Universidad de Concepción

Tarea 2: Sincronización y Memoria Virtual

Asignatura	Sistemas Operativos
Integrantes	Pablo Bettancourt Pinto Bastián Ceballos Zapata Angie Ramírez González Walter Zárate Solar
Profesor	Juan Felipe González

1 de diciembre de 2025

Universidad de Concepción

1 Objetivos

El presente informe aborda dos aspectos fundamentales de los sistemas operativos: la sincronización de hebras y la gestión de memoria virtual. Esta tarea se realiza con el propósito de aprender a diseñar y aplicar primitivas de sincronización, manejar condiciones de carrera y exclusión mutua, y comprender la traducción de direcciones y la gestión de memoria con paginación.

Para lograr esto, se incluyen pruebas de funcionamiento y análisis experimental en ambas partes, lo que permite verificar el correcto comportamiento de las implementaciones y evaluar su desempeño bajo distintas configuraciones, integrando así la teoría con la práctica.

Todos los códigos y desarrollo de la tarea se pueden encontrar en el siguiente repositorio de GitHub: https://github.com/Rhussu/SO-Tarea-2.

2 Parte I: Sincronización con Barrera reutilizable

En esta sección se implementa y prueba una barrera reutilizable para coordinar N hebras concurrentes. Se verifica su correcto funcionamiento mediante un programa de prueba que asegura que ninguna hebra avance hasta que todas hayan llegado al punto de encuentro, permitiendo la sincronización en múltiples etapas.

2.1 Implementación

La barrera reutilizable fue implementada utilizando el modelo de monitor con las primitivas pthread_mutex_t y pthread_cond_t. El objetivo es permitir que un conjunto de N hebras llegue a un punto de sincronización y que ninguna continúe a la siguiente etapa hasta que todas hayan alcanzado dicho punto. Además, la barrera debe ser reutilizable, lo que implica que esta coordinación debe repetirse en múltiples etapas.

La estructura principal (barrera_t) mantiene tres variables de estado:

- **count**: número de hebras que han llegado a la barrera en la etapa actual.
- **N**: cantidad total de hebras que participan en la sincronización.
- **etapa**: identificador de la etapa actual, usado para permitir la reutilización.

A esto se suman un mutex, que garantiza exclusión mutua al acceder al estado compartido, y una variable de condición, que permite que las hebras esperen hasta que todas lleguen.

La función barrera_wait() implementa el patrón característico de los monitores:

1. **lock** del mutex.
2. **Actualización del estado** (incremento de count) y captura local de la etapa actual.

3. **Decisión**:
    - Si la hebra no es la última, debe esperar mientras la etapa no cambie.
    - Si la hebra es la última, debe avanzar la etapa, reiniciar el contador y despertar a todas mediante pthread_cond_broadcast().
4. **unlock** del mutex.

El uso del campo etapa permite distinguir entre diferentes rondas de sincronización, garantizando que las hebras liberadas pertenecen efectivamente a la misma etapa, evitando así condiciones de carrera entre etapas consecutivas. Esta lógica asegura que la barrera pueda ser reutilizada indefinidamente sin inconsistencias ni bloqueos.

El código final se divide en los archivos barrera.h y barrera.c, lo que facilita su integración con distintas aplicaciones y permite reutilizarlo.

2.2 Aplicación de verificación

Para validar el correcto funcionamiento de la barrera se desarrolló un programa de prueba en el archivo main.c. Este programa crea N hebras (5 por defecto), las cuales atraviesan E etapas de sincronización (4 por defecto). En cada etapa, cada hebra:

1. Ejecuta una pausa aleatoria mediante usleep(), simulando trabajo real.
2. Imprime un mensaje indicando que está esperando en la etapa e.
3. Llama a barrera_wait(), quedando bloqueada si aún no han llegado todas las hebras.
4. Una vez liberada, imprime que pasó la barrera en la etapa e.

El criterio de verificación se cumple si, en cada etapa:

- Todas las hebras imprimen “esperando en etapa e” antes de que cualquiera imprima “paso barrera en etapa e”.
- Ninguna hebra inicia la etapa siguiente (e+1) antes de que todas hayan completado la etapa actual.
- La secuencia puede intercalarse, pero siempre manteniendo consistencia global.

La ejecución del programa confirmó que el comportamiento era el esperado:
en cada etapa, las hebras esperaron correctamente hasta que la última hebra llegó a la barrera, y solo entonces todas fueron liberadas simultáneamente. La correcta actualización del valor de etapa, junto con el uso de pthread_cond_broadcast(), garantiza que la barrera opere de manera determinística y reutilizable.

Este programa permite demostrar empíricamente que la implementación del monitor funciona sin condiciones de carrera, sin deadlocks y respetando la semántica de sincronización requerida.

3 Parte II: Simulador simple de Memoria Virtual

En esta sección se implementa un simulador de memoria virtual con paginación simple y reemplazo de páginas mediante el algoritmo Reloj. Se prueban diferentes configuraciones de marcos y tamaños de página, evaluando la cantidad de fallos y la tasa de fallos de página para analizar el comportamiento del sistema. El programa procesa un archivo de traza con direcciones virtuales y entrega, para cada configuración, las estadísticas de referencias, fallos y tasa de fallos, mostrando opcionalmente paso a paso la traducción de cada dirección.

3.1 Implementación

El simulador está implementado en C y utiliza dos estructuras principales: Marco, que representa cada marco físico mediante el número de página virtual asignado y un bit de uso, y EntradaTabla, que modela una tabla de páginas simple mediante un arreglo que almacena pares (página virtual, marco). Este diseño permite resolver rápidamente si una página ya está cargada recorriendo la tabla mediante la función buscar_tabla(), mientras que las funciones insertar_tabla() y borrar_tabla() mantienen la consistencia cuando se cargan o reemplazan páginas.

Las direcciones virtuales del archivo de traza se leen como texto y se convierten a enteros usando strtol() para direcciones hexadecimales o atoi() para direcciones decimales. A partir de cada dirección, el simulador extrae el offset aplicando una máscara bit a bit, y obtiene el número de página virtual desplazando la dirección b posiciones a la derecha, donde b = log2(PAGE_SIZE).

El algoritmo de reemplazo se implementa directamente dentro del main(). En cada referencia, primero se consulta si la página ya está en memoria. Si se produce un fallo de página, el simulador busca un marco libre; si ninguno está disponible, se activa el algoritmo Reloj utilizando el índice puntero_reloj. Este puntero avanza circularmente por el arreglo de marcos: si encuentra un marco con el bit de uso en 0, esa página se reemplaza y su entrada se elimina con borrar_tabla(). Si el bit está en 1, simplemente se limpia y el puntero continúa avanzando hasta encontrar un candidato válido.

Durante la simulación se contabilizan las referencias totales y los fallos de página, permitiendo calcular la tasa final al terminar la lectura del archivo. En modo --verbose, el programa imprime cada paso de la traducción, mostrando el número de página, si ocurrió hit o fallo, el marco asignado y la dirección física calculada. En conjunto, esta implementación reproduce de forma directa y estructurada el comportamiento del algoritmo Reloj y permite observar cómo influyen el tamaño de página y el número de marcos en la tasa de fallos.

3.2 Experimentación

3.2.1. Definición del Experimento

El objetivo de este análisis es evaluar el comportamiento de la Tasa de Fallos al variar los recursos de memoria física disponible. Se utilizaron los archivos del paquete traces.zip bajo las siguientes configuraciones:

1. Variables Independientes:
    - Número de Marcos: Se probaron los valores 8, 16 y 32.
    - Tamaño del Marco:
        - Para trace1.txt: 8 bytes.
        - Para trace2.txt: 4096 bytes (4 KB).
2. Volumen de Datos: 8192 referencias de memoria por traza.

Con $tasa\ de\ fallos = \frac{fallos}{referencias}$

3.2.2. Resultados Obtenidos

Las siguientes tablas y gráfico resumen los resultados de la simulación:

Nº Marcos	**Fallos Totales**	**Tasa de Fallos**
8	8073	0.985
16	7943	0.969
32	7713	0.941

Tabla 1: Fallos para Trace1 (Tamaño de Marco: 8 bytes)

Nº Marcos	**Fallos Totales**	**Tasa de Fallos**
8	7649	0.933
16	7138	0.871
32	6142	0.749

Tabla 2: Fallos para Trace2 (Tamaño de Marco: 4096 bytes)

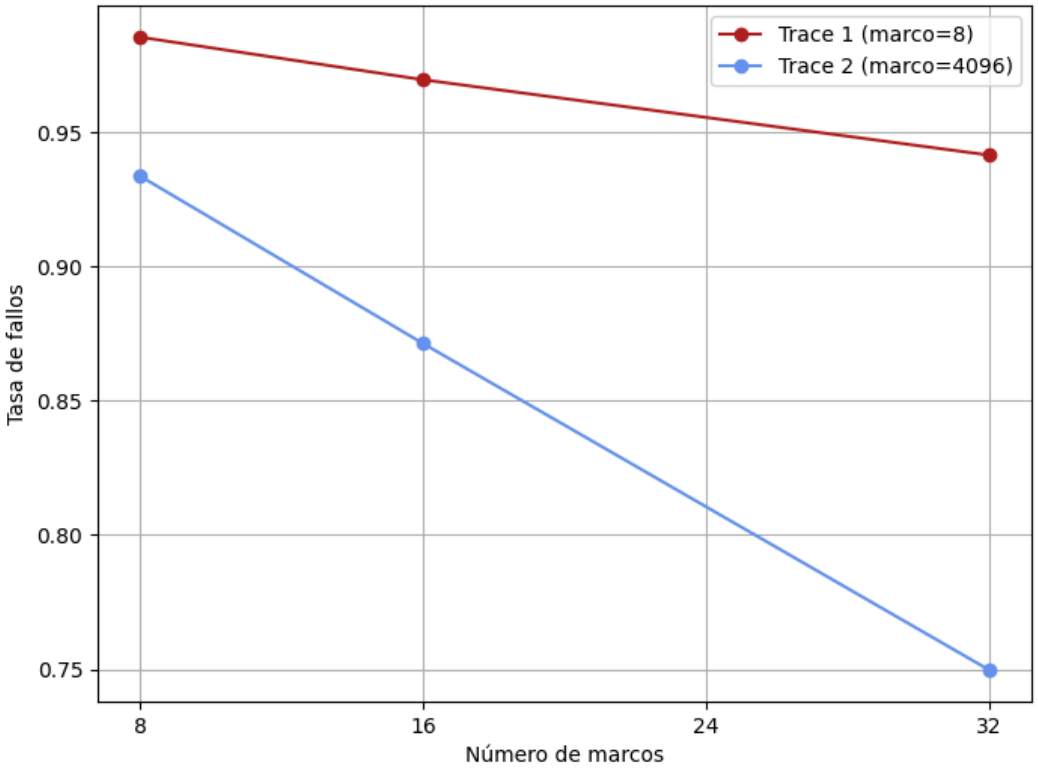

Gráfico 1: Tasa de fallos vs Número de marcos

2.3. Discusión de Resultados

2.3.1 Comportamiento de trace1.txt

- Esta traza se ejecutó con un tamaño de marco extremadamente reducido (8 bytes).
- La tasa de fallos es consistentemente crítica, manteniéndose por encima del 94% incluso en el mejor escenario (32 marcos).
- Al ser el marco tan pequeño, cada página cargada contiene muy poca información útil. Esto impide aprovechar la localidad espacial (la probabilidad de acceder a datos contiguos). El aumento de marcos de 8 a 32 solo logra una mejora marginal (aprox. 4%), lo que indica que el simulador está saturado por la granularidad de los datos, no solo por la falta de espacio.

2.3.2 Comportamiento de trace2.txt

- Esta traza utilizó un tamaño de marco de 4096 bytes, más parecido al estándar en sistemas modernos.
- Se evidencia una sensibilidad mucho mayor al aumento de memoria física.

- Existe una mejora drástica en el rendimiento al aumentar los marcos. Al pasar de 8 a 32 marcos, la tasa de fallos cae casi un 18.4%.

En general,

- En ambos casos, aumentar los marcos reduce los fallos, pero el beneficio es mucho mayor cuando el tamaño del marco es adecuado (trace2).
- Un tamaño de marco demasiado pequeño (trace1) genera una sobrecarga de fallos que no se soluciona simplemente añadiendo más memoria (marcos), ya que obliga a la CPU a realizar demasiadas operaciones de E/S para leer datos muy pequeños.
- La configuración de trace2 (4096 bytes) demuestra ser una simulación más realista y escalable, donde duplicar la memoria (de 16 a 32 marcos) ofrece una ganancia de rendimiento tangible (reducción de tasa de fallos del 0.87 al 0.74).

4 Conclusión

La realización de esta tarea permitió integrar conceptos fundamentales de sincronización y memoria virtual mediante implementaciones prácticas que reflejan el comportamiento real de un sistema operativo. En la primera parte, la construcción de una barrera reutilizable demostró la importancia del control correcto de la concurrencia, validando que el uso coordinado de mutex y variables de condición permite evitar condiciones de carrera y garantizar un avance ordenado entre múltiples hebras.

En la segunda parte, el desarrollo del simulador de memoria virtual y el análisis experimental evidenciaron cómo el tamaño del marco y la cantidad de marcos disponibles influyen directamente en la tasa de fallos de página. Los resultados mostraron que, mientras un tamaño de marco demasiado pequeño limita severamente el rendimiento incluso al aumentar la memoria, configuraciones más realistas permiten apreciar mejoras significativas gracias al algoritmo de reemplazo Reloj.

En conjunto, ambas implementaciones fortalecen la comprensión práctica de mecanismos esenciales de los sistemas operativos, mostrando cómo las decisiones de diseño, tanto en sincronización como en gestión de memoria, impactan directamente el desempeño y el comportamiento global del sistema.