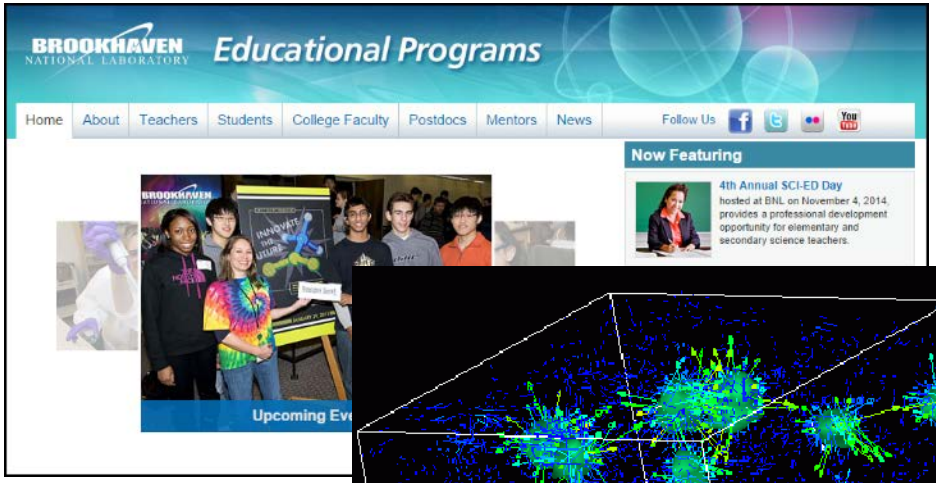
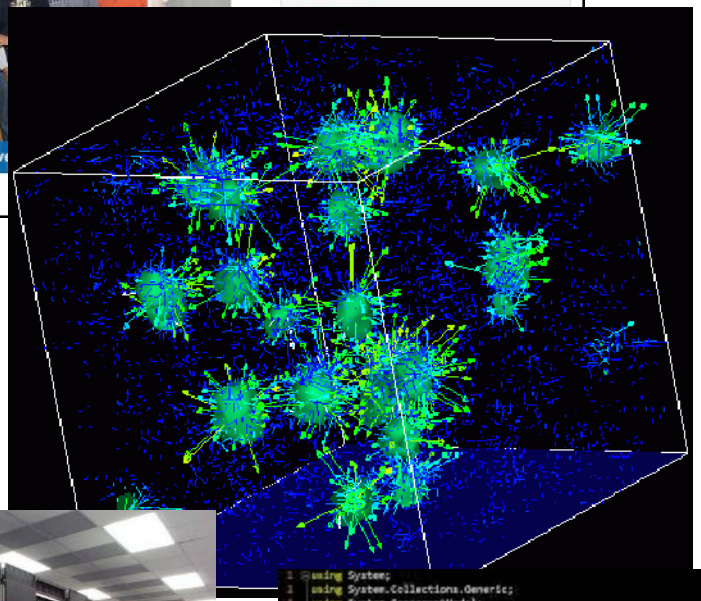
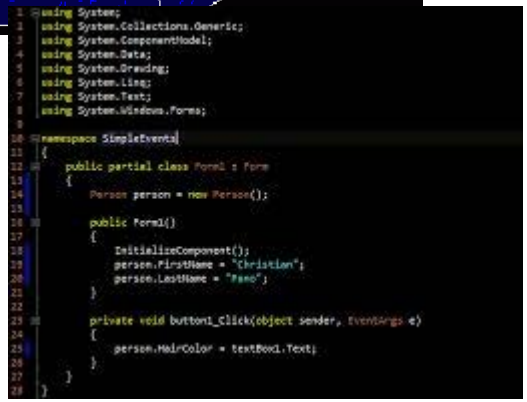






Survey of Scientific Computing (SciComp 301)

Dave Biersach
Brookhaven National Laboratory
dbiersach@bnl.gov

Session 27
Parallel Programming Using Threads

Session Goals

- Learn about PC system architecture (symmetric multiprocessing vs **multicore**)
- Appreciate OS task and thread scheduling (multitasking vs. **multithreading**)
- Implement the **Thread Control Block (TCB) pattern** of central dispatch control, in order to **parallelize numerical integration using Simpson's Rule**

Admiral Grace Hopper

Grace Brewster Murray Hopper (née **Murray** December 9, 1906 – January 1, 1992) was an American computer scientist and United States Navy rear admiral.[1] One of the first programmers of the Harvard Mark I computer, she was a pioneer of computer programming who invented one of the first linkers. She popularized the idea of machine-independent programming languages, which led to the development of COBOL, an early high-level programming language still in use today.

Prior to joining the Navy, Hopper earned a Ph.D. in mathematics from Yale University and was a professor of mathematics at Vassar College. Hopper attempted to enlist in the Navy during World War II but was rejected because she was 34 years old. She instead joined the Navy Reserves. Hopper began her computing career in 1944 when she worked on the Harvard Mark I team led by Howard H. Aiken. In 1949, she joined the Eckert–Mauchly Computer Corporation and was part of the team that developed the UNIVAC I computer. At Eckert–Mauchly she began developing the compiler. She believed that a programming language based on English was possible. Her compiler converted English terms into machine code understood by computers. By 1952, Hopper had finished her program linker (originally called a compiler), which was written for the A-0 System.[2][3][4][5] During her wartime service, she co-authored three papers based on her work on the Harvard Mark 1.

Admiral Grace Hopper

In accordance with Navy attrition regulations, Hopper retired from the Naval Reserve with the rank of commander at age 60 at the end of 1966.[26] She was recalled to active duty in August 1967 for a six-month period that turned into an indefinite assignment. She again retired in 1971 but was again asked to return to active duty in 1972. She was promoted to captain in 1973 by Admiral Elmo R. Zumwalt, Jr.[27]

After Republican Representative Philip Crane saw her on a March 1983 segment of *60 Minutes*, he championed H.J.Res. 341, a joint resolution originating in the House of Representatives, which led to her promotion on 15 December 1983 to commodore by special Presidential appointment by President Ronald Reagan.[27][28][29][30] She remained on active duty for several years beyond mandatory retirement by special approval of Congress.[31] Effective November 8, 1985, the rank of commodore was renamed rear admiral (lower half) and Hopper became one of the Navy's few female admirals.

My Dinner with Admiral Hopper - **1987**

"When farmers wanted to plow bigger fields, they didn't breed bigger and bigger horses…

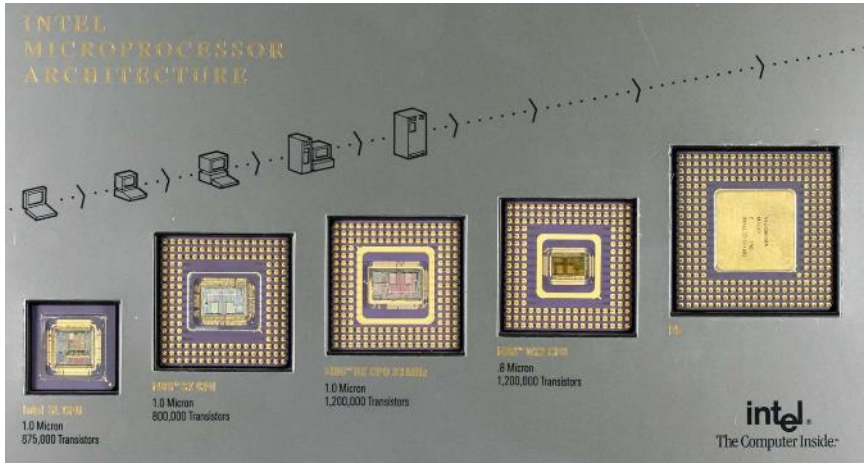
In 1987 Admiral Hopper came to West Point (USMA) to receive an award for a lifetime of service. I was selected to have dinner with her before the presentation…

… so I asked for her thoughts about the future of computing…

… they learned how to stitch multiple horses together."

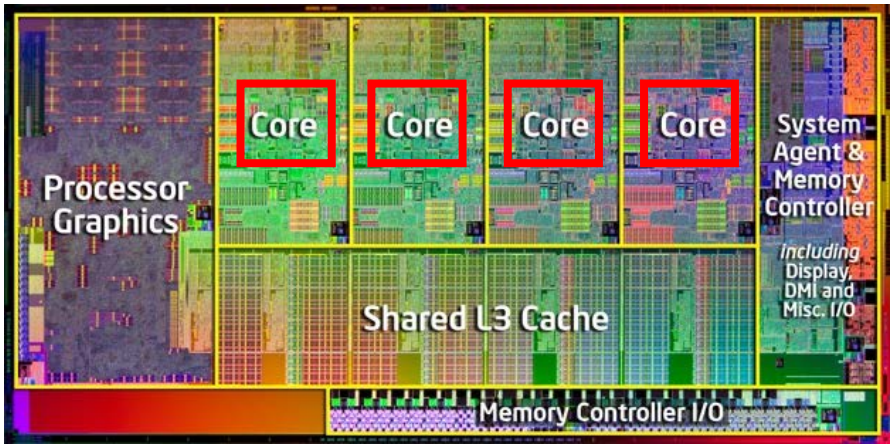
My Dinner with Admiral Hopper - **1987**

"When farmers wanted to plow bigger fields, they didn't breed bigger and bigger horses…



Intel CPUs from 1985 - 2008

… they learned how to stitch multiple horses together."



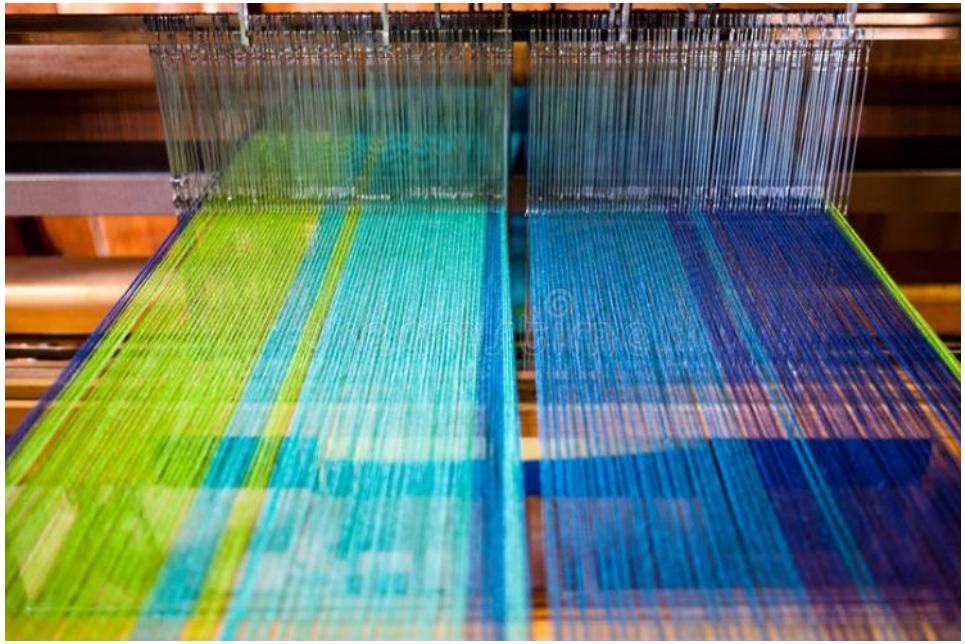
Intel CPUs from 2008 - 2020

My Dinner with Admiral Hopper - **1987**

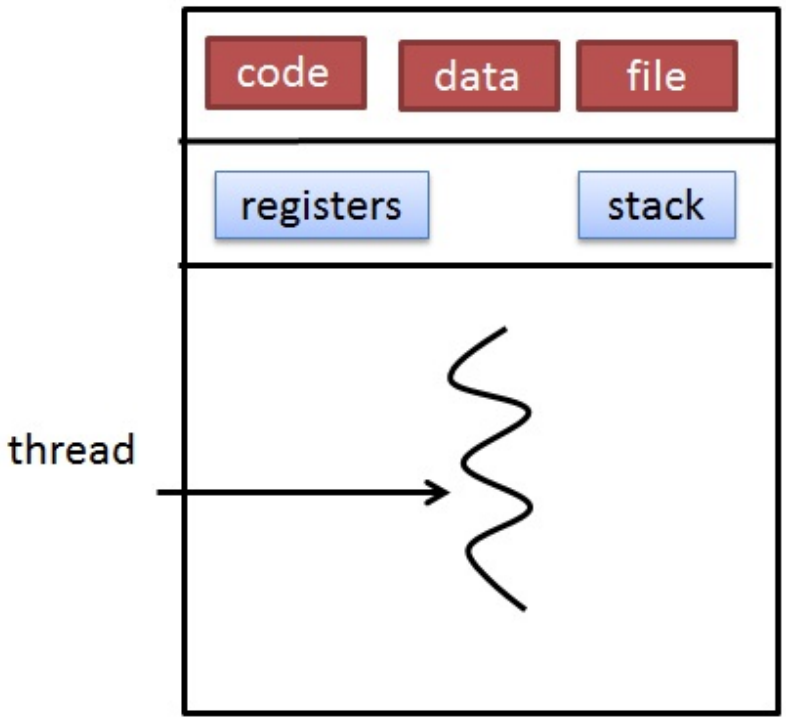
"A task is like a blanket.
A multitasking OS switches between blankets…

… but a blanket is made from individual threads. A multithreaded OS switches between the parts of each blanket.

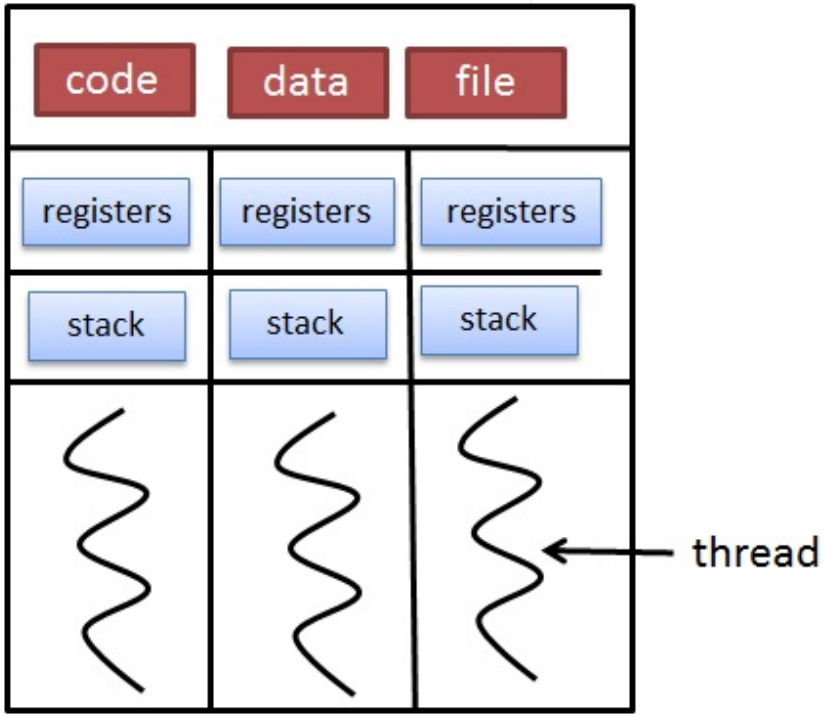
Future programming will all be about how to *properly* use **threads**."



A "**Thread**" of Execution



Single-threaded Process



Multithreaded Process

System Architecture

- Due to Moore's law and exponentially rising power demands, system designers moved away from computers having multiple physical CPUs (**SMP** - symmetric multi-processing)
  - The new model became "multi-core" with computers having one main CPU that is divided internally into multiple execution cores
  - Each core can run independently from the others
  - System memory (global variables) are shared between all cores – but each core gets its own local (stack based) variables
- Multicore designs are lower cost and demand less power since core-to-core communication (wiring) is all "on-chip" (inside the CPU) versus SMP where wiring is on the circuit board connecting the physically separated CPUs

CPU⟺OS Coordination

- A single-threaded application can only run on **one** core at a time – the other cores (and even other CPUs in an SMP machine) will be *underutilized* (a waste of money)
- Modern programmers try to extract maximum performance from the locally available hardware
  - Applications must be designed to run sections of the code in **parallel** on multiple threads (and hence multiple cores) at the same time
  - For time sensitive apps, a good software design will keep each CPU Core **utilization near 100%** for the duration of the run
  - However there is no sense in launching too many threads – the OS then spends too much time **context switching** (*thrashing*) between all the threads trying to ensure no thread gets starved for CPU time

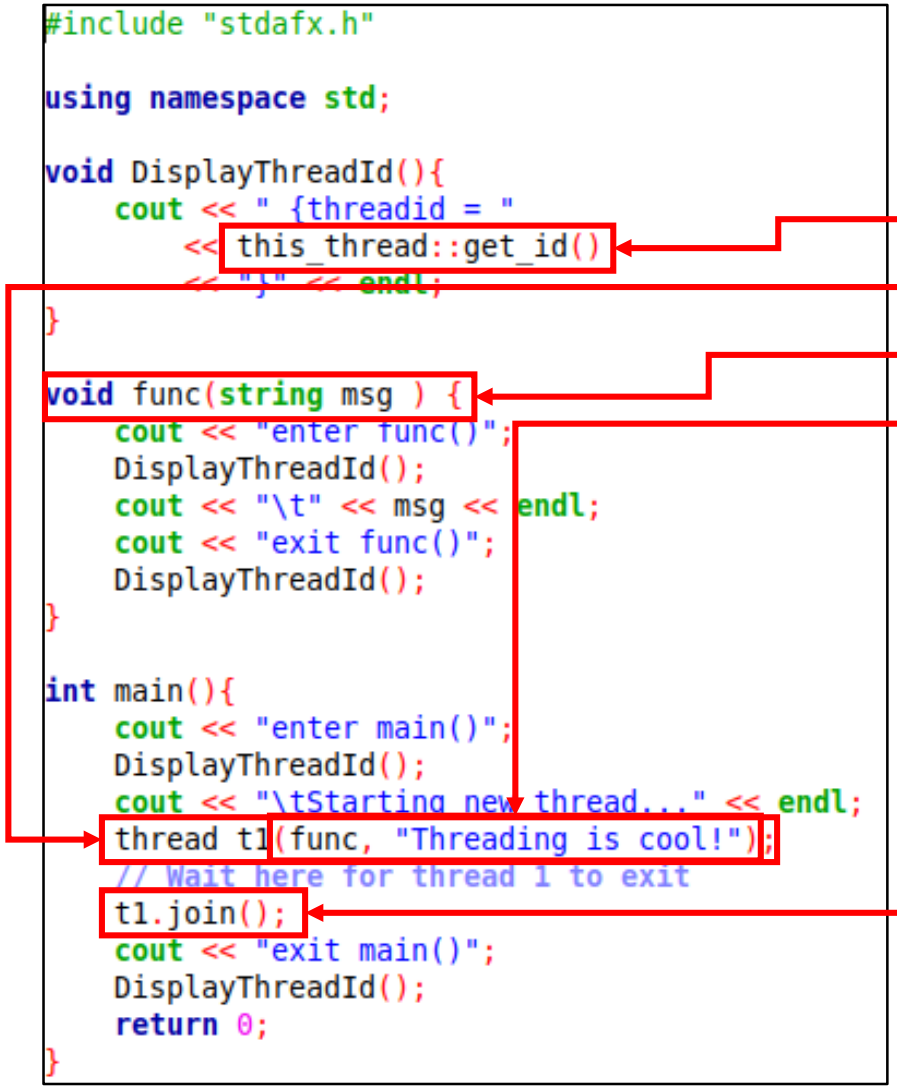
Managing Threads

- Each application has as single **primary thread** of execution, created by the OS when the program is launched
  - All apps start off as **single-threaded**
  - You as the developer must explicitly *ask* the OS (via system calls) to **create** additional threads to run parts of your code in parallel (to the maximum degree logically possible)
- You control the lifetime of the child threads you create
  - The **primary thread** is controlled by the **OS** and is terminated *only* when your application ends
  - Any child thread you create is killed by the OS when the main thread dies, so you must ensure all of your child threads **cleanly exit** before **main() returns**

Using the C++ Thread Library

- Threads cannot "see" or modify the local variables in another thread – threads can only modify **global** (heap) memory
  - Threads start running the instant your create them!
  - Each thread has **its own stack** – it gets its **own copy** of any local variables, including passed in function parameters
- You don't want to spin a thread in a "runnable" (hot) state
  - A thread should invoke **yield()** to surrender its time quantum back to the OS scheduler if there is no meaningful work that can be done
  - Threads are often waiting for an external resource to continue
- C++ has **native** support for creating & managing threads

Open Lab 1
Simple Threading



```
#include "stdafx.h"

using namespace std;

void DisplayThreadId(){
    cout << " {threadid = "
        << this_thread::get_id()
        << "}" << endl;
}

void func(string msg ) {
    cout << "enter func()";
    DisplayThreadId();
    cout << "\t" << msg << endl;
    cout << "exit func()";
    DisplayThreadId();
}

int main(){
    cout << "enter main()";
    DisplayThreadId();
    cout << "\tStarting new thread..." << endl;
    thread t1(func, "Threading is cool!");
    // Wait here for thread 1 to exit
    t1.join();
    cout << "exit main()";
    DisplayThreadId();
    return 0;
}
```

- this_thread::**get_id**()
- thread() *constructor*
- A thread `function`
- Passing thread parameters
- Wait for exit via t1.**join**()

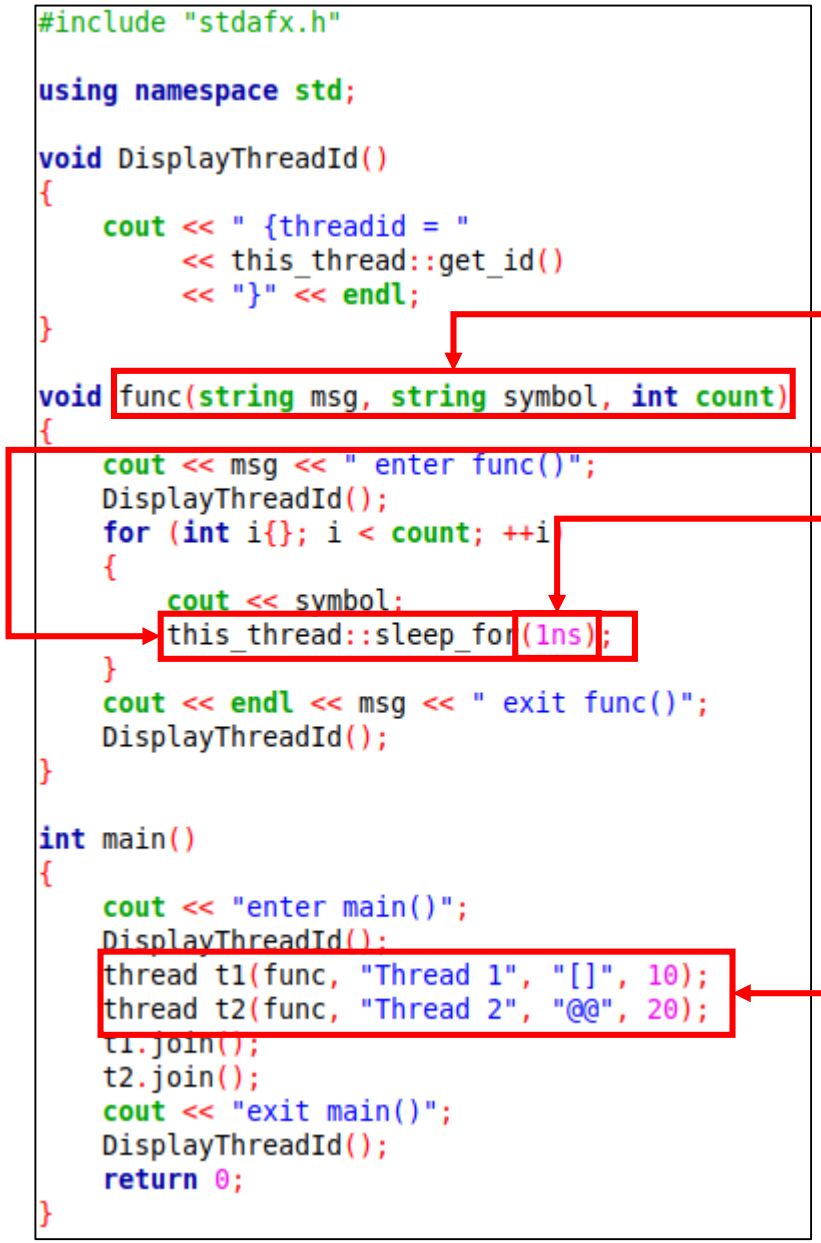
Run Lab 1
Simple Threading

```
#include "stdafx.h"

using namespace std;

void DisplayThreadId(){
    cout << " {threadid = "
        << this_thread::get_id()
        << "}" << endl;
}

void func(string msg ) {
    cout << "enter func()";
    DisplayThreadId();
    cout << "\t" << msg << endl;
    cout << "exit func()";
    DisplayThreadId();
}

int main(){
    cout << "enter main()";
    DisplayThreadId();
    cout << "\tStarting new thread..." << endl;
    thread t1(func, "Threading is cool!");
    // Wait here for thread 1 to exit
    t1.join();
    cout << "exit main()";
    DisplayThreadId();
    return 0;
}
```

- this_thread::**get_id**()
- thread() *constructor*
- A thread `function`
- Passing thread parameters
- Wait for exit via t1.**join**()

```
simple-threading
File  Edit  View  Terminal  Tabs  Help
enter main() {threadid = 140228593728704
        Starting new thread...
enter func() {threadid = 140228354119424
        Threading is cool!
exit func() {threadid = 140228354119424
exit main() {threadid = 140228593728704

Process returned 0 (0x0)   execution time : 0.043 s
Press ENTER to continue.
```

Open Lab 2
Mutex



```cpp
#include "stdafx.h"

using namespace std;

void DisplayThreadId()
{
    cout << " {threadid = "
         << this_thread::get_id()
         << "}" << endl;
}

void func(string msg, string symbol, int count)
{
    cout << msg << " enter func()";
    DisplayThreadId();
    for (int i{}; i < count; ++i)
    {
        cout << symbol;
        this_thread::sleep_for(1ns);
    }
    cout << endl << msg << " exit func()";
    DisplayThreadId();
}

int main()
{
    cout << "enter main()";
    DisplayThreadId();
    thread t1(func, "Thread 1", "[]", 10);
    thread t2(func, "Thread 2", "@@", 20);
    t1.join();
    t2.join();
    cout << "exit main()";
    DisplayThreadId();
    return 0;
}
```

- Shared thread function
- Two simultaneous threads
- this_thread::**sleep()**
- C++ STL Numeric *literals*

Run Lab 2
Mutex

```cpp
#include "stdafx.h"

using namespace std;

void DisplayThreadId()
{
    cout << " {threadid = "
         << this_thread::get_id()
         << "}" << endl;
}

void func(string msg, string symbol, int count)
{
    cout << msg << " enter func()";
    DisplayThreadId();
    for (int i{}; i < count; ++i)
    {
        cout << symbol;
        this_thread::sleep_for(1ns);
    }
    cout << endl << msg << " exit func()";
    DisplayThreadId();
}

int main()
{
    cout << "enter main()";
    DisplayThreadId();
    thread t1(func, "Thread 1", "[]", 10);
    thread t2(func, "Thread 2", "@@", 20);
    t1.join();
    t2.join();
    cout << "exit main()";
    DisplayThreadId();
    return 0;
}
```

- Shared thread function
- Two simultaneous threads
- this_thread::**sleep()**
- C++ STL Numeric *literals*
- Console isn't thread "safe"

```
mutex
File  Edit  View  Terminal  Tabs  Help
enter main() {threadid = 140526266379456}
Thread 2 enter func() {threadid = 140526018377472}
@@Thread 1 enter func() {threadid = 140526026770176}
[]@@[]@@[]@@[]@@[]@@[]@@[]@@[]@@[]@@
Thread 1 exit func() {threadid = 140526026770176}
@@@@@@@@@@@@@@@@@@@@
Thread 2 exit func() {threadid = 140526018377472}
exit main() {threadid = 140526266379456}

Process returned 0 (0x0)   execution time : 0.038 s
Press ENTER to continue.
```

Mutex = Mutually Exclusive (only one owner)

Edit Lab 2
Mutex

main.cpp

```
1   #include "stdafx.h"
2
3   using namespace std;
4
5   mutex g_console_mutex;
6
7   void DisplayThreadId()
8   {
9       cout << " {threadid = "
10           << this_thread::get_id()
11           << "}" << endl;
12  }
13
14  void func(string msg, string symbol, int count)
15  {
16      g_console_mutex.lock();
17      cout << msg << " enter func()";
18      DisplayThreadId();
19      for (int i{}; i < count; ++i)
20      {
21          cout << symbol;
22          this_thread::sleep_for(1ns);
23      }
24      cout << endl << msg << " exit func()";
25      DisplayThreadId();
26      g_console_mutex.unlock();
27  }
28
```

- Declare global mutex variable
- This thread **locks** the mutex
- Perform assigned work
- Thread **unlocks** the mutex

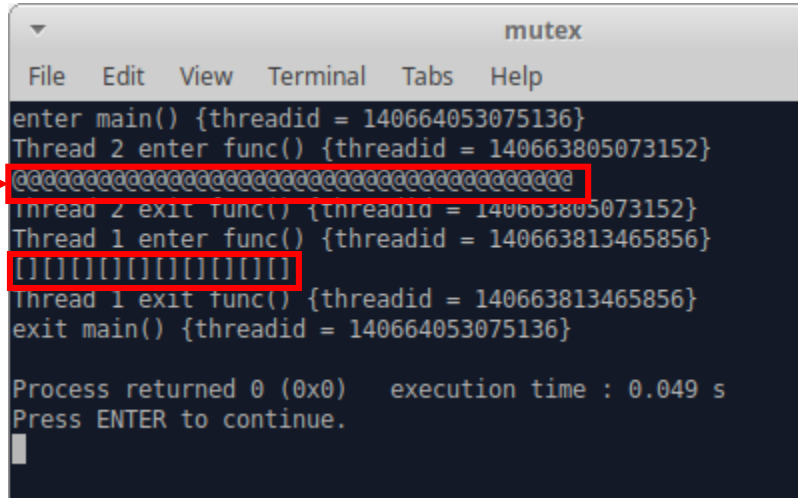
Mutex = Mutually Exclusive (only one owner)

Run Lab 2
Mutex

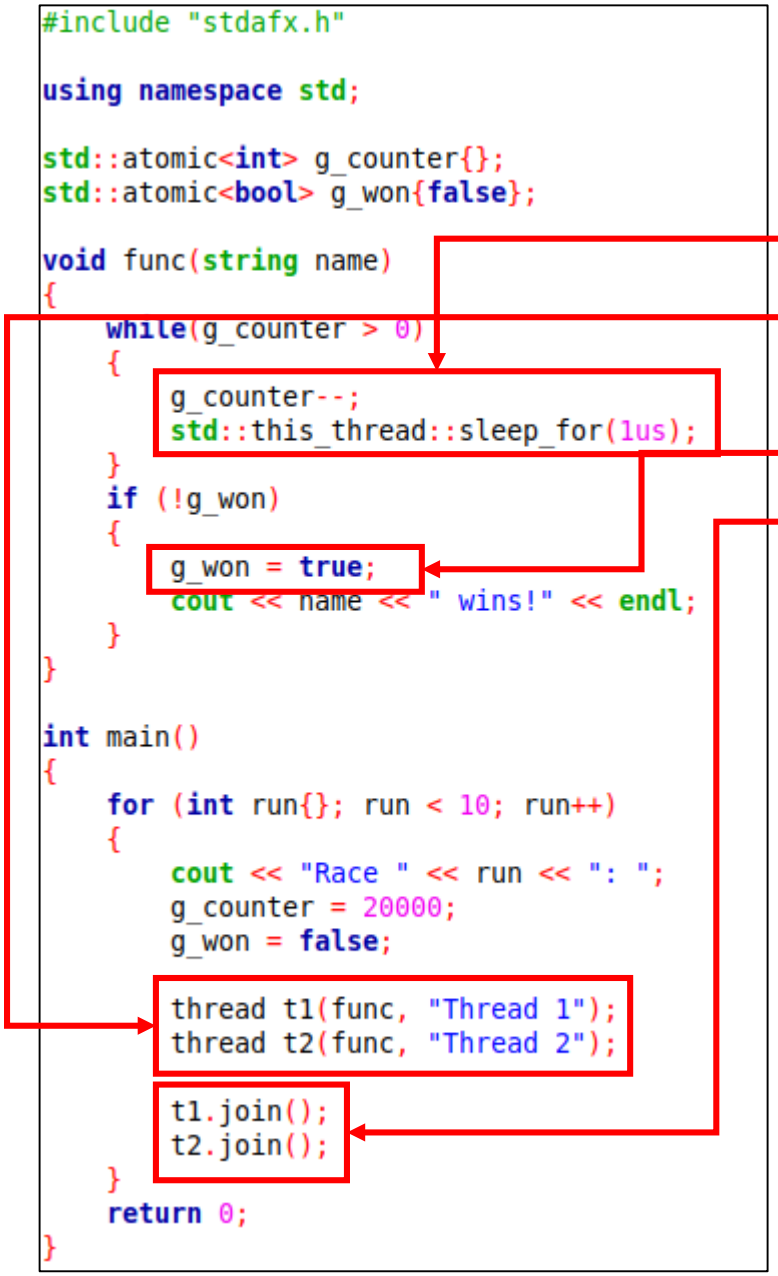
main.cpp

```
1  #include "stdafx.h"
2
3  using namespace std;
4
5  mutex g_console_mutex;
6
7  void DisplayThreadId()
8  {
9      cout << " {threadid = "
10         << this_thread::get_id()
11         << "}" << endl;
12 }
13
14 void func(string msg, string symbol, int count)
15 {
16     g_console_mutex.lock();
17     cout << msg << " enter func()";
18     DisplayThreadId();
19     for (int i{}; i < count; ++i)
20     {
21         cout << symbol;
22         this_thread::sleep_for(1ns);
23     }
24     cout << endl << msg << " exit func()";
25     DisplayThreadId();
26     g_console_mutex.unlock();
27 }
28
```

- Declare global mutex variable
- This thread **locks** the mutex
- Perform assigned work
- Thread **unlocks** the mutex
- Threads are now *synchronized*



Open Lab 3
Race Condition



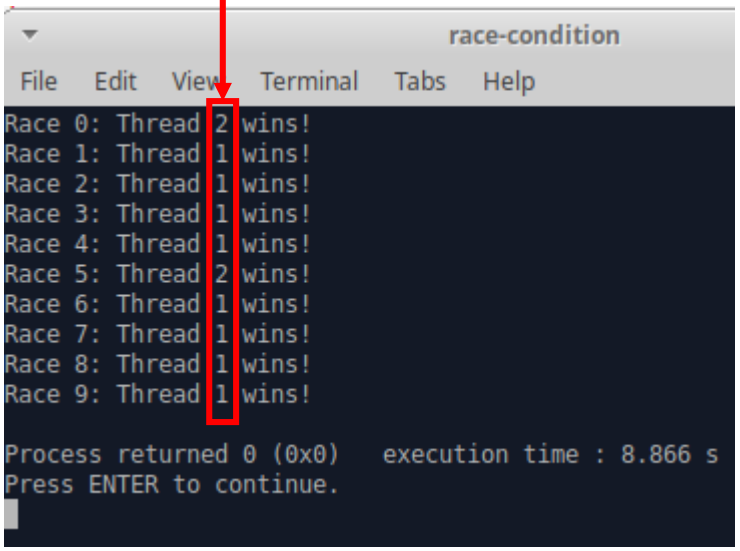
```cpp
#include "stdafx.h"

using namespace std;

std::atomic<int> g_counter{};
std::atomic<bool> g_won{false};

void func(string name)
{
    while(g_counter > 0)
    {
        g_counter--;
        std::this_thread::sleep_for(1us);
    }
    if (!g_won)
    {
        g_won = true;
        cout << name << " wins!" << endl;
    }
}

int main()
{
    for (int run{}; run < 10; run++)
    {
        cout << "Race " << run << ": ";
        g_counter = 20000;
        g_won = false;

        thread t1(func, "Thread 1");
        thread t2(func, "Thread 2");

        t1.join();
        t2.join();
    }
    return 0;
}
```

- Two simultaneous threads
- Decrement a global counter
- First to reach zero wins
- Wait for both threads to end

```cpp
#include "stdafx.h"

using namespace std;

std::atomic<int> g_counter{};
std::atomic<bool> g_won{false};

void func(string name)
{
    while(g_counter > 0)
    {
        g_counter--;
        std::this_thread::sleep_for(1us);
    }
    if (!g_won)
    {
        g_won = true;
        cout << name << " wins!" << endl;
    }
}

int main()
{
    for (int run{}; run < 10; run++)
    {
        cout << "Race " << run << ": ";
        g_counter = 20000;
        g_won = false;

        thread t1(func, "Thread 1");
        thread t2(func, "Thread 2");

        t1.join();
        t2.join();
    }
    return 0;
}
```

Run Lab 3
Race Condition

- Two simultaneous threads
- Decrement a global counter
- First to reach zero wins
- Result is non-deterministic
- This is a *Race Condition*



Open Lab 4 : Non-Atomic Operations

1. Implement a C++ multithreaded console program that performs **10** runs of the same experiment
2. Each run of the experiment should first reset a **global counter** to **zero**, and then launch **20** threads
3. Each thread should single increment (++) the global counter **50,000** times
4. Once all **20** threads complete, the program should display the global counter value for that particular run
5. With **20** threads incrementing the global counter **50,000** times, the final value for each run *should be* **1,000,000**
6. Once all **10** runs complete, the program should end

Open Lab 4 : Non-Atomic Operations

```
14  void func()
15  {
16      //g_console_mutex.lock();
17      for (int i{}; i < 50000; i++) {
18          g_counter++;
19      }
20      //g_console_mutex.unlock();
21  }
```

Increment the global counter 50,000 times

```
23  int main()
24  {
25      cout.imbue(std::locale(""));
26
27      for (int i{}; i < 10; ++i) {
28          g_counter = 0;
29          vector<thread> threadPool;
30          for (size_t t{}; t < 20; ++t)
31              threadPool.push_back(
32                  thread(func));
33          for (auto& t : threadPool)
34              t.join();
35          cout << "Run " << i << ": "
36               << "Counter = " << (int)g_counter
37               << endl;
38      }
39      return 0;
40  }
41
```

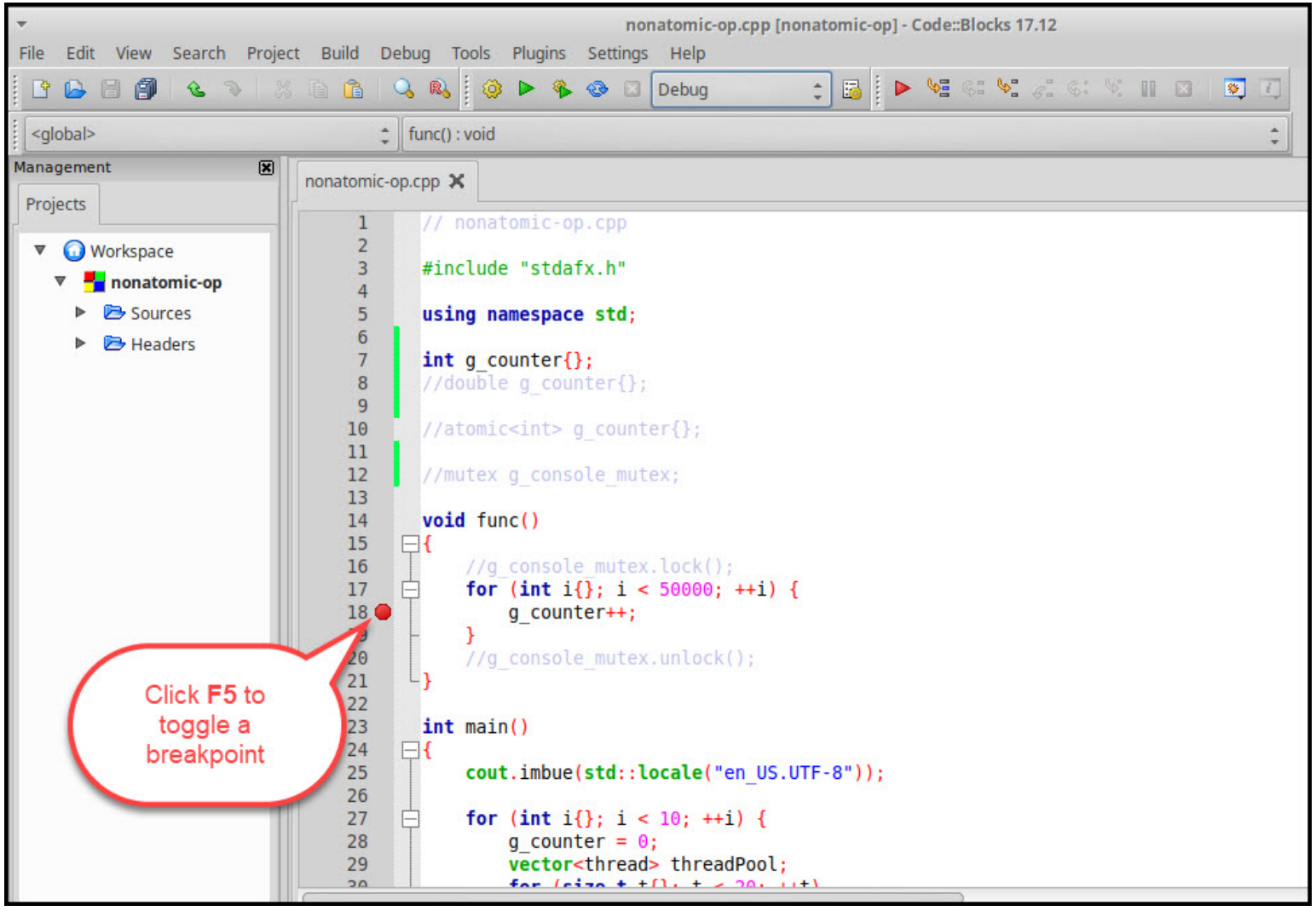
10 runs

20 threads

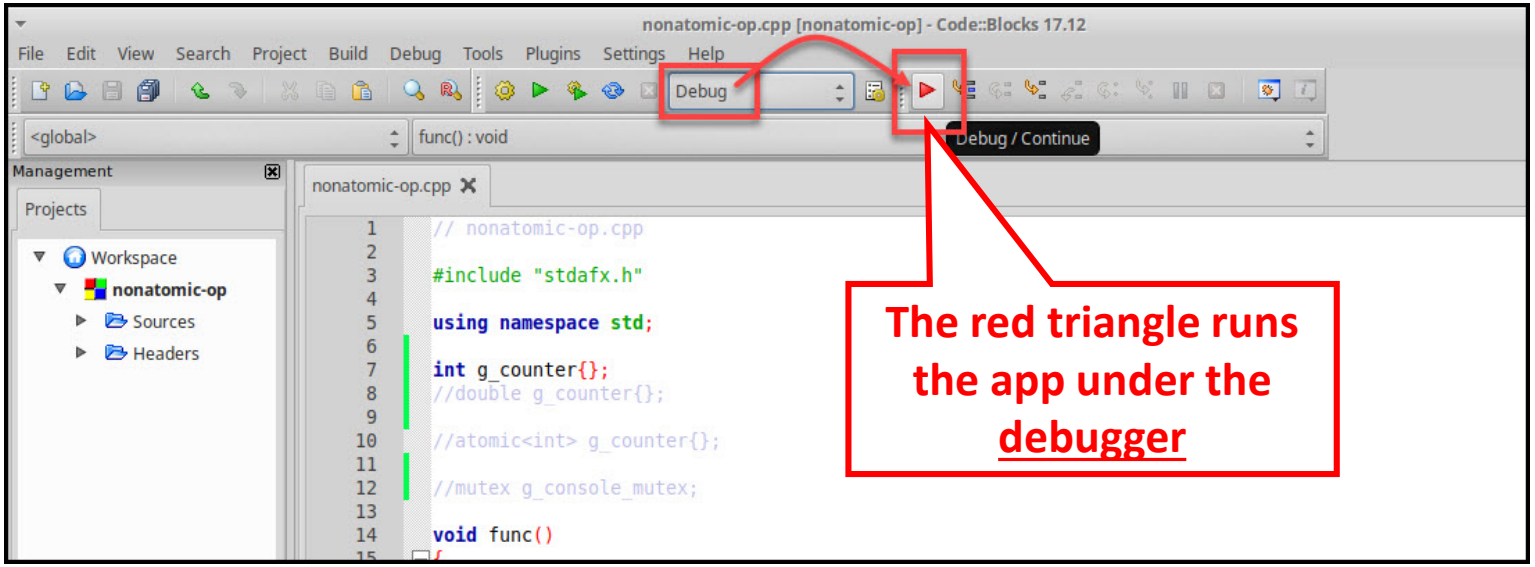
Wait for all threads to end

Display the global counter value

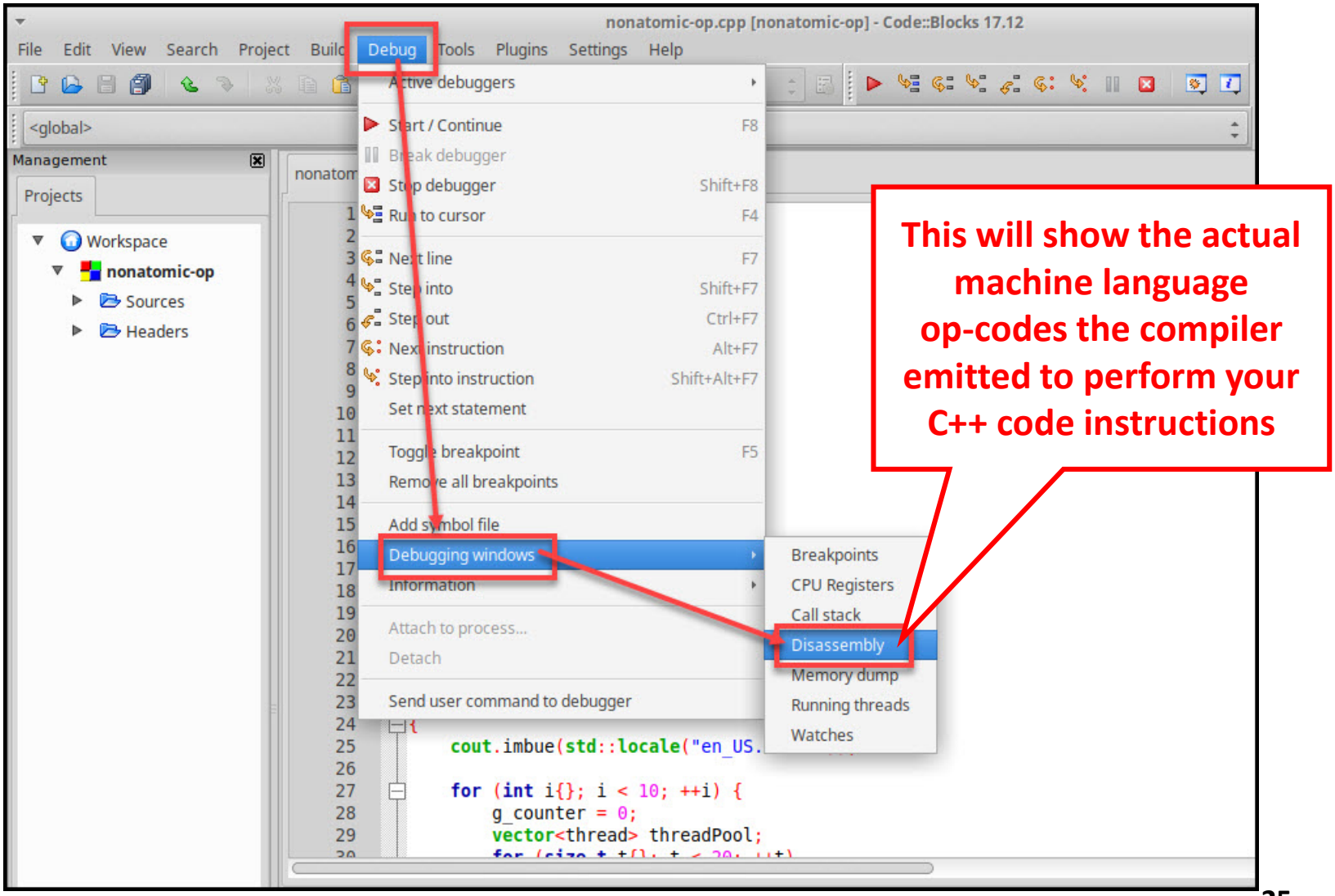
Edit Lab 4 : Non-Atomic Operations



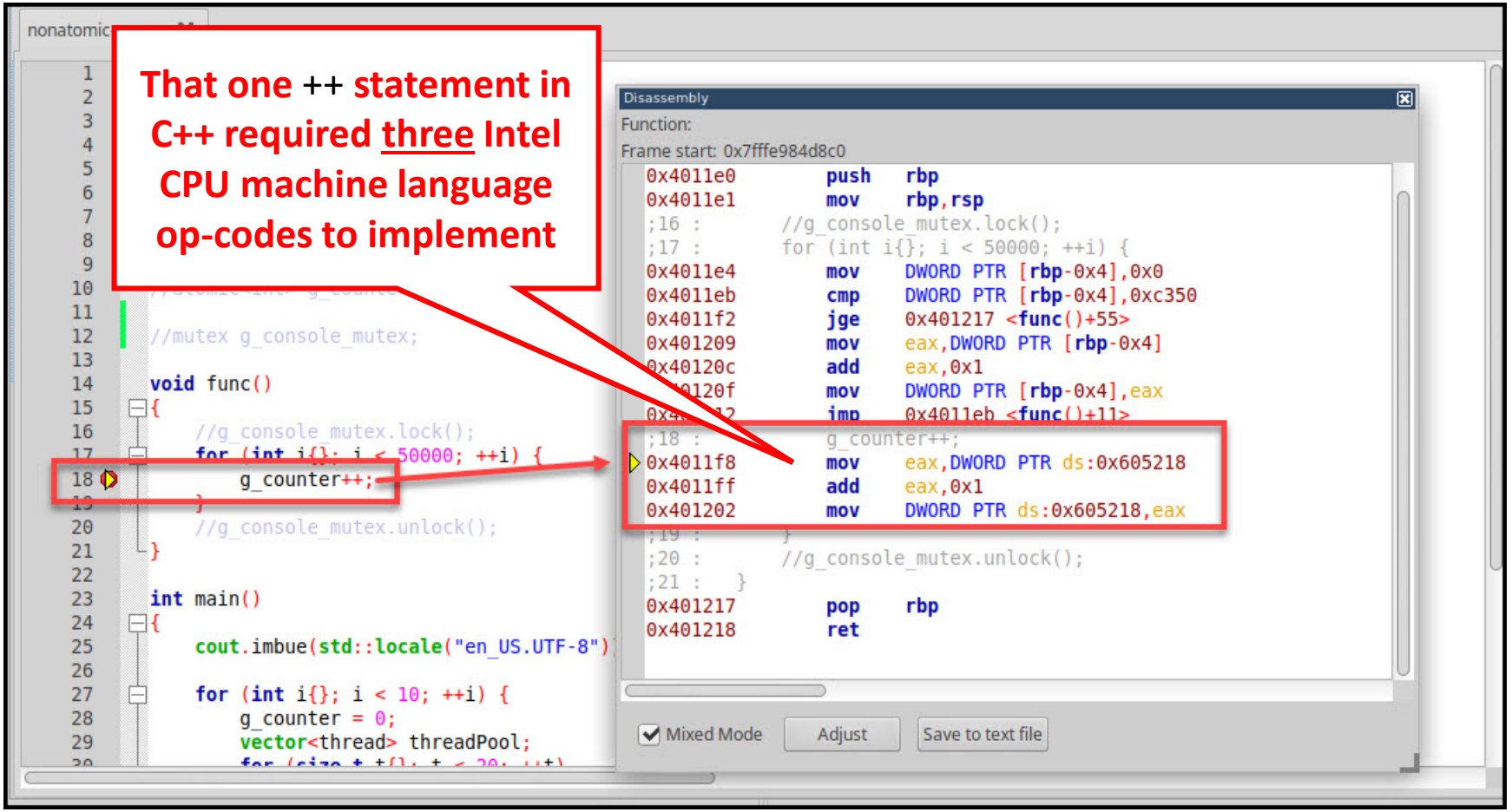
Run Lab 4 : Non-Atomic Operations



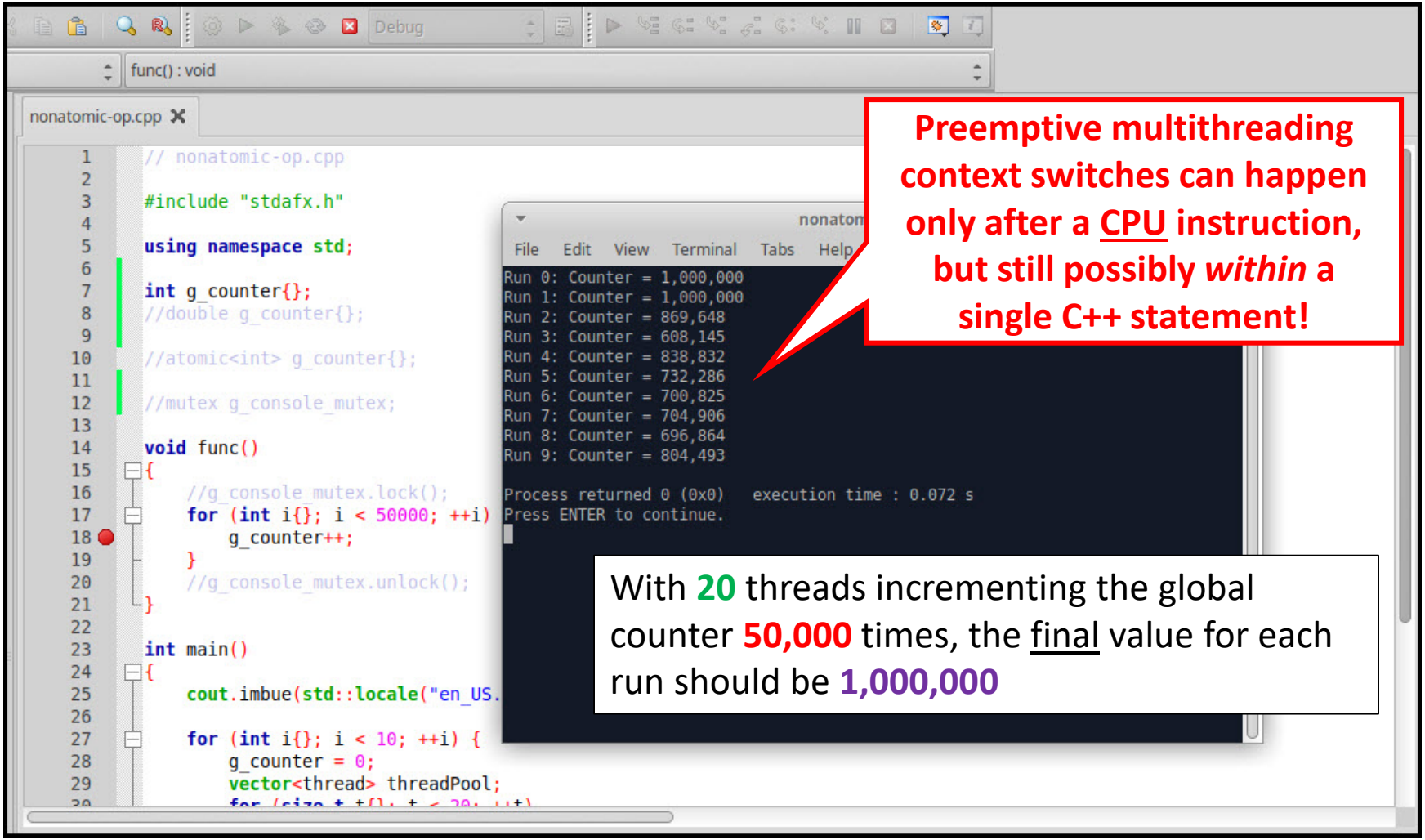
Run Lab 4 : Non-Atomic Operations



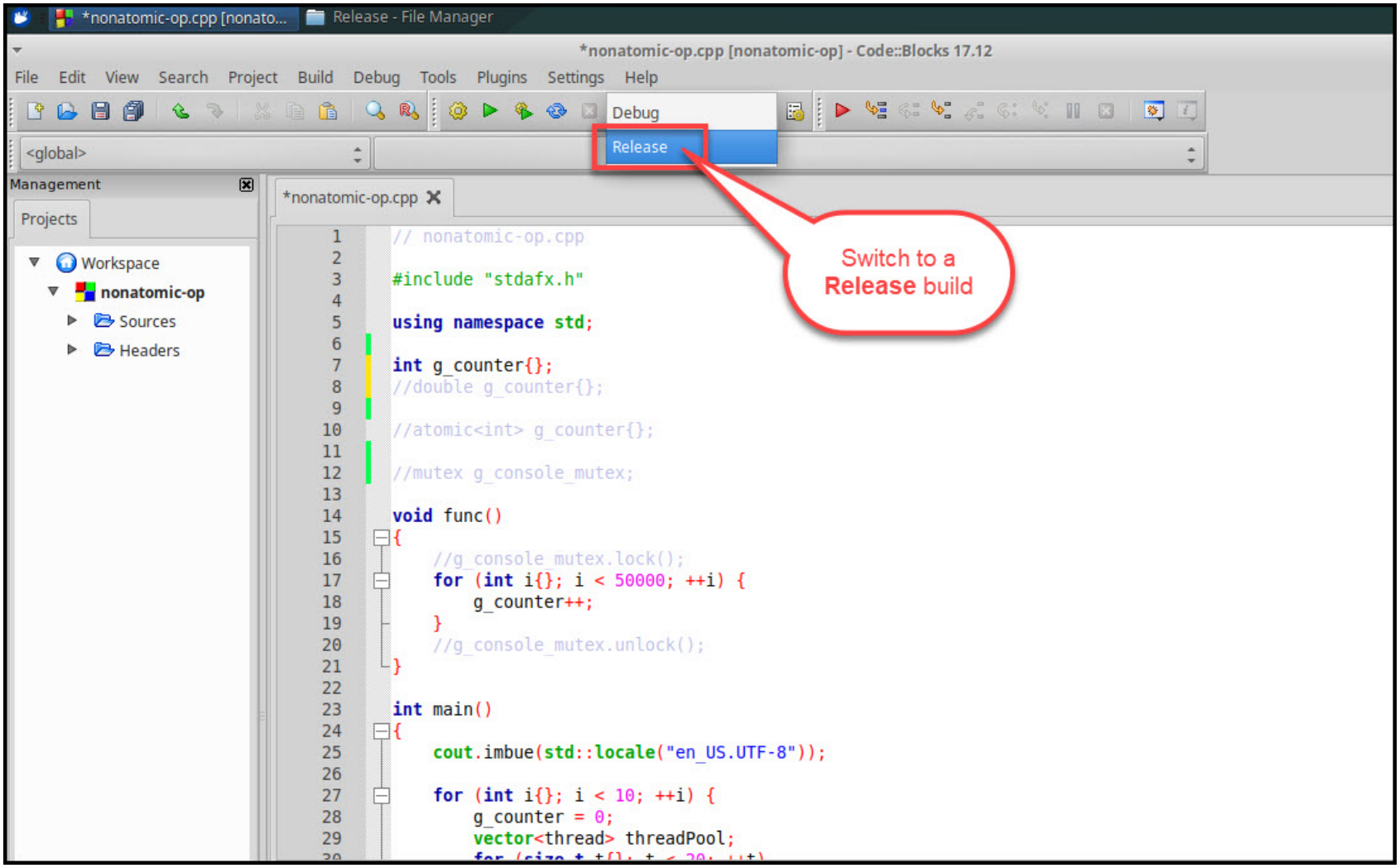
Run Lab 4 : Non-Atomic Operations



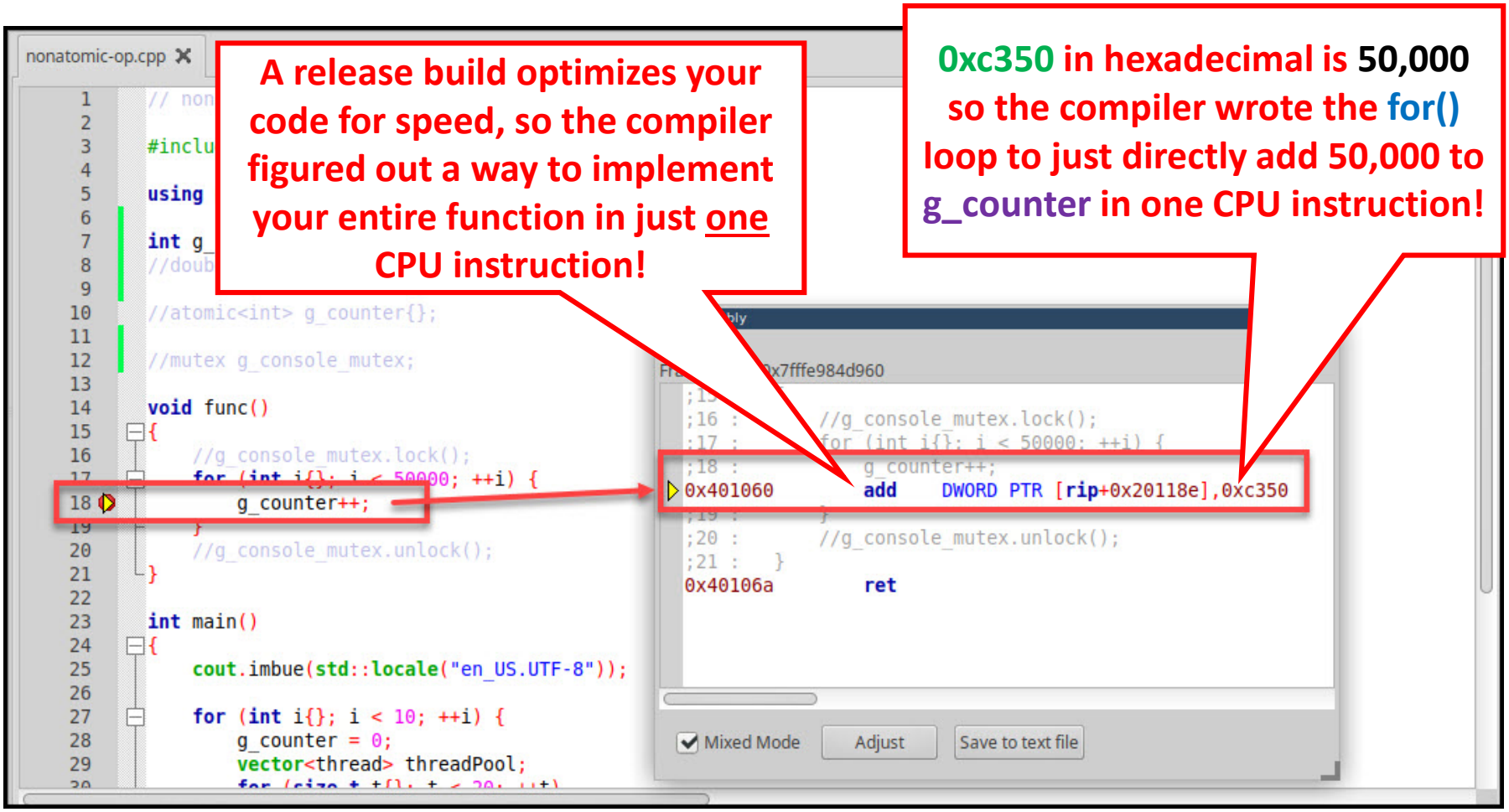
Run Lab 4 : Non-Atomic Operations



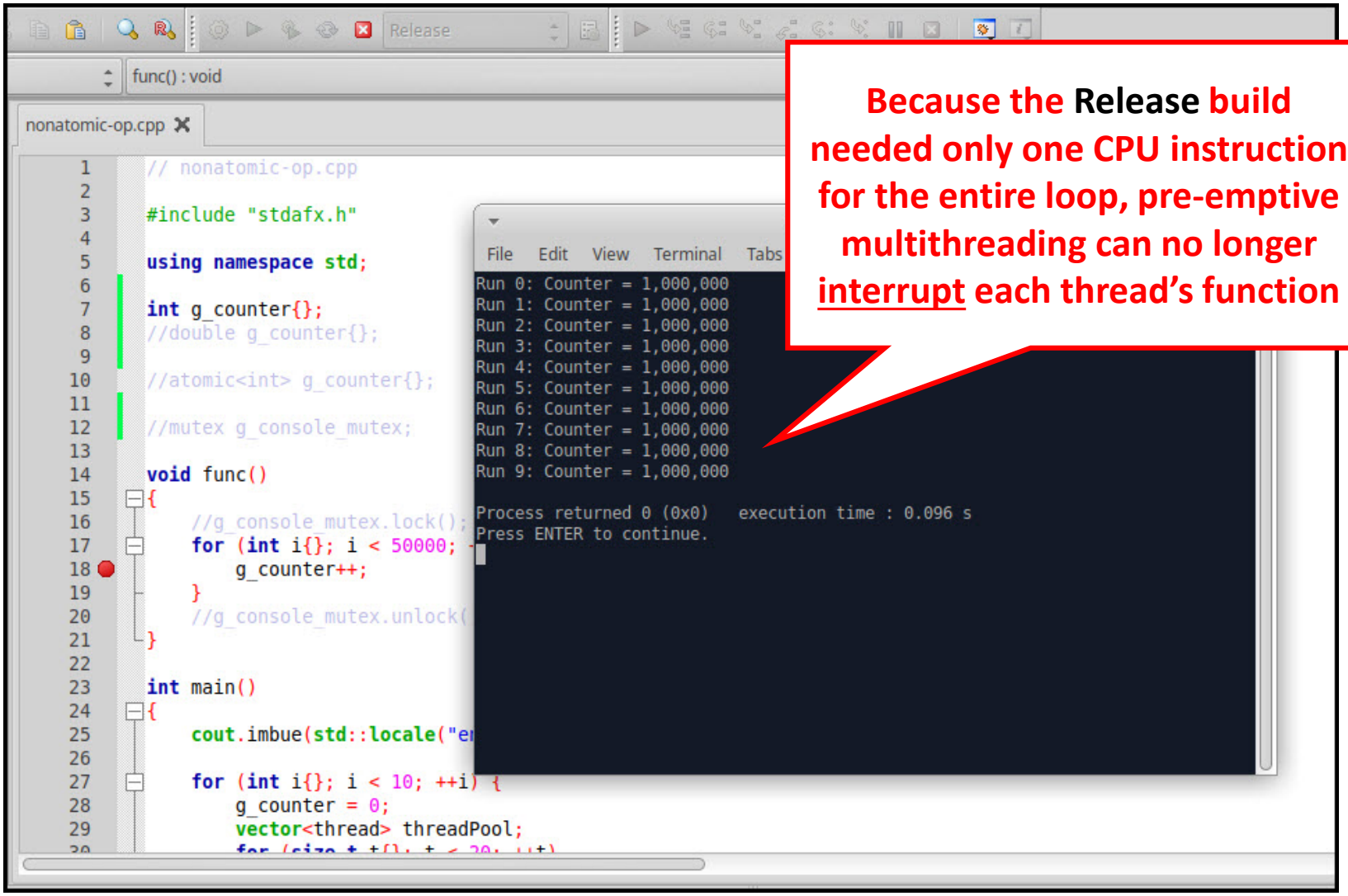
Edit Lab 4 : Non-Atomic Operations



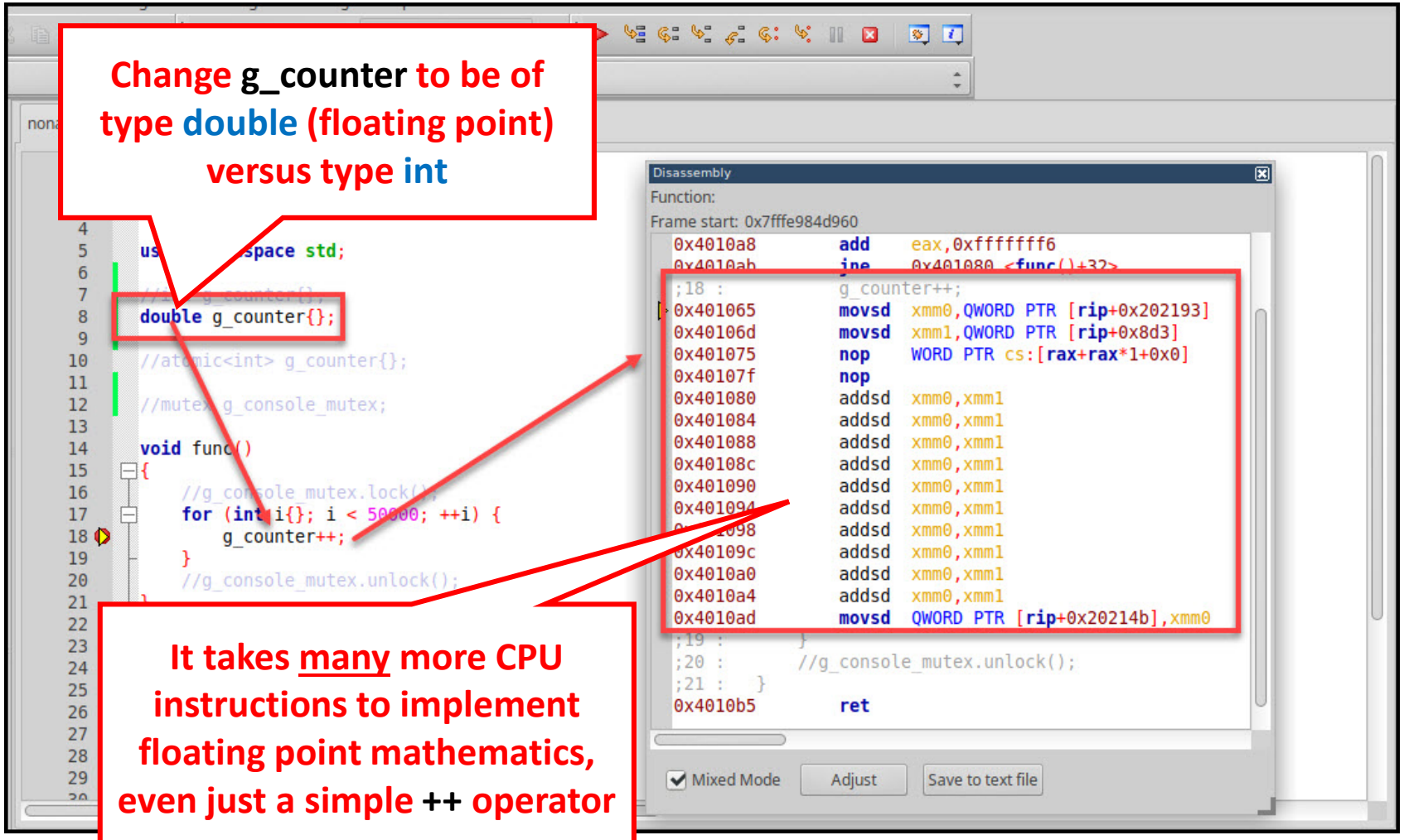
Run Lab 4 : Non-Atomic Operations



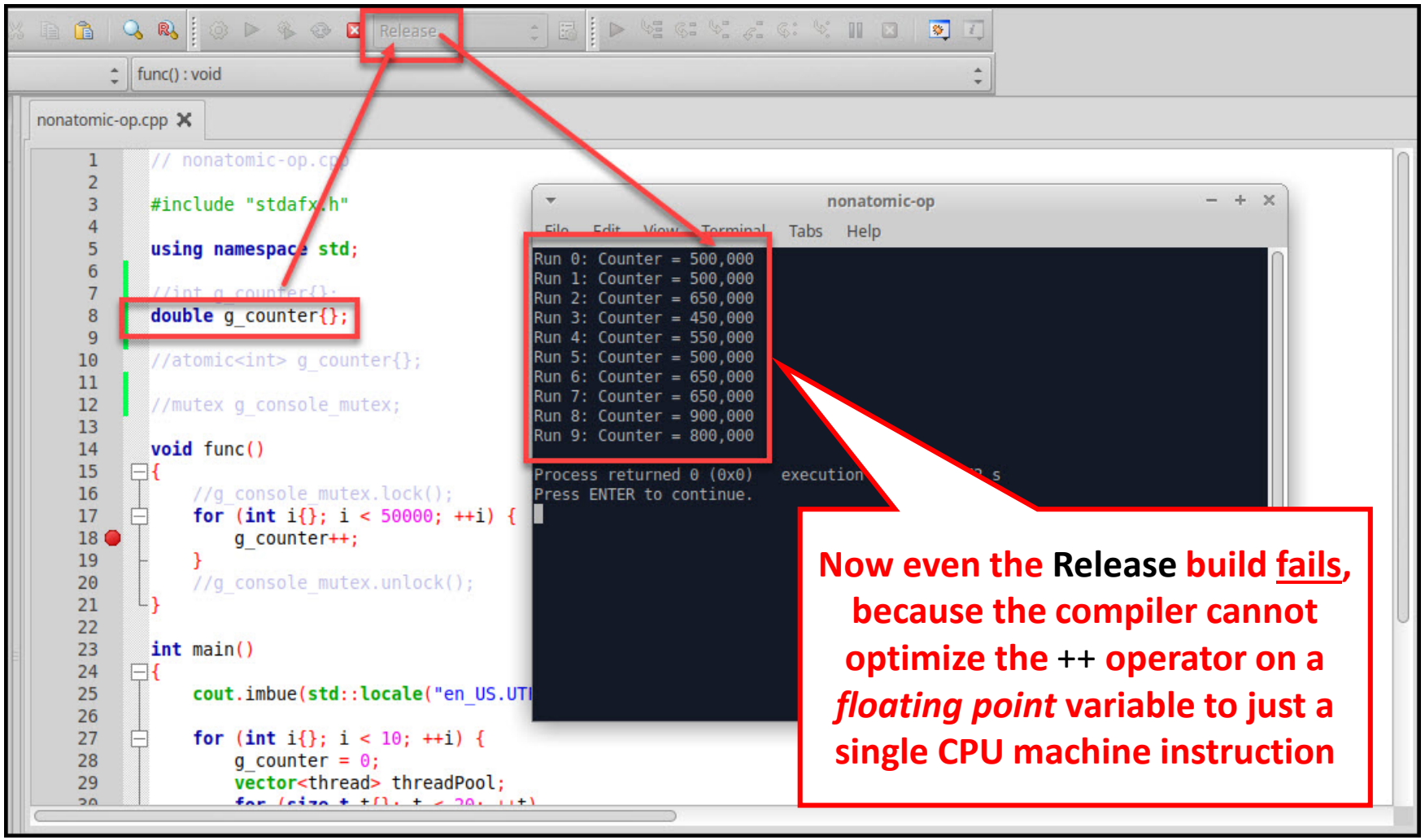
Run Lab 4 : Non-Atomic Operations



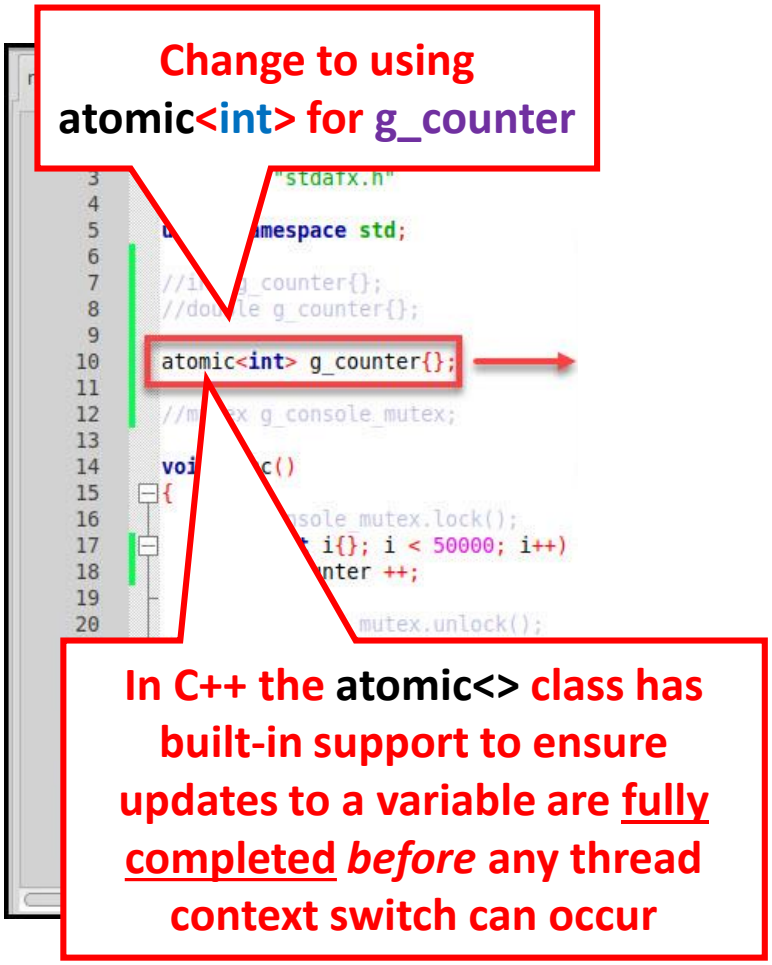
Edit Lab 4 : Non-Atomic Operations



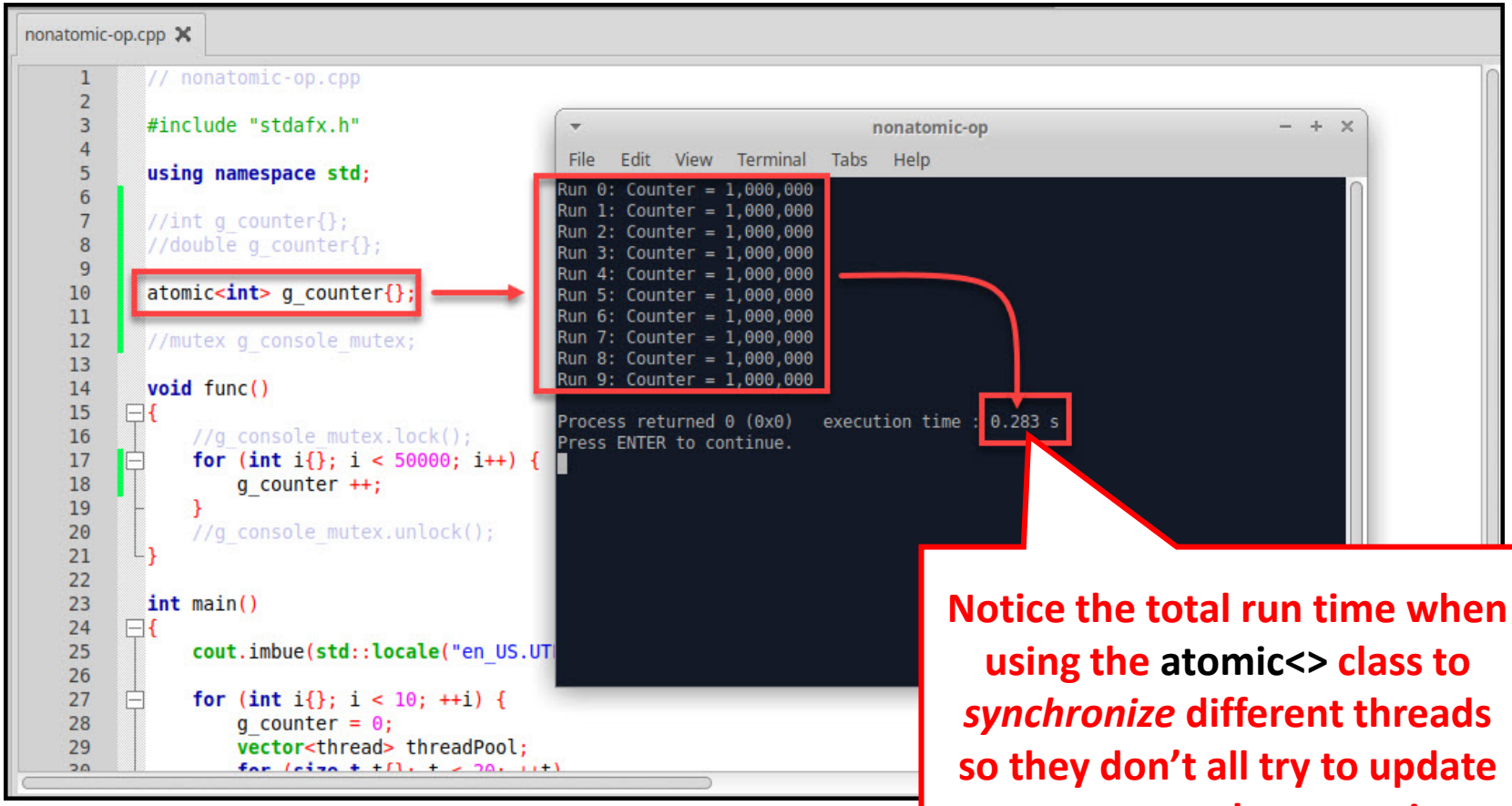
Run Lab 4 : Non-Atomic Operations



Edit Lab 4 : Non-Atomic Operations



Run Lab 4 : Non-Atomic Operations



Notice the total run time when using the atomic<> class to *synchronize* different threads so they don't all try to update g_counter at the same time

Edit Lab 4 : Non-Atomic Operations

nonatomic-op.cpp

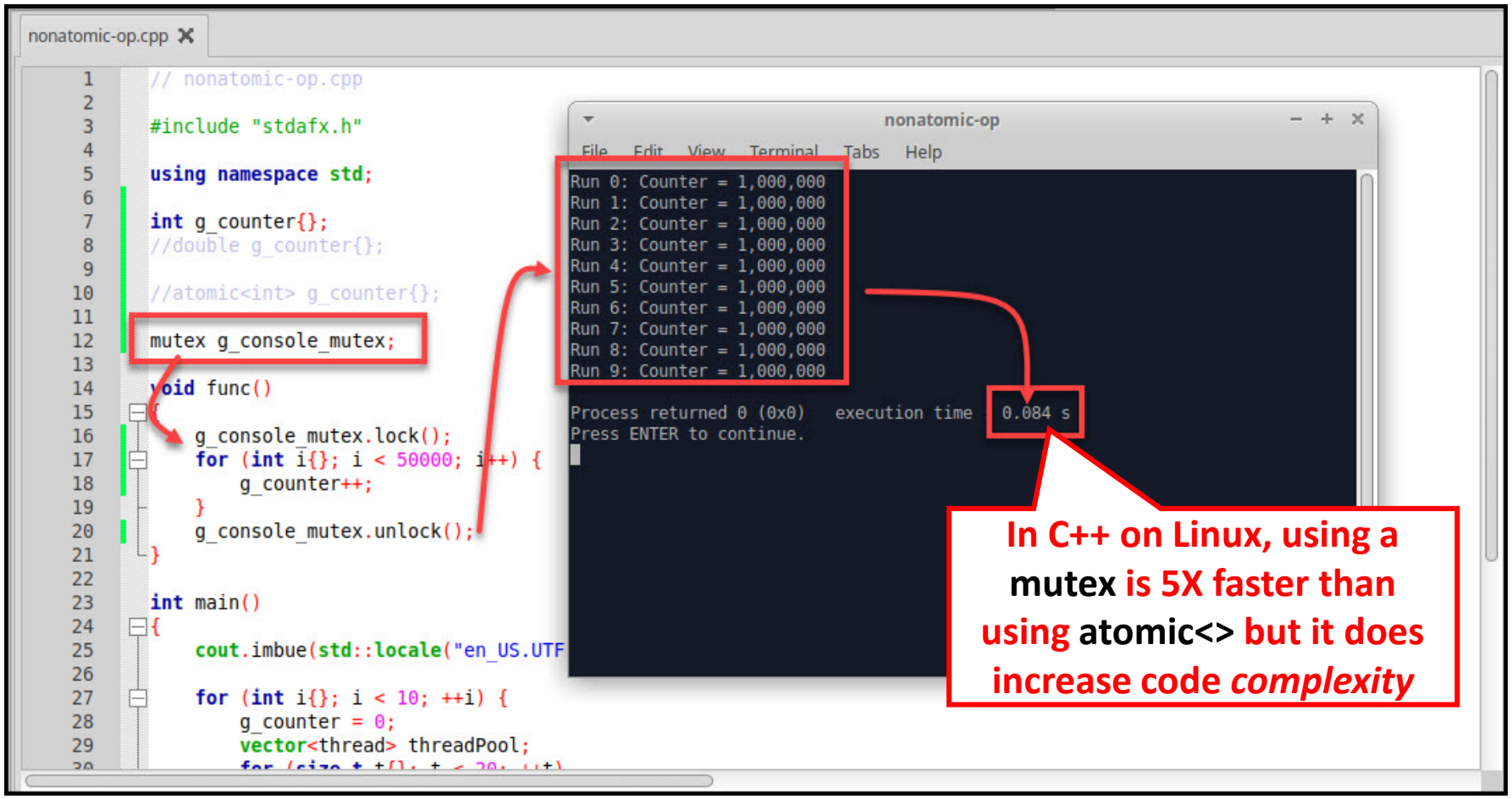
```
1   // nonatomic-op.cpp
2
3   #include "stdafx.h"
4
5   using namespace std;
6
7   int g_counter{};
8   //double g_counter{};
9
10  //atomic<int> g_counter{};
11
12  mutex g_console_mutex;
13
14  void func()
15  {
16      g_console_mutex.lock();
17      for (int i{}; i < 50000;
18          g_counter++;
19      }
20      g_console_mutex.unlock();
21  }
22
23  int main()
24  {
25      cout.imbue(std::locale("e
26
27      for (int i{}; i < 10; ++i
28          g_counter = 0;
29          vector<thread> thread
```

Switch back to a regular int type for g_counter

Declare a mutex

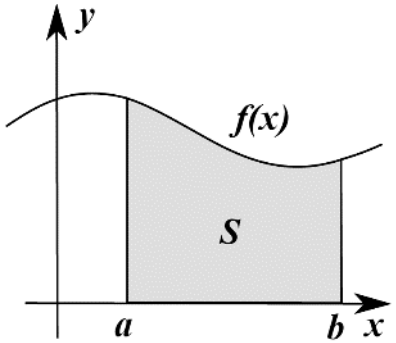
Implement the mutex lock() and unlock() to guard against simultaneous edits to g_counter by other threads

Run Lab 4 : Non-Atomic Operations



Why do we need integrals?

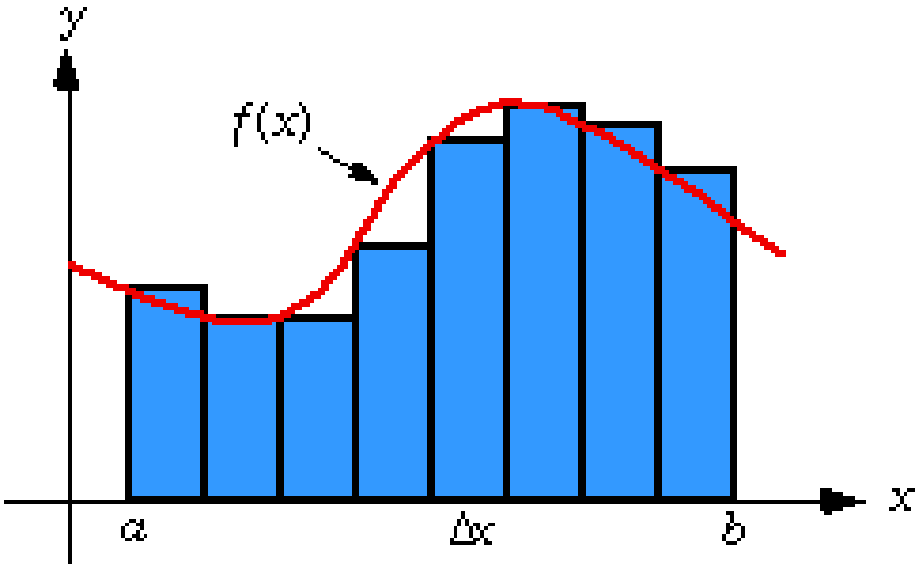
- The **integral** of a function can be defined as the **area under a curve** f(x) within the region [a,b]



- There are ways to often determine exactly the value of the integral of **f(x)** which we would write $\mathbf{F(x)} = \int_a^b \boldsymbol{f(x)}$
- However, sometimes it is not possible to find an analytic expression for **F(x)** – so we use ***numerical* integration**

Riemann Sums

- One way we can integrate f(x) is to divide the area under the curve into strips (**intervals**) and sum the area of each strip
- This estimate may not be totally accurate because we might have **gaps** between the true value of f(x) and the top of a strip

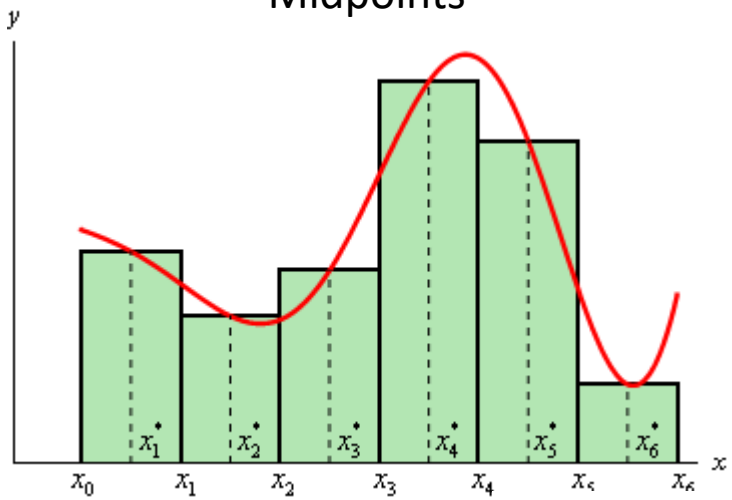


Riemann Sums

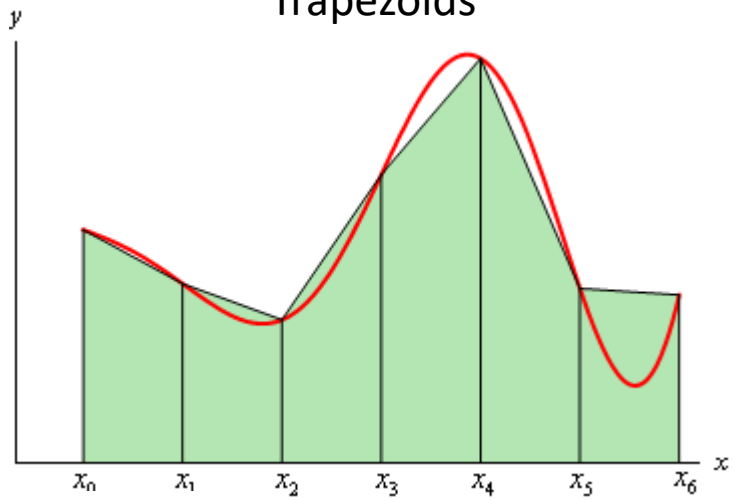
- The width of each strip is $\boldsymbol{\Delta x} = \frac{(\boldsymbol{b}-\boldsymbol{a})}{\boldsymbol{\#\ of\ intervals}}$
- We can minimize the gaps by increasing the number of intervals, which makes the $\boldsymbol{\Delta x}$ ***smaller***
- There are different strategies for determine the shape and height of each strip
  - Left-hand Rule, Right-hand Rule, Midpoint Rule
  - Fit Trapezoids
  - Fit Parabolas (Simpson's Rule)
- Depending upon the particular shape of $\boldsymbol{f}(\boldsymbol{x})$, one method might be more <u>accurate</u> than the others

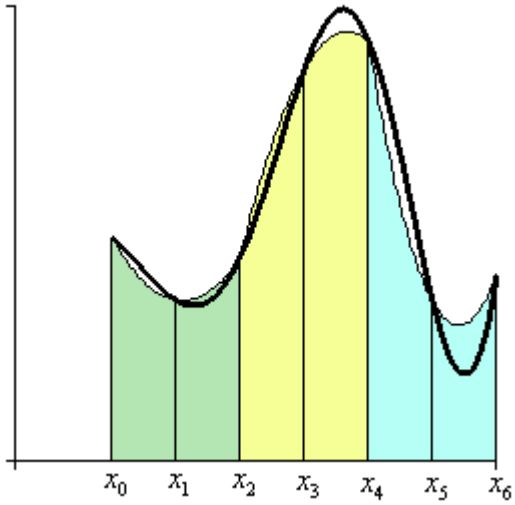
Riemann Sums

Midpoints



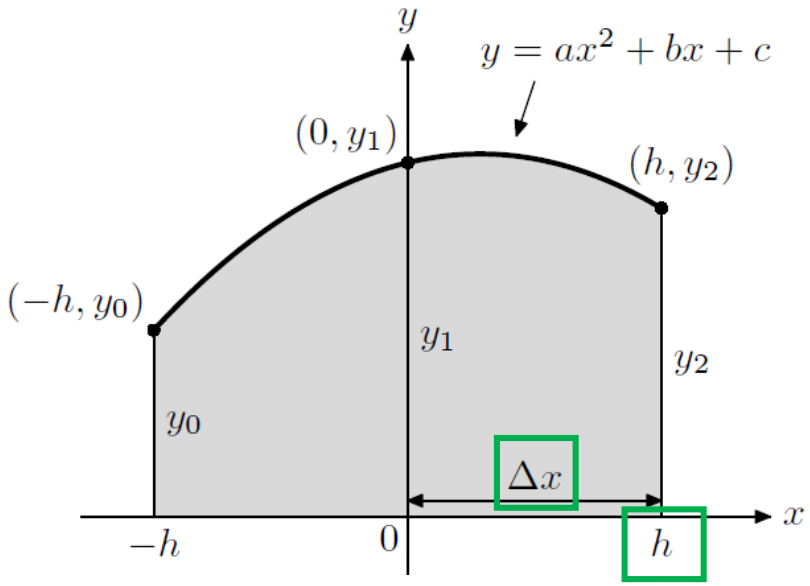
Trapezoids





Parabolas
(Simpson's Rule)

Simpson's Rule is more accurate!



$$A = \int_{-h}^{h} (ax^2 + bx + c)\, dx$$

$$= \left(\frac{ax^3}{3} + \frac{bx^2}{2} + cx \right) \Bigg|_{-h}^{h}$$

$$= \frac{2ah^3}{3} + 2ch$$

$$= \frac{h}{3}\left(2ah^2 + 6c\right)$$

$$A = \frac{h}{3}(y_0 + 4y_1 + y_2) = \frac{\Delta x}{3}(y_0 + 4y_1 + y_2)$$

$$y_0 = ah^2 - bh + c$$

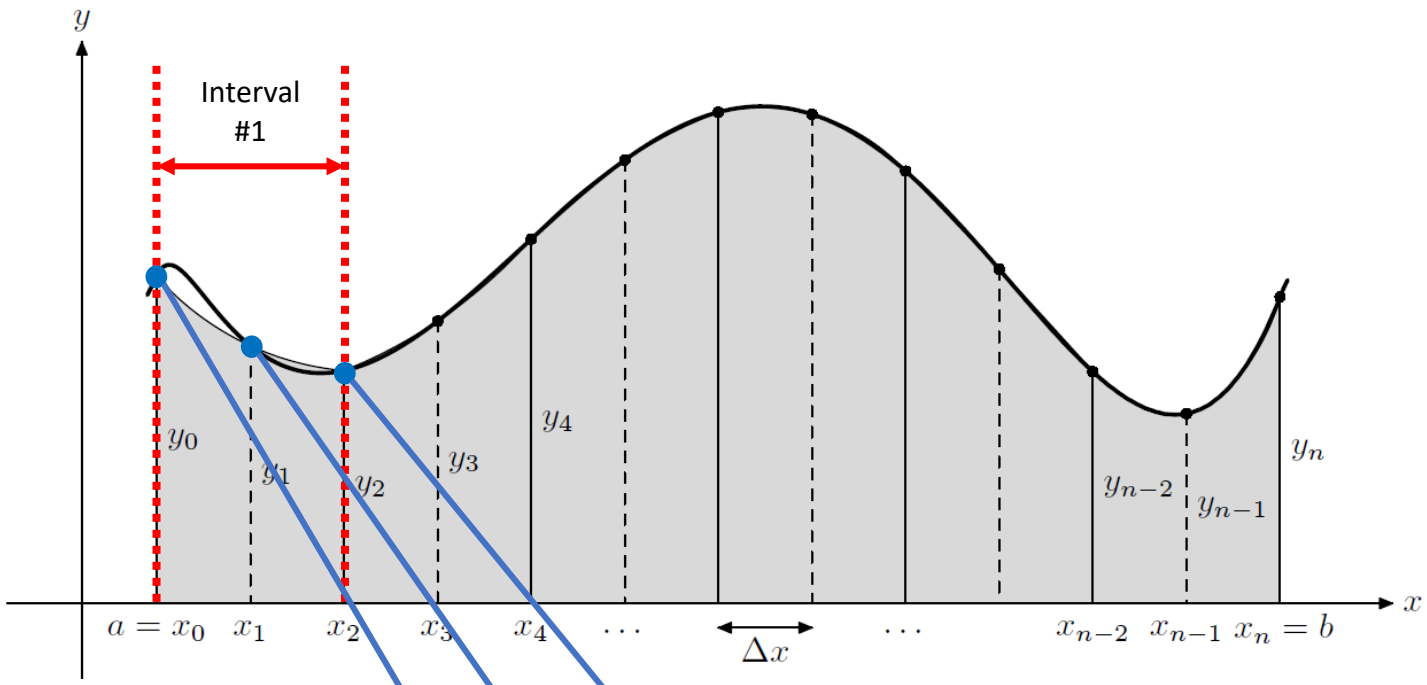
$$y_1 = c$$

$$y_2 = ah^2 + bh + c$$

$$y_0 + 4y_1 + y_2 = (ah^2 - bh + c) + 4c + (ah^2 + bh + c) = 2ah^2 + 6c$$

Simpson's Rule is more accurate!

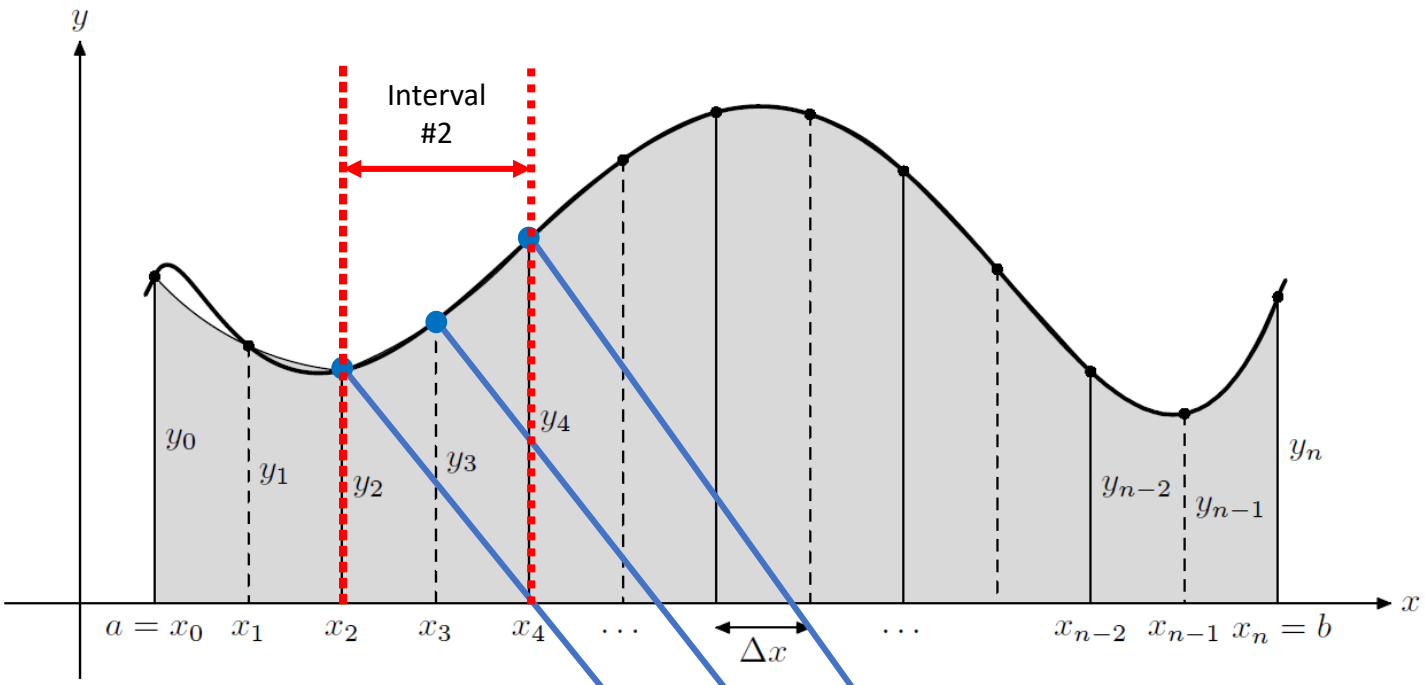
$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \ldots, \quad y_n = f(x_n).$$



$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\left(\boxed{y_0} + \boxed{4y_1} + \boxed{1y_2} + \cdots\right)$$

Simpson's Rule is more accurate!

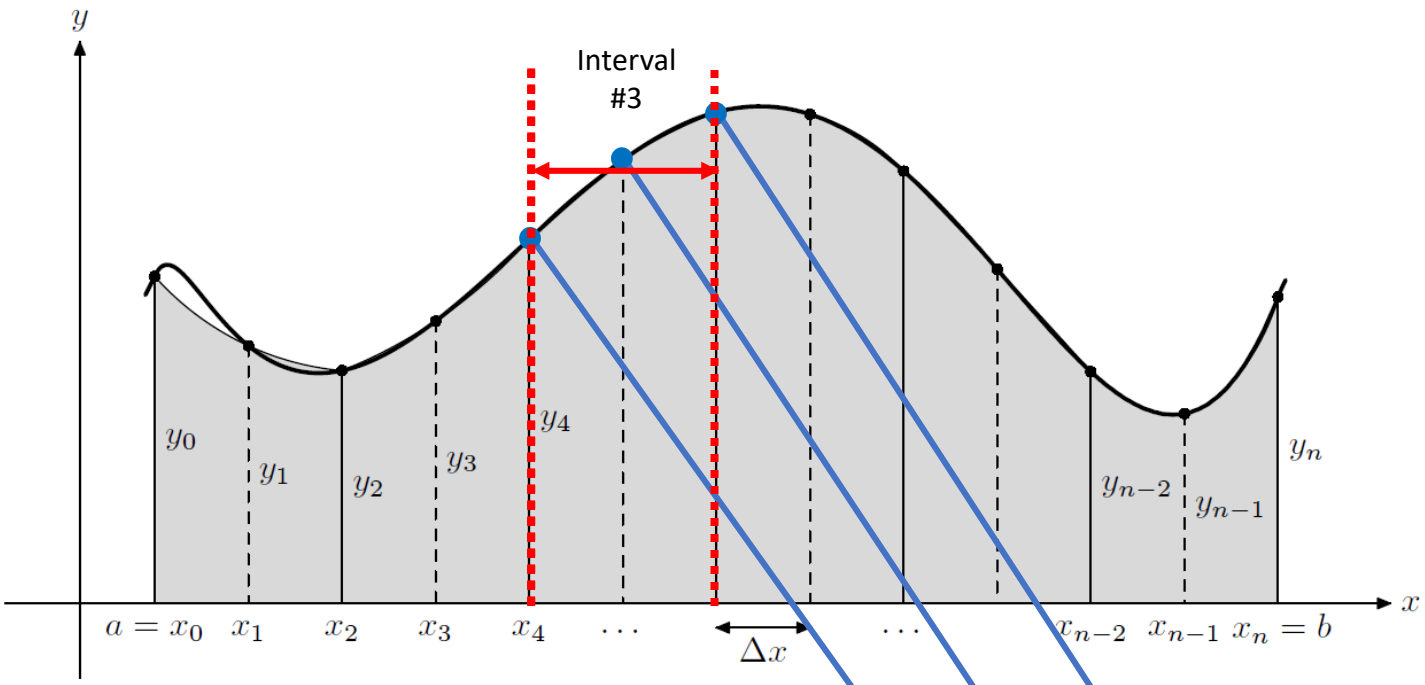
$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \ldots, \quad y_n = f(x_n).$$



$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\,(y_0 + 4y_1 + \boxed{2y_2} + \boxed{4y_3} + \boxed{1y_4} \ldots)$$

Simpson's Rule is more accurate!

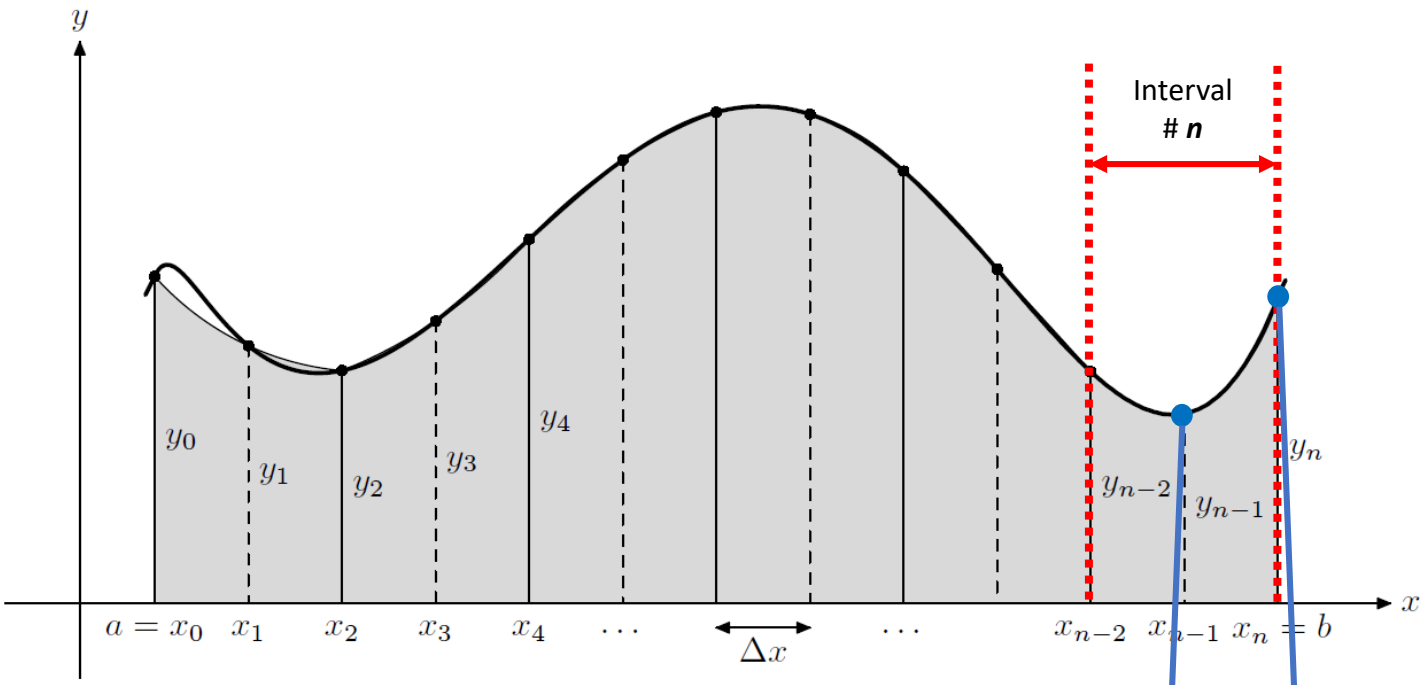
$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \ldots, \quad y_n = f(x_n).$$



$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\,(y_0 + 4y_1 + 2y_2 + 4y_3 + \boxed{2y_4} + \boxed{4y_5} + \boxed{1y_6} + \cdots)$$

Simpson's Rule is more accurate!

$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \ldots, \quad y_n = f(x_n).$$



$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\left(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + \boxed{4y_{n-1}} + \boxed{y_n}\right)$$

Simpson's Rule is more accurate!

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\left(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 4y_{n-1} + y_n\right)$$

Point	y0	y1	y2	y3	y4	y5	y6
Coeff	1	4	2	4	2	4	1

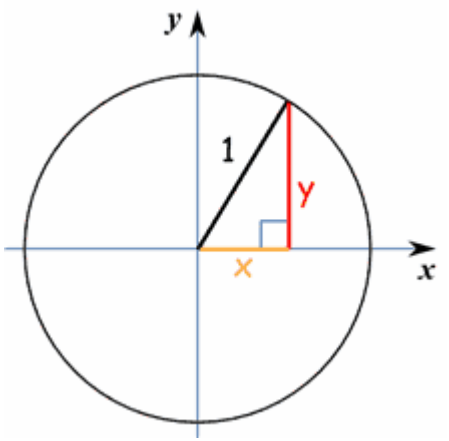
The first and last point have coefficient = 1
Every point with an odd index has coefficient = 4
Every point with an even index has coefficient = 2

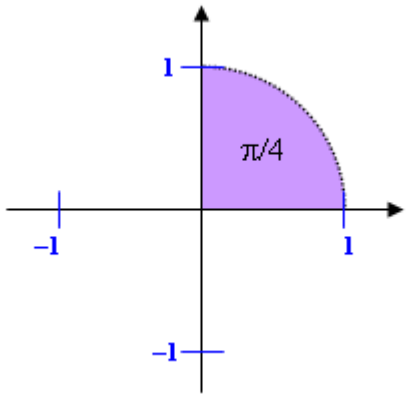
```
void simpsons(TCB* tcb)
{
    double sum = f(tcb->b) + f(tcb->a);
    double deltaX = (tcb->b - tcb->a) / tcb->intervals;
    tcb->a += deltaX;
    for (int i=1; i < tcb->intervals; i++) {
        int coeff = 2 * (i % 2 + 1);
        double y=f(tcb->a);
        if (!isnan(y)) sum += coeff*y;
        tcb->a += deltaX;
    }
    tcb->integral = (deltaX / 3) * sum;
}
```

Open Lab 5 – Parallel Simpson's Rule

Numerically calculate to **12** significant digits the value of **4** × **the area of a unit circle** in the **first quadrant** using Simpson's Rule

$$F(x) = 4\int_0^1 \sqrt{1-x^2}\,dx = \boldsymbol{\pi}$$





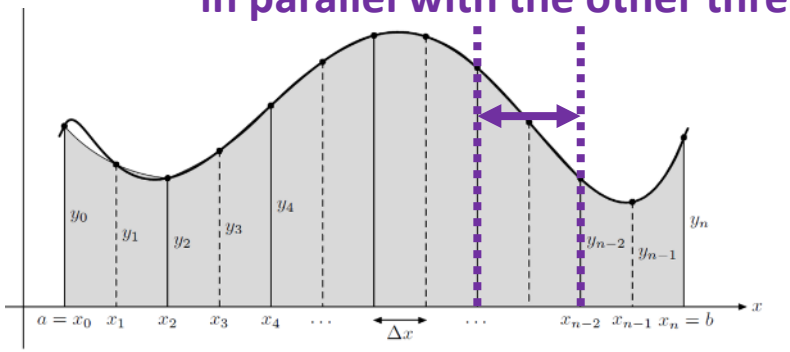
Open Lab 5 – Parallel Simpson's Rule

```
struct TCB {
    std::thread* t = nullptr;
    double a{};
    double b{};
    long intervals{};
    double integral{};
};
```

- Every thread has its own thread control block (**TCB**)
- Each **TCB** stores all the information its thread needs to compute **its portion** of the integral

```
inline double f(double x)
{
    return sqrt(1 - pow(x, 2));
}

void simpsons(TCB* tcb)
{
    double sum = f(tcb->b) + f(tcb->a);
    double deltaX = (tcb->b - tcb->a) / tcb->intervals;
    tcb->a += deltaX;
    for (int i=1; i < tcb->intervals; i++) {
        int coeff = 2 * (i % 2 + 1);
        double y=f(tcb->a);
        if (!isnan(y)) sum += coeff*y;
        tcb->a += deltaX;
    }
    tcb->integral = (deltaX / 3) * sum;
}
```

Each thread computes its own slice in parallel with the other threads



```
for (int threads{1}; threads <= max_threads * 4; threads *= 2) {

    auto startTime = chrono::steady_clock::now();

    double interval_width = (g_b - g_a) / threads;

    // Each thread gets its own
    // thread control block (TCB)
    vector<TCB*> threadPool;

    for (int i{}; i < threads; i++) {
        TCB* tcb = new TCB();
        tcb->a = i * interval_width;
        tcb->b = tcb->a + interval_width;
        tcb->intervals = 134217728 / threads;  //2^27
        tcb->t = new std::thread(simpsons, tcb);
        // New thread is now running, so add its TCB
        threadPool.push_back(tcb);
    }

    double integral{};
    for (auto& tcb : threadPool)
    {
        // Wait here for this thread to finish
        tcb->t->join();
        // Now we read this thread's result
        integral += tcb->integral;
        delete tcb->t;
     }
     threadPool.clear();

    auto endTime = chrono::steady_clock::now();

    auto totalTime = chrono::duration_cast<chrono::milliseconds>
        (endTime - startTime).count();

    cout << setw(15) << fixed << setprecision(11)
         << integral * 4.0
         << setw(10) << threads
         << setw(10) << fixed << setprecision(3)
         << totalTime << "(ms)" << endl;
}
```

```
int max_threads = thread::hardware_concurrency();
```

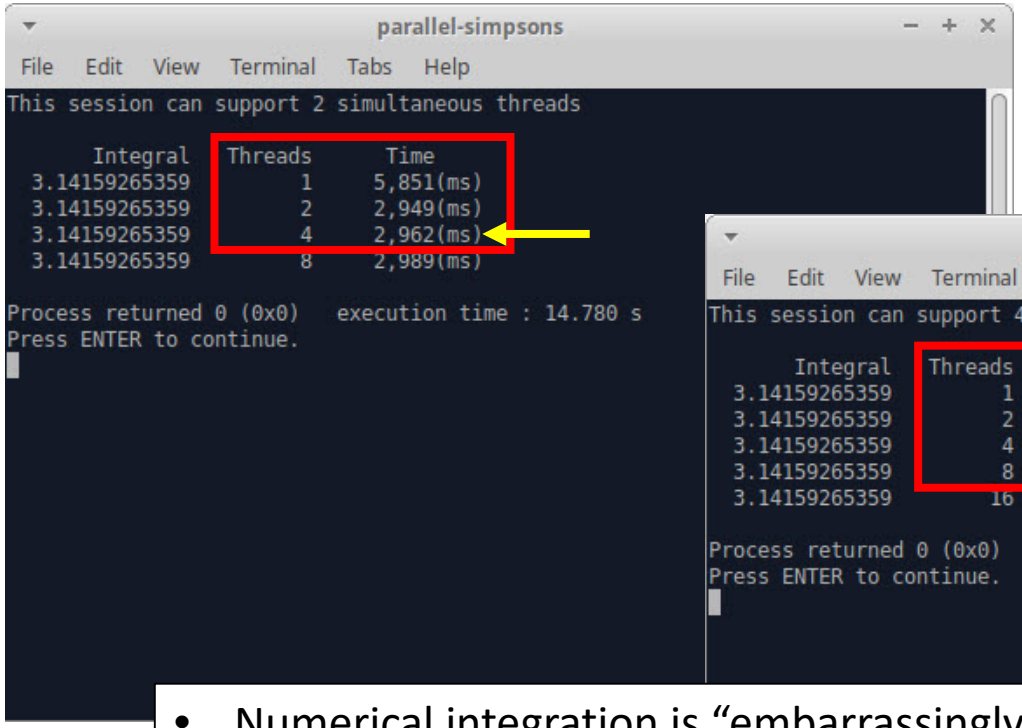
$$4 \int_{a=0}^{b=1} \sqrt{1 - x^2}\, dx$$

Given $\boldsymbol{n} = \#\ of\ threads$:

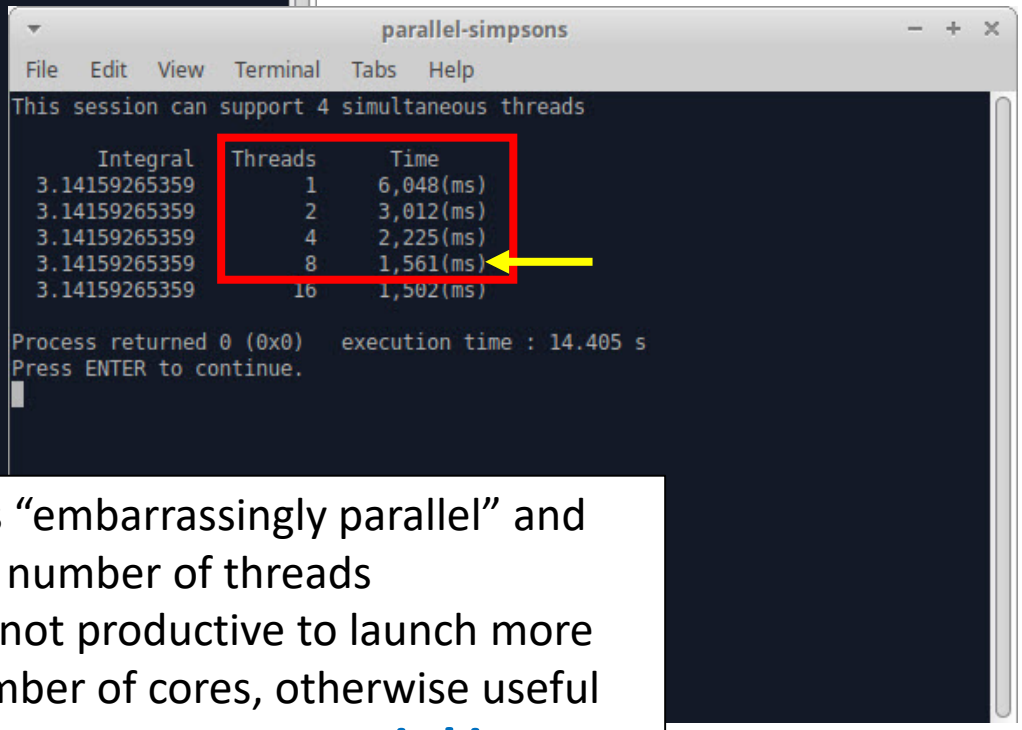
- Divide entire integration region [0,1] into $\boldsymbol{n}$ distinct slices ∴ each thread has its <u>own</u> assigned $[a, b]$ values
- To achieve accuracy goal, use 2^{27} steps across the **entire** region [0,1] ∴ each thread only needs to use $\left(\frac{2^{27}}{\boldsymbol{n}}\right)$ intervals across its slice

Run Lab 5 – Parallel Simpson's Rule

This machine had 2 cores:



This machine had 4 cores:



- Numerical integration is "embarrassingly parallel" and *scales* well according to number of threads
- However, it is generally not productive to launch more threads than **2x** the number of cores, otherwise useful CPU time is lost due to excessive **context switching**

Now you know…

- Using **C++ Threads** enables programmers to distribute computation tasks across multiple CPU cores simultaneously, executing many of them in parallel, for a potentially **significant** speed-up over traditional *sequential* algorithms
- However, launching too many threads can actually *decrease* performance – rule of thumb is to dispatch no more threads than **2X** the number of hardware "cores" available
- Critical data operations and input/output may need to be sequenced (protected) by **mutexes** and other synchronization *primitives* (e.g. **atomic<int>**)

Now you know…

- The **Thread Control Block** (TCB) design pattern is a common way to achieve simple *decoupled* multithreading
  - Each **TCB** can pass multiple controlling parameters to its thread, and can receive output back from its thread in a "lock free" manner
  - The **TCB** pattern fulfills the requirement that every **Scatter** phase (launch threads) has a **Gather** phase (join threads) which collects the work results from each thread before the main program ends
- The advent of many-core GPUs (NVIDIA's latest **Ampere** has **8,192 cores!!**) requires programmers to wield *parallel* algorithms to maximize infrastructure value
  - High performance scientific computing requires expertise with handling **threads** – start to learn them **now**!