Chapter 1. Concurrency

Concurrency is doing multiple things at the same time (not necessarily simultaneously, possible just switching between several tasks). *Parallelism* is kind of concurrency when multiple things are done simultaneously.

D has good support for different kinds of concurrency, including parallelism.

True parallelism is possible with multicore CPUs. When (here and below) speaking about cores, I mean *virtual cores*. Single CPU core may consist of several virtual cores (it is called *hyperthreading*), that is sometimes a single core executes multiple tasks in parallel.

When several tasks (*threads*) are executed simultaneously, it is called *multithreading*. If threads are memory-isolated from each other (to prevent them from harming each other by their bugs or by intention), such threads are called *processes*. Sometimes another terminology is used: a *process* is a memory-isolated group of threads working for a common purpose. Sometimes the terms *process* and *threads* are used even in other slightly different meanings, but the exact details of what is a process and what is a thread do not matter for us.

Widely used modern operating systems can execute more threads or processes than the number of cores (even if there is

just one core) by switching between several threads many times per second.

In the most widely used implementation of Python, CPython, there is no true parallelism. In CPython several Python threads work inside one OS thread switching between these Python threads many times per second. (So if there is just one CPU core, a Python may be one of the OS threads that are quickly switching, and moreover inside this OS thread there may be several quickly switching Python threads).

So, Python is not a great choice for parallelism: Except if you use several Python processes (what is inconvenient and inefficient), you can use only one CPU core by a Python program, the rest cores are uselessly idle). D on the other hand has full support for both single-core and multi-core programming.

However, as far as I know, all modern processors are done in such a way, that except if you explicitly instruct the computer to switch between multiple cores, they use only one core. So far, all our programs were single-core (however, it is possible that library routines, like `writeln` may use multiple cores without you having to instruct them or even taking this into account).

In this chapter we will explain how to use D multithreading (a kind of parallelism) and other D concurrency features.

async/await

Modern Python has so called async/await feature built-in into the Python syntax. In D there is no built-in async/await, but don't mind: there are open source D libraries that implement features similar to async/await of Python or JavaScript.

There are at least two such libraries. We will consider both.

async/await in Python is basically switching between several tasks happening when a task starts or pauses execution (generally *not* many times per a second as in multithreading). These tasks execute in one thread. This is just like having multiple threads but they don't execute in parallel (neither they necessarily quickly switch between each other).

async/await is a kind of *collaborative concurrency*: switch from the executing task to another task may happen only when the executing task itself requests the switch to happen (so they say that tasks "collaborate" with each other). Collaborative tasks are also called *fibers*.

They implement async/await by having several call stacks (state of which functions call which functions and what are the values of their local variables) of execution at once. When the tasks switch, the CPU starts to use another stack.

`dawait`

The first library we will consider is `dawait` (https://code.dlang.org/packages/dawait). It has collaborative concurrency async/await just like as in Python, but it also

support parallelism. The interface is quite simple: There is just one module with three functions.

Function	Description
`startScheduler(void delegate() callback)`	Starts the scheduler with an initial task.
`async(void delegate() callback)`	Runs the given delegate in a separate fiber.
`await(lazy T task)`	Runs the expression in a separate thread. Once the thread has completely, the result is returned.

Code example:

```
import std.stdio;

int calculateTheAnswer() {
  import core.thread : Thread;
  Thread.sleep(5.seconds);
  return 42;
}

void doTask() {
  writeln("Calculating the answer to life, the universe,
and everything...");
  int answer = await(calculateTheAnswer());
  writeln("The answer is: ", answer);
}

void main() {
  startScheduler({
    doTask();
  });
}
```

Deficiencies of this library:

- `async` and `await` miss having the same (that is compatible with each other) interface.

- You need to call somewhere `startScheduler` manually to use `async`. If several libraries used by your project each call `startScheduler`, they cannot use `dawait-.async` concurrently.
- It is not documented and not tested in the unittests what happens in the case of an exception.
- There are no advanced features, like a function that waits for several futures to complete.

`future`

The second library we will consider is `future` (https://code.dlang.org/packages/future).

Unlike Python and `dawait`, the async/await of `future` library uses multiple cores.

It can be used so:

```
auto x = async!((y,z) => y + z)(1,2);
x.await;
assert(x.result.success == 3);
```

Note that except of simple examples like the above that use only pure functions, you will need to use **`shared`** variables and other means of synchronization between threads (see below), when using this library.

`async!f` will execute `f` in a separate thread when invoked. (You can also pass arguments to `f`.)

While `f` is being computed, `c.isPending`.

If `f` succeeds, `c.isReady` and `b == c.result.success`.

If `f` throws, `c.isReady` and `c.result.failure` is inhabited with the caught error.

Attempting to read `c.result` before `c.isReady` is an error.

`async!f` can be made blocking at any time with `c.await`. In other words, `c.await` awaits till `f` finishes executing.

For convenience, `c.await` returns `c` so that `c.await.result == c.result`.

`a.next!f(c)` chains two futures (`a` and `f(a.result, c)`) into a single future.

`a.then!f(c)` returns a future which contains `f(a.result, c)` when `a` is ready.

`when(a1, …, aN)` is a future `Tuple(a1.result, …, aN.result)`, ready when all `aI` are ready.

`race(a1, …, aN)` is a future `Union(a1.result, …, aN.result)`, ready when some `aI` is ready.

Remark: `Union` is defined in https://code.dlang.org/packages/universal.

Deficiencies of this library:

- Under very high load (or because of a software bug), many threads may be created simultaneously overloading the OS and so becoming very slow and possibly leaving you no other choice than to hard-reboot the computer (unless you prevent such software of running

by limiting the number of threads using OS settings). Ideally, a library like this should support a so called thread-pool that is a mechanism to limit the maximum number of used threads and execute the tasks synchronously if this number is exceeded; but this is not yet done by D programmers.

- As usually with parallel programming, the execution becomes indeterministic (you may get a different result every time you run the program), so making debugging sometimes very hard.
- Exceptions are not automatically propagated to the caller, you need to check exceptions manually.

Shared variables

When you use parallelism, you may need variables shared between several threads.

If you don't declare a variable as **`shared`** (or **`__gshared`**) it is *thread-local* that is each thread would have a separate copy of this variable. (If you declare one global or static non-shared variable and start N threads, you would actually have N copies of this variable.)

On the other hand, if you declare a variable as **`shared`**, there is only one copy of this variable for the entire process (that may consist of several threads). Moreover, **`shared`** modifiers makes a variable *atomic* that is it can't be written to or read

"by parts": If, for example, two threads store two different values into a shared variable simultaneously, if can't happen that it gets a mix of some bytes from a value from one thread and some bytes from a value from another thread.

Remark: Because several fibers may run in one thread, they share the same data and **`shared`** is not necessary when you base your concurrency on fibers not threads.

So, D provides safety of atomic reads and writes to a variable in both cases if a variable is non-shared (because only one thread can access this variable) and shared variable (because the compiler warrants that despite several threads can access it, the values don't "intermix").

Remark: You can break this safety by using type conversions: you can for example case a shared variable to a non-shared type, then several threads could be able to write to it simultaneously and the values from several threads would possibly intermix in an unpredictable way.

Example:

```
shared int zeroOrMinusOne = 0;

void f() {
  zeroOrMinusOne = 0;
}

void g() {
  zeroOrMinusOne = -1;
}
```

You could simultaneously call `f` and `g` from two different threads and be sure it is always set to zero or -1 and never to

a mix of bytes from these two values, that would produce a different number.

Remark: In Python all variables are shared.

- When you cast from unshared to shared, make sure there are no other unshared references to that same data.
- When you cast from shared to unshared, make sure there are no other shared references to that same data.

Two above rules may look somehow silly, but they ensure no undefined behavior, but consistent program semantics.

Synchronized code sections

Let

```
shared double[2] twoValues;
```

contains two values which must differ no more than by `1.0`.

Consider the function

```
// Programming error!
void setTwoValues(double first, double second) {
  import std.math : abs;
  assert(abs(first – second) <= 1.0);
  twoValues[0] = first;
  twoValues[1] = second;
}
```

We have declared the variable `twoValues` shared, so all should be OK, right? No! If `setTwoValues` is simultaneously called from more than one thread, it may happen that `twoValues[0]` would get the value from one call of `setTwoValues`

and `twoValues[1]` from another call. Then the condition `abs(twoValues[0] - twoValues[1])` may break.

It should be rewritten as

```
void setTwoValues(double first, double second) {
  import std.math : abs;
  assert(abs(first – second) <= 1.0);
  synchronized {
    twoValues[0] = first;
    twoValues[1] = second;
  }
}
```

`synchronized` code sections may run simultaneously only on one thread, so this would work as intended (values don't intermix).

If our function is a member of a class, then alternatively you could make the entire function `setTwoValues` synchronized (but this would unnecessarily include the **`assert`** statement into the synchronized code section, increasing the latency of the software):

```
class C {
  shared double[2] twoValues;

  synchronized void setTwoValues(double first, double second) {
    import std.math : abs;
    assert(abs(first - second) <= 1.0);
    twoValues[0] = first;
    twoValues[1] = second;
  }
}
```

A synchronized class member function cannot be called simultaneously for the same instance of class from more than one thread.

Or the entire class can be synchronized. All member functions (even static ones, see the specification) of a synchronized class are implicitly synchronized:

```
synchronized class C {
  shared double[2] twoValues;

  void setTwoValues(double first, double second) {
    import std.math : abs;
    assert(abs(first - second) <= 1.0);
    twoValues[0] = first;
    twoValues[1] = second;
  }
}
```

Member fields of a synchronized class cannot be public.

The **synchronized** statement has also the full form with an object in parentheses:

```
class C {
  shared double[2] twoValues;
}

void setTwoValues(C object, double first, double second) {
  import std.math : abs;
  assert(abs(first – second) <= 1.0);
  synchronized(object) {
    object.twoValues[0] = first;
    object.twoValues[1] = second;
  }
}
```

This form provides instead the warranty that the content of synchronized statement is not executed simultaneously for the same object by more than one thread.

Usually, the right way to write synchronized code is to split it into several classes, each class having synchronized member functions, so that the state of the object would be always consistent between calls. (It may be temporarily inconsistent dur-

ing a call, but that does not matter, as other threads cannot access the object while a synchronized method is running.)

Threads

Now let us go to the "real" threads with all the features.

Threads can be run this way:

```
import core.thread;

void threadFunc() {
  … // code executed in the thread
}

auto thread = new Thread(&threadFunc).start();
```

Alternatively you can derive from the class `Thread`:

```
class DerivedThread : Thread {
  this() {
    super(&run);
  }

private:
  void run() {
    …
  }
}

auto thread = new DerivedThread().start();

thread.join(); // waits till the thread completes
```

See the reference about the arguments of `join` as well as about other methods in class `Thread` and classes in this module.

`std.parallelism`

This module defines the class `TaskPool` and global property `taskPool` of this class that encapsulate several threads that

are allocated to accomplish some task by some parallel working threads but not allocating unlimitedly many threads not to overload the system.

D magic allows code like:

```
int calculateTheNumber(int i) {
  return …;
}

auto numbers = new double[10_000_000];

foreach(i, ref elem; taskPool.parallel(numbers)) {
  elem = calculateTheNumber(i);
}
```

that “splits” the array `numbers` into bunches and creates several threads that calculate the numbers in parallel.

The code can be a little shortened

```
auto numbers = new double[10_000_000];

foreach(i, ref elem; parallel(numbers)) { // use the global
task pool taskPool
  elem = calculateTheNumber(i);
}
```

Or you can enqueue a function to be executed when an idle thread in a task pool becomes available:

```
auto t = task!read("foo.txt");
taskPool.put(t);
```

To get a task's result call `yieldForce` on it. It will block until the result is available:

```
auto fileData = t.yieldForce;
```

Fibers

This is similar to Python’s coroutines.

Fibers are implemented by the class `Fiber` defined in the module `core.thread`.

Fibers are like a function split into parts (by calls of `Fiber.yield()`). Each part is called separately:

```
import core.thread;

void foo() {
  writeln("Hello");
  Fiber.yield();
  writeln("World");
}

// ...

auto f = new Fiber(&foo);
f.call(); // prints Hello
f.call(); // prints World
```

For example in `vibe.d` library non-blocking (asynchronous) I/O is implemented using fibers.

Generators

D generators are analogous to Python's generators. A class `Generator` is a range:

```
import std.concurrency;
import std.algorithm;

auto r = new Generator!int({
    foreach (i; 1 .. 5)
        yield(i);
});

r.each!writeln; // prints numbers 1, 2, 3, 4
```

So a generator (they are implemented using fibers) switches stacks when `yield` is called and "returns" values in the middle of executing a function.