# C++ - parallelization and synchronization

Jakub Yaghob

Martin Kruliš

# The problem

- Race conditions
  - Separate threads with shared state
  - Result of computation depends on OS scheduling

# Race conditions – simple demo

- Linked list
- Shared state

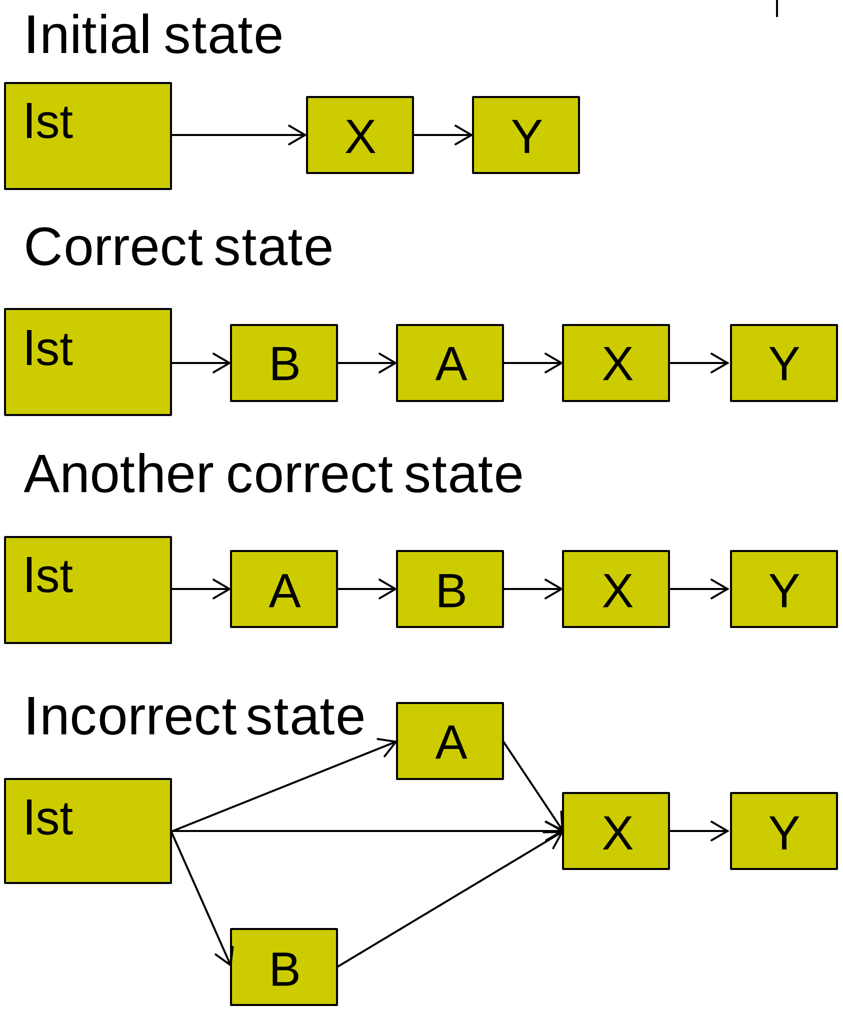
```
List lst;
```

- Thread A

```
lst.push_front(A);
```

- Thread B

```
lst.push_front(B);
```



# Race conditions – advanced demo

```
struct Counter {
  Counter():value(0) { }
  int value;
  void increment()
   { ++value; }
  void decrement()
   { --value; }
  int get()
   { return value; }
};
```

- Shared state

```
Counter c;
```

- Thread A

```
c.increment();
cout << c.get();
```

- Thread B

```
c.increment();
cout << c.get();
```

- Possible outputs

12, 21, 11

# C++ 11 features

- Atomic operations
- Low-level threads
- High-level futures
- Synchronization primitives
- Thread-local storage

# C++ 14 and C++17 features

- C++14 features
  - Shared timed mutex
- C++17 features
  - Parallel algorithms
  - Shared mutex

# C++ 11 – atomic operations

- Atomic operations
  - Header <atomics>
  - Allows creating portable lock-free algorithms and data structures
  - Memory ordering
  - Fences
  - Lock-free operations, algorithms, data-structures

# C++ 11 – atomic operations

- Memory ordering
  - `enum memory_order;`
    - `memory_order_seq_cst`
      - Sequentially consistent, most restrictive memory model
    - `memory_order_relaxed`
      - Totally relaxed memory model, allows best freedom for CPU and compiler optimizations
    - `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`
      - Additional barriers, weaker then sequentially consistent, stronger the relaxed

# C++ 11 – atomic operations

- Barriers
  - Acquire barrier
    - All loads read after acquire will perform after it (loads do not overtake acquire)
  - Release barrier
    - All stores written before release are committed before the release (writes do not delay)

# C++ 11 – atomic operations

- Easy way to make the demo safe

```
#include <atomic>

struct Counter {
  std::atomic<int> value;
  void increment(){ ++value; }
  void decrement(){ --value; }
  int get(){ return value.load(); }
};
```

# C++ 11 – atomic operations

- Template atomic
  - Defined for any type
    - Load, store, compare_exchange
  - Specialized for bool, all integral types, and pointers
    - Load, store, compare_exchange
    - Arithmetic and bitwise operations
      - fetch_add

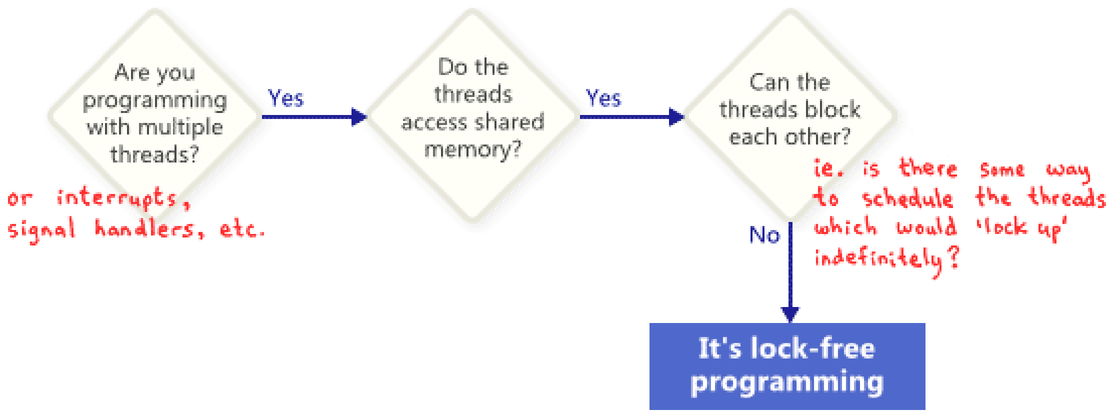
# C++ 11 – atomic operations

- Atomic flag
  - `atomic_flag` allows one-bit test and set
- Atomic operations for `shared_ptr`

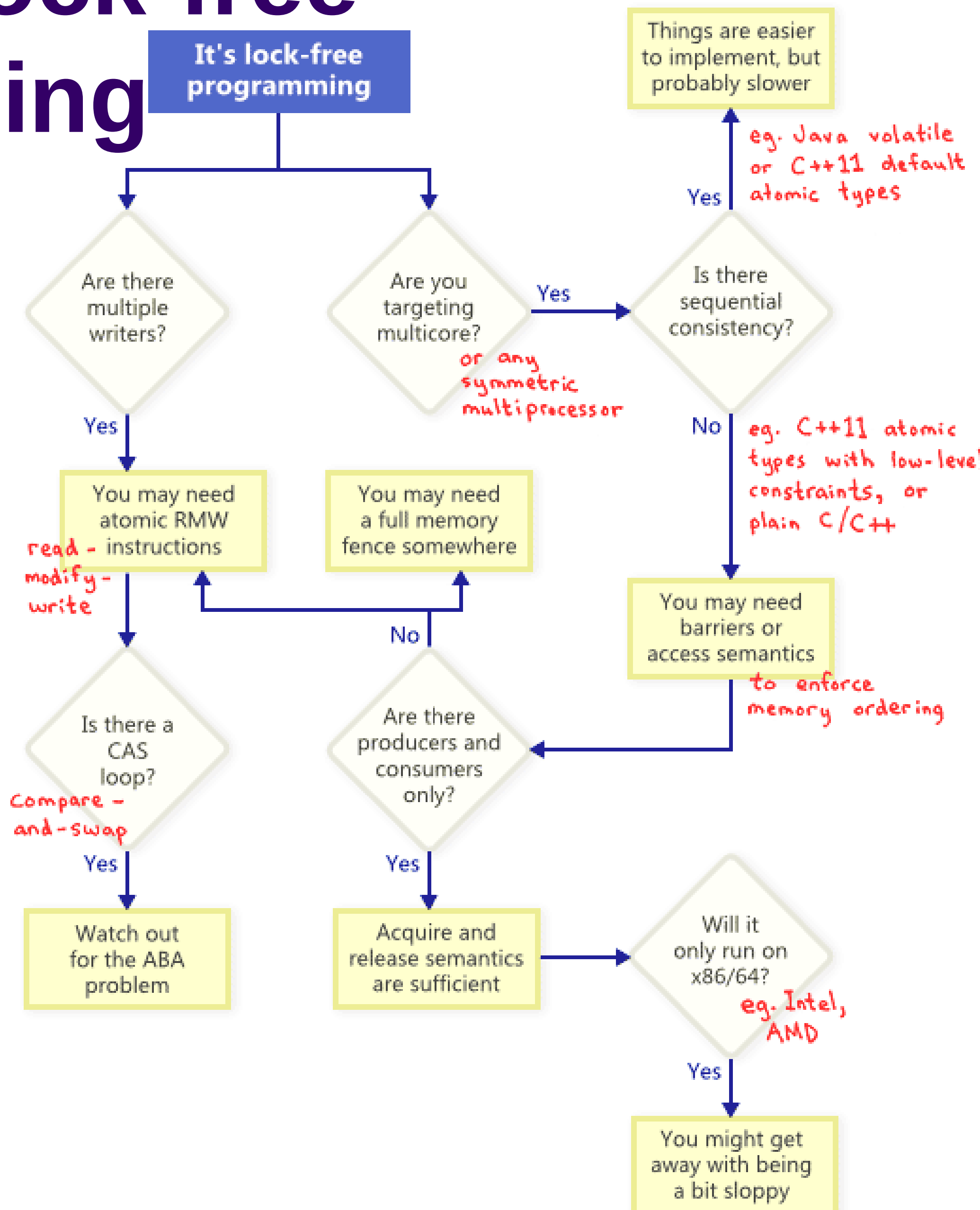
# C++ 11 – atomic operations

- Fences
  - Explicit memory barrier
  - `void atomic_thread_fence(memory_order order) noexcept;`
    - `memory_order_relaxed`
      - No effect
    - `memory_order_acquire`
      - An acquire fence
    - `memory_order_release`
      - A release fence
    - `memory_order_acq_rel`
      - Both an acquire and a release fence
    - `memory_order_seq_cst`
      - Sequentially consistent

# C++ 11 – lock-free programming



# C++ 11 – lock-free programming



# C++ 11 – threads

- Low-level threads
  - Header <thread>
  - `thread` class
  - Fork-join paradigm
  - Namespace `this_thread`

# C++ 11 – threads

- Class thread
  - Constructor
    - ```
      template <class F, class ...Args>
      explicit thread(F&& f, Args&&... args);
      ```
  - Destructor
    - If joinable() then `terminate()`
  - `bool joinable() const noexcept;`
  - `void join();`
    - Blocks, until the thread *this has completed
  - `void detach();`
  - `id get_id() const noexcept;`
  - `static unsigned hardware_concurrency();`
      ```

# C++ 11 – threads

- Namespace `this_thread`
  - `thread::id get_id() noexcept;`
    - Unique ID of the current thread
  - `void yield() noexcept;`
    - Opportunity to reschedule
  - `sleep_for`, `sleep_until`
    - Blocks the thread for relative/absolute timeout

# C++ 11 – threads

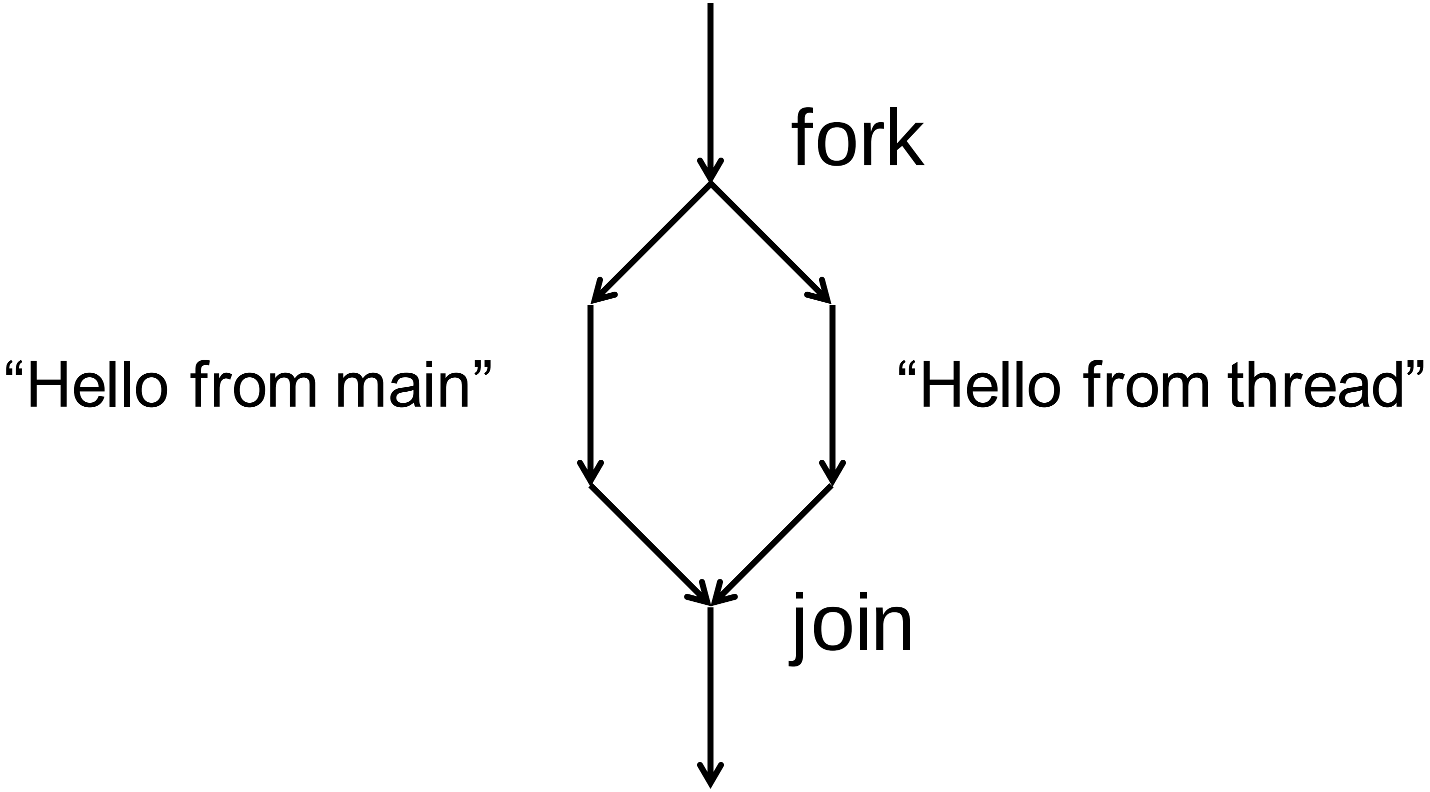
- Demo

```
#include <iostream>
#include <thread>

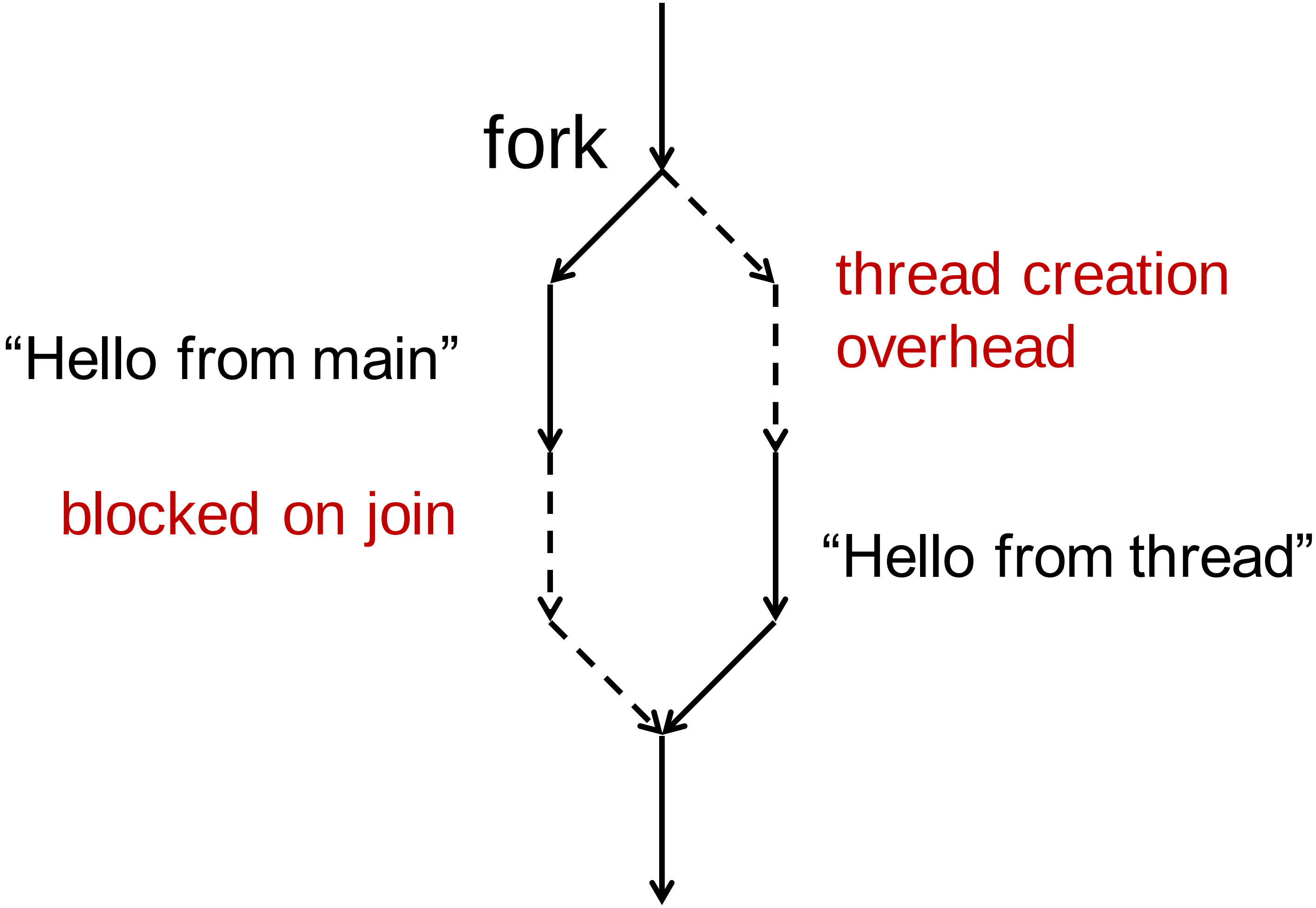
void thread_fn() { std::cout << “Hello from thread” <<
   std::endl; }

int main(int argc, char **argv) {
  std::thread thr(&thread_fn);
  std::cout << “Hello from main” << std::endl;
  thr.join();
  return 0;
}
```

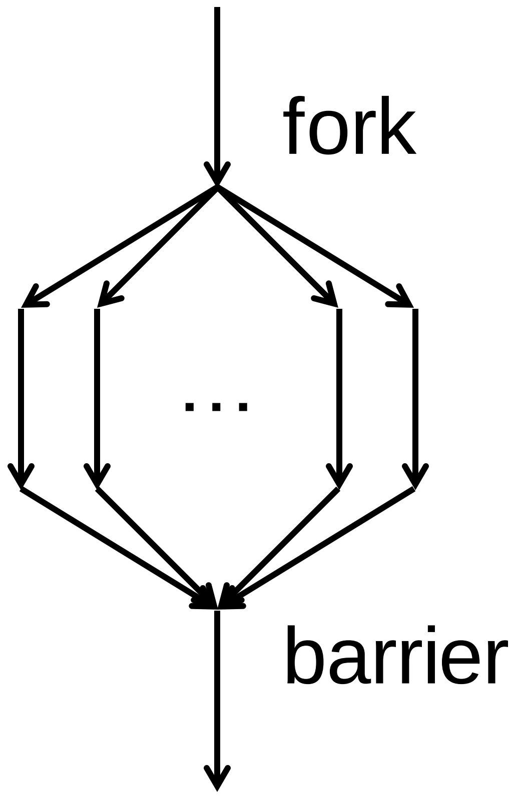
# C++ 11 – threads



# C++ 11 – threads


fork
thread creation
overhead
“Hello from main”
blocked on join
“Hello from thread”

# C++ 11 – threads



# C++ 11 – threads

- Demo

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

int main(int argc, char **argv) {
  std::vector<std::thread> workers;
  for(int i=0;i<10;++i)
    workers.push_back(std::thread([i]() {
      std::cout << “Hello from thread “ << i << std::endl;
    }));
  std::cout << “Hello from main” << std::endl;
  for(auto &t : workers)
    t.join();
  return 0;
}
```

# C++ 11 – threads

- Passing arguments to threads
  - By value
    - Safe, but you MUST make deep copy
  - By move (rvalue reference)
    - Safe, as long as strict (deep) adherence to move semantics
  - By const reference
    - Safe, as long as object is guaranteed deep-immutable
  - By non-const reference
    - Safe, as long as the object is monitor

# C++ 11 – futures

- Futures
  - Header <future>
  - High-level asynchronous execution
  - Future
  - Promise
  - Async
  - Error handling

# C++ 11 – futures

- Shared state
  - Consist of
    - Some state information and some (possibly not yet evaluated) result, which can be a (possibly void) value or an exception
  - Asynchronous return object
    - Object, that reads results from an shared state
  - Waiting function
    - Potentially blocks to wait for the shared state to be made ready
  - Asynchronous provider
    - Object that provides a result to a shared state

# C++ 11 – futures

- Future
  - `std::future<T>`
  - Future value of type T
  - Retrieve value via `get()`
    - Waits until the shared state is ready
  - `wait()`, `wait_for()`, `wait_until()`
  - `std::shared_future<T>`
    - Value can be read by more then one thread

# C++ 11 – futures

- Async
  - `std::async`
  - Higher-level convenience utility
  - Launches a function potentially in a new thread
- Async usage

```
int foo(double, char, bool);
auto fut = std::async(foo, 1.5, 'x', false);
auto res = fut.get();
```

# C++ 11 – futures

- Packaged task
  - `std::packaged_task`
  - How to implement async with more control
  - Wraps a function and provides a future for the function result value, but the object itself is callable

# C++ 11 – futures

- Packaged task usage

```
std::packaged_task<int(double, char, bool)>
  tsk(foo);
auto fut = tsk.get_future();
std::thread thr(std::move(tsk), 1.5, 'x', false);
auto res = fut.get();
```

# C++ 11 – futures

- Promise
  - `std::promise<T>`
  - Lowest-level
  - Steps
    - Calling thread makes a promise
    - Calling thread obtains a future from the promise
    - The promise, along with function arguments, are moved into a separate thread
    - The new thread executes the function and fulfills the promise
    - The original thread retrieves the result

# C++ 11 – futures

- Promise usage
  - Thread A

```
std::promise<int> prm;
auto fut = prm.get_future();
std::thread thr(thr_fnc, std::move(prm));
auto res = fut.get();
```

  - Thread B

```
void thr_fnc(std::promise<int> &&prm) {
  prm.set_value(123);
}
```

# C++ 11 – futures

- Constraints
  - A default-constructed promise is inactive
    - Can die without consequence
  - A promise becomes active, when a future is obtained via `get_future()`
    - Only one future may be obtained
  - A promise must either be satisfied via `set_value()`, or have an exception set via `set_exception()`
    - A satisfied promise can die without consequence
    - `get()` becomes available on the future
    - A promise with an exception will raise the stored exception upon call of `get()` on the future
    - A promise with neither value nor exception will raise “broken promise” exception

# C++ 11 – futures

- Exceptions
  - All exceptions of type `std::future_error`
    - Has error code with enum type `std::future_errc`

- inactive promise

```
std::promise<int> pr;
// fine, no problem
```

- active promise, unused

```
std::promise<int> pr;
auto fut = pr.get_future();
// fine, no problem
// fut.get() blocks indefinitely
```

- too many futures

```
std::promise<int> pr;
auto fut1 = pr.get_future();
auto fut2 = pr.get_future();
// error "Future already
retrieved"
```

# C++ 11 – futures

- satisfied promise

```
std::promise<int> pr;
auto fut = pr.get_future();
{ std::promise<int> pr2(std::move(pr));
  pr2.set_value(10);
}
auto r = fut.get();
// fine, return 10
```

- too much satisfaction

```
std::promise<int> pr;
auto fut = pr.get_future();
{ std::promise<int> pr2(std::move(pr));
  pr2.set_value(10);
  pr2.set_value(11);
// error "Promise already satisfied"
}
auto r = fut.get();
```

# C++ 11 – futures

- exception

```
std::promise<int> pr;
auto fut = pr.get_future();
{ std::promise<int> pr2(std::move(pr));
  pr2.set_exception(
    std::make_exception_ptr(
      std::runtime_error(“bububu”)));
}
auto r = fut.get();
// throws the runtime_error
```

# C++ 11 – futures

- broken promise

```
std::promise<int> pr;
auto fut = pr.get_future();
{ std::promise<int> pr2(std::move(pr));
  // error “Broken promise”
}
auto r = fut.get();
```

# C++ 11 – synchronization primitives

- Synchronization primitives
  - Mutual exclusion
    - Header <mutex>
  - Condition variables
    - Header <condition_variable>

# C++ 11 – mutex

- Mutex
  - A synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads
  - `mutex` offers exclusive, non-recursive ownership semantics
    - A calling thread *owns* a `mutex` from the time that it successfully calls either `lock` or `try_lock` until it calls `unlock`
    - When a thread owns a `mutex`, all other threads will block (for calls to `lock`) or receive a false return value (for `try_lock`) if they attempt to claim ownership of the `mutex`
    - A calling thread must not own the `mutex` prior to calling `lock` or `try_lock`
  - The behavior of a program is undefined if a `mutex` is destroyed while still owned by some thread

# C++ 11 – mutex example

- Shared state

```
List lst;
std::mutex mtx;
```

- Thread A

```
mtx.lock();
lst.push_front(A);
mtx.unlock();
```

- Thread B

```
mtx.lock();
lst.push_front(B);
mtx.unlock();
```

# C++ 11 – mutex variants

- Other `mutex` variants
  - `timed_mutex`
    - In addition, `timed_mutex` provides the ability to attempt to claim ownership of a `timed_mutex` with a timeout via the `try_lock_for` and `try_lock_until`
  - `recursive_mutex`
    - exclusive, recursive ownership semantics
      - A calling thread *owns* a `recursive_mutex` for a period of time that starts when it successfully calls either `lock` or `try_lock`. During this period, the thread may make additional calls to `lock` or `try_lock`. The period of ownership ends when the thread makes a matching number of calls to `unlock`
      - When a thread owns a `recursive_mutex`, all other threads will block (for calls to `lock`) or receive a false return value (for `try_lock`) if they attempt to claim ownership of the `recursive_mutex`
      - The maximum number of times that a `recursive_mutex` may be locked is unspecified, but after that number is reached, calls to `lock` will throw `std::system_error` and calls to `try_lock` will return false
  - `recursive_timed_mutex`
    - Combination

# C++ 11 – mutex wrappers

- `std::unique_lock`
  - Lock class with more features
    - Timed wait, deferred lock
- `std::lock_guard`
  - Scope based lock (RAII)
  - Linked list demo, code for one thread

```
{
std::lock_guard<std::mutex> lk(mtx);
lst.push_front(X);
}
```

# C++ 14 – mutex variants and wrappers

- Other mutex variants in C++ 14
  - **`std::shared_timed_mutex`**
    - Multiple threads can make shared lock using lock_shared()
- Additional wrapper
  - **`std::shared_lock`**
    - Calls lock_shared for the given mutex

# C++ 17 – mutex variants, wrappers, and others

- Another mutex variant
  - `std::shared_mutex`
- Variadic wrapper
  - `template <typename … MutexTypes> class scoped_lock;`
    - Multiple locks at once
- Interference size
  - `std::size_t hardware_destructive_interference_size;`
  - Size of a cache line

# C++ 11 – locking algorithms

- **`std::lock`**
  - locks specified mutexes, blocks if any are unavailable
- **`std::try_lock`**
  - attempts to obtain ownership of mutexes via repeated calls to **`try_lock`**

```
// don't actually take the locks yet
std::unique_lock<std::mutex> lock1(mtx1, std::defer_lock);
std::unique_lock<std::mutex> lock2(mtx2, std::defer_lock);
// lock both unique_locks without deadlock
std::lock(lock1, lock2);
```

# C++ 11 – call once

- `std::once_flag`
  - Helper object for `std::call_once`
- `std::call_once`
  - invokes a function only once even if called from multiple threads

```
std::once_flag flag;
void do_once() {
   std::call_once(flag, [](){ do something only once }); }
std::thread t1(do_once);
std::thread t2(do_once);
```

# C++ 11 – condition variable

- `std::condition_variable`
  - Can be used to block a thread, or multiple threads at the same time, until
    - a notification is received from another thread
    - a timeout expires, or
    - a spurious wakeup occurs
      - Appears to be signaled, although the condition is not valid
      - Verify the condition after the thread has finished waiting
  - Works with `std::unique_lock`
  - `wait` atomically manipulates mutex, `notify` does nothing

# C++11 – condition variable example

```
std::mutex m;
std::condition_variable cond_var;
bool done = false; bool notified = false;
```

- Producer

```
for () {
  // produce something
  { std::lock_guard<std::mutex>
   lock(m);
    queue.push(item);
    notified = true; }
  cond_var.notify_one();
}
std::lock_guard<std::mutex> lock(m);
notified = true;
done = true;
cond_var.notify_one();
```

- Consumer

```
std::unique_lock<std::mutex> lock(m);
while(!done) {
  while (!notified) {
    // loop to avoid spurious wakeups
    cond_var.wait(lock);
  }
  while(!produced_nums.empty()) {
    // consume
    produced_nums.pop();
  }
  notified = false;
}
```

# C++ 11 – thread-local storage

- Thread-local storage
  - Added a new storage-class
  - Use keyword `thread_local`
    - Must be present in all declarations of a variable
    - Only for namespace or block scope variables and to the names of static data members
      - For block scope variables `static` is implied
  - Storage of a variable lasts for the duration of a thread in which it is created

# C++ extensions – parallelism

- Parallelism
  - TS v1 adopted in C++ 17, TS v2 finished
    - In headers <algorithm>, <numeric>
  - Parallel algorithms
    - Execution policy in <execution>
      - `seq` – execute sequentially
      - `par` – execute in parallel on multiple threads
      - `par_unseq` – execute in parallel on multiple threads, interleave individual iterations within a single thread, no locks
      - `unseq` – (C++20) execute in single thread+vectorized
    - `for_each`
    - `reduce`, `scan`, `transform_reduce`, `transform_scan`
      - Inclusive scan – like `partial_sum`, includes i-th input element in the i-th sum
      - Exclusive scan – like `partial_sum`, excludes i-th input element from the i-th sum
    - No exceptions should be thrown
      - Terminate

# C++ extensions – parallelism v1

- Parallel algorithms
  - Not all algorithms have parallel version
  - adjacent_difference, adjacent_find, all_of, any_of, copy, copy_if, copy_n, count, count_if, equal, exclusive_scan, fill, fill_n, find, find_end, find_first_of, find_if, find_if_not, for_each, for_each_n, generate, generate_n, includes, inclusive_scan, inner_product, inplace_merge, is_heap, is_heap_until, is_partitioned, is_sorted, is_sorted_until, lexicographical_compare, max_element, merge, min_element, minmax_element, mismatch, move, none_of, nth_element, partial_sort, partial_sort_copy, partition, partition_copy, reduce, remove, remove_copy, remove_copy_if, remove_if, replace, replace_copy, replace_copy_if, replace_if, reverse, reverse_copy, rotate, rotate_copy, search, search_n, set_difference, set_intersection, set_symmetric_difference, set_union, sort, stable_partition, stable_sort, swap_ranges, transform, transform_exclusive_scan, transform_inclusive_scan, transform_reduce, uninitialized_copy, uninitialized_copy_n, uninitialized_fill, uninitialized_fill_n, unique, unique_copy

# C++ extensions – parallelism v2

- Task block
  - Support for fork-join paradigm
  - Spawn other task_blocks and wait for their completion
  - Exceptions
    - Each task_block has an exception list
    - Exceptions from forked task_blocks are stored in the exception list
    - Exceptions are invoked when task_block finishes

# C++ extension – executors

- Executors
  - Now separate TS, maybe finished in C++23 timeframe
- Executor
  - Controls how a task (=function) is executed
  - Direction
    - One-way execution
      - Does not return a result
    - Two-way execution
      - Returns future
    - Then
      - Execution agent begins execution after a given future becomes ready, returns future
  - Cardinality
    - Single
      - One execution agent
    - Bulk executions
      - Group of execution agents
      - Agents return a factory
- Thread pool
  - Controls where the task is executed

# C++ extensions – concurrency

- Concurrency
  - TS published, depends on executors TS
  - Improvements to future
    - future<T2> then(F &&f)
      - Execute asynchronously a function f when the future is ready
  - Latches
    - Thread coordination mechanism
      - Block one or more threads until an operation is completed
    - Single use
  - Barriers
    - Thread coordination mechanism
    - Reusable
    - Multiple barrier types
      - barrier
      - flex_barrier – calls a function in a completion phase

# C++ extension – transactional memory

- TS v1 finished
- Transactional memory
  - Added several keywords for statements and declarations
  - **synchronized** *compound-statement*
    - Synchronized with other synchronized blocks
    - One global lock for all synchronized blocks
  - Atomic blocks
    - Execute atomically and not concurrently with synchronized blocks
    - Can execute concurrently with other atomic blocks if no conflicts
    - Differs in behavior with exceptions
  - **atomic_noexcept** *compound-statement*
    - Escaping exception causes undefined behavior
  - **atomic_cancel** *compound-statement*
    - Escaping exception rolls back the transaction, but must be transaction safe
    - Functions can be declared **transaction_safe**
  - **atomic_commit** *compound-statement*
    - Escaping exception commits the transaction