Sleep and Wait

This lesson gives a brief introduction to sleep/wait and its usage in C++ with the help of interactive examples.

WE'LL COVER THE FOLLOWING

- Conventions
- Various waiting strategies

One important feature that multithreading components such as threads, locks, condition variables, and futures have in common is the notion of time.

Conventions

The methods for handling time in multithreading programs follow a simple convention: Methods ending with `_for` have to be parametrized by a time duration; methods ending with `_until` by a time point. Here is a concise overview of the methods that deal with sleeping, blocking, and waiting:

Multithreading Component	_until	_for
`std::thread th`	`th.sleep_until(in2min)`	`th.sleep_for(2s)`
`std::unique_lock lk`	`lk.try_lock_until(in2min)`	`lk.try_lock(2s)`
`std::condition_variable cv`	`cv.wait_until(in2min)`	`cv.wait_for(2s)`

`std::future fu`	`fu.wait_until(in2min)`	`fu.wait_for(2s)`
`std::shared_future shFu`	`shFu.wait(in2min)`	`shFu.wait_for(2s)`

`in2min` stands for a time 2 minutes in the future; `2s` is a time duration of 2 seconds. Although I use `auto` in the initialization of the time point `in2min`, the following is still verbose:

```
auto in2min= std::chrono::steady_clock::now() + std::chrono::minutes(2);
```

Time literals from C++14 come to our rescue when using typical time durations, e.g. `2s` stands for 2 seconds. Let's look at different waiting strategies.

Various waiting strategies

The main idea of the following program is that the promise provides its result for four shared futures. That is possible because more than one ***`shared_future`*** can wait for the notification of the same promise. Each future has a different waiting strategy. Both the promise and every future will be executed in different threads. For simplicity reasons, I will only speak about a waiting thread in the rest of this subsection, although it will be the corresponding future that is waiting. Below are the details of the promises and the futures.

Here are the strategies for the four waiting threads:

- `consumeThread2`: waits up to 20 seconds for the result of the promise.
- `consumeThread3`: asks the promise for the result and goes back to sleep for 700 milliseconds.
- `consumeThread4`: asks the promise for the result and goes back to sleep. Its sleep duration starts with 1 millisecond and doubles each time.

Here is the program:

```
// sleepAndWait.cpp


#include <utility>
#include <iostream>
#include <future>
#include <thread>
#include <utility>

using namespace std;
using namespace std::chrono;

mutex coutMutex;

long double getDifference(const steady_clock::time_point& tp1,
                          const steady_clock::time_point& tp2){
    const auto diff= tp2 - tp1;
    const auto res= duration <long double, milli> (diff).count();
    return res;
}

void producer(promise<int>&& prom){
    cout << "PRODUCING THE VALUE 2011\n\n";
    this_thread::sleep_for(seconds(5));
    prom.set_value(2011);
}

void consumer(shared_future<int> fut,
              steady_clock::duration dur){
    const auto start = steady_clock::now();
    future_status status= fut.wait_until(steady_clock::now() + dur);
    if ( status == future_status::ready ){
        lock_guard<mutex> lockCout(coutMutex);
        cout << this_thread::get_id() << " ready => Result: " << fut.get()
             << endl;
    }
    else{
        lock_guard<mutex> lockCout(coutMutex);
        cout << this_thread::get_id() << " stopped waiting." << endl;
    }
    const auto end= steady_clock::now();
    lock_guard<mutex> lockCout(coutMutex);
    cout << this_thread::get_id() << " waiting time: "
         << getDifference(start,end) << " ms" << endl;
}

void consumePeriodically(shared_future<int> fut){
    const auto start = steady_clock::now();
    future_status status;
    do {
        this_thread::sleep_for(milliseconds(700));
        status = fut.wait_for(seconds(0));
        if (status == future_status::timeout) {
            lock_guard<mutex> lockCout(coutMutex);
            cout << "     " << this_thread::get_id()
                 << " still waiting." << endl;
        }
        if (status == future_status::ready) {
            lock_guard<mutex> lockCout(coutMutex);
            cout << "     " << this_thread::get_id()
                 << " waiting done => Result: " << fut.get() << endl;
        }
```

```
    } while (status != future_status::ready);
    const auto end= steady_clock::now();
    lock_guard<mutex> lockCout(coutMutex);
    cout << "     " << this_thread::get_id() << " waiting time: "
         << getDifference(start,end) << " ms" << endl;
}

void consumeWithBackoff(shared_future<int> fut){
    const auto start = steady_clock::now();
    future_status status;
    auto dur= milliseconds(1);
    do {
        this_thread::sleep_for(dur);
        status = fut.wait_for(seconds(0));
        dur *= 2;
        if (status == future_status::timeout) {
            lock_guard<mutex> lockCout(coutMutex);
            cout << "         " << this_thread::get_id()
                 << " still waiting." << endl;
        }
        if (status == future_status::ready) {
            lock_guard<mutex> lockCout(coutMutex);
            cout << "         " << this_thread::get_id()
                 << " waiting done => Result: " << fut.get() << endl;
        }
    } while (status != future_status::ready);
    const auto end= steady_clock::now();
    lock_guard<mutex> lockCout(coutMutex);
    cout << "         " << this_thread::get_id()
         << " waiting time: " << getDifference(start,end) << " ms" << endl;
}

int main(){

    cout << endl;

    promise<int> prom;
    shared_future<int> future= prom.get_future();
    thread producerThread(producer, move(prom));

    thread consumerThread1(consumer, future, seconds(4));
    thread consumerThread2(consumer, future, seconds(20));
    thread consumerThread3(consumePeriodically, future);
    thread consumerThread4(consumeWithBackoff, future);

    consumerThread1.join();
    consumerThread2.join();
    consumerThread3.join();
    consumerThread4.join();
    producerThread.join();

    cout << endl;

}
```

I create the promise in the main function (line 98), use the promise to create the associated future (line 99), and move the promise into a separate thread (line 100). I have to move the promise into the thread because it does not support the copy semantic. That is not necessary for the shared futures (lines 102 - 105); they support the copy semantic and can, therefore, be copied.

Before I talk about the work package of the thread, let me say a few words about the auxiliary function `getDifference` (lines 14 - 19). The function takes two-time points and returns the time duration in milliseconds. I will use the function a few times.

What about the five created threads?

- `producerThread`: executes the function `producer` (lines 21 - 25) and publishes its result 2011 after 5 seconds of sleep. This is the result the futures are waiting for.

- `consumerThread1`: executes the function `consumer` (lines 27 - 44). The thread is waiting for 4 seconds at most (line 30) before it continues with its work; This waiting period is not long enough to get the result from the promise.

- `consumerThread2`: executes the function `consumer` (lines 27 - 44). The thread is waiting 20 seconds at most before it continues with its work.

- `consumerThread3`: executes the function `consumePeriodically` (lines 46 - 67). It sleeps for 700 milliseconds (line 50) and asks for the result of the promise (line 60). Because of the `std::chrono::seconds(0)` in line 51, there is no waiting. If the result of the calculation is available, it will be displayed in line 60.

- `consumerThread4`: executes the function `consumeWithBackoff` (lines 69 - 92). It sleeps in the first iteration 1 second and doubles its sleeping period every iteration. Otherwise, its strategy is similar to the strategy of `consumerThread3`.

Now to the synchronization of the program. Both the clock determining the current time and `std::cout` are shared variables, but no synchronization is necessary. Firstly, the method call `std::chrono::steady_clock::now()` is thread-safe (for example in lines 30 and 40); Secondly, the C++ runtime guarantees that the characters will be written *thread-safe* to `std::cout`. I only used a `std::lock_guard` to wrap `std::cout` (for example in lines 32, 37, and 41).

`std::lock_guard` to wrap `std::cout` (for example in lines 32, 37, and 41).

Although the threads write one after the other to `std::cout`, the output is not easy to understand.

The first output is from the promise, with the remaining outputs from the futures. At first `consumerThread4` asks for the result. The output is indented by 8 characters. `consumerThread4` also displays its ID. `consumerThread3` immediately follows. Its output is indented by 4 characters. The output of `consumerThread1` and `consumerThread2` is not indented.

- `consumeThread1`: waits unsuccessfully 4000.18 ms seconds without getting the result.
- `consumeThread2`: gets the result after 5000.3 ms although its waiting duration is up to 20 seconds.
- `consumeThread3`: gets the result after 5601.76 ms. That's about 5600 milliseconds= 8 * 700 milliseconds.
- `consumeThread4`: gets the result after 8193.81 ms. To say it differently. It waits 3 seconds too long.