The Essentials Of Stackless Python

or: this is the real thing!

A Note About Hardware

- Hardware matters.
  - I learned that after suffering from Stroke since last June
- Sorry, no interactive session today
  - Fingers are still learning to tyzpe
- Restored my brain from backups ☺

- *The show must go on*

Stackless as we know it

- Uses tasklets to encapsulate threads of execution
- Uses channels for control flow between tasklets (ok, also schedule())
  - No direct switching
  - No naming of jump targets
  - Learned that from Limbo language
- http://www.vitanuova.com/inferno/papers/limbo.html

Implementation

- Written in C
- Minimal patch
- Cooperative swiching (soft)
- Brute-force switching (hard)

1) Hard Switching

- Very powerful
  - Hard to know when switching is allowed
  - Not too fast
- Requires assembly
  - GC problems
  - No pickling

2) Soft switching

- The real thing
  - No assembly
  - Ultra-fast
  - Pickling possible
- But hard to implement
  - Needs stackless style
  - Unwind the stack
  - Avoid recursive interpreter call

The Compromize

- C-Stackless uses 90 % soft switching
  - Implemented support for the most commonly used functions
- Patching about 5 % of functions
- The rest is hard switching
- PyPy has shown that 50% needs to change for a complete soft implementation

PyPy: the real Stackless

- Stackless transform
  - Built into the translation chain
  - Stack unwinding under the hood
  - 100 % soft switching
- Relief: never have to write stackless style again ☺
- Stackless features available at low-level

Stackless RPython

- Acts like a C compiler that knows how to unwind/restore
- Convenient, almost pythonic language
- Has a built-in primitive coroutine implementation.
- Coroutines on application level are built on top of RPython coroutines

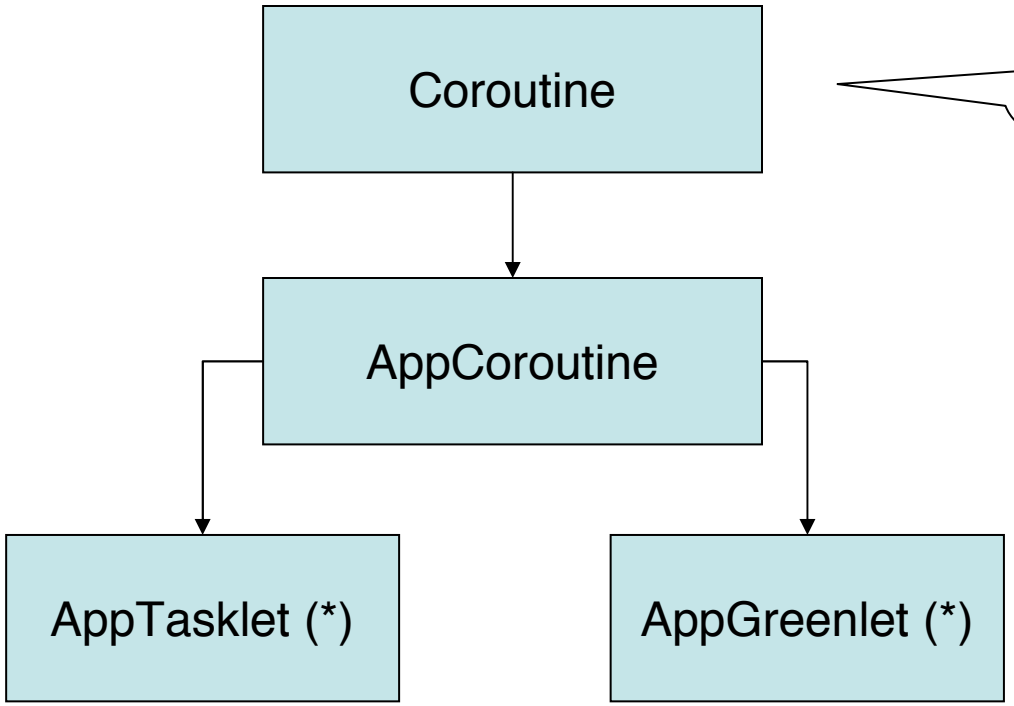
Is That Essential?

- It is not.
- How we switch doesn't matter, whether co-operative, with stack fiddling, or using the Stackless transform.
- It all works.

What is a coroutine?

- Coroutines can „switch“ to each other
- There is always one „current“ coroutine monitored in a costate structure
- Current‘s state is on the machine stack
- Others are stored as a structure
- By switching, we replace „current“ by a different coroutine and update.
- This works multiple times

Class Hierarchy



Interpreter-level only

Exposed as distinct

- coroutine
- greenlet
- tasklet
- … others as needed

(!) Inheritance just for implementation brevity, not exposed to the user

(*) right now done in app-level

Simple API

- c = coroutine()
- c.bind(func, args)
- c.switch()

- c.alive
- c.kill()
- coroutine.getcurrent()

Enough to build everything else on top

‘What Am I’ Problem

- How do we define where a coroutine starts and ends?
- What is ‘current’?
- What is running right now? Am I a coroutine, a tasklet, a greenlet, something else?

You Are What You Switch To

- ‘current’ is never stored.
- There is no switching between concepts.
- The running program is whatever you like.
- You determine what it was when you jump to something else.
- The power of this concept lies in doing nothing at all ☺

How can they co-exist?

- Every coro-class has its own costate singleton instance (could be a class variable if RPython supported it)
- Coro-classes are created with an active instance representing the whole program
- A coro-class‘ current is by definition active until we change this coro-class‘ costate
- Coroutines don‘t see greenlets don‘t see tasklets don‘t see what has a different costate.

Remarks On Generators

- They are only one frame deep
- ‘Who am I’ is simple because it is exactly determined be entering/leaving the single frame

Remarks on Tasklets

- Well isolated by design
- Channels are an abstraction that frees the user from the need to know a jump target
- ‘rendevouz’ point. The addition of transferring data is just for convenience
- Not much more than coroutines plus the automatic jump management

Composability

- By co-states, we can run different concepts at the same time, and there is no overhead added
- We can run different sets of tasklets, grouped by giving them different costates
- We can mix this all

Things To Do for C

- C-Stackless has tasklets, only. Provide coroutines as the basic switching concept.
- Let tasklets inherit from that.
- Make the co-states concept available to allow for multiple concepts

Things To Do For PyPy

- Greenlets and tasklets are pure application-level classes right now. At least tasklets should exist as low-level RPython classes for speed

Better naming

- We don’t have a proper name for co-states, yet. There are many proposals.
- In some sense they are ‘view’s. Different views of the rest of the program, partitioned by belonging to a certain co-state.

Python 3000?

- Is there a way to integrate Stackless into Python 3000?
  - Not sure if I want this. Incompleteness, assembly,… but maybe I'm doing this for too long

Stackless is fully integrated as an option for PyPy. Is this the final solution?

Will PyPy be Python 3000?

Not in the near future, but we will see…