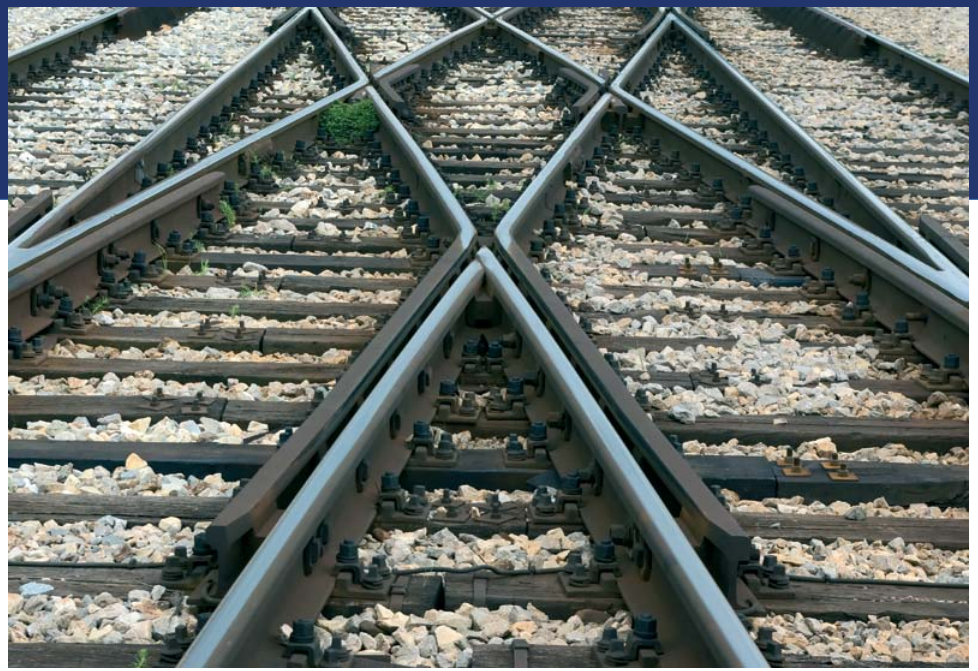
Parallel programming
C++11 threads
Part 2



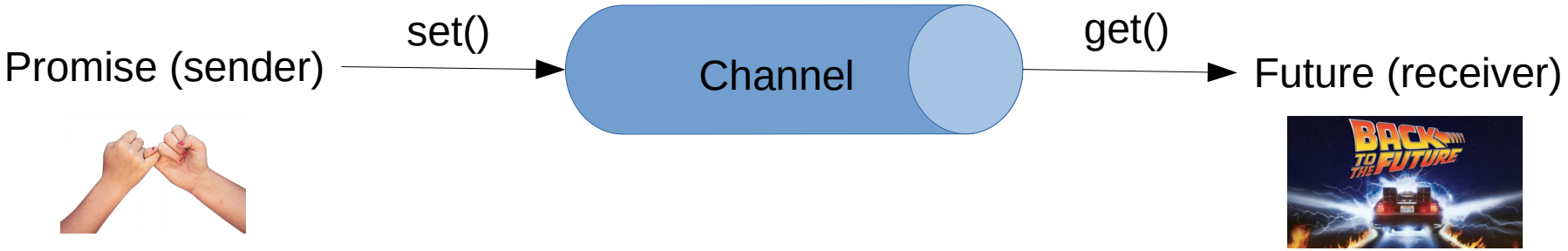
FAKULTA
ELEKTROTECHNICKÁ
ČVUT V PRAZE

Lab topics

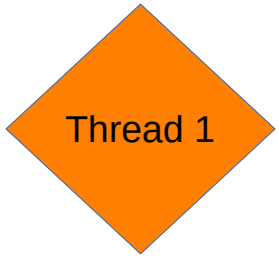
- **Future**, **promise** – synchronized access to values
  - e.g., returning values from threads
- Executing tasks by **async** object.
- **Atomic types** in C++11

Promise and future

- **promise** is used to store a value that is subsequently obtained by using the associated **future** object (synchronization point) in another thread.



Promise and future



Promise and future API

- `#include <future>`
  - Include the header with promise+future objects
- `promise<T> prom;`
  - Creates promise **prom**.
  - The promise is usually passed as reference to the callee thread (or moved).
- `future<T> fut = prom.get_future();`
  - Get future **fut** associated with promise **prom**.
- `prom.set_value(T());`
  - Sets **value** on promise **prom**.
- `T value = fut.get();`
  - Gets **value** from future **fut.**
  - Blocks the calling thread until the value is set in the associated promise.
  - Use **wait_for()** and **wait_until()** if you want to wait for the value only for some time

https://en.cppreference.com/w/cpp/thread/promise
https://en.cppreference.com/w/cpp/thread/future

Promise and future task

- Example
  - Task:
    - Create a worker thread that **gets two integers** from the main thread and pass **their multiplication** to the main thread
    - Use **promise** and **future** for passing/returning values to/from the worker thread
    - Try pass the input values to the worker thread
      - **Before** creating the worker thread
      - **After** creating the worker thread
      - Not at all :-)

Promise and future example

`lab_codes/src/PromiseAndFuture.cpp`

Async

- Using thread() is considered low-level, **async** is little bit more programmer friendly
  - Especially for returning values
  - Async functions **look like ordinary C++ functions** with return value and input arguments
- `#include <future>`
  - Include the header with **async** function
- `future<T> async(launch policy, Fn &&function, Args &&… args)`
  - Creates function that runs asynchronously. Apart from **policy,** the rest of the arguments are the same as for **thread()**
  - Returns future object containing the value returned by **function**
- async policy:
  - **launch::async** – creates a new thread for **function** (*eager* evaluation)
  - **launch::deferred** – function is started after its return value is requested from the future object (*lazy* evaluation). It is possible that new thread is not created, **function** may be run in the main thread.
    - If the value of future is not requested, **function** won't start
  - If not specified, the policy is left to the runtime implementation

Async task

- Example
  - Task:
    - Create a worker thread that **gets two integers** from the main thread and pass **their multiplication** to the main thread
    - Implement using **async**

Atomicity in C++11

- Atomic operations are **indivisible**, i.e. they behave like one instruction.
- Useful for a non-blocking synchronization between threads.
- Often lock-free for **integer types**.
- Atomic operation:
  - **load** value
  - **modify** value
  - **write** value

+= is required to be indivisible

```
int x = 0;
x += 5;
```

```
atomic<int> x(0);
x.fetch_add(5);
```

Atomicity in C++11

- https://en.cppreference.com/w/cpp/atomic/atomic
- `#include <atomic>`
  - Include the header with atomic class
- Basic operations with atomic class:
  - load, store
  - Operator++, ++operator, --operator, operator--
  - fetch_add, fetch_sub, fetch_and, fetch_or, fetch_xor...
  - bool compare_exchange_strong (T& expected, T desired)
    - Sets the contained value to be desired if the contained value equals the expected value
    - Returns true if expected is the same as the contained value
    - **Weak** version: may fail, useful for performance when used in loop

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!
    - Use **atomic** for synchronization among threads

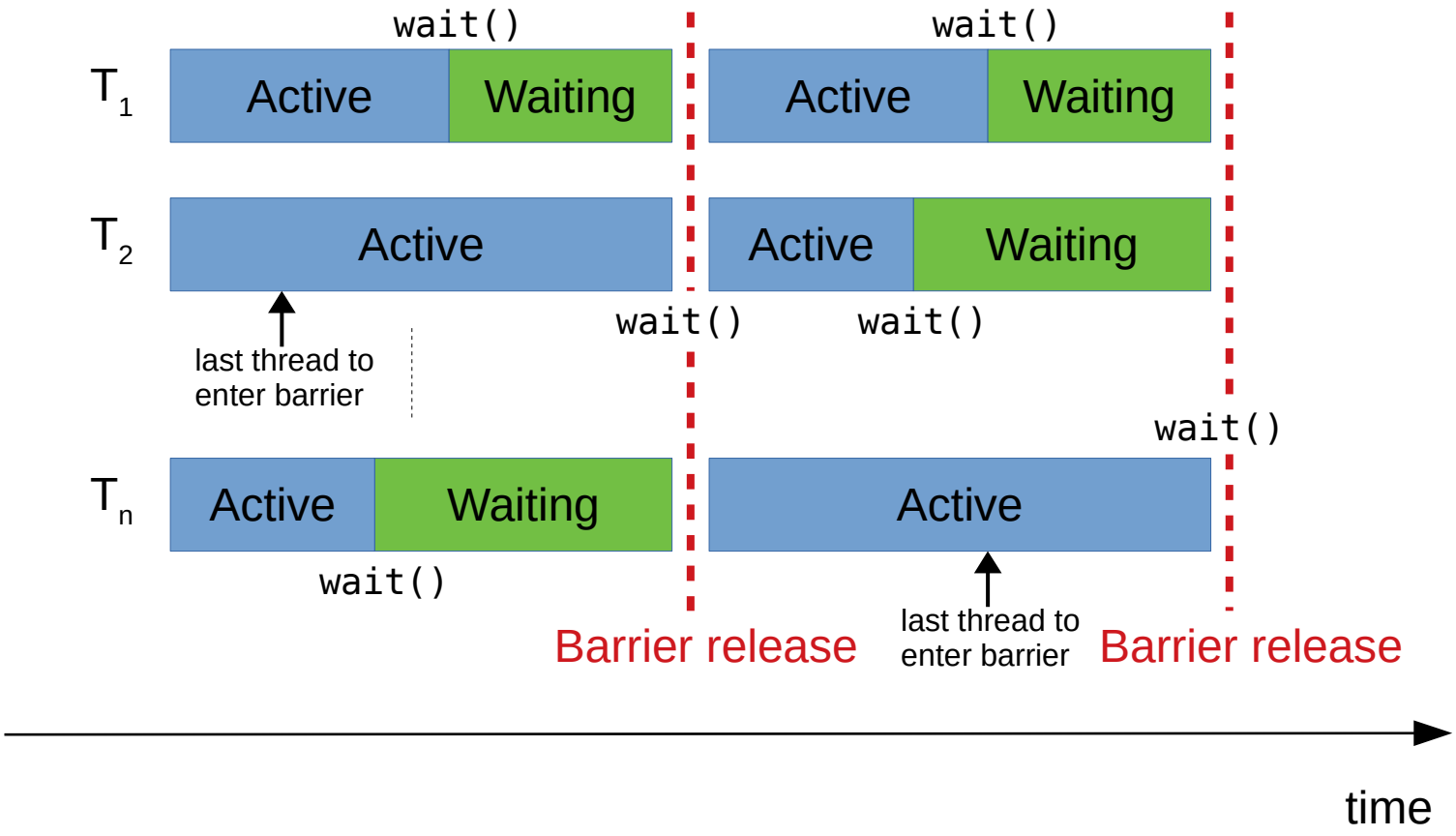
Atomic example

`lab_codes/src/AtomicCounter.cpp`

Main exercise – barrier

- API
  - Barrier(int numThreads);
  - Barrier.wait();
- synchronization of ***n*** threads
- threads wait on barrier until the last thread calls ***wait***, which releases the barrier
- The barrier must be reusable, i.e., it can be released multiple times

Main exercise – barrier



Main exercise – barrier

- **Hints:**
  - Use two atomic variables and busy waiting
  - One atomic variable counts the number of waiting threads
  - Second atomic variable counts the barrier releases (*phase counter*)
    - Last thread use this variable to signal the release of barrier to other threads
- **Advanced:** replace busy waiting with waiting on a conditional variable

Additional exercise - sorting

- Write a parallel program for odd-even sort
  - Split the input array into numThreads * 2 buckets

6,3,9,1,9,7,2,6,2,1,6,5,7,6,4,4,2,3,9,6,7,9,2,6

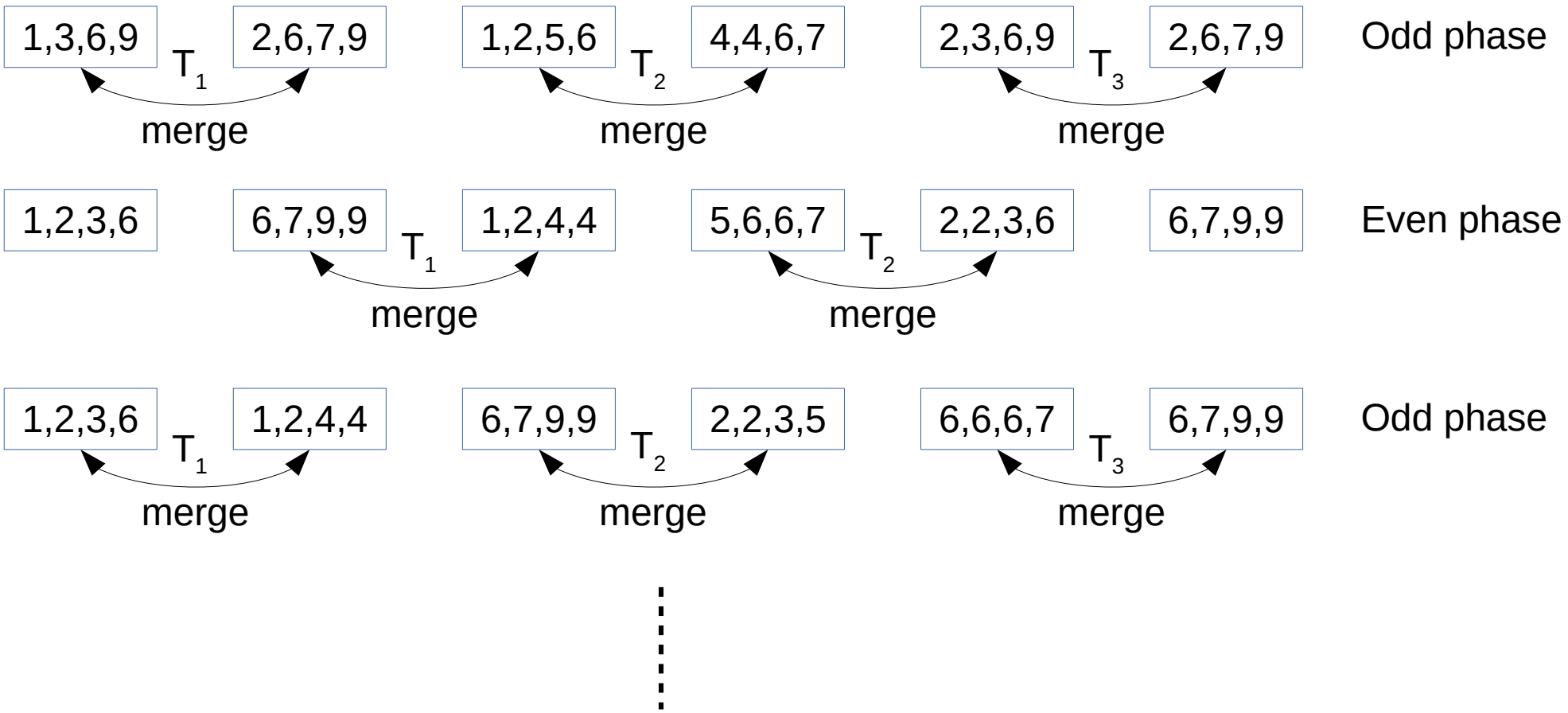
6,3,9,1	9,7,2,6	2,1,6,5	7,6,4,4	2,3,9,6	7,9,2,6

  - Initially, each thread sorts two buckets

1,3,6,9	2,6,7,9	1,2,5,6	4,4,6,7	2,3,6,9	2,6,7,9
T_1	T_1	T_2	T_2	T_3	T_3

  - Iteratively and in parallel merge adjacent buckets

Additional exercise - sorting



- Use barrier to synchronize threads between phases