

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE
AREQUIPA

FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN



Trabajo PThreads 1

Alumno:

GARCIA DIAZ , GERMAN

Docente:

Mamani Aliaga , Alvaro

Arequipa - Perú

1. Implementar y comparar las técnicas de sincronización Busy-waiting y Mutex, obtener una tabla similar a la Tabla 4.1 del libro.

1.1. Busy-Waiting

La ejecución repetida de un bucle de código mientras se espera que ocurra un evento se denomina Busy-Waiting.

La CPU no se dedica a ninguna actividad productiva real durante este período y el proceso no avanza hacia su finalización.

Ocupado en bucle o girando es una técnica en la que un proceso verifica repetidamente si una condición es verdadera, como si la entrada del teclado o un candado está disponible.

```
1 #include <stdio.h>
2 #include <time.h>
3 #include <pthread.h>
4 #include <stdlib.h>
5 // #include "timer.h"
6
7 long double sum;
8 long flag;
9 int thread_count;
10 int n;
11
12 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
13
14 void * thread_sum(void * rank){
15     long my_rank = (long) rank;
16     double factor;
17     long long my_n = n / thread_count;
18     long long my_first_i = my_n * my_rank;
19     long long my_last_i = my_first_i + my_n;
20
21     if(my_first_i % 2 == 0){
22         factor = 1.0;
23     }
24     else{
25         factor = -1.0;
26     }
27
28     for(long long i = my_first_i; i < my_last_i; i++, factor = -factor){
29         while(flag != my_rank);
30         sum += factor / (2*i+1);
31         flag = (flag+1) % thread_count;
32     }
33 }
```

```

34     return NULL;
35 }
36
37 int main(int argc, char ** argv){
38     long thread;
39     clock_t t;
40     clock_t start, end;
41
42     thread_count = 1;
43     while(thread_count < 1000){
44         pthread_t *threads;
45         threads = malloc(thread_count*sizeof(pthread_t));
46
47         start=clock();
48         for(thread= 0; thread < thread_count; thread++){
49             pthread_create(&threads[thread], NULL, thread_sum,&thread);
50         }
51         //printf("Hello from main thread\n");
52         for(thread = 0; thread < thread_count; thread++){
53             pthread_join(threads[thread],NULL);
54         }
55         end=clock();
56         free(threads);
57
58         t = end-start;
59         printf("No de Threads : %d ,Tiempo Tomado es %f seconds \n",
60             thread_count,((float)t)/CLOCKS_PER_SEC);
61         thread_count=thread_count*2;
62     }
63 }

```

1.2. Mutex

Un mutex es una bandera mutuamente excluyente. Actúa como un guardián de una sección de código que permite un hilo y bloquea el acceso a todos los demás. Esto asegura que el código que se está controlando solo será alcanzado por un único hilo a la vez.

El objetivo de un mutex es sincronizar dos subprocesos. Cuando tiene dos subprocesos que intentan acceder a un solo recurso, el patrón general es tener el primer bloque de código que intenta acceder, para establecer el mutex antes de ingresar el código. Cuando el segundo bloque de código intenta acceder, ve que el mutex está configurado y espera hasta que el primer bloque de código este completo (y desestablece el mutex), luego continúa.

```

1 #include <stdio.h>
2 #include <pthread.h>

```

```

3 #include <stdlib.h>
4
5
6 long double sum;
7 long flag;
8 int thread_count;
9 int n;
10
11 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
12
13 void * thread_sum(void * rank){
14     long my_rank = (long) rank;
15     double factor;
16     long long my_n = n / thread_count;
17     long long my_first_i = my_n * my_rank;
18     long long my_last_i = my_first_i + my_n;
19     double my_sum = 0.0;
20
21     if(my_first_i % 2 == 0){
22         factor = 1.0;
23     }
24     else{
25         factor = -1.0;
26     }
27
28     for(long long i = my_first_i; i < my_last_i; i++, factor = -factor){
29         my_sum += factor / (2 * i + 1);
30     }
31
32     pthread_mutex_lock(&mutex);
33     sum += my_sum;
34     pthread_mutex_unlock(&mutex);
35
36     return NULL;
37 }

```

1.3. Resultados y Comparaciones :

Figura 1. Comparaciones de valores obtenidos por 1ra vez:

[Busy-Waiting]

```

No de Threads : 512 ,Tiempo Tomado es 0.023520 seconds
german@german-Inspiron-7577:~/Downloads$ mpicc busy.c -o busy
german@german-Inspiron-7577:~/Downloads$ ./busy
No de Threads : 1 ,Tiempo Tomado es 0.000070 seconds
No de Threads : 2 ,Tiempo Tomado es 0.000133 seconds
No de Threads : 4 ,Tiempo Tomado es 0.000229 seconds
No de Threads : 8 ,Tiempo Tomado es 0.000350 seconds
No de Threads : 16 ,Tiempo Tomado es 0.000498 seconds
No de Threads : 32 ,Tiempo Tomado es 0.001097 seconds
No de Threads : 64 ,Tiempo Tomado es 0.002043 seconds
No de Threads : 128 ,Tiempo Tomado es 0.003403 seconds
No de Threads : 256 ,Tiempo Tomado es 0.006308 seconds
No de Threads : 512 ,Tiempo Tomado es 0.012635 seconds
german@german-Inspiron-7577:~/Downloads$

```

[Mutex]

```

No de Threads : 512 ,Tiempo Tomado es 0.024172 seconds
german@german-Inspiron-7577:~/Downloads$ mpicc mutex.c -o mutex
german@german-Inspiron-7577:~/Downloads$ ./mutex
No de Threads : 1 ,Tiempo Tomado es 0.000279 seconds
No de Threads : 2 ,Tiempo Tomado es 0.000391 seconds
No de Threads : 4 ,Tiempo Tomado es 0.000394 seconds
No de Threads : 8 ,Tiempo Tomado es 0.000871 seconds
No de Threads : 16 ,Tiempo Tomado es 0.001296 seconds
No de Threads : 32 ,Tiempo Tomado es 0.002869 seconds
No de Threads : 64 ,Tiempo Tomado es 0.003626 seconds
No de Threads : 128 ,Tiempo Tomado es 0.002714 seconds
No de Threads : 256 ,Tiempo Tomado es 0.007203 seconds
No de Threads : 512 ,Tiempo Tomado es 0.010991 seconds
german@german-Inspiron-7577:~/Downloads$

```

Figura 2. Comparaciones de valores obtenidos por 2da vez:

[Busy-Waiting]

```

No de Threads : 512 ,Tiempo Tomado es 0.023520 seconds
german@german-Inspiron-7577:~/Downloads$ ./busy
No de Threads : 1 ,Tiempo Tomado es 0.000285 seconds
No de Threads : 2 ,Tiempo Tomado es 0.000302 seconds
No de Threads : 4 ,Tiempo Tomado es 0.000454 seconds
No de Threads : 8 ,Tiempo Tomado es 0.000880 seconds
No de Threads : 16 ,Tiempo Tomado es 0.001637 seconds
No de Threads : 32 ,Tiempo Tomado es 0.002791 seconds
No de Threads : 64 ,Tiempo Tomado es 0.004912 seconds
No de Threads : 128 ,Tiempo Tomado es 0.010441 seconds
No de Threads : 256 ,Tiempo Tomado es 0.020880 seconds
No de Threads : 512 ,Tiempo Tomado es 0.024172 seconds
german@german-Inspiron-7577:~/Downloads$

```

[Mutex]

```

No de Threads : 512 ,Tiempo Tomado es 0.010991 seconds
german@german-Inspiron-7577:~/Downloads$ ./mutex
No de Threads : 1 ,Tiempo Tomado es 0.000222 seconds
No de Threads : 2 ,Tiempo Tomado es 0.000263 seconds
No de Threads : 4 ,Tiempo Tomado es 0.000495 seconds
No de Threads : 8 ,Tiempo Tomado es 0.001167 seconds
No de Threads : 16 ,Tiempo Tomado es 0.001581 seconds
No de Threads : 32 ,Tiempo Tomado es 0.003232 seconds
No de Threads : 64 ,Tiempo Tomado es 0.003712 seconds
No de Threads : 128 ,Tiempo Tomado es 0.003314 seconds
No de Threads : 256 ,Tiempo Tomado es 0.006242 seconds
No de Threads : 512 ,Tiempo Tomado es 0.012933 seconds
german@german-Inspiron-7577:~/Downloads$

```

1.4. Analisis

- ✓ Como observamos en los resultados obtenidos, por lo general Busy-Waiting es quien mas tiempo demora.
- ✓ Funcionamiento de un Mutex :

Cuando tengo una gran discusión en el trabajo, uso un pollo de goma que guardo en mi escritorio para esas ocasiones. La persona que sostiene el pollo es la única persona a la que se le permite hablar. Si no sostienes el pollo, no puedes hablar. Solo puede indicar que quiere el pollo y esperar hasta obtenerlo antes de hablar. Una vez que haya terminado de hablar, puede devolver el pollo al moderador, quien se lo entregará a la siguiente persona que hable. Esto asegura que las personas no hablen entre sí y que también tengan su propio espacio para hablar. Reemplaza el pollo con Mutex y personas con hilo y básicamente tienes el concepto de mutex.

- ✓ el funcionamiento de mutex aparenta ser bueno , solo que en el caso de los procesos necesariamente necesitamos que estos avancen casi a la par o dependiendo de su prioridad , es decir en el caso mutex se espera hasta que un proceso acabe sus tareas y desde ahi recien otro proceso ingresa y puede empezar a trabajar.

2. Basado en la sección 4.7, implementar un ejemplo de productor-consumidor. Explicar porque no se debe utilizar MUTEX

El problema del productor-consumidor en computación es un problema clásico de control de concurrencia. Se suele presentar de forma semi-concreta, lo que ayuda a identificar otros contextos donde su solución se puede aplicar.

El problema lo tienen dos entes concurrentes, uno que produce algo que el segundo consume. Ambos tienen acceso a un espacio finito donde se alojan los productos, comúnmente llamado buffer y de naturaleza circular. Dado que los entes son concurrentes, uno está agregando productos al buffer mientras el otro está al mismo tiempo consumiéndolos (o quitándolos) del buffer. El problema surge cuando ambos entes actúan a diferente velocidad. Si el consumidor es más rápido que el productor, podría consumir productos que aún no se han colocado en el buffer. Si el productor es más rápido que el consumidor, lo alcanza y sobrescribe productos que el consumidor aún no ha procesado.

El problema se resuelve cuando el consumidor procesa todos los productos y en el mismo orden en que el productor los genera. Al igual que ocurre con seres humanos, la solución implica imponer espera. Si el consumidor es más veloz, vaciará el buffer y deberá esperar a que el productor genere el próximo producto. Si el productor es más veloz y satura el buffer, deberá esperar a que el consumidor libere al menos un espacio para poder continuar produciendo.

2.1. Código Ejemplo:

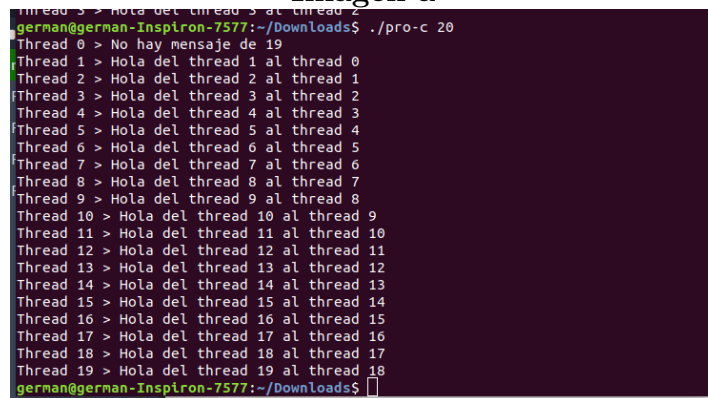
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 const int MAX_THREADS = 1024; //variable para hilos
6 const int MSGMAX = 100;
7
8 /* Variables Globales: accessible to all threads */
9 int thread_count; // para contr los hilos
10 char** messages; //vector
11
12 void Usage(char* prog_name);
13
14 void *Send_msg(void* rank) {
15     long my_rank = (long) rank;
16     long dest = (my_rank + 1) % thread_count;
17     long source = (my_rank + thread_count - 1) % thread_count;
18     char* my_msg = (char*) malloc(MSGMAX * sizeof(char));
19
20     sprintf(my_msg, "Hola del thread %d al thread %d", dest, my_rank);
21     messages[dest] = my_msg;
22
23     if (messages[my_rank] != NULL)
24         printf("Thread %d > %s\n", my_rank, messages[my_rank]);
25     else
26         printf("Thread %d > No hay mensaje de %d\n", my_rank, source);
27
28     return NULL;
29 }
30
31 /*-----*/
32 int main(int argc, char* argv[]) {
33     long thread;
34     pthread_t* thread_handles;
35
36     if (argc != 2)
37         Usage(argv[0]);
38     thread_count = strtol(argv[1], NULL, 10);
39     if (thread_count <= 0 || thread_count > MAX_THREADS)
40         Usage(argv[0]);
41
42     thread_handles = (pthread_t*) malloc(thread_count * sizeof(pthread_t));
43     messages = (char**) malloc(thread_count * sizeof(char*));
44     for (thread = 0; thread < thread_count; thread++)
45         messages[thread] = NULL;
46
47     for (thread = 0; thread < thread_count; thread++)
48         pthread_create(&thread_handles[thread], (pthread_attr_t*) NULL,
49             Send_msg, (void*) thread);
```

```

50
51 for (thread = 0; thread < thread_count; thread++) {
52     pthread_join(thread_handles[thread], NULL);
53 }
54
55 for (thread = 0; thread < thread_count; thread++)
56     free(messages[thread]);
57 free(messages);
58
59 free(thread_handles);
60 return 0;
61 } /* main */
62
63
64 void Usage(char* prog_name) {
65
66     fprintf(stderr, "usage: %s <numero de hilos>\n", prog_name);
67     exit(0);
68 } /* Usage */

```

Imagen a



```

thread 3 > Hola del thread 3 al thread 2
german@german-Inspiron-7577:~/Downloads$ ./pro-c 20
Thread 0 > No hay mensaje de 19
Thread 1 > Hola del thread 1 al thread 0
Thread 2 > Hola del thread 2 al thread 1
Thread 3 > Hola del thread 3 al thread 2
Thread 4 > Hola del thread 4 al thread 3
Thread 5 > Hola del thread 5 al thread 4
Thread 6 > Hola del thread 6 al thread 5
Thread 7 > Hola del thread 7 al thread 6
Thread 8 > Hola del thread 8 al thread 7
Thread 9 > Hola del thread 9 al thread 8
Thread 10 > Hola del thread 10 al thread 9
Thread 11 > Hola del thread 11 al thread 10
Thread 12 > Hola del thread 12 al thread 11
Thread 13 > Hola del thread 13 al thread 12
Thread 14 > Hola del thread 14 al thread 13
Thread 15 > Hola del thread 15 al thread 14
Thread 16 > Hola del thread 16 al thread 15
Thread 17 > Hola del thread 17 al thread 16
Thread 18 > Hola del thread 18 al thread 17
Thread 19 > Hola del thread 19 al thread 18
german@german-Inspiron-7577:~/Downloads$

```

Imagen b


```
File Edit View Search Terminal Help
german@german-Inspiron-7577:~/Downloads$ ./pro-c 30
Thread 2 > No hay mensaje de 1
Thread 3 > Hola del thread 3 al thread 2
Thread 1 > No hay mensaje de 0
Thread 4 > Hola del thread 4 al thread 3
Thread 5 > Hola del thread 5 al thread 4
Thread 7 > No hay mensaje de 6
Thread 0 > No hay mensaje de 29
Thread 8 > Hola del thread 8 al thread 7
Thread 9 > Hola del thread 9 al thread 8
Thread 10 > Hola del thread 10 al thread 9
Thread 6 > Hola del thread 6 al thread 5
Thread 12 > No hay mensaje de 11
Thread 13 > Hola del thread 13 al thread 12
Thread 11 > Hola del thread 11 al thread 10
Thread 14 > Hola del thread 14 al thread 13
Thread 15 > Hola del thread 15 al thread 14
Thread 16 > Hola del thread 16 al thread 15
Thread 18 > No hay mensaje de 17
Thread 19 > Hola del thread 19 al thread 18
Thread 17 > Hola del thread 17 al thread 16
Thread 20 > Hola del thread 20 al thread 19
Thread 22 > No hay mensaje de 21
Thread 23 > Hola del thread 23 al thread 22
Thread 24 > Hola del thread 24 al thread 23
Thread 25 > Hola del thread 25 al thread 24
Thread 26 > Hola del thread 26 al thread 25
Thread 27 > Hola del thread 27 al thread 26
Thread 28 > Hola del thread 28 al thread 27
Thread 29 > Hola del thread 29 al thread 28
Thread 21 > Hola del thread 21 al thread 20
german@german-Inspiron-7577:~/Downloads$
```

Otro código :

El programa recibirá seis argumentos en la línea de comandos:

- ✓ El tamaño del buffer.
- ✓ La cantidad de rondas, vueltas, o veces que se debe llenar y vaciar el buffer.
- ✓ La duración mínima en milisegundos que el productor tarda en generar un producto.
- ✓ La duración máxima en milisegundos que el productor tarda en generar un producto.
- ✓ La duración mínima en milisegundos que el consumidor tarda en consumir un producto.
- ✓ La duración máxima en milisegundos que el consumidor tarda en consumir un producto.

El programa crea un buffer para almacenar números de precisión flotante, usando la notación r.i, donde r es el número de la ronda y i el número del producto en esa ronda, ambos iniciando en 1. Por ejemplo 3.08 es el octavo producto de la tercera ronda.

Generar un producto no es inmediato. El tiempo que el productor tarda depende de la naturaleza del contexto en que se aplique el problema del consumidor-productor. Para efectos de la simulación, los argumentos 3 y 4 sirven de rango para generar números pseudoaleatorios. Cada vez que el productor tiene que construir un producto, se generará una duración pseudoaleatoria y el proceso esperará esta cantidad de milisegundos, simulando hasta que el producto esté acabado. Luego el productor agrega el producto

al buffer e imprime en la salida estándar el texto r.i generated.

Consumir un producto no es inmediato. El tiempo que el consumidor tarda se genera también en forma pseudoaleatoria con los argumentos 5 y 6. Una vez que el consumidor elimina un producto del buffer imprime en la salida estándar el texto r.i consumed. Sugerencia: indente la salida del consumidor para distinguirla de la salida del productor.

En el siguiente ejemplo de ejecución, los entes hacen dos rondas llenando y consumiendo un buffer de 3 elementos. El productor es rápido y tarda máximo 100ms creando un producto. El consumidor es más lento y podría tardar máximo 750ms consumiendo un producto. Como se puede ver en la salida, el productor rápidamente llena el buffer y debe esperar a que el consumidor libere espacio en el buffer para continuar produciendo. En la salida se comprueba que el consumidor procesa todos los datos generados por el productor y en el mismo orden.

```
1 #include <pthread.h>
2 #include <semaphore.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <time.h>
6 #include <unistd.h>
7
8 typedef struct
9 {
10     size_t buffer_size;
11     double* buffer;
12     size_t rounds;
13     useconds_t min_producer_delay;
14     useconds_t max_producer_delay;
15     useconds_t min_consumer_delay;
16     useconds_t max_consumer_delay;
17     sem_t producer_semaphore;
18     sem_t consumer_semaphore;
19     pthread_mutex_t stdout_mutex;
20 } shared_data_t;
21
22 int analyze_arguments(int argc, char* argv[], shared_data_t* shared_data)
23 ;
24 int create_threads(shared_data_t* shared_data);
25 void* produce(void* data);
26 void* consume(void* data);
27 void random_sleep(useconds_t min_milliseconds, useconds_t
28 max_milliseconds);
29
30 int main(int argc, char* argv[])
31 {
```

```

31 | srand( time(NULL) );
32 | shared_data_t* shared_data = (shared_data_t*) calloc(1, sizeof(
    |     shared_data_t));
33 | if ( shared_data == NULL )
34 |     return (void)fprintf(stderr, "error: could not allocate shared memory
    |     \n"), 1;
35 |
36 | int error = analyze_arguments(argc, argv, shared_data);
37 | if ( error == 0 )
38 | {
39 |     shared_data->buffer = (double*) calloc(shared_data->buffer_size,
    |     sizeof(double));
40 |     if ( shared_data->buffer )
41 |     {
42 |         sem_init(&shared_data->producer_semaphore, 0 /*pshared*/,
    |         shared_data->buffer_size);
43 |         sem_init(&shared_data->consumer_semaphore, 0 /*pshared*/, 0);
44 |         pthread_mutex_init(&shared_data->stdout_mutex, /*attr*/ NULL);
45 |
46 |         struct timespec start_time;
47 |         clock_gettime(CLOCK_MONOTONIC, &start_time);
48 |
49 |         error = create_threads(shared_data);
50 |         if ( error == 0 )
51 |         {
52 |             struct timespec finish_time;
53 |             clock_gettime(CLOCK_MONOTONIC, &finish_time);
54 |
55 |             double elapsed_seconds = finish_time.tv_sec - start_time.tv_sec
    |             + 1e-9 * (finish_time.tv_nsec - start_time.tv_nsec);
56 |
57 |             printf("Simulation time %.9lfs\n", elapsed_seconds);
58 |         }
59 |
60 |         pthread_mutex_destroy(&shared_data->stdout_mutex);
61 |         free(shared_data->buffer);
62 |     }
63 |     else
64 |     {
65 |         fprintf(stderr, "error: could not allocate memory for %zu products\
    |         n", shared_data->buffer_size);
66 |         error = 2;
67 |     }
68 | }
69 |
70 | free(shared_data);
71 | return error;
72 | }
73 |
74 |
75 | int analyze_arguments(int argc, char* argv[], shared_data_t* shared_data)
76 | {
77 |     if ( argc != 7 )

```

```

78 {
79     fprintf(stderr, "usage: producer-consumer buffer-size rounds"
80         " min_producer_delay max_producer_delay"
81         " min_consumer_delay max_consumer_delay\n");
82     return 1;
83 }
84
85 shared_data->buffer_size = strtoull(argv[1], NULL, 10);
86 if ( shared_data->buffer_size == 0 )
87     return 2;
88
89 if ( sscanf(argv[2], "%u", &shared_data->rounds) != 1 || shared_data->
90     rounds == 0 )
91     return (void)fprintf(stderr, "invalid rounds: %s\n", argv[2]), 2;
92
93 if ( sscanf(argv[3], "%u", &shared_data->min_producer_delay) != 1 )
94     return (void)fprintf(stderr, "invalid min producer delay: %s\n", argv
95 [3]), 3;
96
97 if ( sscanf(argv[4], "%u", &shared_data->max_producer_delay) != 1
98     || shared_data->max_producer_delay < shared_data->min_producer_delay
99 )
100     return (void)fprintf(stderr, "invalid max producer delay: %s\n", argv
101 [4]), 4;
102
103 if ( sscanf(argv[5], "%u", &shared_data->min_consumer_delay) != 1 )
104     return (void)fprintf(stderr, "invalid min consumer delay: %s\n", argv
105 [5]), 5;
106
107 if ( sscanf(argv[6], "%u", &shared_data->max_consumer_delay) != 1
108     || shared_data->max_consumer_delay < shared_data->min_consumer_delay
109 )
110     return (void)fprintf(stderr, "invalid max consumer delay: %s\n", argv
111 [6]), 6;
112
113 return EXIT_SUCCESS;
114 }
115
116 int create_threads(shared_data_t* shared_data)
117 {
118     pthread_t producer_thread;
119     pthread_t consumer_thread;
120
121     pthread_create(&producer_thread, NULL, produce, shared_data);
122     pthread_create(&consumer_thread, NULL, consume, shared_data);
123
124     pthread_join(producer_thread, NULL);
125     pthread_join(consumer_thread, NULL);
126
127     return 0;
128 }

```

```

123 void* produce(void* data)
124 {
125     shared_data_t* shared_data = (shared_data_t*)data;
126
127     for ( size_t round = 1; round <= shared_data->rounds; ++round )
128     {
129         for ( size_t index = 0; index < shared_data->buffer_size; ++index )
130         {
131             sem_wait(&shared_data->producer_semaphore);
132
133             random_sleep(shared_data->min_producer_delay, shared_data->
max_producer_delay);
134
135             shared_data->buffer[index] = round + (index + 1) / 100.0;
136
137             pthread_mutex_lock(&shared_data->stdout_mutex);
138             printf("Produced %.2lf\n", shared_data->buffer[index]);
139             pthread_mutex_unlock(&shared_data->stdout_mutex);
140
141             sem_post(&shared_data->consumer_semaphore);
142         }
143     }
144
145     return NULL;
146 }
147
148 void* consume(void* data)
149 {
150     shared_data_t* shared_data = (shared_data_t*)data;
151
152     for ( size_t round = 1; round <= shared_data->rounds; ++round )
153     {
154         for ( size_t index = 0; index < shared_data->buffer_size; ++index )
155         {
156             sem_wait(&shared_data->consumer_semaphore);
157
158             random_sleep(shared_data->min_consumer_delay, shared_data->
max_consumer_delay);
159
160             pthread_mutex_lock(&shared_data->stdout_mutex);
161             printf("\t\t\tConsumed %.2lf\n", shared_data->buffer[index]);
162             pthread_mutex_unlock(&shared_data->stdout_mutex);
163
164             sem_post(&shared_data->producer_semaphore);
165         }
166     }
167
168     return NULL;
169 }
170
171 void random_sleep(useconds_t min_milliseconds, useconds_t
max_milliseconds)

```

```

172 {
173     useconds_t duration = min_milliseconds;
174     useconds_t range = max_milliseconds - min_milliseconds;
175     if ( range > 0 )
176         duration += rand() % range;
177     usleep( 1000 * duration );
178 }

```

2.2. Resultados :

Imagen c

```

seconds
No de threads : 312 , tiempo tomado es 0.024207 s
german@german-Inspiron-7577:~/Downloads$ mpicc producer-consumer.c -o procon
german@german-Inspiron-7577:~/Downloads$ ./procon 3 2 0 100 0 750
Produced 1.01
Produced 1.02
Produced 1.03
Consumed 1.01
Consumed 1.02
Consumed 1.03
Produced 2.01
Produced 2.02
Produced 2.03
Consumed 2.01
Consumed 2.02
Consumed 2.03
Simulation time 3.047551932s
german@german-Inspiron-7577:~/Downloads$ ./procon 3 2 0 100 0 750
Produced 1.01
Produced 1.02
Produced 1.03
Consumed 1.01
Consumed 1.02
Consumed 1.03
Produced 2.01
Produced 2.02
Produced 2.03
Consumed 2.01
Consumed 2.02
Consumed 2.03
Simulation time 1.963721989s
german@german-Inspiron-7577:~/Downloads$

```

2.3. Analisis :

- ✓ En la imagen a y b tenemos la primera columna como consumidores y los de la ultima columna de productores .
- ✓ Observamos que los consumidores responden con un Hola cuando obtienen el producto.
- ✓ En la imagen c tenemos otro ejemplo, en este caso se muestra que varios procesos muestran sus productos y los consumidores van llegando y obteniendolos en diferente tiempo.
- ✓ Con el metodo Mutex este proceso no se podria realizar , ya que el productor 1 necesariamente esperaria a liberarse de su producto. para poder pasar al otro productor.

- ✓ Como ya vimos anteriormente Mutex aparenta trabajar bien , pero si usamos en este caso , tendríamos: el productor 1 tiene su producto este para volver a producir o poder usar otro proceso tendria que esperar que el consumidor "x" adquiera el producto del productor 1.
- ✓ Mutex es mas "*Rapido*" al realizar cada proceso por separado , pero al momento de realizar intercambios este genera retrasosza que si lo hacemos de forma creciente , de 20 procesos , el proceso 20 tiene que esperar que se realizen los 19 procesos primero para luego poder realizar o entrar en funcionamiento.
- ✓

3. Implementar y explicar las diferentes formas de barreras en PThreads mostradas en el libro.

3.1. Mutex y busy waiting

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <stdlib.h>
4
5
6 long double sum;
7 long flag;
8 int thread_count;
9 int n=0;
10 int counter = 0;
11
12 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
13
14 void * thread_sum(void * rank){
15     long my_rank = (long) rank;
16     double factor;
17     long long my_n = n / thread_count;
18     long long my_first_i = my_n * my_rank;
19     long long my_last_i = my_first_i + my_n;
20     double my_sum = 0.0;
21
22     if(my_first_i % 2 == 0){
23         factor = 1.0;
24     }
25     else{
26         factor = -1.0;
27     }
28
29     for(long long i = my_first_i; i < my_last_i; i++, factor = -factor){
30         my_sum += factor / (2 * i + 1);
31     }

```

```

32     /* barrera*/
33     pthread_mutex_lock(&mutex);
34     counter++;
35     sum += my_sum;
36     pthread_mutex_unlock(&mutex);
37     while(counter<thread_count);
38
39     return NULL;
40 }

```

```

512 ,Tiempo Tomado es 0.021840 seconds
german@german-Inspiron-7577:~/Downloads$ mpicc mutex.c -o mutex
german@german-Inspiron-7577:~/Downloads$ ./mutex
*No de Threads : 1 ,Tiempo Tomado es 0.000311 seconds
**No de Threads : 2 ,Tiempo Tomado es 0.000375 seconds
***No de Threads : 4 ,Tiempo Tomado es 0.000542 seconds
*****No de Threads : 8 ,Tiempo Tomado es 0.000896 seconds
*****No de Threads : 16 ,Tiempo Tomado es 0.001346 seconds
*****No de Threads : 32 ,Tiempo Tomado es 0.003688 seconds
*****No de Threads : 64 ,Tiempo Tomado es 0.005253 seconds
*****No de Threads : 128 ,Tiempo Tomado es 0.011407 seconds
*****No de Threads : 256 ,Tiempo Tomado es 0.022141 seconds
*****No de Threads : 512 ,Tiempo Tomado es 0.024207 seconds
german@german-Inspiron-7577:~/Downloads$

```

3.2. Semaforos :

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <stdlib.h>
4 #include <semaphore.h>
5
6 long double sum;
7 long flag;
8 int thread_count;
9 int n=0;
10
11 int counter = 0;
12 sem_t count_sem;
13 sem_t barrier_sem;
14
15 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
16
17 void * thread_sum(void * rank){
18     long my_rank = (long) rank;

```



```

19 double factor;
20 long long my_n = n / thread_count;
21 long long my_first_i = my_n * my_rank;
22 long long my_last_i = my_first_i + my_n;
23 double my_sum = 0.0;
24
25 if(my_first_i % 2 == 0){
26     factor = 1.0;
27 }
28 else{
29     factor = -1.0;
30 }
31
32 for(long long i = my_first_i; i < my_last_i; i++, factor = -factor){
33     my_sum += factor/(2 * i + 1);
34 }
35
36 pthread_mutex_lock(&mutex);
37 sum += my_sum;
38 pthread_mutex_unlock(&mutex);
39 /* barrera*/
40 sem_wait(&count_sem);
41 if(counter == thread_count-1){
42     counter =0;
43     sem_post(&count_sem);
44     for(int j=0 ;j<thread_count-1;j++)
45         sem_post(&barrier_sem);
46 }else{
47     counter++;
48     sem_post(&count_sem);
49     sem_wait(&barrier_sem);
50 }
51
52 return NULL;
53 }

```

3.3. Variables de condicion :

```

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <stdlib.h>
4
5
6 long double sum;
7 long flag;
8 int thread_count;
9 int n=0;
10
11 int counter = 0;

```

```

12 pthread_cond_t cond_var;
13
14 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
15
16 void * thread_sum(void * rank){
17     long my_rank = (long) rank;
18     double factor;
19     long long my_n = n / thread_count;
20     long long my_first_i = my_n * my_rank;
21     long long my_last_i = my_first_i + my_n;
22     double my_sum = 0.0;
23
24     if(my_first_i % 2 == 0){
25         factor = 1.0;
26     }
27     else{
28         factor = -1.0;
29     }
30
31     for(long long i = my_first_i; i < my_last_i; i++, factor = -factor){
32         my_sum += factor/(2 * i + 1);
33     }
34     /*Barrier */
35     int cont=0;
36     pthread_mutex_lock(&mutex);
37     counter++;
38     sum += my_sum;
39     if(counter == thread_count){
40         counter =0;
41         pthread_cond_broadcast(&cond_var);
42         printf(">");
43     }else{
44         while(pthread_cond_wait(&cond_var,&mutex)!=0);
45         cont++;
46         printf("*");
47     }
48     pthread_mutex_unlock(&mutex);
49     //printf("2 una vez: %a",cont);
50
51     return NULL;
52 }

```

```
german@german-Inspiron-7577:~/Downloads$ mpicc mutexVariable.c -o mutexV
german@german-Inspiron-7577:~/Downloads$ ./mutexV
>No de Threads : 1 ,Tiempo Tomado es 0.000359 seconds
>*No de Threads : 2 ,Tiempo Tomado es 0.000435 seconds
>***No de Threads : 4 ,Tiempo Tomado es 0.000750 seconds
>*****No de Threads : 8 ,Tiempo Tomado es 0.001767 seconds
>*****No de Threads : 16 ,Tiempo Tomado es 0.003129 seconds
>*****No de Threads : 32 ,Tiempo Tomado es 0.005831 seconds
>*****No de Threads : 64 ,Tiempo Tomado es 0.008628 seconds
>*****No de Threads : 128 ,Tiempo Tomado es 0.010535 seconds
>*****No de Threads : 256 ,Tiempo Tomado es 0.009941 seconds
>*****No de Threads : 512 ,Tiempo Tomado es 0.020864 seconds
german@german-Inspiron-7577:~/Downloads$
```

3.4. Analisis

- ✓ En todos los casos es distinto la forma en que se utiliza la Barrera: Aunque todos buscan cumplir el mismo objetivo .
- ✓ En el caso de la barrera para Mutex y Busy-Waiting .se recorre la cantidad de procesos que se ingresaron o se estan utilizando para luego detener la funcion al momento de exceder la cantidad de procesos.
- ✓ En el caso de barrera por variables :
Se muestra en la imagen el mismo proceso o trabajo que se realizo en el caso de barrera mutex ,es decir trabajan todos los "procesos" y estos se detienen con la barrera o simplemente con el contador.

4. Referencias :

- ✓ <https://stackoverflow.com/questions/34524/what-is-a-mutex>
- ✓ <http://jeisson.ecci.ucr.ac.cr/concurrente/2019b/ejemplos/>

5. GitHub :

<https://github.com/Garcia05/Paralelos/tree/master/lab-Phtreads1>