



40.317 Lecture 5

Dr. Jon Freeman

2 June 2020

Slido Event #S692

Agenda

- Multithreading
- Unit Testing

Multithreading: Motivation

Why are we studying this topic?

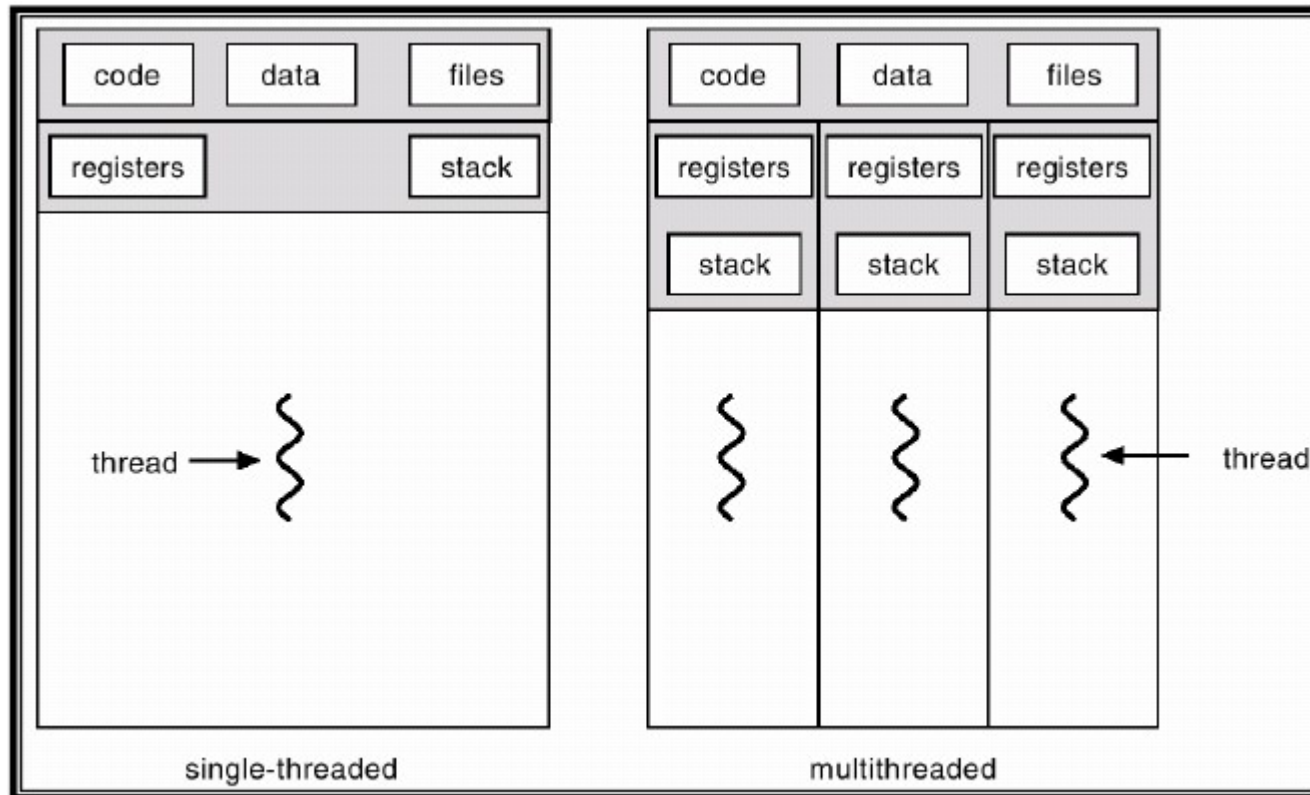
Multithreading is a widely used technique which provides these benefits:

- Faster “throughput”
 - More responsive server processes
 - More responsive client processes
 - Fewer hourglasses! ⌚
- Greater usage of the cores on a multi-core processor

Threads vs. Processes

- Both are *independent sequences of execution*.
- Threads (of the same process) run in the same memory space, whilst processes run in separate memory spaces.
- Threads are much more “lightweight,” i.e. they make minimal use of system resources.
- Threads can share data much more easily than processes.

Threads vs. Processes, continued



Source: <https://sites.google.com/site/sureshdevang/thread-vs-process>

A BETTER WORLD BY DESIGN.



“Concurrency” vs. “Parallelism”

Concurrency is the *composition* of independent processes; parallelism is the *simultaneous execution* of computations.

Concurrency is about *dealing with* lots of things at once; parallelism is about *doing* lots of things at once.

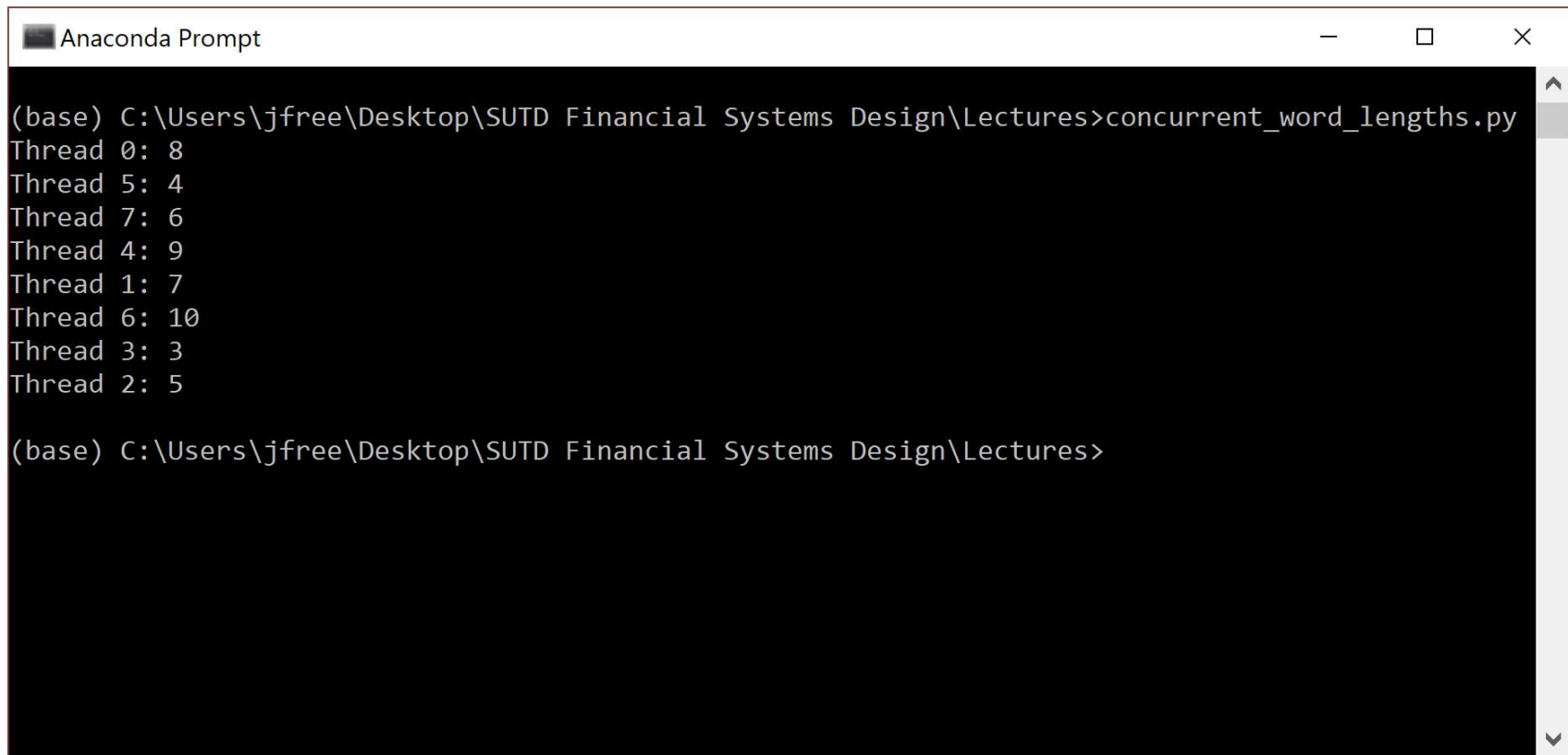
Concurrency is about *structure*; parallelism is about *execution*.

In-Class Discussion

Think back to the trading game demonstrated in the first lecture. Where do you suppose it used threads?

A Simple Working Example

Demo: `concurrent_word_lengths.py`



```
Anaconda Prompt
(base) C:\Users\jfree\Desktop\SUTD Financial Systems Design\Lectures>concurrent_word_lengths.py
Thread 0: 8
Thread 5: 4
Thread 7: 6
Thread 4: 9
Thread 1: 7
Thread 6: 10
Thread 3: 3
Thread 2: 5
(base) C:\Users\jfree\Desktop\SUTD Financial Systems Design\Lectures>
```

A BETTER WORLD BY DESIGN.



In Class Discussion

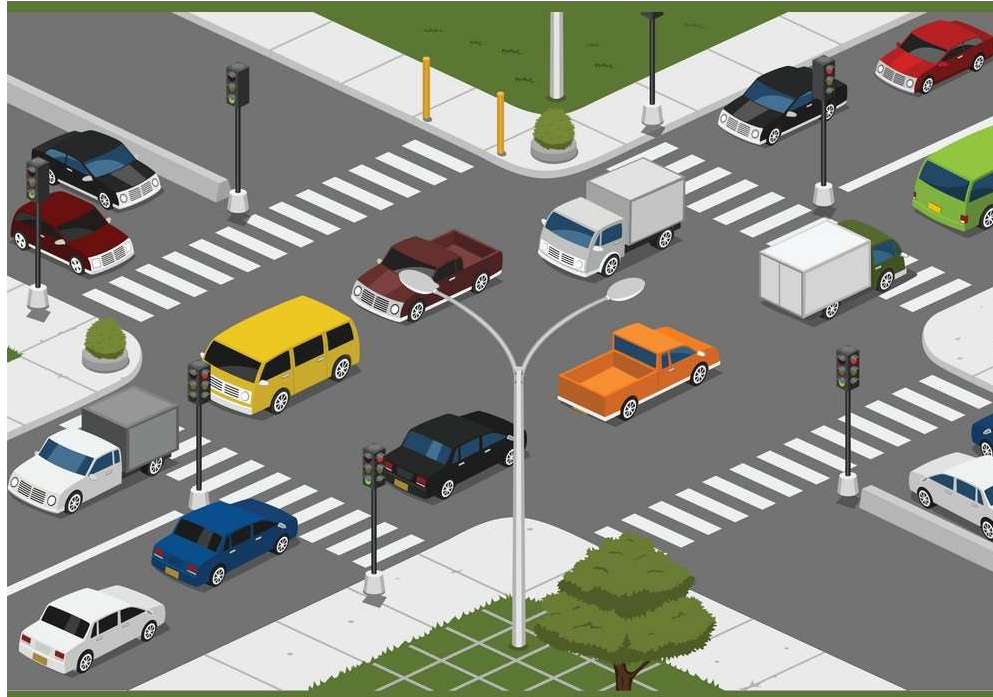
How would we modify this simple example to find the longest word length, using threads?

What could (sometimes) go wrong with this thread-based approach?

What guarantee(s) do we need to ensure this approach will *always* work correctly?

The General Problem

There is a shared resource which multiple threads sometimes need to use, and *only one thread at a time should access it.*



Source: <http://www.vectorstock.com>

A BETTER WORLD BY DESIGN.



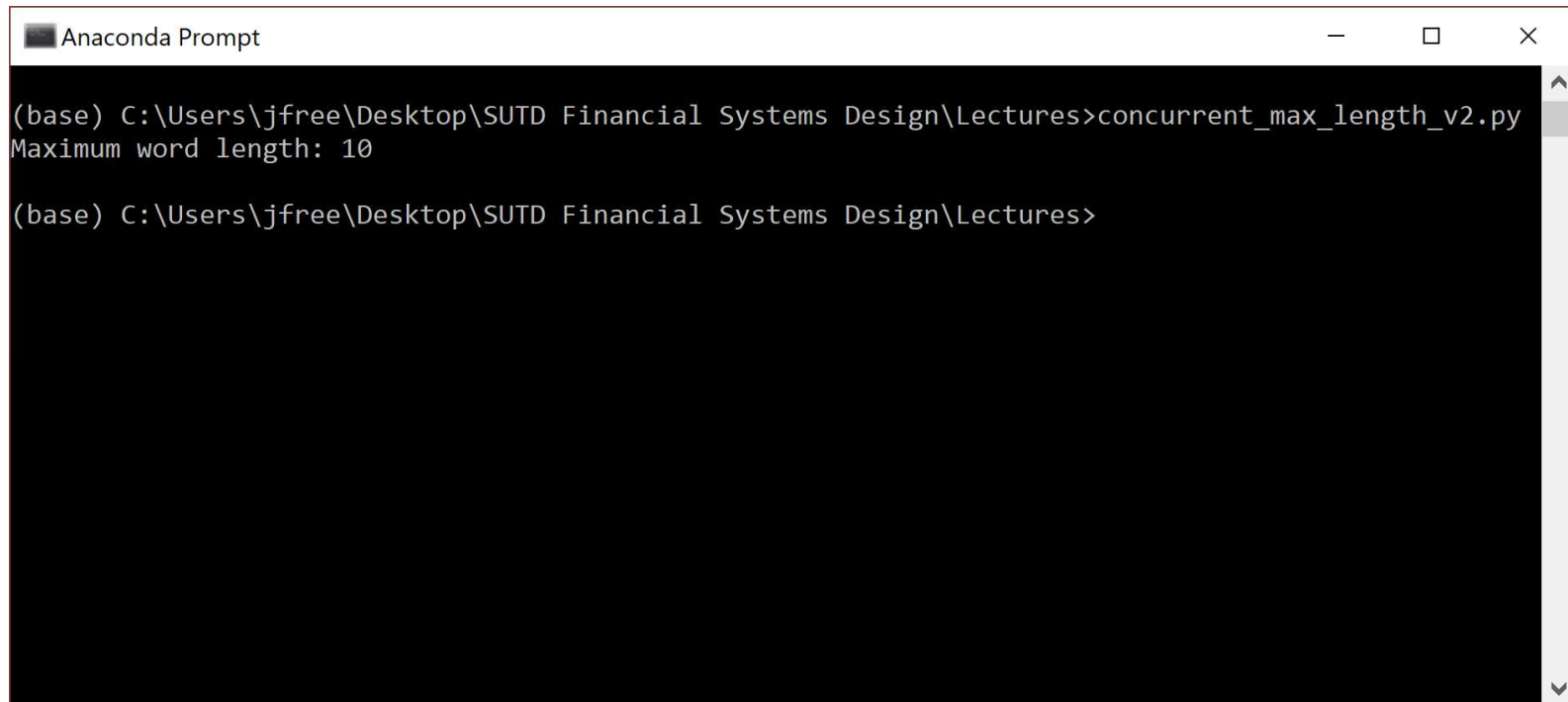
A Simple Solution: Locks

The most straightforward solution is to introduce a simple entity called a lock.

- We create one lock per shared resource.
- To access a resource, a thread must acquire its corresponding lock.
- After using the resource, the thread must release this lock.
- A crucial guarantee: when a lock is available, *only one thread can acquire it*.
- Acquires and releases must be “atomic.”

Our Simple Example with a Lock

Demo: `concurrent_max_length_v2.py`



```
Anaconda Prompt
(base) C:\Users\jfree\Desktop\SUTD Financial Systems Design\Lectures>concurrent_max_length_v2.py
Maximum word length: 10
(base) C:\Users\jfree\Desktop\SUTD Financial Systems Design\Lectures>
```

A BETTER WORLD BY DESIGN.



Locks Introduce New Problems

Brainstorm solutions to the following:

- Reentrancy
 - Thread i has already acquired Lock A, but tries to acquire it again.
- Deadlock
 - Threads i and j each need Locks A and B. i acquires B, and j acquires A. Now what?
- Starvation
 - There are 50 threads fighting for Lock A, and Thread 37 never gets to acquire it even once.

Multithreading in Python 3

- Concurrency vs. parallelism
- Reentrant locks
- Daemon threads
- The `with` statement

Python: Concurrency vs. Parallelism

A surprisingly complicated topic in Python!

- “What *Python* does” vs. “what *CPython* does” is a common source of confusion.
- CPython has a “GIL”, for “Global Interpreter Lock.”
- Due to its GIL, *in CPython only one thread at a time can execute code.*
- I.e. in CPython, threads can provide concurrency but *never* parallelism.
 - Some *libraries* provide parallelism though

Python: Re-entrant Locks

Re-entrant locks are a complete, immediate solution to the re-entrancy problem mentioned earlier.

In Python, just create an `RLock` instead of a `Lock`.

To unlock, a thread's *total* number of calls to `release()` must match its total number of calls to `acquire()`.

Python: Daemon Threads

To create one, set a Thread's `daemon` property to `True` before calling `start()`.

A daemon thread does not need to exit in order for the overall program to exit.

That is, a Python program can exit if all its remaining live threads are daemon threads.

- The daemon threads will not be shut down cleanly, which hopefully doesn't matter

Python: the `with` Statement

- Explicit calls to `acquire()` and `release()` are a ready source of bugs
 - E.g., later on you add a `return` in between, forgetting to add a second `release`
- The `with` statement solves this particular problem completely
- Demo: `concurrent_max_length_v3.py`

Alternatives to Lock and RLock

Besides `Lock` and `RLock`, Python 3 offers several other mechanisms for synchronising threads (“synchronisation primitives”), all out of scope for today:

- Condition Variables
- Semaphores
- Event Objects
- Barrier Objects

A General Caution

There are many reasons to avoid multithreading *with synchronisation*, at least in more complex cases.

Briefly, non-deterministic behaviour is *much* harder to understand, and prove correct, than deterministic behaviour.

For many convincing examples, see http://blog.csdn.net/win_lin/article/details/8274810 (the text of a 2008 MSDN article)

Alternatives to Synchronised Threads

- Redesign to eliminate the need for shared resources, hence synchronisation
 - Demo: `concurrent_max_length_v4.py`
 - c.f. [MapReduce](#), [originally from Google](#)
- Use higher-level abstractions your language provides. Top Python choices:
 - The `multiprocessing` package
 - `Pool` is especially elegant
 - Also see `Process`, `Queue`, `Value`, and `Array`
 - The `concurrent.futures` package

An Example of multiprocessing.Pool

Let's use `multiprocessing.Pool` to estimate π via Monte Carlo integration, obtaining a genuine speedup with even cleaner code.

- `estimating_pi_via_pool.py`

Discussion Question

What are the characteristics of a problem which is well suited for parallelism?

- It can be split into sub-problems.
- *Most of it* can be split into sub-problems.
- Most of it can be split into *largely self-contained* sub-problems.
- Most of it can be split into largely self-contained sub-problems *of similar size*.
- The workers don't need much input data, and don't generate much output data.

Unit Testing

Unit testing is testing of the smallest testable parts of a system – its “units of functionality.”

Unit testing should be:

- *fully automated*
- *performed routinely*

A system's unit tests should be considered first-class components of it.

Unit Testing: Motivation

Why are we studying this topic?

- Unit tests are indispensable for helping to prove your code is correct.
- Thorough unit tests give you confidence to refactor or even rewrite working code.
- Unit tests underpin state of the art practices such as continuous integration (c.f. [GitHub Actions](#)) and [DevOps](#).
 - Passing unit tests can be a “quality gate”

More Motivation: Flickr's 2009 Slides

In 2009, a slide deck from Flickr went viral: it said they were able to change their production system up to *ten times a day (!)*.

Let's take a look:

<https://www.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr>

In Class Discussion

- What are the “units of functionality” in our asset manager application?
- What exactly would we test for each?
- In order to conduct these tests, our server must be running; how do we ensure this?

In Class Discussion, continued

How would we create unit tests for things like:

- Printing a file?
- Sending an email?
- Changing a password?
- Creating an account, with email verification?

Unit Testing in Python 3

In a nutshell:

1. Import the `unittest` package
2. Create a subclass** of `unittest.TestCase`
3. Each unit test is a method** of this subclass** whose name starts with “test”
4. The tests are executed in alphabetical order vis-à-vis these method** names

** *We will define these terms next week*

Unit Tests for our Asset Manager App

Demo: `test_holdings_server.py`

```
Anaconda Prompt
(base) C:\Users\jfree\Desktop\SUTD Financial Systems Design\03 Holdings Manager with VWAP>test_h
oldings_server.py
I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.I am the server, now listening on port 5990
.
-----
Ran 17 tests in 10.032s

OK

(base) C:\Users\jfree\Desktop\SUTD Financial Systems Design\03 Holdings Manager with VWAP>
```

A BETTER WORLD BY DESIGN.



Other Types of Tests

Unit testing is not nearly enough to help ensure the quality of larger applications. The other main types of testing are:

- Integration testing
- Regression testing
 - A special case of unit + integration testing
- (User) acceptance testing
- Performance testing
- Penetration testing

Thank you
A BETTER WORLD BY DESIGN.

