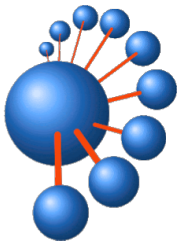

UNIVERSIDAD DE CONCEPCIÓN

Departamento de Ingeniería Informática y

Ciencias de la Computación

Tarea 2: Sistemas Operativos

Sincronización con Barrera Reutilizable y Simulador de Memoria Virtual

Profesor: Juan Felipe González
Ayudante: Dreyko Paredes Mansilla
Asignatura: Sistemas Operativos
Estudiantes:

- Joaquín Ávalos Panes
- Matias López Jara
- Joaquín San Martín Vargas
- Matías Salgado Orellana

Concepción, 01 de Diciembre de 2025

1. Introducción

Este trabajo aborda dos componentes centrales del curso de Sistemas Operativos. La primera parte consiste en la implementación de una barrera reutilizable para la sincronización de hebras, utilizando exclusivamente `pthread_mutex_t` y `pthread_cond_t`. El propósito de esta barrera es asegurar que todas las hebras participantes alcancen un mismo punto de ejecución antes de continuar, permitiendo además que el mecanismo se utilice en múltiples etapas sucesivas. Posteriormente, su funcionamiento se verifica mediante ejecuciones controladas.

La segunda parte del trabajo corresponde al desarrollo de un simulador de memoria virtual basado en paginación simple. El simulador procesa direcciones virtuales obtenidas desde una traza, determina su página y desplazamiento, y gestiona los fallos de página aplicando el algoritmo de reemplazo Reloj. De esta forma, es posible analizar cómo influye la cantidad de marcos físicos disponibles en la tasa de fallos producida durante la ejecución.

2. Desarrollo

2.1. Parte I: Barrera Reutilizable

2.1.1. Estructura del Monitor

La barrera se implementó como un monitor que agrupa tanto el estado interno como las herramientas necesarias para sincronizar a las hebras. La idea es que ninguna hebra avance hasta que todas hayan llegado al mismo punto. Para esto, la estructura de la barrera guarda el número de hebras que han llegado, la etapa actual y las primitivas de sincronización (un mutex y una variable de condición). Esto permite coordinar a varias hebras sin que ocurran condiciones de carrera.

```
typedef struct {
    int count;              // Hebras en etapa actual
    int N;                  // Total de hebras
    int etapa;              // Identificador de etapa
    pthread_mutex_t lock;   // Exclusion mutua
    pthread_cond_t cond;    // Variable de condicion
} barrier_t;
```

Listing 1: Estructura barrier_t

2.1.2. Operación Principal: `barrier_wait`

La función que realiza la sincronización es `barrier_wait`. Cada hebra que llega a la barrera toma el mutex, guarda la etapa actual en una variable local y aumenta el contador. Si esa hebra es la última en llegar, avanza la etapa, reinicia el contador y despierta a todas las hebras que estaban esperando. Si no es la última, se queda esperando mientras la etapa no cambie. Después de eso, libera el mutex y continúa.

```
void barrier_wait(barrier_t *b) {
    pthread_mutex_lock(&b->lock);

    int etapa_local = b->etapa;   // Capturar etapa actual
    b->count++;

    if (b->count == b->N) {       // Ultima hebra
        b->etapa++;               // Avanzar etapa
        b->count = 0;             // Resetear contador
        pthread_cond_broadcast(&b->cond);  // Despertar todas
```

```
    } else {
        while (etapa_local == b->etapa) {
            pthread_cond_wait(&b->cond, &b->lock);
        }
    }

    pthread_mutex_unlock(&b->lock);
}
```

Listing 2: Implementación de barrier_wait

2.1.3. Componentes clave

- El mutex protege el acceso a las variables internas.
- Se usa una variable local para detectar cambios de etapa.
- El patrón `while` evita despertares espurios.
- El uso de `broadcast` despierta a todas las hebras al mismo tiempo.

2.2. Parte II: Simulador de Memoria Virtual

El simulador implementa un modelo sencillo de paginación. Para cada dirección virtual leída desde la traza, el programa determina su número de página y su offset, y revisa si esa página está cargada en memoria. Si no lo está, ocurre un fallo y la página se carga en un marco libre o, si no quedan marcos disponibles, se reemplaza una página utilizando el algoritmo Reloj. Durante toda la ejecución se cuentan referencias, fallos y la tasa final de fallos.

2.2.1. Estructura de los marcos

Cada marco físico almacena:

- el número de página virtual cargada,
- un bit de uso que indica si la página fue accedida recientemente.

```
typedef struct {
    uint64_t pagina;   // Pagina virtual cargada
    int usado;         // Bit de uso (algoritmo Reloj)
} marco_t;
```

Listing 3: Estructura marco_t

2.2.2. Traducción de direcciones

Para una dirección virtual `DV` y una página de tamaño $P = 2^b$:

- `offset = DV & (P - 1)`
- `nvp = DV >> b`
- `DF = (marco << b)  offset`

2.2.3. Manejo de hits, fallos y marcos libres

- Si la página está cargada → *hit*.
- Si no está → fallo.
- Si hay marco libre → se usa.
- Si no hay → se aplica el algoritmo Reloj.

En un hit, el bit de uso se marca en 1.

2.2.4. Algoritmo de reemplazo Reloj

El puntero recorre los marcos circularmente:

1. Si el bit `usado` es 1 → se limpia y avanza.
2. Si el bit es 0 → ese marco se reemplaza.
3. La nueva página se marca como usada.

```
// Al acceder: marcar usado = 1
if (HIT) {
    frames[marco].usado = 1;
}

// Al reemplazar: buscar usado = 0
while (frames[puntero].usado == 1) {
    frames[puntero].usado = 0;  // Segunda oportunidad
    puntero = (puntero + 1) % N;
}

// Reemplazar victima
frames[puntero].pagina = nvp;
frames[puntero].usado = 1;
puntero = (puntero + 1) % N;
```

Listing 4: Fragmento del algoritmo Reloj

2.2.5. Componentes clave

- La separación en `nvp` y `offset` permite traducir correctamente las direcciones virtuales.
- El bit `usado` refleja si la página fue accedida recientemente.
- El puntero circular recorre los marcos en un orden estable.
- La segunda oportunidad evita reemplazar páginas activas.
- El primer marco con `usado = 0` se selecciona como víctima.
- El conteo de `hits` y fallos permite analizar el rendimiento del algoritmo.

3. Resultados y Evaluación

3.1. Parte I: Verificación de la Barrera

Se ejecutó el programa de prueba con 5 hebras y 4 etapas. La salida obtenida fue:

```
Creando 5 hebras, 4 etapas...
[H0] esperando en etapa 0
[H3] esperando en etapa 0
[H2] esperando en etapa 0
[H1] esperando en etapa 0
[H4] esperando en etapa 0
[H4] paso barrera en etapa 0
[H1] paso barrera en etapa 0
[H3] paso barrera en etapa 0
[H2] paso barrera en etapa 0
[H0] paso barrera en etapa 0
[H4] esperando en etapa 1
[H0] esperando en etapa 1
[H2] esperando en etapa 1
[H1] esperando en etapa 1
[H3] esperando en etapa 1
[H3] paso barrera en etapa 1
...
```

Listing 5: Ejecución: 5 hebras, 4 etapas

Evaluación. En cada etapa, las cinco hebras imprimen primero el mensaje `esperando en etapa X`. Sólo después de que todas han llegado al punto de encuentro aparece algún mensaje `paso barrera en etapa X`. La ejecución confirma que:

- Ninguna hebra avanza prematuramente.
- La barrera se reutiliza correctamente entre etapas.
- El contador y la variable de condición operan como se espera.

Esto valida la correctitud funcional de la barrera.

3.2. Parte II: Evaluación del Simulador

3.2.1. Experimentos Realizados

Se ejecutaron experimentos con dos trazas de 8192 referencias cada una. Cada traza contiene 8192 referencias. Se evaluaron tres configuraciones de marcos (8, 16 y 32) para dos tamaños de página: 8 bytes (trace1) y 4096 bytes (trace2). Los resultados son::

Cuadro 1: Resultados para trace1.txt (PAGE_SIZE = 8 bytes)

Marcos	Referencias	Fallos	Tasa
8	8192	8073	0.9855
16	8192	7943	0.9696
32	8192	7713	0.9415

Cuadro 2: Resultados para trace2.txt (PAGE_SIZE = 4096 bytes)

Marcos	Referencias	Fallos	Tasa
8	8192	7649	0.9337
16	8192	7138	0.8713
32	8192	6142	0.7498

3.2.2. Análisis de los resultados

1. Impacto del número de marcos. Al aumentar el número de marcos, ambas trazas muestran una reducción en la tasa de fallos. El efecto es más pronunciado en `trace2`, donde la disminución entre 8 y 32 marcos es del 18.4 %.

- trace1: 0.9855 → 0.9415 (reducción de 4.4 %)
- trace2: 0.9337 → 0.7498 (reducción de 18.4 %)

Esto indica que la segunda traza presenta mejor localidad y se beneficia más de un número mayor de marcos.

2. Efecto del tamaño de página. Con 32 marcos, el contraste entre ambas trazas es evidente:

trace1: 0,9415 vs trace2: 0,7498

El tamaño mayor de página de `trace2` (4096 bytes) agrupa más direcciones dentro de una misma página, reduciendo el número de cambios de página y aprovechando mejor la localidad espacial.

3. Comportamiento del algoritmo Reloj. Los resultados muestran que:

- El algoritmo entrega reducciones consistentes al aumentar marcos.
- La segunda oportunidad evita expulsiones innecesarias de páginas activas.
- En ambas trazas, el algoritmo se comporta de forma estable y sin oscilaciones.

Aunque su peor caso es $O(N)$ por la búsqueda secuencial del puntero, en la práctica recorre pocas posiciones porque el bit de uso proporciona una buena heurística.

4. Conclusiones

Logros principales

Parte I – Barrera reutilizable

- Se implementó correctamente una barrera reutilizable usando solo mutex y variables de condición, cumpliendo con las restricciones de la tarea.
- Las pruebas mostraron que las hebras siempre esperan a que todas lleguen a la etapa antes de avanzar. En ningún caso se observó que una hebra continuara antes de tiempo.
- La barrera funcionó de forma estable durante varias etapas, lo que confirma que el cambio de etapa y el reseteo interno están bien diseñados.

Parte II – Simulador de memoria virtual

- Se desarrolló correctamente un simulador capaz de leer direcciones virtuales, extraer página y offset, y manejar fallos usando el algoritmo Reloj.
- Los resultados evidencian que al aumentar los marcos desde 8 hasta 32, la tasa de fallos disminuye en ambas trazas, siendo más notorio en la segunda.
- El tamaño de página también tuvo un efecto importante: con páginas de 4096 bytes, la cantidad de fallos bajó notablemente comparado con páginas de 8 bytes, especialmente en configuraciones con más marcos.

Observaciones técnicas

- El uso de `while` junto a `pthread_cond_wait` fue clave para evitar problemas por despertares espurios y asegurar que cada hebra realmente espere el cambio de etapa. Guardar la etapa en una variable local evitó condiciones de carrera.
- El algoritmo Reloj se comportó de forma estable. Aunque en teoría puede recorrer todos los marcos, en la práctica rara vez fue necesario porque el bit de uso ayuda a identificar rápidamente páginas activas o inactivas.
- La diferencia entre las dos trazas muestra que la localidad espacial influye mucho en este tipo de simulación: cuando las referencias se agrupan mejor dentro de una misma página, se reducen los fallos.

Limitaciones y trabajo futuro

Limitaciones

- La barrera no incluye un sistema para detectar hebras que podrían quedar bloqueadas o nunca llegar a la sincronización.
- El simulador no modela elementos como TLB ni tiempos reales de acceso a disco, por lo que las mediciones representan solo la lógica del reemplazo, no el costo total de una traducción real.

Líneas de mejora

- Agregar una TLB pequeña para ver qué tan útil es guardar traducciones recientes en vez de consultar siempre la tabla de páginas.
- Probar otros algoritmos de reemplazo (como FIFO, LRU o Second-Chance) usando las mismas trazas y comparar si es que cambian los resultados.
- Revisar a futuro cómo el *working set* de cada traza se relaciona con los fallos que obtuvimos.