Parallel programming

C++11 threads

Libor Bukata a Jan Dvořák

FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

C++11 threads? - What is it?

- Standard **thread support library** for C++
- Defined in **C++ 11 standard**
- Language built-in support for
  - threads
  - mutual exclusion
  - condition variables
  - futures

Why C++11 threads

- A new standard of C++11 defined 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.
- Support for **atomicity** and memory ordering.
- Disadvantages:
  - Not all synchronization primitives are implemented, e.g. barriers, read-write locks, semaphores...
  - A modern compiler is needed, it is not so well tested as pthreads.

Basic building blocks

- 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 on gcc, clang or MinGW)

```
g++ hellothreads.cpp –std=C++11 -lpthread
```

  - in case of Cmake, you can add flag by

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -lpthread")
set (CMAKE_CXX_STANDARD 11)
set (CMAKE_CXX_STANDARD_REQUIRED ON)
```

Hello world! Object oriented...

```
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>

using namespace std;
using namespace std::chrono;

class Company {
	public:
		void finishProject() {
			vector<thread> workers;
			int numOfWorkers = thread::hardware_concurrency();
			for (int jobId = 0; jobId < numOfWorkers; ++jobId)
				workers.push_back(thread(&Company::doJob, this, jobId));

			for (thread& worker : workers)
				worker.join();

			cout<<"Project completed..."<<endl;
		}
	private:
		void doJob(int id) {
			this_thread::sleep_for(chrono::seconds((6*id+3) % 5));
			cout<<"The job "<<id<<" has been completed!"<<endl;
			// The result of the ,,job" can be saved to a private variable.
		}
};

int main()	{
	Company noname;
	noname.finishProject();
	return 0;
}
```

Thread creation - constructor

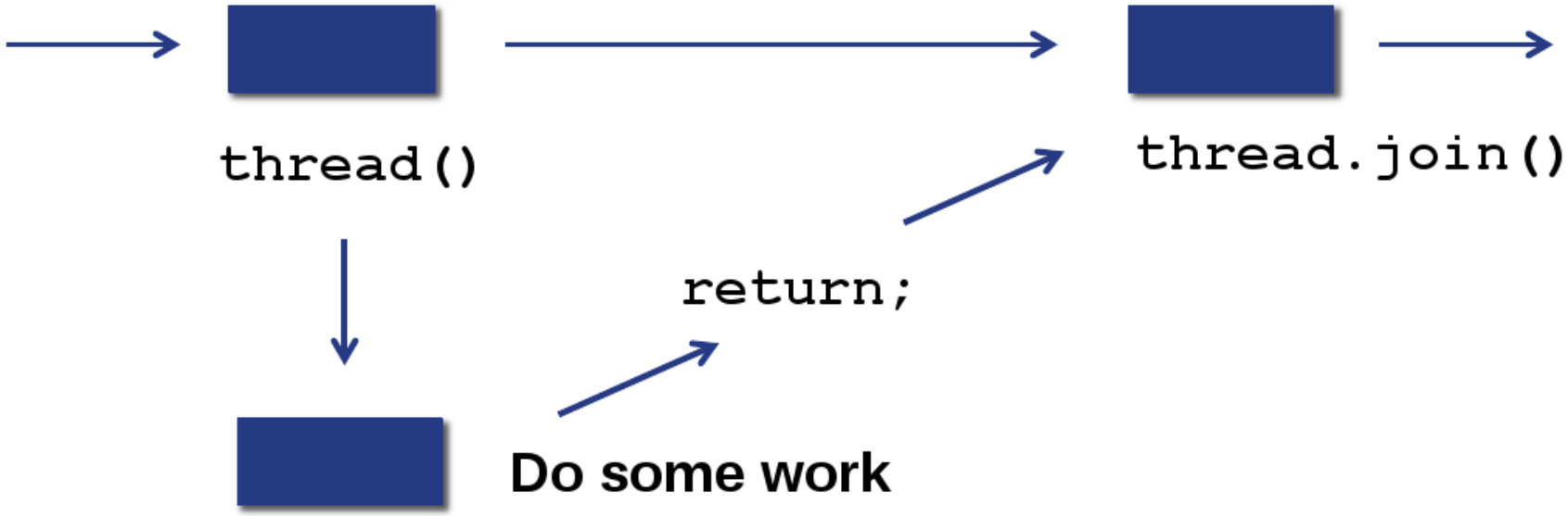
- **`thread thread( Function&& f, Args&&... args );`**
- **Parameters:**
  - ***f*** **– function that will be executed by the thread**
  - ***args*** **– arguments for the start_routine function**
    - **if the start routine *f* is a class member function, the first argument has to be the object of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

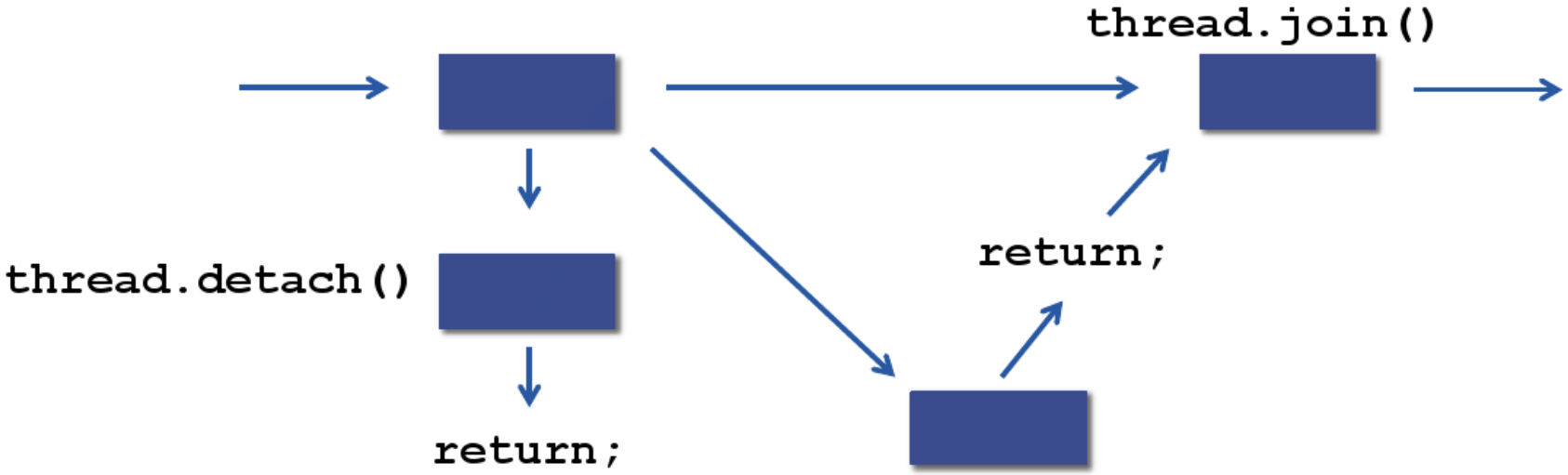


- `void thread.join();`
  - The function waits for the thread to terminate.
  - It is not possible to join one thread more than once.
    - `bool thread.joinable()` - checks if it is possible to join the thread

What happens if the thread is not joined?

- After the thread was terminated, the internal data are stored for further usage.
- The **`thread.join()`** function reads this data to provide status information about terminated thread. Afterwards, the function wipes the date out.
- If the **`thread.join()`** function is not called we need to let system know that we do not care about the thread and it can release the data.
- It can cause a serious memory leak problem when huge number of threads is used or each thread returns huge structure if those data are not wiped out.

Detaching threads



- **`void thread.detach();`**
  - The function marks the thread identified by thread as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

Let's try it.

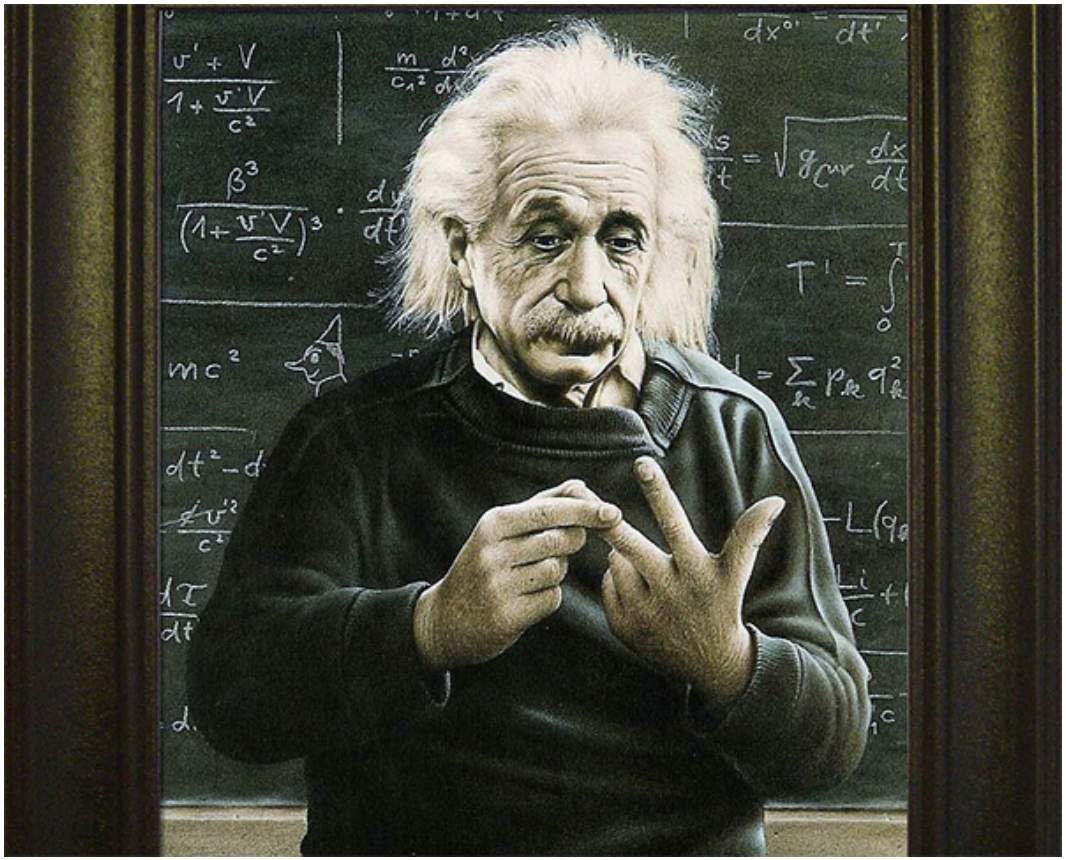
- 1. Example – Counter
  - Task:
    - Create global integer variable ***`counter`***
    - Create 4 threads and each thread:
      - 10000000-times increment the ***`counter`***
    - Print the resulting value of the ***`counter`*** after all the threads are done!

Counter – Naive solution

```cpp
#include <iostream>
#include <thread>
#include <vector>
using namespace std;

int counter = 0;

void counterThread()
{
    for(int i = 0; i < 10000000; i++)
        counter++;
    return;
}

int main() {
    vector<thread> threads;
    for(int i = 0; i < 4; i++)
        threads.push_back(thread(counterThread));

    for(int i = 0; i < 4; i++)
        threads[i].join();
    cout << counter << endl;
    return 0;
}
```

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:

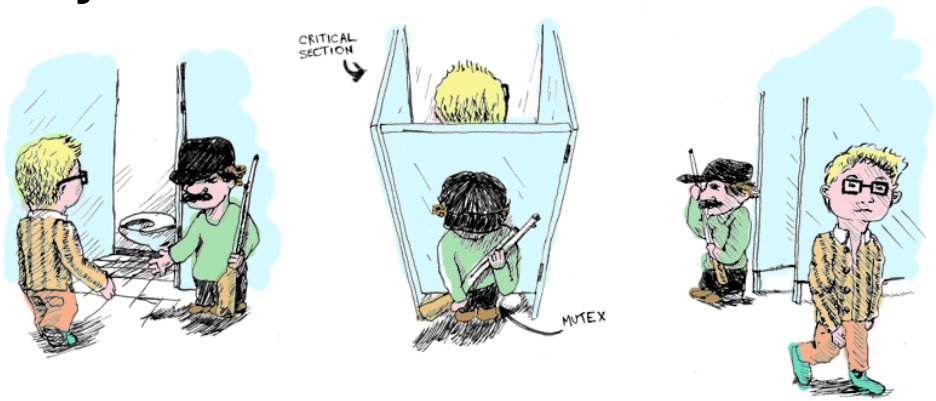
Thread 0	Thread 1	Account balance
Client requests a deposit	Client requests a deposit	0 CZK
Check current balance = 0 CZK		0 CZK
	Check current balance = 0 CZK	0 CZK
Ask for deposit 1000 CZK	Ask for deposit 2000 CZK	0 CZK
	Compute new balance = 2000CZK	0 CZK
Compute new balance = 1000CZK	Write new balance 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 certain common pieces of data are accessed and modified by a **single thread**

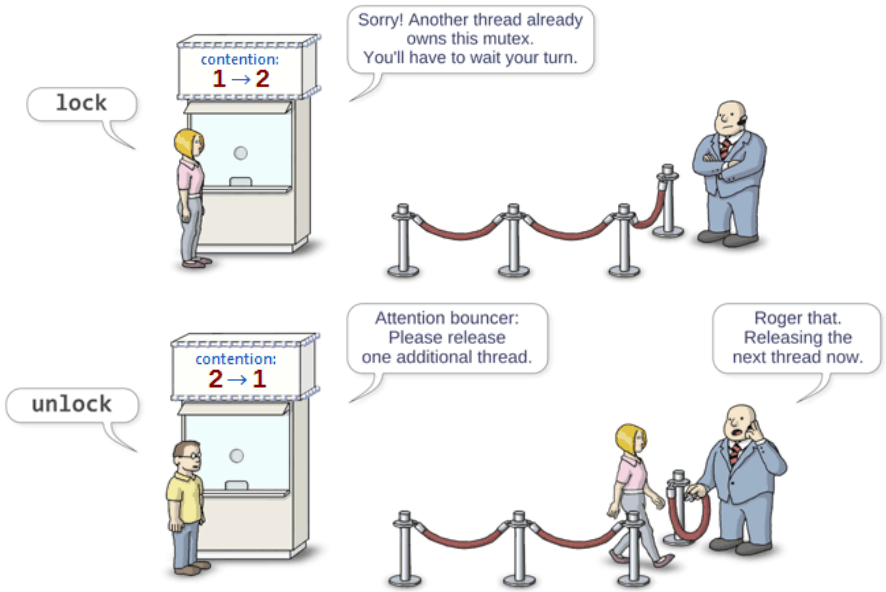
Mutex

- 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
  - The owner thread **performs** some set of actions
  - The **owner unlocks** the mutex
  - **Another thread** acquires the mutex and repeats the process
  - The mutext should be **destroyed** at the end.



Mutex in C++11 threads - API

- **`#include <mutex>`**
  - Include the header file with mutex interface
- **`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.



Unique lock - API

- The mutexes are encapsulated by **unique_lock** classes, that simplify the usage, e.g. they automatically unlock the held mutex during their destruction (exceptions).
- **`unique_lock unique_lock(mutex_type& m)`**
  - Takes mutex ***m*** and and locks it
- **`unique_lock unique_lock(mutex_type& m, std::defer_lock_t t)`**
  - Takes mutex ***m*** and and keeps it unlocked
- **`unique_lock.lock()`**
  - Locks the unique_lock
- **`unique_lock.unlock()`**
  - Unlocks the unique_lock

```
void doJob(int id) {
        unique_lock<mutex> outputGuard(mMtx, defer_lock);
        this_thread::sleep_for(chrono::seconds((6*id+3) % 5));
        outputGuard.lock();
        cout<<"The job "<<id<<" has been completed!"<<endl;
        // The result of the ,,job" can be saved to a private variable.
}

mutex mMtx;
```

It is time to repair our counter!

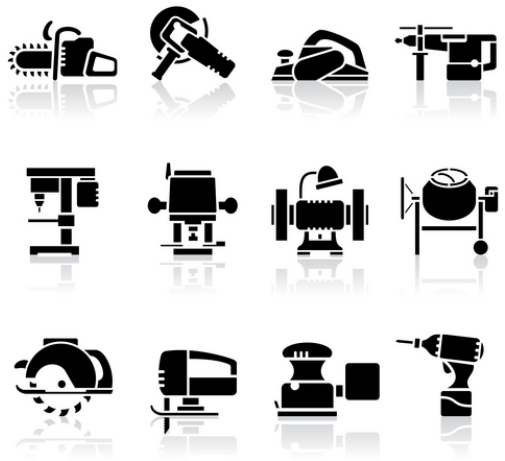
- Now, you know how to repair our Example 1.
- So, let's do it.

Aaah, that's the solution

```cpp
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
using namespace std;

int counter = 0;
mutex counter_mutex;

void counterThread()
{
    for(int i = 0; i < 10000000; i++)
    {
        unique_lock<mutex> counter_lock(counter_mutex);
        counter++;
    }
    return;
}

int main() {
    vector<thread> threads;
    for(int i = 0; i < 4; i++)
        threads.push_back(thread(counterThread));

    for(int i = 0; i < 4; i++)
        threads[i].join();
    cout << counter << endl;
    return 0;
}
```

Everything repaired?

- If you repaired your code and it works correctly, you can try to code different task:
- Tool rental simulator
  - Rental shop offers – hammer, screwdriver, saw
  - Three handy guys:
    1) Borrow hammer, work, borrow screw driver, work, return all
    2) Borrow screw driver, work, borrow saw, work, return all
    3) Borrow saw, work, borrow hammer, work, return all
  - They are doing that repeatedly.
  - Work means in our case:
    ```
    for (int i = 0; i < 1000000; i++);
    ```



Handy guy = Thread

```
void* guyThread(void *args)
{
    argsStruct_t *tool = (argsStruct_t *)args;
    while(true) {
        {
                unique_lock<mutex> tool1_lock(*tool->tool1);
                cout << "Guy " <<tool->threadID << " borrowed "
                     << tool->tool1Name << "." << endl;
                work();
                unique_lock<mutex> tool2_lock(*tool->tool2);
                cout << "Guy " << tool->threadID << " borrowed "
                     << tool->tool2Name << "." << endl;
                work();
        }
        if ((*tool->counter) > COUNTER_TRESHOLD)
            break;
        {
                unique_lock<mutex> counter_lock(*tool->counterMutex);
                (*tool->counter)++;
        }
    }
    return 0;
}
```

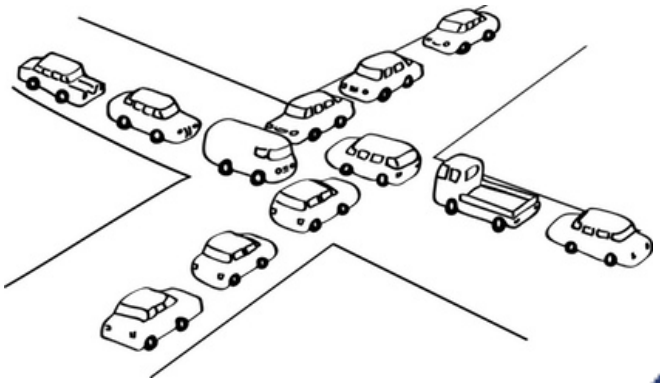
```
typedef struct argsStruct_t{
    mutex *counterMutex;
    int *counter;
    mutex *tool1;
    string tool1Name;
    mutex *tool2;
    string tool2Name;
    int threadID;
};

void work()
{
    for (int i = 0; i < WORK_ITERATIONS; i++);
}
```

Open the RentalShop – Main thread

```
int main() {
    vector<thread> threadGuys;
    vector<mutex> mutexes(MUTEXES_COUNT);
    int counter = 0;
    vector<argsStruct_t> threadTools(THREADS_COUNT);
    threadTools[0] = {&mutexes[COUNTER], &counter, &mutexes[HAMMER],
            "hammer", &mutexes[SCREW_DRIVER], "screw driver", 0};
    threadTools[1] = {&mutexes[COUNTER], &counter, &mutexes[SCREW_DRIVER],
            "screw driver", &mutexes[SAW], "saw", 1};
    threadTools[2] = {&mutexes[COUNTER], &counter, &mutexes[SAW],
            "saw", &mutexes[HAMMER], "hammer", 2};

    for(int i = 0; i < THREADS_COUNT; i++)
        threadGuys.push_back(thread(guyThread, (void *) &threadTools[i]));
    for(int i = 0; i < THREADS_COUNT; i++)
        threadGuys[i].join();
    return 0;
}
```

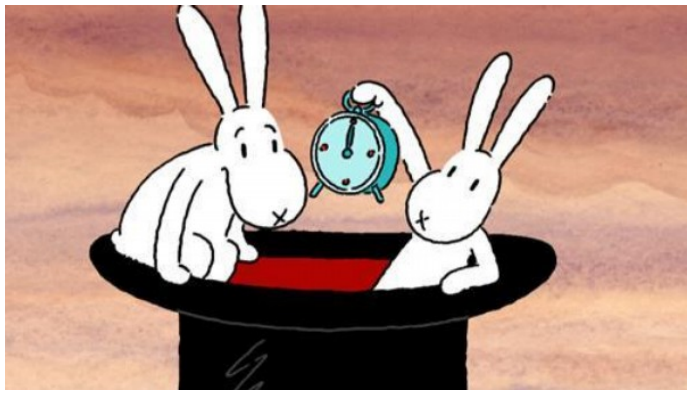
It is stuck somehow - Deadlock

- Guy 1 borrows a **hammer** and work
- Guy 2 borrows a **screw driver** and work
- Guy 3 borrows a **saw** and work
- Guy 1 needs a **screw driver** – waits for it
- Guy 2 needs a **saw** – waits for it
- Guy 3 needs a **hammer** – waits for it
- No one returns anything in this case.



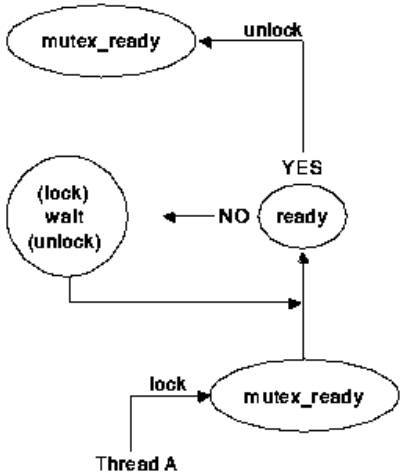
Condition variables

- Allows **signaling** among threads
- Threads can wait until some **event** occurs
- Another thread wake 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)
    while (!condition_ready())
        cv.wait(lk);
    compute_something();
}
```

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

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