LABORATORIO: PThreads

Estudiante: Randú Jean Franco Cerpa García

1. Secuencial

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main(int argc, char **argv){
    if (argc < 2){ fprintf(stderr,"Uso: %s N\n", argv[0]); return 1; }
    long long n = strtoll(argv[1], NULL, 10);

    double sum = 0.0;
    for (long long i = 0; i < n; i++){
        double term = ((i & 1) ? -1.0 : 1.0) / (double)(2*i + 1);
        sum += term;
    }
    printf("Secuencial: N=%lld  pi=%.12f\n", n, 4.0*sum);
    return 0;
}
```

Calcula pi con la serie de Leibniz sin threads.

```
jean@DESKTOP-I276I98:~$ ./pi_seq 100000000
Secuencial: N=100000000  pi=3.141592643589
```

2. Busy-waiting dentro del for
   Hay turnos estrictos, en cada iteración del bucle un hilo gira (busy-wait) hasta que el “banderín” (flag) indique “es mi turno”. Entonces suma 1 término y pasa el turno al siguiente hilo.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>

static long long n = 0;
static int thread_count = 0;
static double sum = 0.0;

// turno global 0..thread_count-1
static atomic_int flag = 0;

void* Thread_sum(void* rank){
    long my_rank = (long)rank;

    long long my_n       = n / thread_count;
    long long my_first_i = my_n * my_rank;
    long long my_last_i  = my_first_i + my_n;
    // último hilo absorbe residuo
    if (my_rank == thread_count - 1) my_last_i += n % thread_count;

    double factor = (my_first_i % 2 == 0) ? 1.0 : -1.0;

    for (long long i = my_first_i; i < my_last_i; i++, factor = -factor) {
        // Busy-wait (espera activa) por TURNO en CADA iteración
        while (atomic_load_explicit(&flag, memory_order_acquire) != my_rank) { /* spin */ }
        sum += factor / (2.0*i + 1.0);
        atomic_store_explicit(&flag, (my_rank + 1) % thread_count, memory_order_release);
    }
    return NULL;
}
```

Se declaran las variables y se divide el trabajo, “n” entre la cantidad de threads que hay, el último bloque absorbe el residuo (si es que hay). El factor (que es una variable local) se inicia en positivo o negativo de acuerdo al valor del primer número, y va cambiando de signo para cumplir con la serie de Leibniz. Se hace una “my_n” cantidad de turnos por hilo. Dentro del for se encuentra la región crítica que es donde esperan activamente por su turno en cada término. Derrocha CPU.

```
jean@DESKTOP-I276I98:~$ ./pi_busy_in 100000000 8
Busy-wait DENTRO: N=100000000 T=8  pi=3.141592643589
jean@DESKTOP-I276I98:~$ ./pi_busy_in 100000000 2
Busy-wait DENTRO: N=100000000 T=2  pi=3.141592643589
```

3. Busy-waiting afuera del for
   Cada hilo trabaja en paralelo sobre su suma local y entra a la región crítica una sola vez al final, usando también un turno (busy-wait) para sumar el parcial a sum. Muchísima menos contención.

```
23          for (long long i = my_first_i; i < my_last_i; i++, factor = -factor) {
24              local += factor / (2.0*i + 1.0);
25          }
26          // Busy-wait por turno UNA vez (sección crítica mínima)
27          while (atomic_load_explicit(&flag, memory_order_acquire) != my_rank) { /* spin */ }
28          sum += local;
29          atomic_store_explicit(&flag, (my_rank + 1) % thread_count, memory_order_release);
30
31          return NULL;
32      }
```

Acumulación local y no compartida, y la región crítica solo pasa una vez por hilo para hacer la suma.

```
jean@DESKTOP-I276I98:~$ ./pi_busy_out 100000000 8
Busy-wait FUERA: N=100000000 T=8  pi=3.141592643590
jean@DESKTOP-I276I98:~$ ./pi_busy_out 100000000 2
Busy-wait FUERA: N=100000000 T=2  pi=3.141592643590
```

4. Mutex

```
 9      static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
10
11      void* Thread_sum(void* rank){
12          long my_rank = (long)rank;
13
14          long long my_n       = n / thread_count;
15          long long my_first_i = my_n * my_rank;
16          long long my_last_i  = my_first_i + my_n;
17          if (my_rank == thread_count - 1) my_last_i += n % thread_count;
18
19          double local = 0.0;
20          double factor = (my_first_i % 2 == 0) ? 1.0 : -1.0;
21
22          for (long long i = my_first_i; i < my_last_i; i++, factor = -factor) {
23              local += factor / (2.0*i + 1.0);
24          }
```

Igual que el busy-out (acumular local), pero la región crítica final se protege con pthread_mutex_t. No hay turnos ni busy-wait, si el mutex está ocupado, el hilo se bloquea sin quemar CPU.

```
26          pthread_mutex_lock(&mtx);
27          sum += local;
28          pthread_mutex_unlock(&mtx);
```

Se bloque si otro hilo está dentro, lo que garantiza la exclusión mutua y el orden de memoria.

```
jean@DESKTOP-I276I98:~$ ./pi_mutex 100000000 8
MUTEX: N=100000000 T=8  pi=3.141592643590
```