C++ Software Engineering

for engineers of other disciplines

Module 10

"C++ Parallelism"

3rd Lecture: Multithreading

ALTEN

Spring 2022

Gothenburg, Sweden

CPU & Threads

ALTEN

- CPU sockets are divided into cores and cores are divided into threads.
- Numbers of *CPUs* is “socket * cores * threads”.
- To get the best performance, it is ideal to launch a thread per *CPU* – even better if threads are not siblings.

```
mrz@vu:~$ cat /sys/devices/system/cpu/cpu0/topology/thread_siblings_list
0
```

- Threads which are in the same cores are called *siblings* and in *most* of the cases have inferior performance compared to threads on separate cores.

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 158
Model name:            Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz
```

std::thread

- A class representing a single thread of execution.
- Begin execution upon construction.
- Threads can communicate via:
  - `std::promise`
  - `std::condition_variable`
- Synchronizations on shared resources can happen using:
  - `std::mutex`
  - `std::atomic`
- Threads should either be `join`ed or `detach`ed

Observers

`joinable`	checks whether the thread is joinable, i.e. potentially running in parallel context (public member function)
`get_id`	returns the *id* of the thread (public member function)
`native_handle`	returns the underlying implementation-defined thread handle (public member function)
`hardware_concurrency` [static]	returns the number of concurrent threads supported by the implementation (public static member function)

Operations

`join`	waits for a thread to finish its execution (public member function)
`detach`	permits the thread to execute independently from the thread handle (public member function)
`swap`	swaps two thread objects (public member function)

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

join & detach

ALTEN

- Undetached threads which are not joined cause segfault!

```cpp
void foo()
{
    // simulate expensive operation
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

void bar()
{
    // simulate expensive operation
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main()
{
    std::cout << "starting first helper...\n";
    std::thread helper1(foo);

    std::cout << "starting second helper...\n";
    std::thread helper2(bar);

    std::cout << "waiting for helpers to finish..." << std::endl;
    helper1.join();
    helper2.join();

    std::cout << "done!\n";
}
```

https://en.cppreference.com/w/cpp/thread/thread/join

```cpp
void independentThread()
{
    std::cout << "Starting concurrent thread.\n";
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "Exiting concurrent thread.\n";
}

void threadCaller()
{
    std::cout << "Starting thread caller.\n";
    std::thread t(independentThread);
    t.detach();
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Exiting thread caller.\n";
}

int main()
{
    threadCaller();
    std::this_thread::sleep_for(std::chrono::seconds(5));
}
```

https://en.cppreference.com/w/cpp/thread/thread/detach

Mutual Exclusion

- STL provides mutual exclusion:
  - lock in `std::mutex`
  - Different means for acquiring a lock:
    - `std::lock_gaurd`
    - `std::shared_lock`
    - `std::unique_lock`
    - `std::scoped_lock`
  - Different strategies for locking.
  - Reentrant mutex to avoid dead lock by `std::recursive_mutex`
  - `std::lock` avoid locks any given number of *lockables* using an algorithm which never deadlocks yet:

Locking

lock	locks the mutex, blocks if the mutex is not available (public member function)
try_lock	tries to lock the mutex, returns if the mutex is not available (public member function)
unlock	unlocks the mutex (public member function)

https://en.cppreference.com/w/cpp/thread/mutex

Type	Effect(s)
defer_lock_t	do not acquire ownership of the mutex
try_to_lock_t	try to acquire ownership of the mutex without blocking
adopt_lock_t	assume the calling thread already has ownership of the mutex

https://en.cppreference.com/w/cpp/thread/lock_tag

RAII wrapper for this function, and is generally preferred to a naked call to std::lock.

https://en.cppreference.com/w/cpp/thread/lock

std::lock_gaurd

- RAII-style mechanism for owning a mutex for duration of scoped block, to avoid hassle of *lock* and *release*.
- **std::lock_gaurd** is blocking and non-copyable.
- **std::unique_lock** is a general mutex wrapper.

```
int g_i = 0;
std::mutex g_i_mutex;  // protects g_i

void safe_increment()
{
    const std::lock_guard<std::mutex> lock(g_i_mutex);
    ++g_i;

    std::cout << std::this_thread::get_id() << ": " << g_i << '\n';

    // g_i_mutex is automatically released when lock
    // goes out of scope
}
```

https://en.cppreference.com/w/cpp/thread/lock_guard

```
void transfer(Box &from, Box &to, int num)
{
    // don't actually take the locks yet
    std::unique_lock<std::mutex> lock1(from.m, std::defer_lock);
    std::unique_lock<std::mutex> lock2(to.m, std::defer_lock);

    // lock both unique_locks without deadlock
    std::lock(lock1, lock2);

    from.num_things -= num;
    to.num_things += num;

    // 'from.m' and 'to.m' mutexes unlocked in 'unique_lock' dtors
}
```

Defined in header <mutex>

```
template< class Mutex >
class unique_lock;
```

Defined in header <mutex>

```
template< class Mutex >
class lock_guard;
```

Defined in header <shared_mutex>

```
template< class Mutex >
class shared_lock;
```

(since C++14)

Defined in header <mutex>

```
template< class... MutexTypes >
class scoped_lock;
```

(since C++17)

std::condition_variable

- Another mean of inter-thread communication.
- One thread needs to acquire a lock, change a value, and notify others.

```
std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;
```

```
int main()
{
    std::thread worker(worker_thread);

    data = "Example data";
    // send data to the worker thread
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, []{return processed;});
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
}
```

Notification

`notify_one`	notifies one waiting thread (public member function)
`notify_all`	notifies all waiting threads (public member function)

Waiting

`wait`	blocks the current thread until the condition variable is woken up (public member function)
`wait_for`	blocks the current thread until the condition variable is woken up or after the specified timeout duration (public member function)
`wait_until`	blocks the current thread until the condition variable is woken up or until specified time point has been reached (public member function)

Native handle

`native_handle`	returns the native handle (public member function)

```
void worker_thread()
{
    // Wait until main() sends data
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{return ready;});

    // after the wait, we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // Send data back to main()
    processed = true;
    std::cout << "Worker thread signals data processing completed\n";

    // Manual unlocking is done before notifying, to avoid waking up
    // the waiting thread only to block again (see notify_one for details)
    lk.unlock();
    cv.notify_one();
}
```

Output

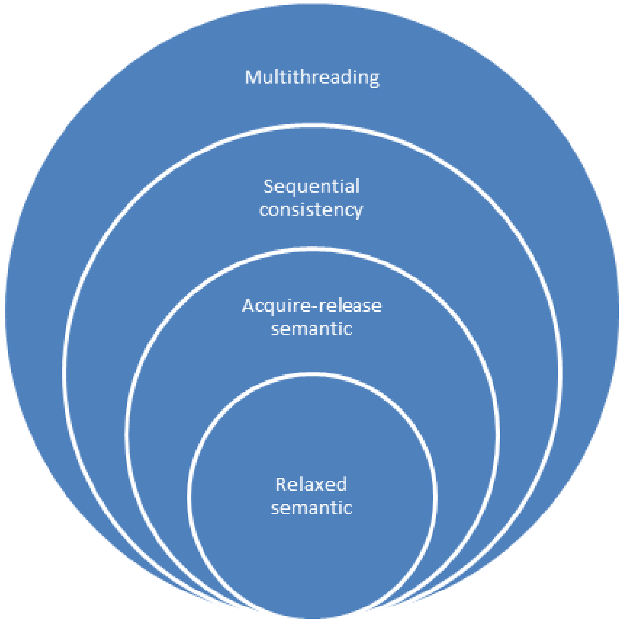
```
main() signals data ready for processing
Worker thread is processing data
Worker thread signals data processing completed
Back in main(), data = Example data after processing
```

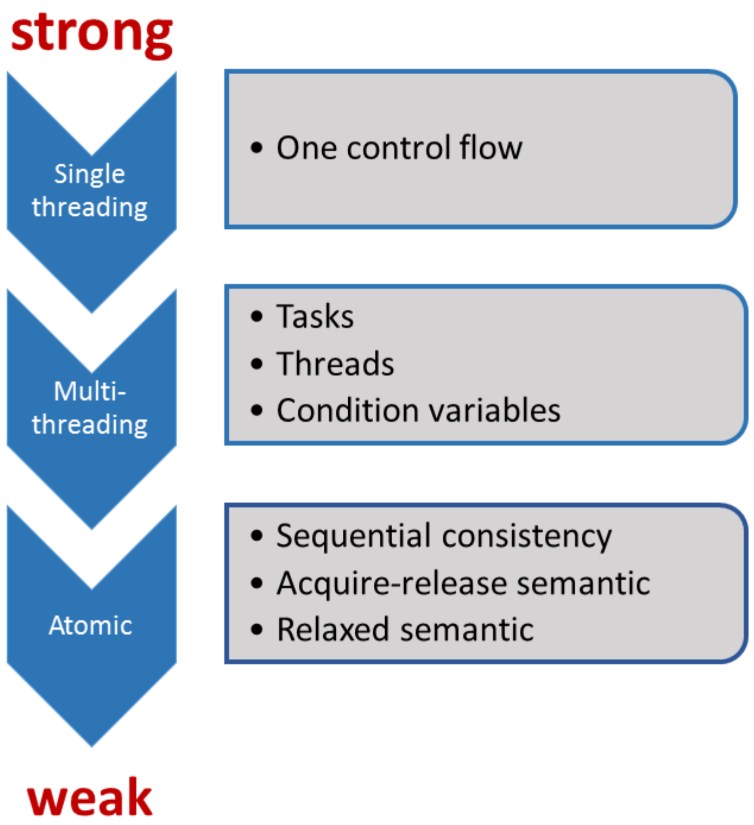
Memory Modeling Contracts

- The weaker the memory model, the higher optimization potential!

In computing, a memory model describes the interactions of threads through memory and their shared use of the data. A memory model allows a compiler to perform many important optimizations. https://en.wikipedia.org/wiki/Memory_model_(programming)

Expert levels





Atomic

- An area in memory where there is no race! If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined.

```
int cnt = 0;
auto f = [&]{cnt++;};
std::thread t1{f}, t2{f}, t3{f}; // undefined behavior
```

```
std::atomic<int> cnt{0};
auto f = [&]{cnt++;};
std::thread t1{f}, t2{f}, t3{f}; // OK
```

https://en.cppreference.com/w/cpp/language/memory_model

- Memory order specifies how memory (including regular non-atomic accesses) are to be ordered around an atomic operation.

```
inline constexpr memory_order memory_order_relaxed = memory_order::relaxed;
inline constexpr memory_order memory_order_consume = memory_order::consume;
inline constexpr memory_order memory_order_acquire = memory_order::acquire;
inline constexpr memory_order memory_order_release = memory_order::release;
inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;
```

https://en.cppreference.com/w/cpp/atomic/memory_order

std::atomic

- User datatypes could be used as templated parameter for **std::atomic** but it is not guaranteed that the operations would be lock-free.The only guaranteed lock-free data type is **std::atomic_flag**.

is_lock_free	checks if the atomic object is lock-free (public member function)
store	atomically replaces the value of the atomic object with a non-atomic argument (public member function)
load	atomically obtains the value of the atomic object (public member function)
operator T	loads a value from an atomic object (public member function)
exchange	atomically replaces the value of the atomic object and obtains the value held previously (public member function)
compare_exchange_weak compare_exchange_strong	atomically compares the value of the atomic object with non-atomic argument and performs atomic exchange if equal or atomic load if not (public member function)
wait (C++20)	blocks the thread until notified and the atomic value changes (public member function)
notify_one (C++20)	notifies at least one thread waiting on the atomic object (public member function)
notify_all (C++20)	notifies all threads blocked waiting on the atomic object (public member function)

https://en.cppreference.com/w/cpp/atomic/atomic

```
std::atomic<bool> lock(false); // holds true when locked
                               // holds false when unlocked

void f(int n)
{
    for (int cnt = 0; cnt < 100; ++cnt) {
        while(std::atomic_exchange_explicit(&lock, true, std::memory_order_acquire))
            ; // spin until acquired
        std::cout << "Output from thread " << n << '\n';
        std::atomic_store_explicit(&lock, false, std::memory_order_release);
    }
}
int main()
{
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n) {
        v.emplace_back(f, n);
    }
    for (auto& t : v) {
        t.join();
    }
}
```

```
std::atomic_flag lock = ATOMIC_FLAG_INIT;

void f(int n)
{
    for (int cnt = 0; cnt < 100; ++cnt) {
        while(std::atomic_flag_test_and_set_explicit(&lock, std::memory_order_acquire))
             ; // spin until the lock is acquired
        std::cout << "Output from thread " << n << '\n';
        std::atomic_flag_clear_explicit(&lock, std::memory_order_release);
    }
}

int main()
{
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n) {
        v.emplace_back(f, n);
    }
    for (auto& t : v) {
        t.join();
    }
}
```

https://en.cppreference.com/w/cpp/atomic/atomic_flag_test_and_set

DEMO!

CPU affinity


```
cpu_set_t cpuset;
CPU_ZERO(&cpuset);
CPU_SET(i, &cpuset);
int rc = pthread_setaffinity_np(threads[i].native_handle(),
                                sizeof(cpu_set_t), &cpuset);
if (rc != 0) {
  std::cerr << "Error calling pthread_setaffinity_np: " << rc << "\n";
}
```

Where to have mutex?


```cpp
//std::lock_guard<std::mutex> guard(g_pages_mutex);
size_t i = std::rand()%5;
std::cout << "Sleeping for " << i << " second in thread "
std::this_thread::sleep_for(std::chrono::seconds(i));
std::string result = "fake content";

std::lock_guard<std::mutex> guard(g_pages_mutex);
g_pages[url] = result;
```

Reenterant Mutex


```
class X {
    std::recursive_mutex m;
    std::string shared;
  public:
    void fun1() {
      std::lock_guard<std::recursive_mutex> lk(m);
      shared = "fun1";
      std::cout << "in fun1, shared variable is now " << shared << '\n';
    }
    void fun2() {
      std::lock_guard<std::recursive_mutex> lk(m);
      shared = "fun2";
      std::cout << "in fun2, shared variable is now " << shared << '\n';
      fun1(); // recursive lock becomes useful here
      std::cout << "back in fun2, shared variable is " << shared << '\n';
    };
};
```

Scoped Lock


```cpp
// use std::scoped_lock to acquire two locks without worrying about
// other calls to assign_lunch_partner deadlocking us
// and it also provides a convenient RAII-style mechanism

std::scoped_lock lock(e1.m, e2.m);

// Equivalent code 1 (using std::lock and std::lock_guard)
// std::lock(e1.m, e2.m);
// std::lock_guard<std::mutex> lk1(e1.m, std::adopt_lock);
// std::lock_guard<std::mutex> lk2(e2.m, std::adopt_lock);

// Equivalent code 2 (if unique_locks are needed, e.g. for condition variables)
// std::unique_lock<std::mutex> lk1(e1.m, std::defer_lock);
// std::unique_lock<std::mutex> lk2(e2.m, std::defer_lock);
// std::lock(lk1, lk2);
```

Threads are not FAIR!


```
alice and bob are waiting for locks
christina and alice are waiting for locks
dave and bob are waiting for locks
christina and bob are waiting for locks
alice and bob got locks
dave and bob got locks
christina and bob got locks
christina and alice got locks
Employee alice has lunch partners: bob christina
Employee bob has lunch partners: alice dave christina
Employee christina has lunch partners: bob alice
Employee dave has lunch partners: bob
```

```
alice and bob are waiting for locks
dave and bob are waiting for locks
christina and bob are waiting for locks
alice and bob got locks
dave and bob got locks
christina and bob got locks
christina and alice are waiting for locks
christina and alice got locks
Employee alice has lunch partners: bob christina
Employee bob has lunch partners: alice dave christina
Employee christina has lunch partners: bob alice
Employee dave has lunch partners: bob
```

Conditional Variables


```
std::unique_lock<std::mutex> lk(cv_m);
std::cerr << "Waiting... \n";
cv.wait(lk, []{return i == 1;});
std::cerr << "...finished waiting. i == 1\n";
```

```
std::this_thread::sleep_for(std::chrono::seconds(1));
{
    std::lock_guard<std::mutex> lk(cv_m);
    std::cerr << "Notifying...\n";
}
cv.notify_all();

std::this_thread::sleep_for(std::chrono::seconds(1));

{
    std::lock_guard<std::mutex> lk(cv_m);
    i = 1;
    std::cerr << "Notifying again...\n";
}
//cv.notify_all();
```