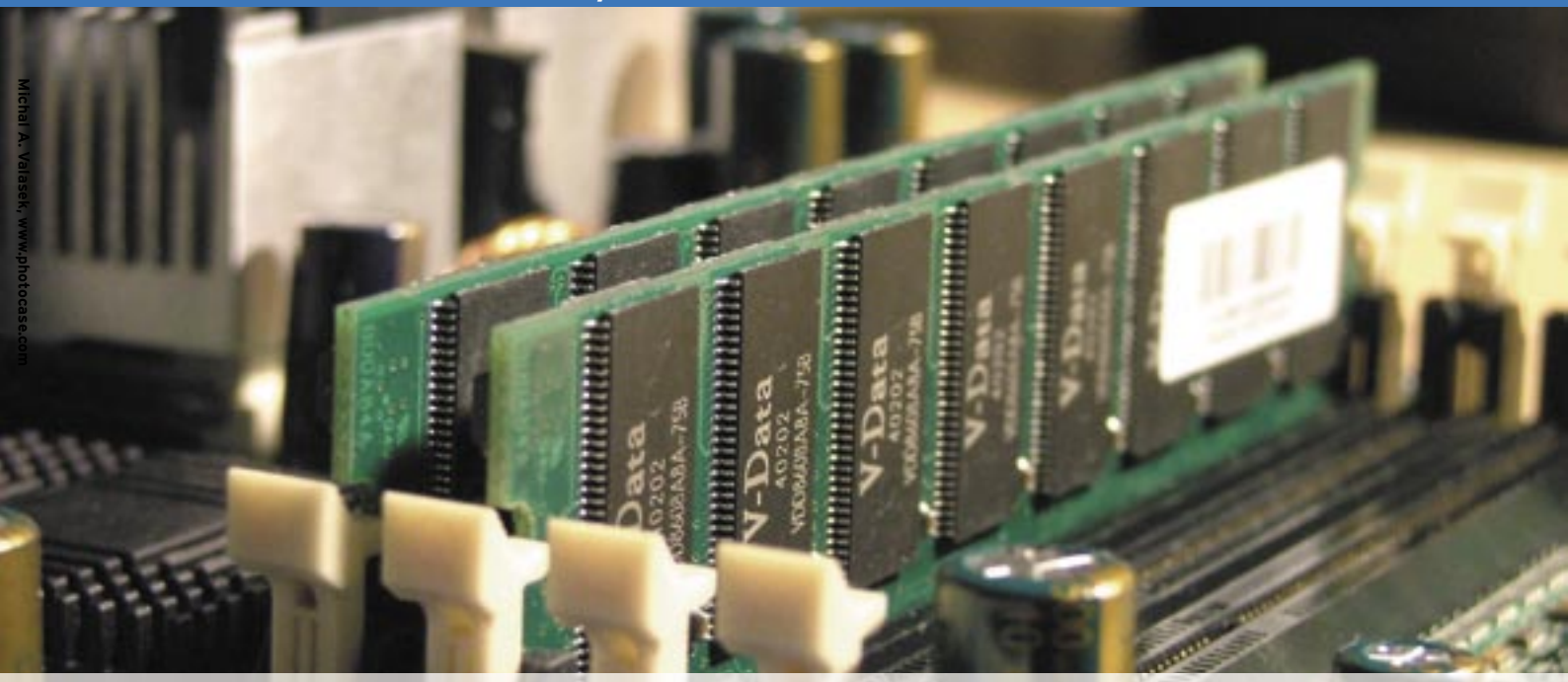

Michal A. Valasek, www.photocase.com

Programming with Stackless Python

IN THE HEAP

The Stackless extension brings lightweight processes to Python, opening a new style of programming with dynamic heap access.

BY STEPHAN DIEHL

Stackless Python [1] by Christian Tismer extends the popular Python interpreter, adding elements that facilitate the development of scalable applications. Small, independent program sections are encapsulated in *tasklets*. These tasklets use channels to communicate in an approach reminiscent of the Erlang [2] and Oz [3] languages. The name *Stackless* refers to the encapsulated functions that tasklets allow to be swapped out from stack [4] to heap [5] (dynamic) memory. Data stored at this location can be accessed at any time, regardless of the order in which it arrived. Figure 1 shows how this approach can save memory, especially with parallel functions.

This architecture gives programmers the ability to use functions as coroutines [6]. Coroutines are characterized by the peer relationships in which they coexist.

Listing 1: First Steps

```
01 Python 2.5 Stackless 3.1b3
   060516 (release25-maint:53626,
   Feb 3 2007, 15:30:37)
02 [GCC 4.0.3 (Ubuntu
   4.0.3-1ubuntu5)] on linux2
03 Type "help", "copyright",
   "credits" or "license" for
   more information.
04 >>> import stackless
05 >>> def f():
06 ...   print "1"
07 ...   stackless.schedule()
08 ...   print "2"
09 ...
10 >>> f_task = stackless.
   tasklet(f)()
11 <stackless.tasklet object at
   0xb7d50e2c>
12 >>> stackless.schedule(None)
13 1
14 >>> stackless.schedule(None)
15 2
```

In contrast to the legacy threads provided by the *Threading* module, several thousand tasklets can run simultaneously. Tasklets are considered lightweight because they can be switched several hundred thousand times per second. If you have a task that relies on this ability, a stackless implementation will

Installation

As of this writing, no Stackless Python binary packages are available for any distribution. You will need to use Subversion to download the current release; the Stackless homepage points to the current address. Versions for Python 2.4 and 2.5 are available. After downloading, just follow the normal steps to build and install:

```
./configure --prefix=➥
/targetdirectory/
make
make install
ln -s /targetdirectory/bin/
python ➥
/usr/local/bin/stackless
```

The softlink avoids possible conflicts with your existing Python installation. For more details, refer to the Stackless documentation on the project homepage.

be far faster than a threaded version. The online role-playing game, Eve Online [7], is a good example of this.

It is no coincidence that CCP [8], the people behind Eve Online, help to keep Stackless Python up to date. Apart from CCP, Ironport [9] also uses Stackless for its network security appliances.

Keeping to the Schedule

Stackless relies on a cooperative scheduler that uses a round-robin approach [10]; that is, it lets every tasklet run in succession for a short while. Although programmers can pretend that the tasklets are actually running parallel to one another, this is not strictly true; this said, there are moves to extend Stackless Python to support parallel execution. For the time being, programmers have to live with the fact that the whole program freezes if a tasklet freezes. Wherever possible, it is critical to avoid system calls that slow down an application such as network or database connections.

The main program relies on the tasklets' cooperative behavior; as soon as a tasklet is called by the scheduler, it has full control of the program flow. The tasklet has two options for returning control to the program:

- making a call to *stackless.schedule()*;
- reading or writing on a channel.

stackless.tasklet(function)() initializes a tasklet and activates it in the scheduler. Listing 1 shows the first steps in the interactive interpreter. This slightly convoluted code gives you the same results:

```
t = stackless.tasklet()
t.bind(f)
# pass in f start parameters:
t.setup()
# Append t to scheduler list:
t.insert()
```

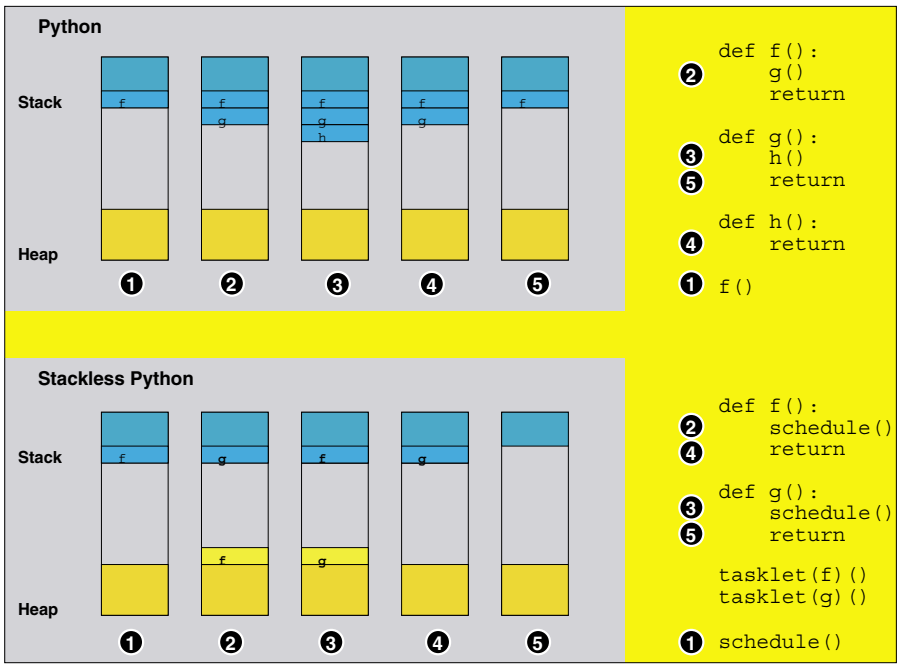

Figure 1: CPython stores the data belonging to a subroutine in stack memory above the function that is superordinate to the subroutine in the hierarchical structure. Stackless Python swaps tasklets out to heap memory.

In legacy threading, the main program is also a thread. In line with this, Stackless has a main tasklet.

In Listing 1, the main tasklet hands control over to *f_task* the first time it calls *stackless.schedule()*. *f_task*'s call to *schedule()* then hands control back to the interactive console. The console waits for input at this point, so nothing happens initially. The scheduler then lets *f_task* run its final command.

Listing 2 shows a simple program that does not really do much more than the previous example. It creates two tasklets, but this time it starts processing them by calling *stackless.run()*.

In contrast to *stackless.schedule()*, the tasklet that is called – the main tasklet in this case – is removed from the scheduler. The call finishes when the scheduler runs out of tasklets, and the parent process is then scheduled (Figure 2). Like all the other examples in this article, the program shown in Listing 2 is available for download at the *Linux Magazine* website [11].

If the main program of Listing 2 were to use *schedule()* instead of *run()*, the results would be different; neither tasklet would even reach the *print c, 'after'* line in *f()* because the main program would exit before running the outstanding tasklets. Listing 3 demonstrates the use of channels with an example of a fairly inefficient sort algorithm. As you can see, Stackless' program flow is deterministic, in contrast to operating system threads.

Only one tasklet runs at any time, which means that genuine parallel pro-

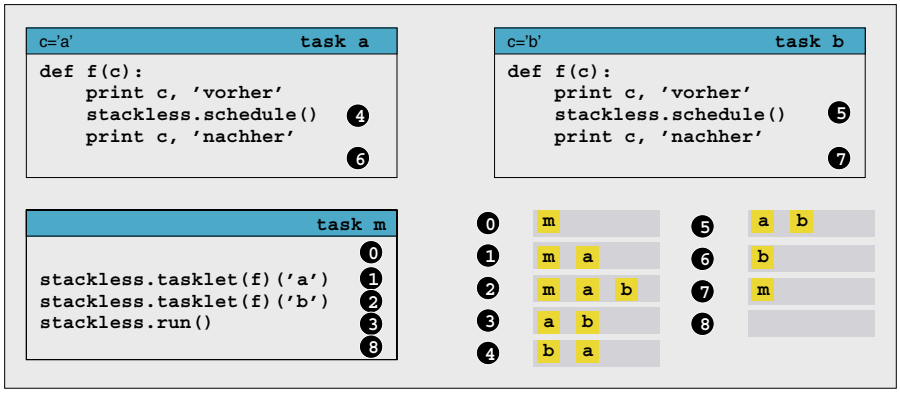

Figure 2: The order in which the program from Listing 2 is executed.

Listing 2: example.p

```
01 import stackless
02
03 def f(c):
04   print c, 'before'
05   stackless.schedule()
06   print c, 'after'
07
08 stackless.tasklet(f)('a')
09 stackless.tasklet(f)('b')
10 stackless.run()
```

cessing does not occur even on multiprocessor systems.

Besides this, tasklets decide themselves when they hand over control of program flow; thus, there are no external factors that the program cannot handle.

network_simulation.py implements a simple network simulation. The main elements are nodes that represent network-attached computers and hubs that connect the computers. A node receives packets and forwards any packets not addressed to it.

A hub receives packets and forwards them to all the devices connected to the network.

To keep things simple, all the objects have just one reception channel. Listing 4 shows the basic structure of the nodes and hubs.

Whether you have 10 or 1000 computers, it makes no difference in the simulation. Each element acts independently of all other elements, with communications simply relying on the messages that they exchange.

When designing program flow, it is important to avoid tasklets blocking each other. In our example, the individual elements block the program flow until a message reaches the control channel.

The tasklets also take a break when they have something to send. This makes the program loop for the hub slightly more complex (see Listing 5).

The hub will only send a message when a node or hub is listening at the other end of the line. The *out.balance < 0* request takes care of this. Additionally, the hub needs to pick up incoming pack-

Listing 3: sort.py

```
01 import stackless
02 import random
03
04 numbers = range(20)
05 random.shuffle(numbers)
06 print numbers
07 print 'Sorting...'
08
09 def counter(n, ch):
10   for i in xrange(n):
11     stackless.schedule()
12   ch.send(n)
13
14 ch = stackless.channel()
15 for each in numbers:
16   stackless.
   tasklet(counter)(each, ch)
17 stackless.run()
18 rlist = []
19 while ch.balance:
20   rlist.append(ch.receive())
21 print rlist
```

Listing 4: The Elements in network_simulation.py

```
01 class Element:
02   def __init__(self, channel):
03     stackless.tasklet(self.
   taskloop)(channel)
04     self.channel = channel
05
06   def taskloop(self):
07     while True:
08       message = self.channel.
   receive()
09       # do something with the
   message
10       [...]
```

Listing 5: Class HUB in network_simulation.py

```
01 class HUB(Actor):
02   def __init__(self, name, in_
   channel):
03     Actor.__init__(self, name,
   in_channel)
04     self.connectors = []
05     self.messages = []
06
07   def action(self, msg):
08     # dispatch incoming packet
   to all connected devices
09     self.messages.append(msg)
10     while self.messages:
11       msg = self.messages.
   pop()
12       conn = self.connectors[:]
13       while conn:
14         out = conn.pop()
15         if out.balance < 0:
16           out.send(msg)
17         else:
18           conn.insert(0,out)
19           if self.in_channel.
   balance > 0:
20             self.messages.
   append(self.in_channel.
   receive())
21           stackless.schedule()
```

The tasklets also take a break when they have something to send. This makes the program loop for the hub slightly more complex (see Listing 5).

The hub will only send a message when a node or hub is listening at the other end of the line. The *out.balance < 0* request takes care of this. Additionally, the hub needs to pick up incoming pack-

Listing 6: squareroot.py

```
01 import stackless
02 import pickle
03
04 def squareroot(x):
05   # Newton method
06   print
07   print "Square root of ",x
08   print "-------------"
09   i = 0
10   y = x
11   print i, ":", y
12   while True:
13     i += 1
14     y = (y + x/y)/2.0
15     print i, ":", y
16     stackless.schedule()
17
18 if __name__ == '__main__':
19   import sys
20   x = float(sys.argv[5])
21   task = stackless.
   tasklet(squareroot)(x)
22   stackless.schedule()
23   stackless.schedule()
24   print 'pickle'
25   pickled_task = pickle.
   dumps(task)
26   task.remove()
27   print 'unpickle'
28   newtask = pickle.
   loads(pickled_task)
29   newtask.insert()
30   stackless.schedule()
31   stackless.schedule()
```

ets. If the *balance* attribute of a channel is positive, another network node is waiting to send something.

Pickled

squareroot.py in Listing 6 shows a less well known Stackless feature; tasklets can be stored in a binary format – this is referred to as *pickling* in Python speak.

The internal state of the tasklet is kept and can be restored at a later stage. The sample program uses Newton's method to calculate the square root of a number.

squareroot.py generates the output shown below with a parameter of 2:

```
$ stackless squareroot2
.py 2

Square root of 2.0
-------------
0 : 2.0
1 : 1.5
2 : 1.41666666667
pickle
unpickle
3 : 1.41421568627
4 : 1.41421356237
```

After storing the tasklet, you can run it in another process, on another machine, and possibly even on another architecture. ■

Table 1: Stackless Classes and Functions

task=stackless.tasklet (<function>) (<function arguments>)	Creates a tasklet *task*
stackless.schedule()	Switches to next tasklet
stackless.run()	Switches to next tasklet and removes itself from the scheduler list
channel=stackless.channel()	Creates a new channel object *channel*
channel.send(message)	Sends *message* to channel; the tasklet blocks until *message* has been picked up
message=channel.receive()	Receives *message* from channel; the tasklet blocks until *message* has arrived
channel.balance	< 0: Somebody is waiting to receive #new line > 0: somebody is waiting to send

INFO

[1] Stackless Python: *http://www.stackless.com*

[2] Erlang: *http://www.erlang.org*

[3] Oz: *http://www.mozart-oz.org*

[4] Stack: *http://en.wikipedia.org/wiki/Stack_(data_structure)*

[5] Heap: *http://en.wikipedia.org/wiki/Heap*

[6] Coroutine: *http://en.wikipedia.org/wiki/Coroutine*

[7] Eve Online: *http://www.eve-online.com*

[8] CCP: *http://www.ccpgames.com*

[9] Ironport: *http://www.ironport.com*

[10] Round-robin approach: *http://en.wikipedia.org/wiki/Round_robin*

[11] Listings: *http://www.linux-magazine.com/Magazine/Downloads/81*

[12] Pypy: *http://codespeak.net/pypy/dist/pypy/doc/news.html*