More Libraries

We now give, as an alternative view to POSIX, a sketch of how threads are natively supported in C++ and in Java, though this could be argued to be more properly in the “design of the language” section
More Libraries

We now give, as an alternative view to POSIX, a sketch of how threads are natively supported in C++ and in Java, though this could be argued to be more properly in the “design of the language” section

First, C++
C++ Threads

While C++ can use POSIX threads, just like C, it has defined — as part of the language specification — its own threads.
C++ Threads

While C++ can use POSIX threads, just like C, 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, just like C, 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

```cpp
#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
    std::lock_guard<std::mutex> lock(mut);
    *n += 1;
    // lock guard released at end of scope by
    // normal C++ destructor method
}
```
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 << "\";

    return 0;
}
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

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.

Exercise. In a similar way, the C11 standard for C also has some language support for threads, though it is optional and not widely supported. 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 will need to be subclasses of the Thread class
Java Threads

```java
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 will 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`
Your classes will 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.
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

```java
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
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)
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 also has higher-level support for parallelism in constructs like *parallel streams* that run concurrently.
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 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 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*.
And Python...
And Python...

Python was designed without parallel support, and typical implementations of the Python interpreter are strongly not-parallel
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.*
So, practically speaking, doing anything in Python is necessarily wrapped by a lock
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 interpreter, but this is quite heavyweight.
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 interpreter, but this is quite heavyweight.

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