"Sigan brillando donde quieran que aterricen"

By

Valentina Muñoz Rincón, valentina.munozr1@udea.edu.co

Juan Felipe Escobar Rendón, juan.escobar15@udea.edu.co

Ricardo Contreras Garzón, ricardo.contreras1@udea.edu.co

Santiago Arenas Gómez, santiago.arenas1@udea.edu.co

Jonathan David Fernández, jonathand.fernandez@udea.edu.co

Valeria Álvarez Fernández valeria.alvarezf@udea.edu.co

Permite que varias tareas parezcan ejecutarse al mismo tiempo (aunque en realidad se turnan si hay un solo procesador)

Concurrencia

Es cuando varias tareas realmente se ejecutan al mismo tiempo (en varios núcleos).

Paralelismo

Concurrencia e Hilos

Los hilos que comparten datos pueden generar condiciones de carrera (resultados inesperados por acceso concurrente). → Para evitarlo, se usan mecanismos como locks (bloqueos) para proteger regiones críticas.

Problema...

- Aprovechar varios núcleos (hacer programas más rápidos).
- No bloquear el programa cuando hay tareas lentas (por ejemplo, espera de red o disco).

Es una "subdivisión" dentro de un proceso que tiene su propio contador de programa y stack, pero comparte memoria y datos con otros hilos del mismo proceso.

Hilo

Locks

Los locks tienen dos operaciones básicas:

`acquire()` `release()`

Se usan locks para garantizar exclusión mutua, es decir, que solo un hilo entre a la sección crítica a la vez.

Solución...

Cuando varios hilos acceden y modifican datos compartidos al mismo tiempo, pueden ocurrir errores (condiciones de carrera).

Problema...

Los locks son mecanismos de sincronización que se usan para proteger secciones críticas en un programa multihilo.

Variables de condición

Operaciones básicas de variables de condición

`wait()` `signal()`

Objetivo → Coordinar el orden de ejecución entre hilos y evitar que accedan a recursos cuando no deben (ej. buffer vacío o lleno).

Son mecanismos que permiten que un hilo espere (se "duerma") hasta que una condición se cumpla, y otro hilo pueda "despertarlo" cuando sea necesario.

Problema...

Usar **if** en lugar de **while** para verificar la condición puede generar errores y bloqueos (hilos se quedan dormidos).

Solución...

Siempre usar **while** para volver a chequear la condición al despertar. Usar dos variables de condición (por ejemplo empty y fill) para evitar despertar hilos incorrectos.

Semáforos

Son primitivas de sincronización que permiten controlar el acceso a recursos compartidos entre hilos. Cada semáforo tiene un valor entero, que representa el número de recursos disponibles.

Semáforo binario

Valor inicial 1; funciona como un lock (permite solo un hilo).

Semáforo general (contador)

Valor inicial N; permite que hasta N hilos entren.

Operaciones básicas de variables de condición

`wait()` `signal()`

Ventajas

- Pueden reemplazar locks y variables de condición.
- Son versátiles para diferentes problemas de concurrencia.

Problema de concurrencia

Son primitivas de sincronización que permiten controlar el acceso a recursos compartidos entre hilos. Cada semáforo tiene un valor entero, que representa el número de recursos disponibles.

Tipos principales:

Violación de atomicidad

Dos o más hilos acceden a la misma variable compartida sin la protección adecuada, causando resultados incorrectos.

Solución: usar locks.

Violación de orden

El orden esperado de ejecución entre operaciones se rompe (una operación ocurre antes de otra que debería ir primero).

Solución: usar variables de condición para forzar el orden.

Interbloqueos

Ocurren cuando varios hilos se quedan esperando recursos entre sí, sin poder avanzar. Se presentan cuando se cumplen estas 4 condiciones:

- Exclusión mutua.
- Retención y espera.
- No apropiación (no se pueden quitar recursos).
- Espera circular.

Formas de prevención

- **Evitar espera circular: definir un orden fijo para adquirir locks.**
- **Retención y espera: adquirir todos los locks al mismo tiempo.**
- **No apropiativo: usar trylock() y reintentar si no se obtiene el lock.**
- **Uso de estructuras wait-free: operaciones atómicas sin locks (ej. compare-and-swap).**