CS130: Software Engineering

Lecture 12: Threading and Concurrency

https://bit.ly/3N1wlSY

- A word: A random word.
- A tweet: it'll show up after you give the word.

Assignment 7 Notes

Major pieces:

- API request handler for
  - Create
  - Read
  - Update
  - Delete
  - List
- Unit tests
- Integration tests

What you can reuse:

- Common API
- Static Request Handler to read files
- Logic to map from request path to local file path
- Integration test skeleton

Major pieces:

- API request handler for
  - Create
  - Read
  - Update
  - Delete
- Unit tests
- Integration tests

Nice things:

- Create and Update are very similar
  - Create a new file vs write to existing file
- Read is basically static request handler
- Delete is very simple
  - If file exists delete it
- List is simple
  - List the files, return them

Major pieces:

- API request handler for
  - Create
  - Read
  - Update
  - Delete
- Unit tests
- Integration tests

New things:

- Parse data from request for upload
- Abstraction for file operations
- Integration tests with commands other than GET

Possible workflows

1. Parse request into:
    a. Verb
    b. Path
    c. Request data (if present)
2. Request handler methods to:
    a. Map a request path to a file path
    b. Read data from a file path
    c. Write data to a file path
    d. List file names from file path
    e. Format file names into JSON text
    f. Format JSON text into response
3. Create filesystem abstraction for unit tests
    a. read_file(path)
    b. write_file(path)
    c. delete_file(path)
    d. list_files(path)
4. Update integration tests to support POST, PUT, DELETE commands

Concurrency

Background

Concepts you should already know about:

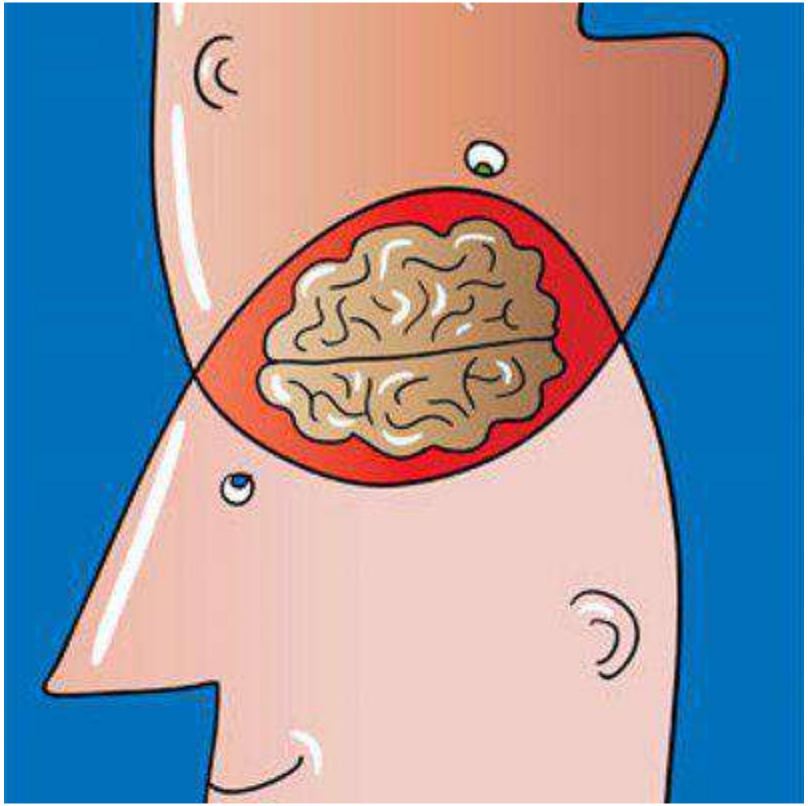
- Threads, processes
- Synchronization primitives: mutexes, semaphores, etc.
- Critical sections, race conditions, deadlocks

Background (refresher)

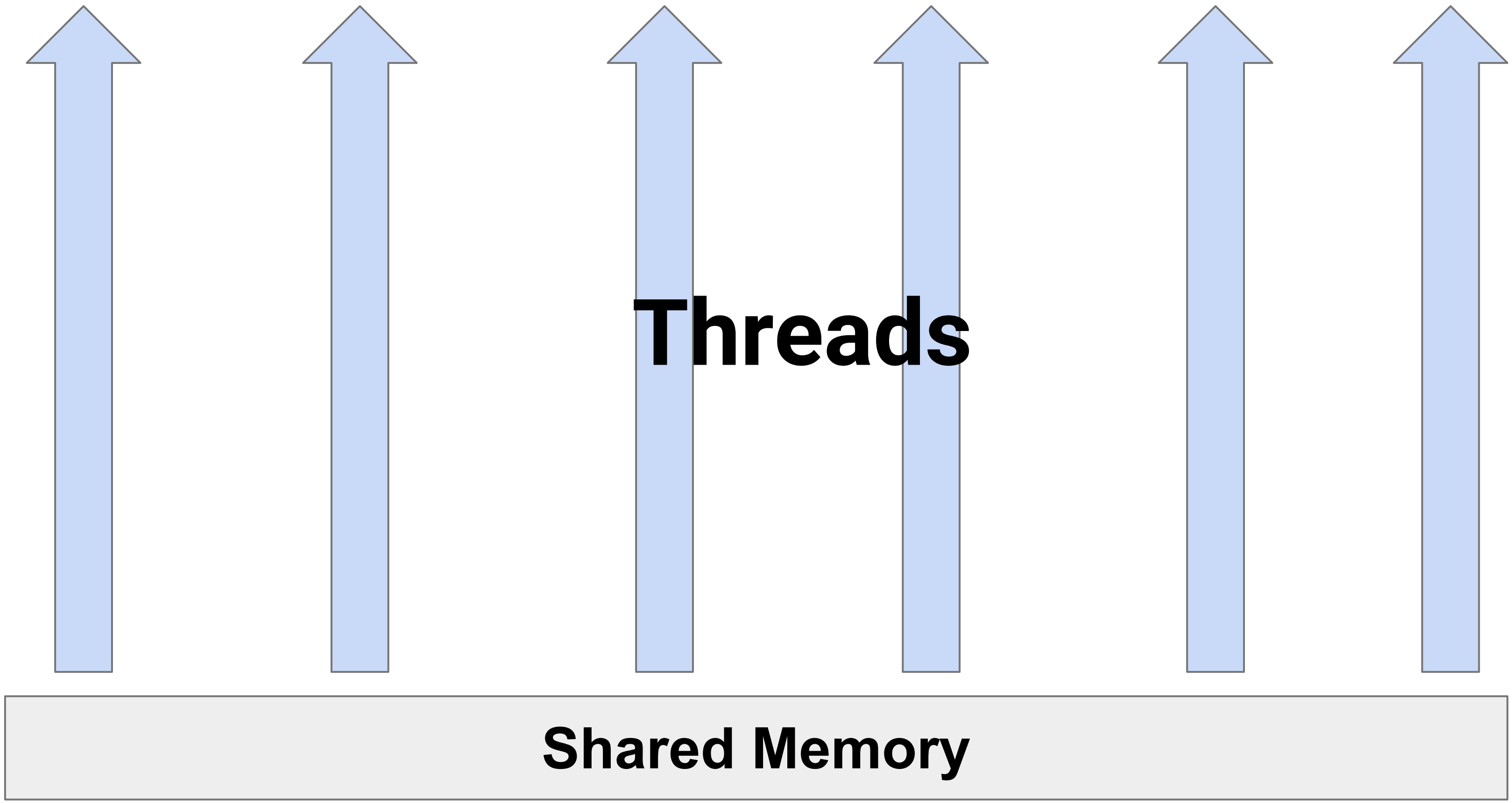
Concepts you should already know about:

- **Threads, processes**
- Synchronization primitives: mutexes, semaphores, etc.
- Critical sections, race conditions, deadlocks

TDK TDK
TOYOTA
SEIKO
9.5
9
8
7
6
5
4
3
2
PRESCOD
USA
JAMAICA
BLAKE
SIMBINE
USA
COLEMAN
JAMAICA
BOLT
VICAUT
CHINA
SU



Threads have shared memory


Threads
Shared Memory


Crayola.com

Background (refresher)

Concepts you should already know about:

- Threads, processes
- **Synchronization primitives: mutexes, semaphores, etc.**
- Critical sections, race conditions, deadlocks


RESTROOM
RESTROOM


MAXIMUM
OCCUPANCY
350


MAXIMUM
OCCUPANC
350


NIGHT
CLUB

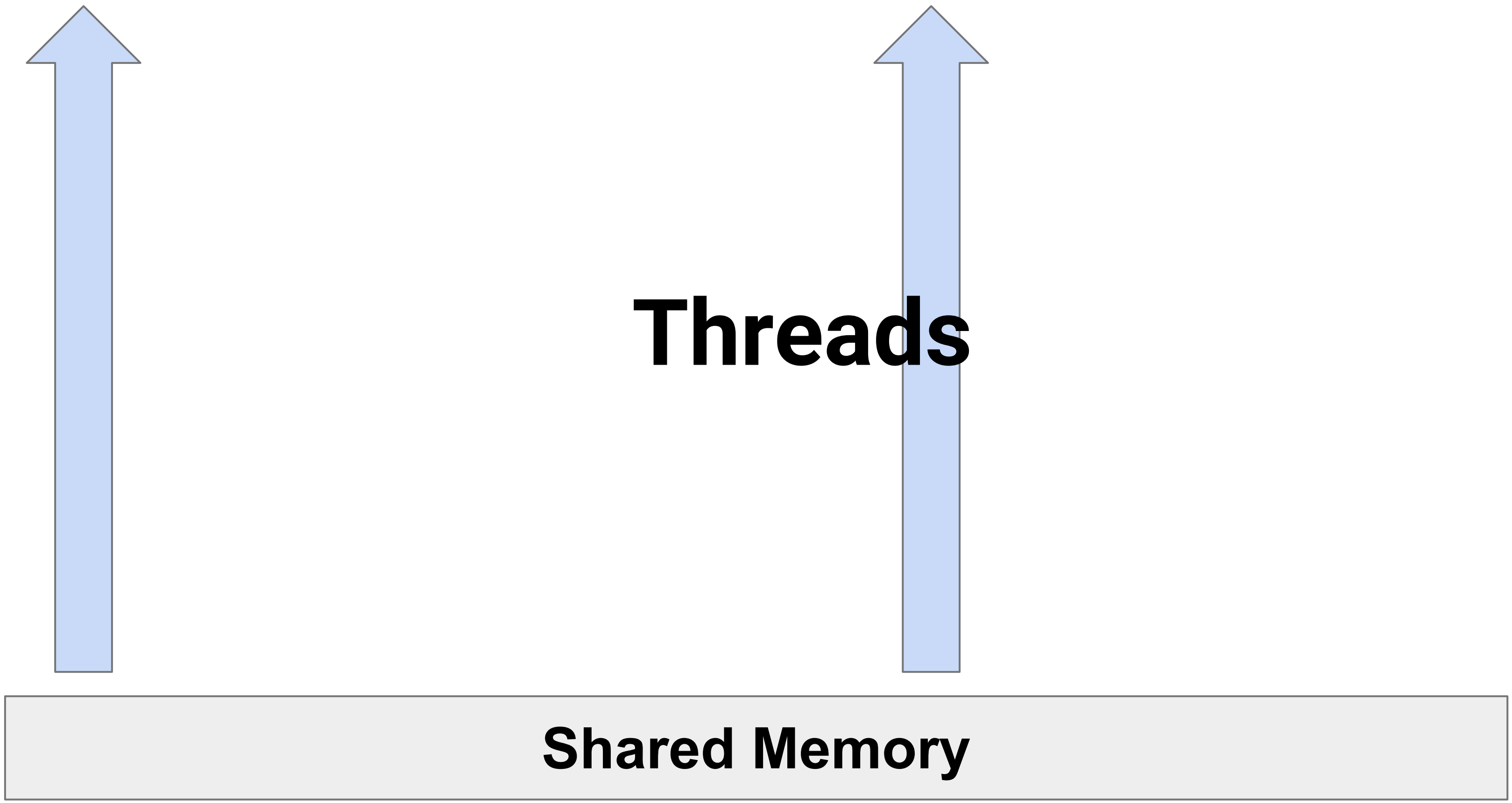
Background (refresher)

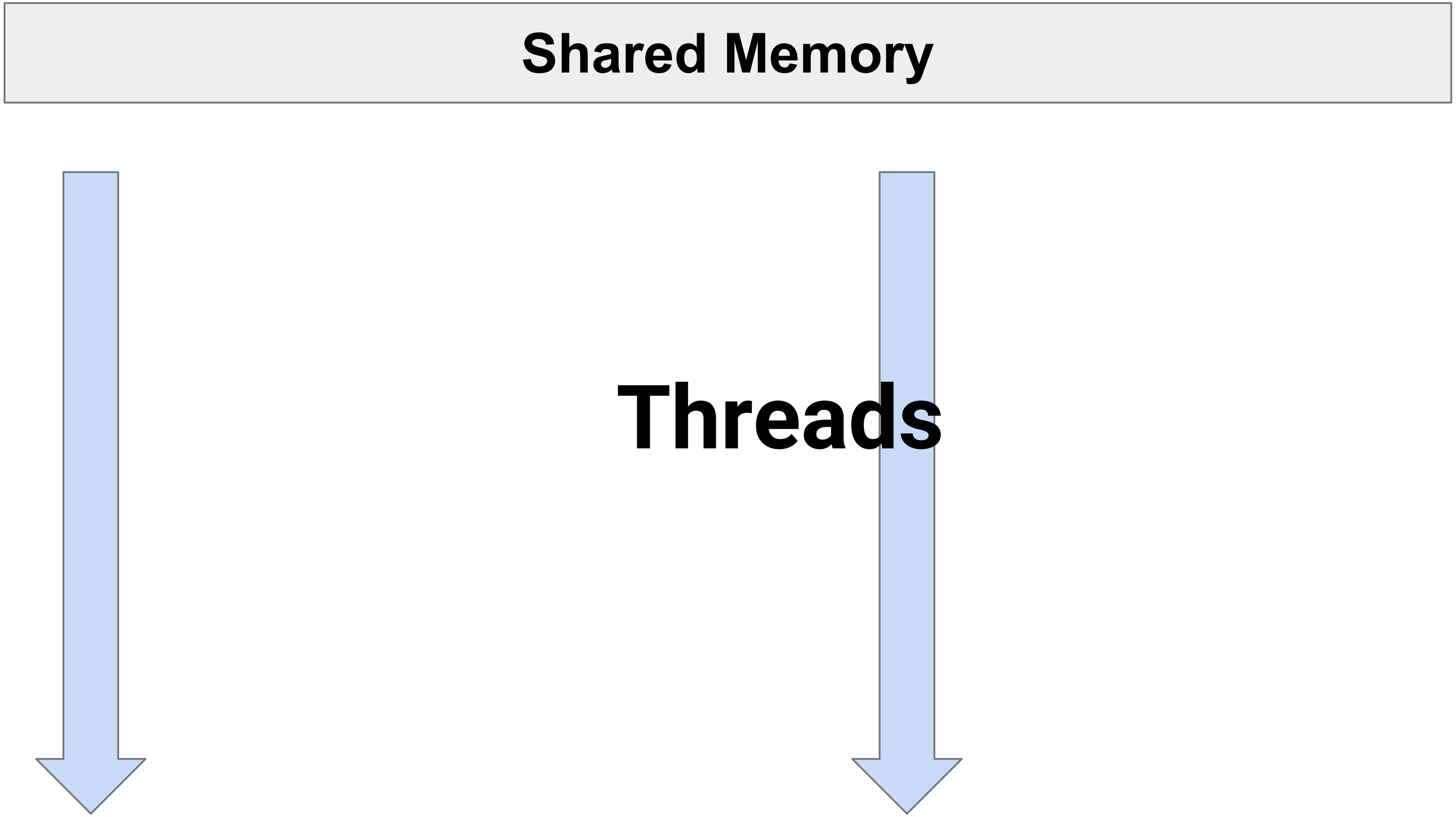
Concepts you should already know about:

- Threads, processes
- Synchronization primitives: mutexes, semaphores, etc.
- **Critical sections, race conditions, deadlocks**


FORMULA TECH DYNO
Walkinshaw
PARTS SALES RENTAL
FUCHS
LUBRICANTS. TECHNOLOGY. PEOPLE.
FUCHS
FUCHS
Elders
BTP
RACING
EARTHMOVING EQUIPMENT
PARTS SALES RENTAL

Threads




Shared Memory
Threads

Shared Memory

```
1: sharedCounter->nextNumber()
  // returns 1

2: sharedCounter->nextNumber()
  // returns 2?
```

Shared Memory

```
1: sharedCounter->nextNumber()
  // returns 1

2: sharedCounter->nextNumber()
  // returns 2?
  // oh no returns 3
```

```
1: sharedCounter->nextNumber()
  // returns 1? Oh no
  // returns 2 :(

2: sharedCounter->nextNumber()
  // returns 2?
  // not even close, returns 4
```

Why concurrency?

Make things go faster!

- Don't sit around waiting on something. Work on something else
- Modern CPUs have many cores. Keep them busy.
- Make UIs responsive

Concurrency is hard

Concurrency is really hard

Even experienced engineers find it hard to write concurrent code that doesn't race/deadlock/crash.

This lecture is about strategies for coping with concurrency in the real world.

Googlers do it wrong:

Before:
https://chromium.googlesource.com/chromium/src/+/fd5b108c82d56f6022dfbe62a023d1e81ff6f83b/base/files/file_tracing.cc

After:
https://chromium.googlesource.com/chromium/src/+/00dd6010cdda17cecd624c0f274aeb630f31fb83/base/files/file_tracing.cc

Where we messed up:
https://codereview.chromium.org/2114993003

Instructors do it wrong:

```
class A-RPCImplementationDetail {
  String method_; ← local variable
}

class B-RPCImpl : A-RPCImpl {
  void CopyRpc() {
      rpc_copy.set_string_view(method_);
  }
}

class RpcCopyUser {
  // somewhere deep in tear-down logic, now using B-RPCImpl.
  print(rpc.method());  ← accesses const string-view
}
```

We do it wrong: www.cs130.org goes down

Where’s the problem?

```
steps:
- name: 'gcr.io/cloud-builders/gcloud-slim'
  entrypoint: 'bash'
  args:
  - '-c'
  - |
    mkdir _site
- name: 'jekyll/jekyll:3.8'
  args: [ 'jekyll', 'build' ]
  env:
  - 'JEKYLL_VERSION=3.8'
- name: '18fgsa/html-proofer:latest'
  args: [ '_site/', '--disable-external' ]
- name: 'gcr.io/cloud-builders/gcloud-slim'
  args:
  - 'compute'
  - 'scp'
  - '_site/'
  - 'chronos@${_GCE_INSTANCE}:${_DST_PATH}.tmp'
  - '--recurse'
  - '--zone=${_GCE_ZONE}'
  - '--ssh-key-expire-after=1m'
```

```
- name: 'gcr.io/cloud-builders/gcloud-slim'
  args:
  - 'compute'
  - 'ssh'
  - 'chronos@${_GCE_INSTANCE}'
  - '--zone=${_GCE_ZONE}'
  - '--ssh-key-expire-after=1m'
  - '--command'
  - |
    if [ -d ${_DST_PATH}.last ]; then
      rm -rf ${_DST_PATH}.last;
    fi;
    if [ -d ${_DST_PATH} ]; then
      mv ${_DST_PATH} ${_DST_PATH}.last;
    fi;
    mv ${_DST_PATH}.tmp ${_DST_PATH}
```

What’s the problem?

In one step:

- scp _site/ X.tmp

In subsequent step:

- If X.last exists, remove it
- If X exists, mv it to X.last
- Move X.tmp to X

How can we solve it?

```
- 'compute'
- 'scp'
- '_site/'
- 'chronos@${_GCE_INSTANCE}:${_DST_PATH}.tmp'
```

… then …

```
if [ -d ${_DST_PATH}.last ]; then
  rm -rf ${_DST_PATH}.last;
fi;
if [ -d ${_DST_PATH} ]; then
  mv ${_DST_PATH} ${_DST_PATH}.last;
fi;
mv ${_DST_PATH}.tmp ${_DST_PATH}
```

The problem:

scp _site/ X.tmp

scp _site/ X.tmp

rm X.last

X -> X.last

X.tmp -> X

rm X.last

X -> X.last

X.tmp -> X (error: X.tmp not found)

The fix:

scp _site/ X.1

scp _site/ X.2

rm X.last

X -> X.last

X.1 -> X

rm X.last

X -> X.last

X.2 -> X

Strategy 1: Be slow

Example: Regression test for ML models

1. Copy models to a staging directory
    a. Copy baseline models
    b. Copy test models
2. Evaluate the models on the same inputs. Compare the outputs.

Strategy 1: Be slow

Example: Regression test for ML models

1. Copy models to a staging directory
    a. Copy baseline models (10 minutes)
    b. Copy test models (10 minutes)
2. Evaluate the models on the same inputs. Compare the outputs.

Idea: Do 1a and 1b in parallel!

Strategy 1: Be slow

Example: Regression test for ML models

1. Copy models to a staging directory
    a. Copy baseline models (10 minutes)
    b. Copy test models (10 minutes)
2. Evaluate the models on the same inputs. Compare the outputs. (2 hours)

~~Idea: Do 1a and 1b in parallel!~~

Lesson: Always profile before you optimize

Strategy 2: Isolate

Example 1: An operating system

- Your first program: `void main() { printf ("Hello, world\n"); }`
- Runs in parallel with other processes, without you having to do anything.

It wasn't always like this.

- In early versions of MacOS, developers had to explicitly yield to the OS.
- Terrible API! Bad for developers (complex), bad for users (crashy).

Strategy 2: Isolate

Example 2: A web indexer

- Input: crawled web pages
- Indexer analyzes each web page, writes what it learned to a database
- Examples of analysis:
  - What other pages does it link to?
  - What entities (e.g. people) are mentioned in the page

```
while (true) {
  document = GetNextDocument();  // Blocks until next document arrives.
  ProcessDocument(document);
}
```

Strategy 2: Isolate

Process each document in a separate thread:

- Concurrency code is isolated to your main loop.
- All the code in `ProcessDocument` is single-threaded.
  - Easier to understand.
  - Other teams may be contributing their own analyses to `ProcessDocument`. Easier for them.
- Natural. When you're processing a document, you shouldn't care about other documents being processed at the same time.
- Throughput more important than latency for a web indexer
  - Care about overall documents processed per second, not time to process each document
  - Just increase the number of threads until your CPU is saturated.

Strategy 2: Isolate

Example 3: A web server

- Run each request in a separate thread

Strategy 3: Use a Library

Example: An RPC server

- We've already isolated each request in its own thread
- But we want to make requests faster

```
void HandleRequest(...) {
  DoStuff1();
  DoStuff2();
}
```

- We want `DoStuff1()` and `DoStuff2()` to happen in parallel.

Threads in C++11

```
#include <thread>
void HandleRequest(...) {
  std::thread t1(DoStuff1);
  std::thread t2(DoStuff2);
  t1.join();
  t2.join();
}
```

Compiler flags: -std=c++11 -pthread

Threads in C++11: Passing Data

Story time

- I wanted to write the simplest possible threading example for this lecture.

Story time

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

void f(int x) {
  std::cout << "f(" << x << ")\n";
}

int main() {
  std::thread t1(f, 1);
  std::thread t2(f, 2);
  t1.join();
  t2.join();
  return 0;
}
```

Story time

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

void f(int x) {
  std::cout << "f(" << x << ")\n";
}

int main() {
  std::thread t1(f, 1);
  std::thread t2(f, 2);
  t1.join();
  t2.join();
  return 0;
}
```

What's the output:

a) f(1)
   f(2)

b) f(2)
   f(1)

c) Either (a) or (b)

Story time

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

void f(int x) {
  std::cout << "f(" << x << ")\n";
}

int main() {
  std::thread t1(f, 1);
  std::thread t2(f, 2);
  t1.join();
  t2.join();
  return 0;
}
```

What's the output:

a) f(1)
   f(2)

b) f(2)
   f(1)

c) Either (a) or (b)

d) f(f(12)
)

Time for a design review

- Find a teammate, describe your problem, ask for advice.
- My experience: 90% of the time, they will suggest a better, simpler, safer way.

An actual design discussion

- Teammate says to me:
  "I want to run some code in a thread every 10 seconds."
- Josh already wrote that! (In 2006)
- He also thought about what happens if the code runs for more than 10 seconds.

Threads in C++11: Passing Data

Pretty safe:

- Pass by value. Copy the data.
- `DoStuff()` has its own private copy that no other thread can touch.

```
void DoStuff(int x) {
  std::cout << x << std::endl;  // 2, for sure.
}

int i = 2;
std::thread t(DoStuff, i);
```

Threads in C++11: Passing Data

Not so safe:

- Pass by const reference or ptr
- `DoStuff()` can't change the data … but another thread can!

```
void DoStuff(const int& x) {
  std::cout << x << std::endl;
}

int i = 2;
std::thread t(DoStuff, std::ref(i));
```

```
void DoStuff(const int* x) {
  std::cout << *x << std::endl;
}

int i = 2;
std::thread t(DoStuff, &i);
```

Threads in C++11: Passing Data

Not so safe:

- Pass by const reference or ptr
- `DoStuff()` can't change the data ... but another thread can!

```
void DoStuff(const int& x) {
  std::cout << x << std::endl;
}

int i = 2;
std::thread t(DoStuff, std::ref(i));
i = 3;
```

```
void DoStuff(const int* x) {
  std::cout << *x << std::endl;
}

int i = 2;
std::thread t(DoStuff, &i);
i = 3;
```

Threads in C++11: Passing Data

Asking for trouble:

- Pass a (non-const) pointer.
- `DoStuff()` is practically promising to change its value.

```
void DoStuff(int* x) {
  *x = 4;
}

int i = 2;
std::thread t(DoStuff, &i);
// What is i now?
i = 3;  // Surely i == 3 now!
```

Threads in C++11: Returning values

```
int DoStuff() { ... }
std::thread t(DoStuff);
t.join();  // return value of DoStuff is lost.
```

Threads in C++11: Returning values

```cpp
#include <future>

int Square(int x) {
  return x*x;
}

auto a = std::async(Square, 2);
int v = a.get();  // v == 4
```

Threads in C++11: Returning values

```
#include <future>

int Square(int x) {
  return x*x;
}

std::future<int> a = std::async(Square, 2);
int v = a.get();  // v == 4
```

Strategy 4: Use a mutex

Example: A thread-safe counter

```
Counter counter;
counter.Increment("requests", 1);
counter.Increment("errors", 1);
```

Strategy 4: Use a mutex

Example: A thread-safe counter

```
class Counter {
 public:
  void Increment(const string name, int by);
  void Get(const string name);
 private:
  map<string, int> counters_;  // STL containers not threadsafe.
};
```

Strategy 4: Use a mutex

```
void Counter::Increment(const string name, int by) {
 counter_mutex_.lock();
 counters_[name] += by;
 counter_mutex_.unlock();
}
```

Strategy 4: Use a mutex

Example: A thread-safe counter

```
#include <mutex>

class Counter {
 public:
  void Increment(const string name, int by);
  void Get(const string name);
 private:
  map<string, int> counters_;  // STL containers not threadsafe.
  std::mutex counter_mutex_;
};
```

Strategy 4: Use a mutex

At this point, you may get nervous, and start wondering:

- Is the mutex going to be too slow?
- Do I need to worry about lock contention?
- Should I use a spinlock?
- Should I use a lock-free algorithm?
- Should I use an atomic increment hardware operation?
- Do I need richer semantics? Should I use a semaphore?

Strategy 4: Use a mutex

At this point, you may get nervous, and start wondering:

- Is the mutex going to be too slow? No!
- Do I need to worry about lock contention? No!
- Should I use a spinlock? No!
- Should I use a lock-free algorithm? No!
- Should I use an atomic increment hardware operation? No!
- Do I need richer semantics? Should I use a semaphore? No!

The answer to all these questions is (almost always) "no".

Just use a mutex.

A cautionary example

```
class SomeClass {
 private:
  // Helper is expensive to construct, and may not be needed.
  Helper* helper_ = nullptr;
  Helper* GetHelper() {
    if (helper_ == nullptr) helper_ = new Helper;
    return helper_;
  }
};
```

A cautionary example

How can we make `GetHelper()` thread-safe?

- We want to construct helper_ exactly once, and always return the same value

But then you might think...

How can we make it fast?

- After we've constructed helper_, we don't need to lock to return it (faster).
- We only need to lock if helper_ is null, to avoid constructing it twice.

A cautionary example

Clever … and wrong:

```
Helper* GetHelper() {
 if (helper_ == nullptr) {  // Fast!
   helper_mutex_.lock();  // Slow!
   // GetHelper could have been called while waiting to lock.
   if (helper_ == nullptr) {
     helper_ = new Helper;
   }
   helper_mutex_.unlock();
 }
 return helper_;
}
```

Correct, simpler, and fast enough

```
Helper* GetHelper() {
 helper_mutex_.lock();
 if (helper_ == nullptr) {
   helper_ = new Helper;
 }
 helper_mutex_.unlock();
 return helper_;
}
```

Strategy 5: Never forget to unlock

Recall:

```
class Counter {
 public:
  void Increment(const string name, int by);
  void Get(const string name);
 private:
  map<string, int> counters_;
  std::mutex counter_mutex_;
};
```

Strategy 5: Never forget to unlock

Let's implement `Get()`:

```
int Counter::Get(const string name) {
 counter_mutex_.lock();
 return counters_[name];
 counter_mutex_.unlock();  // Uh-oh
}
```

Strategy 5: Never forget to unlock

Let's implement `Get()`:

```
int Counter::Get(const string name) {
  std::lock_guard<std::mutex> lock(counter_mutex_);
  return counters_[name];
  // Mutex is unlocked when lock goes out of scope.
}
```

lock_guard and unique_ptr

```
template<class T>
class lock_guard {
 public:
  lock_guard(T& m) : mutex_(m) {
    mutex_.lock();
  }
  ~lock_guard() {
    mutex_.unlock();
  }
 private:
  T& mutex_;
};
```

```
template<class T>
class unique_ptr {
 public:
  unique_ptr(T* p) : ptr_(p) {}
  ~unique_ptr() {
    delete ptr_;
  }
 private:
  T* ptr_;
};
```

Strategy 6: Use tools

Clang thread-safety analysis

- Add annotations which are checked at compile-time
- Developed and used extensively at Google
- Available as of Clang 3.5
- A little tricky to get working. Requires a custom header file.

Strategy 6: Use tools

```
#include "mutex.h"  // Custom header. Wraps std::mutex.
class Counter {
 public:
  void Increment(const string name, int by);
  void Get(const string name);
 private:
  map<string, int> counters_ GUARDED_BY(counter_mutex_);
  Mutex counter_mutex_;  // Wrapped type
};
```

Forgetting to lock

```
void Counter::Increment(const string name, int by) {
  counters_[name] += by;
}
```

```
$ clang-3.5 -std=c++11 -c -Wthread-safety thread.cc
```

```
thread.cc:8:5: warning: reading variable 'counters_' requires
holding mutex 'counter_mutex_' [-Wthread-safety-analysis]
```

Forgetting to unlock

```
void Counter::Increment(const string name, int by) {
 counter_mutex_.lock();
 counters_[name] += by;
}
```

```
$ clang-3.5 -std=c++11 -c -Wthread-safety thread.cc
```

```
thread.cc:10:3: warning: mutex 'counter_mutex_' is still held at the
end of function [-Wthread-safety-analysis]
```

Strategy 7: Comment your code

When writing a new class, add a comment about its thread-safety

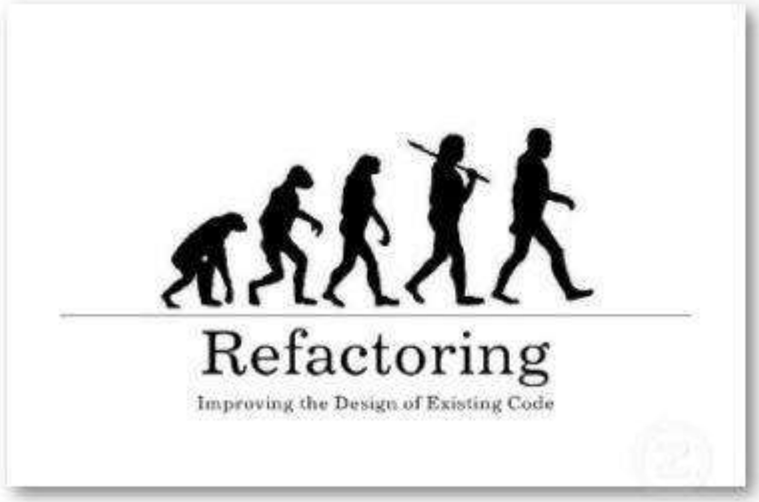
`// Thread-safe` → Can be called safely from multiple threads.

`// Thread-compatible` → Caller needs to do their own locking.

`// Thread-hostile` → Unsafe even if you lock it.

Refactoring

What is this “refactoring” you speak of?



- Simply, it is rewriting (editing in the traditional sense) code to improve some property.
- In this case, we are restructuring the code to be more testable.
- Could also refactor to make it more maintainable:
    - Divide up long functions (extract method)
    - Make a class do fewer things (extract class)

Refactoring Examples

Before:

```
void HandleRequest(
  string url,
  String body,
  String request_type,
  map<string, string> headers,
  MimeType* mime_type,
  map<string, string>* headers,
  string* response_body);
```

After:

```
Reply HandleRequest(Request req);
```

- Introduce param object when a method's param list gets too long
- Certain functions have many params
  - Often triggered by dependency injection or tunability
- Can create a param object (also known as an options struct) to encapsulate these params
- Nice side effect: you can define defaults and have a bit more control over values before the function executes

Refactoring Examples

Before:

```
…
if url.find(piece) == config_chunk {
  Return my_echo_handler.Handle(request);
}
...
```

After:

```
RequestHandler* h =
CreateHandlerForConfig(config_chunk.name,
config_chunk);

CreateHandlerForConfig(
   const string& name,
   const NginxConfig& config) {
  if (name == “static”) {
     return new StaticHandler::Init(config); }
  …
}
```

- Extract Function when a method is too long
- Create a helper to encapsulate a bit of logic

Refactoring Examples In Use

```
void ProcessRequest(tcp::socket* sock) {
    while (true) {
        const int kBufferLength = 1024;
        char data[kBufferLength];

        boost::system::error_code error;
        const size_t length =
            sock->read_some(boost::asio::buffer(data), error);

        HandleRequest(data, length);
    }
}

void HandleRequest(const char* buf, const size_t length) {
    [...]
}
```

Replace magic number with symbolic constant

Extract Method

Introduce param object (possibly)

Refactoring Approaches

- Write tests first!
  - Separate no-functionality-change architecture refactoring changes from testing changes.
  - Ensure your tests are functionally-driven (state, not interaction)
  - Introducing new boundaries means new tests -- old tests mean to remain intact

- Updating methods on existing classes means migrating the tests too
  - Frequently easier to do this BEFORE making architectural changes.
  - For small chunks of code, can do this simultaneously.
  - Usually easiest to proceed “outside-in”
    - Make new methods and ke-e-e-ep the o-o-old, one is gold, the other horrible dead-weight you want to delete ASAP.
    - Implement new methods in terms of old ones and update tests to new API.
    - Migrate implementation separately (hopefully without needing to touch tests).

- Sometimes you have to re-implement
  - If your present code is too distant from the goal, you may need to do a large-scale replacement.

Refactoring Risks

// TODO: finish migrating to new “Fish” interface by the end of 2014

Refactoring Tradeoffs

- You are trying to create value in a refactor.
  - More flexible interfaces
  - Better feature velocity
  - More compatibility
  - Support larger team
  - Better performance
  - ...
- Dealing with refactoring risks
  - Have a well-motivated plan!
  - Thorough milestones
  - Acceptable mid-point states (lots of places to stop and/or pause and have things better than before)
  - Getting to 100% can take a lot of effort.

https://bit.ly/3vZ5SzU

Two tweets:

- What’s the best part of the course so far?
- What more would you like to see?