Arquitectura de Sistemas

Práctica 10: Barreras

Gustavo Romero López

Updated: 20 de febrero de 2019

Arquitectura y Tecnología de Computadores

Objetivos

- Aprender a utilizar **variables condición**.
- Diseñar e implementar una **barrera**:
  - Barrera que no funciona (sólo probar)
  - Barrera que funciona sólo una vez (sólo probar).
  - Barrera con espera ocupada (sólo probar).
  - Barrera sin espera ocupada.
  - Barrera con variables condición.
  - Barrera de Pthread.
- Comprobar la funcionalidad y el rendimiento de las diferentes versiones con: `make all`.

makefile

```
EXE = $(basename $(SRC))
DAT = $(EXE:=.dat)
LOG = $(EXE:=.log)
SEE = $(EXE:=.see)

#CXXFLAGS = -march=native -O3 -pthread --std=c++1z
    -Wall -Wl,--no-as-needed
CXXFLAGS = -lboost_thread -march=native -O3 -
    pthread --std=c++17 -Wall -Wl,--no-as-needed

default: stat
```

Con `make all` o `make stat` podrá comparar fácilmente los resultados producidos por las soluciones que vaya programando.

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.

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

```
class barrera_t
{
public:
  barrera_t(unsigned _limite) {}
  void esperar() {}
} barrera(N);

void msg(int yo, const char *txt)
{
  static std::mutex m;
  std::unique_lock<std::mutex> l(m);
  std::cout << yo << ": " << txt << std::endl;
}

//-------------------------------------------------------

void hebra(int yo)
{
  while(true)
  {
```

Copie el programa barrera-una.cc y **verifique** que la secuencia de ejecución sólo es correcta la primera vez.

```
class barrera_t
{
public:
  barrera_t(unsigned _limite): contador(0), limite(_limite) {}

  void esperar()
  {
    m.lock();
    ++contador;
    m.unlock();
    while (contador < limite);
  }

private:
  std::mutex m;
  int contador, limite;
} barrera(N);
```

Copie el programa barrera-ceo.cc y **verifique** que la secuencia de ejecución es correcta.

```
class barrera_t
{
public:
  barrera_t(unsigned _limite): en_espera{0, 0}, uso(0), limite(_limite) {}

  void esperar()
  {
    unsigned uso_local = uso;

    m.lock();
    ++en_espera[uso_local];
    m.unlock();

    if (en_espera[uso_local] == limite)
    {
      uso = 1 - uso;
      en_espera[uso_local] = 0;
    }
    else
    {
      while(en_espera[uso_local] > 0);
```

```
    }
  }

private:
  std::mutex m;
  volatile unsigned en_espera[2], uso;
  unsigned limite;
} barrera(N);
```

Implementación de barrera-seo.cc

- **Copie** barrera-ceo.cc en barrera-seo.cc.
- **Elimine** la espera ocupada de barrera-seo.cc modificando la clase `barrera_t`.
- **Compare** barrera-seo.cc con las soluciones anteriores.

Implementación de barrera-vc.cc

- **Copie** barrera.cc en barrera-vc.cc.
- **Modifique** la clase `barrera_t` de barrera-vc.cc para que funcione empleando variables condición. Puede utilizar la versión de pthread o la de C++11.
- **Compare** barrera-vc.cc con las soluciones anteriores.

Implementación de barrier.cc

- **Copie** barrera.cc en barrier.cc.
- **Modifique** la función `hebra()` para que funcione empleando las barreras de la biblioteca pthread.
- **Compare** barrier.cc con las soluciones anteriores.

Escriba una solución mejor...

- ¿Se le ocurre alguna forma de mejorar alguna de las soluciones propuestas para obtener mejores resultados?
- ¿Menor tiempo?
- ¿Mayor cantidad de mensajes?