Facultad de Ingeniería
Universidad de Concepción

Departamento Ingeniería Informática y Ciencias de la Computación

Ingeniería Civil Informática

501204-1: Sistemas Operativos

Tarea 2

Ignacio Contardo - Francisco Arentsen - Cristobal Figueroa

Introducción:

El presente informe detalla el desarrollo y los resultados de las 2 partes en que se divide la tarea: **Sincronización con Barrera Reutilizable** y **Simulador Simple de Memoria Virtual**. La **Parte I** aborda un problema fundamental de la programación concurrente: el diseño e implementación de una primitiva de sincronización capaz de coordinar a N hebras en múltiples puntos de encuentro.
Esta sección incorpora la conceptualización de **Monitor** para garantizar la exclusión mutua y la espera condicional eficiente, utilizando exclusivamente las primitivas pthread_mutex_t y pthread_cond_t.

La **Parte II** se enfoca en el desarrollo de un simulador que modela el manejo de memoria por un sistema operativo bajo un esquema de **Paginación de un Nivel**. El objetivo es evaluar el rendimiento del sistema al procesar una traza de referencias a direcciones virtuales. La implementación incluye los procedimientos para la **traducción de direcciones** y la gestión de fallos de página, incorporando el algoritmo de reemplazo **del Reloj** para la selección de la página víctima. Finalmente, el informe analiza métricas clave, como el número total de Fallos de Página y la Tasa de Fallos, variando parámetros del simulador como son el número de marcos de página y el tamaño de los marcos de página para un análisis experimental.

Facultad de Ingeniería
Universidad de Concepción

Objetivos del Trabajo:

Primera Parte:

- 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.

Segunda Parte:

- 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.

Facultad de Ingeniería
Universidad de Concepción

Desarrollo de la Primera Parte:

Actividad I:

Funcionalidades:

`int reusable_barrier_init()`: Se utiliza principalmente para configurar el estado inicial del monitor y se encarga de qué las hebras tengan un valor positivo. Y realiza las llamadas para preparar las primitivas de sincronización del sistema operativo.

`void reusable_barrier_wait()`: Encapsula la lógica del monitor descrita anteriormente: gestión del *lock*, evaluación de la condición de espera (count < N), y ejecuta el broadcast qué elimina arriba la última hebra (si count == N).

`int reusable_barrier_destroy()`: Realiza la limpieza de recursos. Para liberar la memoria de las primitivas de sincronización, cuando la barrera ya no va a ser utilizada.

Actividad II:

Para la primera actividad de la parte 1 de la tarea, se desarrolló una barrera reutilizable siguiendo el patrón de monitor, en que se garantiza tanto la exclusión mutua como la espera condicional. La implementación considera los siguientes componentes encapsulados en la struct reusable_barrier que engloban tanto el estado interno del monitor como sus primitivas de sincronización:

Componente	**Variable/Primitiva**	**Propósito**
Exclusión Mutua	pthread_mutex_t mutex	Protege el acceso concurrente a las variables count y etapa.
Espera Condicional	pthread_cond_t cond	Variable de condición que permite que las hebras que llegan antes del punto de encuentro "duerman" sin consumir CPU
Estado Compartido	int N int count Int etapa	N almacena el número total de hebras que deben sincronizarse, count es el

		contador para las hebras que han llegado a la barrera en la fase actual y etapa identifica la fase e incrementa en cada liberación, siendo clave para poder reutilizar la barrera.

Además se incluyeron las funciones reusable_barrier_init y resuable_barrier_destroy para inicializar y limpiar la barrera respectivamente y la función reusable_barrier_wait cuyo propósito es sincronizar a todas las hebras concurrentes que la invocan, deteniendo su ejecución hasta que todas hayan llegado al punto de encuentro. Para lograr esto reusable_barrier_wait lleva a cabo el siguiente proceso (siguiendo el patrón sugerido **lock -> modificar estado/decidir -> wait/bradcast -> unlock**):

1. **Exclusión Mutua:** Toda la función se ejecuta bajo pthread_mutex_lock() y pthread_mutex_unlock().
2. **Mecanismo de Reutilización:** Al inicio, cada hebra obtiene el valor actual de la fase en una variable local (int current_etapa). Esto es para manejar la liberación y asegurar que una hebra liberada no intente reentrar (condición de carrera del último hilo) prematuramente en la misma fase.
3. **Hebras en Espera:** Si la hebra no es la última, incrementa count y se bloquea usando pthread_cond_wait(). La espera se mantiene dentro de un bucle while para manejar despertares espurios y garantizar que la hebra sólo continúe cuando la variable global etapa haya sido modificada por la última hebra.
4. **Liberación de la Barrera (count == N):** La última hebra en llegar realiza las tres acciones atómicas necesarias para el reseteo y la liberación:
    - **Incrementar etapa:** Avanza el identificador de fase.
    - **Resetear count:** Inicializa el contador para la siguiente fase.
    - **pthread_cond_broadcast():** Despierta a todas las hebras bloqueadas.

Utilización:

En cada iteración, las hebras simulan carga de trabajos aleatoriamente, antes de invocar la primitiva de sincronización, permitiendo verificar visualmente que ninguna hebra logre cruzar la barra hacia la etapa siguiente.

Resultados de la parte 1:

La siguiente ejecución muestra el comportamiento observado durante el cambio de etapas:

Caso normal:

```
ignaciocontardo@fedora:~/Escritorio/2025-2/Sistemas operativos/120_Tareas/fork_Sitemas_Tarea-2/act1$ ./barrera 2 2
Iniciando prueba con 2 hebras y 2 etapas.
[1] esperando en etapa 0
[0] esperando en etapa 0
[0] paso barrera en etapa 0
[1] paso barrera en etapa 0
[1] esperando en etapa 1
[0] esperando en etapa 1
[0] paso barrera en etapa 1
[1] paso barrera en etapa 1
Todas las etapas finalizadas correctamente.
```

Como se puede observar, ninguna hebra ocurre un cruce de la barrera. Y cuando llega la última hebra, momento en el cual el *broadcast* libera el grupo simultáneamente. La transición exitosa a etapas posteriores es válida a la lógica.

Parte 2:

Desarrollo de la Segunda Parte:

Para el desarrollo del simulador de traducción de direcciones utilizando el algoritmo de reemplazo del reloj, se diseñó un sistema basado en una arquitectura de 32 bits. La implementación se estructura en componentes principales que abarcan la definición de estructuras de datos, la inicialización del sistema, el ciclo de traducción y la lógica de reemplazo de páginas.

Se definieron dos estructuras fundamentales para gestionar el mapeo y el estado de la memoria. La primera es PageTableEntry, que representa una entrada en la tabla de páginas y almacena el mapeo lógico, incluyendo el número de marco físico, y los bits de control como el de validez y el de referencia. La segunda estructura es Marco, que representa un espacio en la memoria física. Esta estructura es crítica para el algoritmo, ya que mantiene el mapeo inverso almacenando que página virtual reside actualmente en el marco, además de una copia del bit de referencia que es consultado y modificado por el puntero del reloj.

El simulador recibe como argumentos de entrada la cantidad de marcos físicos, el tamaño de cada marco y el archivo de traza. Al inicio, el sistema valida que el tamaño del marco sea una potencia de dos y calcula dinámicamente la cantidad de bits necesarios para el offset y para el número de página virtual, si el número de páginas v. Con estos parámetros, se instancia la tabla de páginas, se asigna memoria para el arreglo de marcos físicos y se inicializa el puntero del reloj en cero para comenzar el barrido circular.

El simulador itera línea por línea sobre el archivo de traza. Para cada acceso de memoria, la dirección virtual se descompone en número de página virtual y offset. Posteriormente se consulta la tabla de páginas usando el número de página virtual. Si el bit de validez indica que la página está cargada, se produce un hit, se actualiza el bit de referencia a uno para indicar uso reciente y se retorna la dirección física. Si el bit de validez indica que la página no está en memoria, se incrementa el contador de fallos y el sistema busca un marco libre. Si no existe espacio disponible, se invoca al algoritmo de reemplazo.

Cuando la memoria está llena, la función del algoritmo de reloj determina que página desalojar utilizando un puntero circular que recorre los marcos físicos. Si el marco apuntado tiene el bit de referencia activado, se le da una segunda oportunidad reiniciando el bit a cero y avanzando el puntero. Si el marco tiene el bit de referencia desactivado, se selecciona como víctima. Una vez seleccionada la víctima, se invalida su entrada correspondiente en la tabla de páginas para mantener la consistencia del sistema. Finalmente, la nueva página se carga en dicho marco, se actualiza la tabla con el nuevo mapeo físico y el bit de referencia se inicializa en uno.

El simulador concluye reportando las estadísticas de rendimiento, específicamente el número total de accesos, la cantidad de fallos de página y la tasa de fallos resultante. Adicionalmente, se implementó un modo verbose opcional para facilitar la depuración y visualización paso a paso del estado de la memoria durante la ejecución.

Análisis y Evaluación de los Resultados:

Resultados Parte 2:

Haciendo una prueba experimental con el siguiente archivo de texto, podemos ver que el programa funciona de manera correcta a través de la opción verbose.

trazas.txt

```
0x000100
0x001000
0x000100
0x002000
0x000100
```

```
mai@Cristbal:~/Tarea2-Sistemas-Operativos/act2$ ./sim 3 4096 --verbose ./trazas.txt
========================
Marcos físicos: 3
Tamaño de marco: 4096
Bits de offset: 12
========================
Acceso #1: DV = 256 (0x100)
FALLO: Página 0 no está en memoria. Asignando marco 0
NVP = 0, offset = 256
Direccion física = 256, (0x100)FALLO DE PÁGINA

Acceso #2: DV = 4096 (0x1000)
FALLO: Página 1 no está en memoria. Asignando marco 1
NVP = 1, offset = 0
Direccion física = 4096, (0x1000)FALLO DE PÁGINA

Acceso #3: DV = 256 (0x100)
HIT : Página 0 ya está en marco 0
NVP = 0, offset = 256
Direccion física = 256, (0x100)HIT

Acceso #4: DV = 8192 (0x2000)
FALLO: Página 2 no está en memoria. Asignando marco 2
NVP = 2, offset = 0
Direccion física = 8192, (0x2000)FALLO DE PÁGINA

Acceso #5: DV = 256 (0x100)
HIT : Página 0 ya está en marco 0
NVP = 0, offset = 256
Direccion física = 256, (0x100)HIT


===RESULTADOS===
Total de accesos: 5
Fallos de página: 3
Tasa de fallos: 60.00%
================
```

Luego, analizaremos trace1.txt con un tamaño de 8 bytes para el marco y rangos de 8, 16 y 32 para los números de marco tenemos lo siguiente:

```
mai@Cristbal:~/Tarea2-Sistemas-Operativos/act2$ ./sim 8 8 trace1.txt

===RESULTADOS===
Total de accesos: 8192
Fallos de página: 8073
Tasa de fallos: 98.55%
================
mai@Cristbal:~/Tarea2-Sistemas-Operativos/act2$ ./sim 16 8 trace1.txt

===RESULTADOS===
Total de accesos: 8192
Fallos de página: 7943
Tasa de fallos: 96.96%
================
mai@Cristbal:~/Tarea2-Sistemas-Operativos/act2$ ./sim 32 8 trace1.txt

===RESULTADOS===
Total de accesos: 8192
Fallos de página: 7713
Tasa de fallos: 94.15%
================
```

Como podemos observar, al aumentar el número de marcos disminuye la tasa de fallos, sin embargo la tasa de fallos es muy alta en todos los casos, puesto que al tener un tamaño pequeño para los marcos entonces no es posible aprovechar eficientemente la localidad temporal.

Finalmente, para trace2.txt con un tamaño de 4Kb para el marco y rangos de 8, 16 y 32 para los números de marco tenemos lo siguiente:

```
mai@Cristbal:~/Tarea2-Sistemas-Operativos/act2$ ./sim 8 4096 trace2.txt

===RESULTADOS===
Total de accesos: 8192
Fallos de página: 7649
Tasa de fallos: 93.37%
================
mai@Cristbal:~/Tarea2-Sistemas-Operativos/act2$ ./sim 16 4096 trace2.txt

===RESULTADOS===
Total de accesos: 8192
Fallos de página: 7138
Tasa de fallos: 87.13%
================
mai@Cristbal:~/Tarea2-Sistemas-Operativos/act2$ ./sim 32 4096 trace2.txt

===RESULTADOS===
Total de accesos: 8192
Fallos de página: 6142
Tasa de fallos: 74.98%
================
```

Observamos nuevamente que, al aumentar el número de marcos disminuye la tasa de fallos, sin embargo esta vez la tasa de fallos disminuyó más que en el análisis anterior, esto porque al aumentar el tamaño del marco de página la probabilidad de que dos direcciones estén en el mismo marco aumenta.

Conclusión:

En conclusión, el presente trabajo ha abordado con éxito los objetivos planteados en la Tarea 2, logrando desarrollar e implementar soluciones funcionales en dos áreas fundamentales de los Sistemas Operativos: la **sincronización concurrente** al crear con éxito una Barrera Reutilizable siguiendo el patrón de Monitor y la **gestión de memoria virtual** al elaborar un simulador de **Paginación de un Nivel**, funcional que procesa trazas de referencias a direcciones virtuales e incorpora el mecanismo de traducción de direcciones y el algoritmo de reemplazo de Reloj.

LINK REPOSITORIO GITHUB DEL PROYECTO:
“https://github.com/Farentsen999/Tarea1-Sistemas-Operativos.git”