Explanation of the Baboons Crossing Problem Solution

Problem Overview

The problem describes a synchronization challenge where baboons attempt to cross a canyon using a single rope. The constraints are:

- Baboons crossing in opposite directions must not meet on the rope.
- The rope supports at most 5 baboons at a time.
- No starvation: baboons waiting to cross in one direction should eventually get a chance even if there is a continuous stream in the opposite direction.

Code Explanation

Global Declarations

```c
#define MAX_BABOONS 100
#define MAX_BABOONS_ON_ROPE 5

sem_t mutex;
sem_t leftQueue, rightQueue;

int count = 0;
int direction = 0;
int leftWaiting = 0;
int rightWaiting = 0;

char rope[MAX_BABOONS_ON_ROPE];
```

Explanation:

- `MAX_BABOONS` and `MAX_BABOONS_ON_ROPE` define the total number of baboons and the rope capacity.
- `mutex`: a binary semaphore for mutual exclusion.
- `leftQueue`, `rightQueue`: semaphores for baboons waiting on each side.
- `count`: tracks the number of baboons currently on the rope.
- `direction`: tracks the current direction of crossing (`0` = none, `-1` = left-to-right, `1` = right-to-left).
- `leftWaiting`, `rightWaiting`: count baboons waiting to cross from each side.
- `rope[]`: visual representation of baboons on the rope.

Baboons Crossing Functions

Two thread functions represent baboons crossing from each side.

Left to Right

```
void* cross_left_to_right(void* arg) {
    sem_wait(&mutex);
    leftWaiting++;
    while (direction == 1 || count == MAX_BABOONS_ON_ROPE) {
        sem_post(&mutex);
        sem_wait(&leftQueue);
        sem_wait(&mutex);
    }
    ...
}
```

Explanation:

- Baboons wait if direction is the opposite or the rope is full.
- Once safe, direction is set and `count` is incremented.

After crossing:

```
    count--;
    rope[count] = ' ';
    if (count == 0) {
        direction = 0;
        if (rightWaiting > 0) {
            for (int i = 0; i < rightWaiting && i < MAX_BABOONS_ON_ROPE; i++) {
                sem_post(&rightQueue);
            }
        } ...
    }
```

Explanation:

- When no baboons are on the rope, the direction is reset.
- Baboons waiting on the opposite side are released, ensuring fairness (prevents starvation).

Right to Left

```
void* cross_right_to_left(void* arg) {
    sem_wait(&mutex);
    rightWaiting++;
    while (direction == -1 || count == MAX_BABOONS_ON_ROPE) {
        sem_post(&mutex);
        sem_wait(&rightQueue);
        sem_wait(&mutex);
    }
    ...
}
```

Explanation:

- Similar to left-to-right but for the opposite direction.
- Ensures only one direction uses the rope at a time and max 5 baboons.

Main Function

```c
int main() {
    sem_init(&mutex, 0, 1);
    sem_init(&leftQueue, 0, 0);
    sem_init(&rightQueue, 0, 0);

    for (int i = 0; i < MAX_BABOONS; i++) {
        if (rand() % 2 == 0)
            pthread_create(&baboons[i], NULL, cross_left_to_right, NULL);
        else
            pthread_create(&baboons[i], NULL, cross_right_to_left, NULL);
    }
    ...
}
```

Explanation:

- Semaphores are initialized.
- Randomly creates threads representing baboons from both sides.
- Waits for all threads to finish and cleans up.

Mapping to Problem Constraints

- **Safety (no fights):** Direction is enforced via the `direction` variable. Baboons wait if the rope is used in the opposite direction.
- **Capacity (no breakage):** The `count` variable ensures no more than 5 baboons are on the rope.
- **No Starvation:** When rope is empty, baboons on the opposite side are signaled. If none, same-side baboons are allowed.