Milton Palaguachi

Fall 2020
Concurrency and Multithreading using C++ Libraries

The City College of New York
Professor: Devendra Kumar

Introduction

In real world applications sometimes downloading data from the internet, or computing a long task requires time and one single process or thread might not do the job fast enough as a result our application becomes very slow. As we all have experience sometimes when an application is slow no one wants to wait for the application to respond after a while. Developers need to tackle this problem so that the application becomes more responsive to users. In this report I will be discussing what concurrency is, how concurrency plays an important role in computer science and how we can use the C++ library to make our application faster and more user interactive.

What is concurrency?

Concurrency occurs when multiple copies of a program run simultaneously while communicating with each other. In other words, concurrency is when two tasks overlap. A simple concurrent application will use a single machine to store the program's instruction, but that process is executed by multiple, different threads. This setup creates a kind of control flow, where each thread executes its instruction before passing to the next one. The threads act independently and to make decisions based on the previous thread as well. However, some issues can arise in concurrency that make it tricky to implement.

What is concurrent in Computer Systems

A single system performs multiple independent activities in parallel, rather than sequentially, or one after the other.

Serialy vs Concurrent Computation

I just want to clarify the difference between serial and current so we understand since I will be using this term very often in this report.
Serialy Computation:

- A problem is broken into secrete series of instruction
- Instruction are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time.

Concurrent computing :

- Two or more separate tasks are executing at the same time
- Multiple calculations are made within an overlapping time frame.
- It takes advantage of the concept of multiple threads or processes can make progress on a task without waiting for others to complete.

Synchronous vs Asynchronous

1. **Synchronously**. The thread will stop the execution and wait for the task to complete. Meaning the function WILL BLOCK the current thread until it has completed

2. **Asynchronously.** The thread will dispatch the task and will continue executing the code
it won't stop or wait. Meaning it will be handled in the background and the function WILL NOT BLOCK the current thread.

What is Multithreading

Multithreading is a specialized form of multitasking and multitasking is the feature that allows our computer to run two or more programs concurrently. In general, there are two types of multitasking: process-based and thread-based.
Process-based multitasking handles the concurrent execution of programs.
Thread-based multitasking deals with the concurrent execution of pieces of the same program.
A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Pros vs Cons of using Multithreading
Pros

- Fast to start
- Low overhead

Cons

- Difficult to implement
- Can’t run on a distributed system.

Race Condition

One of the typical problems in the multithreading environment is data race condition. The problem is when sharing data between threads is due to the consequences of modifying data. If all shared data is read-only, there’s no problem, however if one or more threads start modifying the data, there’s a lot of potential for trouble.

How to avoid problematic race conditions

The simplest option is to wrap our data structure with a protection mechanism, to ensure that only the thread actually performing a modification can see the intermediate states where the invariants are broken. From the point of view of other threads accessing that data structure Using the C++ 11, one can use std::mutex, lock it with a call to the member function lock(), unlock it with a call to the member function unlock(). However this isn’t recommended in practice as mentioned in the book Concurrency in Action, because we will have to remember to call unlock() on every code path out of a function. The Standard C++ Library provides the std::lock_guard class template, which implements that RAII idiom for a mutex; it locks the supplied mutex on construction and unlocks it on destruction, thus ensuring a locked mutex is always correctly unlocked. The following example shows how to protect a list that can be accessed by multiple threads using a std::mutex, along with std::lock_guard. Both of these are declared in the <mutex> header.

```
#include <list>
#include <mutex>
#include <algorithm>
#include <iostream>
std::list<int> some_list;
std::mutex some_mutex;

void add_to_list(int new_value)
```

```
{
  std::lock_guard<std::mutex> guard(some_mutex);
  //Critical section begin
  some_list.push_back(new_value);
 //Critical section end
}
int main()
{
        add_to_list(20);
}
```

Another way to avoid data race conditions is to use a third party library Boost, it offers many libraries for handling data being accessed and modified by multiple threads. In the example below, we are using barrier to manage threads to complete and wait for the other threads. One important thing to notice is that the scope_lock mutex locks the pencil on the scope surrounded by the curly braces. Once the thread goes outside the scope the mutex pencil gets automatically unlocked. This is one of the namy futures that C++ 17 provides.

```
//Deciding how many bags of chips to buy for the party
/*
  order of operation
  1. jessy_shopper()   add: bag_of_chips +=3, and wait()
  2. milton_shopper wait() for all NUMBER_OF_THREAD defined before continuing, double: bag_of_chips *=2

  First. it will perform addition to bag_of_chips all five threads that call jessy_shopper() function and wait.
  Second .it will wait for the all 10 thread before doubling in milton_shopper()

*/
#include<thread>
#include<mutex>
#include <boost/thread/barrier.hpp>
#define NUMBER_OF_THREADS 10 // 10 threads
/*
* critical condition has bags_of_chip
*/
unsigned int bags_of_chip = 1; //start with one on the list
unsigned int count = 1000000;
```

```
std::mutex pencil;
/*  N = 10, barrier constructor takes  an argument for the number of thread to wait
   before it releases it to continue the program.
*/
boost::barrier first_dump(NUMBER_OF_THREADS);
// cpu does some work
void cpu_work(unsigned long workUnits){
   unsigned long x = 0;
   for (unsigned long i = 0; i < workUnits*count; i++)
   { x++; }
}

void milton_shopper() {
   cpu_work(1); //do a bit of work first
   first_dump.wait();
   std::scoped_lock<std::mutex>lock(pencil);
   bags_of_chip *= 2;
   puts("Milton Double the bags of chips.");
}
void jessy_shopper(){
   cpu_work(1); // do a bit of work first
   {
      std::scoped_lock<std::mutex> lock(pencil);
      bags_of_chip += 3;
   }
   puts("Jessy Added 3 bags of chip.");
   first_dump.wait();
}
int main(){
   std::thread shoppers[NUMBER_OF_THREADS];
   for(int i = 0; i < NUMBER_OF_THREADS; i +=2){
      shoppers[i] = std::thread(milton_shopper);
      shoppers[i+1] = std::thread(jessy_shopper);
   }
   for (auto & s: shoppers){
      s.join();
   }
```

```
  printf("We need to buy %u bags of chips.\n", bags_of_chip);
  return 0;
}
```

Synchronization

As mentioned above synchronization is very important when managing threads. For example one may sometimes need to synchronize action on separate threads. Let say that one thread needs to wait for another thread to complete a task before the first thread can complete its own. Luckily for us C++ Standard Library provides to handle it, in form of *condition_variable and futures*

What is Condition Variable in C++

The cppreference.com page defines condition variable as class that is a synchronization primitives that can be used to block a thread, or multiple threads at the same time, until another thread both modifies a shared variable(the condition), and notifies the condition variable

The thread that intends to modify the variable has to

1. acquire a std::mutex (typically via std::lock_guard)
2. perform the modification while the lock is held
3. execute notify_one or notify_all on the std::condition_variable (the lock does not need to be held for notification)

Even if the shared variable is atomic, it must be modified under the mutex in order to correctly publish the modification to the waiting thread.

Any thread that intends to wait on std::condition_variable has to

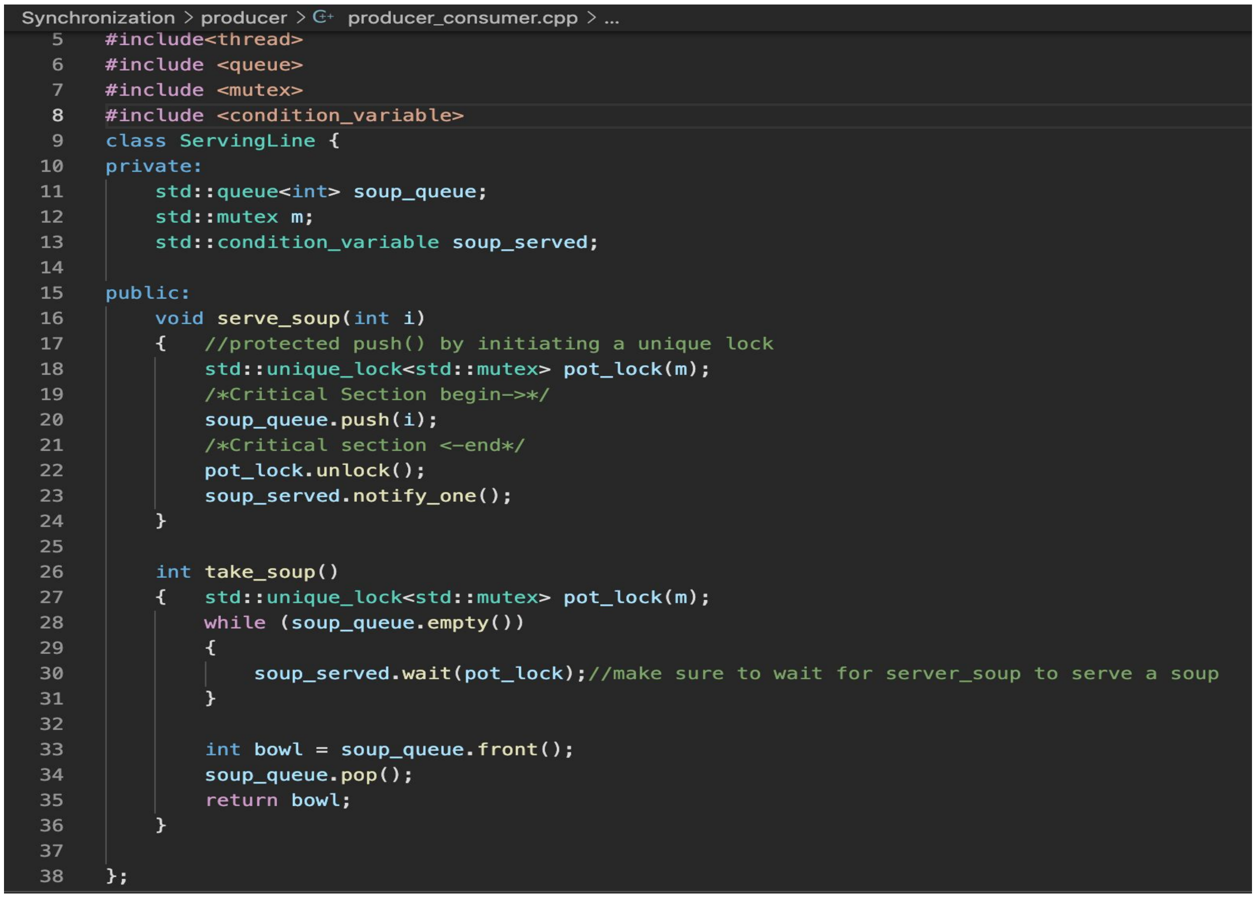
1. acquire a std::unique_lock<std::mutex>, on the same mutex as used to protect the shared variable
2. either
    1. check the condition, in case it was already updated and notified
    2. execute wait, wait_for, or wait_until. The wait operations atomically release the mutex and suspend the execution of the thread.
    3. When the condition variable is notified, a timeout expires, or a spurious wakeup occurs, the thread is awakened, and the mutex is atomically reacquired. The

thread should then check the condition and resume waiting if the wake up was spurious.

or

1. use the predicated overload of wait, wait_for, and wait_until, which takes care of the three steps above

The producer consumer example below I have created an array t of five threads, the first thread will produce(serve_soup) and the rest will consume(take_soup). A class Servingline is created with some properties, the one we are interested in is the std:condition_variable soup_served. The conditional variable works along the std::unique_lock<std::mutex> and notifies the next available thread to consume(take soup) once the unique_lock gets released. Notice in the take_soup() method the thread that calls this function gets locked and waits while soup_queue is empty. Moreover, in the while loop the condition variable soup_served waits until the condition is notified by the server_soup method, this allows multiple threads to wait while one thread finishes consuming and all consumer threads have a fair chance to take soup.



Synchronization > producer > producer_consumer.cpp > ...

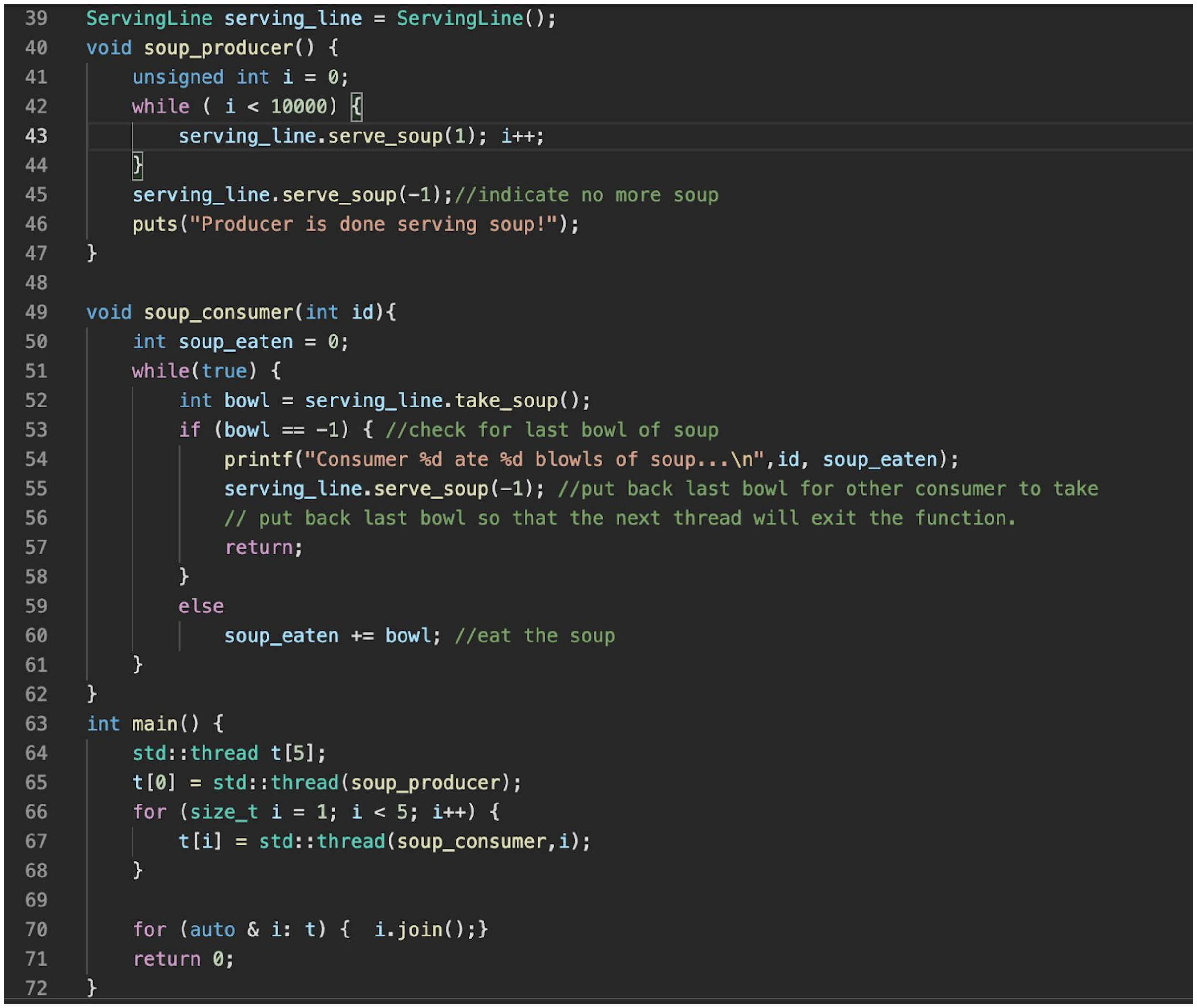
```cpp
#include<thread>
#include <queue>
#include <mutex>
#include <condition_variable>
class ServingLine {
private:
    std::queue<int> soup_queue;
    std::mutex m;
    std::condition_variable soup_served;

public:
    void serve_soup(int i)
    {   //protected push() by initiating a unique lock
        std::unique_lock<std::mutex> pot_lock(m);
        /*Critical Section begin->*/
        soup_queue.push(i);
        /*Critical section <-end*/
        pot_lock.unlock();
        soup_served.notify_one();
    }

    int take_soup()
    {   std::unique_lock<std::mutex> pot_lock(m);
        while (soup_queue.empty())
        {
            soup_served.wait(pot_lock);//make sure to wait for server_soup to serve a soup
        }

        int bowl = soup_queue.front();
        soup_queue.pop();
        return bowl;
    }

};
```



```
ServingLine serving_line = ServingLine();
void soup_producer() {
    unsigned int i = 0;
    while ( i < 10000) {
        serving_line.serve_soup(1); i++;
    }
    serving_line.serve_soup(-1);//indicate no more soup
    puts("Producer is done serving soup!");
}

void soup_consumer(int id){
    int soup_eaten = 0;
    while(true) {
        int bowl = serving_line.take_soup();
        if (bowl == -1) { //check for last bowl of soup
            printf("Consumer %d ate %d blowls of soup...\n",id, soup_eaten);
            serving_line.serve_soup(-1); //put back last bowl for other consumer to take
            // put back last bowl so that the next thread will exit the function.
            return;
        }
        else
            soup_eaten += bowl; //eat the soup
    }
}
int main() {
    std::thread t[5];
    t[0] = std::thread(soup_producer);
    for (size_t i = 1; i < 5; i++) {
        t[i] = std::thread(soup_consumer,i);
    }

    for (auto & i: t) {  i.join();}
    return 0;
}
```

Results from running in the simulator:

```
milton@Miltons-MBP producer % ./producer_consumer
Producer is done serving soup!
Consumer 1 ate 2628 blowls of soup...
Consumer 2 ate 2492 blowls of soup...
Consumer 4 ate 2403 blowls of soup...
Consumer 3 ate 2477 blowls of soup...
```

Conclusion

In my short report I have covered some techniques on how to tackle what is multithreading, data race conditions, synchronization, and how to use a third party library. For this research I have been reading books such as Concurrency in Action, Effective Modern C++, Parallel Algorithms, As you have noticed by reading my report I have just started my research therefore the report is a bit short. I will continue reading these books and adding more to my report paper. My future work will focus on memory model and operation on atomic types, Designing lock-based concurrent data structure. The goal of my research is to expand my knowledge and learn new set tools to create efficient systems that can help communities improve their lives.

Reference Page

"Sharing Data between Threads." *C++ Concurrency in Action*, by Anthony Williams, Manning, 2019.

"Std::condition_variable." *Cppreference.com*, en.cppreference.com/w/cpp/thread/

condition_variable.