Concurrency and Hyperthreading

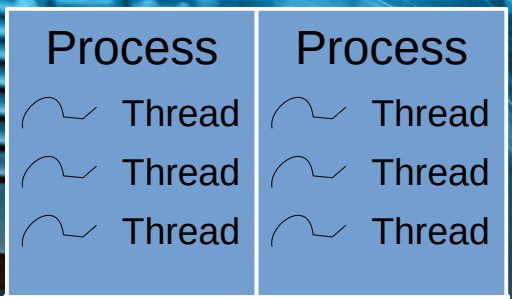
A Brief History of Concurrency

- **Moore's Law** (paraphrased): Computer tech (originally transistor density) doubles every 2 years
  - CPU speed, transistor and memory density, disk capacity, etc.
- By the 21st century, Moore's Law began to crack
  - Processor speeds topped out around 4 GHz (2.8 – 3.4 common)
  - Transistor density continued for some time – but how to best use?
- **Multi-Core Processor** – a chip with multiple **cores**, or ALU/register sets, each running a separate thread
  - Intel worked out use of one ALU with 2 register sets, interleaving 2 threads of execution – **Hyperthreading**
- Recently deployed 24 hyperthreaded core machines – but how to utilize so many cores?

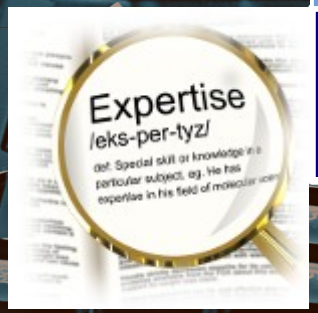
Concurrency

Concurrency

- “I do one thing, I do it very well, and then I move on” – Dr. Charles Emmerson Winchester III
- “Move on, Chaaarles” – Hawkeye
- **Concurrency** – Performing 2 or more algorithms (as it were) simultaneously
- **Process** – A self-contained execution environment including its own memory space.
- **Thread** – An independent path of execution within a process, running concurrently (as it appears) with other threads within a shared memory space.



Conceptual Model



https://gcc.gnu.org/wiki/cauldron2015?action=AttachFile&do=get&target=Torvald+Riegel_+Modern+concurrent+code+in+C.pdf

Good Uses for Concurrency

- Perform background processing independent of the user interface, e.g., communicating the game state to apps
- Programming logically independent program units, e.g., the behavior of each non-player character in a game
- Processing small, independent units of a large problem, e.g., calculating the shaded hue of each pixel in a rendered photograph – or rendering an entire movie frame by frame!
- Periodic updating of a display or hardware unit, e.g., updating the second hand of a clock
- Periodic collection of data, e.g., capturing wind speed from an anemometer every 15 seconds

Creating a C++ Thread via Thread

- Class std::thread represents a thread of execution
  - Each thread has a unique thread ID (.get_id())
  - Each initialized thread can be “joined” back to the main thread (a non-initialized thread can't)

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

// The function we want to execute on the new thread
void task1(string msg) {
    cout << "task1 says: " << msg << endl;
}

int main() {
    // Constructs the new thread and runs it. Does not block execution.
    thread t1(task1, "Hello");
    cout << "Thread 1 ID is " << t1.get_id() << endl;

    // Makes the main thread wait for the new thread to finish execution
    t1.join();
}
```

Creating a C++ Thread via Thread

- Class std::thread represents a thread of execution
  - Each thread has a unique thread ID (.get_id())
  - Each initialized thread can be “joined” back to the main thread (a non-initialized thread can't)

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

// The function we w
void task1(string ms
    cout << "task1 s
}

int main() {
    // Constructs the new thread and runs it. Does not block execution.
    thread t1(task1, "Hello");
    cout << "Thread 1 ID is " << t1.get_id() << endl;

    // Makes the main thread wait for the new thread to finish execution
    t1.join();
}
```

```
ricegf@pluto:~/dev/cpp/201608/threads$ g++ -std=c++11 threadid.cpp
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
terminate called after throwing an instance of 'std::system_error'
  what():  Enable multithreading to use std::thread: Operation not permitted
Aborted (core dumped)
ricegf@pluto:~/dev/cpp/201608/threads$ g++ -std=c++11 -pthread threadid.cpp
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
Thread 1 ID is 7f6211307700
task1 says: Hello
ricegf@pluto:~/dev/cpp/201608/threads$
```

Must compile with -pthread

C++ Offers a *Hint* at HW Support

- thread:hardware_concurrency() returns a rough estimate of the number of *concurrent* threads
  - This may represent cores or hyperthreaded half-cores
  - This may return 0 or nothing relevant at all

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

using namespace std;

int main()
{
    // Show rough approximation of thread capacity
    cout << "Hardware concurrency is " << thread::hardware_concurrency() << endl;
}
```

```
ricegf@pluto:~/dev/cpp/201608/threads$ g++ -std=c++11 -pthread hw_concurrency.cpp
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
Hardware concurrency is 4
task1 says: Hello
ricegf@pluto:~/dev/cpp/201608/threads$
```

Threads can be Confusing

- Both main() and t1() execute independently
  - Threads can switch execution *between microprocessor instructions* (not C++ lines) at any time
  - This can garble output

Ungarbled output

```
ricegf@pluto:~/dev/cpp/threads$ g++ -std=c++0x -pthread threadid.cpp
ricegf@pluto:~/dev/cpp/threads$ ./a.out
Thread 1 ID is 140301458634496
task1 says: Hello
ricegf@pluto:~/dev/cpp/threads$
```

Garbled output

```
ricegf@pluto:~/dev/cpp/threads$ ./a.out
Thread 1 ID is 1404682637125 12task1 says:
Hello
ricegf@pluto:~/dev/cpp/threads$
```

  - We'll see how to avoid this in a bit

Sleeping a Thread

- It's tempting to pause a thread using a “busy loop”

```
for (int i = 0; i < 100000; ++i) { } // Wait a while
```

- This is *very* problematic
  - Compilers are very smart nowadays, and may optimize away the useless loop
  - Processor speeds vary widely, so timing is uncertain
  - If it runs the instructions, it's burning valuable CPU cycles that could be used by other threads
- Instead, use this_thread::sleep_for

```
this_thread::sleep_for(chrono::milliseconds(2000); // or seconds(2)
```

Sleeping 3 Threads Randomly

```
#include <string>
#include <iostream>
#include <thread>
#include <chrono>
#include <time.h>
using namespace std;

void task1(string msg) {
    this_thread::sleep_for(chrono::milliseconds(200+rand()%200));
    cout << "task1 says: " << msg << endl;
}

int main() {
    // Randomize the pseudorandom number generator
    srand(time(NULL));

    // Construct new threads
    thread t1(task1, "Hello");
    thread t2(task1, "Bonjour");
    thread t3(task1, "Hola");

    // Join all threads back
    t1.join();
    t2.join();
    t3.join();
}
```

Sleeping 3 Threads Randomly

```
#include <string>
#include <iostream>
#include <thread>
#include <chrono>
#include <time.h>
using namespace std;

void task1(string msg) {
    this_thread::sleep_for(chrono::milliseconds(200+rand()%200));
    cout << "task1 says: " << msg << endl;
}

int main() {
    // Randomize the pseudorandom number generator
    srand(time(NULL));

    // Construct new threads
    thread t1(task1, "Hello"
    thread t2(task1, "Bonjou
    thread t3(task1, "Hola")

    // Join all threads back
    t1.join();
    t2.join();
    t3.join();
}
```

```
ricegf@pluto:~/dev/cpp/201608/threads$ g++ -std=c++11 -pthread 3thread.cpp
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
task1 says: Bonjour
task1 says: Hola
task1 says: Hello
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
task1 says: Hello
task1 says: Hola
task1 says: Bonjour
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
task1 says: Hola
task1 says: Hello
task1 says: Bonjour
ricegf@pluto:~/dev/cpp/201608/threads$
```

Tasks may start and run *in any order*

Applying Threads to Serious Work (Race Horses)

horse.h

```
#include <string>

using namespace std;

class Horse {
    public:
        Horse(string name, int speed)
      : _name{name}, _speed{speed}, _position{30} { }
        string name();
        int position();
        int speed();
        int move();
    protected:
        string _name;
        int _position;
        int _speed;
};
```

horse.cpp

```
#include "horse.h"

using namespace std;

string Horse::name() {return _name;}
int Horse::position() {return _position;}
int Horse::speed() {return _speed;}
int Horse::move() {if (_position > 0) --_position;}
```

Utility functions

horserace.cpp

```
#include <string>
#include <iostream>
#include <thread>
#include <chrono>
#include <time.h>
#include "horse.h"

using namespace std;

// Our three competitors, assigned random speeds (smaller is faster)
Horse horse1("Legs of Spaghetti", 100 + rand() % 100);
Horse horse2("Ride Like the Calm", 100 + rand() % 100);
Horse horse3("Duct-taped Lightning", 100 + rand() % 100);

// The gallop threads, which decrement position after a random delay
void gallop(Horse& horse) {
    while (horse.position() > 0) {
        this_thread::sleep_for(
            chrono::milliseconds(horse.speed() + rand() % 200));
        horse.move();
    }
}

// Utility function to print the horse track
void view(int position) {
    for (int i = 0; i < position; ++i) cout << (i%5 == 0 ? ':' : '.');
}                                          // Continued next page
```

main()

horserace.cpp

```
int main() {
    // Randomize the pseudorandom number generator
    srand(time(NULL));

    // Construct the new threads (horses) and run them
    thread t1{gallop, ref(horse1)};
    thread t2{gallop, ref(horse2)};  // ref( forces the parameter to pass as
    thread t3{gallop, ref(horse3)};  // a reference rather than a value

    // Display the horse track as the race runs
    while (horse1.position() > 0
       &&  horse2.position() > 0
       &&  horse3.position() > 0) {
        cout << endl << endl << endl << endl;
        view(horse1.position());
        cout << " " << horse1.name() << endl;
        view(horse2.position());
        cout << " " << horse2.name() << endl;
        view(horse3.position());
        cout << " " << horse3.name() << endl;
        this_thread::sleep_for(chrono::milliseconds(100));
    }

    // Join the threads
    t1.join();
    t2.join();
    t3.join();
}
```

The Obligatory Makefile

Makefile

```
CXXFLAGS += -std=c++11 -pthread

all: main

debug: CXXFLAGS += -g
debug: main

rebuild: clean main

main: horserace.o horse.o
        g++ -o horserace $(CXXFLAGS) horserace.o horse.o
horserace.o: horserace.cpp horse.h
        g++ $(CXXFLAGS) -c horserace.cpp
horse.o: horse.cpp horse.h
        g++ $(CXXFLAGS) -c horse.cpp
clean:
        -rm -f *.o *~ horserace
```

```
:.. Ride Like the Calm
:.... Duct-taped Lightning


:....: Legs of Spaghetti
:.. Ride Like the Calm
:.... Duct-taped Lightning


:....: Legs of Spaghetti
:.. Ride Like the Calm
:... Duct-taped Lightning


:.... Legs of Spaghetti
:.. Ride Like the Calm
:... Duct-taped Lightning


:.... Legs of Spaghetti
:. Ride Like the Calm
:... Duct-taped Lightning


:.... Legs of Spaghetti
:. Ride Like the Calm
:.. Duct-taped Lightning


:.... Legs of Spaghetti
: Ride Like the Calm
:.. Duct-taped Lightning
```

Running the Race

(It's a lot easier to follow live!)

Concurrency is Harder than Single-Threaded

- Non-reentrant code can lose data
  - **Reentrant** – An algorithm can be paused while executing, and then safely executed by a different thread
  - Non-reentrant code can experience **Thread Interference**
- Methods that aren't **thread safe** enable Threads to corrupt objects
  - Thread A updates a portion of the object's data, while Thread B updates a dependent portion, leaving the object in an inconsistent state – the bug's impact occurs much later!
- Memory Consistency Errors
  - Because variables may be cached, one thread's change may never be incorporated by a different thread's algorithm
- Concurrent bugs tend to appear only when multiple threads happen to align, thus appearing to be both rare and random
  - Nightmare debugging scenario

The "Garbled Output" Problem Revisited

- Here's a (Java) example of a thread sync problem

(`javap -c Counter.class` disassembles bytecode)

```
public class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public void decrement() {
        count--;
    }

    public int getCount() {
        return count;
    }
}
```

```
public void increment();
  Code:
     0: aload_0
     1: dup
     2: getfield      #2
          // Field count:I
     5: iconst_1
     6: iadd
     7: putfield      #2
          // Field count:I
    10: return
```

Thread A

Paused here!

```
public void decrement();
  Code:
     0: aload_0
     1: dup
     2: getfield      #2
          // Field count:I
     5: iconst_1
     6: isub
     7: putfield      #2
          // Field count:I
    10: return
```

Thread B runs decrement

Thread A resumes

The change made by decrement in Thread B is overwritten by the obsolete data in Thread A. $+1-1 \stackrel{?}{=} 1$

Hint: Immutable Classes!

- If the data can't change, synchronization issues can't crop up *in that class*
  - Create and use immutable classes when practical
  - In an immutable class, mark all data fields as const to indicate those fields won't change
  - This isn't always possible...
  -

General C++ Solution: The Mutex

- Assume a crowd of people want to use an old-fashioned phone booth. The first to grab the door handle gets to use it first, but must hang on to the handle to keep the others out. When finished, the person exits the booth and releases the door handle. The next person to grab the door handle gets to use the phone booth next.

- A **thread** is: Each person
  The **mutex** is: The door handle
  The **lock** is: The person's hand
  The **resource** is: The phone

Hat tip to Nav on Stack Overflow for this analogy

The Mutex in Code

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std

mutex m;
int i = 0;

void makeACallFromPhoneBooth() {
    m.lock(); // person grabs the phone booth door and locks it, all others wait
    cout << i << " talks on phone" << endl;
    i++; //no other thread can access variable i until m.unlock() is called
    m.unlock(); // person releases the door handle and unlocks the door
}

int main() {
    thread person1(makeACallFromPhoneBooth); // Despite appearances, as we saw
    thread person2(makeACallFromPhoneBooth); // earlier, it is not clear who will
    thread person3(makeACallFromPhoneBooth); // reach the phone booth first!

    person1.join(); // person1 finished their phone call and joins the crowd
    person2.join(); // person2 finished their phone call and joins the crowd
    person3.join(); // person3 finished their phone call and joins the crowd
}
```

```
ricegf@pluto:~/dev/cpp/201608/threads$ g++ -std=c++11 -pthread phone.cpp
ricegf@pluto:~/dev/cpp/201608/threads$ ./a.out
0 talks on phone
1 talks on phone
2 talks on phone
ricegf@pluto:~/dev/cpp/201608/threads$
```

Testing the Mutex

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

static const int num_threads = 50;
static const int num_decrements = 5000;

int counter = num_threads * num_decrements;

// This is the code to be run as threads
void decrementer() {
    for (int i=num_decrements; i > 0; --i) {
        --counter;
    }
}
int main() {
    //Launch a group of threads
    thread t[num_threads];
    for (int i = 0; i < num_threads; ++i) t[i] = thread(decrementer);

    //Join the threads with the main thread
    for (int i = 0; i < num_threads; ++i) t[i].join();

    cout << "This should be 0: " << counter << endl;
    return counter;
}
```

Allow 5000 chances for thread interference

```
ricegf@pluto:~/dev/cpp/201708/21$ ./no_mutex
This should be 0: 18067
ricegf@pluto:~/dev/cpp/201708/21$ ./no_mutex
This should be 0: 24515
ricegf@pluto:~/dev/cpp/201708/21$ ./no_mutex
This should be 0: 135590
ricegf@pluto:~/dev/cpp/201708/21$ ./no_mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201708/21$
```

Testing the Mutex

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

static const int num_threads = 50;
static const int num_decrements = 5000;

int counter = num_threads * num_decrements;

mutex m;

// This is the code to be run as threads
void decrementer() {
    for (int i=num_decrements; i > 0; --i) {
        m.lock(); --counter; m.unlock();
    }
}
int main() {
    //Launch a group of threads
    thread t[num_threads];
    for (int i = 0; i < num_threads; ++i) t[i] = thread(decrementer);

    //Join the threads with the main thread
    for (int i = 0; i < num_threads; ++i) t[i].join();

    cout << "This should be 0: " << counter << endl;
    return counter;
}
```

Allow 5000 chances for thread interference

```
ricegf@pluto:~/dev/cpp/201708/21$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201708/21$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201708/21$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201708/21$ ./mutex
This should be 0: 0
ricegf@pluto:~/dev/cpp/201708/21$
```

Warning

- This just scratches the surface of concurrency
  - Avoiding race conditions is exceptionally tricky
  - Other dangers lurk, e.g., priority inversion
  - Bugs in concurrent systems are often catastrophic, but appear only once in a blue moon
- This lecture gives you just enough knowledge to get in trouble... or to motivate you to learn much more about a field growing in importance
  - Your decision...