Tarea 2

Sistemas Operativos

Segundo Semestre 2025, Prof. Cecilia Hernández y Juan Felipe González

11 de noviembre de 2025

Fecha Inicio: Martes 11 de noviembre, 2025.
Fecha Entrega: Lunes 1 de diciembre, 2025 (23:59 hrs).
Metodología: Trabajo en grupo: Integrado con 3 o 4 estudiantes. No se aceptarán trabajos individuales.

Contenidos

Esta tarea tiene dos partes, una relacionada con las primitivas de sincronización y otra con el manejo de memoria virtual. Cada parte tiene sus propios objetivos, descripción y actividades a realizar.

Parte I. Sincronización con Barrera reutilizable

1. Objetivos

   - Diseñar e implementar una barrera reutilizable que coordine múltiples hebras concurrentes.
   - Aplicar la conceptualización de monitor usando `pthread_mutex_t` y `pthread_cond_t` de la biblioteca pthread.
   - Razonar sobre condiciones de carrera, exclusión mutua, espera condicional usando variables de condición y conceptualización de monitor.
   - Construir una aplicación de verificación que demuestre la correctitud de su implementación de barrera.

2. Definición

   Una barrera es primitiva que sincroniza a N hebras concurrentes en puntos de encuentro: ninguna hebra puede continuar hasta que todas hayan llegado. Una barrera reutilizable es una barrera que permite sincronizar las hebras en distintos puntos de encuentro. Para ello, la barrera debe resetease automáticamente cuando todas las hebras llegan a un punto de encuentro y pasan a una siguiente etapa.

Actividad 1: Implementación de la barrera (Monitor) (2.0 puntos) KPIs 1.2 y 1.3

Implemente una barrera reutilizable como un **monitor**:

1. Estado compartido mínimo recomendado:

- `int count;` número de hebras que han llegado a la barrera en la etapa actual.
- `int N;` número total de hebras que deben esperar (parámetro de inicialización).
- `int etapa;` identificador de etapa donde operan N hebras.

2. Sincronización:
   - Un `pthread_mutex_t` para la exclusión mutua del estado interno.
   - Un `pthread_cond_t` para que los hebras esperen hasta que lleguen todas al punto de espera.
3. Operaciones
   - La barrera debe poder crearse e inicializarse para sincronizar N hebras, destruirse y soportar su operación fundamental que es el *wait()*, la que bloquea la hebra si es que no es la última de las N.
4. Restricciones
   - Usar **únicamente** `pthread_mutex_t` y `pthread_cond_t` como primitivas de sincronización (no se permite el uso de semáforos ni la barrera definida en pthreads, es decir `pthread_barrier_t`).
   - La implementación debe ser reutilizable, es decir se pueda usar el mismo objeto en múltiples etapas.
   - El código debe compilar en Linux con `gcc` y la opción `-pthread`.
5. Pistas de diseño
   - Capturar la *etapa* actual en una variable local y esperar mientras no cambie.
   - La última en llegar debe: (i) incrementar *etapa*, (ii) resetear `count`, (iii) `broadcast` para despertar a todas.
   - Usar el patrón: *lock* → modificar estado / decidir → *wait/broadcast* → *unlock*.

Actividad 2: Aplicación de verificación (1.0 puntos) KPIs 1.2 y 1.3

Implemente un programa `main.c` que verifique el comportamiento de su barrera. Debe:

1. Crear N hebras (parámetro de línea de comandos, por defecto $N = 5$).
2. Ejecutar E *etapas* (parámetro, por defecto $E = 4$). En cada etapa, cada hebra:
   a) Realiza trabajo simulado (`usleep` con ± aleatorio).
   b) Imprime antes de esperar: `[tid] esperando en etapa` e.
   c) Llama a *wait()*.
   d) Imprime después de pasar: `[tid] paso barrera en etapa` e.
3. Verificación:
   - Para cada etapa e, todas las hebras deben imprimir el mensaje "pasó barrera en etapa e)" después de que **todas** hayan impreso el mensaje "esperando en etapa e".
   - La secuencia de impresiones puede intercalarse entre las hebras, pero ninguna hebra debe pasar la barrera de la etapa e a la $e+1$ antes de que todas hayan llegado al punto de encuentro.

Parte II. Simulador simple de Memoria Virtual

1. Objetivos

   - Implementar un simulador secuencial de memoria virtual que procese una traza de páginas virtuales.
   - Soportar el algoritmo de reemplazo Reloj.
   - Medir y reportar el número de fallos de página y la tasa de fallos para distintas parámetros de configuración.

2. Descripción

   En esta parte debe implementar un simulador, `sim` con paginación simple y soporte de algoritmo de reemplazo Reloj con un solo puntero.

Actividad 1: Simulador de Traducción de Direcciones

Implemente un simulador, `sim`, que lea una traza de *direcciones virtuales* (una por línea, en decimal o `0xHEX`), realice la traducción a direcciones físicas usando *paginación simple*, y reporte *el número de fallos de página* y la *dirección física* resultante. Mantenga un conjunto de *marcos* físicos de tamaño fijo y use una política simple de asignación: si hay marcos libres, asignar; si no, reemplazar usando el algoritmo del *Reloj*.

1. Definición: Traducción en hexadecimal
   Sea una dirección virtual DV en hexadecimal, con un tamaño de marco de página PAGE_SIZE = 2^b bytes. Sea además MASK = $2^b - 1$ el valor que permite extraer los bits de desplazamiento (*offset*) dentro de la página. Recuerde que en notación binaria, la dirección virtual se divide en componente nvp y offset:

   - Descomposición de la dirección virtual:

     $$\text{offset} = \text{VA} \;\&\; \text{MASK},$$

     $$\text{npv} = \text{DV} \gg b.$$

     donde & denota la operación *AND* bit a bit y $\gg b$ el desplazamiento a la derecha de b bits.

   - Cálculo de la dirección física:
     Si la tabla de páginas asocia la página virtual nvp a un marco físico marco, entonces la dirección física (DF) se obtiene como:

     $$\text{DF} = (\text{marco} \ll b) \mid \text{offset},$$

     donde $\ll b$ representa el desplazamiento a la izquierda (equivalente a multiplicar por 2^b), y | la operación *OR* bit a bit.

2. Reglas de traducción

   - DV compuesta por npv y offset usando PAGE_SIZE.
   - Si npv está en la *Tabla de páginas* hay un *HIT*: luego, se obtiene marco y computa DF.
   - Si no está hay un *FALLO*: luego, si hay marco libre asignarlo a pv; en otro caso, expulsar víctima según algoritmo del *Reloj*. Actualizar la tabla de páginas (invalidar npv expulsada, mapear npv con fallo a marco).

3. Parámetros del simulador

```
./sim Nmarcos tamañomarco [--verbose] traza.txt

Donde Nmarcos corresponde al número de margos disponibles, tamaño
del marco en bytes y verbose es opcional e indica la secuencia paso a paso en
el proceso de traducción. En caso de no realizarse solo reportar el número
de páginas.

# Ejemplo:
./sim 8 4096 traza.txt
```

4. Salida de simulación
   - Totales: Referencias, Fallos de página, y Tasa de fallos.
   - Con `-verbose`: por cada DV, imprimir: DV, nvp, offset, `HIT/FALLO`, marco usado, DF calculada.
5. Datos de prueba
   - Genere entre 5 y 10 referencias y verifique sus resultados.
   - Use los dos archivos de referencias en traces.zip y realice un análisis experimental para ver como se comporta la tasa de fallos variando el número de marcos disponibles.
   - Considere los siguientes rangos de marcos de páginas $(8, 16, 32)$ y tamaño de marco de 8 para trace1.txt y 4096 para trace2.txt.

Entregables

- Sus implementaciones deben ser propocionadas en un repositorio por grupo.
- Su repositorio debe contener un Readme donde se proporcione información de compilación y modo de uso.
- Un informe que describa los principales componentes de su implementación y su evaluación de resultados.