Concurrent programming and memory consistency laboratory

J. Daniel García Sánchez (coordinator)

Computer Architecture
ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

1. Objective

This laboratory has as main objective the familiarization of the student with lock free programming and understand the impact that it can have over the performance of an application.

Concretely, the advantages and disadvanatges of the usage of certain types of atomic data will be evaluated versus the usage of techniques based on locks.

2. Descripción

In this laboratory different alternatives will be evaluated in order to implement a (*circular bounded buffer*). To evaluate the security of these structures considering the usage of threads two techniques will be employed: **lock based programming** and **free lock programming**. In both cases the standard **ISO/IEC 14882:2017** (C++17) will be used.

2.1. Materials supplied

For the completion of this lab you are given the implementation of a circular bounded buffer with different implementation strategies:

- **seq_buffer.h**: Sequential circular buffer.
- **locked_buffer.h**: Concurrent circular buffer with locks.
- **atomic_buffer.h**: Concurrent circular buffer free of locks.

also, you are given **generators** (header file **generators.h**) and **reducers** (header file **reducers.h**).

2.2. Generators

The applications makes use of different **generators** of data to generate the data that is placed in the buffer:

- **random_numeric_generator<T>**: It is a generator of a randon number sequence of any numerical type **T**.

- **file_generator<T>** It a generator of a sequence of values read from a text file of generic type **T**. Among others, it can be a numerical type (for example **long**) or of type **string**.

All generators behave as function objects (*functors*) and have a constructor to stablish its state.

```
random_numeric_generator<long> rand_gen{10}; // It generates up to 10 long values
file_generator<string> word_gen{"file.txt"}; // It generates words read from the file "file.txt"
file_generator<long> num_gen{"num.txt"}; // It generates numbers read from the file "num.txt"
```

Moreover, they also have their function invocation operator redefined. This way, they can invoked as a function:

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

The generators return an optional value of the class **std::optional** An optional value can contain a value or be empty.

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

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

2.3. Reducers

The application uses several **reducers** of data to generate reduced information of the data obtained from the buffer:

- **numeric_reducer<T>**: It is a generic reducer for any type of numerical type T. It accepts numerical values and allows to obtain the minimum and maximum of teh processed values.
- **word_counter**: It is a reducer that counts the frequency of the words. It accept values of type **string** and allows to obatin the number of occurrences of the most frequent word.

All reducers have a constructor without any parameters that stablish their initial state:

```
numeric_reducer<long> num_red; // Numerical reducer for long
word_counter freq_red; // Word frequency reducer
```

Moreover, all reducers have the operator += redefined to aggregate values to the reducer.

```
num_red += 10L; // Aggregates the value 10L to num_red
num_red += 30L // Aggregates the value 30L to num_red;
num_red += 20L // Aggregates the value 20L to num_red;

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

The numerical reducer allows to obtain the value **maximum** y **minimum**:

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

The word frequency reducer allows to obtain th emost frequent word and the number of ocurrences.

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

2.4. Buffer alternatives

There are three alternatives for the bounded buffer:

- **Sequential buffer**: It is a bounded buffer designed for sequential applications. It does not support to obatin data from an empty buffer or putting data into a a full buffer. It is implemented using the class **seq_buffer<T>** (file **seqbuffer.h**).

- **Locked buffer**: It is a bounded buffer designed to support concurrent applications. When trying to obatin data from an empty buffer or trying to put data into a full buffer, the called is blocked until th eoperation can be completed. It is implemented using the class **locked_buffer<T>** (file **lockedbuffer.h**).

- **Lock free buffer**: It is a bounded buffer designed to support concurrent applications. It does not use **mutex** nor **conditional variables**. It is implemented using the class **atomic_buffer<T>** (file **atomicbuffer.h**).

IMPORTANT: It is recommended that the student review the implementation of these classes, using if necessary the documentation of the standard library (for example in`http://en.cppreference.com/w/`) or recommended textbooks of the subject (*C++ Concurrency in Action. Practical multithreading*).

2.5. Generic Producer

The code includes a generic value producer (class **producer** in header file **prodcons.h**) that can be configured with a value generator type and a buffer type.

```
using gen_type = numeric_generator<long>;
using buf_type = locked_buffer<long>;
gen_type gen{1000}; // Number generator
buf_type buf{10}; // Locked buffer of size 10

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

A producer can be invoked in two ways. The simplest way is to not pass any argument. In this case, the producer generates values that are places in the buffer until the generator produces an empty value that is considered as an indication of the end of the generated values.

```
producer<gen_type,buf_type> p{gen,buf};
p(); // Generates values until the end of the sequence

producer<gen_type,buf_type> q{gen,buf};
thread t{q}; // It created a thread that generates values in buf
t.join();
```

A producer can alos be invoked passing as argument a predicate that indicates when it should stop producing values:

```
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. Generic consumer

The code also includes a generic value consumer (class **consumer** in header file **prodcons.h**) that can be configures with a reducer type and a buffer type.

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

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

A consumer can be invoked in two ways. The simplest way is to not pass any arguments. In this case, the consumer obtains values from the buffer until it finds an empty value that indicates the end of the generated values.

```
consumer<word_counter,atomic_buffer<string>> cons{wc,buf};
cons(); // It consumes values until the end of the sequence
consumer <word_counter, atomic_buffer<string>> q{wc,buf};
thread t{q}; // It creates a thread to consume values of buf
```

Un consumer can also be invoked passing as argument a predicate that indicates when it should stop consuming values:

```
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 that an invocation of a consumer returns a boolean that indicates if the indication of the end of the sequence (returning **true**) has been received or it has stopped because the predicate was not true and the buffer was empty.

2.7. Execution alternatives

The code also supplies two function objects to abstract the execution of applications with a producer and a consumer.

- **sequential_runner**: It is a function object that executes a generic application with a producer, a consumer, and a buffer. It can be found in the header file **seqrunner.h**.
- **concurrent_runner**: It is a function objetc that executes a generic application with a producer, a consumer and a buffer. The producer and consumer are executed in different threads. It can be found in the header file **concrunner.h**.

For example, if you want to execute concurrently to search the most frequent word in a text file:

```
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. Evaluation programs

The code contains three programs:

- **seq_test**: Sequential version that alternates a producer and a consumer. It uses a sequential runner and a sequential buffer.
- **locked_test**: Multi-threaded version with a producer and a consumer. It useas a concurrent runner and a locker buffer.
- **atomic_test**: Multi-threaded version with a producer and a consumer. It uses a concurrent runner and a lock free buffer.

These three executables can be invoked with different arguments in order to invoke different test functions (all defined in **tests.h**).

NOTE: The rest of this section **prog** refers indistinctly to any of the three programs.

2.8.1. Maximum and minimum of random numbers

This mode uses a producer that generates a certain number of integer numbers ans a reducer that computes the maximum and minimum of the sequence. The size of the buffer must be specified.

For example, the following command:

```
prog random 10 1000
```

Executes the mode of random numbers with a buffer size of 10 elements and a generation of 1000 random numbers.

2.8.2. Maximum and minimum of numbers in a file

This mode uses a producer that generates integer numbers from a text file and uses a reducer that computes the maximum an minimum values of the sequence. The size of the buffer must be specifies.

For example, the following command:

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

Executes the mode of numbers read from a file with a buffer size of 10 and reading the numbers from the file **datos.txt**.

2.8.3. Words frequency

This mode uses a producer that generates words read from a text file and uses a reducer that determines which is teh most frequent word and its numebr of occurrences. The buffer size must also be specified.

For example, the following command:

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

Executes the mode of wors read from a text file with a buffer size of 10 elements and reading the words from the file **../data/quijote.txt**.

2.9. Performance Evaluation

To carry out the performance evaluation, one of the following methods could be used:

- Measure of the time using the standard libray of C++ (namespace **chrono**).
- Accessing the Linux kernel module `perf`.

IMPORTANT: Do not forget to activate the optimizations of the compiler before executing the evaluation (mode **Release** of **CMake**).

3. Tasks

3.1. Source code study

Study the source code supplied and analyze how it works.

3.1.1. Sequential buffer

Study the implementation of the sequential buffer (header file **seqbuffer.h**) and consider the following questions:

1. What functions of **seq_buffer** can throw exceptions?
2. Can the constructor of **seq_buffer** throw any exception? Which one or ones?
3. What is the use of the data member **next_read_** in **seq_buffer**?
4. What is the use of the data member **next_write_** in **seq_buffer**?
5. How many elements can be stored in a **seq_buffer** created with **size_ == 100**?
6. What happens when executing a **put()** over a **seq_buffer** that is full?
7. What happens when executing a **put()** over a **seq_buffer** that is empty?
8. What happens when executing a **get()** over a **seq_buffer** that is full?
9. What happens when executing a **get()** over a **seq_buffer** that is empty?

3.1.2. Locked buffer

Study the implementation of the locked buffer (header file **lockedbuffer.h**) and consider the following questions:

1. What functions of **locked_buffer** can throw exceptions?
2. Can the constructor of **locked_buffer** throw any exception? Which one or ones?
3. Can the member function **put()** throw an exception? Which one or ones?
4. Can the member function **get()** throw an exception? Which one or ones?
5. What is the difference between **full()** and **is_full()**?
6. What is the difference between **empty()** and **is_empty()**?

7. How many elements can be stored, as maximum, in a **locked_buffer** created with **size_ == 100**?
8. What happens when executing a **put()** over a **locked_buffer** that is full?
9. What happens when executing a **put()** over a **locked_buffer** that is empty?
10. What happens when executing a **get()** over a **locked_buffer** that is full?
11. What happens when executing a **get()** over a **locked_buffer** that is empty?
12. Study the effect of the reserved word **mutable**. If the mutable keyword is removed from the data member **mut_**, what member functions should be modified?. How?
13. Why is it not necessary to mark as **mutable** th edata members **not_full_** and **not_empty_**?

3.1.3. Lock free buffer

Study the implementation of the lock free buffer (header file **atomicbuffer.h**) and consider the following questions:

1. What functions of **atomic_buffer** can throw exceptions?
2. Can the constructor of **atomic_buffer** throw any exception? Which one or ones?
3. Can the member function **put()** throw an exception? Which one or ones?
4. Can the member function **get()** throw an exception? Which one or ones?
5. How many elements can be stored, as maximum, in a **atomic_buffer** created with **size_ == 100**?
6. What happens when executing a **put()** over a **atomic_buffer** that is full?
7. What happens when executing a **put()** over a **atomic_buffer** that is empty?
8. What happens when executing a **get()** over a **atomic_buffer** that is full?
9. What happens when executing a **get()** over a **atomic_buffer** that is empty?
10. Study the effect of the language atribute **alignas**. What effect could it have to remove this atribute from the data members **next_read_** and **next_write_**?
11. Why is an alignment value of **64** used in **alignas**?
12. Is there any operation potentially blocking in **atomic_buffer**?

3.2. Performance Evaluation

Evaluate the 3 programs with teh following modes: **random** and **count**.

3.2.1. Evaluation with random

Evaluate the program generating 1000 values and 1000000 values. In both cases study the total execution time for a buffer size of 2, 10, 100 and 1000.

3.2.2. Evaluation with count

Evaluate the program counting words from the files **quijote.txt** and **king-lear.txt** (available in folder **data**).

In both cases study the total execution time for a buffer size of 2, 10, 100, and 1000.

3.2.3. Compilación

To compile the programs you can make use of **CMake**.

At the main folder of the material provided, create a subdirectory with name **build**. Generate the configuration files of **CMake** indicating that you wish to compile in **Release** mode. Then, invoke **make** to compile.

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

4. Submission

The following rules will be applied:

- All the submissions will be done through Aula Global.
- The only format admissible for the submission will be the fullfilment of a quiz through Aula Global.
- The submission and realization of the quizzes will be done individually as well as the development of exercises.
- Once the quiz is initiated the student will have a maximum time to complete it of 20 minutes.
- Each student will have only one try for completing the quiz.
- The maximum number of questions in each quiz will be 10.