Cálculo de pi, usando busy wait y mutex

Número de términos de la serie de Maclaurin
n = 100 000 000

# Threads	Busy wait 1	Busy wait 2	Mutex
1	1.527445e+00	3.621290e-01	3.280840e-01
2	7.114722e+00	2.269261e-01	1.932700e-01
4	7.695819e+00	1.556952e-01	1.457739e-01
8	7.695819e+00	1.602950e-01	1.737642e-01
16	-	2.296741e-01	1.736269e-01
32	-	4.273121e-01	1.548951e-01
64	-	1.036318e+00	1.453881e-01

Probado en una laptop i5 con 4 cores, con linux 18.04

Sección crítica Busy 1:

```
14      for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15         while (flag != my_rank);
16         sum += factor/(2*i+1);
17         flag = (flag+1) % thread_count;
18      }
```

Los hilos alternarán entre la espera y la ejecución, y evidentemente, la espera y el incremento al flag aumentan mucho el tiempo de ejecución general.

Sección crítica Busy 2:

```
   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

   return NULL;
}  /* Thread_sum */
```

Este segundo código mejora el rendimiento al usar una variable privada para almacenar su suma. Luego, cada subproceso puede agregar su suma obtenida a la suma global una vez, después del loop.

Sección crítica Mutex

```
for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
   my_sum += factor/(2*i+1);
}
pthread_mutex_lock(&mutex);
sum += my_sum;
pthread_mutex_unlock(&mutex);
```

Similar al busy wait 2, sólo que usa una variable especial, llamada Mutex que significa **mutual exclusion**. El orden en el que se ejecuta es más o menos aleatorio, el primer thread que llegue a la sección crítica ejecutará, luego el que llegué después y así sucesivamente. Es más eficiente que el busy wait, pues no se desperdician recursos mientras se espera el turno para ejecutar la sección crítica.

Comparación Barriers usando Variables de Condición y Busy Wait

# Threads	Busy wait	Cond
1	3.271103e-04	3.080368e-04
2	4.279613e-04	1.946926e-03
4	3.734422e-02	9.428978e-03
8	1.246465e+00	3.563786e-02
16	2.602350e+00	3.555489e-02
32	3.758308e+00	5.151892e-02

Condition Variable

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&mutex);
   counter++;
   if (counter == thread_count) {
      counter = 0;
      pthread_cond_broadcast(&cond_var);
   } else {
      while (pthread_cond_wait(&cond_var, &mutex) != 0);
   }
   pthread_mutex_unlock(&mutex);
   . . .
}
```

Una variable de condición es un objeto que permite que un subproceso suspenda la ejecución hasta que ocurra un determinado evento o condición. Cuando ocurre el evento o condición, otro hilo puede indicar al hilo que se “despierte”. Una variable de condición siempre está asociada con un mutex

Busy Wait

```
/* Shared and initialized by the main thread */
int counter;  /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&barrier_mutex);
   counter++;
   pthread_mutex_unlock(&barrier_mutex);
   while (counter < thread_count);
   . . .
}
```

Para sincronizar los threads se usa un mutex y un bucle. Cuando un thread llega a cierto punto en su ejecución aumenta la variable “counter” en 1, y luego se queda en el while esperando a que los demás threads aumenten el “counter” hasta que sea igual al número de threads.