Threads and Async Programming

THREAD CONCEPTS

- Multi-Threading
- Parallelism
- Impact on performance
- Scheduling
- Synchronization
- Maintainability

THREADING VERSUS PARALLELISM

Threading

- Single CPU switches between different threads very quickly, giving a time-sliced concurrency.
- Only one thread is running at any given time.

Parallelism

- Threads are running in parallel, simultaneously, each on different CPU core..
- Multiple threads are running at any given time

THREAD SCHEDULING (WINDOWS)

- Thread scheduling is a combination of process priority, thread priority, and round-robin preemptive scheduling.
- Preemptive scheduling means each thread receives one or more quantums of execution; but can be preempted if a high priority thread starts.
- A thread is preempted after completing the quantum(s). The schedule then looks for the next thread to schedule – either a higher priority thread or round-robin fashion.
- Threads running over their base priority are eroded 1 priority when completing a quantum(s). Threads may also receive a priority boost.

THREAD SAFETY

- All shared state must be thread-safe
- Make use of the Immutable Object pattern.
  - If an object can't be modified after creation, then you can't have uncoordinated updates. Make a copy if you want to modify it.
- Follow the Command Query Segregation Principle.
  - Separate code that modifies the object from code that reads because reading can happen concurrently, but modification can’t.
- Don’t reinvent the wheel. C++ includes built-in support for thread safety: atomic operations, mutual exclusion, condition variables, promises and futures.

CREATE A THREAD

- First – create a function
- Initialize a thread object
- Synchronize thread to prevent the thread from simply ending.

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

void FuncA() {
    std::cout << "Hello, world‟ << std::endl;
}

int main() {
    thread t1{ FuncA};
    t1.join();
    return 0;
}
```

THREAD PARAMETERS

- When using functions, thread parameters are passed in a comma separated list.
- For lambdas, use standard syntax: parameters or captured variables.

```
void Func(int a, int b){
    cout << a + b;
}
int main() {
    int i = 2;
    thread t1{ Func, i, i + 5 };
    thread t2{ [=] {cout << i; } };
    t1.join();
    t2.join();
    return 0;
}
```

THREAD RETURN

Variety of techniques to return a value from a thread.

- Parameters that are references
- Wrap thread in a structure or class
- Globals run the risk of data corruption (require locking)

THREAD WITH STATE

- Thread parameters allow minimal transfer of thread specific data.
- You can also use a class or structure to create a thread object.
- The data members provide rich information available to the threads.

```
class Foo {
private:
    int _data;
public:
    Foo(int data) : _data(data){}
    void operator()() /* worker */ {  ++_data;  }
    int get() const { return _data; }
};
int main() {
    Foo foo(100);
    thread t1(std::ref(foo));
    t1.join();
    cout << foo.get() << endl; // prints 101
}
```

PROMISE AND FUTURE

- A promise guarantees that at some time there will be a state change or return value from thread.

```
auto promise = std::promise<std::string>();
auto producer = std::thread([&] {
    // simulate some long-ish running task
    std::this_thread::sleep_for(std::chrono::seconds(5));
    promise.set_value("Some Message");
});
```

- The promise can be used to get a **std::future**, which is the object that gives us this return result.

```
auto future = promise.get_future();
auto consumer = std::thread([&] {
    std::cout << future.get() << std::endl;
});
```

ASYNC

- std::async returns a std::future that holds the return value that will be calculated by the function.

```
string message;
(std::async(launch::async, [&message](){
    this_thread::sleep_for(chrono::seconds(5));
    message = "Welcome to threads!";
}).wait();
```

- When that future gets destroyed it waits until the thread completes making your code effectively single threaded.
- This is easily overlooked when you don't need the return value
  - launch::async policy means fn must run asynchronously on a separate thread
  - launch::deferred policy means fn may run when get() or wait() is called on the future

```
future<int> g = std::async( launch::deferred, [](int n)  { return n * n; },  10);
```

PACKAGED TASK

- A packaged_task wraps a callable element and allows its result to be retrieved asynchronously..
- It is similar to std::function, but transferring its result automatically to a future object.
- The object contains internally two elements:
  - A stored task, which is some callable object (such as a function pointer, pointer to member or function object) whose call signature shall take arguments of the types in Args... and return a value of type Ret.
  - A shared state, which is able to store the results of calling the stored task (of type Ret) and be accessed asynchronously through a future.