Facultad de Ingeniería
Departamento de Ingeniería de Sistemas

Taller POSIX Sincronización

Noviembre 2025

Integrantes:

Danna Rojas Bernal
María Fernanda Velandia Gracia
Christian Becerra Enciso
Giovanny Andrés Durán Rentería

Asignatura: Sistemas Operativos
Docente: John Corredor Franco

Bogotá D.C., Colombia
15 de noviembre de 2025

Índice

1. Introducción

Se desarrollaron dos ejercicios relacionados con concurrencia y sincronización con el objetivo de entender los procesos, hilos y los mecanismos de sincronización, en este caso usando la interfaz POSIX.

La primera parte se enfocó en desarrollar el modelo clásico de productor-consumidor utilizando semáforos POSIX con nombres y memoria compartida, de tal manera que dos procesos independientes pudieran coordinarse sin generar condición de carrera.

La segunda parte del trabajo fue la sincronización entre hilos dentro de un mismo proceso. En este caso se desarrolló con `pthreads`, `mutex` y variables de condición, lo anterior con un programa que calcula el máximo de un vector usando varios hilos y luego en un ejemplo de varios productores y un hilo que era el encargado de imprimir las líneas en un punto que se comparte.

2. Marco teórico

2.1. Concurrencia, procesos y hilos

La concurrencia aparece cuando varias tareas parecen ejecutarse al mismo tiempo o se interponen entre sí, ya sea en uno o varios núcleos de CPU [1], además y como se abordó en las clases, la concurrencia puede ser de dos tipos explícita, que es cuando el programador define directamente los hilos o programas o implícita, que es cuando el mismo sistema maneja la concurrencia en automático. Por su parte, un proceso es una instancia de un programa en ejecución, con su propio espacio de direcciones, archivos abiertos y contexto de ejecución, mientras que un hilo es la unidad de ejecución más ligera que comparte el espacio de memoria con otros hilos del mismo proceso [1], [3].

2.2. Sincronización y secciones críticas

Ahora cuando varios hilos o procesos comparten datos, puede suceder que dos de ellos intenten leer y escribir sobre la misma variable al mismo tiempo esto puede generar lo que se conoce como condiciones de carrera y resultados imprecisos [1], [2]. Entonces para evitar lo anterior se usan mecanismos de sincronización que controlan el acceso a las secciones críticas, que básicamente son los fragmentos de código donde se accede a recursos compartidos.

2.3. Semáforos POSIX con nombre

Los semáforos son uno de los mecanismos clásicos para sincronizar procesos y un semáforo es básicamente una variable entera protegida que solo se puede modificar a través de operaciones atómicas de espera (wait) y señalización (post) [2]. Ahora bien, en POSIX existen semáforos sin nombre los cuales se usan dentro de un mismo proceso o entre procesos relacionados, también hay semáforos con nombre (named semaphores), que se identifican por una cadena y permanecen en el sistema hasta que se eliminen, estos últimos permiten que procesos no relacionados puedan sincronizar si conocen el mismo nombre del semáforo [4].

2.4. Pthreads, mutex y variables de condición

Entonces dentro de un mismo proceso, y a través de la librería de `pthreads` (POSIX threads) podemos crear múltiples hilos que comparten memoria y otra información del proceso. Cada hilo tiene su propia pila y contador de programa, pero puede acceder al mismo conjunto de datos globales [3]. Ahora para proteger secciones críticas se utilizan `mutex`, que ayuda a garantizar que solo un hilo pueda ejecutar cierto código a la vez. Además, las variables de condición se usan para que un hilo pueda esperar a que se cumpla una condición mientras libera el `mutex`, y otro hilo pueda ayudar a "despertar" a los demás hilos con funciones como `pthread_cond_signal` o `pthread_cond_broadcast`.

3. Actividad 1

3.1. Descripción general

En la primera actividad se implementó el problema productor–consumidor usando semáforos POSIX con nombre y memoria compartida, aquí la idea general es que el productor escribe elementos y el consumidor los va leyendo en orden y, para así poder evitar que el productor escriba cuando está lleno o que el consumidor lea cuando está vacío, se utilizan dos semáforos: vacío (lleva la cuenta de los espacios disponibles) y lleno (lleva la cuenta de los espacios ocupados).

3.2. Funcionamiento del productor

Aquí lo que se hace es que, dentro de un ciclo, el productor va creando un grupo de enteros. Y antes de escribir un nuevo elemento en el búfer, ejecuta `sem_wait(vacío)` para decrementar el contador de espacios libres; entonces, si el búfer está lleno, el productor se bloquea hasta que el consumidor libere un espacio. Ya luego escribe el dato en `bus` y se actualiza el índice de entrada `(entrada + 1) % BUFFER` para mantener el ciclo y que quede en el inicio. Ya al final, llama a `sem_post(lleno)` para indicar que ahora hay un entero más disponible para el consumidor.

3.3. Funcionamiento del consumidor

Ahora el consumidor lo que realiza es el proceso complementario. Básicamente, en su ciclo principal primero ejecuta `sem_wait(lleno)` para asegurarse de que haya al menos un elemento disponible y, si esto se cumple es cero, entonces el consumidor se bloquea hasta que el productor agregue algo. Luego lee el valor de `bus`, imprime un mensaje y actualiza el índice de salida `(salida + 1) % BUFFER`. Por último, llama a `sem_post(vacío)` para aumentar el contador de espacios libres.

4. Actividad 2

La actividad consiste en comprender cómo funcionan los subprocesos POSIX (pthreads) dentro de un programa y cómo se manejan los problemas de sincronización cuando varios hilos comparten datos. Los pthreads permiten ejecutar varios flujos de trabajo dentro del mismo proceso, compartiendo las variables globales. Por esta razón, es necesario aplicar mecanismos de sincronización —como exclusión mutua con mutex— para evitar

que dos hilos accedan o modifiquen datos compartidos al mismo tiempo. El objetivo central es identificar el archivo posixSincro.c, analizar su funcionamiento y entender cómo implementa estas técnicas de sincronización. La actividad esta modularizada por 4 ficheros y adicionalmente un fichero para la automatización de la ejecucion del programa y otro tipo .txt para las pruebas del programa:

- concurrenciaPosix.c:
- concurrenciaPosix.h:
- posixSincro.c:
- posixSincro.h:
- Makefile:
- prueba.txt:

4.1. ConcurrenciaPosix

El programa comienza leyendo desde un archivo la cantidad de elementos y los valores del vector. Luego, el usuario indica cuántos hilos quiere usar, y el programa divide el vector en partes iguales para asignar un segmento a cada hilo. Después de esto, cada hilo inicia su ejecución y recorre únicamente su porción del vector para encontrar el máximo parcial dentro de su sección. Cada hilo guarda su resultado en una posición distinta de la estructura compartida, evitando conflictos.

Una vez todos los hilos han terminado su búsqueda, el hilo principal los espera mediante pthread-join. Cuando todos han finalizado, el programa toma los valores máximos parciales generados por cada hilo y realiza una reducción, comparándolos para obtener el máximo global del vector completo. Finalmente, el programa imprime en pantalla el resultado. Todo el flujo ilustra cómo dividir una tarea grande en subtareas independientes permite acelerar la ejecución usando paralelismo simple.

4.2. posixSincro

El programa inicia creando varios hilos productores y un único hilo spooler. También configura un buffer compartido que tiene un número limitado de espacios. Cada productor comienza intentando escribir un mensaje en este buffer; si no hay espacio disponible, el productor se queda esperando mediante una variable de condición hasta que pueda continuar. Cuando hay espacio, toma el mutex, escribe su mensaje y luego libera el mutex, avisando al spooler que hay un nuevo mensaje para consumir.

Mientras tanto, el spooler está en ciclo continuo revisando si el buffer contiene algo. Si está vacío, el spooler se bloquea y espera una señal de los productores. Cuando un productor genera un mensaje, el spooler despierta, toma el mutex, extrae el mensaje y lo imprime. Luego libera el mutex y notifica a los productores que se ha liberado un espacio en el buffer. Este proceso se repite hasta que todos los mensajes han sido procesados. El flujo evidencia cómo la exclusión mutua y las variables de condición permiten que varios hilos compartan recursos sin interferencias, evitando condiciones de carrera y garantizando un acceso ordenado.

5. Análisis y resultados

En la actividad 1 el productor inicia creando los semáforos y la memoria compartida, establece los índices y comienza a generar valores: antes de escribir cada elemento espera que haya espacio disponible mediante sem-wait(vacio), coloca el dato en la posición de entrada, avanza el índice y luego avisa al consumidor con sem-post(lleno). Mientras tanto, el consumidor abre esos mismos recursos, inicia su índice de salida y en cada iteración espera a que el productor haya colocado un elemento usando sem- -wait(lleno); después lo lee desde el buffer, avanza su posición y libera un espacio con sem-post(vacio) para permitir que el productor siga escribiendo.

```
chris@Chrisbe:/mnt/c/Users/chris/Downloads/Sincronización/Sincronización/Actividad_1$ ./productor
Productor: Produce 1
Productor: Produce 2
Productor: Produce 3
Productor: Produce 4
Productor: Produce 5
^C
chris@Chrisbe:/mnt/c/Users/chris/Downloads/Sincronización/Sincronización/Actividad_1$ ./consumidor
Consumidor: Consume 1
Consumidor: Consume 2
Consumidor: Consume 3
Consumidor: Consume 4
Consumidor: Consume 5
```

Figura 1: Actividad 1 - Productor y consumidor

En la actividad 2 el programa concurrenciaPosix recibe como entrada un archivo con una lista de números (prueba.txt) y un número de hilos a utilizar. Internamente, el programa divide el vector de datos en 4 segmentos (numero de hilos) y asigna cada uno a un hilo para que calcule un máximo parcial. Cada hilo recorre únicamente su sección del arreglo y devuelve su mayor valor encontrado. Después de que todos los hilos terminan, el hilo principal compara los máximos parciales y obtiene el máximo global del archivo.

```
chris@Chrisbe:/mnt/c/Users/chris/Downloads/Sincronización/Sincronización/Actividad_2$ ./concurrenciaPosix prueba.txt 4
Máximo encontrado: 20
```

Figura 2: Actividad 2 concurrenciaPosix

En el fichero posixSincro cada productor genera 10 mensajes, pero antes de escribir en un buffer necesita asegurarse de que haya un espacio libre. Esto se logra usando un mutex que protege el arreglo de buffers y una variable de condición (buf-cond) que obliga al hilo a esperar cuando el buffer está lleno. Una vez que el productor escribe un mensaje en su posición correspondiente, incrementa el contador de líneas por imprimir y despierta al spooler mediante pthread-cond-signal().

```
chris@Chrisbe:/mnt/c/Users/chris/Downloads/Sincronización/Sincronización/Actividad_2$ ./posixSincro
Thread 1: 1
Thread 2: 1
Thread 3: 1
Thread 1: 2
Thread 2: 2
Thread 3: 2
Thread 1: 3
Thread 2: 3
Thread 3: 3
Thread 1: 4
Thread 2: 4
Thread 3: 4
Thread 1: 5
Thread 2: 5
Thread 3: 5
Thread 3: 6
Thread 1: 6
Thread 2: 6
Thread 3: 7
Thread 1: 7
Thread 2: 7
Thread 3: 8
Thread 1: 8
Thread 2: 8
Thread 3: 9
Thread 1: 9
Thread 2: 9
Thread 3: 10
Thread 1: 10
Thread 2: 10
```

Figura 3: Acitividad 2 -posixSincro

6. Conclusiones

Las dos actividades permitieron comprender de manera práctica los mecanismos fundamentales de sincronización en sistemas POSIX, abordando tanto la comunicación entre procesos como la coordinación entre hilos dentro de un mismo programa. En la primera actividad, el uso de semáforos mostro cómo procesos independientes pueden cooperar de forma ordenada compartiendo una variable, garantizando que el productor y el consumidor operen sin condiciones de carrera. En la segunda actividad, el trabajo con pthreads mostró la importancia del control sobre el acceso a memoria compartida, utilizando mutex y variables de condición para coordinar múltiples flujos de ejecución que comparten datos globales.

En conjunto, los ejercicios demuestran la forma en como aplicar concurrencia sin condiciones de carrera y de evitar las posibilidades de sobreescirbir los datos perdiendo coherencia en el código. Gracias a los semáforos y mutex que permiten evitar bloqueos, inconsistencias y comportamientos incoherentes en los procesos paralelos. Estos ejercicios permiten entender la necesidad de controlar el acceso a recursos compartidos y garantizar la correcta interacción entre múltiples unidades de ejecución.

7. Referencias

[1] A. Silberschatz, P. B. Galvin y G. Gagne, *Operating System Concepts*, 9th ed. Wiley, 2018.

[2] A. S. Tanenbaum y H. Bos, *Modern Operating Systems*, 4th ed. Pearson, 2015.

[3] B. Nichols, D. Buttlar y J. Farrell, *Pthreads Programming*. O'Reilly Media, 1996.

[4] W. Stallings, *Operating Systems: Internals and Design Principles*, 7th ed. Pearson, 2012.