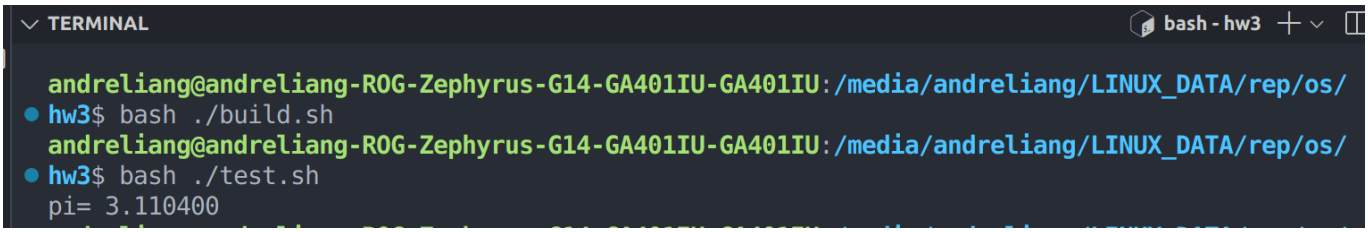
Operating System, Spring 2023 HW3

B11705009 An-Che, Liang

Execution result:

This is the execution result of problem Q2:



And these are the bash script I use to build and test my program:

```
gcc ./Q2.c -o ./Q2 -lm
```

```
./Q2
```

Problem Q1:

```
do{
    flag[i] = true;

    while(flag[j]){
        if (turn == j){
            flag[i] = false;
            while(turn == j)
                ;
            flag[i] = true;
        }
    }

    /* critical section */

    turn = j;
    flag[i] = false;

    /* remainder section */
}while(true)
```

The three requirements of the critical-section problem are:

1. Mutual exclusion.
2. Progress.
3. Bounded Waiting.

We can see that the `turn` variable actes like a lock, `turn=i` means thread `i` is currently in the critical section, and it is set by the thread `j`, makes it impossible for two thread to enter the critical section at the same time, thus satisfy the first requirement.

We can also see that the loop:

```
while(flag[j]){
        if (turn == j){
            flag[i] = false;
            while(turn == j)
                ;
            flag[i] = true;
        }
    }
```

Is waiting for the other thread to leave the critical section, and there are no such waiting in the remainder section, thus satisfy the second requirement.

Finally, since there are only two threads, thus when a thread is waiting to enter the critical section, the other thread is currently in the critical section, then after the other thread exit the critical section, the origanl thread can enter the critical section immediately. Therefore satisfy the third requirement.

Probelm Q2:

First, we create a lock by with `pthread_mutex_t mutex;`, then before we generate all the threads, we can initialize the lock by `pthread_mutex_init(&mutex, 0);`.

Afterward, we define the thread function:

```
void *thread_run(void *arg)
{
    for (int i = 0; i < POINT_COUNT; i++)
    {
        double x, y, dx, dy;
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;
        dx = x - 0.5;
        dy = y - 0.5;
        if (sqrt(dx * dx + dy * dy) <= 0.5)
        {
            pthread_mutex_lock(&mutex);
            cnt += 1;
            pthread_mutex_unlock(&mutex);
        }
    }
```

```
    pthread_exit(0);
}
```

Note that the variable `cnt` is a global variable, so in order to ensure data consistency, we have to change it's value inside the critical section. `pthread_mutex_lock(&mutex)` will lock the mutex if the mutex is free, and will wait for the mutex to be free if the resource is currently be held by another thread. `pthread_mutex_unlock(&mutex)` will release the mutex.

Then, we generate 5 threads with `pthread_create()` and wait for them to complete with `pthread_join()`. Then we can calculate `p=cnt/(THREAD_COUNT * POINT_COUNT)`, while value `p` represent the ratio of the circle's area to the square's area, since circle's area is π/4, we can estimate π=p*4.