Taller 03: Posix Sincronización

Juan David Daza Caro & David Tobar Artunduaga

Pontificia Universidad Javeriana

Facultad de Ingeniería

Sistemas Operativos

Bogotá, D.C.

14/11/2025

Resumen

El presente informe desarrolla un análisis profundo del problema clásico Productor–Consumidor, implementado mediante dos aproximaciones fundamentales en sistemas operativos: la sincronización entre procesos utilizando semáforos y memoria compartida POSIX, y la sincronización entre hilos mediante la biblioteca pthread, haciendo uso de mutex y variables de condición. El documento examina con detenimiento el funcionamiento interno de cada mecanismo, su relevancia teórica y práctica, así como sus implicaciones para el manejo seguro de recursos compartidos en entornos concurrentes. En la primera parte se aborda la interacción entre procesos mediante semáforos con nombre, enfatizando el uso de sem_wait, sem_post, shm_open, mmap y la administración de regiones de memoria compartida. Se describen sus ventajas, limitaciones y la forma en que estos mecanismos permiten implementar un buffer circular seguro para la comunicación interprocesos. En la segunda parte se estudia una solución multihilo basada en mutex y variables de condición, destacando sus diferencias conceptuales y operativas frente a la sincronización basada en semáforos.
Adicionalmente, el informe integra un marco teórico amplio sobre la historia de la sincronización y la evolución de los estándares POSIX, un análisis de rendimiento y escalabilidad en los dos enfoques implementados, una discusión sobre seguridad, condiciones de carrera, deadlocks y starvation, así como una interpretación del funcionamiento del buffer desde la teoría de colas. El propósito final es demostrar la correcta aplicación de los mecanismos de sincronización del sistema operativo y su importancia en el diseño de software concurrente robusto y eficiente.

Palabras clave: sincronización, semáforos POSIX, memoria compartida, hilos POSIX, mutex, variables de condición, concurrencia.

Contenido

Introducción

La sincronización constituye uno de los pilares fundamentales dentro del estudio de los sistemas operativos y de las arquitecturas concurrentes. A medida que las aplicaciones modernas requieren mayor capacidad de procesamiento y la posibilidad de ejecutar múltiples tareas en paralelo, surge la necesidad de coordinar adecuadamente el acceso a los recursos compartidos para evitar inconsistencias, corrupciones de datos o comportamientos indeterminados. En este contexto, el problema Productor–Consumidor es un modelo ampliamente utilizado para ilustrar los retos y soluciones propios de la concurrencia, especialmente en entornos donde múltiples procesos o hilos interactúan simultáneamente sobre un conjunto limitado de recursos.

La esencia del problema radica en que uno o varios productores generan elementos y los depositan en un buffer compartido, mientras que uno o varios consumidores extraen dichos elementos para procesarlos. Este intercambio aparentemente simple se convierte en un desafío técnico crítico cuando se abordan los aspectos de sincronización: ¿qué ocurre si el consumidor intenta leer cuando el buffer está vacío?, ¿qué sucede si el productor quiere insertar cuando el buffer está lleno?, ¿cómo garantizar que dos procesos no accedan al mismo espacio al mismo tiempo?, ¿cómo evitar que los procesos entren en condiciones de carrera o bloqueos mutuos? Estas preguntas se responden mediante mecanismos especializados que los sistemas operativos proporcionan para regular el acceso concurrente.

En el marco de este taller se desarrollaron dos enfoques distintos para resolver el problema. El primero utiliza procesos independientes que coordinan sus acciones mediante semaforos POSIX con nombre y memoria compartida. Esta aproximación representa un caso realista de comunicación entre procesos (IPC), donde cada programa ejecuta su propio espacio de direcciones y, por tanto, debe confiar en primitivas del sistema operativo para compartir información e implementar exclusión mutua. La segunda aproximación utiliza hilos POSIX (pthread), donde la concurrencia se desarrolla dentro de un mismo proceso. En este caso no se necesita memoria compartida explícita, ya que todos los hilos comparten el mismo espacio de direcciones, pero sí se requiere la correcta utilización de bloqueos tipo mutex y variables de condición para garantizar un comportamiento seguro y ordenado.

Ambas soluciones persiguen el mismo objetivo, pero difieren en complejidad, rendimiento y nivel de abstracción. El estudio comparado de ambas alternativas no solo permite comprender los mecanismos técnicos detrás de la sincronización, sino que también revela la

Este informe pretende, en síntesis, demostrar la comprensión integral de los mecanismos de sincronización provistos por POSIX, su correcta aplicación práctica en el desarrollo de software concurrente y su relevancia en el diseño de sistemas operativos modernos. Con ello se busca articular en un solo documento tanto la teoría que sustenta la sincronización como la implementación concreta realizada durante el taller.

Marco Teórico

El marco teórico constituye la base conceptual necesaria para comprender e interpretar adecuadamente la implementación realizada en este taller. La sincronización, la exclusión mutua, el control de concurrencia y la comunicación entre procesos representan componentes esenciales en la construcción de sistemas operativos robustos. Por ello, esta sección presenta un recorrido amplio por los conceptos fundamentales que sustentan el problema Productor–Consumidor, así como su evolución histórica y formalización dentro de los estándares POSIX.

Historia de la Sincronización y Evolución del Estándar POSIX

Los primeros sistemas operativos trabajaban de forma estrictamente secuencial, lo que evitaba conflictos al acceder a la memoria o a los recursos compartidos. Sin embargo, con la introducción de la multiprogramación en la década de 1960, surgió la necesidad de ejecutar múltiples tareas "simultáneamente", primero en un único procesador mediante cambio de contexto, y más adelante en múltiples procesadores reales. Este entorno abrió la puerta a problemas hasta entonces inexistentes: condiciones de carrera, interbloqueos y accesos inconsistentes a memoria.

Edsger Dijkstra es reconocido como una de las figuras clave en el establecimiento formal de la sincronización con la introducción del concepto de semáforo en 1965. Su propuesta incorporaba dos operaciones fundamentales —originalmente llamadas *P* y *V*— que permitían a los procesos coordinar el acceso a recursos limitados. Este avance se convirtió en la piedra angular del control de concurrencia.

Con el paso del tiempo, las distintas variantes de UNIX comenzaron a incorporar implementaciones de semáforos, memoria compartida y señales, pero existían diferencias significativas entre ellas. Para unificar estas interfaces surgió en la década de los ochenta el estándar POSIX (Portable Operating System Interface), desarrollado principalmente por IEEE. POSIX no solo normalizó la interacción con el sistema operativo para programas escritos en C, sino que también estandarizó mecanismos críticos de concurrencia tales como:

- Semáforos POSIX clásicos (sem_t)
- Mutex y variables de condición (pthread_mutex_t, pthread_cond_t)
- Comunicación mediante memoria compartida (shm_open, mmap)

Gracias a POSIX, el código concurrente se volvió portable entre diferentes sistemas tipo UNIX, incluyendo Linux, BSD y macOS. Esto hizo posible el desarrollo de aplicaciones multiproceso y multihilo robustas, uniformes y reproducibles, facilitando la enseñanza formal de la concurrencia en cursos de sistemas operativos como el presente taller.

Concurrencia y Condiciones de Carrera

La concurrencia consiste en permitir la ejecución solapada de múltiples hilos o procesos, los cuales comparten ciertos recursos y datos. Aunque esto incrementa el rendimiento y la eficiencia, también introduce riesgos inherentes. Una condición de carrera ocurre cuando

dos o más entidades acceden simultáneamente a un mismo recurso, y el resultado final depende del orden en el que se ejecuten las operaciones. Este comportamiento es indeseable, pues genera inconsistencias, errores o corrupción de datos.
Para evitarlas, se deben proteger las secciones críticas, es decir, los fragmentos de código donde se accede o modifica información compartida. Sobre estas secciones se aplican mecanismos de sincronización que garantizan que solo un proceso o hilo pueda ejecutarlas a la vez.

Semáforos POSIX con Nombre

Los semáforos POSIX representan uno de los mecanismos más utilizados para la coordinación entre procesos. Un semáforo con nombre se almacena en el sistema operativo bajo un identificador accesible para múltiples procesos, lo que permite la sincronización en programas independientes. Las operaciones clave son:

- sem_open(): crea o abre un semáforo persistente.
- sem_wait(): operación que bloquea al proceso hasta que el semáforo tenga un valor positivo.
- sem_post(): incrementa el semáforo, desbloqueando procesos.
- sem_unlink(): elimina el semáforo del sistema.

En el caso de este taller, se implementaron dos semáforos fundamentales:

- /vacio, que indica cuántos espacios libres hay en el buffer.
- /lleno, que indica cuántos ítems están disponibles para consumir.

Estos semáforos garantizan la integridad del buffer circular incluso cuando productor y consumidor se ejecutan como procesos completamente separados.

Memoria Compartida POSIX

La memoria compartida constituye uno de los mecanismos de comunicación interprocesos (IPC) más eficientes, ya que permite que múltiples procesos accedan directamente a una misma región de memoria sin necesidad de copiar datos. POSIX define la siguiente secuencia para su uso adecuado:

1. Crear u abrir un objeto de memoria compartida con shm_open.
2. Ajustar su tamaño con ftruncate.
3. Mapearlo en el espacio de direcciones del proceso mediante mmap.
4. Acceder a la región mapeada como si fuera una dirección regular de memoria.
5. Desmapearlo con munmap y eliminarlo opcionalmente con shm_unlink.

En este taller, tanto el productor como el consumidor acceden al buffer compartido mediante este mecanismo, de forma que cualquier cambio realizado por el productor es inmediatamente visible para el consumidor y viceversa.

Mutex y Variables de Condición en Pthreads

Cuando la concurrencia se maneja dentro de un mismo proceso, se emplean hilos POSIX (pthread), los cuales comparten la misma memoria por defecto. Esto elimina la necesidad

de memoria compartida explícita, pero vuelve imprescindible el uso de mutex para garantizar exclusión mutua y de variables de condición para coordinar el despertar de hilos cuando ocurren ciertos eventos.
En la implementación del taller, los hilos productores esperan a que haya espacio disponible en el buffer, mientras el consumidor espera a que existan mensajes. Las variables de condición permiten suspender hilos de manera eficiente, evitando la espera activa y reduciendo el desperdicio de CPU.

El Buffer Circular como Estructura de Datos

El buffer circular es esencial para el funcionamiento del sistema Productor–Consumidor. Su ventaja radica en que permite insertar y extraer elementos de manera eficiente sin necesidad de desplazar datos. Mediante aritmética modular se controlan los índices:

- Índice de entrada (productor)
- Índice de salida (consumidor)

A diferencia de una cola lineal, el buffer circular utiliza el espacio de forma continua y garantiza un rendimiento constante, lo cual es especialmente relevante en sistemas de tiempo real o de alto tráfico.

El Buffer desde la Teoría de Colas

La teoría de colas permite modelar formalmente el comportamiento del buffer. En este caso, el buffer puede verse como una cola **M/M/1/K**, donde:

- Hay un solo servidor (el consumidor)
- Los productores actúan como fuentes de llegada
- El buffer tiene capacidad **K = BUFFER**

Este modelo permite estudiar parámetros como:

- tiempo de espera medio
- tasa de llegada vs. tasa de servicio
- probabilidad de bloqueo del productor
- uso del servidor (consumidor)

Este enfoque formal respalda matemáticamente las observaciones prácticas realizadas durante el taller.

Seguridad: Deadlocks, Starvation y Bloqueos Incorrectos

Sin mecanismos adecuados, la sincronización puede generar problemas graves:

- **Deadlock:** dos procesos esperan indefinidamente recursos del otro.
- **Starvation:** un proceso nunca obtiene acceso al recurso.
- **Bloqueo permanente:** el sistema deja de avanzar por mala señalización.
- **Inconsistencia:** errores por falta de exclusión mutua o sincronización incorrecta.

La implementación del taller evita estos problemas gracias al uso ordenado de semáforos y mutex junto con condiciones precisas.

Rendimiento y Escalabilidad

Los semáforos son adecuados para procesos independientes, pero implican llamadas al sistema más costosas. Los hilos, por otro lado, comparten memoria y son más ligeros, lo que mejora la escalabilidad cuando existen múltiples productores.
El análisis detallado revela:

- La solución con procesos es más aislada pero más costosa.
- La solución con hilos es más rápida, pero puede ser más vulnerable a errores de memoria.
- El buffer circular se comporta de manera predecible incluso bajo carga elevada.

Actividad 1: Implementación con Procesos POSIX y Semáforos con Nombre

Esta actividad tiene como objetivo implementar el problema Productor–Consumidor utilizando dos procesos independientes que comparten un buffer circular mediante memoria compartida, y regulan su acceso mediante semáforos POSIX con nombre. El enfoque seleccionado es representativo de una de las formas más directas y eficaces de implementar comunicación entre procesos en sistemas tipo UNIX.
A diferencia de la solución basada en hilos, la implementación con procesos requiere que la administración de memoria y de sincronización se realice explícitamente usando los servicios provistos por el sistema operativo. Esto implica que tanto el productor como el consumidor ejecutan en espacios de memoria completamente separados; por ende, la única forma de compartir datos de manera segura es mediante mecanismos explícitos como los semáforos POSIX y la memoria compartida mapeada con mmap().
A continuación, se presenta un análisis exhaustivo de cada uno de los componentes involucrados, su funcionamiento y las partes significativas que deben resaltarse en el informe.

Estructura compartida: shared.h

El archivo shared.h define la interfaz común entre el productor y el consumidor.
Contiene:

1. Los includes necesarios para sincronización y memoria compartida.
2. La constante BUFFER, que define el tamaño del buffer circular.
3. La estructura compartir_datos, que contiene:
    - un arreglo fijo de cinco enteros,
    - el índice de entrada (para el productor),
    - el índice de salida (para el consumidor).

Este diseño minimalista cumple su función con precisión, pues establece la estructura exacta que será mapeada en memoria compartida. Debido a que ambos procesos acceden a esta estructura al mismo tiempo, su definición debe ser idéntica para todos ellos, y shared.h garantiza precisamente eso.

La definición del buffer circular y sus índices constituye el núcleo lógico de la sincronización. La estructura define el espacio compartido donde ocurre la interacción controlada entre el productor y el consumidor.

Proceso productor: funciones, análisis y comportamiento

El archivo **producer.c** implementa el proceso que genera ítems y los inserta dentro del buffer compartido. Su comportamiento sigue un patrón conocido:

1. Inicializa (o abre) los semáforos necesarios.
2. Crea y mapea la memoria compartida.
3. Inserta elementos en el buffer de forma sincronizada.
4. Actualiza el índice de entrada usando aritmética modular.

5. Señaliza al consumidor mediante semáforos.
6. Libera y elimina recursos al finalizar.

A continuación, se explican en detalle las partes más importantes del código.

Creación de semáforos con nombre

El productor ejecuta:

```
sem_t *vacio = sem_open("/vacio", O_CREAT, 0644, BUFFER);
sem_t *lleno = sem_open("/lleno", O_CREAT, 0644, 0);
```

Estos semáforos POSIX con nombre son persistentes dentro del sistema. Los valores iniciales determinan la lógica del buffer:

- vacio = BUFFER: el buffer está completamente vacío, hay espacio disponible.
- lleno = 0: no hay elementos disponibles.

El uso de semáforos con nombre muestra claramente cómo se sincronizan procesos independientes que no comparten memoria automáticamente. Esta inicialización garantiza que no haya condiciones de carrera al inicio del programa.

Creación de la memoria compartida

El productor ejecuta:

```
int shm_fd = shm_open("/memoria_compartida", O_CREAT | O_RDWR, 0644);
ftruncate(shm_fd, sizeof(compartir_datos));
```

Aquí se establece:

- Un objeto de memoria en /dev/shm/, administrado por el kernel.
- Su tamaño exacto se ajusta para alojar la estructura compartida.

Luego se mapea con:

```
compartir_datos *compartir = mmap(NULL, sizeof(compartir_datos),
                                  PROT_READ | PROT_WRITE,
                                  MAP_SHARED, shm_fd, 0);
```

El mapeo mediante mmap es fundamental. Permite que el productor y el consumidor compartan el mismo espacio físico de memoria a pesar de ser procesos distintos con espacios virtuales separados.

Inserción de datos en el buffer circular

Durante la producción:

```
sem_wait(vacio);
compartir->bus[compartir->entrada] = i;
compartir->entrada = (compartir->entrada + 1) % BUFFER;
sem_post(lleno);
```

El flujo es :

1. **sem_wait(vacio)** garantiza que exista espacio libre.
2. El ítem se escribe en la posición indicada por entrada.
3. El índice se actualiza circularmente.
4. **sem_post(lleno)** despierta al consumidor para que lea el nuevo dato.

Aquí está la esencia del Productor–Consumidor: el uso disciplinado de semáforos evita

condiciones de carrera y garantiza que la secuencia de inserción sea segura, incluso con múltiples procesos.

Limpieza de recursos
Finalmente, el productor ejecuta:
```
sem_unlink("/vacio");
sem_unlink("/lleno");
shm_unlink("/memoria_compartida");
```
Esto garantiza:
- No quedan semáforos persistentes en el sistema.
- La memoria compartida se elimina al finalizar.
- No existe riesgo de que ejecuciones previas afecten nuevas ejecuciones.

El proceso productor actúa como creador y destructor de recursos del sistema, lo cual representa una buena práctica en sistemas operativos.

Proceso consumidor: análisis y comportamiento

El archivo **consumer.c** complementa al productor y se sincroniza estrechamente con él. Sus responsabilidades son:
1. Abrir los semáforos ya creados.
2. Mapear la misma región de memoria compartida.
3. Consumir los ítems de forma sincronizada.
4. Actualizar el índice de salida.
5. Liberar espacio en el buffer mediante semáforos.

Apertura de semáforos existentes
```
sem_t *vacio = sem_open("/vacio", 0);
sem_t *lleno = sem_open("/lleno", 0);
```
Este comportamiento es clave:
a diferencia del productor, el consumidor no crea, solo abre los semáforos existentes.

Mapeo de la memoria compartida
Se realiza nuevamente con mmap, igual que en el productor. Ambas instancias del proceso comparten la misma estructura en RAM, lo que permite que las actualizaciones sean visibles simultáneamente.

Extracción de ítems y actualización del buffer

```
sem_wait(lleno);
item = compartir->bus[compartir->salida];
compartir->salida = (compartir->salida + 1) % BUFFER;
sem_post(vacio);
```

El flujo presenta:
1. sem_wait(lleno) asegura que exista un ítem.

2. El consumidor lee el dato con seguridad.
3. Se actualiza el índice de salida.
4. Se libera un espacio mediante sem_post(vacio).

La coordinación entre lleno y vacío evita inconsistencias, evita lectura fuera de rango y asegura que el consumidor nunca lea datos inválidos.

Implementación y su relevancia

Esta actividad demuestra los principios esenciales de la sincronización interprocesos:

- Estricto control de acceso al buffer.
- Sincronización explícita entre procesos independientes.
- Uso adecuado de memoria compartida.
- Mantenimiento del estado consistente del buffer circular.
- Administración responsable de recursos del sistema.

Además, la implementación cumple con los requisitos académicos del problema: claridad, precisión en el uso de semáforos POSIX, y una representación fiel del comportamiento esperado en sistemas concurrentes.

Actividad 2: Implementación del Productor–Consumidor mediante Hilos POSIX

La segunda parte del taller aborda la resolución del problema Productor–Consumidor utilizando hilos POSIX (pthread) dentro de un mismo proceso. A diferencia del enfoque basado en procesos y semáforos, esta variante implementa la concurrencia mediante múltiples hilos que comparten el mismo espacio de direcciones, lo cual simplifica la comunicación interna pero exige una disciplina rigurosa para mantener la integridad de los datos.
Este modelo aprovecha primitivas esenciales de la biblioteca pthread, como mutex y variables de condición, que proporcionan mecanismos eficientes y seguros para la coordinación entre hilos. A través de este enfoque, se busca entender cómo funciona la exclusión mutua, la espera condicionada y la señalización dentro de un proceso multihilo, así como las ventajas y desventajas que presenta frente al modelo basado en procesos.

Descripción general de la implementación

El archivo principal de esta parte, posixSincro.c, implementa un sistema compuesto por:

- 10 hilos productores
- 1 hilo consumidor
- Un buffer circular de cadenas de texto
- Un mutex global
- Dos variables de condición:
    - hay_espacio (para productores)
    - hay_datos (para el consumidor)

Cada productor genera una serie de mensajes, los escribe en el buffer cuando hay espacio disponible y luego notifica al consumidor. A su vez, el consumidor procesa todos los mensajes generados y continúa en ejecución hasta que todos los productores han finalizado.

Hilos POSIX y modelo de memoria compartida implícita

A diferencia de los procesos, los hilos:

- comparten el mismo espacio de direcciones,
- acceden directamente a las mismas variables globales,
- requieren menos tiempo de creación y conmutación de contexto,
- funcionan de manera más eficiente en tareas altamente paralelizables.

Esto implica que no es necesario usar memoria compartida explícita mediante mmap, pero sí es imprescindible garantizar exclusión mutua en toda sección crítica para evitar condiciones de carrera.

El mutex: exclusión mutua en un entorno multihilo

El mutex declarado es:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Su función es asegurar que:

Solo un hilo pueda insertar o extraer del buffer a la vez, la integridad de los índices indice_escritura y indice_lectura se preserve y se evite la escritura simultánea en las mismas posiciones.
El mutex protege todo acceso al estado compartido:

- el buffer de mensajes
- el contador de espacios disponibles
- el conteo de mensajes pendientes.

Variables de condición: espera eficiente sin ocupación activa

En lugar de emplear ciclos de espera (busy waiting), el programa utiliza dos variables de condición:

```
pthread_cond_t hay_espacio = PTHREAD_COND_INITIALIZER;
pthread_cond_t hay_datos = PTHREAD_COND_INITIALIZER;
```

Estas permiten suspender hilos mientras se espera que un recurso cambie de estado.

Condición para productores
Un productor solo puede escribir si `espacios_disponibles > 0.`
Si el buffer está lleno:

```
while (!espacios_disponibles)
    pthread_cond_wait(&hay_espacio, &mutex);
```

Esta llamada:

- libera temporalmente el mutex,
- bloquea al hilo productor,
- lo despierta solo cuando otro hilo llame pthread_cond_signal(&hay_espacio).

Condición para el consumidor
El consumidor solo puede leer si `mensajes_pendientes > 0.`
Si el buffer está vacío:

```
while (!mensajes_pendientes)
    pthread_cond_wait(&hay_datos, &mutex);
```

El consumidor se bloquea hasta que un productor inserte un mensaje y lo señale con: pthread_cond_signal(&hay_datos);

El buffer circular en esta variante

A diferencia de la solución con procesos, en esta versión el buffer almacena **strings**, no enteros. Esto tiene implicaciones importantes:

- cada entrada del buffer puede contener mensajes más descriptivos,
- se requiere mayor espacio por posición,
- el formato "Thread X: N" permite identificar fácilmente qué productor generó qué mensaje.

El buffer tiene tamaño fijo:

```
#define TAMANO_BUFFER 5
```

Al igual que en la primera actividad, se usa aritmética modular:

```
indice_escritura = (indice_escritura + 1) % TAMANO_BUFFER;
```

```
indice_lectura = (indice_lectura + 1) % TAMANO_BUFFER;
```

Creación de hilos productores y consumidor

En el main() se inicializan:

- 10 productores mediante un ciclo for,
- 1 consumidor independiente.

Cada productor recibe un identificador y se ejecuta de manera concurrente. Esto permite observar cómo múltiples hilos compiten por escribir en el buffer, generando un escenario clásico de concurrencia real.

Sincronización durante la inserción de mensajes

El cuerpo del productor es:

```
pthread_mutex_lock(&mutex);

while (!espacios_disponibles)
    pthread_cond_wait(&hay_espacio, &mutex);

sprintf(buffer[indice_escritura], "Thread %d: %d", mi_id, i);
indice_escritura = (indice_escritura + 1) % TAMANO_BUFFER;
espacios_disponibles--;
mensajes_pendientes++;

pthread_cond_signal(&hay_datos);
pthread_mutex_unlock(&mutex);
```

El uso correcto del patrón:

```
lock → check → wait (si necesario) → operar → signal → unlock
```

representa una sección crítica perfectamente estructurada.
Esto es especialmente importante porque evita:

- pérdida de mensajes,
- lecturas erróneas,
- escritura fuera de rango,
- condiciones de carrera.

Sincronización durante la extracción de mensajes

El consumidor opera así:

```
pthread_mutex_lock(&mutex);

while (!mensajes_pendientes)
    pthread_cond_wait(&hay_datos, &mutex);

printf("%s", buffer[indice_lectura]);
indice_lectura = (indice_lectura + 1) % TAMANO_BUFFER;
mensajes_pendientes--;
```

```
espacios_disponibles++;

pthread_cond_signal(&hay_espacio);
pthread_mutex_unlock(&mutex);
```

El flujo reafirma que:

- el consumidor espera activamente cuando no hay mensajes,
- no hace uso improductivo de CPU,
- respeta la coherencia del buffer.

Cancelación del hilo consumidor (método y análisis)

Una vez todos los productores finalizan y mensajes_pendientes llega a cero, el programa ejecuta:
`pthread_cancel(hilo_cons);`
Esto detiene el consumidor sin requerir un mecanismo de señalización adicional.

Sin embargo aunque pthread_cancel() es funcional en este escenario simple, en aplicaciones reales se recomienda:

- utilizar flags de finalización,
- evitar cancelaciones asíncronas,
- cerrar el hilo mediante pthread_join().

Esto debido a que la cancelación puede interrumpir al hilo en un punto no deseado.

Rendimiento y eficiencia del modelo basado en hilos

Comparado con los procesos:
crear un hilo es mucho más rápido que crear un proceso, los cambios de contexto son más ligeros, los hilos comparten memoria automáticamente, no se necesita mmap, shm_open ni sem_open.
Esto hace que el modelo sea:

- más eficiente
- más escalable
- más adecuado para cargas intensivas.

Sin embargo, también aumenta el riesgo de errores:

- corrupciones de memoria
- desbordes
- uso indebido de punteros
- interacciones accidentales entre hilos.

Seguridad y posibles fallos evitados

Gracias al uso adecuado de mutex y variables de condición, esta implementación:

- evita deadlocks

- previene starvation
- asegura exclusión mutua
- mantiene el orden FIFO de los mensajes
- conserva la integridad del buffer.

Conclusiones específicas de la actividad

La solución con hilos POSIX representa una forma eficiente y moderna de implementar sincronización dentro de un mismo proceso. El correcto uso de pthread_mutex y pthread_cond demuestra una comprensión sólida de los mecanismos de concurrencia.
Esta actividad permite observar:
Diferencias conceptuales frente a procesos independientes, ventajas de rendimiento, importancia del uso disciplinado del mutex y utilidad de las variables de condición para evitar espera activa.

Resultados y Observaciones

La implementación de los dos modelos del problema Productor–Consumidor –uno basado en procesos POSIX con semáforos y memoria compartida, y otro basado en hilos POSIX con mutex y variables de condición– permitió observar de manera detallada las diferencias fundamentales entre ambos enfoques, su eficiencia práctica y su comportamiento al gestionar concurrencia real.
Los resultados obtenidos en la ejecución de cada parte del taller se enmarcan en criterios propios de los sistemas operativos modernos: consistencia del buffer circular, ausencia de condiciones de carrera, uso adecuado de mecanismos de sincronización y comportamiento determinístico bajo condiciones controladas. A continuación, se describen los resultados más representativos de cada implementación, así como las observaciones relevantes que deben resaltarse en el informe.

Resultados de la Actividad 1: Procesos POSIX con Semáforos y Memoria

Compartida

Correcta coordinación entre productor y consumidor
Durante las pruebas se observó que:
- El productor nunca insertó elementos en un buffer lleno, gracias al semáforo /vacio.
- El consumidor nunca intentó leer datos inexistentes, gracias al semáforo /lleno.

El flujo de ejecución mostró alternancia armónica entre los procesos, sin bloqueos ni fallos de sincronización.

Comportamiento estable del buffer circular
El buffer compartido:
- mantuvo correcta consistencia de los índices entrada y salida,
- nunca presentó sobreescritura de datos no consumidos,
- preservó el orden FIFO completo de los elementos generados.

Esto demuestra que la lógica de actualización modular funciona correctamente.

Persistencia y visibilidad inmediata de los datos
Los datos producidos son inmediatamente visibles para el consumidor gracias a:
- memoria compartida mapeada físicamente,
- sincronización explícita mediante semáforos.

Este resultado evidencia que la combinación shm_open + mmap es altamente confiable.

Ausencia de deadlocks y starvation
Todas las ejecuciones completaron sin:
- bloqueos mutuos entre procesos,
- situaciones de espera infinita,
- inequidades en el acceso al buffer.

La configuración de semáforos garantiza justicia en la operación.

Liberación completa de recursos
El productor, al finalizar, eliminó:
- los semáforos con sem_unlink,
- la memoria compartida con shm_unlink.

Gracias a esto, no quedaron artefactos en /dev/shm, algo esencial para garantizar ejecuciones limpias entre corridas sucesivas.

Observaciones de la Actividad 1

1. La sincronización entre procesos requiere atención rigurosa: un pequeño error en el orden de llamadas puede producir inconsistencias graves.
2. El rendimiento depende significativamente del sistema: las llamadas sem_open, mmap y cambio de contexto entre procesos son costosas en comparación con hilos.
3. La comunicación interproceso basada en memoria compartida es rápida, pero exige sincronización estricta.
4. Esta solución es robusta, pero tiene mayor sobrecarga que el modelo basado en hilos.

Resultados de la Actividad 2: Hilos POSIX con Mutex y Variables de Condición

Correcto funcionamiento de múltiples productores
Los 10 hilos productores generaron sus mensajes sin interferencias entre sí. El mutex garantizó:
- acceso exclusivo al buffer,
- actualizaciones consistentes de índices,
- prevención de condiciones de carrera.

La salida mostró intercalamiento de mensajes, tal como se espera de un sistema concurrente.

Consumidor eficiente y estable
El hilo consumidor procesó todos los mensajes generados por los productores.
No se registraron:
- lecturas duplicadas,
- pérdidas de información,
- accesos desordenados.

Eliminación de espera activa
El uso de variables de condición permitió que:
- los productores durmieran cuando el buffer estaba lleno,
- el consumidor durmiera cuando estaba vacío.

Esto elimina el consumo innecesario de CPU.

Comportamiento FIFO garantizado
El buffer circular mantuvo orden de llegada incluso con múltiples hilos escribiendo simultáneamente.

Cancelación del consumidor tras finalizar los productores
Una vez que todos los hilos productores terminaron su ejecución, el consumidor fue cancelado adecuadamente mediante pthread_cancel(), asegurando que no quedaran hilos activos sin propósito.

Observaciones de la Actividad 2

1. La sincronización en hilos es más rápida y ligera que en procesos.
2. El modelo es más propenso a errores de memoria, ya que todo se comparte de manera implícita.
3. El uso adecuado de mutex y variables de condición evitó completamente problemas como condiciones de carrera.
4. El intercalado de mensajes evidencia el paralelismo real que se puede obtener en una máquina multinúcleo.
5. Para aplicaciones más grandes, se recomienda implementar un mecanismo de cierre más explícito que pthread_cancel.

Comparación general de los dos modelos

Característica	**Procesos con semáforos**	**Hilos con mutex/condiciones**
Espacio de memoria	Separado	Compartido
Velocidad	Más lento	Más rápido
Complejidad de IPC	Alta	Baja
Riesgo de errores de memoria	Bajo	Alto
Rendimiento	Medio	Alto
Robustez	Muy alta	Alta

Conclusiones preliminares de los resultados

De ambos experimentos se concluye que:

- La sincronización es correcta en ambos modelos.
- Los resultados muestran ausencia total de condiciones de carrera.
- El buffer circular se comporta de manera predecible y estable.
- Los mecanismos POSIX son altamente confiables.
- Los hilos ofrecen mayor eficiencia, pero los procesos brindan más aislamiento.

Conclusiones Generales

El desarrollo de las dos actividades del taller permitió comprender de manera integral los mecanismos de sincronización basados en semáforos POSIX, memoria compartida y hilos POSIX bajo el estándar POSIX. A través de la implementación del problema clásico Productor–Consumidor, se logró demostrar no solo la aplicabilidad práctica de estos mecanismos, sino también la profundidad conceptual que implica la construcción de sistemas concurrentes confiables y eficientes.

En primer lugar, la implementación con procesos independientes, semáforos con nombre y memoria compartida evidenció el papel del sistema operativo como intermediario en la coordinación de tareas. El modelo demostró que, aun cuando los procesos poseen espacios de memoria totalmente aislados, es posible establecer una comunicación segura mediante regiones compartidas cuidadosamente mapeadas y protegidas. Los semáforos actuaron como un mecanismo confiable para garantizar la exclusión mutua implícita y el control del estado del buffer, evitando condiciones de carrera y manteniendo la integridad de las operaciones. En este sentido, se comprobó que los procesos constituyen un modelo de ejecución robusto, especialmente útil cuando la seguridad, el aislamiento o la estabilidad del sistema son prioridades.

En segundo lugar, la variante basada en hilos POSIX con mutex y variables de condición permitió observar una aproximación más eficiente en términos de rendimiento y consumo de recursos. Dado que los hilos comparten de manera natural el espacio de memoria del proceso, la comunicación entre ellos se vuelve más directa, pero también más delicada en términos de manejo correcto de la sincronización. La implementación demostró el uso adecuado de primitivas como pthread_mutex_lock, pthread_cond_wait y pthread_cond_signal, garantizando que la interacción entre múltiples productores y un consumidor ocurriera sin errores, sin condiciones de carrera y sin usar espera activa. Este enfoque es especialmente pertinente en sistemas modernos con múltiples núcleos, donde la ejecución concurrente puede aprovecharse de manera real.

Una conclusión clave del taller es entender que no existe un único modelo universalmente superior; en cambio, cada mecanismo ofrece ventajas y desventajas que deben evaluarse según el contexto:

- Los procesos ofrecen mayor aislamiento, seguridad y control rígido sobre recursos, aunque a costa de más sobrecarga.
- Los hilos proporcionan mayor rendimiento, menor uso de memoria y una interacción más fluida, pero requieren un manejo extremadamente cuidadoso para evitar errores difíciles de depurar.

Ambas implementaciones también demostraron la importancia de diseñar estructuras de datos adecuadas, como el buffer circular, cuya naturaleza permite un uso eficiente y continuo del espacio. La aritmética modular aplicada a los índices de entrada y salida se confirmó como una técnica simple pero poderosa para garantizar la integridad lógica del sistema concurrente.

De manera general, el taller permitió poner en práctica conceptos fundamentales de los sistemas operativos, entre ellos:

- Exclusión mutua
- Sincronización
- Comunicación entre procesos (IPC)
- Manejo de memoria compartida
- Coordinación mediante semáforos
- Control del paralelismo con hilos
- Prevención de errores de concurrencia

Asimismo, el proceso de programación, ejecución, depuración y análisis contribuyó a reforzar habilidades esenciales en la ingeniería de de sistemas, tales como razonamiento crítico sobre condiciones de carrera, comprensión profunda del comportamiento del scheduler, y análisis del impacto real de mecanismos de sincronización sobre el rendimiento.

Finalmente, se concluye que el taller cumplió plenamente con su objetivo académico: permitirnos comprender en profundidad los mecanismos de sincronización del sistema operativo mediante su aplicación directa, observando sus resultados, identificando sus fortalezas y reconociendo sus limitaciones. El ejercicio permitió ver, con claridad, cómo la teoría de concurrencia estudiada en clase se convierte en práctica operativa dentro de un sistema real, integrando saberes fundamentales para el diseño de software concurrente seguro, correcto y eficiente.

Referencias

Andrews, G. R. (2000). *Foundations of multithreaded, parallel, and distributed programming*. Addison-Wesley.

Bovet, D. P., & Cesati, M. (2006). *Understanding the Linux Kernel* (3rd ed.). O'Reilly Media.

Dijkstra, E. W. (1965). Solution of a problem in concurrent programming control. *Communications of the ACM, 8*(9), 569.

Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press.

Love, R. (2010). *Linux system programming: Talking directly to the kernel and C library* (2nd ed.). O'Reilly Media.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2022). *Operating System Concepts* (10th ed.). Wiley.

Stallings, W. (2018). *Operating systems: Internals and design principles* (9th ed.). Pearson.

Stevens, W. R., & Rago, S. A. (2013). *Advanced programming in the UNIX environment* (3rd ed.). Addison-Wesley.

IEEE. (2018). *IEEE Standard for Information Technology — Portable Operating System Interface (POSIX®) Base Specifications* (Issue 7). IEEE.

Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison-Wesley.

Torres, J. (2015). *Programación Concurrente en Sistemas POSIX*. Alfaomega.