



Pontificia Universidad
JAVERIANA
Colombia

Facultad de Ingeniería

Taller 03

Sincronización con Semáforos POSIX

Diego Alejandro Mendoza Cruz

Juan Diego Ariza Lopez

Asignatura: Sistemas Operativos

Docente: Dr. John Corredor, PhD

Fecha: 14 de noviembre de 2025

Tabla de contenidos

| | |
|--|---|
| Contenido | |
| Introducción | 3 |
| Objetivos del Taller | 3 |
| Actividad 1: Productor-Consumidor..... | 4 |
| Descripción del problema | 4 |
| Implementación | 4 |
| Análisis de Resultados | 5 |
| Actividad 2: Sincronización con Threads | 6 |
| Programa 1: Sistema Producer-Spooler | 6 |
| Programa 2: Búsqueda Paralela del Máximo | 8 |
| Conclusiones | 9 |

Introducción

La sincronización es un concepto fundamental en sistemas operativos, permitiendo que múltiples procesos o hilos coordinen su acceso a recursos compartidos de manera segura y eficiente.

Este taller indaga dos aproximaciones principales de sincronización en sistemas POSIX: Semáforos para comunicación entre procesos (IPC) y mutex con variables de condición para sincronización entre hilos.

Objetivos del Taller

1. Implementar el problema común productor-consumidor usando semáforos POSIX.
2. Utilizar memoria compartida para la comunicación entre procesos.
3. Sincronizar múltiples hilos usando mutex y variables de condición.
4. Analizar y comparar diferentes mecanismos de sincronización.
5. Desarrollar aplicaciones que sean robustas y eficientes.

Actividad 1: Productor-Consumidor

Descripción del problema

El problema productor-consumidor es un ejemplo muy común de sincronización donde un productor genera datos y los coloca en un buffer compartido, mientras que un consumidor recibe los datos almacenados en dicho buffer.

El desafío consiste en coordinar ambos procesos para evitar condiciones de carrera (race condition) y garantizar lo siguiente:

- El productor no escriba cuando el buffer se encuentre lleno.
- EL consumidor no lea cuando el buffer se encuentre vacío.
- No se produzca una corrupción de datos ni pérdida de información.

Mecanismos de IPC utilizados:

| Semáforo | Función | Valor Inicial |
|--------------------|-------------------------------------|------------------------|
| /vacío | Espacios disponibles en buffer | 5 |
| /lleno | Elementos disponibles para consumir | 0 |
| Memoria compartida | /memoria_compartida | Buffer de 5 posiciones |

Implementación

La solución se implementó en 3 archivos principales:

shared.h: Define la estructura de datos compartida y el tamaño del buffer.

producer.c: Crea los semáforos y la memoria compartida, luego produce 10 elementos a razón de uno por segundo. Utiliza `sem_wait(vacio)` para bloquear si el buffer está lleno y `sem_post(lleno)` para señalar elementos disponibles.

consumer.c: Abre los recursos existentes y consume 10 elementos a razón de uno cada 2 segundos. Utiliza `sem_wait(lleno)` para bloquear si el buffer está vacío y `sem_post(vacio)` para señalar espacios liberados.

Código Clave – Productor:

```
// Esperar si el buffer está lleno (sem_wait decrementa el semáforo)
sem_wait(vacio);

// Agregar elemento al buffer
compartir->bus[compartir->entrada] = i;
printf("Productor: Produce %d en posición %d\n", i, compartir->entrada);

// Avanzar el índice de entrada
compartir->entrada = (compartir->entrada + 1) % BUFFER;

// Señalar que hay un elemento más lleno
sem_post(lleno);
```

Análisis de Resultados

Ejecución Exitosa:

Los programas se ejecutaron correctamente, demostrando una sincronización perfecta entre productor y consumidor. El buffer circular funcionó como esperado, permitiendo que el productor continuara generando datos mientras el consumidor procesaba a su propio ritmo.

```
estudiante@NGEN304:~/Taller_Posix_Sincronizacion/actividad1$ ./producer
== PRODUCTOR INICIADO ==
Producido 10 elementos...

Productor: Produce 1 en posición 0
Productor: Produce 2 en posición 1
Productor: Produce 3 en posición 2
Productor: Produce 4 en posición 3
Productor: Produce 5 en posición 4
Productor: Produce 6 en posición 0
Productor: Produce 7 en posición 1
Productor: Produce 8 en posición 2
Productor: Produce 9 en posición 3
Productor: Produce 10 en posición 4

== PRODUCTOR FINALIZADO ==
```

```
estudiante@NGEN304:~/Taller_Posix_Sincronizacion/actividad1$ ./consumer
== CONSUMIDOR INICIADO ==
Consumiendo 10 elementos...

Consumidor: Consumo 1 de posición 0
Consumidor: Consumo 2 de posición 1
Consumidor: Consumo 3 de posición 2
Consumidor: Consumo 4 de posición 3
Consumidor: Consumo 5 de posición 4
Consumidor: Consumo 6 de posición 0
Consumidor: Consumo 7 de posición 1
Consumidor: Consumo 8 de posición 2
Consumidor: Consumo 9 de posición 3
Consumidor: Consumo 10 de posición 4

== CONSUMIDOR FINALIZADO ==
```

Observaciones clave:

- El productor genera elementos más rápido que el consumidor.
- El buffer se llena progresivamente hasta bloquear al productor.
- No se observaron condiciones de carrera ni pérdida de datos.
- Los índices de entrada y salida funcionaron correctamente.
- La sincronización mediante semáforos fue precisa y eficiente.

Actividad 2: Sincronización con Threads

Programa 1: Sistema Producer-Spooler

Este programa implementa un sistema donde múltiples hilos productores generan mensajes que son impresos por un único hilo spooler. Es un ejemplo de patrón productor-consumidor pero usando hilos en lugar de procesos.

Arquitectura:

- 10 hilos productores, cada uno genera 10 mensajes.
- 1 hilo spooler que imprime todos los mensajes.
- Buffer circular compartido de 5 posiciones.
- Sincronización mediante mutex y variables de condición.

Primitivas de sincronización:

| Primitiva | Propósito |
|---------------------------|--|
| pthread_mutex_t buf_mutex | Protege acceso al buffer compartido |
| pthread_cond_t buf_cond | Señala cuando hay buffers disponibles |
| pthread_cond_t spool_cond | Señala cuando hay líneas para imprimir |

Uso de variables de condición:

Las variables de condición permiten que los hilos esperen eficientemente sin consumir CPU. Se ve así:

```
pthread_mutex_lock(&buf_mutex))
```

```
while (!buffers_available) {
    pthread_cond_wait(&buf_cond, &buf_mutex);
}
```

```
// Sección crítica: escribir en el buffer
int j = buffer_index;
buffer_index++;
if (buffer_index == MAX_BUFFERS) {
    buffer_index = 0; // Buffer circular
}
buffers_available--;
```

```
pthread_cond_signal(&spool_cond);
```

```
pthread_mutex_unlock(&buf_mutex))
```

Resultados:

El programa ejecutó exitosamente, generando e imprimiendo 100 mensajes en aproximadamente 10 segundos. Los hilos productores trabajaron en paralelo, y el spooler procesó todos los mensajes sin pérdidas ni duplicaciones. No se observaron condiciones de carrera gracias a la correcta utilización del mutex.

Múltiples hilos generando mensajes en paralelo

Continuación de mensajes intercalados

Todos los productores terminaron correctamente

```
== SISTEMA PRODUCTOR-SPOOLER INICIADO ==
Buffers disponibles: 5
Número de productores: 10

Thread 0: mensaje 1
Thread 1: mensaje 1
Thread 2: mensaje 1
Thread 3: mensaje 1
Thread 4: mensaje 1
Thread 5: mensaje 1
Thread 6: mensaje 1
Thread 7: mensaje 1
Thread 8: mensaje 1
Thread 9: mensaje 1
Thread 0: mensaje 2
Thread 1: mensaje 2
Thread 2: mensaje 2
Thread 3: mensaje 2
Thread 4: mensaje 2
Thread 5: mensaje 2
Thread 6: mensaje 2
Thread 7: mensaje 2
Thread 8: mensaje 2
Thread 9: mensaje 2
Thread 0: mensaje 3
Thread 1: mensaje 3
Thread 2: mensaje 3
Thread 4: mensaje 3
Thread 5: mensaje 3
Thread 3: mensaje 3
Thread 7: mensaje 3
Thread 8: mensaje 3
Thread 6: mensaje 3
Thread 9: mensaje 3
Thread 0: mensaje 4
Thread 1: mensaje 4
Thread 2: mensaje 4
Thread 4: mensaje 4
Thread 5: mensaje 4
Thread 3: mensaje 4
Thread 7: mensaje 4
Thread 8: mensaje 4
Thread 6: mensaje 4
Thread 9: mensaje 4
Thread 0: mensaje 5
Thread 2: mensaje 5
Thread 1: mensaje 5
Thread 4: mensaje 5
```

```
Thread 5: mensaje 5
Thread 3: mensaje 5
Thread 7: mensaje 5
Thread 6: mensaje 5
Thread 8: mensaje 5
Thread 9: mensaje 5
Thread 0: mensaje 6
Thread 1: mensaje 6
Thread 4: mensaje 6
Thread 2: mensaje 6
Thread 5: mensaje 6
Thread 7: mensaje 6
Thread 8: mensaje 6
Thread 9: mensaje 6
Thread 0: mensaje 7
Thread 1: mensaje 7
Thread 5: mensaje 7
Thread 6: mensaje 7
Thread 7: mensaje 7
Thread 2: mensaje 7
Thread 4: mensaje 7
Thread 3: mensaje 7
Thread 8: mensaje 7
Thread 9: mensaje 7
Thread 0: mensaje 8
Thread 1: mensaje 8
Thread 5: mensaje 8
Thread 6: mensaje 8
Thread 7: mensaje 8
Thread 2: mensaje 8
Thread 4: mensaje 8
Thread 3: mensaje 8
Thread 8: mensaje 8
Thread 9: mensaje 8
Thread 0: mensaje 9
Thread 1: mensaje 9
Thread 5: mensaje 9
Thread 2: mensaje 9
Thread 6: mensaje 9
Thread 4: mensaje 9
Thread 8: mensaje 9
Thread 7: mensaje 9
Thread 9: mensaje 9
Thread 3: mensaje 9
Thread 0: mensaje 10
Thread 1: mensaje 10
```

```
Thread 0: mensaje 10
Thread 1: mensaje 10
Thread 5: mensaje 10
Thread 4: mensaje 10
Thread 6: mensaje 10
Thread 2: mensaje 10
Thread 8: mensaje 10
Thread 7: mensaje 10
Thread 9: mensaje 10
Thread 3: mensaje 10
[Productor 0 terminó]
[Productor 5 terminó]
[Productor 1 terminó]
[Productor 4 terminó]
[Productor 6 terminó]
[Productor 8 terminó]
[Productor 7 terminó]
[Productor 9 terminó]
[Productor 2 terminó]
[Productor 3 terminó]

--- Todos los productores han terminado ---
--- Todas las líneas han sido impresas ---
```

Programa 2: Búsqueda Paralela del Máximo

Este programa demuestra paralelización mediante el patrón Map-Reduce. El objetivo es encontrar el valor máximo en un vector utilizando múltiples hilos para acelerar el proceso.

Estrategias:

1. **Dividir:** El vector se divide en N segmentos (uno por hilo).
2. **Map:** Cada hilo encuentra el máximo en su segmento.
3. **Reduce:** El hilo principal combina los máximos parciales.

Ejemplos de Ejecución:

| Hilo | Rango | Máximo Parcial |
|------|---------|----------------|
| 0 | [0,5) | 87 |
| 1 | [5,10) | 98 |
| 2 | [10,15) | 91 |
| 3 | [15,20) | 82 |
| | | 98 |

```
estudiante@NGEN304:~/Taller_Posix_Sincronizacion/actividad2$ ./concurrency datos.txt 4
== BÚSQUEDA DE MÁXIMO CON 4 HILOS ==
Vector de tamaño: 20
Elementos por hilo: 5

Hilo 0: buscando en rango [0, 5)
Hilo 1: buscando en rango [5, 10)
Hilo 2: buscando en rango [10, 15)
Hilo 3: buscando en rango [15, 20)

--- Resultados parciales ---
Hilo 0: máximo parcial = 87
Hilo 1: máximo parcial = 98
Hilo 2: máximo parcial = 91
Hilo 3: máximo parcial = 82

== RESULTADO FINAL ==
Máximo: 98
```

Se puede observar que al momento de ejecutar el programa nos da sus resultados parciales por cada segmento y al final determina cual es el resultado final, el cual es el número máximo entre esos segmentos.

Conclusiones

1. **Sincronización Efectiva:** Ambos mecanismos proporcionan sincronización robusta y eficiente. La elección depende del contexto: procesos independientes vs hilos dentro de un proceso.
2. **Problema Productor-Consumidor:** El problema clásico se resolvió exitosamente usando semáforos POSIX, demostrando cómo coordinar procesos que operan a diferentes velocidades sin pérdida de datos y de una forma eficiente.
3. **Memoria Compartida:** El uso de memoria compartida entre procesos mediante `shm_open` y `mmap` es fundamental para IPC, aunque requiere sincronización explícita mediante semáforos.
4. **Variables de Condición:** Las variables de condición nos permite que los hilos esperen eficientemente sin consumir CPU, mejorando el rendimiento del sistema.
5. **Paralelización:** El programa de búsqueda paralela demostró el patrón Map-Reduce, mostrando cómo dividir problemas para aprovechar múltiples núcleos y mejorar el rendimiento.
6. **Prevención de Race Conditions:** El uso correcto de mutex y semáforos es esencial para prevenir condiciones de carrera y garantizar la integridad de datos compartidos en sistemas concurrentes.
7. **Buffer Circular:** La implementación de buffers circulares es una técnica eficiente para gestionar productores y consumidores con diferentes velocidades de operación.
8. **Aplicación Práctica:** Los conceptos aprendidos son fundamentales en sistemas operativos modernos, bases de datos, servidores web y cualquier sistema que requiera procesamiento concurrente.