Going Concurrent with Python

Mikko Harju

Taiste

October 18, 2011

About me

- I'm Mikko Harju, Technology Director at Taiste

About me

- I'm Mikko Harju, Technology Director at Taiste
- Python user since 2008

About me

- I'm Mikko Harju, Technology Director at Taiste
- Python user since 2008
- Functional programming enthusiast

Agenda

- We'll look slightly at *threading* and more at *processes* and how do they differ when implementing concurrency in Python.

Threads

- Threads are the smallest unit of processing that can be scheduled (by an operating system).

Threads

- Threads are the smallest unit of processing that can be scheduled (by an operating system).
- They live within the process boundary, sharing the memory space.

Threads

- Threads are the smallest unit of processing that can be scheduled (by an operating system).
- They live within the process boundary, sharing the memory space.
- This makes it trivial to transfer information from one thread to another.

Threads

- To prevent problems when writing data from multiple threads to a shared state variable we need to introduce different kinds of *locking mechanisms*.

Threads

- To prevent problems when writing data from multiple threads to a shared state variable we need to introduce different kinds of *locking mechanisms*.
- Possibilities include e.g. mutexes, semaphores and critical sections.

Threads

- To prevent problems when writing data from multiple threads to a shared state variable we need to introduce different kinds of *locking mechanisms*.
- Possibilities include e.g. mutexes, semaphores and critical sections.
- If the threads only access the data in a read-only way, there is no need for locking.

Threads

- To prevent problems when writing data from multiple threads to a shared state variable we need to introduce different kinds of *locking mechanisms*.
- Possibilities include e.g. mutexes, semaphores and critical sections.
- If the threads only access the data in a read-only way, there is no need for locking.
- This is called Shared-nothing -approach. Erlang has this with its ”lightweight processes“.

GIL

- GIL is Python's locking mechanism to ensure that threads co-operate nicely with non-thread safe constructs

GIL

- GIL is Python’s locking mechanism to ensure that threads co-operate nicely with non-thread safe constructs
- This is done by locking down the interpreter by giving exclusive access to one thread at a time.

GIL

- GIL is Python's locking mechanism to ensure that threads co-operate nicely with non-thread safe constructs
- This is done by locking down the interpreter by giving exclusive access to one thread at a time.
- This effectively means that only one CPU bound thread task is actually doing anything useful in one python interpreter process at a time.

GIL

- GIL is Python's locking mechanism to ensure that threads co-operate nicely with non-thread safe constructs
- This is done by locking down the interpreter by giving exclusive access to one thread at a time.
- This effectively means that only one CPU bound thread task is actually doing anything useful in one python interpreter process at a time.
- When doing I/O, the interpreter can release the lock. The interpreter also has periodic checks to go along with this to make it possible to parallelize CPU bound threads.

GIL

- When we add more processors (or cores) to the mix, things get more complicated.

GIL

- When we add more processors (or cores) to the mix, things get more complicated.
- N threads can be scheduled simultaneously on N processors, making them all compete over the GIL. Nice.

GIL

- When we add more processors (or cores) to the mix, things get more complicated.
- N threads can be scheduled simultaneously on N processors, making them all compete over the GIL. Nice.
- So really, threads are not the right way to go in Python.

GIL

- When we add more processors (or cores) to the mix, things get more complicated.
- N threads can be scheduled simultaneously on N processors, making them all compete over the GIL. Nice.
- So really, threads are not the right way to go in Python.
- There are maybe some use cases where they might come in handy, where the problem is more I/O bound than CPU bound and there is an urgent need to be contained inside a single interpreter.

GIL

- When we add more processors (or cores) to the mix, things get more complicated.
- N threads can be scheduled simultaneously on N processors, making them all compete over the GIL. Nice.
- So really, threads are not the right way to go in Python.
- There are maybe some use cases where they might come in handy, where the problem is more I/O bound than CPU bound and there is an urgent need to be contained inside a single interpreter.
- But let us concentrate on the more fruitful of doing real multiprocessor concurrency on Python: *processes*

Processes

- Processes are OS backed construct. A separate python interpreter is run on each process.

Processes

- Processes are OS backed construct. A separate python interpreter is run on each process.
- Each process has its own memory space, stack, registers and that kind of stuff.

Processes

- Processes are OS backed construct. A separate python interpreter is run on each process.
- Each process has its own memory space, stack, registers and that kind of stuff.
- Scheduling is performed by the operating system.

Let's do this!

- Nothing beats practice, so let's do parallel image processing with PIL using the multiprocessing package and see what we come up with.