Parallel Processing with C++ Multithreading

https://github.com/onu-eccs-faculty/threading-demos

Quick Start: “Hello World”

```
// Compile command: g++ -pthread -std=c++17 -o 01_hello_world 01_hello_world.cpp

#include <iostream>
#include <thread>
using namespace std;

void hello() {
     cout << "Hello
     Concurrent
     World\n";
}
int main() {

     // Launch a sperate thread to execute the function hello()
     thread t(hello);

     // Wait for the new thread to finish
     t.join();

     return 0;
}
```

Try in Demo
01_hello_world.cpp

Today’s demos (ungraded)

- https://github.com/onu-eccs-faculty/threading-demos

Background

- Requirements
  - Recent compiler such as
    - g++,
    - clang++
    - Microsoft Visual Studio
  - C++17 Standard Thread Library (all above compilers ship with it)
- History
  - C++11 standard published in 2011 by the C++ Standard Committee
    - First support for multithreading
    - No need for platform-specific extension: i.e., more portable code
    - No need to third-party libraries
    - Allows more concurrency to improve application performance
  - C++14 and C++17 standards built upon C++11

Launching a Thread

- Task for a thread can be:
  - Function
  - Function object with arguments that receives messages from other threads
  - Function struct with arguments that receives messages from other threads
- Need to construct a thread object

```
void do_some_function();
std::thread my_thread (do_some_function);
```

- Can pass an instance of a class by overloading the () operator:

```
class task {
   public:
      void operator () () const {
         // ... do something
       }
task t;
std::thread my_thread (t);
...
t.join() // wait for task to finish.
```

Waiting for a Thread to Finish

```
int main() {

  int some_local_state = 0;

  // Create thread structure
  new_thread_func my_func (some_local_state);

  // Start new thread
  std::thread t (my_func);

  // Do stuff in old thread
  try {
    old_thread_do_something ();
  }
  catch(...) {
    t.join();
    throw;
  }

  // Wait until the new thread is done
  t.join();
  cout << "Threads rejoined!!!" << endl;
}
```

```
struct new_thread_func {

  // A state variable
  int& i;

  // Structure constructor
  new_thread_func(int& i_):i(i_) {}

  // The action to be taken when the thread is spawned
  // This is a functor
  void operator ()() {
    for (unsigned j = 0; j < 10; ++j) {
      new_thread_do_something (i);
    }
  }
};

void new_thread_do_something (int& i) {
  ++i;
  cout << "Sleep " << i << endl;
  sleep(1);
}

void old_thread_do_something() {
  cout << "Ho hum..." << endl;
}
```

Try in Demo
02_waiting.cpp

Passing args to a thread function

- Additional arguments to thread constructor
- When passing args by reference, safer to wrap with std::ref

```
int main () {

  // Passing a primitive variable
  int n = 3;
  thread primitive_pass (printer, n);
  primitive_pass.join();

  // Passing a reference variable
  vector<int> v = {1,2,3};
  thread t2 (doubler, ref(v));
  t2.join();

  cout << "v is now: ";
  for (int i = 0; i < v.size(); i++){
    cout << v[i] << " ";
  }
  cout << endl;


}
```

```
void printer (int i) {
  cout << i << " is a nifty number!!" << endl;
}

void doubler (vector<int>& v) {
  cout << "v used to be: ";
  for (int i = 0; i < v.size(); i++){
    cout << v[i] << " ";
    v[i] *= 2;
  }
  cout << endl;
}
```

Try in Demo
03_ref_param.cpp

Multithreading Exercise 1

Try in Demo
04_activity.cpp

- Refactor the code to run the “totalVector” function in a new thread
- Print out the total in the parent thread after “totalVector” completes

Make this run as a separate thread

```
void initVec(vector<int>& v) {
  for (unsigned int i = 0; i < v.size(); i++)
    v[i] = i;
}

void totVec(vector<int>& v, int& total) {
  for (unsigned int i = 0; i < v.size(); i++)
    total += v[i];
}

int main () {
  vector<int> v1(10);
  initVec(v1);
  int total_main = 0;

  // make this subroutine run as a separate thread
  totVec(v1,total_main);

  // Print out total
  cout << "total_main: " << total_main << endl;

  return 0;
}
```

Detaching a thread

```
// Example: document editing application
...
void open_document(string const& filename){
    display_edit_gui (filename);
    user_command cmd = get_user_input ();
    while((cmd.type != done_editing)
    {
        if(cmd.type == open_new_document)
        {
            string const new_file_name = get_filename_from_user ();
            thread t (open_document, new_name);
            // New thread to handle editing  a new document in a sperate GUI window
            t.detach ();
        }
        else
        {
            process_user_input(cmd);
        }
    }
}
int main() {
    open_document(”default.doc");
}
```

Thread containers

- Threads objects can be added to object containers, such as vectors
- Example: Spawning a number of threads and waiting for them to finish:

```
void do_work (unsigned id){
...
}

void f() {
     vector<std::thread> threads;
     for(unsigned i = 0; i < 20; ++i) {
          // Spawn threads by calling emplace_back( )
          // This inserts a new thread at the end of the vector. The thread is
          // constructed in place using args as the arguments for its constructor.
          threads.emplace_back (thread (do_work,i));
     }
     // Wait for each thread to finish
     for (auto& entry: threads)
          entry.join()
}

int main()
     { f();
}
```

Choosing the Number of Threads at Runtime

- C++ Standard Library provides:

```
std::thread::hardware_concurrency()
```

- This function returns the number of threads that can execute in parallel.
- The return values could be the number of cores, if the cores do not support SMT (simultaneous multi-threading).
- If the cores support SMT, the returned value could be higher than the number of cores (although thread performance may not be higher).

Identifying Threads

- Threads ID are of type: std::thread::id
- Thread ID can retrieved by
  - Calling get_id() member function on a thread object
  - Calling std::this_thread::get_id() to retrieve the ID of the current thread
- thread::id can be
  - Checked if equal to another ID
  - Ordered or sorted
  - Used as keys in associative containers

```
thread::id master_thread; void

common_function() {
    if (this_thread::get_id() ==
    master_thread) {
        // Mater thread does work differently than other threads
        do_master_thread_work();
    }
    do_some_other_work();
}

int main () {
    master_thread = this_thread.get_id();
    // Spawn new threads with common_function()
    ....
    common_function(); // Master thread calls common_function
}
```