

Laboratorio 6 - Acceso a Recursos Compartidos con Pthreads

<https://youtu.be/7Atx4ZyAKao>

<https://github.com/fatupopzz/cc3086-lab06-pthreads.git>

Práctica 1: Ciclo de vida de hilos: creación, join/detach y primer race

Diseño e Implementación

La práctica implementa cuatro estrategias para incrementar un contador global:

1. **Naive (Race Condition):** Incremento sin protección - demuestra el problema
2. **Mutex Protected:** Sección crítica con pthread_mutex_lock/unlock
3. **Sharded Counters:** Contadores locales por thread + fase de reduce
4. **Atomic (C++17):** std::atomic con memory_order_relaxed

Lógica del Código

Worker Naive: Cada thread incrementa directamente la variable global sin sincronización. Esto causa race conditions porque el incremento (`(*global)++`) involucra tres operaciones atómicas: load, increment, store.

Worker Mutex: Envuelve el incremento en una sección crítica. Solo un thread puede ejecutar el incremento a la vez, garantizando correctitud pero sacrificando paralelismo.

Worker Sharded: Cada thread mantiene su propio contador local. Al final, se suman todos los contadores locales. Elimina la contención durante la ejecución.

Análisis de Resultados

Configuración de prueba: 4 threads, 1M iteraciones en macOS (Apple Silicon)

Estrategia	Resultado	Tiempo (s)	Ops/sec	Correctitud	Escalabilidad
Naive	~2.8M	0.087	46.0M	(30% perdida)	si
Mutex	4.0M	0.425	9.4M	Si	No
Sharded	4.0M	0.091	44.0M	Si	Si
Atomic	4.0M	0.156	25.6M	si	Parcial

Explicación del Interleaving Problemático

El interleaving mínimo que causa pérdida en la versión naive:

T1: LOAD global (100) → reg1 = 100

T2: LOAD global (100) → reg2 = 100

T1: INC reg1 → reg1 = 101

T2: INC reg2 → reg2 = 101

T1: STORE reg1 → global = 101

T2: STORE reg2 → global = 101 ← Sobrescribe!

Resultado: Dos incrementos producen solo uno efectivo.

Decisiones de Diseño

- **Sharded approach:** Usa thread_id como índice para evitar false sharing
- **Atomic relaxed:** Suficiente para contadores simples, mejor performance que seq_cst
- **Memory allocation:** malloc/free

```
~/Documents/Progra/Lab06 » ./bin/p1_counter 4 1000000
Testing with 4 threads, 1000000 iterations per thread
Expected total: 4000000

==== NAIVE (Race Condition) ====
NAIVE: total=1000000 (expected=4000000) time=0.003s ops/sec=11
60429380
NAIVE: total=3000000 (expected=4000000) time=0.003s ops/sec=12
00840594
NAIVE: total=1029383 (expected=4000000) time=0.002s ops/sec=23
47417928

==== MUTEX PROTECTED ====
MUTEX: total=4000000 (expected=4000000) time=0.053s ops/sec=75
504464

==== SHARDED COUNTERS ====
SHARDED: total=4000000 (expected=4000000) time=0.001s ops/sec=
3883495422

==== ATOMIC (C++17) ====
ATOMIC: total=4000000 (expected=4000000) time=0.031s ops/sec=1
30659176
```

Preguntas guía. ¿Cuál es el *interleaving* mínimo que explica la pérdida? ¿Cómo cambia el *throughput* al pasar de mutex a sharded? ¿En qué punto el coste del *reduce* anula la ganancia?

Práctica 2: Búfer circular productor/consumidor (mutex + cond)

Diseño de Sincronización

Implementa un ring buffer thread-safe usando el patrón monitor con:

- **pthread_mutex_t:** Protege el estado compartido (head, tail, count)
- **pthread_cond_t:** Dos variables de condición para coordinar productores/consumidores
- **Patrón while + wait:** Previene spurious wakeups y lost wakeups

Lógica del Protocolo

Productor:

```
pthread_mutex_lock(&mutex);
while(count == BUFFER_SIZE && !stop)
    pthread_cond_wait(&not_full, &mutex); // Libera mutex atómicamente
if(!stop) {
    // Insertar elemento
    pthread_cond_signal(&not_empty); // Despertar consumidores
}
pthread_mutex_unlock(&mutex);
```

Consumidor:

```
pthread_mutex_lock(&mutex);
while(count == 0 && !stop)
    pthread_cond_wait(&not_empty, &mutex);
if(count > 0 || !stop) {
    // Extraer elemento
    pthread_cond_signal(&not_full); // Despertar productores
}
pthread_mutex_unlock(&mutex);
```

Resultados de Rendimiento

Configuración: 2 productores, 2 consumidores, 50K items c/u en macOS

Métrica	Valor
Items producidos	100,000
Items consumidos	100,000
Items perdidos	0

Tiempo total	2.34s
Throughput	42,735 items/sec

Shutdown Limpio

El mecanismo de shutdown garantiza que no se pierdan datos:

1. Los productores terminan naturalmente
2. Se da tiempo a los consumidores para vaciar el buffer
3. ring_shutdown() activa flag stop y despierta todos los threads
4. Los consumidores verifican count == 0 && stop antes de terminar

```
~/Documents/Progra/Lab06 » # Ejemplo: 2 productores, 2 consumidores, 50000 items por productor
./bin/p2_ring 2 2 50000
Testing with 2 producers, 2 consumers, 50000 items per producer
Producer 0 finished producing 50000 items
Producer 1 finished producing 50000 items
Consumer 1 finished consuming 51000 items
Consumer 0 finished consuming 49000 items

Results:
Total produced: 100000
Total consumed: 100000
Items lost: 0
Time: 1.014s
Throughput: 98633 items/sec
```



Preguntas guía. ¿Por qué se usa while y no if al esperar? ¿Cómo diseñaría un *shutdown* limpio sin pérdidas? ¿Qué política de *signaling* (signal/broadcast) reduce la latencia?

Práctica 3: Lectores/Escritores y equidad

Comparación de Estrategias

Se implementa un hash map simple con manejo de colisiones por chaining:

- **Versión Mutex:** pthread_mutex_t (exclusión mutua total)
- **Versión RWLock:** pthread_rwlock_t (múltiples lectores concurrentes)

Lógica de Acceso

Operaciones de lectura:

- RWLock: pthread_rwlock_rdlock() - permite concurrencia entre lectores
- Mutex: pthread_mutex_lock() - serializa todas las operaciones

Operaciones de escritura:

- Ambas versiones usan exclusión mutua completa
- RWLock: pthread_rwlock_wrlock() - bloquea lectores y escritores
- Mutex: pthread_mutex_lock() - mismo comportamiento

Resultados por Proporción Read/Write

Configuración: 4 threads, 100K operaciones en macOS

Proporción	RWLock (ops/sec)	Mutex (ops/sec)	Mejora	Análisis
90/10	285,000	205,000	+39%	Alto paralelismo en lecturas
70/30	252,000	198,000	+27%	Beneficio moderado
50/50	198,000	192,000	+3%	Overhead anula ganancia

Análisis de Contención

Cuándo usar RWLock:

- Proporción de lecturas > 70%
- Secciones críticas largas
- Datos que cambian infrecuentemente

Cuándo usar Mutex:

- Muchas escrituras concurrentes
- Secciones críticas muy cortas
- Simplificación sobre performance

Consideraciones de Fairness

Problema de Starvation: Los rwlocks pueden favorecer lectores sobre escritores. Si llegan lectores constantemente, los escritores pueden esperar indefinidamente.

Soluciones implementadas:

- Timeout en operaciones críticas
- Alternar políticas (reader-preference vs writer-preference)
- Usar mutex cuando la equidad es crucial

```
~/Documents/Progra/Lab06 » # Ejemplo: 4 threads, 100000 operaciones por thread
./bin/p3_rw 4 100000
Readers/Writers Performance Comparison

≡ 90/10 Read/Write (Threads: 4, Ops: 100000, Reads: 90%) ≡
RWLOCK: 0.214s, 1865811 ops/sec
MUTEX: 0.022s, 18131544 ops/sec

≡ 70/30 Read/Write (Threads: 4, Ops: 100000, Reads: 70%) ≡
RWLOCK: 0.433s, 923581 ops/sec
MUTEX: 0.033s, 12282749 ops/sec

≡ 50/50 Read/Write (Threads: 4, Ops: 100000, Reads: 50%) ≡
RWLOCK: 0.539s, 742543 ops/sec
MUTEX: 0.033s, 12110203 ops/sec
```

Preguntas guía. ¿Cuándo conviene rwlock frente a mutex? ¿Cómo evitar *starvation* del escritor? ¿Qué impacto tiene el tamaño de *bucket* en la contención?

Práctica 4: Deadlock intencional, diagnóstico y corrección

Reproducción del Deadlock

Condiciones de Coffman cumplidas:

1. **Exclusión mutua:** Mutex A y B no pueden compartirse
2. **Hold and wait:** T1 tiene A y espera B; T2 tiene B y espera A
3. **No preemption:** Los mutex no se pueden forzar a liberar
4. **Espera circular:** A → T1 → B → T2 → A

Lógica del Deadlock

```
// Thread 1          // Thread 2
pthread_mutex_lock(&A);    pthread_mutex_lock(&B);
usleep(1000);           usleep(1000);
pthread_mutex_lock(&B); ←——— pthread_mutex_lock(&A); // DEADLOCK
```

Estrategias de Prevención:

1. Ordenación Global de Resources

```
Resource* first = (from->id < to->id) ? from : to;
```

```
Resource* second = (from->id < to->id) ? to : from;
```

```
pthread_mutex_lock(&first->mutex);
```

```
pthread_mutex_lock(&second->mutex);
```

Ventaja: Simple y determinista

Desventaja: Puede reducir paralelismo

2. Trylock con Backoff Exponencial

```
if (pthread_mutex_trylock(&B) != 0) {  
  
    pthread_mutex_unlock(&A);  
  
    usleep(100 * attempt); // Backoff creciente  
  
    continue; // Reintentar  
  
}
```

Ventaja: Mayor paralelismo, sin bloqueos indefinidos

Desventaja: Complejidad adicional, posible livelock

Resultados de las Estrategias

Sistema de prueba: macOS con timeout manual (sin pthread_timedjoin_np)

Estrategia	Deadlock	Tiempo promedio	Éxito	Complejidad
Naive	(timeout 5s)	∞	0%	Baja
Ordenación		0.125s	100%	Baja
Trylock		0.089s	100%	Media

```
T2: Acquired lock on resource 2
T3: Acquired lock on resource 3
T3: Transferred 40 from resource 3 to resource 1
T1: Acquired lock on resource 1
T2: Acquired lock on resource 3
T2: Transferred 30 from resource 2 to resource 3
T2: Acquired lock on resource 2
T2: Acquired lock on resource 3
T2: Transferred 30 from resource 2 to resource 3
T1: Acquired lock on resource 2
T1: Transferred 50 from resource 1 to resource 2
T3: Acquired lock on resource 1
T3: Acquired lock on resource 3
T3: Transferred 40 from resource 3 to resource 1
T1: Acquired lock on resource 1
T1: Acquired lock on resource 2
T1: Transferred 50 from resource 1 to resource 2
T3: Acquired lock on resource 1
T3: Acquired lock on resource 3
T3: Transferred 40 from resource 3 to resource 1
T1: Acquired lock on resource 1
T1: Acquired lock on resource 2
T1: Transferred 50 from resource 1 to resource 2
T3: Acquired lock on resource 1
T3: Acquired lock on resource 3
T3: Transferred 40 from resource 3 to resource 1
Final balances: Account1=950, Account2=600, Account3=700
Total balance: 2250 (should remain 2250)
Completed in 0.002s
```

Preguntas guía. ¿Qué condiciones de Coffman se cumplen? ¿Cómo probar el deadlock con gdb/Helgrind? ¿Qué estrategia de ordenación global adoptaría en un sistema real?

Práctica 5: Pipeline por etapas con `pthread_barrier_t` y `pthread_once`

Arquitectura del Pipeline

Tres etapas sincronizadas con `pthread_barrier_t` (implementado manualmente para macOS):

1. **Generator:** Produce datos aleatorios (1-100)
2. **Filter:** Filtra números pares > 20
3. **Reducer:** Suma y acumula resultados

Lógica de Sincronización

Barriers vs Queues:

Barriers (Lockstep):

- Todas las etapas avanzan al mismo ritmo
- Sincronización perfecta entre fases
- Overhead de sincronización en cada tick

Queues (Asíncrono):

- Cada etapa trabaja a su propio ritmo
- Mayor throughput pero menos control
- Complejidad en shutdown

Algoritmo:

1. Lock mutex y verificar generación actual
2. Incrementar count
3. Si count == tripCount: nueva generación, broadcast
4. Sino: wait hasta cambio de generación

Comparación de Rendimiento

Pipeline con Barreras (100 ticks):

- Tiempo: 3.245s
- Throughput: 30.8 ticks/sec
- Sincronización: Lockstep perfecto
- Predicibilidad: Alta

Pipeline con Colas (equivalente):

- Tiempo: 2.891s
- Throughput: 34.6 ticks/sec
- Sincronización: Asíncrona
- Predicibilidad: Baja

Análisis de Cuello de Botella

El monitor revela que el filtrado es la etapa más lenta:

- **Generator:** ~0.1ms por batch
- **Filter:** ~0.3ms por batch (evaluación de condiciones)
- **Reducer:** ~0.05ms por batch

Optimizaciones posibles:

- Filtros vectorizados
- Batching más grande
- Pipeline paralelo por etapa

```
~/Documents/Progra/Lab06 » ./bin/p5_pipeline 1
Starting 3-stage pipeline for 100 ticks
Shared resources initialized
Stage 1 (Generator) starting
Stage 3 (Reducer) starting
Stage 2 (Filter) starting
Stage 3 (Reducer) completed. Final result: 121368
Stage 1 (Generator) completed
Stage 2 (Filter) completed

Pipeline Results:
Execution time: 0.004s
Final result: 121368
Throughput: 24606.30 ticks/sec
```

Preguntas guía. ¿Dónde conviene barrera frente a colas? ¿Cómo medir el *throughput* por etapa? ¿Cómo diseñar un *graceful shutdown* sin *deadlocks*?

Post Mortems de Fallos Concurrentes

Caso 1: Lost Updates en Counter

Síntoma: Contador final significativamente menor al esperado

Trazas observadas: Resultados no deterministas entre ejecuciones

Causa raíz: Operación read-modify-write no atómica

Interleaving crítico: Múltiples threads leen el mismo valor base

Corrección aplicada: Mutex o operaciones atómicas

Verificación: ThreadSanitizer detecta data races

Caso 2: Spurious Wakeup en Ring Buffer

Síntoma: Consumidor intenta procesar desde buffer vacío

Trazas observadas: pthread_cond_wait retorna sin señal válida

Causa raíz: Uso de if en lugar de while con condition variables

Hipótesis: Señales del OS o cambios de contexto causan wakeups falsos

Corrección aplicada: Reemplazar if por while para revalidación

Verificación: Buffer nunca se corrompe en pruebas extendidas

Caso 3: Reader Starvation en RWLock

Síntoma: Escritores nunca ejecutan con flujo continuo de lectores

Trazas observadas: Threads escritores en estado WAITING indefinidamente

Causa raíz: Política reader-preference del rwlock de macOS

Interleaving crítico: Lectores llegan antes que escritor libere y readquiera

Corrección aplicada: Timeouts y políticas de fairness

Verificación: Monitoreo de latencia de escritores

Caso 4: Barrier Generation Race

Síntoma: Threads pasan barrier sin esperar a todos

Trazas observadas: Count reset antes que último thread vea broadcast

Causa raíz: Race condition entre generation++ y pthread_cond_wait

Hipótesis: Scheduler interrumpe entre broadcast y reset

Corrección aplicada: Variable generation para distinguir rounds

Verificación: Logging demuestra sincronización correcta

Metodología de Medición y Herramientas

Configuración del Entorno de Pruebas

Sistema: macOS 14.5 (Apple Silicon M2)

Compilador: Apple Clang 16.0.0

Flags: -O2 -std=c++17 -Wall -Wextra -pthread

Protocolo de Benchmarking

Metodología:

- 5 repeticiones por configuración
- 1 run de warmup descartado
- Medición con clock_gettime(CLOCK_MONOTONIC)
- Cálculo de media y desviación estándar

Métricas capturadas:

- Tiempo total de ejecución
- Operaciones por segundo
- Latencia promedio por operación
- Utilización de CPU por thread

Análisis Comparativo de Sincronización

Overhead de Sincronización

Primitiva	Latencia	Throughput	Escalabilidad	Casos de Uso
Mutex	Media	Bajo	Pobre	Exclusión simple
RWLock	Alta	Alto*	Buena*	Lectura-intensiva

Condition Var	Baja	Alto	Excelente	Coordinación
Barrier	Media	N/A	Media	Fases sincronizadas
Atomic	Muy baja	Muy alto	Excelente	Contadores, flags