Lab: Concurrencia y modelo de memoria

J. Daniel García Sánchez (coordinador)

Arquitectura de Computadores
Departamento de Informática
Universidad Carlos III de Madrid

1. Objetivo

Este laboratorio tiene como objetivo ilustrar el concepto de concurrencia así como el modelo de memoria y su impacto en la presencia y eliminación de carreras de datos.

En particular aprenderás a:

- Compilar un programa con varios hilos.
- Detectar carreras de datos mediante análisis dinámico de programas.
- Corregir carreras de datos.
- Evaluar el rendimiento de soluciones concurrentes.
- Comparar soluciones basadas en cerrojos y libres de cerrojos.

2. Descripción del laboratorio

En este laboratorio partirá de un proyecto suministrado que contiene un programa principal, que ejecuta una aplicación con varios hilos.

Observa que para compilar un programa con varios hilos con CMake debes tener en cuenta:

- Debes incluir una línea para encontrar el paquete de compilación de hilos

  ```
  find_package(Threads REQUIRED)
  ```

- Debes enlazar tu programa con la biblioteca de hilos. Para ello puedes poner la siguiente línea antes de la definición del ejecutable (afectará a todos los ejecutables).

  ```
  link_libraries (Threads::Threads)
  ```

Se suministra un programa **counter.cpp** con la funcionalidad básica.

- Un clase **counter** que encapsula un valor en doble precisión, que se inicia a cero. Tiene las siguientes operaciones:
  - Una función **update()** que incrementa el valor encapsulado.
  - Una función **print()** que imprime el valor del contador.

- Un programa principal que lanza varios hilos concurrentes.
  - El número de hilos se define en la constante **num_threads**.
  - El contador compartido es la variable local **count**.
  - Cada hilo realiza 100,000 actualizaciones del contador.
  - Cuando todos los hilos han finalizado se imprime el resultado final del contador.
  - También se imprime el tiempo transcurrido en la ejecución de todos los hilos.

Para llevar a cabo este laboratorio tienes las siguientes opciones:

- Utilizar tu propio portátil con Ubuntu/Linux como sistema operativo. Se recomienda un mínimo de 2 núcleos físicos, aunque es deseable disponer de 4 o más.
- Conectarte al servicio de Aula Virtual de la UC3M (`https://aulavirtual.uc3m.es`).

3. Tareas

3.1. Compilación y ejecución

En esta tarea deberás compilar y ejecutar el programa. También compararás las versiones de *Debug* y *Release*.

Se describen las acciones a realizar desde el entorno **CLion**. No obstante, estas tareas también pueden realizarse desde línea de comandos.

Compila el proyecto y genera la versión de *Debug*. Puedes hacerlo mediante la acción de menú **Build | Build Project**.

Ejecuta el programa generado 3 veces. Anota para cada ejecución, la siguiente información:

- El valor final de **counter**.
- El tiempo de ejecución medido por el programa.

Ahora genera una versión de *Release*. Recuerda que para ello debes añadir una nueva configuración en **File | Settings... | Build, Execution, Deployment | CMake**.

Vuelve a ejecutar el programa y anota los nuevos resultados.

3.2. Detección de carreras de datos

Para detectar las carreras de datos usarás un **sanitizer** del compilador: **Thread Sanitizer**.

Vamos a generar dos nuevos perfiles de ejecución a partir de los perfiles de *Debug* y *Release*.

Selecciona la ventana de configuraciones de CMake. Recuerda que para ello debes ir a **File | Settings... | Build, Execution, Deployment | CMake**.

Genera un nuevo perfil a partir de *Debug* que llamaremos **DebugTSan**:

- Cambia el nombre a **DebugTSan**.
- En el campo **CMake options** introduce el siguiente valor:

  `-DCMAKE_BUILD_TYPE=Debug -DCMAKE_CXX_FLAGS="-fsanitize=thread"`

Genera un nuevo perfil a partir de *Release* que llamaremos **ReleaseTSan**:

- Cambia el nombre a **ReleaseTSan**.
- En el campo **CMake options** introduce el siguiente valor:
  `-DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_FLAGS="-fsanitize=thread"`

Utiliza estos nuevos perfiles para identificar posibles carreras de datos.

3.3. Protección con cerrojos

Protege el dato miembro **value_** con un **std::mutex**. Para ello:

- Añade un dato miembro a la clase **counter** de tipo **std::mutex**.
- Protege mediante el uso del **std::mutex** las secciones críticas.

Coloca estas modificaciones en el archivo fuente **counter_mutex.cpp**.
Comprueba que el programa esta libre de condiciones de carrera con **Thread Sanitizer**.
Evalúa el rendimiento del programa para los casos de 2, 4, 8 y 16 hilos.

3.4. Protección con un atómico

En C++20 se pueden utilizar tipos atómicos para proteger valores en coma flotante. Vamos a utilizar esta propiedad para tener una solución libre de cerrojos.
Cambia el tipo del dato miembro **value_** a **std::atomic<double>**.
Coloca estas modificaciones en el archivo fuente **counter_atomic.cpp**.
Comprueba que el programa esta libre de condiciones de carrera con **Thread Sanitizer**.
Evalúa el rendimiento del programa para los casos de 2, 4, 8 y 16 hilos.

3.5. Protección con un spin lock y consistencia secuencial

Ahora estudiaremos los casos en que una variable atómica no es suficiente. Consideremos el caso en que en vez de un valor se utilizan dos valores en la sección crítica.

- Modifica la clase **counter** para que tenga un segundo dato miembro llamado **time_** que será de tipo **float**.
- Modifica la función **update()** para que en cada invocación se incremente **time_** en **0.25** unidades.

Para controlar el acceso a la sección crítica, utiliza un objeto **spinlock_mutex** como el siguiente, que utiliza consistencia secuencial:

```
class spinlock_mutex {
public:
  spinlock_mutex() : flag_{} {}

  void lock() {
    while (flag_.test_and_set(std::memory_order_seq_cst)) {}
  }

  void unlock() {
    flag_.clear(std::memory_order_seq_cst);
  }

private:
  std::atomic_flag flag_;
};
```

Coloca estas modificaciones en el archivo fuente **counter_spin_seq.cpp**.
Comprueba que el programa esta libre de condiciones de carrera con **Thread Sanitizer**.
Evalúa el rendimiento del programa para los casos de 2, 4, 8 y 16 hilos.

3.6. Protección con un spin lock y consistencia de liberación/adquisición

Repite el apartado anterior con un **spinlock_mutex** que utilice consistencia de liberación/adquisición:

```
class spinlock_mutex {
public:
  spinlock_mutex() : flag_{} {}

  void lock() {
    while (flag_.test_and_set(std::memory_order_acquire)) {}
  }

  void unlock() {
    flag_.clear(std::memory_order_release);
  }

private:
  std::atomic_flag flag_;
};
```

Coloca estas modificaciones en el archivo fuente **counter_spin_ra.cpp**.
Comprueba que el programa esta libre de condiciones de carrera con **Thread Sanitizer**.
Evalúa el rendimiento del programa para los casos de 2, 4, 8 y 16 hilos.

3.7. Optimización del spinlock

Para los dos casos anteriores, considera la optimización que se consigue cuando se introduce una lectura relajada dentro del bucle de la operación **lock()**

```
void lock() {
  while (flag_.test_and_set(std::memory_order_XXXX)) {
    while (flag_.test(std::memory_order_relaxed)) {}
  }
}
```

Coloca estas modificaciones en los archivos fuente **counter_spin_seq_opt.cpp** y **counter_spin_ra_opt.c**
Comprueba que el programa esta libre de condiciones de carrera con **Thread Sanitizer**.
Evalúa el rendimiento del programa para los casos de 2, 4, 8 y 16 hilos.

4. Entrega

La fecha límite para la entrega de los resultados de este laboratorio se anunciará a través de Aula Global.

Se seguirán las siguientes reglas:

- Todas las entregas se realizarán a través de aula global.
- Se deberá entregar un archivo en formato zip que contendrá los siguientes archivos:
  - **counter_mutex.cpp**.
  - **counter_atomic.cpp**.
  - **counter_spin_seq.cpp**.
  - **counter_spin_seq_opt.cpp**.
  - **counter_spin_ra.cpp**.
  - **counter_spin_ra_opt.cpp**.
- El contenido del informe deberá incluir.
  1. Portada identificativa que incluirá:
     - Título.
     - Nombre y NIA de los estudiantes.
     - Grupo reducido al que pertenecen los estudiantes.
     - Número de grupo de laboratorio.
  2. Datos identificativos de la máquina de ejecución, por ejemplo la salida del mandato **lscpu**.
  3. Datos identificativos del compilador (**g++ –version**).
  4. Tabla de resultados de las evaluaciones.
  5. Breves conclusiones.

En cuanto a los resultados de las evaluaciones deberá entregarse exclusivamente información numérica en formato tabla. Indique claramente al principio de la sección cuantas veces ha ejecutado cada experimento. Cada tabla deberá tener una fila por cada uno de los programas y una columna por cada versión en cuanto a número de hilos (2, 4, 8, 16).

IMPORTANTE: Deberá ejecutar un número suficiente de veces cada experimento como para asegurar una desviación típica aceptable.

Se deberán incluir en el informe las siguientes tablas:

1. Tabla de tiempos medios de ejecución.
2. Tabla de desviaciones típicas.