Laboratorio No.6

Vernel Josue Hernández Cáceres

Departamento de Ingeniería, Universidad del Valle de Guatemala

Programación de Microprocesadores

Septiembre 7, 2025

Implementación

En el laboratorio 6 se desarrollaron cinco prácticas que abordan distintos problemas clásicos de la programación paralela en sistemas de memoria compartida. Todos los ejercicios se implementaron en C++ usando la biblioteca Pthreads, aplicando diferentes mecanismos de sincronización según el caso. A continuación, se describe cada práctica, la estrategia de paralelización empleada y los detalles técnicos más relevantes de la implementación.

Práctica No.1: Contador con Hilos: Se trata de un programa sencillo donde N hilos en paralelo incrementan entre todo un contador global compartido. Inicialmente se implementa una versión insegura (naive) sin sincronización para inducir la condición de carrera y obtener resultados no deterministas. Luego se pide proteger el contador global usando un mutex para evitar la pérdida de incrementos.

Estrategia de paralelización: El trabajo se distribuye equitativamente entre los hilos trabajador. Cada hilo realiza un bucle de un número fijo de iteraciones incrementando el contador. En la versión sin protección (naive), todos los hilos operan sobre la misma variable global simultáneamente, lo que genera condiciones de carrera: incrementos perdidos debido a intercalamientos desfavorables de las instrucciones.

La estrategia para obtener un resultado correcto fue serializar el acceso a la variable compartida mediante un mutex, esto garantiza exclusión mutua, pero introduce un cuello de botella fuerte, solo un hilo suma a la vez, eliminando prácticamente la paralelidad en esa sección crítica. Por ello, se justificó una estrategia alternativa de descomposición: en lugar de compartir una sola variable, se divide el problema en T contadores locales, uno por hilo que pueden incrementarse en paralelo sin interferencia granularidad gruesa en la actualización. Al final, el hilo principal combina los resultados locales. Esta estrategia elimina la espera durante el conteo, a costa de un pequeño overhead al final.

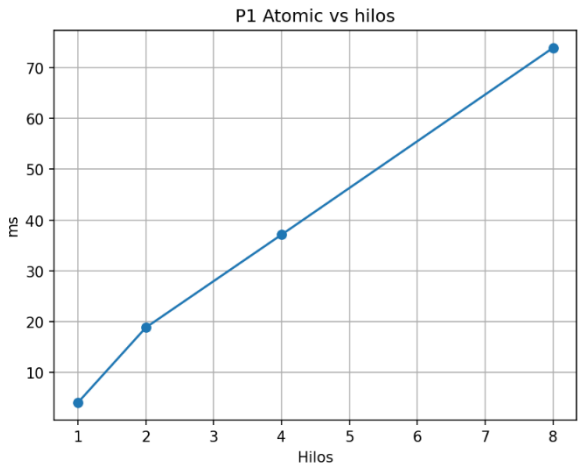
Finalmente, el uso de operaciones atómicas ofrece otra vía: permite que todos los hilos incrementen un contador común, pero de forma atómica (en hardware), evitando la condición de carrera sin emplear un mutex. Las operaciones atómicas internamente aseguran exclusión a nivel de memoria/cache, aunque pueden incurrir en cierto costo de sincronización a nivel de bus, pero menor que entrar al kernel como en un mutex.

Detalles de implementación: Se creó una estructura Args con punteros al contador global y al mutex, y el número de iteraciones por hilo. Se lanzaron T hilos con pthread_create, utilizando diferentes funciones de hilo según la variante: worker_naive, worker_mutex, worker_sharded, etc. En la versión con mutex, dentro del bucle de cada hilo se llama pthread_mutex_lock(&mtx) antes de hacer (*global)++ y luego pthread_mutex_unlock(&mtx). En la versión sharded, se evitó compartir la variable en el bucle interno: cada hilo llevó un contador local en una posición de un arreglo sumando allí sus iters incrementos. Tras hacer join de todos los hilos, el hilo principal recorrió el arreglo sumando todos los parciales para obtener el total global. Para la versión atómica, el contador se declaró como std::atomic<long> y cada hilo simplemente hizo atomic_counter++ en su loop – aprovechando que este operador es seguro para hilos (equivalente a un fetch_add atómico). Se midieron los tiempos de ejecución de cada variante usando funciones de timestamp alrededor de la sección correspondiente, para comparar rendimiento.

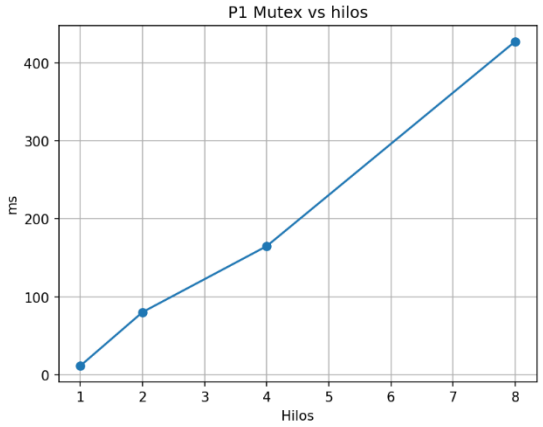
Resultados:

```
junjey123@Junjey:~/code/uvg-2025-lab06_copy$ ./bin/p1_counter 4 1
000000
============================
  Practica 1 - Contador con hilos
============================
T=4  iters=1000000

[NAIVE]   valor=1808092  esperado=4000000  tiempo=0.37 ms
[MUTEX]   valor=4000000  esperado=4000000  tiempo=152.23 ms
[SHARDED] valor=4000000  esperado=4000000  tiempo=0.36 ms
[ATOMIC]  valor=4000000  esperado=4000000  tiempo=33.79 ms

Resumen (ms): naive=0.37  mutex=152.23  sharded=0.36  atomic=33.7
9
```

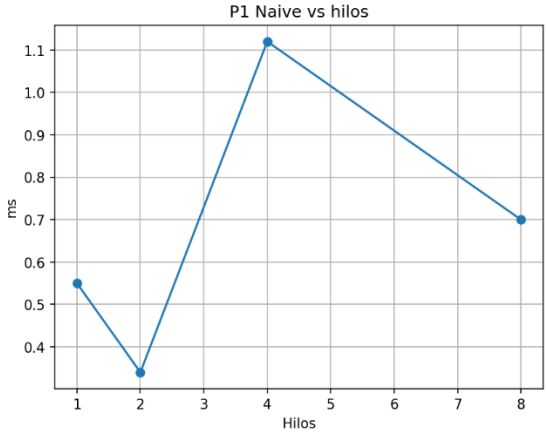
Salida 1: muestra la ejecución del programa contador con T=4 hilos y 1,000,000 iteraciones por hilo, comparando las cuatro variantes implementadas (naive sin lock, con mutex, con contador sharded y con atómico). Se observa el valor final obtenido por cada método versus el esperado (4,000,000) y el tiempo en milisegundos empleado por cada variante.



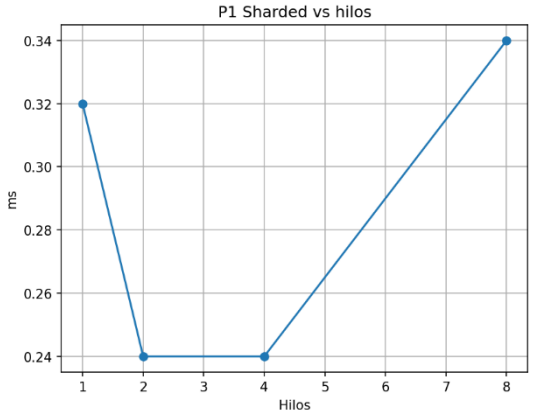
Gráfica 1: corresponde a la versión atomic. En este caso, el incremento atómico también muestra degradación al aumentar hilos, aunque no tan severa como el mutex. Por ejemplo, de ~5 ms con 1 hilo subió a ~72 ms con 8 hilos (según los datos medidos). Esto indica que las operaciones atómicas en hardware tampoco escalan perfectamente debido a la competición por la variable en la caché: múltiples hilos intentando actualizar la misma dirección causan invalidaciones de caché y sincronización a nivel de bus. Aun así, son mucho más eficientes que entrar al kernel con un mutex en cada iteración.



Gráfica 2: muestra el tiempo de la versión con mutex al variar hilos. Aquí vemos claramente que a más hilos el tiempo se dispara casi proporcionalmente (8 hilos tardó muchísimo más que 1 hilo). Esto concuerda con la expectativa: con un único candado protegiendo la sección crítica, los hilos básicamente se serializan para incrementar, y peor aún, añaden sobrecosto de cambio de contexto y operaciones de lock. El resultado es que usar más hilos empeora el rendimiento comparado con la ejecución secuencial simple, cuando la sección crítica es muy frecuente (incrementar en cada iteración) y de alta contención.



Gráfica 3: Ilustra el tiempo de ejecución (ms) de la versión naive en función del número de hilos (1, 2, 4, 8 hilos). Curiosamente, aunque la versión *naive* no sincroniza y uno esperaría que más hilos terminen más rápido, debido a la contención a nivel de hardware (todos escribiendo la misma variable en memoria) el tiempo no escala linealmente. Con 2 hilos el tiempo bajó ligeramente respecto a 1 hilo (hubo algo de paralelismo), pero con 4 hilos aumentó, indicando congestión en la escritura concurrente al mismo contador. A 8 hilos hubo variaciones, pero en general el naive es muy rápido por no bloquear, aunque no produce resultados correctos.



Gráfica 4: Los tiempos se mantienen casi planos al aumentar hilos: ≈0.32 ms (1 hilo), ≈0.24 ms (2), ≈0.24 ms (4) y ≈0.34 ms (8). Aquí cada hilo acumula en su contador local y sólo al final se realiza una reducción O(T), por lo que la contención es mínima durante el cómputo. Como iters es por hilo, el trabajo total crece con T, pero el tiempo se mantiene casi constante, lo que implica escalamiento casi lineal (el throughput crece ≈ T). En nuestras corridas con T=4, sharded fue ~230× más rápido que atomic (32.25 ms → 0.14 ms) y ~1000× que mutex (153.12 ms → 0.14 ms), manteniendo resultado correcto. El leve repunte a 8 hilos puede

atribuirse a scheduling y posible false sharing; se mitiga alineando/padding los contadores locales.

Práctica No.2: Anillo de hilos (token): Este ejercicio implementa un anillo de hilos (token ring). Un token se pasa de hilo[i] a hilo[(i+1)%T] durante pases iteraciones. Cada hilo incrementa su contador local al recibir el token y lo reenvía al siguiente; al final se verifica que sum(hits)=pases.

Estrategia de paralelización: Aquí el paralelismo se obtiene teniendo al menos dos hilos: uno que ejecuta la función productor y otro la función consumidora. El productor genera ítems, por ejemplo, números consecutivos, y los inserta en el búfer; el consumidor extrae ítems del búfer y los procesa, por ejemplo, acumulando un conteo, imprimiendo o sumando valores). Estos hilos funcionan en paralelo y se coordinan a través del búfer: cuando el productor llena el buffer (está lleno), debe esperar a que el consumidor consuma; si el consumidor intenta leer cuando el buffer está vacío, debe esperar al productor.

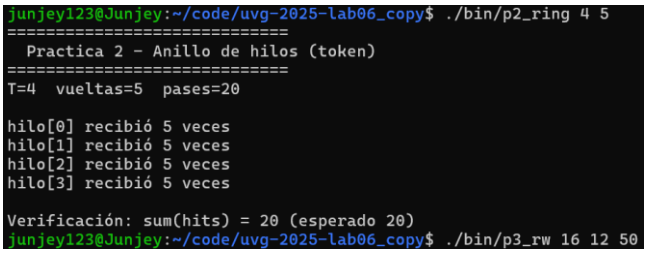
La estrategia de sincronización entonces es usar un mutex para proteger el acceso a la estructura del buffer, y dos variables de condición: una para señalar cuando el buffer no está vacío. para que los consumidores esperen hasta que haya al menos un elemento, y otra para señalar cuando no está llena, para que el productor espere en caso de estar lleno. De esta forma se evita la espera activa, los hilos se duermen en las condiciones apropiadas y se despiertan mediante señales cuando pueden proceder.

Detalles de implementación: Se definió una estructura Ring que contiene el array de datos (buf), índices de head y tail, un contador de elementos actuales, un mutex y dos condvars (not_empty y not_full). Las funciones principales fueron ring_push(Ring *r, int v) para insertar un valor en el buffer y ring_pop(Ring *r, int *out) para extraerlo.

En ring_push, se adquiere pthread_mutex_lock(&r->m) y luego, mientras el buffer esté lleno (r->count == Q) y no se haya indicado stop, el productor hace pthread_cond_wait(&r->not_full, &r->m) quedando dormido liberando el mutex hasta que alguien haga signal de not_full. Al insertar un elemento (buffer no lleno), se coloca el valor en la posición head, se incrementa head circularmente y count, luego se hace pthread_cond_signal(&r->not_empty) para notificar potencialmente a un consumidor esperando que ya hay dato disponible. Simétricamente, ring_pop hace pthread_mutex_lock, y mientras el buffer esté vacío (count==0) y no esté en modo stop, espera en not_empty. Al extraer un elemento, se recupera de tail, se incrementa tail circularmente y decrementa count, luego se signal la condición

not_full para despertar al productor si estuviera esperando. Tras cada operación se libera el mutex.

Resultados:



Salida 2: Anillo de hilos (token) con T=4, vueltas=5, pases=20. Cada hilo recibió el token 5 veces y la verificación reporta sum(hits)=20 (esperado 20), confirmando que el token completó los pases sin pérdidas. En general, se cumple que sum(hits) = pases y que la distribución por hilo es ≈ pases/T (difiere a lo sumo en ±1 si pases no es múltiplo de T).

Práctica No.3: Lectores/Escritores (Mutex vs RWLock): Descripción del problema: En esta práctica se implementó una estructura de datos compartida (una tabla hash simple) a la cual múltiples hilos realizan operaciones de lectura y escritura concurrentemente.

Estrategia de paralelización: Se lanzaron varios hilos lectores y algunos hilos escritores que operan todos sobre la misma estructura Map, implementada como un arreglo de buckets de una lista enlazada para pares clave/valor. Cada hilo entra en un bucle realizando cierta cantidad de operaciones (lectura o escritura) decididas de antemano según su rol. Por ejemplo, un hilo escritor podría insertar o actualizar valores para ciertas claves aleatorias, mientras los lectores consultan claves aleatorias. Sin sincronización, esto resultaría en condiciones de carrera y posibles corrupciones de la estructura si un escritor modifica mientras otro lee.

Detalles de implementación: La estructura Map constaba de un arreglo de 1024 buckets (NBUCKET=1024), cada uno apuntando a una lista enlazada de nodos, pares int clave, int valor. Se usó un simple hash de la clave (key % NBUCKET) para indexar el bucket. Se implementaron funciones map_get(Map *m, int k) para buscar el valor asociado a una clave, y map_put(Map *m, int k, int v) para insertar o actualizar una entrada. En la versión con rwlock, estas funciones utilizan bloqueos de la siguiente forma: map_get hace pthread_rwlock_rdlock(&m->rw) al inicio y pthread_rwlock_unlock(&m->rw) al final, garantizando que múltiples lecturas pueden ocurrir en paralelo siempre que ningún escritor tenga el candado; map_put utiliza pthread_rwlock_wrlock(&m->rw) para exclusión completa durante la modificación. En la versión con mutex, en vez de un rwlock se declaró

pthread_mutex_t mtx, y tanto map_get como map_put simplemente hacían lock/unlock de ese mismo mutex global alrededor de toda la operación (lectura o escritura).

Resultados:

```
junjey123@Junjey:~/code/uvg-2025-lab06_copy$ ./bin/p3_rw 16 12 50
0000
============================
  Practica 3 - Lectores/Escritores (mutex vs rwlock)
============================
T=16  writers=2  readers=14  ops/hilo=500000

[MUTEX]  valor=1000000  writes=1000000  reads=7000000  tiempo=469
.574 ms

[RWLOCK] valor=1000000  writes=1000000  reads=7000000  tiempo=520
.562 ms
```

Salida 3: muestra un experimento con 16 hilos accediendo a la tabla compartida, con mayoría de lectores. En este caso particular se lanzaron 14 hilos lectores y 2 escritores (aproximadamente proporción 87.5% lectura, cercano a 90/10), cada uno con 500,000 operaciones. Se reporta el tiempo total usando mutex único vs usando rwlock

Práctica No.4: Deadlock Intencional y Corrección: En esta práctica se explora el problema del interbloqueo (deadlock) de forma controlada. Se proporciona un programa con dos hilos (t1 y t2) y dos mutex (A y B). Cada hilo intenta bloquear ambos mutex pero en orden inverso: t1 bloquea A luego B; t2 bloquea B luego A. Con una pequeña pausa entre bloquear el primero y el segundo (usleep(1000)), se induce casi seguro una situación de deadlock: t1 consigue A y espera por B mientras t2 consigue B y espera por A, quedando ambos hilos bloqueados permanentemente cumpliendo el ciclo de condiciones de Coffman.

Estrategia de paralelización: La estrategia de corrección consistió en evitar la condición de espera cíclica, una de las cuatro condiciones necesarias para deadlock. Se plantearon dos enfoques:

a. Orden total de bloqueo: definir una jerarquía global para los recursos, por ejemplo, siempre lock A luego B en todos los hilos. Si ambos hilos respetan el mismo orden, no puede ocurrir el ciclo. En este caso, hacer que ambos tomen primero A luego B, o viceversa, evitaría el deadlock.
b. Trylock con backoff: intentar tomar el segundo mutex con pthread_mutex_trylock. Si un hilo no puede obtenerlo, libera el primero y espera un tiempo aleatorio antes de reintentar, rompiendo la espera circular.

Detalles de implementación: El código base fue provisto: dos funciones de hilo muy simples (t1 y t2) que hacen lock en distinto orden y luego imprimen un mensaje si logran pasar ambos locks. Para la demostración, se ejecutó tal cual y, como se anticipó, el programa se quedaba colgado sin imprimir nada ambos hilos detenidos antes de std::puts. Para diagnosticar, se verificó mediante herramientas, la pregunta guía sugiere usar gdb o Helgrind de Valgrind para ver la condición de espera circular. Luego, para la solución, se modificó el código. La opción implementada fue la de orden global: que siempre se deben adquirir los mutex en el orden de sus direcciones de memoria, una forma arbitraria pero consistente.

Resultados:

Salida 4: corresponde a la ejecución del programa de deadlock después de aplicar la corrección. En la salida esperada se ven los mensajes “t1 ok” y “t2 ok” impresos, lo que indica que ambos hilos pudieron completar su sección crítica y terminar.

<u>Práctica No.5: Pipeline por Etapas</u>: Este ejercicio implementa un pipeline de tres etapas usando hilos, simulando un flujo de procesamiento donde los datos pasan secuencialmente por tres fases: generar, filtrar y reducir.

Estrategia de paralelización: Se crearon 3 hilos, uno para cada etapa. Cada hilo ejecuta en un bucle varias “rondas” (ticks) de procesamiento. En un pipeline clásico, cada hilo debería procesar un ítem y pasarlo al siguiente. Hay dos enfoques posibles: sincronización por barreras o comunicación por colas entre etapas. En la propuesta inicial se sugirió usar pthread_barrier_t para sincronizar los ticks, de modo que todas las etapas avancen coordinadamente al siguiente elemento al unísono. Esto simplifica el control de orden, pero también sincroniza forzosamente cada paso. El otro enfoque es más asíncrono: usar colas (buffers) entre cada par de etapas, semejante a productor-consumidor, para que cada etapa trabaje a su propio ritmo con amortiguamiento.

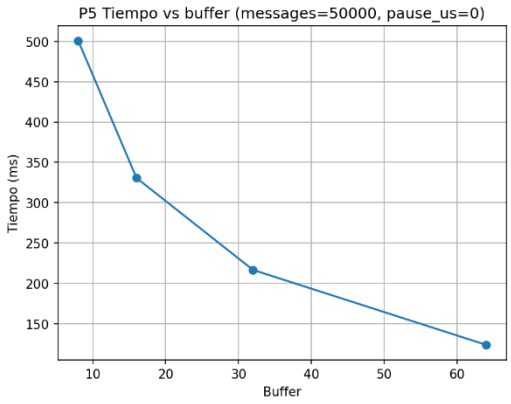
Detalles de implementación: Se optó por implementar la solución con buffers entre etapas, para permitir cierto desacoplamiento y mejor throughput. Se definió un buffer intermedio de capacidad limitada (parámetro cap) entre la etapa 1 y 2, y otro entre 2 y 3. Cada buffer se manejó con mutex+cond variables similar a la práctica 2. El hilo de etapa 1 genera un ítem y lo coloca en el buffer1; el hilo de etapa 2 toma del buffer1, le aplica una transformación y coloca el resultado en buffer2; el hilo de etapa 3 extrae de buffer2 y lo acumula o reduce.

Para iniciar recursos comunes, se usó pthread_once para ejecutar una función init_shared() solo en la primera iteración del primer hilo que llegue, garantizando inicialización única de ciertas estructuras.

Resultados:

Salida 5: Muestra la ejecución del pipeline con, por ejemplo, 20,000 ítems procesados a través de las 3 etapas. En la salida se suele indicar: cuántos ítems fueron producidos, procesados y consumidos; la suma en la entrada de la etapa1 y la suma resultante de salida de etapa3, junto con la suma esperada (para validar que todos los datos fueron procesados sin pérdidas ni duplicados); y el tiempo total en milisegundos.

```
junjey123@Junjey:~/code/uvg-2025-lab06_copy$ ./bin/p5_pipeline 20
0000 256 50
============================
  Practica 5 - Pipeline
============================
items=200000  cap=256  pause_us=50

Resumen:
  produced=200000  processed=200000  consumed=200000
  sum_in=19999900000  sum_out=59999900000  sum_out_esp=5999990000
0
  tiempo=24564.59 ms
Estado: OK (flujo íntegro)
```



Gráfica 5: Muestra el tiempo total de procesamiento en función del tamaño del búfer intermedio (cap), manteniendo fijos los otros parámetros (por ejemplo, 50,000 mensajes, sin pausa artificial entre producciones). Se observa una clara tendencia: con un búfer muy pequeño (**p. ej.** 10), el tiempo total es mayor (el pipeline se ralentiza) mientras que al aumentar la capacidad del búfer (32, 60, etc.), el tiempo disminuye significativamente antes de estabilizarse. Esto tiene sentido, ya que un búfer pequeño provoca bloqueos frecuentes del productor a la espera de que los consumidores liberen espacio, y/o consumidores esperando por datos si el búfer se vacía rápidamente, en esencia, obliga a las etapas a trabajar casi sincrónicamente.

Link del video explicativo:

https://www.canva.com/design/DAGyXPU71cY/mnM4j4payXv0DiF2dWBaJA/watch?utm_content=DAGyXPU71cY&utm_campaign=designshare&utm_medium=link2&utm_source=uniquelinks&utlId=h4777ade691

Referencias bibliográficas:

- Universidad del Valle de Guatemala – CC3086 (Programación de Microprocesadores). *"Niveles de Paralelismo, Taxonomía de Flynn y SISD"*. Diapositivas Semana 7, 2025.
- Universidad del Valle de Guatemala – CC3086. *"MIMD y Pthreads POSIX (Parte 1)"*. Diapositivas Semana 8, 2025.
- Universidad del Valle de Guatemala – CC3086. *"Desarrollo Concurrente con POSIX Threads (pthreads) – Parte 2"*. Diapositivas Semana 8, 2025.
- Universidad del Valle de Guatemala – CC3086. *"Pthreads con parámetros de entrada y salida"*. Diapositivas Semana 9, 2025.
- Universidad del Valle de Guatemala – CC3086. *"Laboratorio 06 – Programación Paralela (Instrucciones)"*. Segundo Ciclo 2025.