Laboratorio de programación concurrente y consistencia de memoria

J. Daniel García Sánchez (coordinador)

Arquitectura de Computadores
Grupo ARCOS
Departamento de Informática
Universidad Carlos III de Madrid

1. Objetivo

Este laboratorio tiene como objetivo fundamental familiarizar al estudiante con la programación libre de cerrojos y compender el impacto que puede tener en el rendimiento de una aplicación.

En concreto, se evaluarán las ventajas e inconvenientes de uso de tipos de datos atómicos frente a las técnicas básicas basadas en cerrojos.

2. Descripción

En esta práctica se evaluarán las distintas alternativas para implementar un buffer acotado circular (*circular bounded buffer*). Para evaluar la seguridad frente a hilos de ambas estructuras de datos se usarán dos técnicas: **programación basada en cerrojos** y **programación libre de cerrojos**. En ambos casos se hará uso del estándar **ISO/IEC 14882:2017** (C++17).

2.1. Material suministrado

Para la realización de la práctica se suministra la implementación de un búfer acotado circular con varias estrategias de implementación:

- **seq_buffer.h**: Búfer circular secuencial.
- **locked_buffer.h**: Búfer circular concurrente con cerrojos.
- **atomic_buffer.h**: Búfer circular concurrente libre de cerrojos.

Así mismo se suministran **generadores** (archivo de cabecera **generators.h**) y **reductores** (archivo de cabecera **reducers.h**).

2.2. Generadores

La aplicación utiliza varios **generadores** de datos para generar los datos que se colocan en el búfer:

- **random_numeric_generator<T>**: Es un un generador de una secuencia de número aleatorios de cualquier tipo numérico **T**.

- **file_generator<T>** Es un generador de una secuencia de valores leídos de un archivo de texto de un tipo genérico **T**. Entre otros, **T** puede ser un tipo numérico (p.ej. **long**) o el tipo **string**.

Todos los generadores se comportan como objetos función (*functors*) y tienen un constructor para establecer su estado.

```
random_numeric_generator<long> rand_gen{10}; // Genera hasta 10 valores long
file_generator<string> word_gen{"file.txt"}; // Genera palabras leídas del archivo "file.txt"
file_generator<long> num_gen{"num.txt"}; // Genera números leídos del archvio "num.txt"
```

Por otra parte, también tienen redefinido el operador de invocación a función. De esta manera, se les puede invocar como una función:

```
auto numval = rgen();
auto word = fgen();
```

Los generadores devuelven un valor opcional de la clase **std::optional**. Un valor opcional puede contener un valor o estar vacío.

```
optional<long> numval = rgen();
long val = -1;
if (numval) { val = *numval; }

optional<string> word = fgen();
if (word) {
  cout << *word << "\n";
}
```

2.3. Reductores

La aplicación utiliza varios **reductores** de datos para generar información reducida de los datos que se obtienen del búfer:

- **numeric_reducer<T>**: Es un reductor genérico para cualquier tipo numérico T. Acepta valores numéricos y permite obtener el mínimo y el máximo de los valores procesados.
- **word_counter**: Es un reductor que cuenta la frecuencia de las palabras. Acepta valores de tipo **string** y permite obtener el número de ocurrencia de la palabra más frequente.

Todos los reductores tienen un constructor sin parámetros que establece su estado inicial:

```
numeric_reducer<long> num_red; // Reductor numérico para long
word_counter freq_red; // Reductor de frecuencia de palabras
```

Además, todos los reductores tienen redefinido el operador += para agregar valores al reductor.

```
num_red += 10L; // Agrega el valor 10L a num_red
num_red += 30L // Agrega el valor 30L a num_red;
num_red += 20L // Agrega el valor 20L a num_red;

freq_red += "Hola"s;
freq_red += "C++"s:
freq_red += "Hola"s;
```

El reductor numérico permite obtener los valores **máximo** y **mínimo**:

```
long a = num_red.max();
long b = num_red.min();
```

El reductor de frecuencias de palabras, permite obtener la palabra más frecuente y el número de apariciones.

```
auto r = freq_red.most_frequent();
cout << "palabra: " << r.first << "\n";
cout << "frecuencia: " << r.second << "\n";
```

2.4. Alternativas de búfer

Se suministran tres alternativas de búfer acotado:

- **Búfer secuencial**: Es un búfer acotado diseñado para aplicaciones secuenciales. No soporta que se pueda obtener datos de un búfer vacío ni colocar un dato en un búfer lleno. Se implementa mediante la clase **seq_buffer<T>** (archivo **seqbuffer.h**).
- **Búfer con cerrojos**: Es un búfer acotado diseñado para aplicaciones concurrentes. Si se intenta obtener de un búfer vacío o colocar un dato en un búfer lleno, el llamante se bloquea hasta que se pueda completar la operación. Se implementa mediante la clase **locked_buffer<T>** (archivo **lockedbuffer.h**).
- **Búfer libre de cerrojos**: Es un búfer acotado diseñado para aplicaciones concurrentes. No utiliza **mutex** ni **variables condición**. Se implementa mediante la clase **atomic_buffer<T>** (archivo **atomicbuffer.h**).

IMPORTANTE: Se recomienda que el estudiante revise con detenimiento la implementación de estas clases, recurriendo si fuese necesario a la documentación de la biblioteca estándar (por ejemplo en `http://en.cppreference.com/w/`) o a textos recomendados en la asignatura (*C++ Concurrency in Action. Practical multithreading*).

2.5. Productor genérico

El código incluye un productor genérico de valores (clase **producer** en archivo de cabecera **prodcons.h**) que se puede configurar con un tipo generador de valores y un tipo de búfer.

```
using gen_type = numeric_generator<long>;
using buf_type = locked_buffer<long>;
gen_type gen{1000}; // Generador de números
buf_type buf{10}; // Búfer con cerrojos de tamaño 10

producer<gen_type,buf_type> prod{gen,buf}; // Productor usando gen
```

Un productor se puede invocar de dos maneras. La forma más simple es sin pasar ningún argumento. En este caso, el productor genera valores que va colocando en el búfer hasta que el generador produce un valor vacío que se considera indicación de fin valores generados.

```
producer<gen_type,buf_type> p{gen,buf};
p(); // Genera valores hasta fin de secuencia

producer<gen_type,buf_type> q{gen,buf};
thread t{q}; // Arranca hilo para generar valores en buf
t.join();
```

Un productor también se puede invocar pasando como argumento un predicado que indica cuando se debe parar de producir valores:

```
file_generator<string> gen{"texto.txt"};
seq_buffer<string> buf{32};

producer<file_generator<string>,seq_buffer<string>> prod{gen,buf};
bool finisehd = false;
while (!finished) {
  prod([&] { return !buf.full(); });
  bool finished = consume(buf);
}
```

2.6. Consumidor genérico

El código incluye también un consumidor genérico de valores (clase **consumer** en archivo de cabecera **prodcons.h**) que se puede configurar con un tipo reductor de valores y un tipo de búfer.

```
word_counter wc;
atomic_buffer<string> buf{16};

consumer<word_counter,atomic_buffer<string>> cons{wc,buf};
```

Un consumidor se puede invocar de dos maneras. La forma más simple es sin pasarle ningún argumento. En este caso, el consumidor obtiene valores del búfer hasta que encuentra un valor vacío que se considera indicación de fin de valores generados.

```
consumer<word_counter,atomic_buffer<string>> cons{wc,buf};
cons(); // Consume valores hasta fin de secuencia

consumer <word_counter, atomic_buffer<string>> q{wc,buf};
thread t{q}; // Arranca hilo para consumir valores de buf
```

Un consumidor también se puede invocar pasando como argumento un predicado que indica cuando se debe parar de consumir valores:

```
using gen_type = file_generator<string>;
gen_type gen{"text.txt"};
using red_type = word_counter;
red_type wc;

using buf_type = locked_buffer<string>;
buf_type buf{40};

producer<gen_type,buf_type> prod{gen,buf};
consumer<cons_type, buf_type> cons{red,buf};
for (;;) {
  prod([&]{ return !buf.full();} );
  bool finished = cons([&]{return !buf.empty();} )
  if (finished) break;
}
```

Observe que una invocación a un consumidor devuelve un booleano que indica si se ha recibido una indicación de fin de secuencia (devolviendo **true**) o se ha terminado porque el predicado no se cumplió y el búfer estaba vacío.

2.7. Alternativas de ejecución

El código de apoyo suministra dos objetos función para abstraer la ejecución de aplicaciones con un productor y un consumidor.

- **sequential_runner**: Es un objeto función que ejecuta una aplicación genérica con un productor, un consumidor y un búfer. Se encuentra en el archivo de cabecera **seqrunner.h**.
- **concurrent_runner**: Es un objeto función que ejecuta una aplicación genérica con un productor, un consumidor y un búfer. El productor y el consumidor se ejecutan en hilos separados. Se encuentra en el archivo de cabecera **concrunner.h**.

Por ejemplo, si se quiere usar una ejecución concurrente para buscar la palabra más frecuente en un archivo de texto se puede hacer:

```
void most_frequent(const std::string & filename) {
  locked_buffer<string> buf{20};
  file_generator<string> gen{filename};
  word_counter wc;
```

```
    concurrent_runner runner;
    runner(gen,wc,buf);

    auto res = reducer.most_frequent();
    cout << "Palabra: " << res.first << "\n";
    cout << "Apariciones: " << res.second << "\n";
}
```

2.8. Programas de evaluación

El código suministrado contiene tres programas:

- **seq_test**: Versión secuencial que alterna un productor y un consumidor. Utiliza una ejecución secuencial y un búfer secuencial.
- **locked_test**: Versión muti-hilo con un productor y un consumidor. Utiliza una ejecución concurrente y un búfer con cerrojos.
- **atomic_test**: Versión muti-hilo con un productor y un consumidor. Utiliza una ejecución concurrente y un búfer libre de cerrojos.

Estos tres ejecutables, se pueden invocar de con distitos argumentos para que invoquen a distitas funciones de prueba (definidas todas en **tests.h**).

NOTA: En el resto de esta sección **prog** se refiere indistintamente a cualquiera de los tres programas.

2.8.1. Máximo y mínimo de números aleatorios

Este modo usa un productor que genera un cierto número de valores enteros y utiliza un reductor que calcula los valores máximo y mínimo de la secuencia. También debe especificarse el tamaño del búfer intermedio.

Por ejemplo, el siguiente mandato:

```
prog random 10 1000
```

Invoca el caso de número aleatorios con un tamaño de búfer de 10 elementos y una generación de 1000 números enteros.

2.8.2. Máximo y mínimo de números de un archivo

Este modo usa un productor que genera números enteros leídos de un archivo de texto y utiliza un reductor que calcula los valores máximo y mínimo de la secuencia. También debe especificarse el tamaño del búfer intermedio.

Por ejemplo, el siguiente mandato:

```
prog file 10 datos.txt
```

Invoca el caso de números leídos de archivo con un tamaño de búfer de 10 elementos y leyendo los números del archivo **datos.txt**.

2.8.3. Frecuencia de palabras

Este modo usa un productor que genera palabras leídas de un archivo de texto y utiliza un reductor que determina la palabra más frecuente y su número de apariciones. También debe especificarse el tamaño del búfer intermedio.

Por ejemplo, el siguiente mandato:

```
prog count 10 ../data/quijote.txt
```

Invoca el caso de palabras leídas de archivo con un tamaño de búfer de 10 elementos y leyendo las palabras del archivo **../data/quijote.txt**.

2.9. Evaluación del rendimiento

Para realizar evaluación del rendimiento, se pueden usar algunos de los siguientes métodos:

- Medición de tiempos usando la biblioteca estándar de C++ (espacio de nombres **chrono**).
- Acceso al módulo del núcleo de Linux `perf`.

IMPORTANTE: No olvide que para realizar la evaluación debe hacerlo con las optimizaciones del compilador activadas (modo **Release** de **CMake**).

3. Tareas

3.1. Estudio del código fuente

Estudie el código fuente suministrado y analice su funcionamiento.

3.1.1. Búfer secuencial

Estudie la implementación del búfer secuencial (archivo de cabecera **seqbuffer.h**) y considere las siguientes cuestiones:

1. ¿Qué funciones de **seq_buffer** pueden lanzar excepciones?
2. ¿Puede el constructor de **seq_buffer** lanzar alguna excepción? ¿Cuál?
3. ¿Para qué sirve el dato miembro **next_read_** de **seq_buffer**?
4. ¿Para qué sirve el dato miembro **next_write_** de **seq_buffer**?
5. ¿Cuál es el número máximo de elementos que puede almacenarse en un **seq_buffer** creado con **size_ == 100**?
6. ¿Qué ocurre si se hace un **put()** sobre un **seq_buffer** que está lleno?
7. ¿Qué ocurre si se hace un **put()** sobre un **seq_buffer** que está vacío?
8. ¿Qué ocurre si se hace un **get()** sobre un **seq_buffer** que está lleno?
9. ¿Qué ocurre si se hace un **get()** sobre un **seq_buffer** que está vacío?

3.1.2. Búfer con cerrojos

Estudie la implementación del búfer con cerrojos (archivo de cabecera **lockedbuffer.h**) y considere las siguientes cuestiones:

1. ¿Qué funciones de **locked_buffer** pueden lanzar excepciones?
2. ¿Puede el constructor de **locked_buffer** lanzar alguna excepción? ¿Cuál?
3. ¿Puede la función miembro **put()** lanzar una excepción? ¿Cuál?
4. ¿Puede la función miembro **get()** lanzar una excepción? ¿Cuál?
5. ¿Qué diferencia hay entre **full()** e **is_full()**?
6. ¿Qué diferencia hay entre **empty()** e **is_empty()**?
7. ¿Cuál es el número máximo de elementos que puede almacenarse en un **locked_buffer** creado con **size_ == 100**?
8. ¿Qué ocurre si se hace un **put()** sobre un **locked_buffer** que está lleno?
9. ¿Qué ocurre si se hace un **put()** sobre un **locked_buffer** que está vacío?
10. ¿Qué ocurre si se hace un **get()** sobre un **locked_buffer** que está lleno?
11. ¿Qué ocurre si se hace un **get()** sobre un **locked_buffer** que está vacío?
12. Investigue el efecto de la palabra reservada **mutable**. Si se eliminase la calificación de **mutable** sobre el dato miembro **mut_** ¿Qué funciones miembro habría que modificar? ¿Cómo?
13. ¿Por qué no es necesario marcar como **mutable** a los datos miembro **not_full_** y **not_empty_**?

3.1.3. Búfer libre de cerrojos

Estudie la implementación del búfer libre de cerrojos (archivo de cabecera **atomicbuffer.h**) y considere las siguientes cuestiones:

1. ¿Qué funciones de **atomic_buffer** pueden lanzar excepciones?
2. ¿Puede el constructor de **atomic_buffer** lanzar alguna excepción? ¿Cuál?
3. ¿Puede la función miembro **put()** lanzar una excepción? ¿Cuál?
4. ¿Puede la función miembro **get()** lanzar una excepción? ¿Cuál?
5. ¿Cuál es el número máximo de elementos que puede almacenarse en un **atomic_buffer** creado con **size_ == 100**?
6. ¿Qué ocurre si se hace un **put()** sobre un **atomic_buffer** que está lleno?
7. ¿Qué ocurre si se hace un **put()** sobre un **atomic_buffer** que está vacío?
8. ¿Qué ocurre si se hace un **get()** sobre un **atomic_buffer** que está lleno?
9. ¿Qué ocurre si se hace un **get()** sobre un **atomic_buffer** que está vacío?

10. Investigue para qué puede utilizarse el atributo del lenguaje **alignas**. ¿Qué efecto podría tener el eliminar al calificación con el mismo de los datos miembro **next_read_** y **next_write_**.
11. ¿Por qué se usa un valor de alineamiento de **64** al usar **alignas**?
12. ¿Hay alguna operación potencialmente bloqueante en **atomic_buffer**?

3.2. Evaluación del rendimiento

Evalúe los 3 programas con los siguientes casos: **random** y **count**.

3.2.1. Evaluación del test random

Evalúe el programa generando 1000 valores y 1000000 valores. En ambos casos estudie el tiempo total de ejecución para un tamaño de búfer de 2, 10, 100 y 1000.

3.2.2. Evaluación del test count

Evalúe el programa contando palabras de los ficheros **quijote.txt** y **king-lear.txt** (disponibles en el directorio **data**).

En ambos casos estudie el tiempo total de ejecución para un tamaño de búfer de 2, 10, 100, y 1000.

3.2.3. Compilación

Para compilar los programas puede hacer uso de **CMake**.

En el directorio principal del material, cree un subdirectorio de nombre **build**. Genere los archivos de configuración de **CMake** indicando que desea compilar en modo **Release**. Por último invoque a **make** para realizar la compilación.

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

4. Entrega

Se seguirán las siguientes reglas:

- Todas las entregas se realizarán a través de aula global.
- El único formato admisible para la entrega será rellenando el cuestionario a través de Aula Global.
- La entrega y realización de los cuestionarios se hará de forma individual al igual que la realización del ejercicio.
- Una vez iniciado un cuestionario el estudiante dispondrá de un tiempo máximo de 30 minutos para completarlo.
- Cada estudiante dispondrá de un único intento para completar el cuestionario.
- El número máximo de cuestiones que tendrá que contestar cada estudiante será de 10.