Arquitectura de Sistemas

Práctica 11: El problema lectores/escritores

Gustavo Romero López

Updated: 22 de mayo de 2019

Arquitectura y Tecnología de Computadores

Objetivos

- Resolver el problema lectores/escritores de la forma más eficiente y justa posible.
- Para ello utilice cualquiera de los mecanismos de sincronización y exclusión mutua vistos hasta.
- Pruebe los cerrojos específicos de lectura/escritura de pthread y compare los resultados con los suyos.
- Para facilitar el proceso de arranque partiremos de analizar un par de soluciones incorrectas:
  - `le.cc`: condición de carrera + inaninición.
  - `le-mutex.cc`: sincronización demasiado restrictiva.

makefile

```
CXXFLAGS = -I. -march=native -O3 -pthread --std=c++17 -Wall
LDFLAGS  = -Wl,--no-as-needed
LDLIBS   = -lpthread -lrt

all: stat

clean:
  -rm -fv $(DAT) $(EXE) $(LIN) $(LOG) core* *~
  -find -mindepth 2 -maxdepth 2 -name makefile -execdir make $@
      \;
```

Las opciones stat y sort le permitirán comparar fácilmente el rendimiento de las soluciones que vaya programando.

Semáforos (POSIX)

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

`sem_t` Tipo semáforo.

`sem_init(sem, attr, valor)` Crea un semáforo `sem` inicializado a `valor` y 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 algun llamador de `sem_wait(sem)`.

Cerrojos Lector/Escritor (pthread)

`pthread_rwlock_t` Tipo cerrojo lector/escritor.

`pthread_rwlock_init(rwlock, attr)` Inicializa el cerrojo lector/escritor `rwlock` con los atributos `attr`.

`pthread_rwlock_destroy(rwlock)` Destruye el cerrojo `rwlock`.

`pthread_rwlock_rdlock(rwlock)` Adquiere el cerrojo `rwlock` para lectura.

`pthread_rwlock_tryrdlock(rwlock)` Intenta adquirir el cerrojo `rwlock` para lectura.

`pthread_rwlock_wrlock(rwlock)` Adquiere el cerrojo `rwlock` para escritura.

`pthread_rwlock_trywrlock(rwlock)` Intenta adquirir el cerrojo `rwlock` para escritura.

`pthread_rwlock_unlock(rwlock)` Libera el cerrojo `rwlock` en función de la versión de aquisición ejecutada con anterioridad, ya sea lector o escritor.

Semáforos binarios de C++

`#include <mutex>` Cabecera.

`std::mutex` Nombre de la clase.

- `lock()` Adquiere el semáforo. Si otra hebra lo ha bloqueado previamente entonces bloquea a la hebra actual hasta que la propietaria del semáforo lo deja libre.
- `unlock()` Desbloquea el semáforo.

`std::lock_guard` Envoltorio que permite adquirir un semáforo en el bloque de ejecución.

`std::shared_mutex` de C++

`#include <shared_mutex>` Cabecera.

`std::shared_mutex` Nombre de la clase.

- `lock()` Adquiere el semáforo de manera exclusiva. Si otra hebra lo ha bloqueado previamente entonces bloquea a la hebra actual hasta que la propietaria del semáforo lo deja libre.
- `unlock()` Desbloquea el semáforo de manera exclusiva.
- `lock_shared()` Adquiere el semáforo de forma compartida. Si otra hebra lo ha bloqueado previamente de manera exclusiva entonces bloquea a la hebra actual hasta que la propietaria del semáforo lo deja libre.
- `unlock_shared()` Desbloquea el semáforo para uso compartido.

`std::shared_lock` Envoltorio que permite adquirir un semáforo de forma compartida.

`std::unique_lock` Envoltorio que permite adquirir un semáforo de forma única y traspasar su propiedad.

Copie el programa le.cc y **verifique** que la secuencia de ejecución no es correcta porque existen condiciones de carrera.

```
atomic<bool> run(true);

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

void seccion_critica(char c)
{
  for (char i = 0; i < 10; ++i)
    cout << c++;
  cout << endl;
}

//-----------------------------------------------
```

```
void lector()
{
  while (run)
    seccion_critica('0');
}

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

void escritor()
{
  while (run)
    seccion_critica('a');
}

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

int main()
```

https://pccito.ugr.es/~gustavo/as/practicas/11/le.cc III

```
{
  const unsigned N = 8;
  thread lectores[N], escritores[N];
  std::default_random_engine engine;

  for (unsigned i = 0; i < N; ++i)
    if (engine() & 1)
    {
        lectores[i] = thread(  lector);
      escritores[i] = thread(escritor);
    }
    else
    {
      escritores[i] = thread(escritor);
        lectores[i] = thread(  lector);
    }
```

```
    this_thread::sleep_for(1s);
    run = false;

    for(thread& i:   lectores) i.join();
    for(thread& i: escritores) i.join();
  }
```

Copie el programa le-mutex.cc y **verifique** que está libre de condiciones de carrera aunque es una solución incorrecta.

```
atomic<bool> run(true);
mutex em;                    // exclusión mutua

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

void lector()
{
  while (run)
  {
    lock_guard<mutex> lock(em);
    seccion_critica('0');
  }
}
```

```
//----------------------------------------------------

void escritor()
{
  while (run)
  {
    lock_guard<mutex> lock(em);
    seccion_critica('a');
  }
```

Trabajo: le-pe.cc

- **Copie** el programa le.cc en el fichero le-pe.cc.
- **Modifique** le-pe.cc de forma que siga libre de condiciones de carrera y además permita el paralelismo entre escritores.
- En clase hemos visto dos soluciones: Bacon y Stallings.
- Compare con las otras soluciones.
- Pista: a lo mejor le puede venir bien un **interruptor**.

https://pccito.ugr.es/~gustavo/as/practicas/11/interruptor.h

```
class interruptor
{
public:
  interruptor(): contador(0) {}

  void lock(std::mutex& llave)
  {
    std::lock_guard<std::mutex> lock(mutex);
    if (++contador == 1)
      llave.lock();
  }

  void unlock(std::mutex& llave)
  {
    std::lock_guard<std::mutex> lock(mutex);
    if (--contador == 0)
      llave.unlock();
  }

private:
  std::atomic<unsigned> contador;
  std::mutex mutex;
};
```

Trabajo: le-torno.cc

- La solución anterior, le-pe.cc, ¿está libre de inanición? Compruébelo poniendo especial atención a los escritores.
- **Copie** el programa le.cc en el fichero le-torno.cc.
- **Modifique** le-torno.cc de forma que siga libre de condiciones de carrera, permita el paralelismo entre escritores y esté libre de inanición.
- Pista: a lo mejor le puede venir bien un **torno**.

ejemplo de torno

```
semáforo s = 1;
...
s.esperar();
s.señalar();
...
```

Trabajo: le-justa.cc

- **Copie** el programa le.cc en el fichero le-justa.cc.
- **Modifique** le-justa.cc de forma que funcione permitiendo una ejecución equilibrada de lectores y escritores.
- Evite las versiones que favorecen a alguna de las partes.