Taller 3 – Posix Semáforos

Juan José Ballesteros Suarez

Juan Diego Rojas Osorio

PROFESOR:

John Corredor, PhD.

Pontificia Universidad Javeriana

Facultad Ingeniería

Ingeniería de Sistemas

Sistemas Operativos

Bogotá D.C.

1. Objetivo 3
2. Marco Teórico 3
Sincronización en Sistemas Operativos 3
Semáforos POSIX 3
Hilos POSIX (pthreads) 4
Problema Productor – Consumidor 4
3. Desarrollo del Programa 4
4. Archivos de prueba y resultados 5
5. Análisis 8
6. Conclusiones 8
7. Referencias 9

1. Objetivo

El objetivo principal de este taller es implementar mecanismos de sincronización y comunicación entre procesos haciendo uso de semáforos POSIX y mutex para evitar la condición de carrera, aplicando esto a un problema clásico como lo es productor_consumidor.

Los objetivos específicos de este taller son:

- Diseñar e implementar un sistema productor–consumidor mediante procesos, empleando semáforos POSIX con nombre para la sincronización y memoria compartida para el intercambio eficiente de datos.

- Desarrollar un sistema multihilo basado en pthreads para calcular concurrentemente el valor máximo de un vector, analizando y aplicando correctamente mutex y variables de condición para asegurar la exclusión mutua, la coordinación entre hilos y documentando las estructuras de datos y mecanismos de sincronización empleados.

2. Marco Teórico

Sincronización en Sistemas Operativos

La sincronización es esencial en entornos concurrentes para evitar condiciones de carrera, garantizar la consistencia de los datos y coordinar el acceso a recursos compartidos. Se usa para evitar el deadlock el cual consiste en que dos procesos intentan acceder al mismo recurso sin tener un orden o un turno en específico, causando que ninguno termine usando dicho recurso.

Semáforos POSIX

Los semáforos con nombre buscan sincronizar procesos diferentes (no solo hilos) que pueden estar incluso en programas separados, y por otro lado están los semáforos sin nombre los cuales se usan para sincronizar hilos, pero dentro de un mismo proceso.

Las operaciones básicas con semáforos que usamos fueron:

- sem_open: Abre o crea un semáforo.
- sem_wait: Decrementa el semáforo (espera si es cero).
- sem_post: Incrementa el semáforo.
- sem_close y sem_unlink: Cierran y eliminan semáforos.

Hilos POSIX (pthreads)

Los hilos POSIX (pthreads) permiten la ejecución concurrente dentro de un mismo proceso, compartiendo el mismo espacio de memoria global, lo que facilita la comunicación y el acceso conjunto a datos. Para evitar condiciones de carrera en este entorno compartido, se hace uso de mutex para garantizar exclusión mutua al proteger las secciones críticas, y variables de condición, que permiten que los hilos esperen y se despierten en función de condiciones específicas, coordinando su ejecución de manera eficiente.

Problema Productor – Consumidor

El problema productor–consumidor describe una situación clásica de sincronización donde un productor se encarga de generar datos y colocarlos en un búfer compartido, mientras que un consumidor extrae esos datos para procesarlos. Debido a que ambos procesos acceden al mismo búfer, es necesario establecer mecanismos de sincronización que garanticen un acceso ordenado y seguro, evitando condiciones de carrera.

Para lograr solucionar este problema se hizo uso de:

- Semáforo vacío: El cual indica cuántos espacios libres quedan en el búfer; el productor debe esperar si el búfer está lleno.
- Semáforo lleno: El cual indica cuántos elementos disponibles hay en el búfer; el consumidor debe esperar si el búfer está vacío.

3. Desarrollo del Programa

El taller se desarrollo en 3 actividades principales:

a. Productor – Consumidor con Semáforos POSIX
producer.c: Programa principal que implementa el proceso productor.
consumer.c: Programa principal que implementa el proceso consumidor.

El productor crea los semáforos /vacio (10) y /lleno (0), además de la memoria compartida que contiene un búfer. Genera números del 1 al 10 y los escribe en el búfer, utilizando sem_wait(vacio) para esperar si el búfer está lleno y sem_post(lleno) para indicar que un nuevo elemento ha sido producido.

Por otro lado, es consumidor abre la memoria compartida y los semáforos creados por el productor, lee los valores del búfer y usa sem_wait(lleno) para esperar si el búfer está vacío y sem_post(vacio) para indicar que un espacio ha sido liberado tras consumir. Los semáforos coordinan la ejecución evitando desbordamientos o lecturas sin datos.

b. Búsqueda Concurrente del Máximo

concurrenciaPosix.c: Programa principal que coordina la búsqueda concurrente
maximo.h: Fichero con las declaraciones de estructuras y funciones.
maximo.c: Implementación de las funciones para búsqueda paralela.

El programa principal (concurrenciaPosix.c) lee los parámetros de entrada archivo de datos y número de hilos, carga el vector desde el archivo y luego invoca a maximoValor() para ejecutar la búsqueda del valor máximo de manera paralela.

Por otro lado, la función maximoValor() divide el vector en segmentos según la cantidad de hilos solicitados, crea los hilos asociados y asigna a cada uno la tarea de ejecutar buscarMax() en su porción correspondiente. Cuando todos los hilos terminan, combina los máximos parciales para obtener el máximo global.

En la función buscarMax(), cada hilo recorre su segmento del vector y determina el valor máximo local, almacenándolo en la estructura param_H. Esta ejecución paralela permite que múltiples hilos analicen simultáneamente diferentes partes del vector, incrementando el rendimiento del proceso de búsqueda.

c. **Productor-Consumidor Multihilo**
posixSincro.c: Programa principal que crea y coordina los hilos.
buffer.h: Cabecera con declaraciones de variables globales y funciones.
buffer.c: Implementación de las funciones productor y consumidor.

El programa principal (posixSincro.c) crea diez hilos productores y un hilo consumidor, espera a que todos los productores finalicen y luego cancela al consumidor cuando ya no quedan líneas por imprimir. Cada productor, implementado en producer() dentro de buffer.c, genera diez mensajes y utiliza un mutex junto con variables de condición para acceder de forma segura al buffer, esperando con pthread_cond_wait() cuando no hay espacio disponible. El consumidor, implementado imprime los mensajes del buffer y también espera con pthread_cond_wait() cuando no existen líneas por procesar, notificando a los productores cuando se libera espacio. En todo el sistema, el mutex garantiza la exclusión mutua sobre el buffer compartido, mientras que las variables de condición coordinan la producción y el consumo.

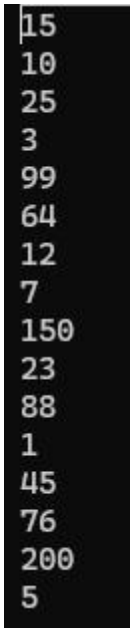
4. Archivos de prueba y resultados

Se realizaron 3 pruebas:

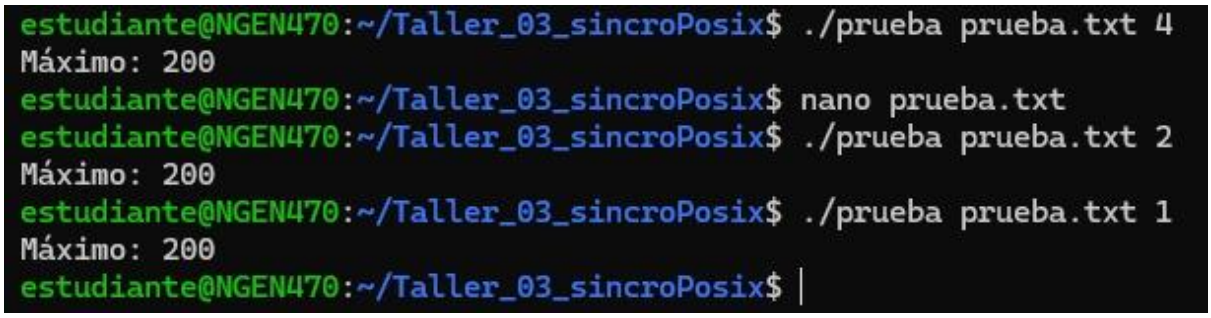
La primera de ellas para evidenciar el funcionamiento del Productor – Consumer, en la cual se evidencia el correcto uso de los mecanismos de sincronización y se puede ver como no se sobre escribe ni existe Deadlock. Observando que desde dos terminales se logra solucionar el

problema de Productor – Consumer.

La segunda prueba fue de la segunda actividad donde se le coloco un archivo prueba.txt de entrada:



Luego de su ejecución se observa la salida con 4, 2 y 1 hilo que el resultado siempre es el mismo sin depender de la cantidad de hilos que se pueden usar, de este modo probamos su consistencia.



La tercera prueba correspondió a la ejecución del sistema productor-consumidor multihilo implementado con pthreads. Durante la ejecución se observó que los diez hilos productores generaron correctamente sus mensajes y el hilo consumidor los imprimió en orden.

```
estudiante@NGEN470:~/Taller_03_sincroPosix$ ./posixSincro
Thread 0: 1
Thread 5: 1
Thread 1: 1
Thread 2: 1
Thread 4: 1
Thread 3: 1
Thread 9: 1
Thread 8: 1
Thread 7: 1
Thread 6: 1
Thread 0: 2
Thread 5: 2
Thread 1: 2
Thread 3: 2
Thread 2: 2
Thread 4: 2
Thread 9: 2
Thread 8: 2
Thread 7: 2
Thread 6: 2
Thread 0: 3
Thread 1: 3
Thread 5: 3
Thread 3: 3
Thread 2: 3
Thread 4: 3
Thread 8: 3
Thread 7: 3
Thread 9: 3
Thread 6: 3
Thread 0: 4
Thread 1: 4
Thread 5: 4
Thread 3: 4
Thread 2: 4
Thread 4: 4
Thread 8: 4
Thread 7: 4
Thread 9: 4
Thread 6: 4
Thread 0: 5
Thread 2: 5
Thread 1: 5
Thread 3: 5
Thread 5: 5
Thread 4: 5
Thread 8: 5
Thread 9: 5
Thread 7: 5
Thread 6: 5
Thread 0: 6
Thread 2: 6
Thread 3: 6
Thread 1: 6
Thread 5: 6
Thread 4: 6
Thread 8: 6
Thread 9: 6
Thread 7: 6
Thread 6: 6
Thread 0: 7
Thread 1: 7
Thread 5: 7
Thread 3: 7
Thread 4: 7
Thread 2: 7
Thread 8: 7
Thread 9: 7
Thread 7: 7
Thread 6: 7
Thread 0: 8
Thread 1: 8
Thread 5: 8
Thread 4: 8
Thread 8: 8
Thread 2: 8
Thread 3: 8
Thread 6: 8
Thread 7: 8
Thread 9: 8
Thread 0: 9
Thread 1: 9
Thread 5: 9
Thread 4: 9
Thread 8: 9
Thread 2: 9
Thread 3: 9
Thread 6: 9
Thread 7: 9
Thread 9: 9
Thread 0: 10
Thread 1: 10
Thread 5: 10
Thread 4: 10
Thread 8: 10
Thread 2: 10
Thread 3: 10
Thread 6: 10
Thread 7: 10
Thread 9: 10
```

Sin embargo, se detectó una condición de carrera en el acceso al buffer compartido, evidenciada por la aparición de valores duplicados en algunos hilos y secuencias inconsistentes en los contadores.

5. Análisis

Actividad 1: El desarrollo de esta actividad demostró la efectividad de los semáforos con nombre para sincronizar procesos independientes. Se logró una comunicación fluida entre productor y consumidor mediante el buffer circular en memoria compartida, donde los semáforos /vacio y /lleno garantizaron que nunca se accediera simultáneamente al recurso compartido. La implementación evitó condiciones de carrera y deadlocks, cumpliendo con los requisitos de sincronización del problema clásico productor-consumidor.

Actividad 2: Se evidenció el beneficio del paralelismo real mediante el uso de hilos POSIX. La división del vector en segmentos permitió que múltiples hilos procesaran concurrentemente diferentes secciones del arreglo, obteniendo el máximo global de manera eficiente. Se comprobó la consistencia de los resultados independientemente del número de hilos utilizados, demostrando la correcta implementación de la sincronización y la ausencia de condiciones de carrera en el cálculo de máximos parciales.

Actividad 3: La implementación multihilo reveló la complejidad de la sincronización en entornos de alta concurrencia. Aunque el sistema funcionó correctamente en términos de completitud y ausencia de deadlocks, se identificó una condición de carrera en el manejo del buffer compartido. Esto destacó la importancia de un diseño cuidadoso de las secciones críticas y la necesidad de expandir la protección del mutex para cubrir operaciones completas de escritura, asegurando la consistencia en el acceso concurrente a estructuras de datos compartidas.

6. Conclusiones

a. **Mecanismo de sincronización:**
   La implementación demostró que mutex y semáforos POSIX son herramientas eficaces para resolver problemas de concurrencia. En la Actividad 1, los semáforos con nombre garantizaron que el productor y consumidor nunca accedieran al mismo tiempo al buffer compartido evitando así el Deadlock, mientras que en la Actividad 3, la combinación de mutex y variables de condición permitió una coordinación precisa entre múltiples hilos.

b. **Procesos VS Hilos:**
   La diferencia entre el manejo de memoria mediante los procesos (Actividad 1) requirieron memoria compartida explícita y semáforos con nombre para comunicarse, mientras que los hilos compartieron automáticamente el espacio

de direcciones, simplificando el acceso a datos, pero exigiendo mayor cuidado en la sincronización.

c. **Paralelismo y Concurrencia:**
Se evidencio el beneficio del paralelismo real al dividir la búsqueda del máximo entre múltiples hilos, se logró un procesamiento más rápido en sistemas multiprocesador. Sin embargo, también se identificó el overhead de la creación y unión de hilos, que en problemas muy pequeños podría no justificar su uso como también se pudo observar en el taller de rendimiento.

7. Referencias

Semaphore in C | Synchronization. (s. f.). Synchronization. https://eric--lo-gitbook-io.translate.goog/synchronization/semaphore-in-c?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc&_x_tr_hist=true

GeeksforGeeks. (2025b, septiembre 3). Producer Consumer Solution using Semaphores. GeeksforGeeks. https://www-geeksforgeeks-org.translate.goog/operating-systems/producer-consumer-problem-using-semaphores-set-1/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

Ippolito, G. (s. f.). Linux tutorial: POSIX threads. https://www-cs-cmu-edu.translate.goog/afs/cs/academic/class/15492-f07/www/pthreads.html?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc