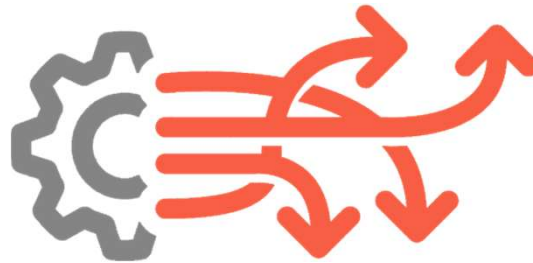


Multithreading

in modern C++



Valentina Gaggero

Index:

1 – key concepts

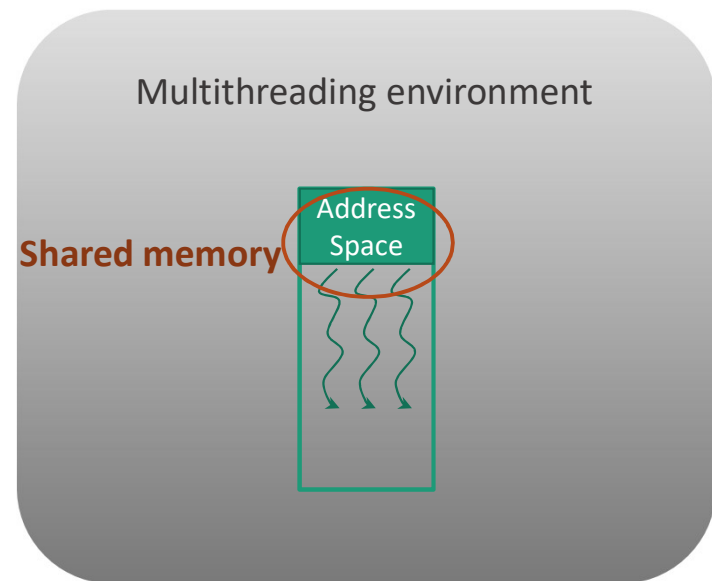
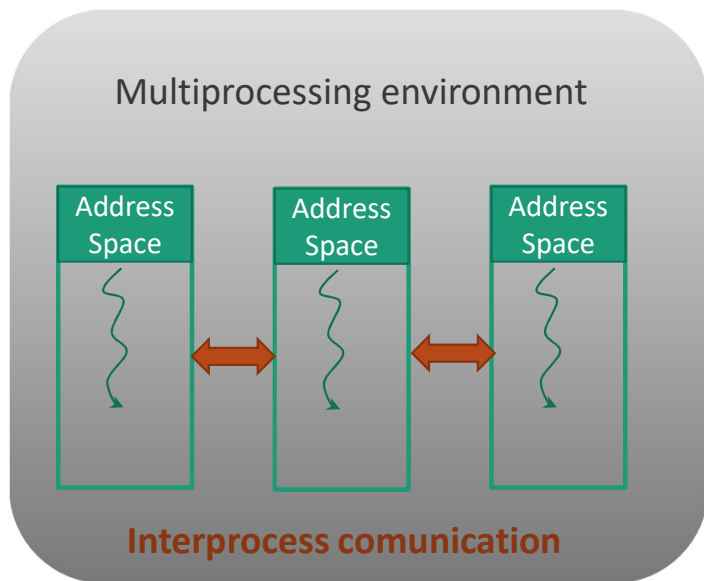
2 – threads

3 – sharing data between threads

4 – threads synchronization

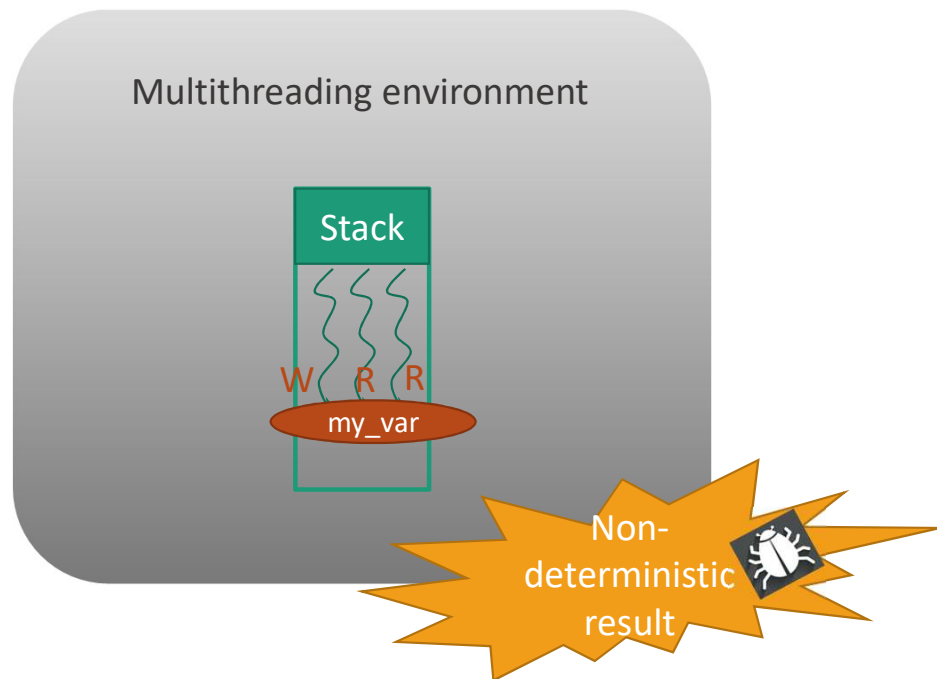
1 – **key concepts** (1/3)

Concept 1: Multiprocessing vs Multithreading



1 – key concepts (2/3)

Concept 2: Data race

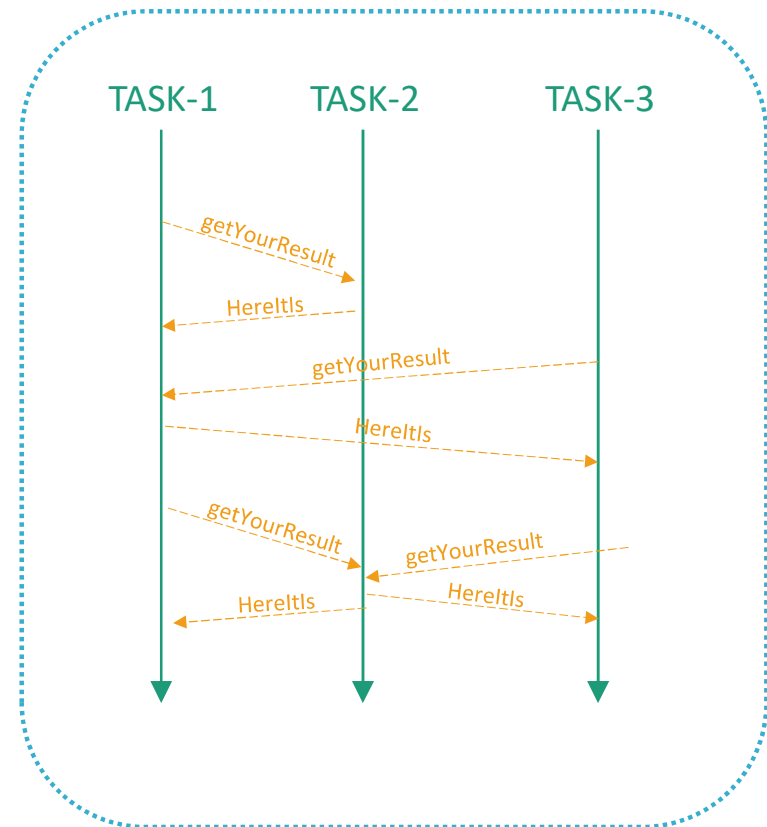
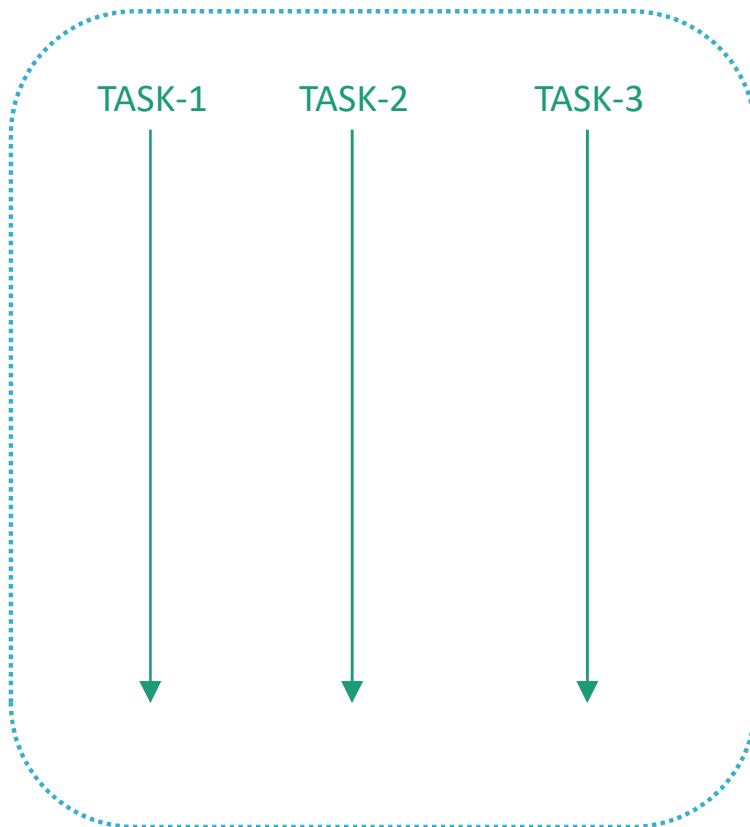


A multi-thread program has a data race if all of these conditions are true:

1. Multiple threads access the same memory location
2. At least one operation is a write access
3. Memory operations don't follow sorting rules, as "Do X before Y"

1 – **key concepts** (3/3)

Concept 3: parallelism vs concurrency



2 – Threads (1/4)

```
int main ()  
{  
    DoSomething();  
    return 0;  
}
```



Create a thread that
executes the main

If we need to perform other tasks



We need to create a thread per task

`std::thread`

- ✓ Represents a single thread of execution
- ✓ Its execution starts immediately after its creation

```
#include <thread>  
std::thread my_th(myTaskFunction)
```



Which type of task?

2 – Threads (2/4)

Type of task we can create

Do something and the main thread waits the result

The main thread is not interested in the result of the task (background task)

```
#include <thread>
use namespace std;
int main ()
{
    DoSomething();
    thread my_th (myTaskFunction);
    DoSomethingElse();
    my_th.join();
    return 0;
}
```

The main thread “waits” the my_th accomplishes its task

```
#include <thread>
use namespace std;
int main ()
{
    DoSomething();
    thread my_th(myTaskFunction);
    my_th.detach();
    DoSomethingElse();
    return 0;
}
```

The main thread continues its works, while the detached thread is running.

If you don't decide before the std::thread object is destroyed, then your program is terminated (the std::thread destructor calls std::terminate()).

2 – Threads (3/4)

How to pass arguments to a thread

```
void retrieveData(uint32_t num, myObject_t obj);
int main()
{
    uint32_t mynum = 3;
    myObject_t myobj;
    .....

    thread my_th (retrieveData, mynum, myobj);
    ...
}
```

by default the arguments are *copied*

```
void stringManipulation(uint32_t num, std::string const &str);
void myfunction()
{
    uint32_t mynum = 15;
    char buffer[100];
    .....

    std::string(buffer)
    thread my_th (stringManipulation, mynum, buffer);
    my_th.detach();
}
```

the std::thread constructor copies the supplied values as is, without converting to the expected argument type.

Avoids dangling pointers

2 – Threads (4/4)

How to pass arguments to a thread by reference

```
void retrieveData(uint32_t num, myObject_t &obj);

int main()
{
    uint32_t mynum = 1000;
    myObject_t myobj;
    .....
    std::ref(myobj)
    thread my_th (retrieveData, mynum, myobj);
    ...
    my_th.join();
    myobj.getComputedData();
}
```

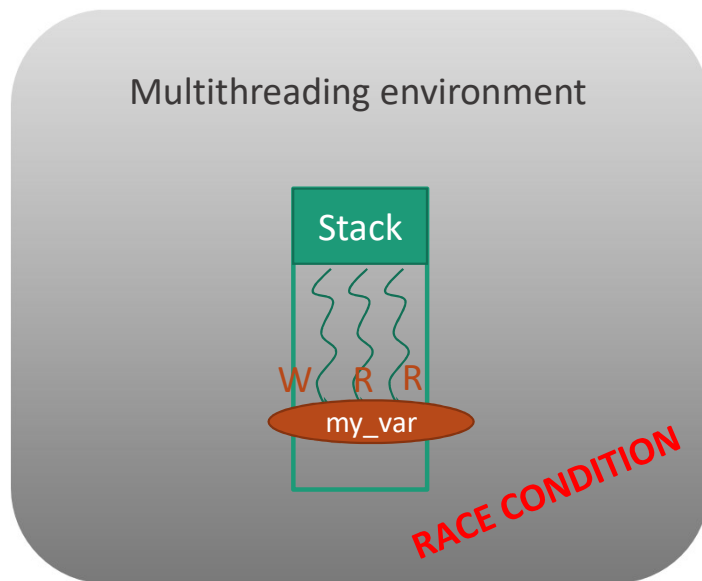
will be correctly passed a reference to data rather than a reference to a *copy* of data.



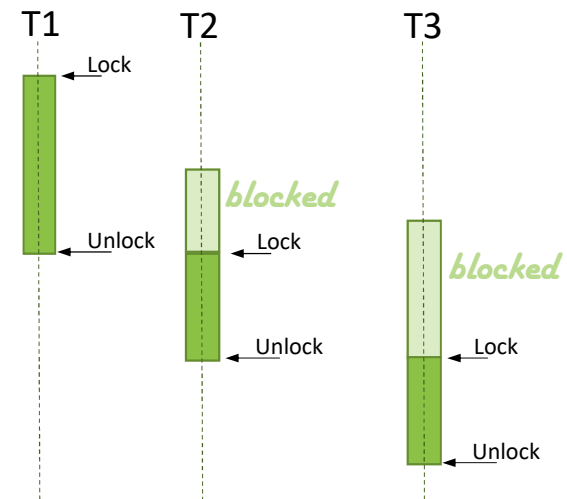
Will I have the updated data in the myobj?

NO!!!

3 – sharing data between threads (1/5)



SOLUTION:
MUTEX (MUTal EXclusion)



3 – sharing data between threads (2/5)

Mutex downsides (1/3)

```
std::mutex mymutex;

void thisIsAFunction()
{
    mymutex.lock();
    //inside the critical section
    bool someError = doSomething();
    if(someError)
        return;
    ....
    mymutex.unlock();
    //end critical section
}
```

Forget to unlock
(also when an exception has been thrown)



Automatic Lock/unlock by RAII
lock_guard or **unique_lock**

```
std::mutex mymutex;

void thisIsAFunction()
{
    std::lock_guard<std::mutex> guard(mymutex);
    //inside the critical section
    bool someError = doSomething();
    if(someError)
        return;
    ....
} //end critical section
```

```
std::mutex mymutex;

void thisIsAFunction()
{
    std::unique_lock<std::mutex> ul(mymutex);
    //inside the critical section
    bool someError = doSomething();
    if(someError)
        return;
    ....
    mymutex.unlock();
    ... //end critical section
    ...
}
```

3 – sharing data between threads (3/5)

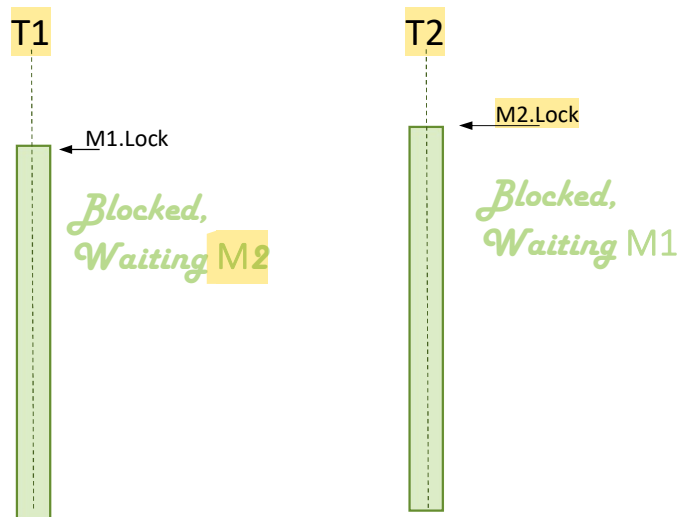
Mutex downsides (2/3)

Deadlock

- T1 and T2 need to lock more resources
- T1 and T2 lock in different order



Multiple lock: all-or-nothing by RAII
scoped_lock

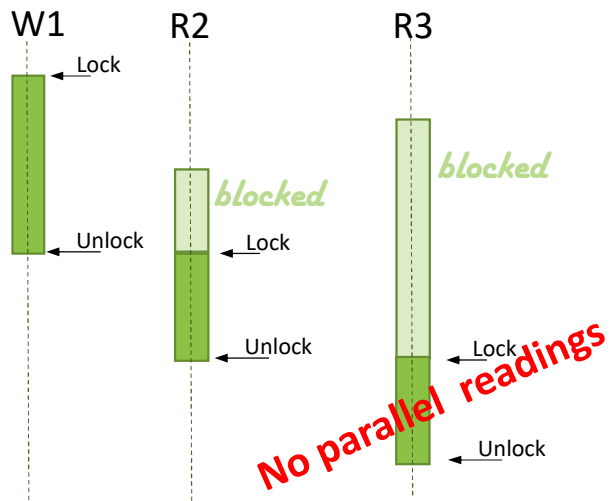


```
std::mutex mymutex1, mymutex2;

void thisIsAFunction()
{
    std::scoped_lock<std::mutex> sl(mymutex1, mymutex2);
    //inside the critical section
    bool someError = doSomething();
    if(someError)
        return;
    //end critical section
}
```

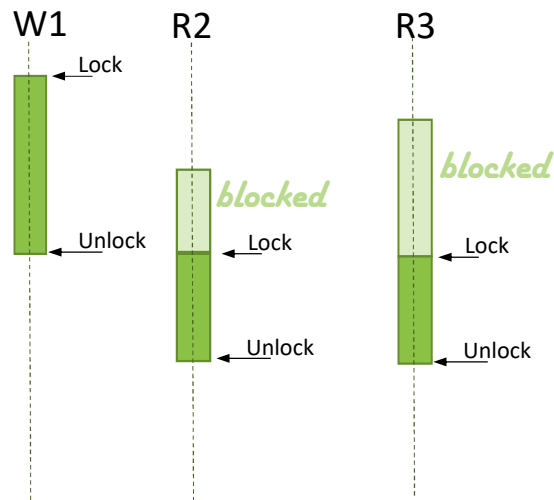
3 – sharing data between threads (4/5)

Mutex downsides (3/3)



3 – sharing data between threads (4/4)

Mutex downsides (3/3)



shared locks:
std::shared_mutex + (unique_lock or shared_lock)

```
std::shared_mutex sh_mutex;

void theWriter()
{
    std::unique_lock<std::shared_mutex> ul(sh_mutex);
    //inside the critical section
    writeData();
    //end critical section
}

void theReader()
{
    std::shared_lock<std::shared_mutex> sl(sh_mutex);
    //inside the critical section
    readData();
    //end critical section
}
```

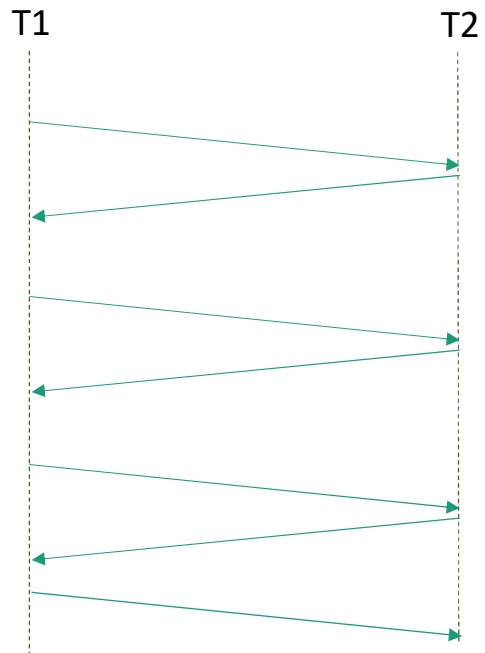
3 – sharing data between threads (4/5)

Mutex - summary

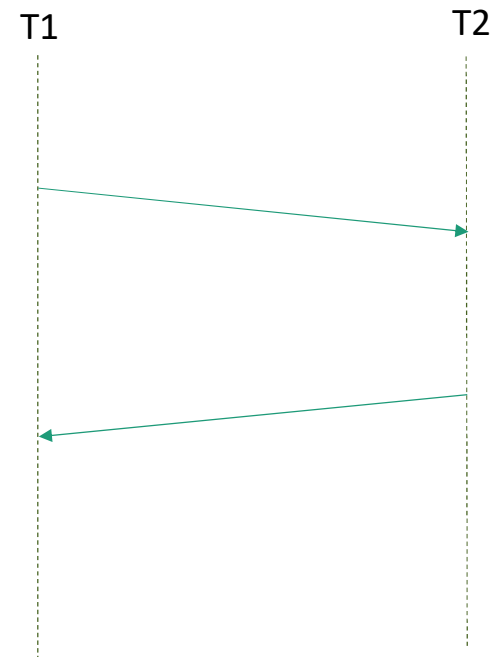
Type	Info
<code>std::mutex</code>	<ul style="list-style-type: none">- Lock/unlock API- Need to remember to unlock (also in exception case)
<code>std::lock_guard</code>	<ul style="list-style-type: none">- Locking with RAII- No need to remember to unlock- Just declare and use
<code>std::unique_lock</code>	Like <code>std::lock_guard</code> + lock/unlock API
<code>std::shared_mutex</code>	<code>std::unique_lock</code> + <code>std::shared_lock</code> (1 write, more readers)
<code>std::scoped_lock</code>	Like <code>std::lock_guard</code> , but for multiple mutex (to avoid deadlock)

4-threads synchronization (1/6)

Use case 1: collaborative threads



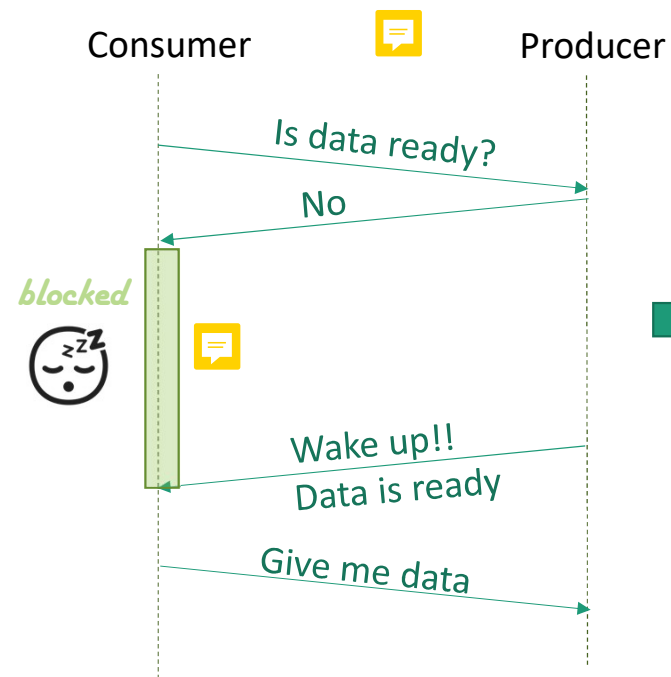
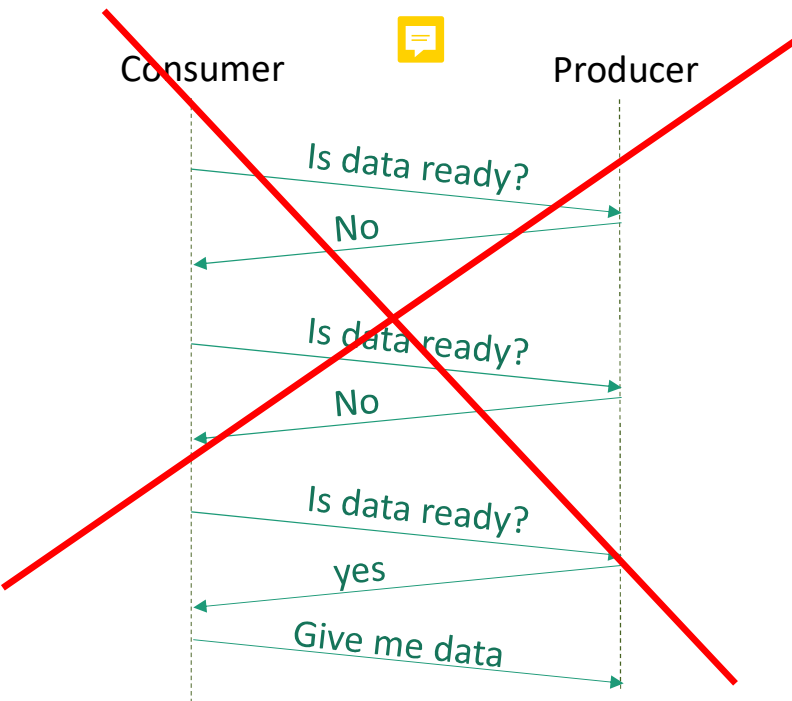
Use case 2: one-off interaction



4-threads synchronization (2/6)

Use case 1: collaborative threads

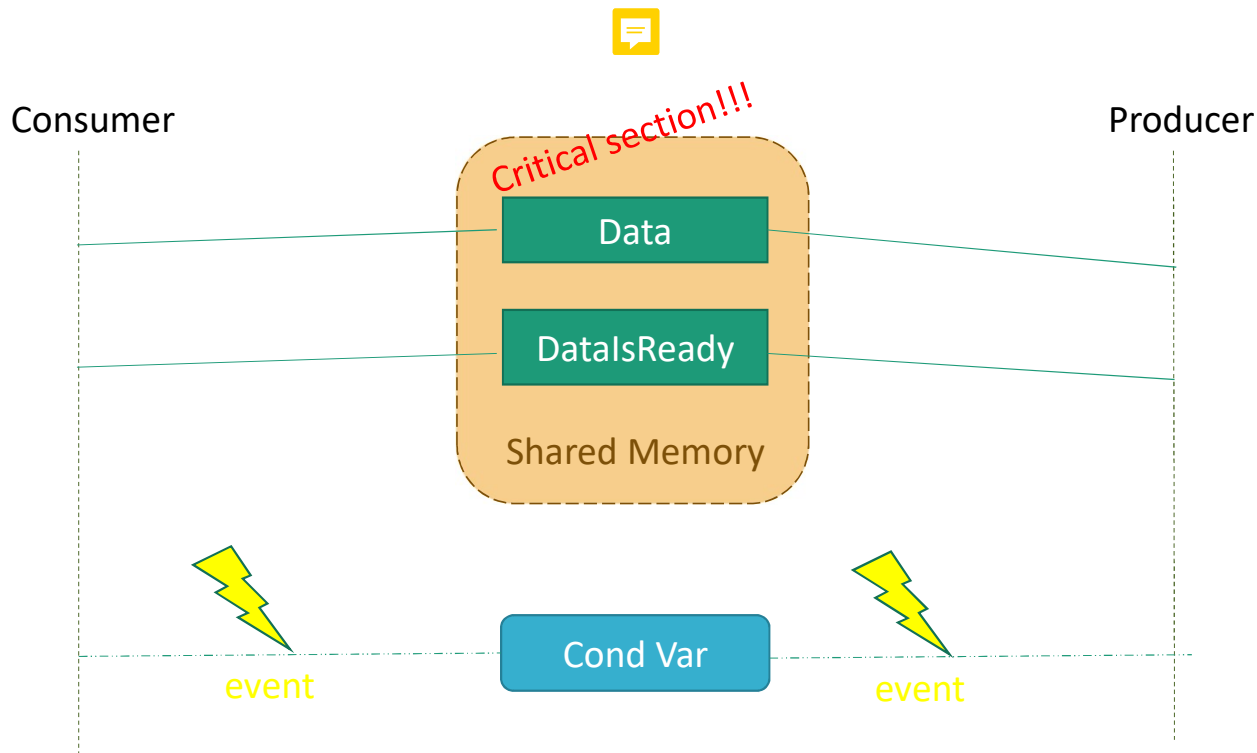
One thread waits an event triggered by another thread. *Example: producer/consumer use case*



➡ Condition variables

4 – synchronize threads (3/6)

Condition Variables



Hands-on example

4 - threads synchronization (4/4)

Use case 2: one-off interaction – get the result of a thread

```
#include <thread>
use namespace std;
int main ()
{
    DoSomething();
    thread my_th (myTaskFunction);
    my_th.join();
    return 0;
}
```



How I retrieve the result of the detached thread?

~~my_th.get() ????~~

```
#include <future>
use namespace std;
int main ()
{
    DoSomething();
    future f = async(calculateSomething);
    DoOtherstuff();
    cout << "the result is " << f.get() << endl;
    return 0;
}
```



std::async function template

4-threads synchronization (5/6)

Where does *calculateSomething* run?

```
#include <feature>
use namespace std;
int main ()
{
    DoSomething();
    future f = async(calculateSomething);
    DoOtherstuff();
    cout << "the result is " << f.get() << endl;
    return 0;
}
```

std::launch policy

→ std::launch::deferred

→ std::launch::async

→ std::launch::deferred | std::launch::async

4 – **threads synchronization** (6/6)

Use case 2: one-off interaction – future & promise

Hands - on exercise

Thank you !!!