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
    - » Ok, maybe we do a little bit
- 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

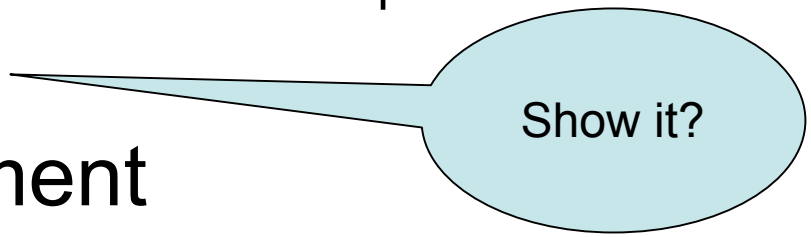
- Written in C
- Minimal patch
- Cooperative switching (soft)
- Brute-force switching (hard)

1) Hard Switching

- Very powerful
  - Hard to know when switching is allowed
  - Not too fast (10 x faster than threads)
- Requires assembly
  - GC problems
  - No pickling possible

2) Soft switching

- The real thing
  - No assembly
  - Ultra-fast (at least 100 x faster than threads)
    - At the order of a generator call's speed
  - Pickling possible
- But hard to implement
  - Needs writing stackless style in C (ugly)
    - Unwind the stack
    - Avoid recursive interpreter call
  - Lots of changes to CPython



```
import pickle, sys
import stackless

ch = stackless.channel()

def recurs(depth, level=1):
    print 'enter level %s%d' % (level*'  ', level)
    if level >= depth:
        ch.send('hi')
    if level < depth:
        recurs(depth, level+1)
    print 'leave level %s%d' % (level*'  ', level)

def demo(depth):
    t = stackless.tasklet(recurs)(depth)
    print ch.receive()
    pickle.dump(t, file('tasklet.pickle', 'wb'))

if __name__ == '__main__':
    if len(sys.argv) > 1:
        t = pickle.load(file(sys.argv[1], 'rb'))
        t.insert()
    else:
        t = stackless.tasklet(demo)(14)
    stackless.run()

# remark: think of fixing cells etc. on the sprint
```

The CPython Compromize

- C-Stackless uses 90 % soft switching
  - Implemented support for the most commonly used functions only
- Patching about 5 % of all functions
  - The rest is still hard switching
- PyPy has shown that 50% needs to change for a complete soft implementation
  - This will probably not happen
- The compromize works fine

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
  - Coroutines at C level possible

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

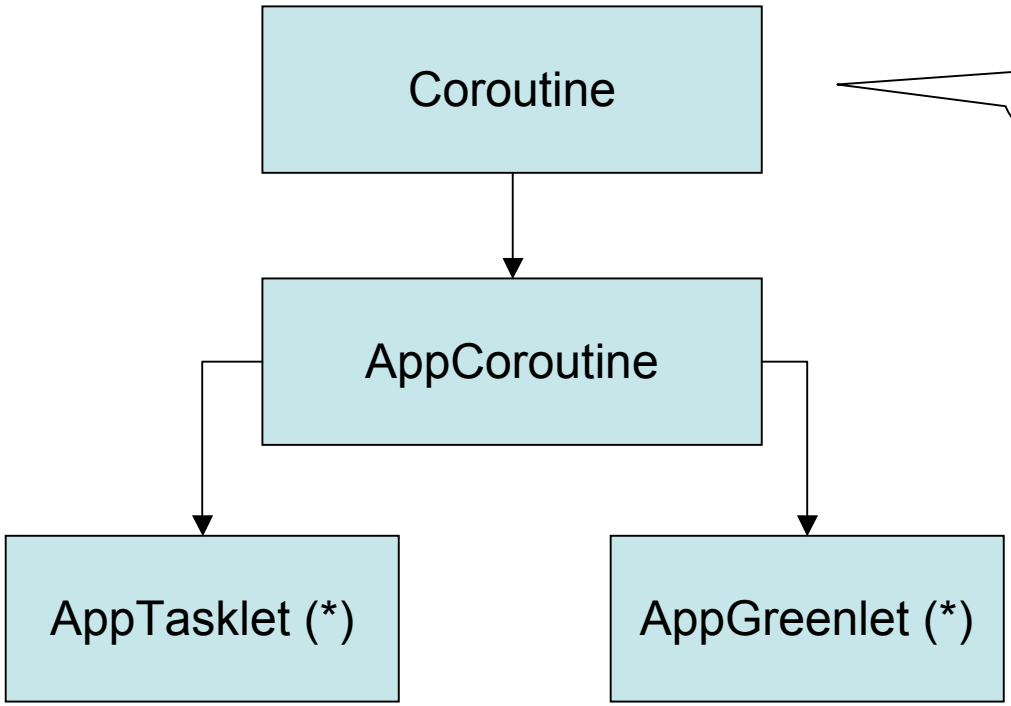
Is That Essential?

- 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 Group structure‘s current
- 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 it‘s group.
  - We‘ll see how this scales

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

‘Who 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?

Remarks On Generators

- They are only one frame level deep
- Special case of coroutine with implicit return target
- ‘Who am I’ is simple because it is exactly determined by 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

Essential Evolution

- Tasklets and generators are special ‘who am I’ solutions
  - *I actually choosed tasklets to avoid the problem*
- Greenlets dealt a bit with it
  - The parent property to organize greenlets
- Coroutines are more basic and needed an explicit concept for maintaining ‘current’
- This led to a general solution!

You Are What You Switch To

- ‘current’ is never stored.
- There is no switching between concepts.
  - Only similar things can be seen.
- The running program is whatever you like.
  - You determine what it was by the jump to something else.
- The power lies in doing nothing at all
  - Just keep track where the history of a jump must be stored

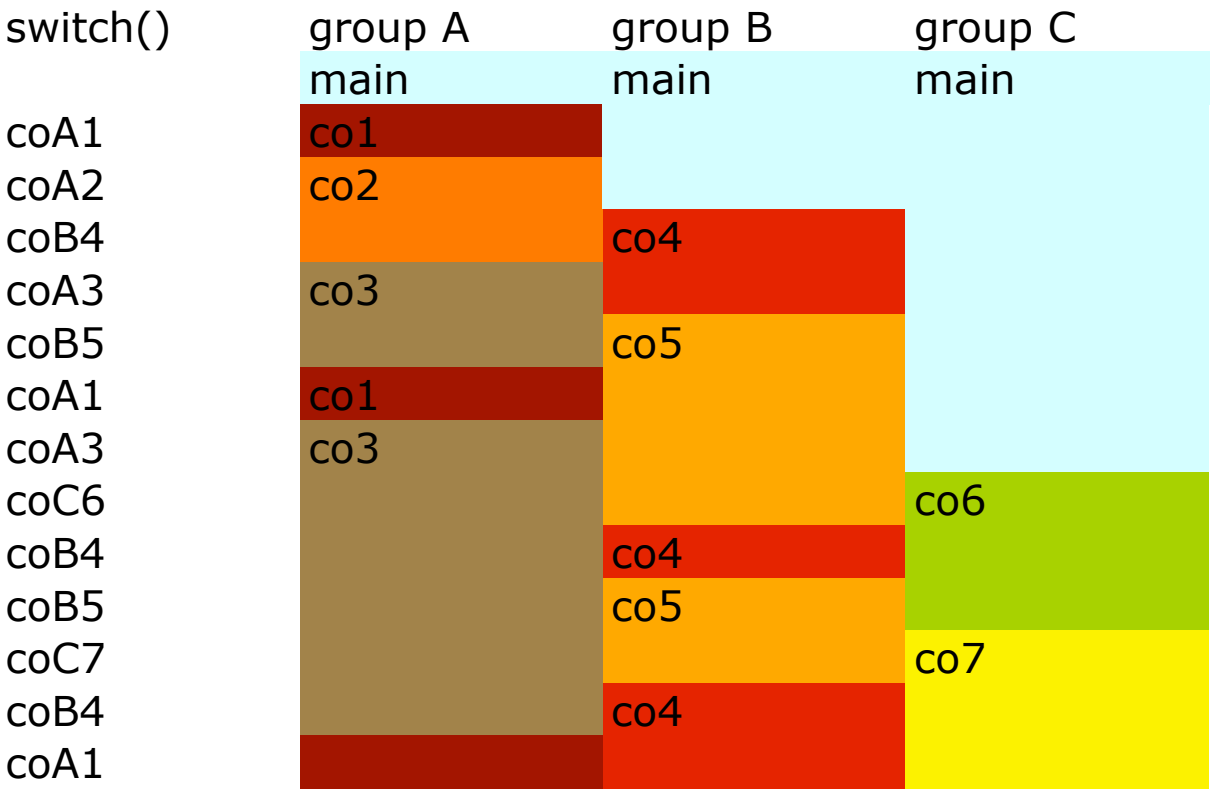
How can things co-exist?

- Every coro-class has its own Group singleton instance
- Coro-classes are created with an active instance representing the whole program
- A coro-class‘ current is by definition active until we update this coro-class‘ Group instance
- Coroutines don‘t see greenlets don‘t see tasklets don‘t see what has a different Group instance.

Finale: Composability

- By views, 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 groups
- We can mix this all, since groups cannot interfere by construction
- Confused? Maybe a picture helps….

Per group view of the world



Note that there is no implied relationship to the actually called functions at all. It is all about switching inside of groups

Things To Do for C

- C-Stackless has tasklets, only. Provide coroutines as the basic switching concept.
- Let tasklets inherit from that.
- Implement Groups 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

Python 3000?

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

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

Will PyPy become the Python 3000?

Not in the near future, but we will see…