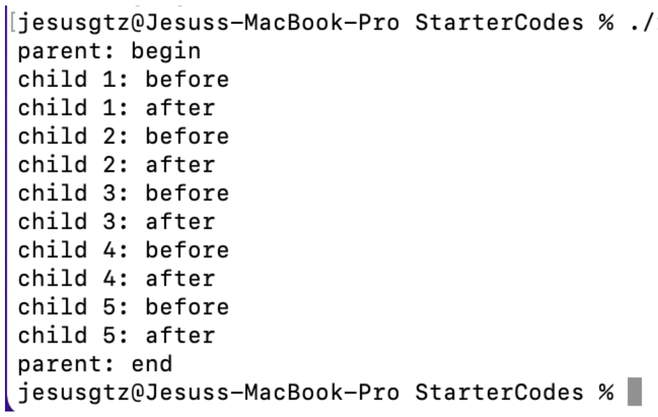
Assignment #4c
Jesus Gutierrez

The goal of this part is to allow you exploring the applicability of semaphores in solving two simple synchronization related problems.

Disclaimer: I implemented two semaphores so I can track the number of empty slots for every child for the rendezvous point. Also, for demonstrating purposes, I created a few other children and threads so I could really understand what was going on with my semaphores.

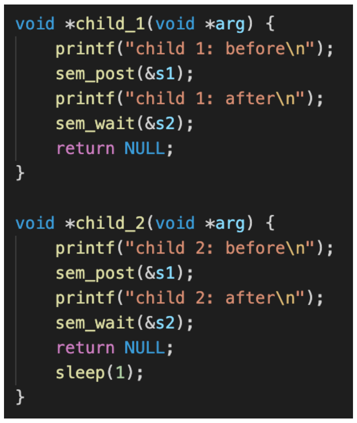
Problem #1:
From what I understood, if done correctly this assignment, each child should print their own "before" message before it prints out the "after" message. Following the next child with a "before" message.



```
jesusgtz@Jesuss-MacBook-Pro StarterCodes % ./r
parent: begin
child 1: before
child 1: after
child 2: before
child 2: after
child 3: before
child 3: after
child 4: before
child 4: after
child 5: before
child 5: after
parent: end
jesusgtz@Jesuss-MacBook-Pro StarterCodes %
```

As you can see in the image above, every child first prints "before" before actually printing the "after" message, following the next child after with its own "before" message.

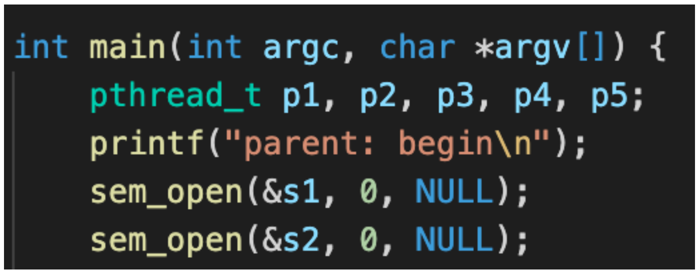
We implemented the two semaphores in every child function as shown below so that ‘post’ function can increment so that we know that the space is filled for the child. The ‘wait’ function is used until there is an empty spot, that is where the next child comes into it with ‘post’.



```
void *child_1(void *arg) {
    printf("child 1: before\n");
    sem_post(&s1);
    printf("child 1: after\n");
    sem_wait(&s2);
    return NULL;
}

void *child_2(void *arg) {
    printf("child 2: before\n");
    sem_post(&s1);
    printf("child 2: after\n");
    sem_wait(&s2);
    return NULL;
    sleep(1);
}
```

Below we show how we initialized the semaphores in the main function, NULL is used as the final parameter due to us not knowing how many child processes were going to have to work with, integer 0 as default so we can begin from the first child:



```
int main(int argc, char *argv[]) {
    pthread_t p1, p2, p3, p4, p5;
    printf("parent: begin\n");
    sem_open(&s1, 0, NULL);
    sem_open(&s2, 0, NULL);
```

HOW TO RUN CODE:

rendezvous.c:

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

Problem #2:

Now, you need to go one step further by implementing a general solution to **barrier synchronization**. Assume there are two points in a sequential piece of code, called P1 and P2. We are needed to put a barrier between P1 and P2, which will guarantee that all threads will execute P1 before any one thread executes P2.

Your task: write the code to implement a *barrier()* function that can be used in this manner. It is safe to assume you know **N** (the total number of threads in the running program) and that all N threads will try to enter the barrier. Again, you should likely use two semaphores to achieve the solution, and some other integers to count things. See ***barrier.c*** for details

<u>*HOW TO RUN CODE:*</u>

<u>*Master-worker.c:*</u>

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