TALLER 03 SINCROPOXIS

SISTEMAS OPERATIVOS

NOVIEMBRE - 13 – 2025

YOSEPH PEÑA - JUAN MANUEL SOLANO

1. Introducción:

En los sistemas operativos modernos, la concurrencia es un aspecto fundamental para aprovechar eficientemente los recursos de hardware. La presencia de múltiples procesos e hilos que comparten datos y recursos trae consigo el problema de la sincronización, especialmente para evitar condiciones de carrera, inconsistencias y bloqueos.

Este taller se centra en dos mecanismos de sincronización de POSIX:

- Semáforos con nombre y memoria compartida (IPC entre procesos independientes).
- Hilos POSIX (pthreads), mutex y variables de condición (sincronización dentro de un mismo proceso multihilo).

A través de dos actividades se implementa el clásico problema productor–consumidor, primero entre procesos y luego entre hilos, analizando las ventajas, desafíos y consideraciones de diseño en cada caso.

2. Objetivos

Objetivo general
Implementar y analizar mecanismos de sincronización en sistemas POSIX utilizando semáforos con nombre, memoria compartida y hilos POSIX, aplicados al problema productor–consumidor.

Objetivos específicos
- Comprender el uso de semáforos POSIX con nombre para coordinar procesos independientes.
- Implementar memoria compartida como mecanismo de comunicación entre procesos.
- Utilizar mutex y variables de condición para sincronizar hilos dentro de un mismo proceso.
- Diseñar e implementar buffers circulares seguros ante concurrencia.
- Evaluar el comportamiento del sistema, identificando cómo se evitan las condiciones de carrera y el acceso inconsistente a los recursos compartidos.
- Documentar adecuadamente el código fuente y elaborar un informe técnico claro y ordenado.

3. Marco Teórico:

3.1. Semáforos POSIX con nombre
Un semáforo es una estructura de sincronización que mantiene un contador entero y expone primitivas atómicas típicas: wait (o P, o sem_wait) y signal (o V, o sem_post).

En POSIX con nombre, se crean mediante sem_open(const char *name, int oflag, ...), lo que permite que procesos no relacionados accedan al mismo semáforo usando únicamente su nombre (por ejemplo, "/vacio" y "/lleno"). Estos semáforos persisten en el sistema hasta que se eliminan explícitamente con sem_unlink.

En el contexto productor–consumidor:
- vacio: cuenta cuántas posiciones libres hay en el buffer.
- lleno: cuenta cuántas posiciones ocupadas hay disponibles para consumir.

3.2. Memoria compartida POSIX

La memoria compartida permite que varios procesos accedan a la misma región de memoria, lo que la convierte en un mecanismo de comunicación muy eficiente. Con POSIX se utiliza:
- shm_open para crear/abrir un objeto de memoria compartida.
- ftruncate para definir su tamaño.
- mmap para mapear el objeto a la memoria virtual del proceso.

Todos los procesos que hagan mmap sobre el mismo objeto compartirán los mismos datos.

3.3. Hilos POSIX (pthreads), mutex y variables de condición

Los hilos POSIX (pthreads) proporcionan múltiples flujos de ejecución que comparten el mismo espacio de direcciones de un proceso. Esto significa que variables globales, estructuras y el heap son visibles para todos los hilos, lo cual facilita la comunicación, pero incrementa el riesgo de condiciones de carrera.

Para controlar el acceso a los datos compartidos se emplea:
- Mutex (pthread_mutex_t): garantiza exclusión mutua: sólo un hilo puede acceder a la región crítica a la vez.
- Variables de condición (pthread_cond_t): permiten que un hilo se duerma hasta que una cierta condición lógica se cumpla, siendo despertado por otro hilo mediante pthread_cond_signal o pthread_cond_broadcast.

3.4. Buffer circular

Un buffer circular es una estructura de datos que utiliza un arreglo de tamaño fijo y dos índices:
- entrada o índice de escritura.
- salida o índice de lectura.

Ambos índices avanzan de forma circular usando la operación módulo: indice = (indice + 1) % BUFFER_SIZE.

Esto permite aprovechar al máximo el arreglo sin necesidad de mover elementos, ideal para implementaciones de productor–consumidor.

4. Actividad 1:

4.1. Descripción general

En esta actividad se implementó el problema productor–consumidor mediante:
- Un archivo de cabecera compartido shared.h.
- Un proceso productor (producer.c).
- Un proceso consumidor (consumer.c).
- Un Makefile para compilar y ejecutar los componentes.

El productor y el consumidor se comunican mediante:
- Un buffer circular de tamaño fijo (BUFFER_SIZE = 5).
- Memoria compartida POSIX.
- Dos semáforos con nombre: "/vacio" y "/lleno".

4.2. Cabecera compartida shared.h
En shared.h se definen las cabeceras estándar requeridas, la constante del tamaño del buffer y la estructura compartida:

- BUFFER_SIZE 5: tamaño del buffer circular.
- typedef struct compartir_datos:
  - int bus[BUFFER_SIZE]; → buffer donde se almacenan los ítems producidos.
  - int entrada; → índice donde el productor escribe.
  - int salida; → índice donde el consumidor lee.

Esta cabecera es utilizada tanto por el productor como por el consumidor para garantizar consistencia en la definición de los datos compartidos.

4.3. Implementación del Productor (producer.c)
El productor realiza las siguientes tareas principales:

1. Creación de semáforos con nombre
   - vacio se inicializa con el valor BUFFER_SIZE (todas las posiciones están libres).
   - lleno se inicializa con 0 (no hay ítems listos para consumir).

2. Creación de memoria compartida
   - Uso de shm_open para crear/abrir el objeto.
   - ftruncate para definir el tamaño.
   - mmap para mapear el objeto en el espacio de direcciones del proceso.

3. Inicialización del buffer
   - entrada = 0;
   - salida = 0;

4. Bucle de producción (10 ítems)
   Para cada ítem:
   - sem_wait(vacio); → espera un espacio libre.
   - Escribe el valor en bus[entrada].
   - Actualiza entrada con aritmética modular.
   - sem_post(lleno); → notifica que hay un ítem disponible.
   - sleep(1) simula tiempo de producción.

5. Limpieza de recursos
   - munmap y close para la memoria compartida.
   - sem_close para los semáforos.

4.4. Implementación del Consumidor (consumer.c)
El consumidor ejecuta el proceso inverso:

1. Apertura de semáforos existentes (sin O_CREAT, pues ya los creó el productor).
2. Apertura y mapeo de la memoria compartida con shm_open y mmap.

3. Inicialización del índice de salida (salida = 0).
4. Bucle de consumo (10 ítems):
   - sem_wait(lleno); → espera hasta que exista un ítem disponible.
   - Lee el valor en bus[salida].
   - Actualiza salida con aritmética modular.
   - sem_post(vacio); → libera un espacio en el buffer.
   - sleep(2) simula tiempo de consumo.
5. Liberación de recursos mediante munmap, close y sem_close.

4.5. Sincronización y ausencia de condiciones de carrera
La sincronización se logra combinando memoria compartida y semáforos:

- El productor sólo escribe cuando hay espacio libre.
- El consumidor sólo lee cuando hay elementos disponibles.

Esto evita situaciones como:
- Intentar consumir de un buffer vacío.
- Intentar escribir en un buffer lleno.

Aunque no se usa un mutex adicional para proteger el buffer, los accesos a entrada y salida están bien ordenados por los semáforos, ya que el productor y el consumidor nunca modifican el mismo índice.

5. Actividad 2:

5.1. Descripción general
En la segunda actividad se trabaja con hilos POSIX dentro de un único proceso. En lugar de usar memoria compartida entre procesos, se aprovecha que los hilos comparten las variables globales del programa.

El código se basa en:
- Un conjunto de productores que generan líneas/mensajes.
- Un spooler (consumidor) que procesa/imprime los mensajes almacenados.
- Un buffer circular de cadenas:
  char buf[MAX_BUFFERS][100];
  int buffer_index;
  int buffer_print_index;

- Variables globales de control:
  int buffers_available = MAX_BUFFERS;
  int lines_to_print = 0;

- Mecanismos de sincronización:
  pthread_mutex_t buf_mutex;
  pthread_cond_t buf_cond;
  pthread_cond_t spool_cond.

5.2. Diseño de la sincronización con mutex y variables de condición
El patrón de sincronización es similar al de productor–consumidor de la Actividad 1, pero adaptado a hilos:

1. Exclusión mutua con buf_mutex
   - Cada acceso a buf, buffer_index, buffer_print_index, buffers_available y lines_to_print se encapsula dentro de pthread_mutex_lock / pthread_mutex_unlock.

2. Variables de condición
   - buf_cond: un productor que desea escribir espera en esta condición cuando no hay buffers disponibles (buffers_available == 0).
   - spool_cond: el spooler espera en esta condición mientras no haya líneas por imprimir (lines_to_print == 0).

3. Buffer circular
   - Los índices de escritura y lectura avanzan de forma circular mediante mod MAX_BUFFERS, permitiendo reutilizar el arreglo sin mover datos.

5.3. Comportamiento de los hilos productores
Cada hilo productor realiza:
1. Generar un mensaje (por ejemplo, una línea de texto).
2. Adquirir el mutex buf_mutex.
3. Comprobar si hay buffers disponibles; si no, esperar en pthread_cond_wait(&buf_cond, &buf_mutex).
4. Copiar el mensaje al buffer buf[buffer_index].
5. Actualizar buffer_index (buffer_index + 1) % MAX_BUFFERS.
6. Ajustar los contadores:
   - buffers_available--;
   - lines_to_print++;
7. Señalar al spooler que hay líneas disponibles: pthread_cond_signal(&spool_cond).
8. Liberar el mutex.

5.4. Comportamiento del hilo spooler (consumidor)
El hilo spooler realiza:
1. Adquirir el mutex buf_mutex.
2. Mientras lines_to_print == 0, esperar en pthread_cond_wait(&spool_cond, &buf_mutex).
3. Tomar la siguiente línea desde buf[buffer_print_index].
4. Actualizar buffer_print_index.
5. Ajustar contadores:
   - lines_to_print--;
   - buffers_available++;
6. Señalar a posibles productores bloqueados: pthread_cond_signal(&buf_cond).
7. Liberar el mutex.
8. Procesar/imprimir el mensaje fuera de la sección crítica.

Este esquema garantiza que:
- El spooler no imprime cuando el buffer está vacío.
- Los productores no sobrescriben entradas que aún no han sido consumidas.
- El acceso al buffer y sus variables de control sea atómico desde el punto de vista de la lógica del programa.

6. Makefile

Se incluye un Makefile que automatiza la compilación de los distintos componentes del taller:

- Objetivos definidos:
  - producer
  - consumer
  - posixSincro

- Compilación con flags comunes:
  - CFLAGS = -Wall -Wextra -std=c99 -pthread

- Enlazado con la librería de tiempo real (-lrt) necesaria para semáforos y memoria compartida POSIX en algunas plataformas.

- Targets adicionales:
  - clean: elimina ejecutables y objetos.
  - clean-sem: elimina archivos de semáforos y memoria compartida del sistema.
  - run-producer, run-consumer, run-both: facilitan la ejecución y pruebas desde el propio Makefile.

Esta organización favorece la modularidad, la reproducibilidad y un manejo más profesional del ciclo de compilación y pruebas.

7. Resultados y Pruebas:

7.1. Actividad 1
Al ejecutar producer y consumer (por ejemplo mediante make run-both), se observa en la salida:

- El productor indicando: "Productor: Produce X en posición Y".
- El consumidor indicando: "Consumidor: Consume X de posición Y".

El orden de producción/consumo respeta la disciplina FIFO del buffer circular. No se observan:
- Lecturas de posiciones no inicializadas.
- Pérdida de ítems.
- Bloqueos permanentes mientras ambos procesos se ejecutan.

La diferencia en tiempos (sleep(1) en el productor y sleep(2) en el consumidor) permite visualizar claramente cómo el buffer se va llenando y vaciando de forma controlada por los semáforos.

7.2. Actividad 2
En la segunda actividad, al ejecutar el programa posixSincro, se evidencia:

- Creación de múltiples hilos productores.
- Un hilo spooler que procesa los mensajes en segundo plano.
- La intercalación de mensajes producidos por distintos hilos, pero sin que se mezclen ni se corrompan.

La correcta sincronización con mutex y variables de condición impide que:
- Dos hilos escriban simultáneamente en el mismo índice del buffer.
- El spooler lea posiciones vacías.
- Se produzcan condiciones de carrera visibles en la salida.

8. Conclusiones:

1. Comprensión de la sincronización POSIX
El taller permitió afianzar el entendimiento de cómo los semáforos con nombre y la memoria compartida se combinan para sincronizar procesos independientes, y cómo los mutex y las variables de condición coordinan múltiples hilos dentro de un mismo proceso.

2. Importancia del diseño del buffer circular
El uso de un buffer circular simplifica la gestión de recursos fijos, evitando desplazamientos de datos y favoreciendo implementaciones eficientes del patrón productor–consumidor tanto entre procesos como entre hilos.

3. Evitar condiciones de carrera y bloqueos
Las soluciones desarrolladas muestran que el uso correcto de sem_wait/sem_post, así como de pthread_mutex_lock/pthread_mutex_unlock y pthread_cond_wait/pthread_cond_signal, es crucial para mantener la consistencia de los datos compartidos y evitar errores sutiles y difíciles de depurar.

4. Modularidad y buenas prácticas
La separación del código en archivos (shared.h, producer.c, consumer.c, posixSincro.c) y el uso de un Makefile mejoran la claridad, la mantenibilidad y la escalabilidad del proyecto, acercándose a prácticas de desarrollo profesional.

5. Aplicabilidad práctica
Los conceptos implementados son directamente aplicables a sistemas reales que requieren procesamiento en paralelo, servicios concurrentes, colas de trabajo y servidores multihilo, constituyendo una base sólida para desarrollos más avanzados en computación de alto rendimiento y sistemas distribuidos.