Parallel programming
C++11 threads
Part 1

FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

C++11 threads? - What is it?

- A new standard of C++11 defines API for threads and synchronization primitives.
- As the standard is accepted by all the modern compilers, it is **portable** to the majority of operating systems.
- More high-level than pthreads, **easier** to write clean code.
- Disadvantages:
  - Not all synchronization primitives are implemented, e.g., barriers (do not worry, you will implement it next week :-)).
  - A modern compiler is needed.

How to use C++11 threads

- C++11 threads require to:
  - **include** thread header to your source code

```
#include <thread>
```

  - add **pthread static library** and **c++11 support** to compilation process (for compilation with gcc, clang or MinGW)

```
g++ main.cpp –std=c++11 -pthread
```

  - in case of Cmake (multiplatform)

```
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
find_package(Threads)
# set sources, add executable ...
target_link_libraries(${PROJECT_NAME} ${CMAKE_THREAD_LIBS_INIT})
```

Thread creation - constructor

- **`thread thread( Function&& start_routine, Args&&... args );`**
- **Parameters:**
  - ***`start_routine`* – function that will be executed by the thread**
  - ***`args`* – arguments for the *start_routine* function**
    - **if the *start_routine* is a class member function, the first argument in *args* has to be the instance of that class**

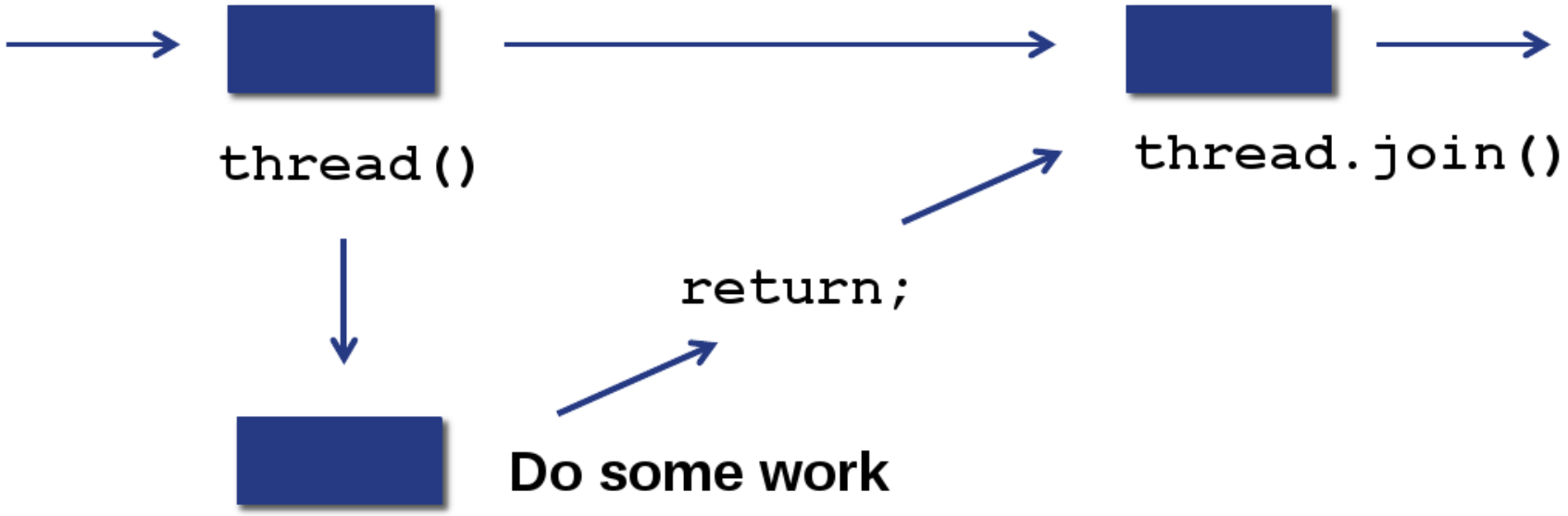
C++ trivia: && represents rvalue reference, which is like “normal” reference but can bind to temporary values (values to be destroyed)

Thread termination

- Thread **terminates** when:
  - It reaches the end of the start_routine
  - It calls *return;*

- **Note:**
  - **Return value**
    - **It is not possible to obtain return code from thread**
    - **If you need to return a value you have to use... hmm... no, wait for next week ;-)**

Joining threads



- `void thread.join()`
  - The calling thread waits for the callee thread to terminate.
  - It is not possible to join the same thread more than once.
    - `bool thread.joinable()` - checks if it is possible to join the thread
      - A finished thread that was not joined yet is joinable!
  - Not joining a thread leads to a process crash (if thread is joinable, its destructor calls `std::terminate()`)
    - Can be mitigated by calling `thread.detach()`, although it is not recommended http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-detached_thread

Hello world!

`lab_codes/src/HelloThreads.cpp`

Counting with threads

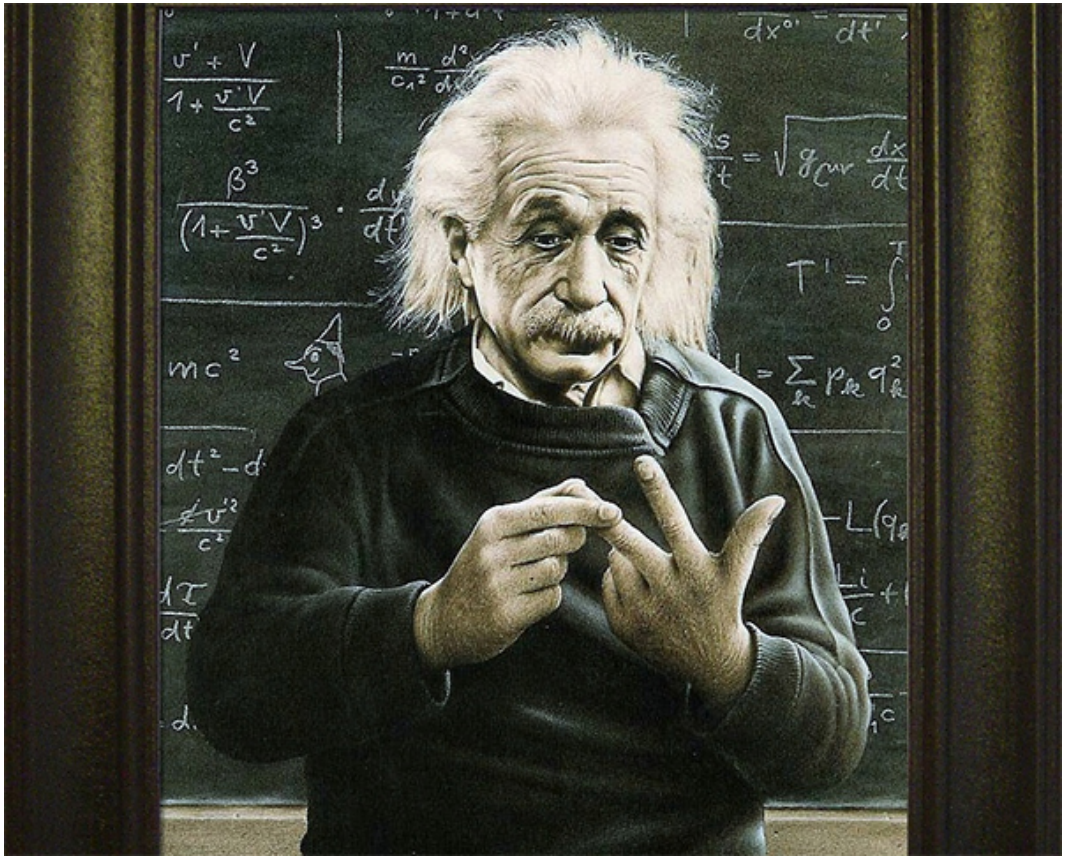
- Example – Counter
  - Task:
    - Create global integer variable ***counter***
    - Create 4 threads and each thread:
      - 10000000-times increment the ***counter***
    - Print the resulting value of the ***counter*** after all the threads are done!

Counter – first try

`lab_codes/src/CounterFirstTry.cpp`

4 * 10000000 = ???

- **Something is wrong... probably.**
- **Don't worry. We are gonna take a look where is a mistake!**



Race condition

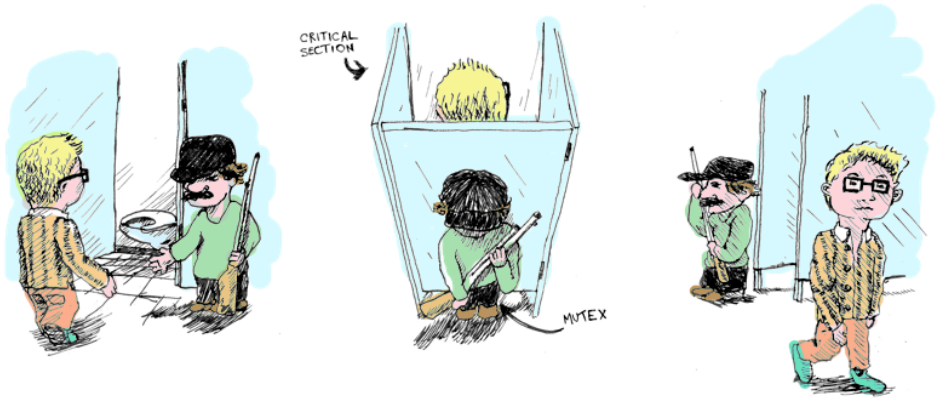
- The problem is that code statements “take time” (or can be composed of **multiple statements**) and can be “interrupted” by other threads attempting to modify the same data.
- This is called a **race condition**: the final result depends on the precise order in which the instructions are executed.
- This issue is addressed using **mutexes** (mutual exclusion).
- They ensure that shared data are accessed and modified by a **single thread**.

Detecting race condition

- Intel Inspector

Mutex

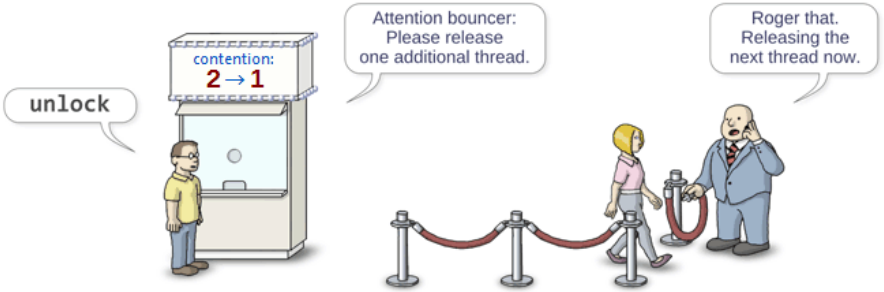
- Mutex protect **regions of code** (critical section) by allowing only **one** thread to execute it at one time.
- A mutex can be in one of two states: **locked** or **unlocked**.
  - Mutex provides functions **lock()** and **unlock()**
- Typical mutex workflow:
  - **Create and initialize** a mutex variable
  - Several threads attempt to **lock** the mutex before entering the critical section
  - **Only one succeeds** and that thread owns the mutex (other threads are blocked) – mutex becomes **locked**
  - The owner thread **performs** some set of actions in the critical section
  - The **owner unlocks** the mutex
  - **One of the blocked threads** is awoken, acquires the mutex and the process is repeated



Mutex in C++11 threads - API

- **`#include <mutex>`**
  - Include the header file with mutex object
- **`mutex mutex;`**
  - Creates new mutex.

- **`void mutex.lock()`**
  - Locks a mutex; blocks if another thread has locked this mutex and owns it.
- **`void mutex.unlock()`**
  - Unlocks mutex; after unlocking, other threads get a chance to lock the mutex.
- **`bool mutex.try_lock()`**
  - Tries to lock the mutex. Returns immediately. On successful lock acquisition returns true, otherwise returns false.



Lock guard - API

- The mutexes can be encapsulated by **lock_guard** classes, that simplify the usage - RAII idiom

`lock_guard<mutex> lock_guard(mutex_type& m)`

  - lock the mutex in **lock_guard** instance initialization
  - unlock the mutex in **lock_guard** instance destruction (e.g., instance goes out of scope)

```
mutex myMutex;
{
    lock_guard<mutex> myGuard(myMutex);   // <-- Locks myMutex.
    // Perform some stuff on shared variables.
} // <-- myGuard goes out of scope, its destructor is called which unlocks myMutex
```

Unique lock - API

- **unique_lock** is similar to **lock_guard** (i.e., unlocks on destruction), but is also useful for more advanced use cases
  - Manual locking and unlocking
  - Some C++ functions work primarily with **unique_lock**

- **`unique_lock<mutex> unique_lock(mutex_type& m)`**
  Takes mutex **m** and and locks it
- **`unique_lock<mutex> unique_lock(mutex_type& m, std::defer_lock)`**
  Takes mutex **m** and and keeps it unlocked (i.e., defers the lock)
- **`unique_lock.lock()`**
  Locks the **unique_lock**
- **`unique_lock.unlock()`**
  Unlocks the **unique_lock**

It is time to repair our counter!

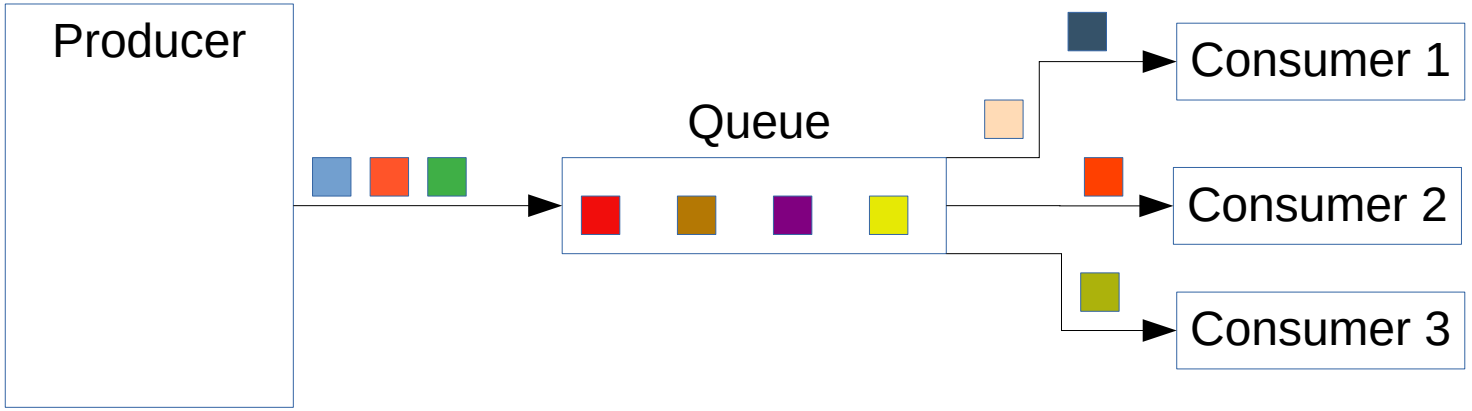
- Now, you know how to repair our Counter example.
- So, let's do it.

Counter – second try

`lab_codes/src/CounterSecondTry.cpp`

Producer/Consumers

- Example – Producer/Consumers
  - Task:
    - Implement Producer/Consumers
    - One producer generates random values that are put into a queue
    - Consumers consume values from the queue
    - After all values have been produced and consumed, the program stops



Producer/Consumers

- Example – Producer/Consumers
  - Task:
    - Use **busy waiting** approach
      - Each consumer periodically checks the queue whether it has some value to consume or if all values were produced+consumed
      - You will need mutexes for that

Producer/Consumers

`lab_codes/src/ProducerConsumerBusyWaiting.cpp`

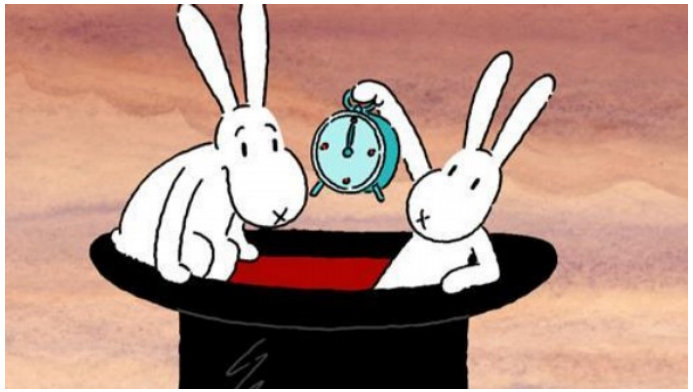
Busy waiting not so great

- Busy waiting unnecessarily consumes the CPU time due to periodic locking and unlocking (*overhead time*)
  - Intel VTune

- Better approach: consumers' **wait** until producer tells them that there is some new value to consume
  - Signaling using **condition variables**

Condition variables

- Allows **signaling** among threads
- Threads can wait until some **event** occurs
- Another thread wakes up the **waiting** thread and inform it that the situation already occurred
- The woken up thread should **check** if all conditions are fulfilled and then continues.



Condition variables - API

- **`#include <condition_variable>`**
  - Include the header with the condition variable interface
- **`void condition_variable.notify_one()`**
  - Sends a signal to a single thread waiting on condition variable***.***
- **`void condition_variable.notify_all()`**
  - Sends a signal to all threads waiting for ***`condition_variable.`***
- **`void condition_variable.wait(unique_lock<mutex>& lock)`**
  - Unlocks ***`lock`*** and puts the thread to sleep until another thread wake it up by sending a signal. When the thread is woken up ***`lock`*** is locked again.
- **`void condition_variable.wait(unique_lock<mutex>& lock, Predicate pred)`**
  - Waits for the event to occur
  - Has the following semantics

```
while (!pred())
    cv.wait(lk);
```

Condition variables – waiting

```
void condition_variable.wait(unique_lock<mutex>& lock, Predicate pred)
```

- When wait() is called by thread
  - Thread locks **lock**
  - Predicate **pred** is called and its value checked
    - If true: the thread continues, i.e., wait() returns while **lock** is still locked
    - If false: the thread unlocks **lock** and the thread is put in blocked state
- When a blocked thread gets notification from another thread
  - Thread locks **lock**
  - Predicate **pred** is called and its value checked
    - If true: the thread continues, i.e., wait() returns while **lock** is still locked
    - If false: the thread unlocks **lock** and the thread is put in blocked state

There is also wait() function without the predicate argument, but it is **not recommended** (problems with lost and spurious wakeups), see this link

Producer/Consumers

- Example – Producer/Consumers
  - Task:
    - Implement Producer/Consumers using **condition variables**

Producer/Consumers

`lab_codes/src/ProducerConsumerCondVar.cpp`

Improved overhead time

- Using condition variables improves the overhead time

References

- Tutorial to C++11 concurrency:
  - C++11 Multithreading
- C++11 threads standard
  - http://en.cppreference.com/w/cpp/thread
- An introduction to Parallel programming
  - Peter Pacheco, University of San Francisco
  - Morgan Kaufmann Publishers is an imprint of Elsevier
- Top 20 C++ threads mistakes