Programación de Microprocesadores

Laboratorio #6

Santiago Cordero (24472)

Sección 10

Licenciatura en Ingeniería en Ciencias de la Computación y Tecnologías de la Información

Guatemala, septiembre del 2025

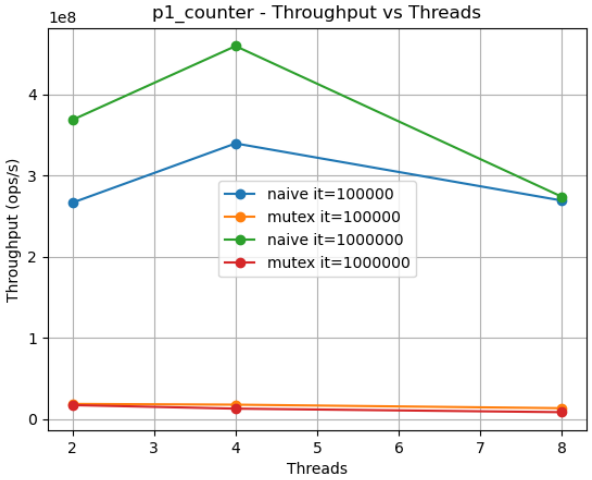
Explicación por Práctica

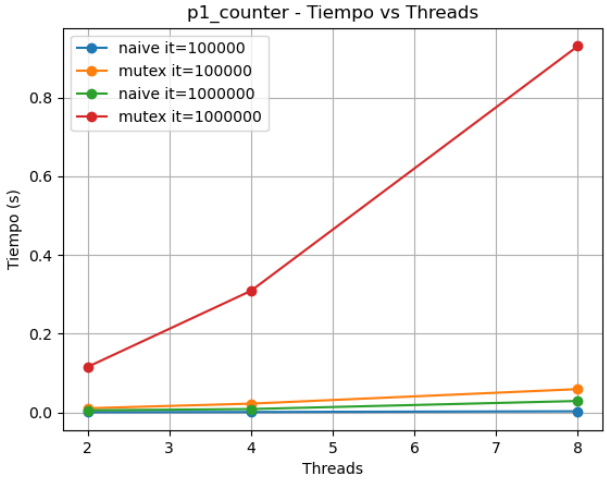
Práctica #1

Captura de funcionalidad del código: (versión sin medición de tiempo)

```
scor05@Dell-Latitude:/mnt/c/Users/monic/Documen
NAIVE total=1493586 (esperado=4000000)
NAIVE total=1493586 (esperado=4000000)
NAIVE total=1493586 (esperado=4000000)
NAIVE total=1493586 (esperado=4000000)
MUTEX total = 3937798 (esperado = 4000000)
MUTEX total = 3943874 (esperado = 4000000)
MUTEX total = 4000000 (esperado = 4000000)
MUTEX total = 4000000 (esperado = 4000000)
```

Gráficas medidas:





Explicación:

El diseño consistió en un contador compartido que varios hilos incrementan en paralelo, con dos variantes: una “naive” sin sincronización y otra usando pthread_mutex_t. La decisión de usar mutex responde a la necesidad de garantizar exclusión mutua y consistencia en el valor final del contador, aunque a costa de mayor tiempo de ejecución.

Los resultados muestran lo esperado: el caso “naive” obtiene tiempos muy bajos y altos throughputs aparentes, pero los valores finales del contador son incorrectos debido a condiciones de carrera. En contraste, el caso con mutex produce siempre el valor esperado, aunque con throughput mucho menor (cientos de veces más lento), reflejando la sobrecarga de sincronización.

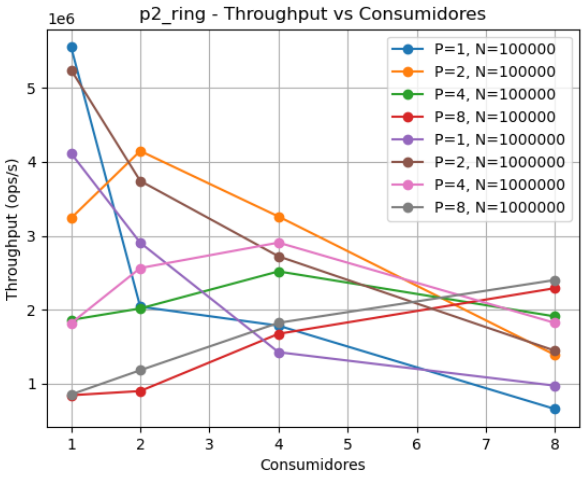
En el post mortem, se concluye que este ejercicio deja clara la diferencia entre rendimiento y correctitud: no basta con ejecutar más rápido, el resultado debe ser correcto. La sincronización con mutex asegura consistencia, y la comparación de gráficas evidencia la penalización en tiempo y throughput al evitar condiciones de carrera.

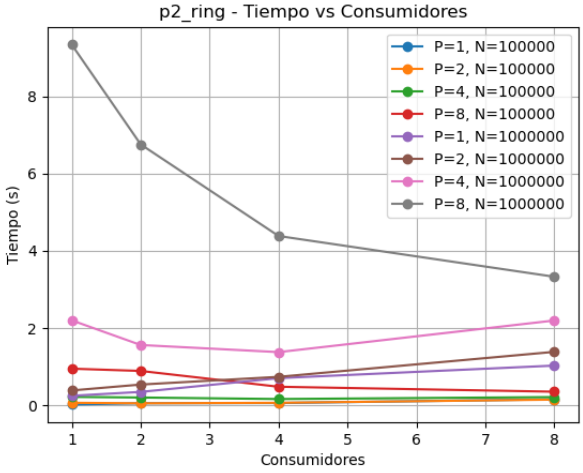
Práctica #2

Captura de funcionalidad: (previo a adición de cálculos de tiempo)

```
$ ./src/p2_ring.exe
Pusheado 0 al ring
Pusheado 1 al ring
Pusheado 2 al ring
Pusheado 3 al ring
Pusheado 4 al ring
Consumidor sacó 0 del ring
Consumidor sacó 1 del ring
Consumidor sacó 2 del ring
Consumidor sacó 3 del ring
Consumidor sacó 4 del ring
```

Gráficas:





Explicación:

Aquí se diseñó un buffer circular de tamaño fijo, con productores y consumidores concurrentes sincronizados mediante mutex y variables de condición (pthread_cond_t). La decisión de este modelo se debe a que el acceso al buffer requiere control tanto en el caso de cola llena como vacía, evitando bloqueos activos y pérdidas de datos.

Las gráficas de tiempo y throughput muestran que, a medida que aumentan productores y consumidores, se alcanzan diferentes equilibrios: con pocos consumidores el buffer se vacía lentamente, mientras que con demasiados consumidores hay espera innecesaria. El throughput máximo se da en configuraciones balanceadas de P=C, pero decrece cuando la contención en el mutex se intensifica.

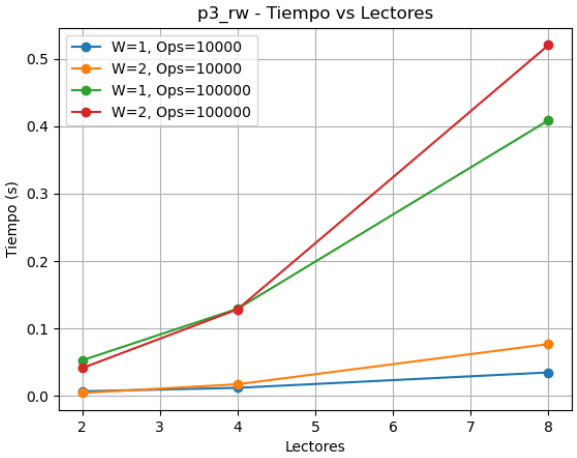
El post mortem indica que el diseño funcionó correctamente para mantener integridad (nunca se pierden ni duplican datos). Sin embargo, se observó que el rendimiento no escala linealmente: la sobrecarga de sincronización limita la ganancia, lo cual es una lección clara sobre los costos reales de la concurrencia.

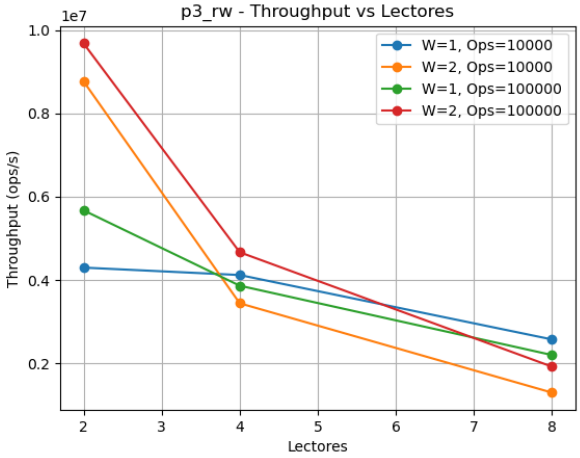
Práctica #3

Captura de funcionalidad: (previo a cálculos de tiempo)

```
Distribución de carga: 90/10

Duración: 0.056311 s, Throughput: 1775.849726 operaciones/s
--------------------------------
Distribución de carga: 70/30

Duración: 0.041176 s, Throughput: 2428.585618 operaciones/s
--------------------------------
Distribución de carga: 50/50

Duración: 0.036865 s, Throughput: 2712.591492 operaciones/s
```

Gráficas:





Explicación:

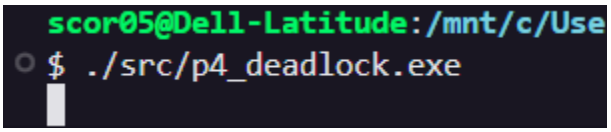
En este ejercicio se implementó un hash map concurrente con pthread_rwlock_t, que permite acceso simultáneo de múltiples lectores pero bloquea a todos durante las escrituras. La decisión de usar rwlocks en lugar de mutex tradicionales se justifica porque la mayoría de cargas prácticas tienden a estar dominadas por lecturas.

Los resultados de las gráficas confirman el comportamiento esperado: en escenarios con muchos lectores y pocos escritores, el throughput es alto, mientras que al aumentar la proporción de escritores el rendimiento cae drásticamente. Además, se observa que el tiempo total crece de forma significativa cuando los escritores bloquean a todos los lectores.

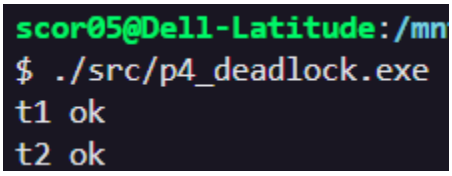
En el post mortem, se confirma que el uso de rwlocks ofrece ventajas frente a mutex en cargas de lectura intensiva, pero no es una solución mágica: cuando la proporción de escrituras aumenta, se pierde el beneficio. La práctica permite entender mejor cómo elegir la primitiva de sincronización adecuada según el perfil de acceso.

Práctica #4

Captura de demostración del deadlock: (al compilar y ejecutar el código se congela la terminal)



Captura de funcionalidad con el deadlock ya corregido con el orden global:



Explicación:

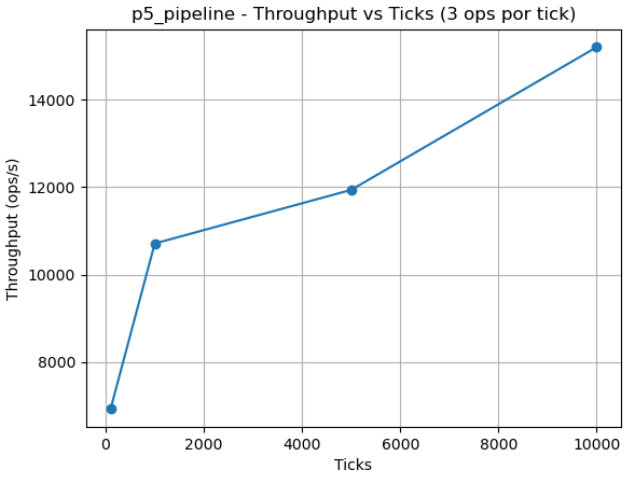
El diseño original provocaba un deadlock porque dos hilos adquirían los locks en distinto orden. La corrección consistió en imponer un orden global (A a B) para todos, lo que elimina la condición circular que caracteriza a los deadlocks. No se midió tiempo ni throughput, ya que el interés de la práctica era cualitativo: demostrar cómo el programa se congela en el caso incorrecto y cómo termina exitosamente en la versión corregida. El post mortem deja claro que la prevención de deadlocks no es cuestión de rendimiento, sino de disciplina en el diseño: reglas simples como “todos adquieren locks en el mismo orden” previenen problemas complejos.

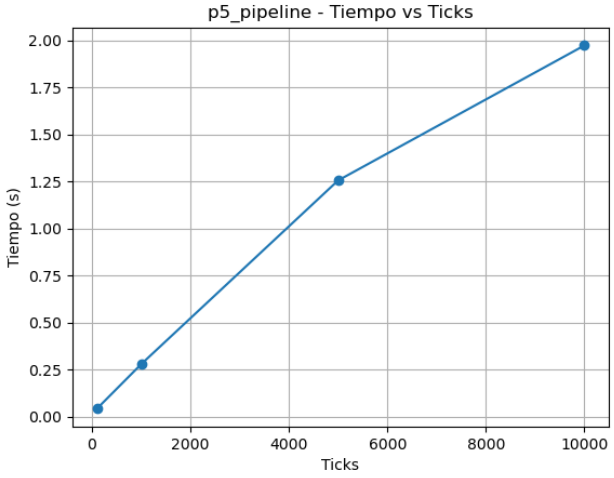
Práctica #5

Captura de funcionalidad: (captura del log)

```
Pipeline inicializado
Tick #0 generó 93
Tick #0 filtró 93
Tick #0 sumó 93 al acumulado: 93
Tick #1 generó 4
Tick #1 descartó a 4
Tick #2 generó 50
Tick #2 descartó a 50
Tick #3 generó 11
Tick #3 filtró 11
Tick #3 sumó 11 al acumulado: 104
```

Gráficas:





Explicación:

Se implementó un pipeline de tres etapas sincronizadas con barreras: generación de números, filtrado de pares y acumulación. La decisión de usar múltiples barreras en cada tick fue crucial para asegurar que ninguna etapa se adelante a otra, manteniendo el flujo correcto de datos.

Las gráficas muestran un tiempo que crece linealmente con el número de ticks, lo cual es esperado ya que cada tick representa un ciclo completo de las tres etapas. El throughput en ticks/s se mantiene relativamente constante, mientras que el throughput en ops/s refleja las tres operaciones por tick, confirmando que el diseño escala de forma proporcional.

En el post mortem, se destaca que el uso de barreras fue apropiado para coordinar fases en paralelo, aunque la sobrecarga de sincronización limita el rendimiento. La práctica refuerza el concepto de que en pipelines sincronizados la latencia por tick es fija, y el diseño debe balancear claridad y eficiencia.

Anexos

Enlace a Youtube del vídeo explicativo:

https://youtu.be/D0_EAbB1Mfc

Link del repositório de github:

https://github.com/scor05/Laboratorio-Acceso-a-Recursos-Compartidos