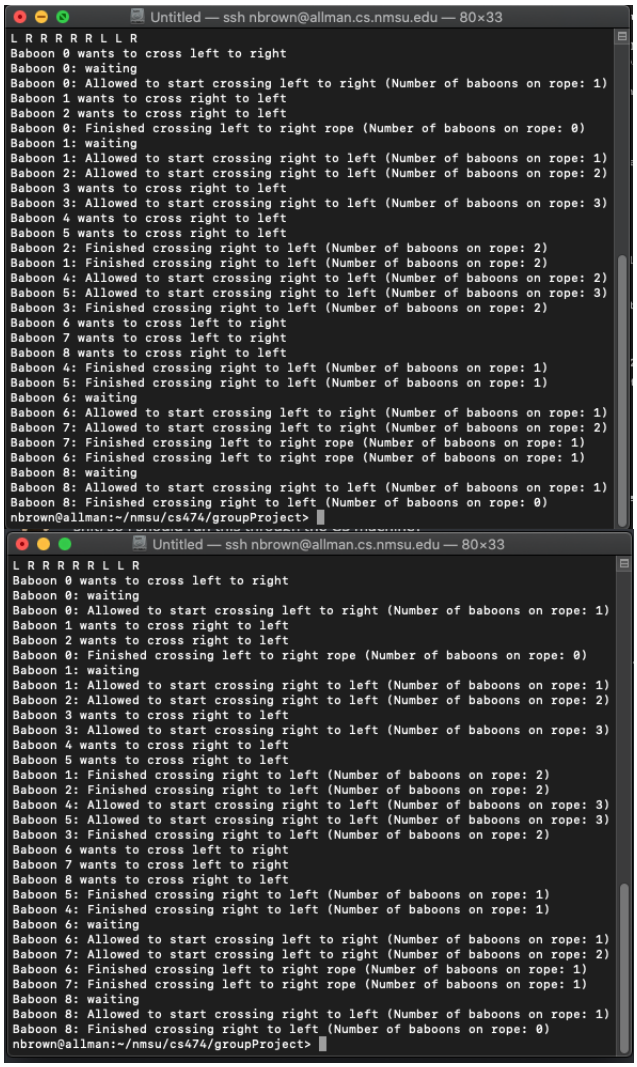


Fig. 9. Test Result