Take a look at the program `thread_fill.c`. It contains three functions:

- `fillArray` is a simple function to fill all buckets of a `float` array with a specified value.
- `threadMain` unpacks a struct containing the thread's arguments then calls `fillArray`.
- `main` allocates a data array, then spawns several worker threads, sending an argument struct to each.

Stack Diagram

Your first task is to draw a stack diagram of this multi-threaded program. In a single-threaded stack diagram, we are capturing a single moment in the execution state of the program. In a multi-threaded program, we can't make any assumptions about how the execution of different threads will be interleaved on the computer's processor(s). However, for this exercise, we will assume that all threads pause on the first line of `fillArray`. Draw a stack diagram for this state of the program, indicating all local variables and all dynamically-allocated memory. We'll change them later, but for now you should assume the default values:

```
#define NUM_THREADS 3
#define ARRAY_SIZE 10
```

Assigning Work to the Threads

Your next task is to update the program so that the threads perform useful work. In particular, we want to divide the task of filling the data array (approximately) equally between the threads. We will need to pass to each thread:

- a pointer to the location where it should start filling in data,
- the size of the array slice for which it is responsible, and
- the value it should be filling in.

Since we can only pass a single `void*` to each thread, we need to package these inputs into a struct. Add additional fields for storing each of these values to the `thread_fill_data` struct definition.

Before the `main` function calls `pthread_create`, it needs to fill in the struct that is being passed to the new thread. Modify the first loop in `main` to send to each thread its starting

point, its size, and the value to fill in. To distinguish what got filled in by which thread, let's pass the `thread_num`, converted to a `float`.

Then inside the `threadMain` function, we need to unpack the struct. The starting-point code already demonstrates casting the `void*` and extracting one value. Add code to this function that extracts all of the struct's contents into local variables. Then modify the call to `fillArray` to use those local variables.

Filling a Big Array

Let's now change the constants at the top of the program to spawn more threads and fill a bigger array.

```
#define NUM_THREADS 8
#define ARRAY_SIZE 10000000
```

What do you notice about the order in which the threads execute? What do you notice about the order in which `main` joins the other threads?

Finally, think about how well you've succeeded in balancing work across threads. What would happen if the total amount of work is close to/far from a multiple of the number of threads?

Incrementing a Shared Variable

Take a look at the program `thread_increment.c`. What do you expect will happen if we run this program?

Once you've made and discussed your guesses, try running it a few times. Did it work out the way you expected? Why or why not? How could we improve this program?