# Multi-Threading

# Multi-Threading in C++

In C++ it is allowed to run multiple threads simultaneously that use the same memory.

- Multiple threads may *read* from the same memory location
- All other accesses (i.e. read-write, write-read, write-write) are called *conflicts*
- Conflicting operations are only allowed when threads are *synchronized*
- This can be done with *mutexes* or *atomic operations*
- Unsynchronized accesses (also called *data races*), deadlocks, and other potential issues when using threads are undefined behavior!

# Threads Library (1)

The header <thread> defines the class std::thread that can be used to start new threads.

- Using this class is the best way to use threads platform-independently
- May require additional compiler flags: -pthread for gcc and clang

```
void foo(int a, int b);
// Starts a thread that calls foo(123, 456)
std::thread t1(foo, 123, 456);
// Also works with lambdas
std::thread t2([] { foo(123, 456); });
// Creates an object that does not refer to a thread
std::thread t3;
```

# Threads Library (2)

The member function `join()` can be used to wait for a thread to finish.

- `join()` must be called exactly once for each thread
- When the destructor of an `std::thread` is called, the program is terminated if it has an associated thread that was not joined

```
std::thread t1([] { std::cout << "Hi\n"; });
t1.join();
{
    std::thread t2([] {});
}
// Program terminated because t2.join() was not called
```

# Threads Library (3)

`std::thread`s are not copyable, but movable, so they can be used in containers. Moving an `std::thread` transfers all resources associated with the running thread. Only the moved-to thread can be joined.

```
std::thread t1([] { std::cout << "Hi\n"; });
std::thread t2 = std::move(t1); // t1 is now empty
t2.join(); // OK, thread originally started in t1 is joined

std::vector<std::thread> threadPool;
for (int i = 1; i <= 9; ++i) {
    threadPool.emplace_back([i] { safe_print(i); });
}
// Digits 1 to 9 are printed (unordered)
for (auto& t : threadPool) {
    t.join();
}
```

# Other Functions of the Thread Library

The thread library also contains other useful functions that are closely related to starting and stopping threads:

- `std::this_thread::sleep_for()`: Stop the current thread for a given amount of time
- `std::this_thread::sleep_until()`: Stop the current thread until a given point in time
- `std::this_thread::yield()`: Let the operating system schedule another thread
- `std::this_thread::get_id()`: Get the (operating-system-specific) id of the current thread

# Mutual Exclusion

When working with threads, *mutual exclusion* is a central concept to synchronize threads.
The standard library defines several useful classes for this in `<mutex>` and `<shared_mutex>`:

- `std::mutex` (mutual exclusion)
- `std::recursive_mutex` (recursive mutual exclusion)
- `std::shared_mutex` (mutual exclusion with shared locks)
- `std::unique_lock` (RAII wrapper for `std::mutex`)
- `std::shared_lock` (RAII wrapper for `std::shared_mutex`)

Note: Mutexes are usually inefficient as they are used very coarse-grained and sometimes require communication with the operating system.

# Mutexes

A mutex is the most basic synchronization primitive which can be locked and unlocked by exactly one thread at a time.

- `std::mutex` has the member functions `lock()` and `unlock()` that lock and unlock the mutex
- `try_lock()` is a member function that tries to lock the mutex and returns `true` if it was successful
- All three functions may be called simultaneously by different threads
- For each call to `lock()` the same thread must call `unlock()` exactly once

```
std::mutex printMutex;
void safe_print(int i) {
    printMutex.lock();
    std::cout << i;
    printMutex.unlock();
}
```

# Recursive Mutexes

Recursive mutexes are regular mutexes that additionally allow a thread that currently holds the mutex to lock it again.

- Implemented in the class `std::recursive_mutex`
- Has the same member functions as `std::mutex`
- `unlock()` must still be called once for each `lock()`
- Useful for functions that call each other and use the same mutex

```
std::recursive_mutex m;
void foo() {
    m.lock();
    std::cout << "foo\n";
    m.unlock();
}
void bar() {
    m.lock();
    std::cout << "bar\n";
    foo(); // This will not deadlock
    m.unlock();
}
```

# Shared Mutexes (1)

A shared mutex is a mutex that differentiates between *shared* and *exclusive* locks.

- Implemented in the class `std::shared_mutex`
- A shared mutex can either be locked exclusively by one thread or have multiple shared locks
- The member functions `lock()` and `unlock()` are exclusive
- The member functions `lock_shared()` and `unlock_shared()` are shared
- The member functions `try_lock()` and `try_lock_shared()` try to get an exclusive or shared lock and return `true` on success

# Shared Mutexes (2)

- Shared mutexes are mostly used to implement read/write-locks
- Readers use shared locks, writers use exclusive locks

```
int value = 0; std::shared_mutex m;
std::vector<std::thread> threadPool;
// Add readers
for (int i = 0; i < 5; ++i)
    threadPool.emplace_back([&] {
        m.lock_shared();
        safe_print(value);
        m.unlock_shared();
    }
// Add writers
for (int i = 0; i < 5; ++i)
    threadPool.emplace_back([&] {
        m.lock();
        ++value;
        m.unlock();
    }
```

# Working with Mutexes

Mutexes have several requirements on how they must be used:

- For each call to `lock()`, `unlock()` must be called exactly once
- `unlock()` must only be called by the thread that called `lock()`
- The above also holds for `unlock_shared()` and `lock_shared()`
- A thread *A* should not wait for a mutex from thread *B* to be unlocked if *B* needs to lock a mutex that *A* is currently holding (i.e. avoid *deadlocks*)

Note the following when using mutexes to make data structures thread-safe:

- The member functions `lock()` and `unlock()` are non-const
- If const member functions of the data structure should also use the mutex, it should be `mutable`
- If a member function that locks the mutex calls other member functions, this can lead to deadlocks
- `recursive_mutex` can be used to avoid this

# Mutex RAII Wrappers (1)

Mutexes can be thought of resources that must be acquired and freed with `lock()` and `unlock()`.

- The RAII pattern should be used
- `std::unique_lock` is an RAII wrapper for Mutexes that calls `lock()` in its constructor and `unlock()` in its destructor
- `std::unique_lock` is movable to "transfer ownership" of the locked mutex
- It also has the member functions `lock()` and `unlock()` to manually control the mutex

```
std::mutex m;
int i = 0;
std::thread t{[&] {
    std::unique_lock l{m}; // m.lock() is called
    ++i;
    // m.unlock() is called
}};
```

# Mutex RAII Wrappers (2)

- Shared mutexes additionally need an RAII wrapper that calls `lock_shared()` and `unlock_shared()`
- For this `std::shared_lock` can be used
- Note: `std::shared_lock` is only movable and not copyable (unlike `std::shared_ptr`)

```
std::shared_mutex m;
int i = 0;
std::thread t{[&] {
    std::shared_lock l{m}; // m.lock_shared() is called
    std::cout << i;
    // m.unlock_shared() is called
}};
```

# Avoiding Deadlocks (1)

- Deadlocks can occur when using multiple mutexes
- In particular, when two different threads each succeed to lock a subset of the mutexes and then try to lock the rest
- Can be avoided by always locking mutexes in a consistent order

```
std::mutex m1, m2, m3;
void threadA() {
    std::unique_lock l1{m1}, l2{m2}, l3{m3};
}
void threadB() {
    std::unique_lock l3{m3}, l2{m2}, l1{m1};
    // DANGER: order not consistent with threadA()
}
```

Concurrent calls to `threadA()` and `threadB()` can lead to deadlocks. E.g., A could get the locks for `m1` and `m2` while B gets a lock for `m3`.

# Avoiding Deadlocks (2)

- Sometimes, it is not possible to always guarantee a consistent order
- The function `std::lock()` takes any number of mutexes and locks them all by using a deadlock-avoiding algorithm
- `std::scoped_lock` is an RAII wrapper for `std::lock()`

```
std::mutex m1, m2, m3;
void threadA() {
    std::scoped_lock l{m1, m2, m3};
}
void threadB() {
    std::scoped_lock l{m3, m2, m1};
}
```

Here, calling `threadA()` and `threadB()` concurrently will not lead to deadlocks.
Note: This should only be used if there is no other way as it is generally very inefficient!

# Condition Variables (1)

A condition variable is a synchronization primitive that allows multiple threads to wait until an (arbitrary) condition becomes true.

- A condition variable uses a mutex to synchronize threads
- Threads can *wait* on or *notify* the condition variable
- When a thread waits on the condition variable, it blocks until another thread notifies it
- If a thread waited on the condition variable and is notified, it holds the mutex
- A notified thread must check the condition explicitly because *spurious wake-ups* can occur

# Condition Variables (2)

The standard library defines the class `std::condition_variable` in the header `<condition_variable>` which has the following member functions:

- `wait()`: Takes a reference to a `std::unique_lock` that must be locked by the caller as an argument, unlocks the mutex and waits for the condition variable
- `notify_one()`: Notify a single waiting thread, mutex does not need to be held by the caller
- `notify_all()`: Notify all waiting threads, mutex does not need to be held by the caller

# Condition Variables Example

One use case for condition variables are worker queues: Tasks are inserted into a queue and then worker threads are notified to do the task.

```
std::mutex m;
std::condition_variable cv;
std::vector<int> taskQueue;

void pushWork(int task) {
  {
    std::unique_lock l{m};
    taskQueue.push_back(task);
  }
  cv.notify_one();
}
```

```
void workerThread() {
  std::unique_lock l{m};
  while (true) {
    if (!taskQueue.empty()) {
      int task = taskQueue.back();
      taskQueue.pop_back();
      l.unlock();
      // [...] do actual work here
      l.lock();
    }
    cv.wait(l);
  }
}
```

# Atomic Operations

Threads can also be synchronized with *atomic operations* that are usually much more efficient than mutexes.

- Atomicity means that an operation is executed as one unit, no other operation on the same object can be executed at the same time
- Consider a function `void atomic_add(int& p, int i)` that represents an atomic operation that adds `i` to the integer `p` and the following program:

```
int a = 10;
void threadA() { atomic_add(a, 1); }
void threadB() { atomic_add(a, 2); }
```

- When `threadA()` and `threadB()` are called concurrently, `a` is guaranteed to always be equal to 13 at the end
- This is usually implemented by using special CPU instructions, no operating system is needed!
- Note: Only the atomicity of *single* operations are guaranteed to be atomic

# Atomic Operations Library

- All classes and functions related to atomic operations can be found in the `<atomic>` header
- `std::atomic<T>` is a class that represents an atomic version of the type `T`
- This class has several member functions that implement atomic operations:
  - `T load()`: Loads the value (allows concurrent writes)
  - `void store(T desired)`: Stores `desired` in the object
  - `T exchange(T desired)`: Stores `desired` in the object and returns the old value
  - `bool compare_exchange_weak(...)` and `bool compare_exchange_strong(...)`: Performs a *compare-and-swap (CAS)* operation
- If `T` is a integral type, the following operations also exist:
  - `T fetch_add(T arg)`: Adds `arg` to the value and returns the old value
  - `T fetch_sub(T arg)`: Same for subtraction
  - `T fetch_and(T arg)`: Same for bitwise and
  - `T fetch_or(T arg)`: Same for bitwise or
  - `T fetch_xor(T arg)`: Same for bitwise xor

# Modification Order

All modifications of a single atomic object are totally ordered in the so-called *modification order*.

- The modification order is consistent between threads (i.e. all threads see the same order)
- The modification order is only total for individual atomic objects

```
std::atomic<int> i = 0;
void workerThread() {
    i.fetch_add(1); // (A)
    i.fetch_sub(1); // (B)
}
void readerThread() {
    int iLocal = i.load();
    assert(iLocal == 0 || iLocal == 1); // This is always true
}
```

Because the memory order is consistent between threads, the reader thread will never see a execution order of `(A)`, `(B)`, `(B)`, `(A)`, for example.

# Memory Order (1)

- The modification order is only total for individual atomic objects
- How the modification orders of different atomic and non-atomic operations are interleaved is determined by the *memory order*
- There exist several memory orders which can be chosen, the most important ones are: *relaxed memory order* and *sequentially consistent memory order*

Relaxed memory order:

- Weakest (but usually most efficient) memory order
- All other atomic and non-atomic operations may be reordered with respect to the relaxed atomic operation

Sequentially consistent memory order:

- Strictest (but usually least efficient) memory order
- All sequentially consistent operations will appear in the same order in all threads
- If a thread sees the result of a sequentially consistent operation of another thread, it is also guaranteed to see all non-atomic writes

# Memory Order (2)

The following values can be used as function arguments to the atomic operations to specify the memory order:

- `std::memory_order_relaxed`: Relaxed memory order
- `std::memory_order_seq_cst`: Sequentially consistent memory order

```
std::atomic<int> i;
// Sequentially consistent memory order is the default
i.store(123);
// Relaxed atomic addition
i.fetch_add(321, std::memory_order_relaxed);
// Explicitly specify sequential consistency
i.store(1, std::memory_order_seq_cst);
```

# Effects of Memory Orders (1)

Relaxed memory order only guarantees the atomicity of the operation, nothing more. This can lead to surprising results:

```
std::atomic<int> a = 0;
std::atomic<int> b = 0;
void threadA() {
    int a_local = a.load(std::memory_order_relaxed); // A1
    b.store(a_local, std::memory_order_relaxed);     // A2
    safe_print(a_local);
}
void threadB() {
    int b_local = b.load(std::memory_order_relaxed); // B1
    a.store(5, std::memory_order_relaxed);           // B2
    safe_print(b_local);
}
```

This program is allowed to print “55”! This is because the operations in lines B1 and B2 are allowed to be reordered by the CPU.

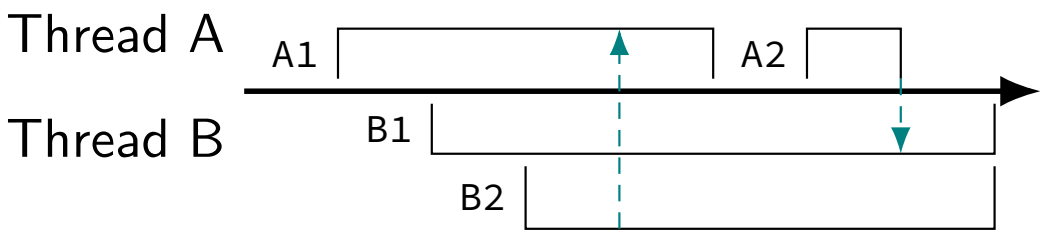
# Effects of Memory Orders (2)

The execution of the last example could look like this on an x86 CPU:

Assembly:

```
threadA():
  movl a(%rip), %edi   (A1)
  movl %edi, b(%rip)   (A2)
  jmp  safe_print(int)
threadB():
  movl b(%rip), %edi   (B1)
  movl $5, a(%rip)     (B2)
  jmp  safe_print(int)
```

Execution on the CPU:




- A2 has a data dependency on A1, so it must wait for A1 to be finished
- B1 and B2 are independent
- In thread A, B2 happens before A1
- In thread B, A2 happens before B1

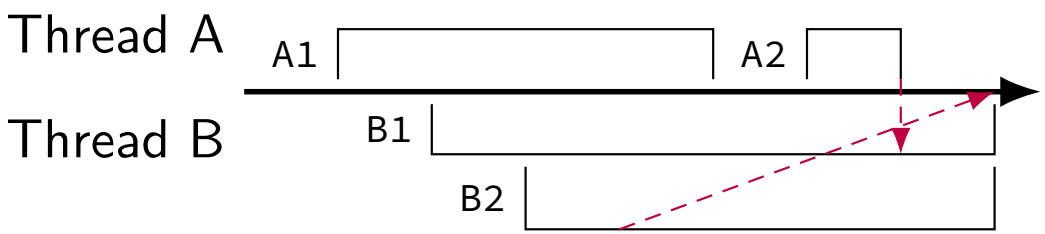
# Effects of Memory Orders (3)

When `std::memory_order_seq_cst` is used instead in the previous example, the execution looks as follows:

Assembly:

```
threadA():
  movl a(%rip), %edi   (A1)
  movl %edi, b(%rip)   (A2)
  mfence
  jmp  safe_print(int)
threadB():
  movl b(%rip), %edi   (B1)
  movl $5, a(%rip)     (B2)
  mfence
  jmp  safe_print(int)
```

Execution on the CPU:




- Due to the special `mfence` instruction, B2 is not allowed to be visible before A2 in thread A
- There is a single total order that is visible in all threads, e.g. $A1 < B1 < B2 < A2$

# Compare-And-Swap Operations (1)

Often, the arithmetic atomic operations are not sufficient for an algorithm.
Instead of falling back to using mutexes, *compare-and-swap* (CAS) operations can be used.
Conceptually, a CAS operation works as follows:

```
bool compare_and_swap(
    int* atomic_variable, int& expected, int desired
) {
    int value = *atomic_variable;
    if (value == expected) {
        *atomic_variable = desired;
        return true;
    } else {
        expected = value;
        return false;
    }
}
```

# Compare-And-Swap Operations (2)

CAS operations are usually used in a loop with the following steps:

1. Load value of atomic variable into local variable
2. Do computation with the local variable assuming that no other thread will modify the value of the atomic variable
3. Generate new desired value for the atomic variable
4. Do a CAS operation on the atomic variable with the local variable as expected value
5. Start the loop from the beginning if the CAS operation fails

Note that steps 2 and 3 are allowed to use operations that are not atomic!

# Example for Compare-And-Swap

CAS operations are commonly used to implement fast, thread-safe data structures. An insert into a singly linked list could be implemented like this:

```
void insert(const T& value) {
    auto* entry = allocateEntry();
    entry->value = value;
    bool casSuccessful;
    do {
        auto* oldHead = listHead.load(); // Step 1
        entry->next = oldHead; // Step 2
        auto* newHead = entry; // Step 3
        casSuccessful = // Step 4
            listHead.compare_exchange_weak(oldHead, newHead);
    } while (!casSuccessful); // Step 5
}
```

# Weak and Strong Compare-And-Swap Operations

The `std::atomic` class has two member functions for CAS operations: `compare_exchange_weak()` and `compare_exchange_strong()`.
Both versions have three parameters:

- The expected value
- The desired value
- The memory order (optional, default is `std::memory_order_seq_cst`)

The weak version is allowed to return false, even when no other thread modified the value. This is called "spurious failure".
General rule: If you use a CAS operation in a loop, always use the weak version.