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
  - 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

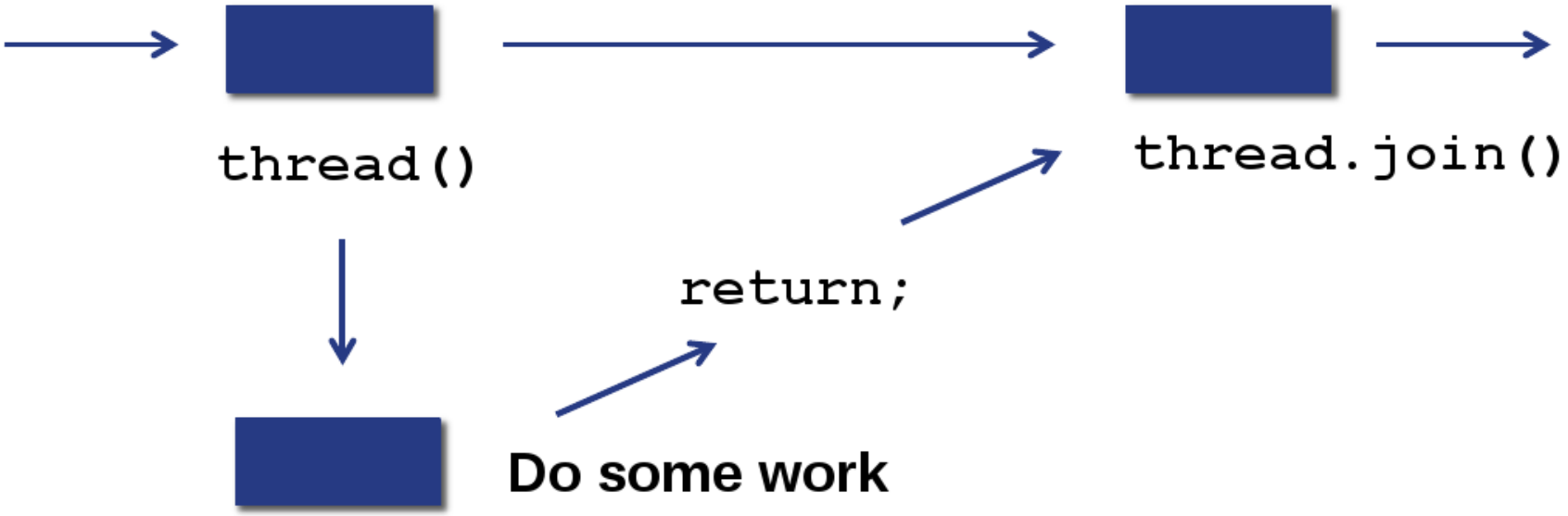
- **`thread thread( Function&& start_routine, Args&&... args );`**
- **Parameters:**
  - ***`start_routine`* – function that will be executed by the thread**
  - ***`args`* – arguments for the *start_routine* function**
    - **if the *start_routine* is a class member function, the first argument in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

Thread termination

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

- **Note:**
  - **The thread releases its stack during termination.**
  - **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 one 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/HelloThreads.cpp`

Counting with threads

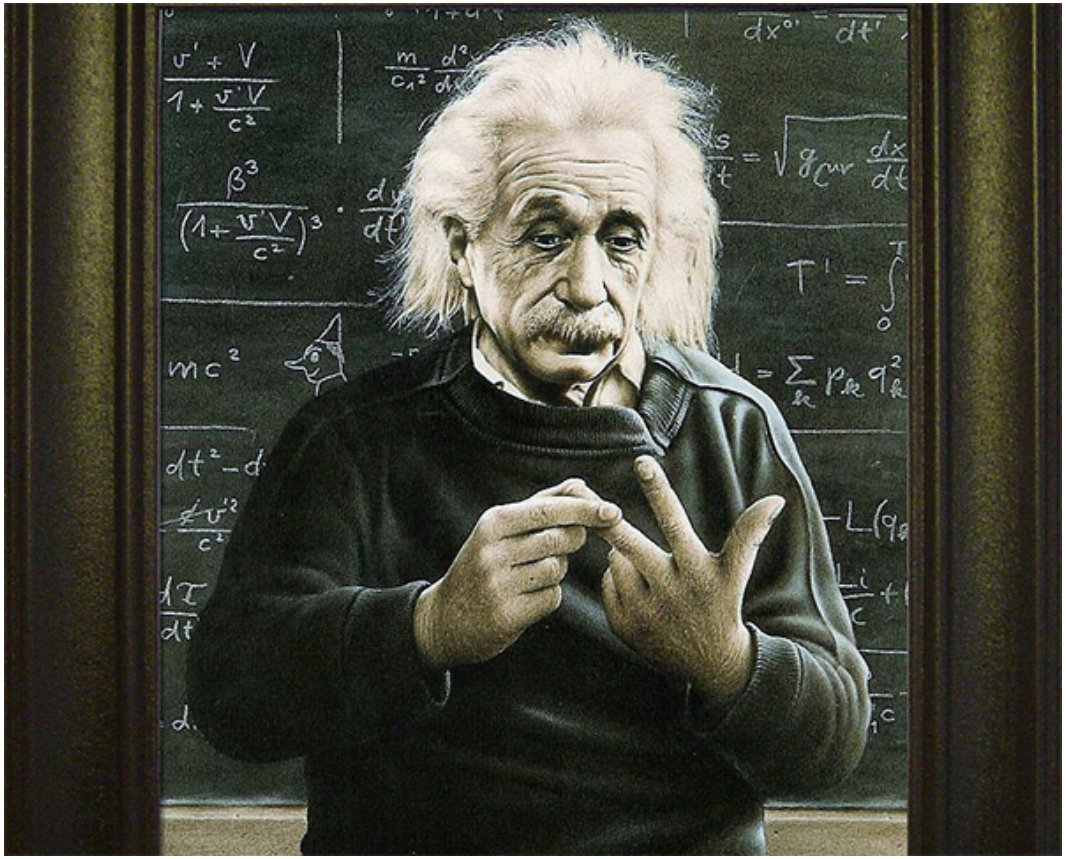
- Example – Counter
  - Task:
    - Create global integer variable ***`counter`***
    - Create 4 threads and each thread:
      - 10000000-times increment the ***`counter`***
    - Print the resulting value of the ***`counter`*** after all the threads are done!

Counter – first try

`lab_codes/CounterFirstTry.cpp`

4 * 10000000 = ???

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



The risks of multi-threaded programming

- Let's assume that a well-known bank company has asked you to implement a multi-threaded code to perform bank transactions.
- You start with the modest goal of allowing deposits.
- Clients deposit money and the amount gets credited to their accounts.
- As a result of having multiple threads running concurrently the following can happen:

Thread 0	Thread 1	Account balance
Client requests a deposit	Client requests a deposit	0 CZK
Check current balance = 0 CZK		0 CZK
	Check current balance = 0 CZK	0 CZK
Ask for deposit 1000 CZK	Ask for deposit 2000 CZK	0 CZK
	Compute new balance = 2000CZK	0 CZK
Compute new balance = 1000CZK	Write new balance to account	2000 CZK
Write new balance to account		**1000 CZK** ☹

Race condition

- The problem is that many operations “take time” 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.
- Unless Thread 0 completes its update before Thread 1 (or vice versa) we get an incorrect result.
- This issue is addressed using **mutexes** (mutual exclusion).
- They ensure that shared data are accessed and modified by a **single thread**.

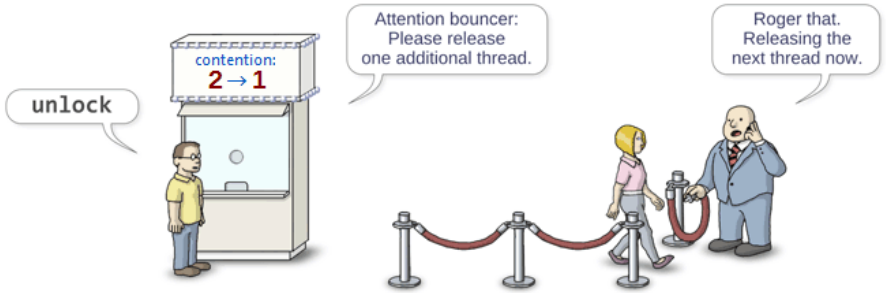
Mutex

- A mutex can only be in two states: **locked** or **unlocked**.
- Once a thread locks a mutex:
  - Other threads attempting to lock the same mutex are **blocked.**
  - Only the thread that **initially locked** the mutex has the ability to **unlock it**.
- This allows to protect **regions of code**.
- Typical mutex workflow:
  - **Create and initialize** a mutex variable
  - Several threads attempt to **lock** the mutex
  - **Only one succeeds** and that thread owns the mutex (other threads are blocked)
  - The owner thread **performs** some set of actions
  - The **owner unlocks** the mutex
  - **Another thread** acquires the mutex and repeats the process

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, e.g., they automatically unlock the held mutex during their destruction (exceptions) – RAII idiom.
  - **`lock_guard<mutex> lock_guard(mutex_type& m)`**
    - Takes mutex ***m*** and and locks it. Mutex is unlocked when the lock guard is destroyed (e.g., goes out of scope).
- Use **unique_lock** for more advanced use cases.

It is time to repair our counter!

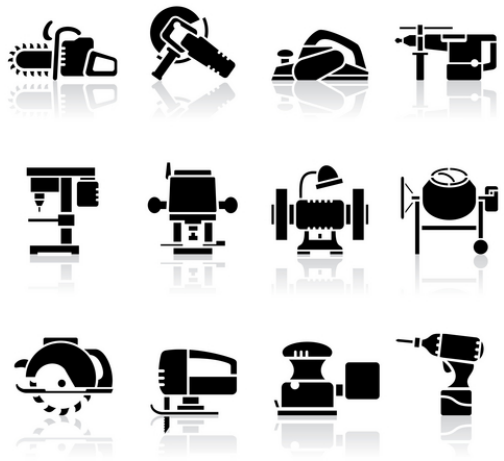
- Now, you know how to repair our Counter example.
- So, let's do it.

Counter – second try

`lab_codes/CounterSecondTry.cpp`

Everything repaired?

- Tool rental simulator
  - Rental shop offers – hammer, screwdriver, saw
  - Three handy guys:
    1) Libor: Borrow hammer, work, borrow screwdriver, work, return all
    2) Honza: Borrow screwdriver, work, borrow saw, work, return all
    3) Premek: Borrow saw, work, borrow hammer, work, return all
  - They are doing that repeatedly.
  - Work means a short delay.

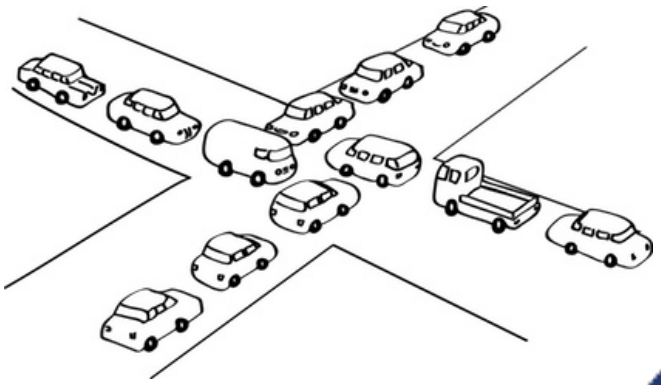


Tool rental – first try

`lab_codes/ToolRentalFirstTry.cpp`

It is stuck somehow - Deadlock

- Guy Libor borrows a **hammer** and work 
- Guy Honza borrows a **screwdriver** and work 
- Guy Premek borrows a **saw** and work 
- Guy Libor needs a **screwdriver** – waits for it
- Guy Honza needs a **saw** – waits for it
- Guy Premek needs a **hammer** – waits for it
- No one returns anything in this case.

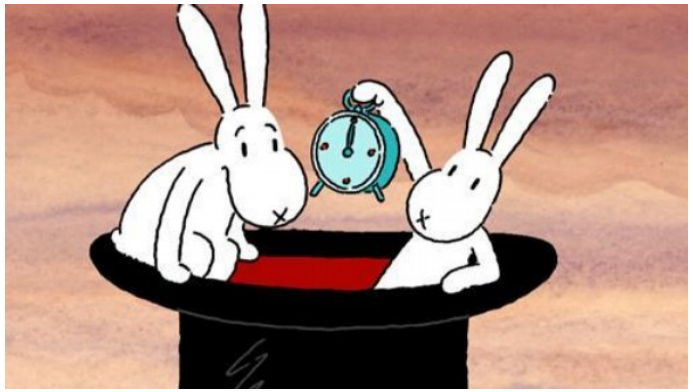


Solution?

- After using a tool, return it.
- Use an additional mutex for acquiring multiple tools.
- Or...

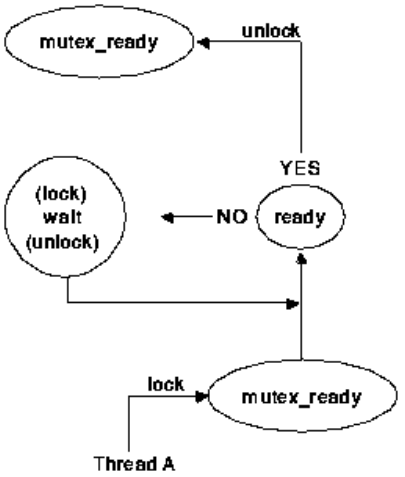
Condition variables

- 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.




```
{
    unique_lock<mutex> lk(mtx)
    cv.wait(lk);
    compute_something();
}
```

- **`void condition_variable.wait(unique_lock<mutex>& lock, Predicate pred)`**
  - Semantically equals to:

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

It is time to repair our counter!

- Now, you should be able to repair our Tool rental simulator example.
- So, let's do it.

Tool rental – second try

`lab_codes/ToolRentalSecondTry.cpp`

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