More on Threads

We return to the idea of threads

More on Threads

We return to the idea of threads

POSIX threads is just one example of many different approaches to threads

More on Threads

We return to the idea of threads

POSIX threads is just one example of many different approaches to threads

And just one example of the many different *kinds* of threads

TBB

We shall look briefly at Threading Building Blocks (TBB) as it contains some interesting ideas

TBB

We shall look briefly at Threading Building Blocks (TBB) as it contains some interesting ideas

It is a standard C++ template library, needing no specific compiler support

TBB

We shall look briefly at Threading Building Blocks (TBB) as it contains some interesting ideas

It is a standard C++ template library, needing no specific compiler support

It provides things like concurrent containers and concurrent operations as well as the usual atomics and synchronisations

TBB Concurrent Operations

```
#include <tbb/tbb.h>
#include <iostream>

using namespace tbb;
using namespace std;

void hi(int n) {
  cout << "hello: " << n << endl;
}

int main() {
  parallel_for<int>(0, 10, hi);

  return 0;
}
```

TBB Concurrent Operations

Though you quickly realise you should have written

```
std::mutex m;

void hi(int n) {
   m.lock();
   cout << "hello: " << n << endl;
   m.unlock();
}
```

TBB Concurrent Operations

Though you quickly realise you should have written

```
std::mutex m;

void hi(int n) {
   m.lock();
   cout << "hello: " << n << endl;
   m.unlock();
}
```

But not a single `pthread_create` in sight!

TBB Concurrent Containers

Containers are things like vectors, queues and hash tables

You have to take care over concurrent access to these as pushing value to a stack at the same time as another thread is popping a value is an easy route to races

TBB Concurrent Containers

Containers are things like vectors, queues and hash tables

You have to take care over concurrent access to these as pushing value to a stack at the same time as another thread is popping a value is an easy route to races

Thus TBB provides safe datastructures that get the details right (we hope!)

TBB Work Stealing

The interesting thing about TBB is that is uses *work stealing* to manage parallelism

TBB Work Stealing

The interesting thing about TBB is that is uses *work stealing* to manage parallelism

In something like a `parallel_for` there are a lot of tasks to be scheduled across the available threads

TBB Work Stealing

The interesting thing about TBB is that is uses *work stealing* to manage parallelism

In something like a `parallel_for` there are a lot of tasks to be scheduled across the available threads

Each thread has a queue of tasks that are ready to be run (actually a *double ended queue*, or *deque*)

TBB Work Stealing

The interesting thing about TBB is that is uses *work stealing* to manage parallelism

In something like a `parallel_for` there are a lot of tasks to be scheduled across the available threads

Each thread has a queue of tasks that are ready to be run (actually a *double ended queue*, or *deque*)

When a new task is spawned it is pushed onto the end of the spawning thread's queue

TBB Work Stealing

The interesting thing about TBB is that is uses *work stealing* to manage parallelism

In something like a `parallel_for` there are a lot of tasks to be scheduled across the available threads

Each thread has a queue of tasks that are ready to be run (actually a *double ended queue*, or *deque*)

When a new task is spawned it is pushed onto the end of the spawning thread's queue

("Spawn" is the terminology for creating a new task)

TBB Work Stealing

When a thread completes a task it pops a task off the **end** of its queue and runs that next

TBB Work Stealing

When a thread completes a task it pops a task off the **end** of its queue and runs that next

That is, the most *recently* created task for that thread

TBB Work Stealing

When a thread completes a task it pops a task off the **end** of its queue and runs that next

That is, the most *recently* created task for that thread

If its queue is empty, the thread *steals* a task off the **start** of another thread's queue and runs that

TBB Work Stealing

When a thread completes a task it pops a task off the **end** of its queue and runs that next

That is, the most *recently* created task for that thread

If its queue is empty, the thread *steals* a task off the **start** of another thread's queue and runs that

That is, the *oldest* created task for that thread

TBB Work Stealing

When a thread completes a task it pops a task off the **end** of its queue and runs that next

That is, the most *recently* created task for that thread

If its queue is empty, the thread *steals* a task off the **start** of another thread's queue and runs that

That is, the *oldest* created task for that thread

Thus keeping all threads busy as long as there are tasks to do

TBB Work Stealing

Note that pushing and popping a task off your own queue is a relatively cheap operation, so the overhead is kept small for this case, which you hope is the common case

TBB Work Stealing

Note that pushing and popping a task off your own queue is a relatively cheap operation, so the overhead is kept small for this case, which you hope is the common case

In other words, when there is no opportunity for more parallelism as every thread is already busy doing its own tasks, the overhead is minimal

TBB Work Stealing

Note that pushing and popping a task off your own queue is a relatively cheap operation, so the overhead is kept small for this case, which you hope is the common case

In other words, when there is no opportunity for more parallelism as every thread is already busy doing its own tasks, the overhead is minimal

The overhead of stealing a task is greater, but this only happens when a thread would otherwise be idle and has time to spare

TBB Work Stealing

So: if a thread has work to do it does its most recently created task first, thus preserving locality of execution: the next task executed is “nearest” to one just finished

TBB Work Stealing

So: if a thread has work to do it does its most recently created task first, thus preserving locality of execution: the next task executed is “nearest” to one just finished

And if a thread has nothing to do it takes the oldest task off another thread, thus disrupting its locality as little as possible

TBB Work Stealing

So: if a thread has work to do it does its most recently created task first, thus preserving locality of execution: the next task executed is “nearest” to one just finished

And if a thread has nothing to do it takes the oldest task off another thread, thus disrupting its locality as little as possible

Exercise It’s *much* more complicated than this, of course. Read about the details

Exercise Work though how work stealing might execute the `parallel_for` example

TBB

Benefits of TBB:

TBB

Benefits of TBB:

- easy-to-write parallelism (for a good C++ programmer)

TBB

Benefits of TBB:

- easy-to-write parallelism (for a good C++ programmer)
- is very flexible and extensible (e.g., `parallel_for` works for any type that you can iterate over)

TBB

Benefits of TBB:

- easy-to-write parallelism (for a good C++ programmer)
- is very flexible and extensible (e.g., `parallel_for` works for any type that you can iterate over)
- purely a library, so you can use a standard compiler

TBB

Benefits of TBB:

- easy-to-write parallelism (for a good C++ programmer)
- is very flexible and extensible (e.g., `parallel_for` works for any type that you can iterate over)
- purely a library, so you can use a standard compiler
- and is easy to update with new versions of the library

TBB

Benefits of TBB:

- easy-to-write parallelism (for a good C++ programmer)
- is very flexible and extensible (e.g., `parallel_for` works for any type that you can iterate over)
- purely a library, so you can use a standard compiler
- and is easy to update with new versions of the library
- it provides sophisticated constructs like pipelines and general graph parallelism

TBB

Benefits of TBB:

- easy-to-write parallelism (for a good C++ programmer)
- is very flexible and extensible (e.g., `parallel_for` works for any type that you can iterate over)
- purely a library, so you can use a standard compiler
- and is easy to update with new versions of the library
- it provides sophisticated constructs like pipelines and general graph parallelism
- contains a large number of features

TBB

Drawbacks:

TBB

Drawbacks:

- the code needs some reasonably advanced C++ constructs (e.g., functors) get the most benefit

TBB

Drawbacks:

- the code needs some reasonably advanced C++ constructs (e.g., functors) get the most benefit
- little checking on the correctness of your use of the constructs: it provides *mechanism* but no *analysis*

TBB

Drawbacks:

- the code needs some reasonably advanced C++ constructs (e.g., functors) get the most benefit
- little checking on the correctness of your use of the constructs: it provides *mechanism* but no *analysis*
- it is tied to C++

TBB

Drawbacks:

- the code needs some reasonably advanced C++ constructs (e.g., functors) get the most benefit
- little checking on the correctness of your use of the constructs: it provides *mechanism* but no *analysis*
- it is tied to C++
- and thus not easily interoperable with other languages

TBB

Drawbacks:

- the code needs some reasonably advanced C++ constructs (e.g., functors) get the most benefit
- little checking on the correctness of your use of the constructs: it provides *mechanism* but no *analysis*
- it is tied to C++
- and thus not easily interoperable with other languages
- contains a large number of features

TBB

Drawbacks:

- the code needs some reasonably advanced C++ constructs (e.g., functors) get the most benefit
- little checking on the correctness of your use of the constructs: it provides *mechanism* but no *analysis*
- it is tied to C++
- and thus not easily interoperable with other languages
- contains a large number of features

Exercise Read about the large number of other features that TBB provides, particularly ranges for load balancing

Cilk Plus

Cilk Plus also has a task-based view of computation (like TBB), rather than thread based

Cilk Plus

Cilk Plus also has a task-based view of computation (like TBB), rather than thread based

This means the programmer thinks about what tasks need to be done, and Cilk Plus thinks about the best way of assigning those tasks to threads

Cilk Plus

Cilk Plus also has a task-based view of computation (like TBB), rather than thread based

This means the programmer thinks about what tasks need to be done, and Cilk Plus thinks about the best way of assigning those tasks to threads

It targets roughly the same area as OpenMP

Cilk Plus

Cilk Plus also has a task-based view of computation (like TBB), rather than thread based

This means the programmer thinks about what tasks need to be done, and Cilk Plus thinks about the best way of assigning those tasks to threads

It targets roughly the same area as OpenMP

And similar to OpenMP, the number of threads used and the threading mechanisms are mostly hidden from the programmer

Cilk Plus

```
int fib (int n) {
   if (n < 2) return n;
   int x, y;
   x = cilk_spawn fib(n-1);// fork
   y = fib(n-2);
   cilk_sync;              // join
   return x+y;
}
```

(from the Cilk Plus website)

Cilk Plus

- Cilk Plus has just three main keywords: `cilk_spawn`, `cilk_sync` and `cilk_for`

Cilk Plus

- Cilk Plus has just three main keywords: `cilk_spawn`, `cilk_sync` and `cilk_for`
- So is much simpler than OpenMP

Cilk Plus

- Cilk Plus has just three main keywords: `cilk_spawn`, `cilk_sync` and `cilk_for`
- So is much simpler than OpenMP
- And more lightweight to use

Cilk Plus

- Cilk Plus has just three main keywords: `cilk_spawn`, `cilk_sync` and `cilk_for`
- So is much simpler than OpenMP
- And more lightweight to use
- And seemingly less flexible: but Cilk Plus provides other mechanisms for more advanced control

Cilk Plus

- Cilk Plus has just three main keywords: `cilk_spawn`, `cilk_sync` and `cilk_for`
- So is much simpler than OpenMP
- And more lightweight to use
- And seemingly less flexible: but Cilk Plus provides other mechanisms for more advanced control
- Ignoring the keywords leaves a valid equivalent sequential C program

Cilk Plus

- Cilk Plus has just three main keywords: `cilk_spawn`, `cilk_sync` and `cilk_for`
- So is much simpler than OpenMP
- And more lightweight to use
- And seemingly less flexible: but Cilk Plus provides other mechanisms for more advanced control
- Ignoring the keywords leaves a valid equivalent sequential C program

A `cilk_for` indicates a parallelisable `for` loop

Cilk Plus

- Cilk Plus has just three main keywords: `cilk_spawn`, `cilk_sync` and `cilk_for`
- So is much simpler than OpenMP
- And more lightweight to use
- And seemingly less flexible: but Cilk Plus provides other mechanisms for more advanced control
- Ignoring the keywords leaves a valid equivalent sequential C program

A `cilk_for` indicates a parallelisable `for` loop

There is an implicit `cilk_sync` at the exit of every function that contains a spawn

Cilk Plus

Cilk Plus also employs work stealing of tasks, but in a more subtle way than TBB

Cilk Plus

Cilk Plus also employs work stealing of tasks, but in a more subtle way than TBB

In the code

```
cilk_spawn fun1();
fun2();
```

the *current* thread actually starts executing `fun1()`

Cilk Plus

In more detail:

Cilk Plus

In more detail:

- when the current thread reaches the `cilk_spawn` it saves the current continuation (i.e., the point in the code just before the `fun2()`) on its continuation stack

Cilk Plus

In more detail:

- when the current thread reaches the `cilk_spawn` it saves the current continuation (i.e., the point in the code just before the `fun2()`) on its continuation stack
- it then starts executing `fun1()`

Cilk Plus

In more detail:

- when the current thread reaches the `cilk_spawn` it saves the current continuation (i.e., the point in the code just before the `fun2()`) on its continuation stack
- it then starts executing `fun1()`
- when done with that, it pops the continuation stack and starts executing what it finds there: `fun2()` in this example

Cilk Plus

An idle other thread can steal a continuation and start executing it

Cilk Plus

An idle other thread can steal a continuation and start executing it

Thus leading to the initially surprising behaviour that `fun2()` might get stolen, not `fun1()`

Cilk Plus

An idle other thread can steal a continuation and start executing it

Thus leading to the initially surprising behaviour that `fun2()` might get stolen, not `fun1()`

In contrast with TBB, where the current thread pushes `fun1()` and so it is that that can be stolen

Cilk Plus

An idle other thread can steal a continuation and start executing it

Thus leading to the initially surprising behaviour that `fun2()` might get stolen, not `fun1()`

In contrast with TBB, where the current thread pushes `fun1()` and so it is that that can be stolen

TBB implements *child stealing*;
Cilk Plus has *continuation stealing*

Cilk Plus

Manipulating continuations is why Cilk Plus needs compiler support. Child stealing as implemented by TBB is implementable in C++ directly as it is essentially just pushing and popping functions on a queue

Cilk Plus

Manipulating continuations is why Cilk Plus needs compiler support. Child stealing as implemented by TBB is implementable in C++ directly as it is essentially just pushing and popping functions on a queue

The difference is that continuation stealing has better memory use patterns than the child stealing and so tends to give more efficient parallelism

Cilk Plus

Manipulating continuations is why Cilk Plus needs compiler support. Child stealing as implemented by TBB is implementable in C++ directly as it is essentially just pushing and popping functions on a queue

The difference is that continuation stealing has better memory use patterns than the child stealing and so tends to give more efficient parallelism

Exercise Child stealing can have unlimited memory use, while continuation stealing does not. Read about this

Cilk Plus

Whatever the relative merits, OpenMP and Thread Building Blocks have wide recognition while Cilk Plus is quite niche

Cilk Plus

Whatever the relative merits, OpenMP and Thread Building Blocks have wide recognition while Cilk Plus is quite niche

In fact, Intel now has deprecated Cilk Plus in favour of their TBB, which being a purely library-based mechanism is easier to support, despite being potentially worse in runtime behaviour

Cilk Plus

Whatever the relative merits, OpenMP and Thread Building Blocks have wide recognition while Cilk Plus is quite niche

In fact, Intel now has deprecated Cilk Plus in favour of their TBB, which being a purely library-based mechanism is easier to support, despite being potentially worse in runtime behaviour

Exercise Read about the many other parts of Cilk Plus, such as *vector sections*

Cilk Plus

Whatever the relative merits, OpenMP and Thread Building Blocks have wide recognition while Cilk Plus is quite niche

In fact, Intel now has deprecated Cilk Plus in favour of their TBB, which being a purely library-based mechanism is easier to support, despite being potentially worse in runtime behaviour

Exercise Read about the many other parts of Cilk Plus, such as *vector sections*

Exercise Work through how continuation stealing might execute the `parallel_for` example

Cilk Plus

Whatever the relative merits, OpenMP and Thread Building Blocks have wide recognition while Cilk Plus is quite niche

In fact, Intel now has deprecated Cilk Plus in favour of their TBB, which being a purely library-based mechanism is easier to support, despite being potentially worse in runtime behaviour

Exercise Read about the many other parts of Cilk Plus, such as *vector sections*

Exercise Work through how continuation stealing might execute the `parallel_for` example

Exercise Compare Cilk Plus, OpenMP, and TBB

Cilk Plus and OpenMP

Exercise Later versions of OpenMP supports *tasks*, which are quite similar in use to Cilk Plus:

```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
#pragma omp task shared(x)
  x = fib(n-1);
  y = fib(n-2);
#pragma omp taskwait
  return x+y;
}
```

Read about tasks, and compare with Cilk Plus

Yet More Threads

We now give, as an alternative view to POSIX, a sketch of how threads are natively supported in a few languages, though this could be argued to be more properly in the "design of a language" part of the unit

Yet More Threads

We now give, as an alternative view to POSIX, a sketch of how threads are natively supported in a few languages, though this could be argued to be more properly in the “design of a language” part of the unit

First, C++

C++ Threads

While C++ can use POSIX threads it has defined — as part of the language specification — its own threads

C++ Threads

While C++ can use POSIX threads it has defined — as part of the language specification — its own threads

Which are often implemented on top of POSIX threads, but are more C++ in the way they are used

C++ Threads

While C++ can use POSIX threads it has defined — as part of the language specification — its own threads

Which are often implemented on top of POSIX threads, but are more C++ in the way they are used

The C++ specification replicates the usual primitives, including thread creation, mutexes, condition variables and so on, but tidying things up a bit to make them more ergonomic and C++-like

C++ Threads

While C++ can use POSIX threads it has defined — as part of the language specification — its own threads

Which are often implemented on top of POSIX threads, but are more C++ in the way they are used

The C++ specification replicates the usual primitives, including thread creation, mutexes, condition variables and so on, but tidying things up a bit to make them more ergonomic and C++-like

Described as “a restricted/simplified subset of POSIX functionality”

C++ Threads

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

std::mutex mut;

void show(const std::string msg, int *n) {
  std::cout << msg << " ";
  // create a lock guard object on the mutex; ownership of
  // the guard is the lock
  std::lock_guard<std::mutex> lock(mut);
  *n += 1; // protected critical region
}
// lock guard deleted at end of scope by
// normal C++ destructor method; thus releasing lock
```

C++ Threads

```
int main() {
  int m = 0;

  std::thread thr1(show, "hello", &m);
  std::thread thr2(show, "world", &m);

  thr1.join();
  thr2.join();

  std::cout << "\nm = " << m << "\n";

  return 0;
}
```

C++ Threads

Producing

```
hello world
m = 2
```

or

```
world hello
m = 2
```

C++ Threads

C++ threads, while mostly similar to POSIX, are closely tied into the rest of the design of C++, thus certain behaviours are better defined

C++ Threads

C++ threads, while mostly similar to POSIX, are closely tied into the rest of the design of C++, thus certain behaviours are better defined

For example, it is not clear how C++'s exception mechanism interacts with POSIX threads, while C++ threads specify a behaviour

C++ Threads

C++ threads, while mostly similar to POSIX, are closely tied into the rest of the design of C++, thus certain behaviours are better defined

For example, it is not clear how C++'s exception mechanism interacts with POSIX threads, while C++ threads specify a behaviour

And they are portable even if there is no (or poor) POSIX support, e.g., Windows

C Threads

In a similar way, the C11 standard for C also has some language support for threads, though it is optional and not universally supported, e.g., not supported by MS at the moment

C Threads

In a similar way, the C11 standard for C also has some language support for threads, though it is optional and not universally supported, e.g., not supported by MS at the moment

It defines types `thrd_t`, `mtx_t`, `cnd_t` and so on

C Threads

In a similar way, the C11 standard for C also has some language support for threads, though it is optional and not universally supported, e.g., not supported by MS at the moment

It defines types `thrd_t`, `mtx_t`, `cnd_t` and so on

It is essentially pthreads with everything that might be non-portable across *all* architectures removed

C Threads

In a similar way, the C11 standard for C also has some language support for threads, though it is optional and not universally supported, e.g., not supported by MS at the moment

It defines types `thrd_t`, `mtx_t`, `cnd_t` and so on

It is essentially pthreads with everything that might be non-portable across *all* architectures removed

C++ threads are widely used, but C11 threads are not, even though they are supported by `gcc` and `clang`

C Threads

In a similar way, the C11 standard for C also has some language support for threads, though it is optional and not universally supported, e.g., not supported by MS at the moment

It defines types `thrd_t`, `mtx_t`, `cnd_t` and so on

It is essentially pthreads with everything that might be non-portable across *all* architectures removed

C++ threads are widely used, but C11 threads are not, even though they are supported by `gcc` and `clang`

Perhaps ingrained use of pthreads, or lack of perception of benefit of using C11 threads?

C Threads

In a similar way, the C11 standard for C also has some language support for threads, though it is optional and not universally supported, e.g., not supported by MS at the moment

It defines types `thrd_t`, `mtx_t`, `cnd_t` and so on

It is essentially pthreads with everything that might be non-portable across *all* architectures removed

C++ threads are widely used, but C11 threads are not, even though they are supported by `gcc` and `clang`

Perhaps ingrained use of pthreads, or lack of perception of benefit of using C11 threads?

Exercise Read about `threads.h` and `stdatomic.h`

Java Threads

Next: Java. It's all based on objects, of course

Java Threads

Next: Java. It's all based on objects, of course

There are two basic ways to create threads in Java:

- as an instance of a subclass of the `Thread` class
- by providing a method for the `Runnable` interface

Java Threads

```
public class Hello extends Thread {
    public void run() {
        System.out.println("Hello world!");
    }
    public static void main(String args[]) {
        Hello t = new Hello();
        t.start();
    }
}
```

Your classes need to be subclasses of the Thread class

Java Threads

```
public class Hello extends Thread {
    public void run() {
        System.out.println("Hello world!");
    }
    public static void main(String args[]) {
        Hello t = new Hello();
        t.start();
    }
}
```

Your classes need to be subclasses of the Thread class

The initial function is the `run` method, which will be called when we execute `start` inherited from `Thread`

Java Threads

```
public class Hello extends Thread {
    public void run() {
        System.out.println("Hello world!");
    }
    public static void main(String args[]) {
        Hello t = new Hello();
        t.start();
    }
}
```

Your classes need to be subclasses of the Thread class

The initial function is the `run` method, which will be called when we execute `start` inherited from `Thread`

A thread can be created, but won't start running until we invoke its `start` method: sometimes separating creation from execution is useful

Java Threads

This way is somewhat constricting in use, as it requires you to design your classes around the `Thread` class

Java Threads

This way is somewhat constricting in use, as it requires you to design your classes around the `Thread` class

So Java gives an alternative way by providing a `Runnable` interface, which you can add to your existing classes

Java Threads

```
public class Hello implements Runnable {
    ...
    public void run() {
        System.out.println("Hello world!");
    }
    public static void main(String args[]) {
        Thread t = new Thread(new Hello());
        t.start();
    }
}
```

`Runnable` requires a `run` method

Java Threads

```
public class Hello implements Runnable {
    ...
    public void run() {
        System.out.println("Hello world!");
    }
    public static void main(String args[]) {
        Thread t = new Thread(new Hello());
        t.start();
    }
}
```

`Runnable` requires a `run` method

The new instance of our class is passed to the `Thread` constructor, which has a `start` method as before

Java Threads

There are `join` methods on `Thread` that wait for thread completion: `join()` and `join(long ms)` and `join(long ms, int ns)`

Java Threads

There are `join` methods on `Thread` that wait for thread completion: `join()` and `join(long ms)` and `join(long ms, int ns)`

Simply returning from `main` waits for threads (actually: non-*daemon* threads)

Java Threads

There are `join` methods on `Thread` that wait for thread completion: `join()` and `join(long ms)` and `join(long ms, int ns)`

Simply returning from `main` waits for threads (actually: non-*daemon* threads)

Explicitly calling `System.exit` does not wait

Java

Java also has higher-level support for parallelism in constructs like *parallel streams* that run concurrently

Java

Java also has higher-level support for parallelism in constructs like *parallel streams* that run concurrently

These fall into the class of “sequential code using parallel operations written by someone else”

Java

Java also has higher-level support for parallelism in constructs like *parallel streams* that run concurrently

These fall into the class of “sequential code using parallel operations written by someone else”

Though they still have the problem of being non-trivial to use correctly

Java

Java also has higher-level support for parallelism in constructs like *parallel streams* that run concurrently

These fall into the class of “sequential code using parallel operations written by someone else”

Though they still have the problem of being non-trivial to use correctly

Exercise Read about *Akka*, a Scala/Java framework for concurrency based on *actors*

Python

And Python...

Python

And Python…

Python was designed without parallel support, and typical implementations of the Python interpreter are strongly not-parallel

Python

And Python…

Python was designed without parallel support, and typical implementations of the Python interpreter are strongly not-parallel

Python supports concurrency, but not parallelism

Python

From the docs:

> *The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the global interpreter lock or GIL, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.*

Python

So, practically speaking, doing anything in Python is necessarily wrapped by a lock

Python

So, practically speaking, doing anything in Python is necessarily wrapped by a lock

You can get some benefit from using process-based parallelism (`import multiprocessing`), where each process has its own separate Python interpreter, but this is quite heavyweight

Python

So, practically speaking, doing anything in Python is necessarily wrapped by a lock

You can get some benefit from using process-based parallelism (`import multiprocessing`), where each process has its own separate Python interpreter, but this is quite heavyweight

The best approach is to call parallel library code written in C, for example

JavaScript

JavaScript is another language that has single threaded interpreters

JavaScript

JavaScript is another language that has single threaded interpreters

Exercise Read about how it uses *Web Workers* to provide parallelism

Go

Go (Golang) has its own kind of threads

Go

Go (Golang) has its own kind of threads

Here threads are called *goroutines*, and are very lightweight (minimal creation overhead) and are managed by the Go runtime

Go

Go (Golang) has its own kind of threads

Here threads are called *goroutines*, and are very lightweight (minimal creation overhead) and are managed by the Go runtime

Note the management is by the Go runtime, not the OS

Go

Go (Golang) has its own kind of threads

Here threads are called *goroutines*, and are very lightweight (minimal creation overhead) and are managed by the Go runtime

Note the management is by the Go runtime, not the OS

The Go runtime gets parallelism by scheduling the goroutines across OS threads

Go

Go (Golang) has its own kind of threads

Here threads are called *goroutines*, and are very lightweight (minimal creation overhead) and are managed by the Go runtime

Note the management is by the Go runtime, not the OS

The Go runtime gets parallelism by scheduling the goroutines across OS threads

Creating new goroutines is very easy — actually encouraged — and you can create “1000s” of goroutines

Go

Go (Golang) has its own kind of threads

Here threads are called *goroutines*, and are very lightweight (minimal creation overhead) and are managed by the Go runtime

Note the management is by the Go runtime, not the OS

The Go runtime gets parallelism by scheduling the goroutines across OS threads

Creating new goroutines is very easy — actually encouraged — and you can create “1000s” of goroutines

And it is OK for them to be short lived

Go

Creating a new goroutine:

```
go fun(x+y, x-y)
```

evaluates the arguments and then creates a new asynchronous goroutine running `fun` with the values of those arguments

Go

However:

Go

However:

- Go provides no particular protection against races; it does provide mutexes and so on, but the programmer must remember to use them (or avoid sharing mutable state)

Go

However:

- Go provides no particular protection against races; it does provide mutexes and so on, but the programmer must remember to use them (or avoid sharing mutable state)
- the runtime that manages the goroutines is quite complex, so Go is less amenable to small or embedded systems

Go

However:

- Go provides no particular protection against races; it does provide mutexes and so on, but the programmer must remember to use them (or avoid sharing mutable state)
- the runtime that manages the goroutines is quite complex, so Go is less amenable to small or embedded systems
- Go is a garbage collected language, so has that complexity in the runtime, too, e.g., having to stop *all* threads during a GC

Go

However:

- Go provides no particular protection against races; it does provide mutexes and so on, but the programmer must remember to use them (or avoid sharing mutable state)
- the runtime that manages the goroutines is quite complex, so Go is less amenable to small or embedded systems
- Go is a garbage collected language, so has that complexity in the runtime, too, e.g., having to stop *all* threads during a GC

Exercise Find out about the current state of Go with regards to GC and parallelism

Go

Go is a well-designed, popular language, but in terms of parallelism is stuck in the mindset of taking a sequential language and adding parallelism and hoping things will be OK

Go

Go is a well-designed, popular language, but in terms of parallelism is stuck in the mindset of taking a sequential language and adding parallelism and hoping things will be OK

Parallelism is *not* an add-on!

Go

Go is a well-designed, popular language, but in terms of parallelism is stuck in the mindset of taking a sequential language and adding parallelism and hoping things will be OK

Parallelism is *not* an add-on!

All these languages (Go, C++, Java, C, etc.) provide mechanism, but no (or insufficient) analysis for concurrency