Assignment #4b
Jesus Gutierrez

This program the master thread produces random numbers continuously and places them in a buffer, where the worker threads will then consume those given numbers (print those numbers on the terminal).

Disclaimer: I implemented two semaphores so I can track the number of empty slots in the buffer, and another one that tracks the number of full slots in the buffer. Also, for demonstrating purposes, I created a buffer of size 10, where the master will generate a random number between 0 and 99 that the worker will consume.

```
sem_t semaphoreEmpty;
sem_t semaphoreFull;
```

```
int buffer[10];
int count = 0;
```

```
// Produce
int x = rand() % 100;
```

TASKS:

1. *This function will remove/consume items from the shared buffer and print them to screen.*

   We implemented the function 'worker' that will work to first remove from the buffer, and then successfully consume the items from the shared buffer. After these executions are done, we will print the generated items to the screen. As shown below:

```c
void* worker(void* args) {
    while (1) {
        int y;
        /*
        wroker will remove the used point in the buffer
        */
        sem_wait(&semaphoreFull);
        /*
        mutex locks are used to prevent master and worker to run exactly
        at the same time, we lock, then unlock for both threads.
        */
        pthread_mutex_lock(&mutex);
        y = buffer[count - 1];
        count--;
        pthread_mutex_unlock(&mutex);
        sem_post(&semaphoreEmpty);
        /*
        wroker will consume the available from buffer,
        then print it to the terminal
        */
        printf("Got %d\n", y);
        /*
        Sleep function used so we can reduce cpu time
        with values that are skipped.
        */
        sleep(1);
    }
}
```

```
4b — -zsh — 80×24
jesusgtz@Jesuss-MBP 4b % ./master
Got 0
Got 1
Got 0
Got 5
Got 33
Got 50
Got 81
Got 24
Got 45
Got 52
Got 42
Got 75
Got 20
Got 19
Got 68
Got 65
Got 13
Got 23
Got 19
Got 81
Got 65
Got 72
Got 12
```

2. *You must add logic to correctly synchronize the producer and consumer threads in such a way that every number is produced and consumed exactly once.*
3. *Producers must not try to produce when the buffer is full, and consumers should not consume from an empty buffer.*

Answer for 2: logic has been correctly added so that the producer's and consumer's threads produce and consume exactly once every execution with sleep(1) as shown above.

Answer for 3: We initialized two semaphores so that producers do not try to produce when the buffer reaches its limit. Same thing with consumers, they cannot consume from an empty buffer.

More comments are provided in code for full understanding of how I used the semaphores to accomplish the tasks.

```
sem_open(&semaphoreEmpty, 0, 10);
sem_open(&semaphoreFull, 0, 0);
```

```
void* worker(void* args) {
    while (1) {
        int y;
        /*
        wroker will remove the used point in the buffer
        */
        sem_wait(&semaphoreFull);
        /*
        mutex locks are used to prevent master and worker to run exactly
        at the same time, we lock, then unlock for both threads.
        */
        pthread_mutex_lock(&mutex);
        y = buffer[count - 1];
        count--;
        pthread_mutex_unlock(&mutex);
        sem_post(&semaphoreEmpty);
        /*
        wroker will consume the available from buffer,
        then print it to the terminal
        */
        printf("Got %d\n", y);
        /*
        Sleep function used so we can reduce cpu time
        with values that are skipped.
        */
        sleep(1);
    }
}
```

```
void* master(void* args) {
    while (1) {
        /*
        master will produce the random number that
        will be used towards the buffer.
        */
        int x = rand() % 100;
        /*
        Sleep function used so we can reduce cpu time
        with values that are skipped.
        */
        sleep(1);
        /*
        master then adds to the buffer. 'sem_wait' is used until
        there's an empty slot. If 'semaphoreEmpty' = 0, there are no slots.
        Therefore we wait -> no point in proceeding
        */
        sem_wait(&semaphoreEmpty);
        /*
        mutex locks are used to prevent master and worker to run exactly
        at the same time, we lock, then unlock for both threads.
        */
        pthread_mutex_lock(&mutex);
        buffer[count] = x;
        count++;
        pthread_mutex_unlock(&mutex);
        /*
        once master passes the addition to buffer, we increment the
        semaphore so that we know the space is full.
        */
        sem_post(&semaphoreFull);
    }
}
```

4. *The main thread must call pthread_join on the master and worker threads, and terminate itself once all threads have joined.*

Answer for 4: 'pthread_join' I used on the master and worker threads to successfully join both threads so we can work together. Once all the threads have joined, we 'sem_close' so that threads terminate themselves once they have joined. Code screenshot shown below of implementation:

```c
    /*
    this loop is used to join both threads using 'pthread_join'
    */
    for (i = 0; i < THREADS; i++) {
        if (pthread_join(thread[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    /*
    at the end of all execution of threads with buffer, we destroy
    the mutex + both semaphores(then exit).
    */
    sem_close(&semaphoreEmpty);
    sem_close(&semaphoreFull);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

5. *Your solution must only use pthreads condition variables for waiting and signaling: busy waiting is not allowed.*

   Answer for 5: Solutions in my code only use pthread condition variables for waiting and signaling.

HOW TO RUN CODE:

Master-worker.c:

- *Gcc -o master master-worker.c*
- *Will generate 2 warnings, dismiss and run with below command:*
- *./c*