Arquitectura de Sistemas

Práctica 10: Barreras

Gustavo Romero López

Updated: 16 de mayo de 2025

Departamento de Ingeniería de Computadores, Automática y Robótica

Objetivos

- Probar el rendimiento de varias implementaciones de una barrera.
- Vamos a probar algunas que funcionan y otras que no.
- Escriba su propia solución e intente que sea competitiva.
- Compruebe la funcionalidad y el rendimiento de las diferentes versiones con: `make stat` y `make sort`.

Semáforos (POSIX)

`#include <semaphore.h>` Cabecera C/C++.

`sem_t` Tipo semáforo.

`sem_init(sem, attr, valor)` Inicializa el semáforo `sem` al calor `valor` con los atributos `attr`.

`sem_destroy(sem)` Destruye el semáforo `sem`.

`sem_wait(sem)` Si el valor del semáforo `sem` es positivo lo decrementa y retorna inmediatamente. En otro se bloquea hasta poder hacerlo.

`sem_trywait(sem)` Versión no bloqueante de `sem_wait(sem)`. En cualquier caso retorna inmediatamente. Es necesario comprobar la salida antes de continuar.

`sem_post(sem)` Incrementa el valor del semáforo `sem`. En caso de cambiar a un valor positivo desbloquea a alguno de los llamadores bloqueados en `sem_wait(sem)`.

Pthreads: API de barreras

`pthread_barrier_t` Tipo barrera.

`pthread_barrier_init(barrier, attr, n)` Inicializa la barrera `barrier` con los atributos `attr` para que funcione con `n` hebras.

`pthread_barrier_destroy(barrier)` Destruye la barrera `barrier`.

`pthread_barrier_wait(barrier)` Bloque a la hebra llamadora hasta que se `n` hebras ejecuten `pthread_barrier_wait(barrier)`.

Pthreads: API de variables condición

`pthread_cond_t` Tipo variable condición. Inicializable a PTHREAD_COND_INITIALIZER.

`pthread_cond_init(cond,attr)` Inicializa la variable condición `cond` con los atributos `attr`.

`pthread_cond_destroy(cond)` Destruye la variable condición `cond`.

`pthread_cond_wait(cond, mutex)` Bloque a la hebra llamadora hasta que se señale `cond`. Esta función debe llamarse mientras `mutex` está ocupado y ella se encargará de liberarlo automáticamente mientas espera. Después de la señal la hebra es despertada y el cerrojo es ocupado de nuevo. El programador es responsable de desocupar `mutex` al finalizar la sección crítica para la que se emplea.

`pthread_cond_signal(cond)` Función para despertar a otra hebra que espera que se señale sobre la variable condición `cond`. Debe llamarse después de que `mutex` esté ocupado y se encarga de liberarlo en `pthread_cond_wait(cond, mutex)`.

`pthread_cond_broadcast(cond)` Igual que la función anterior para el caso de que queramos desbloquear a varias hebras que esperan.

C++11 thread: API de variables condición

`#include <condition_variable>` Cabecera.

`std::condition_variable` Nombre de la clase.

`wait(lock, pred)` Bloquea a la hebra llamadora hasta que se señale la condición o se produzca un despertar engañoso. Mediante `pred` podemos asegurarnos de que la hebra ha sido despertada tras utilizar `notify_*()`.

`notify_one()` Función para despertar a otra hebra que espera que se señale sobre la variable condición.

`notify_all()` Función para despertar a todas las hebras que esperan que se señale sobre la variable condición.

barrera.cc

Copie el programa barrera.cc y **verifique** que la secuencia de ejecución no es correcta.

```
class barrera_t
{
public:
    barrera_t(size_t l) : limite(l) {}
    void esperar() {}
private:
    size_t limite;
} barrera(N);
```

barrera.cc

```
void hebra(size_t yo)
{
    std::string antes = std::to_string(yo) + ": antes\n",
                despues = std::to_string(yo) + ": después\n";
    while (true)
    {
        std::cout << antes;
        barrera.esperar();
        std::cout << despues;
    }
}
```

Copie el programa mutex.cc, **verifique** que la secuencia de ejecución no es correcta y **repárelo**.

```
class barrera_t
{
public:
    barrera_t(size_t l) : limite(l) {}

    void esperar()
    {
        size_t local = uso;

        m.lock();
        ++en_espera[local];
        m.unlock();
```

mutex.cc II

```
        if (en_espera[local] != limite)
        {
            while (en_espera[local] != 0);
        }
        else
        {
            uso ^= 1;
            en_espera[local] = 0;
        }
    }

private:
    std::mutex m;
    size_t en_espera[2] = {0, 0}, uso = 0, limite;
} barrera(N);
```

Pruebe las barreras existentes

- `pthread_barrier_t`
- `std::barrier`
- `boost::barrier`

Compare sus soluciones con estas para medir su calidad.

Cree sus propias versiones de un barrera

- Elimine la espera ocupada mediante...
  - `yield()`
  - marcha atrás exponencial
  - los métodos `wait()` de algunas clases como `std::atomic` y `std::mutex`
- Escriba alguna versión nueva basada en...
  - `std::atomic`
  - `std::semaphore` o `std::binary_semaphore`
  - `std::mutex` y `std::contition_variable`

¿Qué resultado han dado sus esfuerzos?

- ¿No comprede alguna solución?
- ¿Hay algún error que se le resiste?
- ¿Ha mejorado alguna solución?
- ¿Emplea menos tiempo?
- ¿Imprime más mensajes?
- ¿Su ratio es mejor?