Sistemas de Tiempo Real
Departamento de Ingeniería Electrónica
Universidad de Antioquia
2023-2

Práctica No. 4
Implementación de Tareas Periódicas de Tiempo-Real

Realizado por : Yorguin Jose Mantilla Ramos

1. Hilos Periódicos en POSIX-RT:

1.1.1. Analice cuidadosamente el siguiente ejemplo compartido en el *Microsoft Teams* del curso, referente a la implementación de dos hilos periódicos en **POSIX-RT**, disponible en:

https://udeaeduco.sharepoint.com/:u:/r/sites/SistemasdeTiempoReal2023.2/Class%20Materials/LAB/PR4/periodic_threads.tar.gz?csf=1&web=1&e=2A3kUS

1.1.2. Explique el funcionamiento de este código:

1. ¿Cuáles son las *System Call* usadas en todo el programa?

   En el código principal Sleep es el system call mas explicito (usa a nanosleep). Sin embargo, hay funciones que aunque no son system calls propiamente, podrían involucrar su uso.

   Sin embargo, en los demás códigos se pueden encontrar otros system calls, particularmente en periodic_settings.c, donde se llama a clock_nanosleep, clock_gettime, y gettimeofday.

2. ¿Cómo se define la periodicidad de cada hilo, cómo se crea cada hilo y qué hace cada uno de ellos?

   Periodicidad: Mediante las líneas temp->period = XXXXX , dado en microsegundos. Además, se incluye un offset mediante temp->offset = YYYY (también en microsegundos).

   Creación: Mediante las líneas pthread_create(&idX, NULL, threadX, (void*) ptX) .
   &idX: referencia al ID del hilo
   NULL: opciones de creación del hilo, que en este caso usa el default cuando es NULL
   threadX: función que ejecuta el thread
   ptX: los argumentos entrante a la función que ejecuta el thread, que en el caso del

código es un puntero a una estructura que representa la información de la tarea (que es un temporizador periodico).

Función de cada hilo:
Básicamente inician timers periódicos que llaman a una tarea cada que vez que se "activan". Dicha tarea involucra la impresión de la cuenta hasta el momento actual, por lo que se utilizan mutex para proteger el recurso compartido de printf.

3. ¿Cuál es el rol del ***pthread_mutex_t*** definido en el código?

   Justo dicho arriba, su rol es el de definir una sección critica , y en este caso lo hace para las secciones que usan el printf (que es un recurso compartido entre threads).

4. ¿Cuál es el rol del atributo ***struct timespec r*** definido al interior de la estructura ***struct periodic_thread***?

   Este atributo directamente controla el tiempo/fecha unix/momento hasta que se suspende del hilo de forma, esto se hace de forma absoluta. Por lo que para utilizarlo como timer, lo que se hace es que continuamente se actualiza con el tiempo actual mas el periodo, pero habiendo inicializado este con clock_gettime.

5. En general, explique el funcionamiento detallado del programa descrito en los archivos fuente compartidos en la carpeta **PERIODIC_THREADS** del enlace anterior.

   El programa tiene como objetivo crear y gestionar dos hilos periódicos, uno con un período de 100ms y otro con un período de 150ms. Cada hilo simplemente incrementa un contador y muestra su valor utilizando printf. Además, están desfasados entre ellos.

   Para este propósito la estrategia es utilizar la función clock_nanosleep para suspender los hilos de forma absoluta, es decir, pasandole el momento hasta que el hilo despierta (e imprimir/ hacer la tarea que debe repetir).

   Lo anterior involucra funciones auxiliares como:
   timespec_add_us: Maneja la adición de tiempos entre aquellos manejados en nanos y los definidos en micros. Es de notar que la estructura de tiempo maneja simultáneamente variables que guardan este en nanos y en segundos. Este código maneja adecuadamente ambos (solo suma tantos segundos como segundos completos han pasado). Esto es importante porque la estructura timespec esta "normalizada", es decir, la cuenta en tv_nsec no debe exceder un segundo, y todos los segundos deben

estar contabilizados en tv_sec. Esta normalización posiblemente sea para evitar el overflow generado si se utilizara una sola para representar todo.

wait_next_activation: Que simplemente suspende el hilo hasta el tiempo indicado (activación) y posteriormente actualiza el siguiente tiempo al que se debe esperar.

start_periodic_timer: Inicializa los timers, es decir, las estructuras de datos involucradas en el sistema y provee mensajes de inicialización al usuario.

Las funciones thread_X simplemente son las tareas que realmente se ejecutan al activarse los ciclos del timer. Que en este caso solo imprimen (de forma protegida).

Ahora, las funciones complementarias *thread_X ya proveen el encapsulamiento del proceso del timer en un bucle infinito (además de menores mensajes al usuario de forma protegida).

Y la función main simplemente orquesta todo el proceso, separando espacio para las estructuras de datos involucras, proveyendo mensajes al usuario e implementa un tiempo máximo donde main (el thread principal) termina a los 10 segundos.

2. Desarrollo de un programa propio:

Siguiendo su aprendizaje del ejemplo analizado en el ítem anterior, y de los diversos ejemplos sobre temporizadores y tareas periódicas estudiados en clase[1], implemente un Sistema Productor-Consumidor Multihilo basado en **POSIX-RT,** de acuerdo con una de las dos versiones de este sistema asignadas en la siguiente tabla:

Apellidos	Versión Asignada de Productor-Consumidor
Y. Mantilla	**Versión 2**

2.1. Versión 2 – Sistema multihilo periódico mediante relojes de Tiempo-Real:

Un sistema Productor-Consumidor multihilo escrito con relojes (*Clocks*) de Tiempo-Real, compuesto por un módulo (hilo) de producción y almacenamiento de datos enteros aleatorios, y tres módulos (hilos) de consumo de dichos datos (ver **Figura 2**), de modo tal que los cuatro módulos descritos se ejecuten periódicamente con periodos asignados por usted en escala de ***μs*** (entre 0 y 999999 ***μs***).

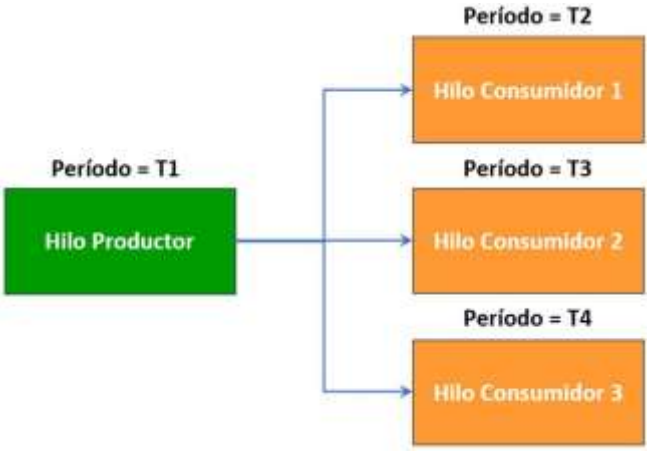


Figura 2: Representación gráfica de la **Versión 2** del sistema periódico Productor-Consumidor

El uso de relojes de tiempo-real implica entonces el uso de tiempos absolutos en la activación periódica de cada módulo, de modo que no se requiere el uso de señales ni de eventos de notificación.

Dado lo anterior, explique:

2.1.1. Su programa desarrollado para la producción y consumo de los datos aleatorios compartidos entre los hilos, considerando:

- La identificación explícita en su código fuente del hilo productor de datos, y los tres hilos consumidores.

  Los hilos se definen y configuran en las siguientes líneas, estos quedan guardados en un vector de hilos. Nótese que el primero es el productor y los siguientes los consumidores. Las variables T y OFF corresponden a los periodos y offsets asignados. Además cada thread tiene un contador que aloja cuantas veces se ha llamado su task (iniciado en 0), una estructura timespec (vacía {}), y un ID (el ultimo argumento).

```
    // Lists to store the configuration of each thread and their IDs.
    std::vector<PeriodicThread*> threadConfigs;
    std::vector<pthread_t> threadIDs;

    // Configurations for the producer and consumer threads.
    threadConfigs.push_back(new PeriodicThread{T1, OFF1, 0, {}, produce,1});
    threadConfigs.push_back(new PeriodicThread{T2, OFF2, 0, {}, consume,2});
    threadConfigs.push_back(new PeriodicThread{T3, OFF3, 0, {}, consume,3});
    threadConfigs.push_back(new PeriodicThread{T4, OFF4, 0, {}, consume,4});
```

- Igualmente, su implementación de la pila o matriz de almacenamiento de los datos producidos y consumidos.
  La interacción entre productor y consumidor se hace mediante un buffer implementado mediante una pila.

```
std::queue<int> buffer;
const int BUFFER_MAX_SIZE = 10;  // Set a limit for the buffer
```

  Además cada thread (sea productor o consumidor) posee una pila de historia propia de cada ellos (no compartida) donde se guardan los valores exitosamente producidos o consumidos.

```
    std::queue<std::optional<int>> history;
};
```

- Muestre su estrategia de implementación de la sincronización requerida en la producción y consumo de los datos de dicha pila.

  Realmente la sincronización se hace mediante la activación en tiempos no solapados (y periódicos) de las tareas de producción y consumo realizadas por cada respectivo thread. Esto se realiza mediante la función waitNextActivation(), que espera hasta la próxima activación basada en tiempos absolutos. No se usan eventos ni señales para activar los threads.

```
void waitNextActivation(PeriodicThread *t) //<--- Updated here
{
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &t->next_activation, nullptr);
    timespecAddUs(&t->next_activation, t->period);
}
```

  Para evitar redundancia se definió una clase de hilo periódico genérico, que simplemente inicializa el timer del thread y luego de forma cíclica ejecuta la tarea cuando se dan los periodos de activación.

```
static std::mutex mtx;  // Mutex used to synchronize threads during print operations.

// Generic function to represent the behavior of the threads.
void* genericThread(void* p_d) {
    // Cast the passed pointer to its original type.
    PeriodicThread* threadData = static_cast<PeriodicThread*>(p_d);

    mtx.lock();  // Lock to ensure serial access to std::cout.
    std::cout << "\nThread with period : " << threadData->period / 1000 << " ms.\n";
    startPeriodicTimer(threadData);  // Start the timer for this thread.
    mtx.unlock();

    // Infinite loop simulating the periodic tasks.
    while (true) {
        waitNextActivation(threadData);  // Wait for the next scheduled execution.
        std::optional<int> num = threadData->taskFunction(&(threadData->count), threadData->id, mtx);
        if (num) {
            threadData->history.push(*num);  // If there's a value, push it to the history.
        }
    }
    return nullptr;
}
```

  Nótese que a cada thread se le pasa el un mismo mutex creado en el script principal. Esto ya que ambos comparten el stream de impresión.

- Indique las instrucciones de sincronización y comunicación necesarias para la ejecución de cada hilo, mediante el uso de **MUTEX**.

  Para ambos el acceso a el buffer (pila común) y el stream de impresión están protegidos por

un mutex.

Productor

```
lock.lock();

std::optional<int> random_value = dist(rng);

if (buffer.size() < BUFFER_MAX_SIZE) {
    buffer.push(random_value.value());
} else {
    std::cout << "Buffer is full, dropping value: " << random_value.value() << std::endl;
    random_value = std::nullopt;
}

if (random_value.has_value()) {
    std::cout << "Thread " << id << " random value: " << random_value.value() << std::endl
}
lock.unlock();
```

Consumidor

```
lock.lock();
std::optional<int> value;
if (!buffer.empty()) {
    value = buffer.front();
    buffer.pop();
} else {
    value = std::nullopt;
}
if (value.has_value()) {
    std::cout << "Thread " << id << " consumed value: " << value.value() << std::endl;
} else {
    std::cout << "Thread " << id << " did not consume a value." << std::endl;
}
std::cout << "Thread " << id << " counting: " << *c << std::endl;
lock.unlock();
```

2.1.2. La definición y configuración de las *System Call* requeridas para la ejecución periódica de sus cuatro hilos, teniendo en cuenta la definición de sus ***offset*** y periodos respectivos.

- En su código fuente identifique explícitamente los cuatro periodos asignados. ¿Qué valor asignó usted a cada periodo **T1**, **T2**, **T3** y **T4**?

  Explore la siguiente asignación.

```
constexpr int T1 = 100000; // Timing values in microseconds.
constexpr int T2 = 200000;
constexpr int T3 = 250000;
constexpr int T4 = 150000;

constexpr int OFF1 = 100000; // Offset values in microseconds.
constexpr int OFF2 = 150000;
constexpr int OFF3 = 160000;
constexpr int OFF4 = 170000;
```

Sin embargo, en la parte de análisis y conclusiones muestro una forma más balanceada que minimiza las perdidas en el consumo y producción de datos. Esta corresponde a:

```
// Constants to specify the runtime and the periodic timings for the threads.
constexpr int SECONDS_TO_RUN = 10;
constexpr int T1 = 33333; // Timing values in microseconds.
constexpr int T2 = 100000;
constexpr int T3 = 100000;
constexpr int T4 = 100000;

constexpr int OFF1 = 100000; // Offset values in microseconds.
constexpr int OFF2 = 133333;
constexpr int OFF3 = 166666;
constexpr int OFF4 = 199999;
```

- Igualmente, muestre en su código el uso de las *System Call* de temporizadores y señales requeridas.

  Para inicializar los timers se obtiene el tiempo actual mediante el system call clock_gettime

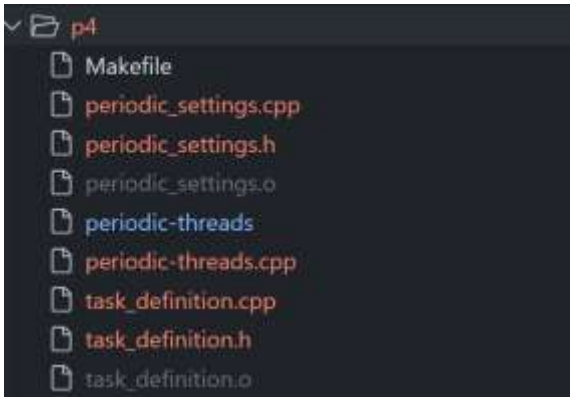
```
    clock_gettime(CLOCK_REALTIME, &perThread->next_activation);
    timespecAddUs(&perThread->next_activation, perThread->offset);
```

  Para suspender los threads (hasta su nueva activación) se usa clock_nanosleep

```
void waitNextActivation(PeriodicThread *t) // <-- Updated here
{
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &t->next_activation, nullptr);
    timespecAddUs(&t->next_activation, t->period);
}
```

2.1.3. Divida su sistema en diversos archivos ***.c** y ***.h** requeridos para la creación del ejecutable de su programa, y documente su código fuente con comentarios claros y explicativos.

Se uso la división modular que separa el programa en tareas (task_definition.cpp) , la periodicidad (periodic_settings.cpp) y el programa principal: periodic-threads.cpp



2.1.4. Escriba un *Makefile* que permita compilar fácilmente los anteriores archivos requeridos en su programa.

Se utilizo el mismo make file dado en clase, solo cambiando al uso de g++ porque decidí utilizar una implementación basada en cpp.

```
CC = g++
CFLAGS = -Wall -g
LIBRT = -lrt
LIBPTHREAD = -lpthread

SETTINGS = periodic_settings
TASKS = task_definition
MAIN = periodic-threads


all: $(MAIN)

$(SETTINGS).o: $(SETTINGS).cpp $(SETTINGS).h
	$(CC) $(CFLAGS) -c $(SETTINGS).cpp $(LIBRT) $(LIBPTHREAD)

$(TASKS).o: $(TASKS).cpp $(TASKS).h
	$(CC) $(CFLAGS) -c $(TASKS).cpp $(LIBRT) $(LIBPTHREAD)

$(MAIN): $(MAIN).cpp $(SETTINGS).o $(TASKS).o
	$(CC) $(CFLAGS) -o $@ $^ $(LIBRT) $(LIBPTHREAD)



clean:
	rm -f *.o
	rm -f $(MAIN)
```

2.1.5. Compile este programa y ejecútelo tantas veces como sea necesaria, a fin de observar el buen funcionamiento de este sistema periódico Productor-Consumidor multihilo.

- Presente pantallazos de la ejecución de su código evidenciando el funcionamiento de su sistema, así como los tiempos de ***offset*** y periodos creados para cada hilo.

  Corri 4 veces el código. Cada corrida tiene también una historia secuencial de lo que esta pasando en cada thread. Además, como implemente una historia para cada thread en cada corrida se puede ver los números producidos por el productor y como se distribuyeron a través de los consumidores:

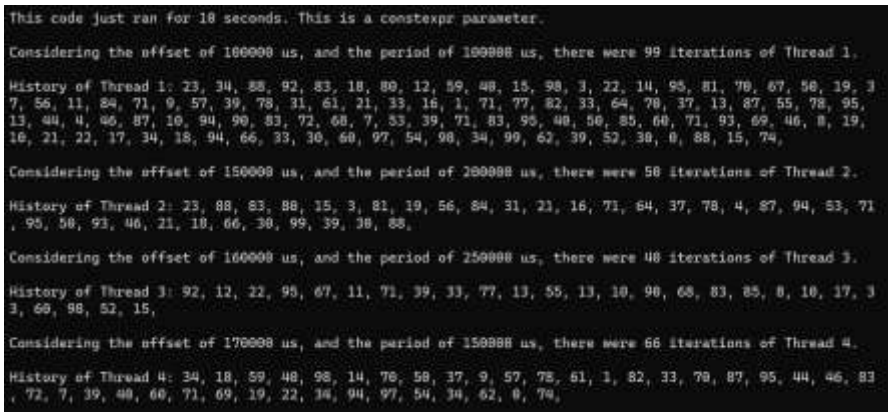
```
Thread 1 random value: 62
Thread 1 random value: 39
Thread 4 consumed value: 62
Thread 4 counting: 61
Thread 2 consumed value: 39
Thread 2 counting: 46
Thread 1 random value: 52
Thread 3 consumed value: 52
Thread 3 counting: 37
Thread 4 did not consume a value.
Thread 4 counting: 62
Thread 1 random value: 30
Thread 2 consumed value: 30
Thread 2 counting: 47
Thread 1 random value: 0
Thread 4 consumed value: 0
Thread 4 counting: 63
Thread 3 did not consume a value.
Thread 3 counting: 38
Thread 1 random value: 88
```

```
This code just ran for 10 seconds. This is a constexpr parameter.

Considering the offset of 100000 us, and the period of 100000 us, there were 99 iterations of Thread 1.

History of Thread 1: 23, 34, 88, 92, 83, 10, 80, 12, 59, 40, 15, 98, 3, 22, 14, 95, 81, 70, 67, 50, 19, 3
7, 56, 11, 84, 71, 9, 57, 39, 78, 31, 61, 21, 33, 16, 1, 71, 77, 82, 33, 64, 70, 37, 13, 87, 55, 78, 95,
13, 44, 4, 46, 87, 10, 94, 90, 83, 72, 60, 7, 53, 39, 71, 83, 95, 40, 50, 85, 60, 71, 93, 69, 46, 8, 19,
16, 21, 22, 17, 34, 18, 94, 66, 33, 30, 60, 97, 54, 98, 34, 99, 62, 39, 52, 30, 0, 88, 15, 74,

Considering the offset of 150000 us, and the period of 200000 us, there were 50 iterations of Thread 2.

History of Thread 2: 23, 88, 83, 80, 15, 3, 81, 19, 56, 84, 31, 21, 16, 71, 64, 37, 78, 4, 87, 94, 53, 71
, 95, 50, 93, 46, 21, 10, 66, 30, 99, 39, 30, 88,

Considering the offset of 160000 us, and the period of 250000 us, there were 40 iterations of Thread 3.

History of Thread 3: 92, 12, 22, 95, 67, 11, 71, 39, 33, 77, 13, 55, 13, 10, 90, 68, 83, 85, 8, 10, 17, 3
3, 60, 98, 52, 15,

Considering the offset of 170000 us, and the period of 150000 us, there were 66 iterations of Thread 4.

History of Thread 4: 34, 18, 59, 48, 98, 14, 70, 58, 37, 9, 57, 78, 61, 1, 82, 33, 70, 87, 95, 44, 46, 83
, 72, 7, 39, 40, 60, 71, 69, 19, 22, 34, 94, 97, 54, 34, 62, 0, 74,
```

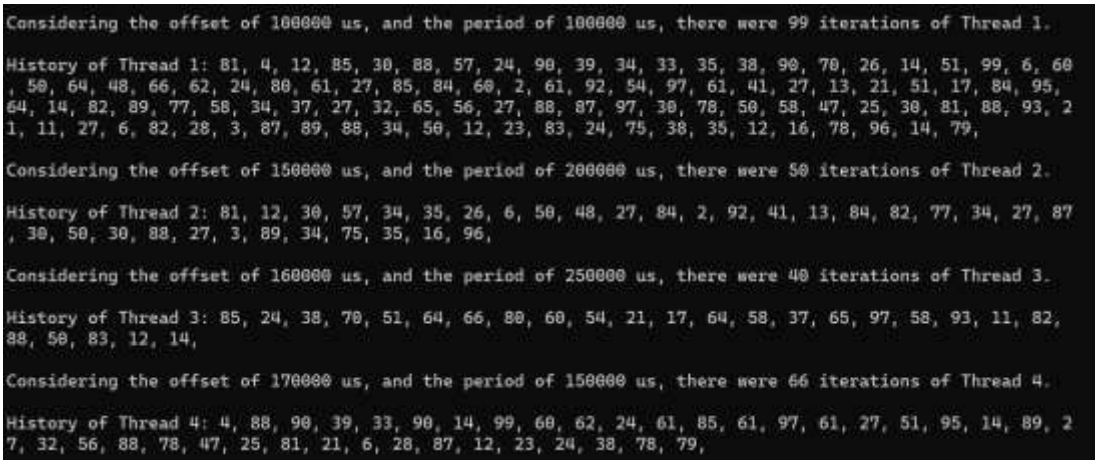
- Demuestre mediante gráficos o tablas la correcta ejecución sincrónica de su modelo de productor-consumidor, indicando los datos producidos y datos consumidos.
  En cada corrida, la historia del thread 1 son los datos producidos, y la de los demás threads los datos consumidos por cada uno de los threads consumidores. Como vemos

el consumo es intercalado.

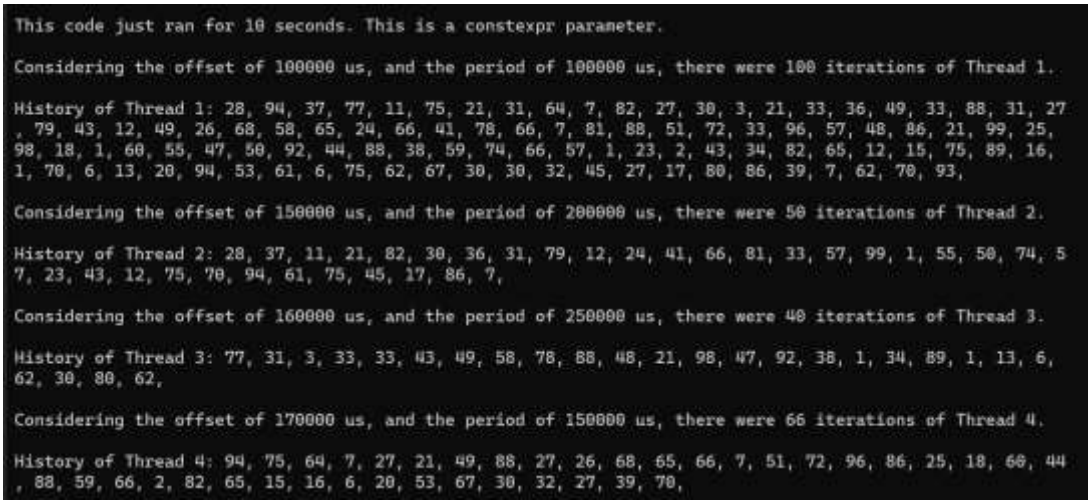
Corrida 1



Corrida 2



Corrida 3



Corrida 4

```
This code just ran for 10 seconds. This is a constexpr parameter.

Considering the offset of 100000 us, and the period of 100000 us, there were 100 iterations of Thread 1.

History of Thread 1: 55, 66, 79, 17, 35, 58, 78, 97, 37, 8, 37, 37, 3, 4, 25, 18, 1, 29, 11, 61, 60, 24,
90, 62, 23, 40, 46, 20, 49, 21, 46, 76, 61, 21, 36, 45, 25, 41, 12, 64, 34, 13, 64, 7, 65, 67, 92, 70, 77
, 2, 45, 83, 74, 90, 5, 35, 13, 59, 37, 40, 87, 22, 84, 57, 11, 90, 90, 23, 6, 79, 74, 56, 66, 38, 70, 19
, 82, 76, 54, 27, 69, 64, 54, 69, 73, 17, 82, 25, 52, 57, 20, 76, 80, 87, 71, 41, 3, 44, 48, 58,

Considering the offset of 150000 us, and the period of 200000 us, there were 50 iterations of Thread 2.

History of Thread 2: 55, 79, 35, 78, 37, 3, 1, 60, 90, 23, 46, 61, 36, 25, 34, 64, 92, 45, 74, 5, 87, 84,
 11, 90, 74, 66, 82, 69, 54, 73, 20, 80, 71, 3,

Considering the offset of 160000 us, and the period of 250000 us, there were 40 iterations of Thread 3.

History of Thread 3: 17, 97, 4, 18, 11, 62, 40, 49, 21, 41, 7, 67, 77, 90, 35, 37, 57, 23, 38, 19, 54, 69
, 17, 52, 87, 44,

Considering the offset of 170000 us, and the period of 150000 us, there were 66 iterations of Thread 4.

History of Thread 4: 66, 58, 37, 8, 37, 25, 29, 61, 24, 46, 20, 21, 76, 45, 12, 64, 13, 65, 70, 2, 83, 13
, 59, 40, 22, 90, 6, 79, 56, 70, 76, 27, 64, 82, 25, 57, 76, 41, 48,
```

3. Conclusiones y Analisis

Para los valores escogidos de Periodos y Offsets el código parece funcionar bien. En particular el productor es más rápido que cualquiera de los 3 consumidores. Los 3 consumidores además tienen offsets y periodos distintos para evitar que tengan que esperar a que otro termine.

Vemos que en efecto cuando el consumo no es balanceado, los consumidores con periodos menores abarcan más consumo (T4 es el líder, y T3 es el más lento):

```
This code just ran for 10 seconds. This is a constexpr parameter.

Considering the offset of 100000 us, and the period of 100000 us, there were 100 iterations of Thread 1.

History of Thread 1: 69, 52, 71, 24, 59, 20, 24, 81, 75, 9, 49, 86, 87, 33, 2, 49, 74, 27, 30, 22, 16, 30
, 59, 51, 18, 37, 41, 78, 16, 98, 34, 80, 39, 55, 67, 43, 62, 71, 91, 63, 63, 6, 94, 82, 57, 79, 93, 83,
91, 51, 34, 80, 19, 31, 22, 54, 29, 42, 76, 32, 94, 46, 51, 50, 85, 8, 73, 43, 33, 10, 93, 21, 56, 58, 57
, 60, 82, 64, 24, 49, 88, 19, 32, 9, 67, 39, 45, 94, 77, 23, 33, 59, 91, 4, 32, 49, 95, 41, 13, 81,

Considering the offset of 150000 us, and the period of 200000 us, there were 50 iterations of Thread 2.

History of Thread 2: 69, 71, 59, 24, 49, 87, 74, 16, 59, 18, 34, 39, 67, 62, 63, 94, 93, 34, 19, 22, 94,
51, 85, 73, 93, 56, 82, 88, 32, 67, 33, 91, 32, 95,

Considering the offset of 160000 us, and the period of 250000 us, there were 40 iterations of Thread 3.

History of Thread 3: 24, 81, 33, 49, 30, 51, 37, 16, 55, 71, 82, 79, 91, 31, 54, 76, 50, 43, 58, 60, 24,
9, 39, 77, 4, 41,

Considering the offset of 170000 us, and the period of 150000 us, there were 66 iterations of Thread 4.

History of Thread 4: 52, 20, 75, 9, 86, 2, 27, 22, 30, 41, 78, 98, 80, 43, 91, 63, 6, 57, 83, 51, 80, 29,
 42, 32, 46, 8, 33, 10, 21, 57, 64, 49, 19, 45, 94, 23, 59, 49, 13,
```

Si el productor se hace mas lento, los no consumos se vuelven más frecuentes:

```
Thread 2 did not consume a value.
Thread 2 counting: 20
Thread 3 did not consume a value.
Thread 3 counting: 16
Thread 4 did not consume a value.
Thread 4 counting: 27
Thread 1 random value: 39
Thread 2 consumed value: 39
Thread 2 counting: 21
Thread 4 did not consume a value.
Thread 4 counting: 28
Thread 3 did not consume a value.
Thread 3 counting: 17
Thread 4 did not consume a value.
Thread 4 counting: 29
Thread 2 did not consume a value.
Thread 2 counting: 22
Thread 1 random value: 72
Thread 3 consumed value: 72
Thread 3 counting: 18
Thread 4 did not consume a value.
Thread 4 counting: 30
Thread 2 did not consume a value.
Thread 2 counting: 23
Thread 4 did not consume a value.
Thread 4 counting: 31
Thread 1 random value: 35
Thread 3 consumed value: 35
Thread 3 counting: 19
Thread 2 did not consume a value.
```

```
This code just ran for 10 seconds. This is a constexpr parameter.

Considering the offset of 100000 us, and the period of 300000 us, there were 33 iterations of Thread 1.

History of Thread 1: 58, 92, 64, 28, 99, 71, 71, 92, 5, 14, 37, 31, 96, 22, 39, 72, 35, 87, 4, 99, 13, 78
, 27, 91, 12, 50, 52, 98, 8, 15, 95, 83, 45,

Considering the offset of 150000 us, and the period of 200000 us, there were 50 iterations of Thread 2.

History of Thread 2: 58, 64, 99, 5, 37, 96, 39, 4, 13, 27, 12, 8, 95, 45,

Considering the offset of 160000 us, and the period of 250000 us, there were 40 iterations of Thread 3.

History of Thread 3: 92, 71, 71, 31, 72, 35, 78, 58, 52, 83,

Considering the offset of 170000 us, and the period of 150000 us, there were 66 iterations of Thread 4.

History of Thread 4: 20, 92, 14, 22, 87, 99, 91, 98, 15,
```

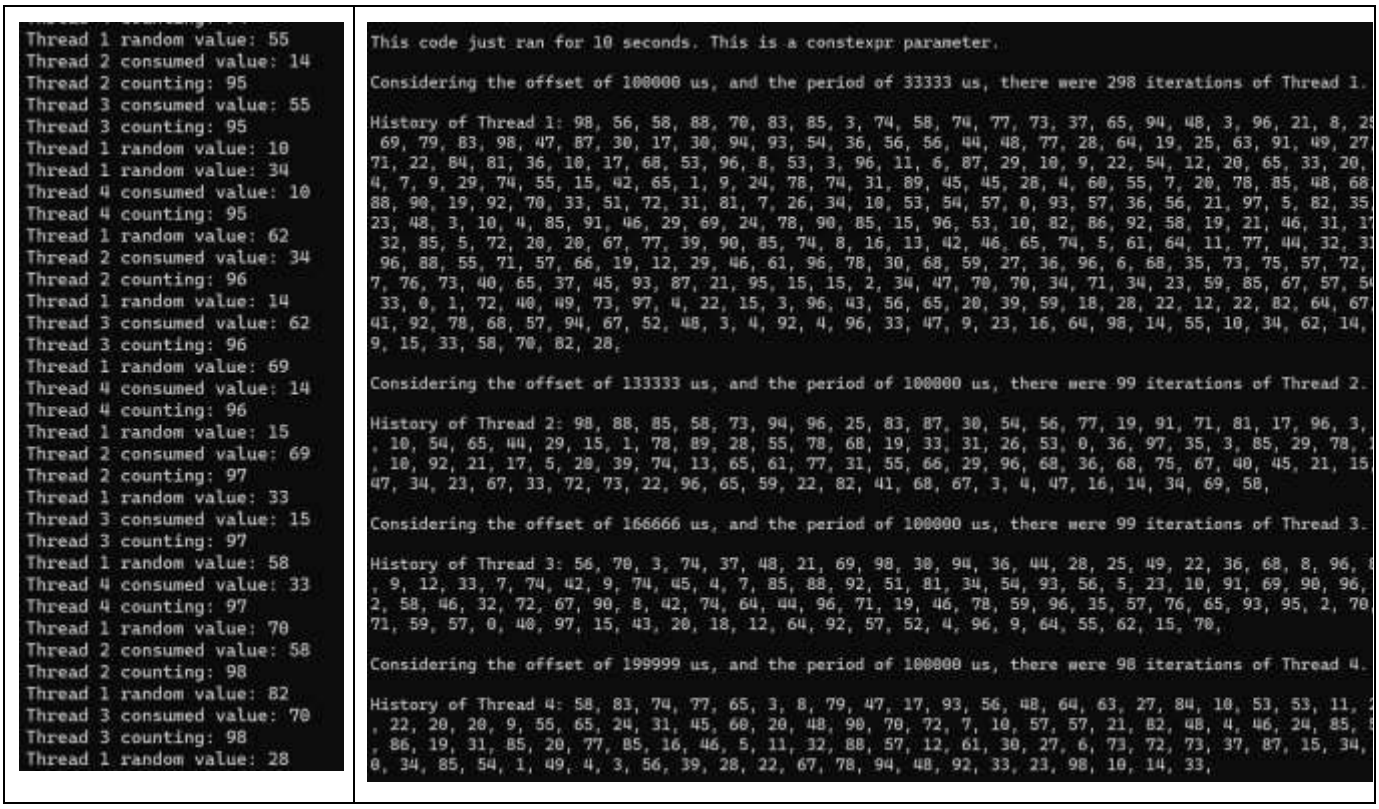
Si hacemos el buffer muy corto y los tiempos de consumo muy lentos, el productor no tiene más remedio que desechar datos:

```
Thread 1 random value: 67
Buffer is full, dropping value: 75
Buffer is full, dropping value: 79
Buffer is full, dropping value: 92
Thread 2 consumed value: 42
Thread 2 counting: 1
Thread 1 random value: 53
Buffer is full, dropping value: 14
Thread 3 consumed value: 67
Thread 3 counting: 1
Thread 1 random value: 42
Thread 4 consumed value: 53
Thread 4 counting: 1
Thread 1 random value: 41
Buffer is full, dropping value: 16
Thread 2 consumed value: 42
Thread 2 counting: 2
Thread 1 random value: 83
Buffer is full, dropping value: 14
Buffer is full, dropping value: 92
Thread 3 consumed value: 41
Thread 3 counting: 2
Thread 1 random value: 38
Buffer is full, dropping value: 90
Thread 2 consumed value: 83
```

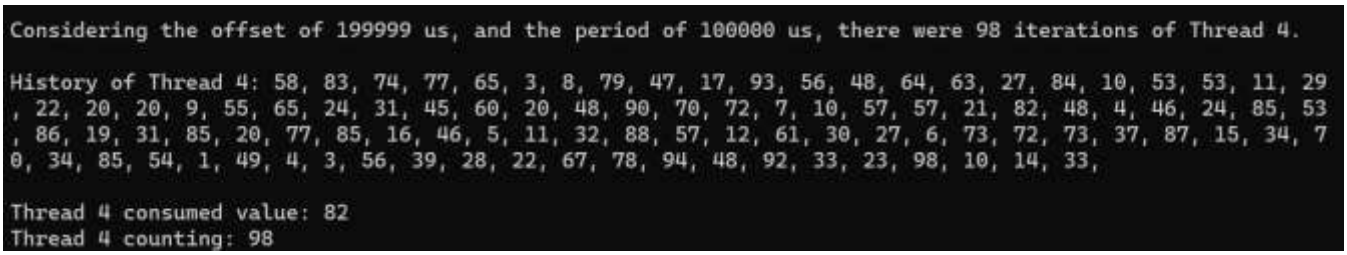
```
This code just ran for 10 seconds. This is a constexpr parameter.

Considering the offset of 100000 us, and the period of 100000 us, there were 100 iterations of Thread 1.

History of Thread 1: 33, 42, 67, 53, 42, 41, 83, 38, 81, 65, 97, 59, 90, 97, 91, 68, 76, 3, 79, 8, 24, 94
, 36, 66, 76, 0, 73, 12, 87, 44, 2, 41, 22, 60, 25, 5, 87, 7, 30, 11, 23, 95, 29, 22, 37, 45, 17, 90, 69,

Considering the offset of 150000 us, and the period of 500000 us, there were 20 iterations of Thread 2.

History of Thread 2: 33, 42, 42, 83, 65, 59, 97, 76, 8, 24, 66, 73, 87, 2, 68, 5, 30, 95, 29, 45,

Considering the offset of 160000 us, and the period of 650000 us, there were 16 iterations of Thread 3.

History of Thread 3: 67, 41, 81, 90, 60, 79, 36, 76, 12, 41, 25, 7, 23, 37, 17,

Considering the offset of 170000 us, and the period of 750000 us, there were 14 iterations of Thread 4.

History of Thread 4: 53, 38, 97, 91, 3, 94, 0, 44, 22, 87, 11, 22, 90,
```

En particular, una sincronización balanceada consistiría en que el consumidor produjera N veces mas rápido que los N consumidores. En este caso con N=3 podemos usar 33 ms para el periodo del productor. Luego que los consumidores consuman cada 100ms pero de forma intercalada. Uno al inicio (0ms offset), otro en el medio (33ms de offset) y otro al final (99ms de offset). De esta forma se puede obtener un sistema más balanceado:

```
constexpr int SECONDS_TO_RUN = 10;
constexpr int T1 = 33333; // Timing values in microseconds.
constexpr int T2 = 100000;
constexpr int T3 = 100000;
constexpr int T4 = 100000;

constexpr int OFF1 = 100000; // Offset values in microseconds.
constexpr int OFF2 = 133333;
constexpr int OFF3 = 166666;
constexpr int OFF4 = 200000;
```



En este caso vemos que la frecuencia de perdidas en la producción o en el consumo es nula y los consumidores lo hacen de forma balanceada. Sin embargo, quedo un último dato por consumir luego de la impresión de el historial:



A lo largo de este ejercicio, hemos trabajado en la implementación y depuración de un programa en C++ que gestiona la ejecución periódica de hilos utilizando POSIX threads (pthreads). Estos hilos tienen la tarea de producir y consumir valores de un búfer compartido, y para ello, es esencial garantizar la sincronización adecuada entre ellos para evitar condiciones de carrera y garantizar la coherencia de datos. Algunos conceptos clave fueron:

Modelo de Productor-Consumidor: Se implementó un modelo clásico de productor-consumidor, donde un hilo produce valores y los coloca en un búfer, mientras que otros hilos consumen estos valores.

Sincronización con Mutex: Se implementó std::mutex para garantizar que el acceso al búfer compartido y

al stream de impresión se realice de manera segura, evitando así posibles condiciones de carrera.

Diseño Modular: El programa realizado es fácilmente generalizable a diferentes combinaciones de m y n productores y consumidores. Mas aun la abstracción de tarea periódica permite cambiar la tarea que ni siquiera tienen que ver con el esquema de producción y consumidor. A su vez se propone pasar a las tareas el mutex del script principal para que haya sincronización con este thread también.

Ejecución Periódica: El código fue diseñado para ejecutarse periódicamente sin la necesidad de señales o eventos de notificación. Esto simplifica el diseño al eliminar la necesidad de manejadores de señales o lógica adicional para eventos.