C++ Programming II

STL - Concurrent Programming I

C++ Programming II
October 29, 2018

Prof. Dr. P. Arnold
Bern University of Applied Sciences

Agenda

- **Producer / Consumer Idiom**

Agenda

- **Producer / Consumer Idiom**
- **Condition Variables**

Producer / Consumer Idiom

Producer / Comsumer Idiom

implemented with C++11 `std::condition_variable`

We are going to implement a typical producer/consumer program with multiple threads. The general idea is that there is:

- one thread is producing items and puts them into a queue.
- another thread is consuming such items.
- if there is nothing to produce, the producer thread sleeps.
- if there is no item in the queue to consume, the consumer sleeps.

Producer / Comsumer Idiom

implemented with C++11 `std::condition_variable`

We are going to implement a typical producer/consumer program with multiple threads. The general idea is that there is:

- one thread is producing items and puts them into a queue.
- another thread is consuming such items.
- if there is nothing to produce, the producer thread sleeps.
- if there is no item in the queue to consume, the consumer sleeps.

Since the queue is a shared resources among two threads it has to be protected by a mutex.

Producer / Comsumer Idiom

implemented with C++11 `std::condition_variable`

We are going to implement a typical producer/consumer program with multiple threads. The general idea is that there is:

- one thread is producing items and puts them into a queue.
- another thread is consuming such items.
- if there is nothing to produce, the producer thread sleeps.
- if there is no item in the queue to consume, the consumer sleeps.

Since the queue is a shared resources among two threads it has to be protected by a mutex.

- But what does the consumer do if there is no item in the queue? Polling?

Producer / Comsumer Idiom

implemented with C++11 `std::condition_variable`

We are going to implement a typical producer/consumer program with multiple threads. The general idea is that there is:

- one thread is producing items and puts them into a queue.
- another thread is consuming such items.
- if there is nothing to produce, the producer thread sleeps.
- if there is no item in the queue to consume, the consumer sleeps.

Since the queue is a shared resources among two threads it has to be protected by a mutex.

- But what does the consumer do if there is no item in the queue? Polling?
- That is not necessary since we can let the consumer wait for *wakeup events* that are triggered by the producer, whenever there are new items.

Producer / Comsumer Idiom

implemented with C++ 11 `std::condition_variable`

We are going to implement a typical producer/consumer program with multiple threads. The general idea is that there is:

- one thread is producing items and puts them into a queue.
- another thread is consuming such items.
- if there is nothing to produce, the producer thread sleeps.
- if there is no item in the queue to consume, the consumer sleeps.

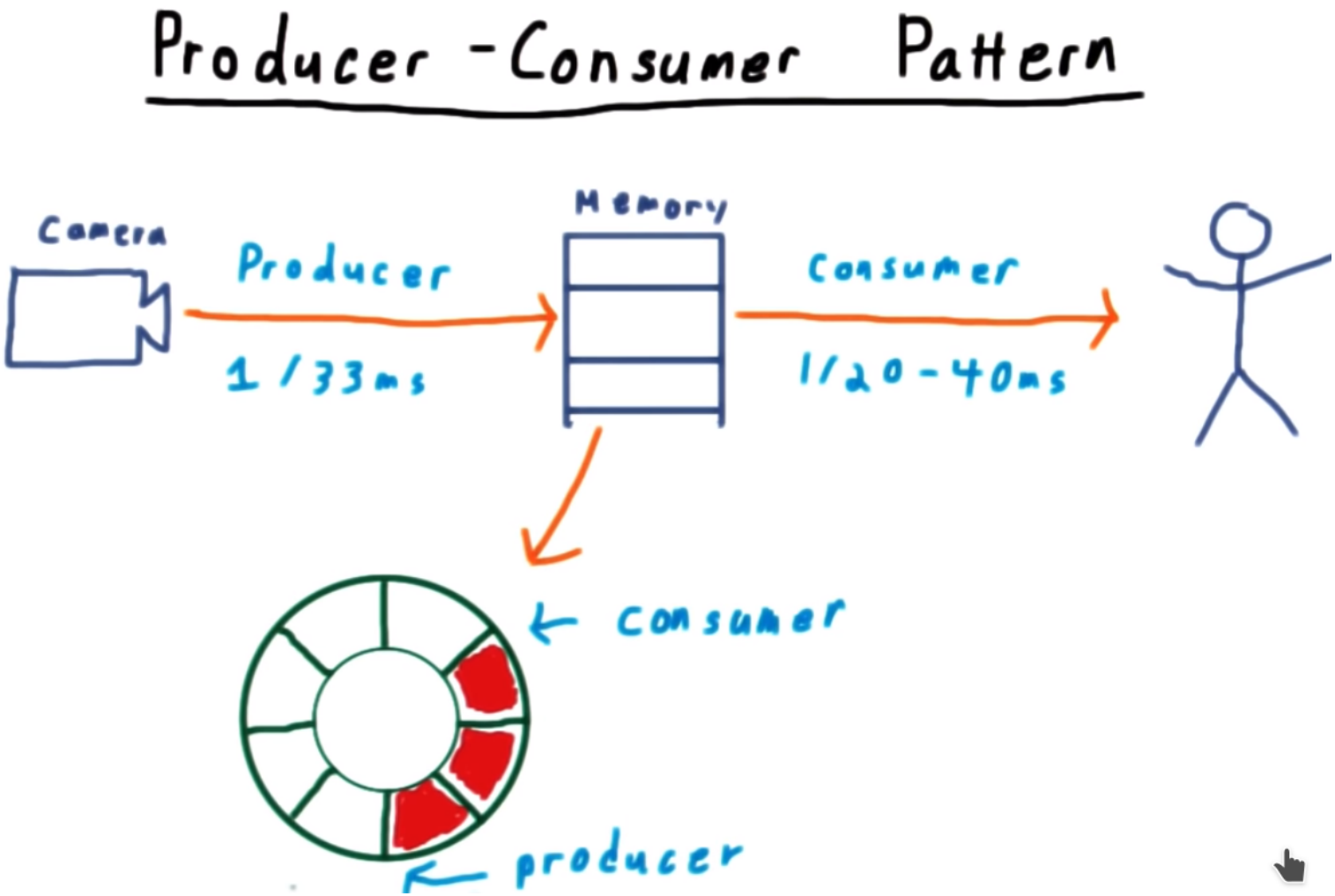
Since the queue is a shared resources among two threads it has to be protected by a mutex.

- But what does the consumer do if there is no item in the queue? Polling?
- That is not necessary since we can let the consumer wait for *wakeup events* that are triggered by the producer, whenever there are new items.

C++ 11 provides a nice data structure called `std::condition_variable` for this kind of events.

Producer / Comsumer Idiom

using a ring buffer queue (FIFO) as shared memory



Producer / Comsumer Idiom

implemented using polling

- First, adding includes and used variables

```
1  #include <iostream>
2  #include <queue> // FIFO Buffer
3  #include <thread>
4  #include <mutex>
5
6  using namespace std;
7  using namespace std::chrono_literals;
8
9  deque<int> q;
10 mutex mu;
11 bool finished{false};
```

Producer / Comsumer Idiom

implemented using polling

- First, adding includes and used variables

```
1  #include <iostream>
2  #include <queue> // FIFO Buffer
3  #include <thread>
4  #include <mutex>
5
6  using namespace std;
7  using namespace std::chrono_literals;
8
9  deque<int> q;
10 mutex mu;
11 bool finished{false};
```

- The producer produces `nbrItems` items

```
1  void produce(size_t nbrItems)
2  {
3      for(size_t count{0}; count < nbrItems; ++count)
4      {
5          std::this_thread::sleep_for(1s); // Producing time
6          std::lock_guard<mutex> qLocker{mu};
7          q.push_front(count);
8      }
9
10     std::lock_guard<mutex> flagLocker{mu};
11     finished = true;
12 }
```

Producer / Comsumer Idiom

implemented using polling

- The consumer consumes until the queue runs empty.

```
void consume()
{
    while (!finished)
    {
        while(!q.empty())
        {
            std::unique_lock<mutex> qLocker{mu};
            cout << "Got " << q.back() << " from Queue" << endl;
            q.pop_back();
            qLocker.unlock();
        }
    }
}
```

Producer / Comsumer Idiom

implemented using polling

- In the main function, we start a producer thread which produces 10 items and a consumer thread.

```
1  int main()
2  {
3      size_t nbrItems{10};
4      std::thread t1{produce, nbrItems};
5      std::thread t2{consume};
6      t1.join();
7      t2.join();
8      cout << "Finished!"<< endl;
9      return 0;
10 }
```

Producer / Comsumer Idiom

implemented using polling

- In the main function, we start a producer thread which produces 10 items and a consumer thread.

```
1  int main()
2  {
3      size_t nbrItems{10};
4      std::thread t1{produce, nbrItems};
5      std::thread t2{consume};
6      t1.join();
7      t2.join();
8      cout << "Finished!"<< endl;
9      return 0;
10 }
```

Output:

```
Got 0 from Queue
Got 1 from Queue
Got 2 from Queue
Got 3 from Queue
Got 4 from Queue
Got 5 from Queue
Got 6 from Queue
Got 7 from Queue
Got 8 from Queue
Got 9 from Queue
Finished!
```

Producer / Comsumer Idiom

implemented using polling

- The consumer consumes until the queue runs empty.

```
void consume()
{
    while (!finished)
    {
        while(!q.empty())
        {
            std::unique_lock<mutex> qLocker{mu};
            cout << "Got " << q.back() << " from Queue" << endl;
            q.pop_back();
            qLocker.unlock();
        }
    }
}
```

- Note: The code works, but If the producer thread is inactive, this leads to continuous locking and unlocking of the mutex
- Needless burns CPU cycles!
- A time out is very difficult to choose → Demo
- `std::condition_variable` provide an elegant solution

Condition Variables

Producer / Comsumer Idiom

implemented using `std::condition_variable`

- First, adding new includes and used variables

```
1  #include <iostream>
2  #include <queue> // FIFO Buffer
3  #include <thread>
4  #include <mutex>
5  #include <condition_variable>
6
7  using namespace std;
8  using namespace std::chrono_literals;
9
10 deque<int> q;
11 mutex mu;
12 condition_variable cv;
13 bool finished{false};
```

Producer / Comsumer Idiom

implemented using `std::condition_variable`

- Then we add notifications in the producer

```
1  void produce(size_t nbrItems)
2  {
3      for(size_t count{0}; count < nbrItems; ++count)
4      {
5          std::this_thread::sleep_for(500ms); // Producing time
6          {
7              std::lock_guard<mutex> qLocker{mu};
8              q.push_front(count);
9          }
10         cv.notify_one(); // Notify new element
11     }
12
13     {
14         std::lock_guard<mutex> flagLocker{mu};
15         finished = true;
16     }
17     cv.notify_one(); // Notify production finished
18 }
```

- The `std::condition_variable` *cv* is used for **signaling a condition**

Producer / Comsumer Idiom

implemented using `std::condition_variable`

- On the consumer side, we catch the notification using `wait()` function

```
1  void consume()
2  {
3      while (!finished)
4      {
5          std::unique_lock<mutex> qLocker{mu};
6          cv.wait(qLocker, []{return !q.empty() || finished; });
7
8          while(!q.empty())
9          {
10             cout << "Got " << q.back()
11                  << " from Queue" << endl;
12             q.pop_back();
13         }
14     }
15 }
```

Producer / Comsumer Idiom

implemented using `std::condition_variable`

- On the consumer side, we catch the notification using `wait()` function

```
1  void consume()
2  {
3      while (!finished)
4      {
5          std::unique_lock<mutex> qLocker{mu};
6          cv.wait(qLocker, []{return !q.empty() || finished; });
7
8          while(!q.empty())
9          {
10             cout << "Got " << q.back()
11                  << " from Queue" << endl;
12             q.pop_back();
13         }
14     }
15 }
```

- The `std::condition_variable` *cv* is used for **waiting for a specific condition**
- `cv.wait(lock, predicate)` will wait until `predicate()` returns true. No polling or continuously unlocking and locking required!
- To wake a thread up that blocks on the `wait` call of a `condition_variable` object, another thread has to call the `notify_one()` or `notify_all()` method on the same object.
- Then sleeping threads are waked up in order to check if `predicate()` holds.

Thank You

Questions

???