Introduction to Concurrent Programming with Stackless Python

Grant Olson

July 7, 2006

Email: olsongt@verizon.net

Contents

Chapter 1

Introduction

1.1 Why Stackless

According the stackless website at http://www.stackless.com/:

```
Stackless Python is an enhanced version of the Python programming language.
It allows programmers to reap the benefits of thread-based programming
without the performance and complexity problems associated with
conventional threads. The microthreads that Stackless adds to Python are a
cheap and lightweight convenience which can if used properly, give the
following benefits:

  + Improved program structure.

  + More readable code.

  + Increased programmer productivity.
```

Which acts as a very concise definition of stackless python, but what does that mean to us? It means that stackless provides the tools to model concurrency more easily than you can currently do in most conventional languages. We're not just talking about Python itself, but Java, C/C++ and other languages as well. Although there are some languages out there that provide concurrency features, they are either languages languages being primarily used in academia (such as Mozart/Oz) or lesser used/special purpose professional languages (such as erlang). With stackless, you get concurrency in addition to all of the advantages of python itself, in an environment that you are (hopefully) already familiar with.

This of course begs the question: Why concurrency?

1.1.1 The Real World is Concurrent

The real world is 'concurrent'. It is made up of a bunch of things (or **actors**) that interact with each other in a loosely coupled way with limited knowledge of each other. One of the purported benefits of object-oriented programming is that objects model things in the real world. To a certain extent, this is true. Object-oriented programming does a good job modelling individual objects, but it doesn't do a very good job representing interactions between these individual objects in a realistic way. For example, what is wrong with the following code example?

```
def familyTacoNight():
    husband.eat(dinner)
    wife.eat(dinner)
    son.eat(dinner)
    daughter.eat(dinner)
```

At first glance, nothing. But there is a subtle programs with the above example; the events occur serially. That is, the wife does not eat until after the husband is completed with his meal, the son doesn't eat until the wife is done, and the daughter is last in line. In the real world, the wife, son, and daughter will eat even if the husband gets caught in a traffic jam. In the above example they'll starve to death. Even worse no one will ever know because they'll never get the opportunity to throw an exception and notify the world!

1.1.2 Concurrency might, just might, be the next big programming paradigm

I personally believe that concurrency is the next big paradigm in the software world. As programs become more complex and resource intensive, we can no longer count on Moore's Law providing us with faster processors every year. Current performance increases in commodity PC's are coming from multi-core and multi-cpu machines. Once an individual CPU is maxed out for performance, software developers will have to move to distributed models where multiple computers interact with each other to create a high powered application (think think GooglePlex). To take advantage of both multi-core machines and distributed programming, concurrency quickly becomes the de-facto standard way of doing things.

1.2 Installing Stackless

Details for installing stackless can be found on the stackless website. Currently, linux users can get the source from subversion and perform a build. For windows users, there is an available .zip file that has to be extracted into an existing installation of python. The rest of this tutorial assumes that you have a working installation of stackless python, along with a basic understanding of the python language itself.

Chapter 2

Getting Started With Stackless

This chapter provides a brief introduction to stackless' primitives. Later chapters will build on these primitives to show more practical functionality.

2.1 Tasklets

Tasklets are the primary building block for stackless. You create a tasklet by feeding it any python callable (usually a function or class method). This creates a tasklet and adds it to the scheduler. Here is a quick demonstration:

```
Python 2.4.3 Stackless 3.1b3 060504 (#69, May  3 2006, 19:20:41) [MSC v.1310 32
bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import stackless
>>>
>>> def print_x(x):
...     print x
...
>>> stackless.tasklet(print_x)('one')
<stackless.tasklet object at 0x00A45870>
>>> stackless.tasklet(print_x)('two')
<stackless.tasklet object at 0x00A45A30>
>>> stackless.tasklet(print_x)('three')
<stackless.tasklet object at 0x00A45AB0>
>>>
>>> stackless.run()
one
two
three
>>>
```

Note that the tasklets queue up and don't run until you call *stackless.run()*.

2.2 The Scheduler

The scheduler controls the order in which the tasklets are run. If you just create a bunch of tasklets, they will run in the order they are created. In practice, you will generally create tasklets that reschedule themselves so that each one can have it's turn. A quick demonstration:

```
Python 2.4.3 Stackless 3.1b3 060504 (#69, May  3 2006, 19:20:41) [MSC v.1310 32
bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import stackless
>>>
>>> def print_three_times(x):
...     print "1:", x
...     stackless.schedule()
...     print "2:", x
...     stackless.schedule()
...     print "3:", x
...     stackless.schedule()
...
>>>
>>> stackless.tasklet(print_three_times)('first')
<stackless.tasklet object at 0x00A45870>
>>> stackless.tasklet(print_three_times)('second')
<stackless.tasklet object at 0x00A45A30>
>>> stackless.tasklet(print_three_times)('third')
<stackless.tasklet object at 0x00A45AB0>
>>>
>>> stackless.run()
1: first
1: second
1: third
2: first
2: second
2: third
3: first
3: second
3: third
>>>
```

Note that when we call *stackless.schedule()*, the active tasklet pauses itself and reinserts itself into the end of the scheduler's queue, allowing the next tasklet to run. Once all of the tasklets ahead of this run, it picks up where it left off. This continues until all active tasklets have finished running. This is how we achieve cooperative multitasking with stackless.

2.3 Channels

Channels allow you to send information between tasklets. This accomplished two things:

1. It allows you to exchange information between tasklets.
2. It allows you to control the flow of execution.

Another quick demonstration:

```
C:\>c:\python24\python
Python 2.4.3 Stackless 3.1b3 060504 (#69, May  3 2006, 19:20:41) [MSC v.1310 32
bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import stackless
>>>
>>> channel = stackless.channel()
```

```
>>>
>>> def receiving_tasklet():
...     print "Recieving tasklet started"
...     print channel.receive()
...     print "Receiving tasklet finished"
...
>>> def sending_tasklet():
...     print "Sending tasklet started"
...     channel.send("send from sending_tasklet")
...     print "sending tasklet finished"
...
>>> def another_tasklet():
...     print "Just another tasklet in the scheduler"
...
>>> stackless.tasklet(receiving_tasklet)()
<stackless.tasklet object at 0x00A45B30>
>>> stackless.tasklet(sending_tasklet)()
<stackless.tasklet object at 0x00A45B70>
>>> stackless.tasklet(another_tasklet)()
<stackless.tasklet object at 0x00A45BF0>
>>>
>>> stackless.run()
Recieving tasklet started
Sending tasklet started
send from sending_tasklet
Receiving tasklet finished
Just another tasklet in the scheduler
sending tasklet finished
>>>
>>>
```

When the receiving tasklet calls *channel.receive()*, it blocks. This means the tasklet pauses until something is sent over the channel. Nothing other than sending something over the channel will re-activate the tasklet.

When something is sent over the channel by the sending tasklet, the receiving tasklet resumes immediately, bypassing the existing schedule. The sending tasklet is send to the back of the schedule, as if *stackless.schedule()* had been called.

Also note that a send will block if there is nothing to immediately receive the message:

```
>>>
>>> stackless.tasklet(sending_tasklet)()
<stackless.tasklet object at 0x00A45B70>
>>> stackless.tasklet(another_tasklet)()
<stackless.tasklet object at 0x00A45BF0>
>>>
>>> stackless.run()
Sending tasklet started
Just another tasklet in the scheduler
>>>
>>> stackless.tasklet(another_tasklet)()
<stackless.tasklet object at 0x00A45B30>
>>> stackless.run()
Just another tasklet in the scheduler
>>>
>>> #Finally adding the receiving tasklet
...
>>> stackless.tasklet(receiving_tasklet)()
<stackless.tasklet object at 0x00A45BF0>
```

```
>>>
>>> stackless.run()
Recieving tasklet started
send from sending_tasklet
Receiving tasklet finished
sending tasklet finished
>>>
```

The sending tasklet will not be re-inserted into the scheduler until it has successfully send data to another tasklet.

2.4 Summary

That covers a majority of stackless' functionality. Doesn't seem like much does it? We dealt with a couple of objects and maybe 4 or 5 function calls. By using this small API as our building blocks, we can start to do some really interesting things.

Chapter 3

Coroutines

3.1 The Problem with to Subroutines

Most conventional programming languages expose the concept of subroutines. A subroutine is called by another routine (which is probably considered a subroutine by another routine) and may or may not return a result. By definition, a subroutine is subordinate to the caller.

Take the following example:

```
def ping():
    print "PING"
    pong()

def pong():
    print "PONG"
    ping()


ping()
```

An experienced programmer will see a problem with this program. It causes a stack overflow. If you run the program, it'll display a big nasty traceback indicating that you ran out of stack space.

3.1.1 The Stack

I debated how much detail I should get into about the C-stack, and decided to bypass the description entirely. It seemed like other attempts to describe it and diagrams only made sense to people who already understood it. I'll try to provide a simplistic explanation of it. Interested readers should look for more information on the web.

Every time you call a subroutine, it creates a **stack frame**. This is the holding area for variables and other information local to the subroutine. In this case, when you call *ping()* it creates a stack frame that holds information for the subroutine call. Simplistically, the frame says that says *ping pinged*. When it calls *pong()*, this creates another stack frame that says *pong ponged*. The stack frames are chained together, with each subroutine call acting as a link in the chain. In this case, the stack basically says *ping pinged so pong ponged*. Of course when *pong()* calls *ping()* this extends the stack. Here is a visual representation:

Frame	Stack
1	ping pinged
2	ping pinged so pong ponged
3	ping pinged so pong ponged so ping pinged
4	ping pinged so pong ponged so ping pinged so pong ponged
5	ping pinged so pong ponged so ping pinged so pong ponged so ping pinged
6	ping pinged so pong ponged so ping pinged so pong ponged so ping pinged ...

Now imagine that the page width represents the memory that your system has allocated to the stack. When you hit the edge of the page, you *overflow* and the system runs out of memory. Hence the term *stack overflow*.

3.1.2 So why do we use stacks?

The above example is intentionally designed to show you the problem with the stack. In most cases, each stack frame cleans itself up when a subroutine returns. In those cases, the stack will clean itself up. This is generally a good thing. In C, the stack is the one area where the programmer doesn't have to do manual memory management. Luckily, python programmers don't have to worry about memory management and the stack directly, but because the python interpreter itself is written in C, the implementors do have to worry about it. Using the stack makes things convenient. That is until we start calling functions that never return, like the example above. Then the stack actually starts working against the programmer, and exhausts available memory.

3.2 Enter Coroutines

In this case, it's a little silly that the stack overflows. *ping()* and *pong()* aren't really subroutines. One isn't subordinate to the other. They are *coroutines*, on equal footing, who should be able communicate seamlessly with each other.

Frame	Stack
1	ping pinged
2	pong ponged
3	ping pinged
4	pong ponged
5	ping pinged
6	pong ponged

With stackless, we create coroutines with channels. If you remember, one of the two benefits of channels is that they allow you to control flow between tasklets. By using channels, we can go back and forth between the ping and pong tasklets as much as we want without a stack overflow:

```
#
# pingpong_stackless.py
#

import stackless

ping_channel = stackless.channel()
pong_channel = stackless.channel()
```

```
def ping():
    while ping_channel.receive(): #blocks here
        print "PING"
        pong_channel.send("from ping")

def pong():
    while pong_channel.receive():
        print "PONG"
        ping_channel.send("from pong")



stackless.tasklet(ping)()
stackless.tasklet(pong)()

# we need to 'prime' the game by sending a start message
# if not, both tasklets will block
stackless.tasklet(ping_channel.send)('startup')

stackless.run()
```

You can run this program as long as you want and it won't crash. Also, if you examine memory usage (with the Task Manager on Windows or **top** on Unix), you'll see that memory usage is constant. The coroutine version of the program uses the same memory whether it runs for 1 minute or a day. If you examine memory usage of the recursive version, you'll see that it grows quickly before the program blows up.

3.3 Summary

If you recall, earlier I mentioned that experience programmers would immediately see a problem with the recursive version of the code. But honestly, there's not 'computer science' issue that would prevent this code from working. It is simply a minor implementation detail that you're stuck with in most conventional languages, because most conventional languages use a stack. In some sense, experienced programmers have simply been brainwashed to accept this as an acceptable problem. Stackless removes this perceived problem.

Chapter 4

Lightweight Threads

Threadlets, as the name implies, are lightweight compared to the threads built into today's OSes and supported by the standard python code. They use less memory than a 'traditional' thread, and switching between threadlets is much less resource intensive than switching between 'traditional' threads.

To demonstrate exactly how much more efficient threadlets are than traditional threads, we'll write the same program using both traditional threads and threadlets.

4.1 The hackysack simulation

Hackysack is a game in which a group of dirty hippies stand in a circle and kick a small bean filled bag around. The object is to keep the bag from hitting the ground, while passing it to other players and doing tricks. Players can only use their feet to kick the ball around.

In our simple simulation, we'll assume that the circle is a constant size once the game starts, and that the players are so good that the game will go on infinitely if allowed.

4.2 Tradional threaded version of the game

```
import thread
import random
import sys
import Queue

class hackysacker:
    counter = 0
    def __init__(self,name,circle):
        self.name = name
        self.circle = circle
        circle.append(self)
        self.messageQueue = Queue.Queue()

        thread.start_new_thread(self.messageLoop,())

    def incrementCounter(self):
        hackysacker.counter += 1
        if hackysacker.counter >= turns:
            while self.circle:
                hs = self.circle.pop()
```

```
                if hs is not self:
                    hs.messageQueue.put('exit')
            sys.exit()

    def messageLoop(self):
        while 1:
            message = self.messageQueue.get()
            if message == "exit":
                debugPrint("%s is going home" % self.name)
                sys.exit()
            debugPrint("%s got hackeysack from %s" % (self.name, message.name))
            kickTo = self.circle[random.randint(0,len(self.circle)-1)]
            debugPrint("%s kicking hackeysack to %s" % (self.name, kickTo.name))
            self.incrementCounter()
            kickTo.messageQueue.put(self)



def debugPrint(x):
    if debug:
        print x

debug=1
hackysackers=5
turns = 5

def runit(hs=10,ts=10,dbg=1):
    global hackysackers,turns,debug
    hackysackers = hs
    turns = ts
    debug = dbg

    hackysacker.counter= 0
    circle = []
    one = hackysacker('1',circle)

    for i in range(hackysackers):
        hackysacker(`i`,circle)

    one.messageQueue.put(one)

    try:
        while circle:
            pass
    except:
        #sometimes we get a phantom error on cleanup.
        pass


if __name__ == "__main__":
    runit(dbg=1)
```

A hackysacker class is initialized with it's name, a reference to a global list *circle* that contains all of the

players, and a message queue derived from the Queue class included in the python standard library.

The Queue class acts similarly to stackless channels. It contains *put()* and *get()* methods. Calling the *put()* method on an empty Queue blocks until another thread *put()s* something into the Queue. The Queue class is also designed to work efficiently with OS-level threads.

The *__init__* method then starts the messageLoop in a new thread using the thread module from the python standard library. The message loop starts an infinite loop that processes messages in the queue. If it receives the special message 'exit', it terminates the thread.

If it receives another message, that indicates that it is receiving the hackysack. It grabs another random player from the loop and 'kicks' the hackysack to it by sending it a message.

Once there have been enough kicks, as counted by the class variable *hackysacker.counter*, each player in the circle is sent the special 'exit' message.

Note that there is also a debugPrint function that prints output if the global variable debug is non-zero. We can print the game to stdout, but this will make the timing inaccurate when we measure it.

Let's run the program to verify that it works:

```
C:\Documents and Settings\grant\Desktop\why_stackless\code>c:\python24\python.exe
hackysackthreaded.py

1 got hackeysack from 1
1 kicking hackeysack to 4
4 got hackeysack from 1
4 kicking hackeysack to 0
0 got hackeysack from 4
0 kicking hackeysack to 1
1 got hackeysack from 0
1 kicking hackeysack to 3
3 got hackeysack from 1
3 kicking hackeysack to 3
4 is going home
2 is going home
1 is going home
0 is going home
1 is going home

C:\Documents and Settings\grant\Desktop\why_stackless\code>
```

And we see that all the hackysackers get together and play a quick game. Now let's time some trial runs. The python standard library include a program timeit.py that does this. In this case, we'll disable debug printing as well:

```
C:\Documents and Settings\grant\Desktop\why_stackless\code>c:\python24\python.ex
e c:\Python24\lib\timeit.py -s "import hackysackthreaded" hackysackthreaded.ru
nit(10,1000,0)
10 loops, best of 3: 183 msec per loop
```

Ten hackysackers going 1000 rounds takes 183 msecs on my computer. Let's increase the number of players:

```
C:\Documents and Settings\grant\Desktop\why_stackless\code>c:\python24\python.ex
e c:\Python24\lib\timeit.py -s "import hackeysackthreaded" hackeysackthreaded.ru
nit(100,1000,0)
10 loops, best of 3: 231 msec per loop

C:\Documents and Settings\grant\Desktop\why_stackless\code>c:\python24\python.ex
e c:\Python24\lib\timeit.py -s "import hackysackthreaded" hackysackthreaded.ru
nit(1000,1000,0)
10 loops, best of 3: 681 msec per loop
```

```
C:\Documents and Settings\grant\Desktop\why_stackless\code>c:\python24\python.ex
e c:\Python24\lib\timeit.py -s "import hackysackthreaded" hackysackthreaded.ru
nit(10000,1000,0)
Traceback (most recent call last):
  File "c:\Python24\lib\timeit.py", line 255, in main
    x = t.timeit(number)
  File "c:\Python24\lib\timeit.py", line 161, in timeit
    timing = self.inner(it, self.timer)
  File "<timeit-src>", line 6, in inner
  File ".\hackeysackthreaded.py", line 58, in runit
    hackysacker('i',circle)
  File ".\hackeysackthreaded.py", line 14, in __init__
    thread.start_new_thread(self.messageLoop,())
error: can't start new thread
```

And we get an error when trying to start 10,000 threads on my 3 Ghz machine with 1 Gig of ram. I don't want to bore you with the details of the output, but with some trial and error, the program starts failing at about 1100 threads on my machine. Also note that 1000 threads takes about three times as long as 10.

4.3 Stackless

```
import stackless
import random
import sys

class hackysacker:
    counter = 0
    def __init__(self,name,circle):
        self.name = name
        self.circle = circle
        circle.append(self)
        self.channel = stackless.channel()

        stackless.tasklet(self.messageLoop)()

    def incrementCounter(self):
        hackysacker.counter += 1
        if hackysacker.counter >= turns:
            while self.circle:
                self.circle.pop().channel.send('exit')

    def messageLoop(self):
        while 1:
            message = self.channel.receive()
            if message == 'exit':
                return
            debugPrint("%s got hackeysack from %s" % (self.name, message.name))
            kickTo = self.circle[random.randint(0,len(self.circle)-1)]
            while kickTo is self:
                kickTo = self.circle[random.randint(0,len(self.circle)-1)]
            debugPrint("%s kicking hackeysack to %s" % (self.name, kickTo.name))
            self.incrementCounter()
            kickTo.channel.send(self)
```

```
def debugPrint(x):
    if debug:print x

debug = 5
hackysackers = 5
turns = 1

def runit(hs=5,ts=5,dbg=1):
    global hackysackers,turns,debug
    hackysackers = hs
    turns = ts
    debug = dbg

    hackysacker.counter = 0
    circle = []
    one = hackysacker('1',circle)

    for i in range(hackysackers):
        hackysacker(`i`,circle)

    one.channel.send(one)

    try:
        stackless.run()
    except TaskletExit:
        pass


if __name__ == "__main__":
    runit()
```

This code is virtually identical to the threaded version. The primary differences are that we start tasklets instead of threads, and we switch between threads via channels instead of a Queue object. Let's run it and verify the output:

```
C:\Documents and Settings\grant\Desktop\why_stackless\code>c:\Python24\python.ex
e hackysackstackless.py
1 got hackeysack from 1
1 kicking hackeysack to 1
1 got hackeysack from 1
1 kicking hackeysack to 4
4 got hackeysack from 1
4 kicking hackeysack to 1
1 got hackeysack from 4
1 kicking hackeysack to 4
4 got hackeysack from 1
4 kicking hackeysack to 0
```

And it works as expected. Now for the timing:

```
C:\Documents and Settings\grant\Desktop\why_stackless\code>c:\Python24\python.ex
e c:\Python24\lib\timeit.py -s"import hackysackstackless" hackysackstackless.r
unit(10,1000,0)
100 loops, best of 3: 19.7 msec per loop
```

It only takes 19.7 msec. This is almost 10 times faster than the threaded version. Now lets start increasing the number of threadlets:

```
C:\Documents and Settings\grant\Desktop\why_stackless\code>c:\Python24\python.ex
e c:\Python24\lib\timeit.py -s"import hackysackstackless" hackysackstackless.r
unit(100,1000,0)
100 loops, best of 3: 19.7 msec per loop

C:\Documents and Settings\grant\Desktop\why_stackless\code>c:\Python24\python.ex
e c:\Python24\lib\timeit.py -s"import hackysackstackless" hackysackstackless.r
unit(1000,1000,0)
10 loops, best of 3: 26.9 msec per loop

C:\Documents and Settings\grant\Desktop\why_stackless\code>c:\Python24\python.ex
e c:\Python24\lib\timeit.py -s"import hackysackstackless" hackysackstackless.r
unit(10000,1000,0)
10 loops, best of 3: 109 msec per loop

C:\Documents and Settings\grant\Desktop\why_stackless\code>c:\Python24\python.ex
e c:\Python24\lib\timeit.py -s"import hackysackstackless" hackysackstackless.r
unit(100000,1000,0)
10 loops, best of 3: 1.07 sec per loop
```

Even by the time we get to 10,000 threads, which the threaded version wasn't even capable of running, we're still running faster than the threaded version did with only 10 threads.

Now I'm trying to keep the code simple, so you'll have to take my word for it, but the increase in timings here is due to the time it takes to setup the hackysack circle. The amount of time to run the game is constant whether you have 10 threadlets or 100000 threadlets. This is because of the way channels work: they block and instantly resume when they get a message. On the other hand, OS threads each take turns checking to see if their message Queue has any elements. This means that the more threads you run, the worse performance gets.

4.4 Summary

Hopefully, I've successfully demonstrated that threadlets run at least an order of magnitude faster than OS threads, and scale much better. The general wisdom with OS threads is (1) don't use them, and (2) if you have to use as little as possible. Stackless' threadlets free us from these constraints.

Chapter 5

Dataflow

5.1 The Factory

Let's pretend we want to write a simulation of a doll factory that has the following requirements:

- One storeroom that contains plastic pellets that will be molded.
- One storeroom that contains rivets used to attach parts.
- An injection molder that takes 0.2 pounds of plastic pellets and creates a pair of arms in 6 seconds.
- An injection molder that takes 0.2 pounds of plastic pellets and creates a pair of legs in 5 seconds.
- An injection molder that takes 0.1 pounds of plastic pellets and creates a head in 4 seconds.
- An injection molder that takes 0.5 pounds of plastic pellets and creates a torso in 10 seconds.
- An assembly station that takes a completed torso, a completed pair of legs, a rivet, and puts them together in 2 seconds.
- An assembly station that takes the combined part from above, a pair of arms, and a rivet and puts them together in 2 seconds.
- An assembly station that takes the completed part from above, a head, and a rivet and puts them together in 3 seconds.
- Each station keeps on doing this forever.

5.2 The 'normal' version

I'll take a stab at writing this 'normally' without using stackless. After we go through the 'normal' example, we'll build one with stackless and compare the resulting code. If you think the example is contrived, and you have the time, you may want to take a break, write a factory with the above requirements yourself, and compare your resulting code to the stackless version.

Here is the code:

```
class storeroom:
    def __init__(self,name,product,unit,count):
```

```
        self.product = product
        self.unit = unit
        self.count = count
        self.name = name

    def get(self,count):
        if count > self.count:
            raise RuntimeError("Not enough %s" % self.product)
        else:
            self.count -= count

        return count

    def put(self,count):
        self.count += count

    def run(self):
        pass

rivetStoreroom = storeroom("rivetStoreroom","rivets","#",1000)
plasticStoreroom = storeroom("plastic Storeroom","plastic pellets","lb",100)

class injectionMolder:
    def __init__(self,name,partName,plasticSource,plasticPerPart,timeToMold):
        self.partName = partName
        self.plasticSource = plasticSource
        self.plasticPerPart = plasticPerPart
        self.timeToMold = timeToMold
        self.items = 0
        self.plastic = 0
        self.time = -1
        self.name = name

    def get(self,items):
        if items > self.items:
            return 0
        else:
            self.items -= items
            return items

    def run(self):
        if self.time == 0:
            self.items += 1
            print "%s finished making part" % self.name
            self.time -= 1
        elif self.time < 0:
            print "%s starts making new part %s" % (self.name,self.partName)
            if self.plastic < self.plasticPerPart:
                print "%s getting more plastic"
                self.plastic += self.plasticSource.get(self.plasticPerPart * 10)
            self.time = self.timeToMold
        else:
            print "%s molding for %s more seconds" % (self.partName, self.time)
            self.time -= 1


armMolder = injectionMolder("arm Molder", "arms",plasticStoreroom,0.2,6)
```

```
legMolder = injectionMolder("leg Molder", "leg",plasticStoreroom,0.2,5)
headMolder = injectionMolder("head Molder","head",plasticStoreroom,0.1,4)
torsoMolder = injectionMolder("torso Molder","torso",plasticStoreroom,0.5,10)


class assembler:
    def __init__(self,name,partAsource,partBsource,rivetSource,timeToAssemble):
        self.partAsource = partAsource
        self.partBsource = partBsource
        self.rivetSource = rivetSource
        self.timeToAssemble = timeToAssemble
        self.itemA = 0
        self.itemB = 0
        self.items = 0
        self.rivets = 0
        self.time = -1
        self.name = name

    def get(self,items):
        if items > self.items:
            return 0
        else:
            self.items -= items
            return items

    def run(self):
        if self.time == 0:
            self.items += 1
            print "%s finished assembling part" % self.name
            self.time -= 1
        elif self.time < 0:
            print "%s starts assembling new part" % self.name
            if self.itemA < 1:
                print "%s Getting item A" % self.name
                self.itemA += self.partAsource.get(1)
                if self.itemA < 1:
                    print "%s waiting for item A" % self.name
            elif self.itemB < 1:
                print "%s Getting item B" % self.name
                self.itemB += self.partBsource.get(1)
                if self.itemB < 1:
                    print "%s waiting for item B" % self.name
            print "%s starting to assemble" % self.name
            self.time = self.timeToAssemble
        else:
            print "%s assembling for %s more seconds" % (self.name, self.time)
            self.time -= 1

legAssembler = assembler("leg Assembler",torsoMolder,legMolder,rivetStoreroom,2)
armAssembler = assembler("arm Assembler", armMolder,legAssembler,rivetStoreroom,2)
torsoAssembler = assembler("torso Assembler", headMolder,armAssembler,
                           rivetStoreroom,3)

components = [rivetStoreroom, plasticStoreroom, armMolder,
                legMolder, headMolder, torsoMolder,
              legAssembler, armAssembler, torsoAssembler]
```

```
def run():
    while 1:
        for component in components:
            component.run()
        raw_input("Press <ENTER> to continue...")
        print "\n\n\n"


if __name__ == "__main__":
    run()
```

5.2.1 Analysis

We start off with a class that represents the storerooms. It is initialized with a product name, a unit measurement (such as pounds or number of parts), and an initial quantity. There is also a run method that does nothing; its usage will become clear a little later. We create two storeroom instances based on this class.

Next up is an injectionMolder class. It is initialized with the name of a finished part, a storeroom that acts a plastic source, the quantity required to build one part, and the time required to create the part. There is a get() method that can be used to get retrieve a finished part and adjust inventory if one exists. For this class the run() method actually does something:

- While the timer is above 0, it continues molding and decrements the counter.
- When the time-to-mold reaches 0, a finished part is created and the time counter is set to -1.
- When the time counter is -1, the molder checks to see if it has enough plastic to mold another part, retrieves it if necessary, and begins molding.

We create four injection molder instances with this class.

Next up is an assembler class. It is initialized with the name of the finished part, the source for part 1, the source for part 2, a storeroom that contains rivets, and the time required to assemble the parts in question. There is a get() method that can be used to get retrieve a finished part and adjust inventory if one exists. For this class the run() method:

- If the timer is greater than one, the assembler has the required parts and continues to assemble.
- If the timer equals 0, a part is completed and inventory is adjusted.
- If the timer is less than one, the assembler tries to grab one of each part required to start building again. It may have to wait here if a part hasn't finished molding yet.

Assembler instances are created to attach the leg, arm, and head.

> **Note**
>
> You'll notice a lot of similarities between the storeroom, injectionMolder, and assember classes. If I was writing a production system, I would probably create a base class and use inheritance. In this case, I thought setting up a class heirarchy would confuse the code, so I kept it intentionally simple.

All instances from all three classes are loaded into an array called compenents. We then create an event loop that repeatedly calls the run() method for each component.

5.3 Enter Dataflow

If you're familiar with unix systems, you've probably used dataflow techniques whether you knew it or not. Take the following shell command:

```
cat README | more
```

In fairness, the Windows equivilent is:

```
type readme.txt | more
```

Although dataflow techniques aren't as pervasive in the Windows world as they are in the Unix world.

For those who are unfamilar with the *more* program, it recieves input from an external source, displays a page's worth of output, pauses until the user hits a key, and shows another page. The '|' operator takes the output from one program, and pipes it into the input of another command. In this case, either *cat* or *type* sends the text of a document to standard output, and *more* recieves it.

The *more* program just sits there until data flows into it from another program. Once enough data flows in, it displays a page on the screen and pauses. The user hits a key, and more lets more data flow in. Once again, more waits until enough data flows in, displays it, and pauses. Hence the term *dataflow*.

Using the stackless round-robin scheduler in addition to channels, we can use dataflow techniques to write the factory simulation.

5.4 The Stackless Version of the Code

```
import stackless


#
# 'Sleep helper functions
#

sleepingTasklets = []
sleepingTicks = 0

def Sleep(secondsToWait):
    channel = stackless.channel()
    endTime = sleepingTicks + secondsToWait
    sleepingTasklets.append((endTime, channel))
    sleepingTasklets.sort()
    # Block until we get sent an awakening notification.
    channel.receive()

def ManageSleepingTasklets():
    global sleepingTicks
    while 1:
        if len(sleepingTasklets):
            endTime = sleepingTasklets[0][0]
            while endTime <= sleepingTicks:
                channel = sleepingTasklets[0][1]
                del sleepingTasklets[0]
                # We have to send something, but it doesn't matter
                # what as it is not used.
                channel.send(None)
                endTime = sleepingTasklets[0][0] # check next
```

```
        sleepingTicks += 1
        print "1 second passed"
        stackless.schedule()

stackless.tasklet(ManageSleepingTasklets)()

#
# Factory implementation
#

class storeroom:
    def __init__(self,name,product,unit,count):
        self.product = product
        self.unit = unit
        self.count = count
        self.name = name

    def get(self,count):
        while count > self.count: #reschedule until we have enough
            print "%s doesn't have enough %s to deliver yet" % (self.name,
                                                                self.product)
            stackless.schedule()
        self.count -= count
        return count

        return count

    def put(self,count):
        self.count += count

    def run(self):
        pass

rivetStoreroom = storeroom("rivetStoreroom","rivets","#",1000)
plasticStoreroom = storeroom("plastic Storeroom","plastic pellets","lb",100)

class injectionMolder:
    def __init__(self,name,partName,plasticSource,plasticPerPart,timeToMold):
        self.partName = partName
        self.plasticSource = plasticSource
        self.plasticPerPart = plasticPerPart
        self.timeToMold = timeToMold
        self.plastic = 0
        self.items = 0
        self.name = name
        stackless.tasklet(self.run)()

    def get(self,items):
        while items > self.items: #reschedule until we have enough
            print "%s doesn't have enough %s to deliver yet" % (self.name,
                                                                self.partName)
            stackless.schedule()
        self.items -= items
        return items

    def run(self):
        while 1:
```

```
            print "%s starts making new part %s" % (self.name,self.partName)
            if self.plastic < self.plasticPerPart:
                print "%s getting more plastic"
                self.plastic += self.plasticSource.get(self.plasticPerPart * 10)
            self.plastic -= self.plasticPerPart
            Sleep(self.timeToMold)
            print "%s done molding after %s seconds" % (self.partName,
                                                        self.timeToMold)
            self.items += 1
            print "%s finished making part" % self.name
            stackless.schedule()


armMolder = injectionMolder("arm Molder", "arms",plasticStoreroom,0.2,5)
legMolder = injectionMolder("leg Molder", "leg",plasticStoreroom,0.2,5)
headMolder = injectionMolder("head Molder","head",plasticStoreroom,0.1,5)
torsoMolder = injectionMolder("torso Molder","torso",plasticStoreroom,0.5,10)


class assembler:
    def __init__(self,name,partAsource,partBsource,rivetSource,timeToAssemble):
        self.partAsource = partAsource
        self.partBsource = partBsource
        self.rivetSource = rivetSource
        self.timeToAssemble = timeToAssemble
        self.itemA = 0
        self.itemB = 0
        self.items = 0
        self.rivets = 0
        self.name = name
        stackless.tasklet(self.run)()

    def get(self,items):
        while items > self.items: #reschedule until we have enough
            print "Don't have a %s to deliver yet" % (self.name)
            stackless.schedule()
        self.items -= items
        return items

    def run(self):
        while 1:
            print "%s starts assembling new part" % self.name
            self.itemA += self.partAsource.get(1)
            self.itemB += self.partBsource.get(1)
            print "%s starting to assemble" % self.name
            Sleep(self.timeToAssemble)
            print "%s done assembling after %s" % (self.name, self.timeToAssemble)
            self.items += 1
            print "%s finished assembling part" % self.name
            stackless.schedule()

legAssembler = assembler("leg Assembler",torsoMolder,legMolder,rivetStoreroom,2)
armAssembler = assembler("arm Assembler", armMolder,legAssembler,rivetStoreroom,2)
torsoAssembler = assembler("torso Assembler", headMolder,armAssembler,
                           rivetStoreroom,3)

def pause():
```

```
    while 1:
        raw_input("Press <ENTER> to continue...")
        print "\n\n\n"
        stackless.schedule()

stackless.tasklet(pause)()

def run():
    stackless.run()

if __name__ == "__main__":
    run()
```

5.4.1 Analysis

Sleeper utilities

First, we create some helper functions that allow our classes to 'sleep'. When a tasklet calls Sleep(), it creates a channel, calculates a time to wake up, and attaches this information to the global sleepingTasklets array. After that, channel.receive() is called. This causes the calling tasklet to pause until reawakened.

Then we create a function to manage the sleeping tasklets. It checks the global sleepingTasklets array, finds any items that are ready to wake up, and reactivates them via the channel. This function is added to the tasklet scheduler.

The classes

These classes are similar to the original classes with a few notable exceptions. The first is that they spawn their *run()* methods at instantiation. We no longer need to manually create a components array and an external *run()* function to process the event loop; stackless handles this implicitly. The second difference is that tasklets will sleep while waiting for a part to be built instead of explictly maintaining a counter to track time. The third difference is that calls to *get()* are more natural. If a material or part is not available, the tasklet will simply reschedule itself until it is available.

5.5 So what have we gained?

Okay, so what's the big deal? Both programs run and basically generate the same results. Lets examine the run method from the original version of the factory:

```
def run(self):
    if self.time == 0:
        self.items += 1
        print "%s finished assembling part" % self.name
        self.time -= 1
    elif self.time < 0:
        print "%s starts assembling new part" % self.name
        if self.itemA < 1:
            print "%s Getting item A" % self.name
```

```
            self.itemA += self.partAsource.get(1)
            if self.itemA < 1:
                print "%s waiting for item A" % self.name
        elif self.itemB < 1:
            print "%s Getting item B" % self.name
            self.itemB += self.partBsource.get(1)
            if self.itemB < 1:
                print "%s waiting for item B" % self.name
        print "%s starting to assemble" % self.name
        self.time = self.timeToAssemble
    else:
        print "%s assembling for %s more seconds" % (self.name, self.time)
        self.time -= 1
```

And then the stackless version:

```
def run(self):
    while 1:
        print "%s starts assembling new part" % self.name
        self.itemA += self.partAsource.get(1)
        self.itemB += self.partBsource.get(1)
        print "%s starting to assemble" % self.name
        Sleep(self.timeToAssemble)
        print "%s done assembling after %s" % (self.name, self.timeToAssemble)
        self.items += 1
        print "%s finished assembling part" % self.name
        stackless.schedule()
```

The stackless version is much simpler, clearer, and intuitive than the original version. It doesn't wrap the event loop infrastructure into the run method. This has been decoupled from the run() method. The run() method only expresses what it is doing, and doesn't worry about the details of how it gets done. It lets the software developer concentrate on how the factory works, and not how the event loop and program works.

5.6 Pushing Data

> **Note**
>
> The completed program from this section is listed as digitalCircuit.py in both the code .zip archive and at the end of this document.

In the case of the factory, we were 'pulling' data. Each component asked for the parts it needed and waited until they arrived. We could also 'push' information so that one component in a system propigates its changes forward to another component. The 'pull' approach is called **lazy data flow** and the 'push' approach is called **eager data flow**.

To demonstrate the push approach, we'll build a digital circuit simulator. The simulator will consist of various parts that that have a state of 0 or 1, and can be interconnected in various ways. In this case, we will take an object oriented approach and define an EventHandler base class that implements most of the functionality:

```
class EventHandler:
    def __init__(self,*outputs):
        if outputs==None:
            self.outputs=[]
        else:
            self.outputs=list(outputs)
```

```
        self.channel = stackless.channel()
        stackless.tasklet(self.listen)()

    def listen(self):
        while 1:
            val = self.channel.receive()
            self.processMessage(val)
            for output in self.outputs:
                self.notify(output)

    def processMessage(self,val):
        pass

    def notify(self,output):
        pass

    def registerOutput(self,output):
        self.outputs.append(output)

    def __call__(self,val):
        self.channel.send(val)
```

The EventHandler class core functionality performs three things:

- It continously listens for messages on a channel on the listen method.
- Then it processes any message it recieves via the processMessage method.
- Then it notifies any registered outputs of the results via the nofify method.

In addition there are two helper methods:

- registerOutput allows you to register additional outputs after instantiation.
- __call__ is overridden as a convienence so that we can send messages in the form of:

  ```
  event(1)
  ```

  instead of:

  ```
  event.channel.send(1)
  ```

Using the EventHandler class as a building block, we can start to implement our digital circuit simulator, starting with a Switch. This is a representation of a user-togglible switch that can be send to either a 0 or 1 value:

```
class Switch(EventHandler):
    def __init__(self,initialState=0,*outputs):
        EventHandler.__init__(self,*outputs)
        self.state = initialState

    def processMessage(self,val):
        debugPrint("Setting input to %s" % val)
        self.state = val

    def notify(self,output):
        output((self,self.state))
```

When initialized, the switch stores its original state. processMessage is overridden to store the received message as the current state. The notify method is overridden to send a tuple containing a reference to the instance itself and the state. As we will see a little further on, we need to send the instance so that components with multiple inputs can tell what the message source was.

> **Note**
>
> If you're typing in the code as you go along, please note that we're using the debugPrint() function originally defined in the **Lightweight Threads** section to provide diagnostics.

The next class we'll create is the Reporter() class. Instances of this class simply display their current state. I suppose we could imagine that these were LED's in a real digital circuit:

```
class Reporter(EventHandler):
    def __init__(self,msg="%(sender)s send message %(value)s"):
        EventHandler.__init__(self)
        self.msg = msg

    def processMessage(self,msg):
        sender,value=msg
        print self.msg % {'sender':sender,'value':value}
```

The initializer accepts an optional format string for subsequent output. Everything else should be self explanitory.

Now we have a good enough framework to test the initial functionality:

```
C:\Documents and Settings\grant\Desktop\why_stackless\code>c:\Python24\python.ex
e
Python 2.4.3 Stackless 3.1b3 060516 (#69, May  3 2006, 11:46:11) [MSC v.1310 32
bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import stackless
>>> from digitalCircuit import *
>>>
>>> reporter = Reporter()
>>> switch = Switch(0,reporter) #create switch and attach reporter as output.
>>>
>>> switch(1)
<digitalCircuit.Switch instance at 0x00A46828> send message 1
>>>
>>> switch(0)
<digitalCircuit.Switch instance at 0x00A46828> send message 0
>>>
>>> switch(1)
<digitalCircuit.Switch instance at 0x00A46828> send message 1
>>>
```

Unlike the factory we created earlier, toggling a switch instantly pushes the results to it's output and the results are displayed.

Now lets create some digital logic components. The first is an inverter. It takes an input and pushes the logical inverse. Inputs of 0 or 1 will push 1 or 0 respectively:

```
class Inverter(EventHandler):
    def __init__(self,input,*outputs):
        EventHandler.__init__(self,*outputs)
```

```
        self.input = input
        input.registerOutput(self)
        self.state = 0

    def processMessage(self,msg):
        sender,value = msg
        debugPrint("Inverter received %s from %s" % (value,msg))
        if value:
            self.state = 0
        else:
            self.state = 1
```

The initializer for the inverter accepts an input, which is some sort of EventHandler, stores it, and registers itself as an output. processMessage() sets the state to the logical opposite of the message recieved. Like the Switch class, the notify event sends a tuple containing itself and its state.

We could chain this between the Switch and Reporter from the example above. Feel free to try it if you like, but I don't think it's necessary to show the interactive session here.

Next up is an AndGate. This is the first class we have that accepts multiple inputs. It has two inputs. If both are set to 1, it sends the message 1, otherwise it sends the message 0:

```
class AndGate(EventHandler):
    def __init__(self,inputA,inputB,*outputs):
        EventHandler.__init__(self,*outputs)

        self.inputA = inputA
        self.inputAstate = inputA.state
        inputA.registerOutput(self)

        self.inputB = inputB
        self.inputBstate = inputB.state
        inputB.registerOutput(self)

        self.state = 0

    def processMessage(self,msg):
        sender, value = msg
        debugPrint("AndGate received %s from %s" % (value,sender))

        if sender is self.inputA:
            self.inputAstate = value
        elif sender is self.inputB:
            self.inputBstate = value
        else:
            raise RuntimeError("Didn't expect message from %s" % sender)

        if self.inputAstate and self.inputBstate:
            self.state = 1
        else:
            self.state = 0
        debugPrint("AndGate's new state => %s" % self.state)

    def notify(self,output):
        output((self,self.state))
```

In the AndGate's process message, we need to determine what input sent the message and assign the state accordingly. This is why we needed to send the self object from the other components.

Lastly, we have the OrGate. It is identical to the AndGate, with the execption that it sends the message 1 if ether input is 1, and 0 only if both inputs are 0:

```
class OrGate(EventHandler):
    def __init__(self,inputA,inputB,*outputs):
        EventHandler.__init__(self,*outputs)

        self.inputA = inputA
        self.inputAstate = inputA.state
        inputA.registerOutput(self)

        self.inputB = inputB
        self.inputBstate = inputB.state
        inputB.registerOutput(self)

        self.state = 0

    def processMessage(self,msg):
        sender, value = msg
        debugPrint("OrGate received %s from %s" % (value,sender))

        if sender is self.inputA:
            self.inputAstate = value
        elif sender is self.inputB:
            self.inputBstate = value
        else:
            raise RuntimeError("Didn't expect message from %s" % sender)

        if self.inputAstate or self.inputBstate:
            self.state = 1
        else:
            self.state = 0
        debugPrint("OrGate's new state => %s" % self.state)

    def notify(self,output):
        output((self,self.state))
```

5.6.1 Half Adder

To finish things up, we'll use the components we have created to build a half-adder. A half-adder performs addition on two bits. [TODO: Make half adder diagram] We chain together several of our components and flip the switches. Flipping the switches changes their state, and propigates these changes through the system via dataflow:

```
if __name__ == "__main__":
    # half adder
    inputA = Switch()
    inputB = Switch()
    result = Reporter("Result = %(value)s")
    carry = Reporter("Carry = %(value)s")
    andGateA = AndGate(inputA,inputB,carry)
```

```
    orGate = OrGate(inputA,inputB)
    inverter = Inverter(andGateA)
    andGateB = AndGate(orGate,inverter,result)
    inputA(1)
    inputB(1)
    inputB(0)
    inputA(0)
```

Chapter 6

Actors

In the actor model, everything is an actor (duh!). Actors are objects (in the generic sense, not necessarily the OO sense) that can:

- Receive messages from other actors.
- Process the received messages as they see fit.
- Send messages to other actors.
- Create new Actors.

Actors do not have any direct access to other actors. All communication is accomplished via message passing. This provides a rich model to simulate real-world objects that are loosely-coupled and have limited knowledge of each others internals.

If we're going to create a simulation, we might as well simulate...

6.1 Killer Robots!

> **Note**
>
> The completed program from this section is listed as actors.py in both the code .zip archive and at the end of this document.

6.1.1 Actor Base Class

In this example, we'll configure a small world where robots can move around and fight utilizing the actor model. To begin with, let's define a base class for all of our actors:

```
class actor:
    def __init__(self):
        self.channel = stackless.channel()
        self.processMessageMethod = self.defaultMessageAction
        stackless.tasklet(self.processMessage)()

    def processMessage(self):
        while 1:
            self.processMessageMethod(self.channel.receive())

    def defaultMessageAction(self,args):
```

```
        print args
```

By default, an actor creates a channel to accept messages, assigns a method to process the messages, and kicks off a loop to dispatch accepted messages to the processing method. The default processor simply prints the message that it received. That's really all we need to implement the actor model.

6.1.2 Message Format

All messages will be sent in in the format of sender's channel, followed by a string containing the message name, followed by optional parameters. Examples are:

```
(self.channel, "JOIN", (1,1) )
(self.channel, "COLLISION")
etc...
```

Note that we're sending the sender's channel only instead of the entire self object. In the actor model, all communication between actors needs to occur through message passing. If we send *self*, then it would be too easy to cheat and access unknown information about the actor that sent a message.

In fact, you'll note that when we instantiate most of the actors in this chapter, we don't even assign them to a variable that is accessible to other actors. We just create them and let them float around on their own with limited knowledge of the environment.

6.1.3 World class

The world actor acts as the central hub through which all of the actors interact. Other actors send a JOIN message to the world actor, and it tracks them. Periodically, it sends out a WORLD_STATE message which contains information about all visible actors for their own internal processing:

```
class world(actor):
    def __init__(self):
        actor.__init__(self)
        self.registeredActors = {}
        stackless.tasklet(self.sendStateToActors)()

    def testForCollision(self,x,y):
        if x < 0 or x > 496:
            return 1
        elif y < 0 or y > 496:
            return 1
        else:
            return 0

    def sendStateToActors(self):
        while 1:
            for actor in self.registeredActors.keys():
                actorInfo = self.registeredActors[actor]
                if self.registeredActors[actor][1] != (-1,-1):
                    VectorX,VectorY = (math.sin(math.radians(actorInfo[2])) * actorInfo[3],
                                       math.cos(math.radians(actorInfo[2])) * actorInfo[3])
                    x,y = actorInfo[1]
                    x += VectorX
                    y -= VectorY
```

```
                    if self.testForCollision(x,y):
                        actor.send((self.channel,"COLLISION"))
                    else:
                        self.registeredActors[actor] = tuple([actorInfo[0],
                                                              (x,y),
                                                                  actorInfo[2],
                                                                  actorInfo[3]])
            worldState = [self.channel, "WORLD_STATE"]
            for actor in self.registeredActors.keys():
                if self.registeredActors[actor][1] != (-1,-1):
                    worldState.append( (actor, self.registeredActors[actor]))
            message = tuple(worldState)
            for actor in self.registeredActors.keys():
                actor.send(message)
            stackless.schedule()

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "JOIN":
            print 'ADDING ' , msgArgs
            self.registeredActors[sentFrom] = msgArgs
        elif msg == "UPDATE_VECTOR":
            self.registeredActors[sentFrom] = tuple([self.registeredActors[sentFrom][0],
                                                     self.registeredActors[sentFrom][1],
                                                     msgArgs[0],msgArgs[1]])
        else:
            print '!!!! WORLD GOT UNKNOWN MESSAGE ' , args

World = world().channel
```

In addition to the message processing tasklet, the world actor spawns a separate tasklet that runs the sendStateToActors() method. This method has a loop which builds the information about the state of the world, and sends it to all actors. This is the only message that the actors can rely on receiving. If necessary, they will respond to this message by sending some sort of UPDATE message back to the world.

As part of the sendStateToActors() method, the world actor needs to update its internal record of the location of moving actors. It creates a vector based on the angle and velocity of a moving actor, makes sure that the updated position doesn't collide with one of the walls of the world, and saves the new location.

The defaultMessageAction() method processes the following known messages and ignores the rest:

JOIN Add an actor to the list of known actors in the world. Parameters include location, angle, and velocity of the actor. A location of -1,-1 indicates that the actor is not visible to other actors, such as the display actor detailed below.

UPDATE_VECTOR Set a new angle and velocity for the actor that send the message.

Lastly, a world actor is instantiated and its channel is saved in the global variable World so that other actors can send their initial JOIN messages.

6.1.4 A Simple Robot

We'll start off with a simple robot that moves at a constant velocity, rotating clockwise one degree as a response to each WORLD_STATE message. In the event of a COLLISION with the world's walls, it will turn 73 degrees and attempt to continue moving forward. Any other message is ignored.

```
class basicRobot(actor):
    def __init__(self,location=(0,0),angle=135,velocity=1,world=World):
        actor.__init__(self)
        self.location = location
        self.angle = angle
        self.velocity = velocity
        self.world = world

        joinMsg =(self.channel,"JOIN",self.__class__.__name__,
                  self.location,self.angle,self.velocity)
        self.world.send(joinMsg)

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            self.location = (self.location[0] + 1, self.location[1] + 1)
            self.angle += 1
            if self.angle >= 360:
                self.angle -= 360

            updateMsg = (self.channel, "UPDATE_VECTOR",
                         self.angle,self.velocity)
            self.world.send(updateMsg)
        elif msg == "COLLISION":
            self.angle += 73
            if self.angle >= 360:
                self.angle -= 360
        else:
            print "UNKNOWN MESSAGE", args

basicRobot(angle=135,velocity=5)
basicRobot((464,0),angle=225,velocity=10)


stackless.run()
```

Note that the constructor for the robot issues a join message to the world object to register it. Other than that, hopefully the code is straightforward.

6.1.5 Detour: pyGame

So far we've been using debug print statements to illustrate the way things are working in our sample programs. I tried to do this to keep the code simple and understandable, but at some point print statements become more confusing than illustrative. We were already pushing it in the section on Dataflow, but the code in this section is getting too complex to even try to represent it with printed output.

> **Note**
>
> You will need to install a current copy of pyGame for the code samples in this section to work. It is available at http://www.pygame.org/

I decided to use pyGame to create a simple visualization engine. Although descriptions of pyGame internals are outside the scope of this tutorial, operation is relatively straight-forward. When the display actor receives a WORLD_STATE message, it places the appropriate actors and updates the display. Luckily, we are able to isolate

all of the pygame code into a single actor, so the rest of the code should remain 'unpolluted' and understandable without knowing or caring how pygame renders the display:

```
class display(actor):
    def __init__(self,world=World):
        actor.__init__(self)

        self.world = World

        self.icons = {}
        pygame.init()

        window = pygame.display.set_mode((496,496))
        pygame.display.set_caption("Actor Demo")

        joinMsg = (self.channel,"JOIN",self.__class__.__name__, (-1,-1))
        self.world.send(joinMsg)

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            self.updateDisplay(msgArgs)
        else:
            print "UNKNOWN MESSAGE", args

    def getIcon(self, iconName):
        if self.icons.has_key(iconName):
            return self.icons[iconName]
        else:
            iconFile = os.path.join("data","%s.bmp" % iconName)
            surface = pygame.image.load(iconFile)
            surface.set_colorkey((0xf3,0x0a,0x0a))
            self.icons[iconName] = surface
            return surface

    def updateDisplay(self,actors):

        for event in pygame.event.get():
            if event.type == pygame.QUIT: sys.exit()

        screen = pygame.display.get_surface()

        background = pygame.Surface(screen.get_size())
        background = background.convert()
        background.fill((200, 200, 200))

        screen.blit(background, (0,0))
        for item in actors:
            screen.blit(pygame.transform.rotate(self.getIcon(item[1][0]),-item[1][2]), item[1][1])
        pygame.display.flip()

display()
```

This takes the WORLD_STATE and creates a display based on that.

> **Note**
>
> You will need to install pyGame in your python installation for the examples in this section to work. You will also want to download the optional icons that I've created and unzip the directory under your source directory.

6.1.6 Round 1 of the code

Now we have enough to run the first version of the program. Upon execution, two of the basicRobots will zoom around and bounce off of the walls.

6.2 More Detours: Simulation Mechanics

> **Note**
>
> The completed program from this section is listed as actors2.py in both the code .zip archive and at the end of this document.

As another detour, we need to implement some game (er... I mean simulation) mechanics. Strictly speaking, these mechanics don't have anything to do with the actor model. However, to create a rich and realistic simulation we need to get these mechanics out of the way. This section will detail what we are trying to accomplish and how we will accomplish it. After that, our environment to toy around with actors will be much more usable.

6.2.1 Actor Properties

As the information that the world actor needs to track becomes more complex, sending a bunch of individual arguments in the initial JOIN message becomes cumbersome. To make this easier, we'll create a properties object to track the info. This will be sent with the JOIN message instead of individual parameters.

```
class properties:
    def __init__(self,name,location=(-1,-1),angle=0,
                 velocity=0,height=-1,width=-1,hitpoints=1,physical=True,
                 public=True):
        self.name = name
        self.location = location
        self.angle = angle
        self.velocity = velocity
        self.height = height
        self.width = width
        self.public = public
        self.hitpoints = hitpoints
```

Note that the properties object is created to transfer information between actors. We will not store a local copy with the actor that creates it. If we did, the world actor would be able to modify the internal workings of actors instead of properly modifying them by sending messages.

6.2.2 Collision Detection

There are a few problems with the collision detection routine in the last version of the program. The most obvious is that actors do not collide with each other. The two robots bouncing around will just drive through each other instead of colliding. The second problem is that we don't account for the size of the actor. This is most obvious when the robots hit the right or bottom walls. They appear to go halfway into the edge of the world before a COLLISION is registered. I'm sure there are whole books on collision detection out there, but we'll try to stick with a reasonably simple version that works well enough for our purposes.

First, we'll add height and width properties to each actor. This allows us to create a 'bounding-box' around the actor. The location property contains the top-left corner of the box, and adding the height and width to this value will create the bottom-right corner of the box. This gives a reasonable approximation of the actor's physical dimensions.

To test for world collisions, we now check to see if any of the corners of the bounding box have collided with the edges of the world. To test for collisions with other objects, we'll maintain a list of items that have already been tested for collision. We'll walk through the list and see if any of the corner-points from either of the actors resides inside another actor's bounding box. If so they collide.

That's really all there is to our basic collision detection system. Here is the function to test for an individual collision:

```
    def testForCollision(self,x,y,item,otherItems=[]):
        if x < 0 or x + item.width > 496:
            return self.channel
        elif y < 0 or y+ item.height > 496:
            return self.channel
        else:
            ax1,ax2,ay1,ay2 = x, x+item.width, y,y+item.height
            for item,bx1,bx2,by1,by2 in otherItems:
                if self.registeredActors[item].physical == False: continue
                for x,y in [(ax1,ay1),(ax1,ay2),(ax2,ay1),(ax2,ay2)]:
                    if x >= bx1 and x <= bx2 and y >= by1 and y <= by2:
                        return item
                for x,y in [(bx1,by1),(bx1,by2),(bx2,by1),(bx2,by2)]:
                    if x >= ax1 and x <= ax2 and y >= ay1 and y <= ay2:
                        return item
            return None
```

There is another method that iterates through all actors and tests. It is called during the sendStateToActors() tasklet:

```
    def updateActorPositions(self):
        actorPositions = []
        for actor in self.registeredActors.keys():
            actorInfo = self.registeredActors[actor]
            if actorInfo.public and actorInfo.physical:
                x,y = actorInfo.location
                angle = actorInfo.angle
                velocity = actorInfo.velocity
                VectorX,VectorY = (math.sin(math.radians(angle)) * velocity,
                                   math.cos(math.radians(angle)) * velocity)
                x += VectorX/self.updateRate
                y -= VectorY/self.updateRate
                collision = self.testForCollision(x,y,actorInfo,actorPositions)
                if collision:
                    #don't move
                    actor.send((self.channel,"COLLISION",actor,collision))
```

```
                if collision and collision is not self.channel:
                    collision.send((self.channel,"COLLISION",actor,collision))
            else:
                actorInfo.location = (x,y)
            actorPositions.append( (actor,
                                    actorInfo.location[0],
                                    actorInfo.location[0] + actorInfo.height,
                                    actorInfo.location[1],
                                    actorInfo.location[1] + actorInfo.width))
```

6.2.3 Constant Time

Another problem with our simulation is that it runs on different speeds on different computers. If your computer is faster than mine, I imagine you can barely even see the robots. If it is slower, they may come to a crawl.

To correct this, we'll issue the WORLD_STATE messages at a constant rate. By default we'll go with one every thirtieth of a second. If we could just standardize on that, things would be easy, but we need to be able to correct if the computer cannot handle the load and maintain this update rate. If it takes longer than 1/30 of a second to run an individual frame (either due to the complexity of the program, or an external program hogging resources) we need to adjust the update rate.

For our example, if we accomplish everything we are using more time than we have based on our update rate, we'll decrease the rate by one part per second. If we have 40% or more free time based on our current update rate, we'll increase the rate by one part per second with a maximum cap of 30 updates per second.

This allows us to run at the same speed on different computers, but it introduces an interesting problem. For example, we're currently updating our basicRobot's angle for one degree for each update, and have velocity set per update. If we run the program for ten seconds on two different computers, one running at 20 updates per second and another running 30 updates per second, the robots will be in different locations. This is clearly undesirable. We need to adjust the updates that actors made based on a time-delta.

In the case of the basicRobots, instead of updating the angle 1 degree per turn and (for example) the velocity 5 points per turn, we should calculate this based on the time that has passed. In this case, we'll want to update the angle 30.0 degrees times the time-delta, and the velocity 150.0 points times the time-delta. This way we'll get consistent behaviour regardless of the update rate.

To facilitate this, we'll need to modify the WORLD_STATE message to include both the current time and the update rate, so that actors receiving the message will be able to calculate appropriate update information.

Code implementing the update rate:

```
    def runFrame(self):
        initialStartTime = time.clock()
        startTime = time.clock()
        while 1:
            self.killDeadActors()
            self.updateActorPositions()
            self.sendStateToActors(startTime)
            #wait
            calculatedEndTime = startTime + 1.0/self.updateRate

            doneProcessingTime = time.clock()
            percentUtilized =  (doneProcessingTime - startTime) / (1.0/self.updateRate)
            if percentUtilized >= 1:
                self.updateRate -= 1
                print "TOO MUCH LOWERING FRAME RATE: " , self.updateRate
            elif percentUtilized <= 0.6 and self.updateRate < self.maxupdateRate:
                self.updateRate += 1
```

```
                print "TOO MUCH FREETIME, RAISING FRAME RATE: " , self.updateRate

            while time.clock() < calculatedEndTime:
                stackless.schedule()
            startTime = calculatedEndTime

            stackless.schedule()
```

6.2.4 Damage, Hitpoints, and Dying

Right now our robots are immortal. They will go on forever. That isn't very fun. They should only be able to take so much damage before they die. To facilitate this, we'll add a couple of new messages. The DAMAGE message includes a parameter that indicates the amount of damage received. This is subtracted from the new property *hitpoints* in the basicRobot class. If damage is less than or equal to 0, the actor receiving the message sends a KILLME message to world actor. Here is the applicable code snipped from the defaultMessageAction() method of the basic robot:

```
        elif msg == "DAMAGE":
            self.hitpoints -= msgArgs[0]
            if self.hitpoints <= 0:
                self.world.send( (self.channel,"KILLME") )
        else:
            print "UNKNOWN MESSAGE", args
```

In addition, we've arbitrarily decided that COLLISION messages will deduct one hitpoint and send a KILLME message to the world if necessary.

When the WORLD actor receives a KILLME message, it will set it's internal record of the sending actor's hitpoints to zero. Later, as part of the normal update, it will delete actors with hitpoints less than or equal to zero:

```
    def killDeadActors(self):
        for actor in self.registeredActors.keys():
            if self.registeredActors[actor].hitpoints <= 0:
                print "ACTOR DIED", self.registeredActors[actor].hitpoints
                actor.send_exception(TaskletExit)
                del self.registeredActors[actor]
```

Note that we've introduced the channels *send_exception()* method here. Instead of a normal send, this causes *channel.receive()* to raise an exception in the receiving tasklet. In this case we're raising stackless' TaskletExit exception, which will cause the tasklet to end silently. You could raise any other exception, but if an arbitrary exception is unhanded, it will be re-raised in the main tasklet.

6.2.5 Round 2 of the code

The completed version of this program still isn't too thrilling, but if you run it you'll see that all of the features we have added above are working. The robots will eventually die and disappear after enough collisions.

6.3 Back to the actors: Let's get crazy

> **Note**
>
> The completed program from this section is listed as actors3.py in both the code .zip archive and at the end of this document.

Now that the simulation mechanics are out of the way, we can start to have some fun with the program. First off...

6.3.1 Explosions

It's not very exciting to have the robots simply disappear when they die. They should at least blow up. Robots will create an explosion actor when they die. This is not physical, so it simply displays the explosion image. It will kill itself after three seconds so that the explosion image will disappear:

```
class explosion(actor):
    def __init__(self,location=(0,0),angle=0,world=World):
        actor.__init__(self)
        self.time = 0.0
        self.world = world
        self.world.send((self.channel,"JOIN",
                         properties(self.__class__.__name__,
                                    location = location,
                                    angle = angle,
                                    velocity=0,
                                    height=32.0,width=32.0,hitpoints=1,
                                    physical=False)))

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            WorldState = msgArgs[0]
            if self.time == 0.0:
                self.time = WorldState.time
            elif WorldState.time >= self.time + 3.0:
                self.world.send( (self.channel, "KILLME") )
```

6.3.2 Mine Dropping Robot

Now we'll create a robot that drops mines. Before the robot class, we'll need the mine class:

```
class mine(actor):
    def __init__(self,location=(0,0),world=World):
        actor.__init__(self)
        self.world = world
        self.world.send((self.channel,"JOIN",
                         properties(self.__class__.__name__,
                                    location=location,
                                    angle=0,
                                    velocity=0,
```

```
                                    height=2.0,width=2.0,hitpoints=1)))

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            pass
        elif msg == "COLLISION":
            if msgArgs[0] is self.channel:
                other = msgArgs[1]
            else:
                other = msgArgs[0]
            other.send( (self.channel,"DAMAGE",25) )
            self.world.send( (self.channel,"KILLME"))
            print "MINE COLLISION"
        else:
            print "UNKNOWN MESSAGE", args
```

This is a simple actor. It simply sits there until something hits it, then sends 25 points damage to whatever it collided with and kills itself.

The mindropperRobot is similar to the basic robot, with a few differences. First, to mix things up, I've configured the minedropperRobot to move in a serpentine fashion instead of slowly turning in the same direction. Secondly, it will create a mine every second and place it directly behind itself:

```
class minedropperRobot(actor):
    def __init__(self,location=(0,0),angle=135,velocity=1,
                 hitpoints=20,world=World):
        actor.__init__(self)
        self.location = location
        self.angle = angle
        self.delta = 0.0
        self.height=32.0
        self.width=32.0
        self.deltaDirection = "up"
        self.nextMine = 0.0
        self.velocity = velocity
        self.hitpoints = hitpoints
        self.world = world
        self.world.send((self.channel,"JOIN",
                         properties(self.__class__.__name__,
                                    location=self.location,
                                    angle=self.angle,
                                    velocity=self.velocity,
                                    height=self.height,width=self.width,
                                    hitpoints=self.hitpoints)))

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            for actor in msgArgs[0].actors:
                if actor[0] is self.channel:
                    break
            self.location = actor[1].location
            if self.deltaDirection == "up":
                self.delta += 60.0 * (1.0 / msgArgs[0].updateRate)
                if self.delta > 15.0:
```

```
                self.delta = 15.0
                self.deltaDirection = "down"
        else:
            self.delta -= 60.0 * (1.0 / msgArgs[0].updateRate)
            if self.delta < -15.0:
                self.delta = -15.0
                self.deltaDirection = "up"
        if self.nextMine <= msgArgs[0].time:
            self.nextMine = msgArgs[0].time + 1.0
            mineX,mineY = (self.location[0] + (self.width / 2.0) ,
                           self.location[1] + (self.width / 2.0))

            mineDistance = (self.width / 2.0 ) ** 2
            mineDistance += (self.height / 2.0) ** 2
            mineDistance = math.sqrt(mineDistance)

            VectorX,VectorY = (math.sin(math.radians(self.angle + self.delta)),
                               math.cos(math.radians(self.angle + self.delta)))
            VectorX,VectorY = VectorX * mineDistance,VectorY * mineDistance
            x,y = self.location
            x += self.width / 2.0
            y += self.height / 2.0
            x -= VectorX
            y += VectorY
            mine( (x,y))

        updateMsg = (self.channel, "UPDATE_VECTOR",
                     self.angle + self.delta ,self.velocity)
        self.world.send(updateMsg)
    elif msg == "COLLISION":
        self.angle += 73.0
        if self.angle >= 360:
            self.angle -= 360
        self.hitpoints -= 1
        if self.hitpoints <= 0:
            explosion(self.location,self.angle)
            self.world.send((self.channel,"KILLME"))
    elif msg == "DAMAGE":
        self.hitpoints -= msgArgs[0]
        if self.hitpoints <= 0:
            explosion(self.location,self.angle)
            self.world.send((self.channel, "KILLME"))
    else:
        print "UNKNOWN MESSAGE", args
```

6.3.3 Spawner Pads

Spawner pads simply create new robots with random attributes at their location every five seconds. There is a little black magic in the constructor. Instead of creating an array of valid robot objects, we use introspection to find all classes that end with then name “Robot” and add them to the list. That way, if you create your own robot classes, you won’t need to go through any sort of registration mechanism with the spawner class. Other than that, the class should be reasonably straightforward:

```
class spawner(actor):
    def __init__(self,location=(0,0),world=World):
        actor.__init__(self)
        self.location = location
        self.time = 0.0
        self.world = world

        self.robots = []
        for name,klass in globals().iteritems():
            if name.endswith("Robot"):
                self.robots.append(klass)

        self.world.send((self.channel,"JOIN",
                         properties(self.__class__.__name__,
                                    location = location,
                                    angle=0,
                                    velocity=0,
                                    height=32.0,width=32.0,hitpoints=1,
                                    physical=False)))

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            WorldState = msgArgs[0]
            if self.time == 0.0:
                self.time = WorldState.time + 0.5 # wait 1/2 second on start
            elif WorldState.time >= self.time: # every five seconds
                self.time = WorldState.time + 5.0
                angle = random.random() * 360.0
                velocity = random.random() * 1000.0
                newRobot = random.choice(self.robots)
                newRobot(self.location,angle,velocity)
```

6.3.4 The Final Simulation

We'll finish up by creating spawners in each of the four corners and the center of the world. We now have a simulation that will keep on going and creating new robots as necessary. Feel free to add new robots and play around with the simulation.

6.4 Summary

We've managed to create a reasonably complex simulation with a small amount of code. Even more importantly, each actor runs on it's own. If you consider the messages we've been passing our API, it doesn't really have much to it:

- WORLD_STATE
- JOIN
- UPDATE_VECTOR

- COLLISION
- KILLME
- DAMAGE

Other than that, everything else an actor needs to know about is encapsulated within itself. With only these six messages that it has to deal with to understand the outside world, it simplifies both the program and our ability to understand it.

Chapter 7

Complete Code Listings

7.1 pingpong.py - recursive ping pong example

```
def ping():
    print "PING"
    pong()

def pong():
    print "PONG"
    ping()


ping()
```

7.2 pingpong_stackless.py - stackless ping pong example

```
#
# pingpong_stackless.py
#

import stackless

ping_channel = stackless.channel()
pong_channel = stackless.channel()

def ping():
    while ping_channel.receive(): #blocks here
        print "PING"
        pong_channel.send("from ping")

def pong():
    while pong_channel.receive():
        print "PONG"
        ping_channel.send("from pong")
```

```
stackless.tasklet(ping)()
stackless.tasklet(pong)()

# we need to 'prime' the game by sending a start message
# if not, both tasklets will block
stackless.tasklet(ping_channel.send)('startup')

stackless.run()
```

7.3 hackysackthreaded.py - OS-Thread based hackysack example

```
import thread
import random
import sys
import Queue

class hackysacker:
    counter = 0
    def __init__(self,name,circle):
        self.name = name
        self.circle = circle
        circle.append(self)
        self.messageQueue = Queue.Queue()

        thread.start_new_thread(self.messageLoop,())

    def incrementCounter(self):
        hackysacker.counter += 1
        if hackysacker.counter >= turns:
            while self.circle:
                hs = self.circle.pop()
                if hs is not self:
                    hs.messageQueue.put('exit')
            sys.exit()

    def messageLoop(self):
        while 1:
            message = self.messageQueue.get()
            if message == "exit":
                debugPrint("%s is going home" % self.name)
                sys.exit()
            debugPrint("%s got hackeysack from %s" % (self.name, message.name))
            kickTo = self.circle[random.randint(0,len(self.circle)-1)]
            debugPrint("%s kicking hackeysack to %s" % (self.name, kickTo.name))
            self.incrementCounter()
            kickTo.messageQueue.put(self)
```

```
def debugPrint(x):
    if debug:
        print x

debug=1
hackysackers=5
turns = 5

def runit(hs=10,ts=10,dbg=1):
    global hackysackers,turns,debug
    hackysackers = hs
    turns = ts
    debug = dbg

    hackysacker.counter= 0
    circle = []
    one = hackysacker('1',circle)

    for i in range(hackysackers):
        hackysacker(`i`,circle)

    one.messageQueue.put(one)

    try:
        while circle:
            pass
    except:
        #sometimes we get a phantom error on cleanup.
        pass


if __name__ == "__main__":
    runit(dbg=1)
```

7.4 hackysackstackless.py - stackless hackysack example

```
import stackless
import random
import sys

class hackysacker:
    counter = 0
    def __init__(self,name,circle):
        self.name = name
        self.circle = circle
        circle.append(self)
        self.channel = stackless.channel()
```

```
        stackless.tasklet(self.messageLoop)()

    def incrementCounter(self):
        hackysacker.counter += 1
        if hackysacker.counter >= turns:
            while self.circle:
                self.circle.pop().channel.send('exit')

    def messageLoop(self):
        while 1:
            message = self.channel.receive()
            if message == 'exit':
                return
            debugPrint("%s got hackeysack from %s" % (self.name, message.name))
            kickTo = self.circle[random.randint(0,len(self.circle)-1)]
            while kickTo is self:
                kickTo = self.circle[random.randint(0,len(self.circle)-1)]
            debugPrint("%s kicking hackeysack to %s" % (self.name, kickTo.name))
            self.incrementCounter()
            kickTo.channel.send(self)



def debugPrint(x):
    if debug:print x

debug = 5
hackysackers = 5
turns = 1

def runit(hs=5,ts=5,dbg=1):
    global hackysackers,turns,debug
    hackysackers = hs
    turns = ts
    debug = dbg

    hackysacker.counter = 0
    circle = []
    one = hackysacker('1',circle)

    for i in range(hackysackers):
        hackysacker(`i`,circle)

    one.channel.send(one)

    try:
        stackless.run()
    except TaskletExit:
        pass


if __name__ == "__main__":
    runit()
```

7.5 assemblyline.py - 'normal' assemblyline example

```
class storeroom:
    def __init__(self,name,product,unit,count):
        self.product = product
        self.unit = unit
        self.count = count
        self.name = name

    def get(self,count):
        if count > self.count:
            raise RuntimeError("Not enough %s" % self.product)
        else:
            self.count -= count

        return count

    def put(self,count):
        self.count += count

    def run(self):
        pass

rivetStoreroom = storeroom("rivetStoreroom","rivets","#",1000)
plasticStoreroom = storeroom("plastic Storeroom","plastic pellets","lb",100)

class injectionMolder:
    def __init__(self,name,partName,plasticSource,plasticPerPart,timeToMold):
        self.partName = partName
        self.plasticSource = plasticSource
        self.plasticPerPart = plasticPerPart
        self.timeToMold = timeToMold
        self.items = 0
        self.plastic = 0
        self.time = -1
        self.name = name

    def get(self,items):
        if items > self.items:
            return 0
        else:
            self.items -= items
            return items

    def run(self):
        if self.time == 0:
            self.items += 1
            print "%s finished making part" % self.name
            self.time -= 1
        elif self.time < 0:
```

```
            print "%s starts making new part %s" % (self.name,self.partName)
            if self.plastic < self.plasticPerPart:
                print "%s getting more plastic"
                self.plastic += self.plasticSource.get(self.plasticPerPart * 10)
            self.time = self.timeToMold
        else:
            print "%s molding for %s more seconds" % (self.partName, self.time)
            self.time -= 1


armMolder = injectionMolder("arm Molder", "arms",plasticStoreroom,0.2,6)
legMolder = injectionMolder("leg Molder", "leg",plasticStoreroom,0.2,5)
headMolder = injectionMolder("head Molder","head",plasticStoreroom,0.1,4)
torsoMolder = injectionMolder("torso Molder","torso",plasticStoreroom,0.5,10)


class assembler:
    def __init__(self,name,partAsource,partBsource,rivetSource,timeToAssemble):
        self.partAsource = partAsource
        self.partBsource = partBsource
        self.rivetSource = rivetSource
        self.timeToAssemble = timeToAssemble
        self.itemA = 0
        self.itemB = 0
        self.items = 0
        self.rivets = 0
        self.time = -1
        self.name = name

    def get(self,items):
        if items > self.items:
            return 0
        else:
            self.items -= items
            return items

    def run(self):
        if self.time == 0:
            self.items += 1
            print "%s finished assembling part" % self.name
            self.time -= 1
        elif self.time < 0:
            print "%s starts assembling new part" % self.name
            if self.itemA < 1:
                print "%s Getting item A" % self.name
                self.itemA += self.partAsource.get(1)
                if self.itemA < 1:
                    print "%s waiting for item A" % self.name
            elif self.itemB < 1:
                print "%s Getting item B" % self.name
                self.itemB += self.partBsource.get(1)
                if self.itemB < 1:
                    print "%s waiting for item B" % self.name
            print "%s starting to assemble" % self.name
            self.time = self.timeToAssemble
        else:
            print "%s assembling for %s more seconds" % (self.name, self.time)
```

```
            self.time -= 1

legAssembler = assembler("leg Assembler",torsoMolder,legMolder,rivetStoreroom,2)
armAssembler = assembler("arm Assembler", armMolder,legAssembler,rivetStoreroom,2)
torsoAssembler = assembler("torso Assembler", headMolder,armAssembler,
                           rivetStoreroom,3)

components = [rivetStoreroom, plasticStoreroom, armMolder,
                legMolder, headMolder, torsoMolder,
              legAssembler, armAssembler, torsoAssembler]

def run():
    while 1:
        for component in components:
            component.run()
        raw_input("Press <ENTER> to continue...")
        print "\n\n\n"


if __name__ == "__main__":
    run()
```

7.6 assemblyline-stackless.py - stackless assembly line example

```
class storeroom:
    def __init__(self,name,product,unit,count):
        self.product = product
        self.unit = unit
        self.count = count
        self.name = name

    def get(self,count):
        if count > self.count:
            raise RuntimeError("Not enough %s" % self.product)
        else:
            self.count -= count

        return count

    def put(self,count):
        self.count += count

    def run(self):
        pass

rivetStoreroom = storeroom("rivetStoreroom","rivets","#",1000)
plasticStoreroom = storeroom("plastic Storeroom","plastic pellets","lb",100)

class injectionMolder:
    def __init__(self,name,partName,plasticSource,plasticPerPart,timeToMold):
```

```
        self.partName = partName
        self.plasticSource = plasticSource
        self.plasticPerPart = plasticPerPart
        self.timeToMold = timeToMold
        self.items = 0
        self.plastic = 0
        self.time = -1
        self.name = name

    def get(self,items):
        if items > self.items:
            return 0
        else:
            self.items -= items
            return items

    def run(self):
        if self.time == 0:
            self.items += 1
            print "%s finished making part" % self.name
            self.time -= 1
        elif self.time < 0:
            print "%s starts making new part %s" % (self.name,self.partName)
            if self.plastic < self.plasticPerPart:
                print "%s getting more plastic"
                self.plastic += self.plasticSource.get(self.plasticPerPart * 10)
            self.time = self.timeToMold
        else:
            print "%s molding for %s more seconds" % (self.partName, self.time)
            self.time -= 1


armMolder = injectionMolder("arm Molder", "arms",plasticStoreroom,0.2,6)
legMolder = injectionMolder("leg Molder", "leg",plasticStoreroom,0.2,5)
headMolder = injectionMolder("head Molder","head",plasticStoreroom,0.1,4)
torsoMolder = injectionMolder("torso Molder","torso",plasticStoreroom,0.5,10)


class assembler:
    def __init__(self,name,partAsource,partBsource,rivetSource,timeToAssemble):
        self.partAsource = partAsource
        self.partBsource = partBsource
        self.rivetSource = rivetSource
        self.timeToAssemble = timeToAssemble
        self.itemA = 0
        self.itemB = 0
        self.items = 0
        self.rivets = 0
        self.time = -1
        self.name = name

    def get(self,items):
        if items > self.items:
            return 0
        else:
            self.items -= items
            return items
```

```
    def run(self):
        if self.time == 0:
            self.items += 1
            print "%s finished assembling part" % self.name
            self.time -= 1
        elif self.time < 0:
            print "%s starts assembling new part" % self.name
            if self.itemA < 1:
                print "%s Getting item A" % self.name
                self.itemA += self.partAsource.get(1)
                if self.itemA < 1:
                    print "%s waiting for item A" % self.name
            elif self.itemB < 1:
                print "%s Getting item B" % self.name
                self.itemB += self.partBsource.get(1)
                if self.itemB < 1:
                    print "%s waiting for item B" % self.name
            print "%s starting to assemble" % self.name
            self.time = self.timeToAssemble
        else:
            print "%s assembling for %s more seconds" % (self.name, self.time)
            self.time -= 1

legAssembler = assembler("leg Assembler",torsoMolder,legMolder,rivetStoreroom,2)
armAssembler = assembler("arm Assembler", armMolder,legAssembler,rivetStoreroom,2)
torsoAssembler = assembler("torso Assembler", headMolder,armAssembler,
                           rivetStoreroom,3)

components = [rivetStoreroom, plasticStoreroom, armMolder,
                legMolder, headMolder, torsoMolder,
              legAssembler, armAssembler, torsoAssembler]

def run():
    while 1:
        for component in components:
            component.run()
        raw_input("Press <ENTER> to continue...")
        print "\n\n\n"


if __name__ == "__main__":
    run()
```

7.7 digitalCircuit.py - stackless digital circuit

```
import stackless

debug=0
def debugPrint(x):
    if debug:print x
```

```
class EventHandler:
    def __init__(self,*outputs):
        if outputs==None:
            self.outputs=[]
        else:
            self.outputs=list(outputs)

        self.channel = stackless.channel()
        stackless.tasklet(self.listen)()

    def listen(self):
        while 1:
            val = self.channel.receive()
            self.processMessage(val)
            for output in self.outputs:
                self.notify(output)

    def processMessage(self,val):
        pass

    def notify(self,output):
        pass

    def registerOutput(self,output):
        self.outputs.append(output)

    def __call__(self,val):
        self.channel.send(val)

class Switch(EventHandler):
    def __init__(self,initialState=0,*outputs):
        EventHandler.__init__(self,*outputs)
        self.state = initialState

    def processMessage(self,val):
        debugPrint("Setting input to %s" % val)
        self.state = val

    def notify(self,output):
        output((self,self.state))

class Reporter(EventHandler):
    def __init__(self,msg="%(sender)s send message %(value)s"):
        EventHandler.__init__(self)
        self.msg = msg

    def processMessage(self,msg):
        sender,value=msg
        print self.msg % {'sender':sender,'value':value}

class Inverter(EventHandler):
    def __init__(self,input,*outputs):
        EventHandler.__init__(self,*outputs)
        self.input = input
        input.registerOutput(self)
        self.state = 0
```

```
    def processMessage(self,msg):
        sender,value = msg
        debugPrint("Inverter received %s from %s" % (value,msg))
        if value:
            self.state = 0
        else:
            self.state = 1

    def notify(self,output):
        output((self,self.state))

class AndGate(EventHandler):
    def __init__(self,inputA,inputB,*outputs):
        EventHandler.__init__(self,*outputs)

        self.inputA = inputA
        self.inputAstate = inputA.state
        inputA.registerOutput(self)

        self.inputB = inputB
        self.inputBstate = inputB.state
        inputB.registerOutput(self)

        self.state = 0

    def processMessage(self,msg):
        sender, value = msg
        debugPrint("AndGate received %s from %s" % (value,sender))

        if sender is self.inputA:
            self.inputAstate = value
        elif sender is self.inputB:
            self.inputBstate = value
        else:
            raise RuntimeError("Didn't expect message from %s" % sender)

        if self.inputAstate and self.inputBstate:
            self.state = 1
        else:
            self.state = 0
        debugPrint("AndGate's new state => %s" % self.state)

    def notify(self,output):
        output((self,self.state))

class OrGate(EventHandler):
    def __init__(self,inputA,inputB,*outputs):
        EventHandler.__init__(self,*outputs)

        self.inputA = inputA
        self.inputAstate = inputA.state
        inputA.registerOutput(self)

        self.inputB = inputB
        self.inputBstate = inputB.state
        inputB.registerOutput(self)
```

```
        self.state = 0

    def processMessage(self,msg):
        sender, value = msg
        debugPrint("OrGate received %s from %s" % (value,sender))

        if sender is self.inputA:
            self.inputAstate = value
        elif sender is self.inputB:
            self.inputBstate = value
        else:
            raise RuntimeError("Didn't expect message from %s" % sender)

        if self.inputAstate or self.inputBstate:
            self.state = 1
        else:
            self.state = 0
        debugPrint("OrGate's new state => %s" % self.state)

    def notify(self,output):
        output((self,self.state))


if __name__ == "__main__":
    # half adder
    inputA = Switch()
    inputB = Switch()
    result = Reporter("Result = %(value)s")
    carry = Reporter("Carry = %(value)s")
    andGateA = AndGate(inputA,inputB,carry)
    orGate = OrGate(inputA,inputB)
    inverter = Inverter(andGateA)
    andGateB = AndGate(orGate,inverter,result)
    inputA(1)
    inputB(1)
    inputB(0)
    inputA(0)
```

7.8 actors.py - first actor example

```
import pygame
import pygame.locals
import os, sys
import stackless
import math

class actor:
    def __init__(self):
        self.channel = stackless.channel()
        self.processMessageMethod = self.defaultMessageAction
        stackless.tasklet(self.processMessage)()
```

```
    def processMessage(self):
        while 1:
            self.processMessageMethod(self.channel.receive())

    def defaultMessageAction(self,args):
        print args

class world(actor):
    def __init__(self):
        actor.__init__(self)
        self.registeredActors = {}
        stackless.tasklet(self.sendStateToActors)()

    def testForCollision(self,x,y):
        if x < 0 or x > 496:
            return 1
        elif y < 0 or y > 496:
            return 1
        else:
            return 0

    def sendStateToActors(self):
        while 1:
            for actor in self.registeredActors.keys():
                actorInfo = self.registeredActors[actor]
                if self.registeredActors[actor][1] != (-1,-1):
                    VectorX,VectorY = (math.sin(math.radians(actorInfo[2])) * actorInfo[3],
                                       math.cos(math.radians(actorInfo[2])) * actorInfo[3])
                    x,y = actorInfo[1]
                    x += VectorX
                    y -= VectorY
                    if self.testForCollision(x,y):
                        actor.send((self.channel,"COLLISION"))
                    else:
                        self.registeredActors[actor] = tuple([actorInfo[0],
                                                              (x,y),
                                                                  actorInfo[2],
                                                                  actorInfo[3]])
            worldState = [self.channel, "WORLD_STATE"]
            for actor in self.registeredActors.keys():
                if self.registeredActors[actor][1] != (-1,-1):
                    worldState.append( (actor, self.registeredActors[actor]))
            message = tuple(worldState)
            for actor in self.registeredActors.keys():
                actor.send(message)
            stackless.schedule()

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "JOIN":
            print 'ADDING ' , msgArgs
            self.registeredActors[sentFrom] = msgArgs
        elif msg == "UPDATE_VECTOR":
            self.registeredActors[sentFrom] = tuple([self.registeredActors[sentFrom][0],
                                                     self.registeredActors[sentFrom][1],
                                                     msgArgs[0],msgArgs[1]])
```

```
        else:
            print '!!!! WORLD GOT UNKNOWN MESSAGE ' , args

World = world().channel

class display(actor):
    def __init__(self,world=World):
        actor.__init__(self)

        self.world = World

        self.icons = {}
        pygame.init()

        window = pygame.display.set_mode((496,496))
        pygame.display.set_caption("Actor Demo")

        joinMsg = (self.channel,"JOIN",self.__class__.__name__, (-1,-1))
        self.world.send(joinMsg)

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            self.updateDisplay(msgArgs)
        else:
            print "UNKNOWN MESSAGE", args

    def getIcon(self, iconName):
        if self.icons.has_key(iconName):
            return self.icons[iconName]
        else:
            iconFile = os.path.join("data","%s.bmp" % iconName)
            surface = pygame.image.load(iconFile)
            surface.set_colorkey((0xf3,0x0a,0x0a))
            self.icons[iconName] = surface
            return surface

    def updateDisplay(self,actors):

        for event in pygame.event.get():
            if event.type == pygame.QUIT: sys.exit()

        screen = pygame.display.get_surface()

        background = pygame.Surface(screen.get_size())
        background = background.convert()
        background.fill((200, 200, 200))

        screen.blit(background, (0,0))
        for item in actors:
            screen.blit(pygame.transform.rotate(self.getIcon(item[1][0]),-item[1][2]), item[1][
        pygame.display.flip()

display()

class basicRobot(actor):
    def __init__(self,location=(0,0),angle=135,velocity=1,world=World):
```

```
        actor.__init__(self)
        self.location = location
        self.angle = angle
        self.velocity = velocity
        self.world = world

        joinMsg =(self.channel,"JOIN",self.__class__.__name__,
                  self.location,self.angle,self.velocity)
        self.world.send(joinMsg)

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            self.location = (self.location[0] + 1, self.location[1] + 1)
            self.angle += 1
            if self.angle >= 360:
                self.angle -= 360

            updateMsg = (self.channel, "UPDATE_VECTOR",
                         self.angle,self.velocity)
            self.world.send(updateMsg)
        elif msg == "COLLISION":
            self.angle += 73
            if self.angle >= 360:
                self.angle -= 360
        else:
            print "UNKNOWN MESSAGE", args

basicRobot(angle=135,velocity=5)
basicRobot((464,0),angle=225,velocity=10)


stackless.run()
```

7.9 actors2.py - second actor example

```
import pygame
import pygame.locals
import os, sys
import stackless
import math
import time

class actor:
    def __init__(self):
        self.channel = stackless.channel()
        self.processMessageMethod = self.defaultMessageAction
        stackless.tasklet(self.processMessage)()

    def processMessage(self):
        while 1:
            self.processMessageMethod(self.channel.receive())
```

```
    def defaultMessageAction(self,args):
        print args

class properties:
    def __init__(self,name,location=(-1,-1),angle=0,
                 velocity=0,height=-1,width=-1,hitpoints=1,physical=True,
                 public=True):
        self.name = name
        self.location = location
        self.angle = angle
        self.velocity = velocity
        self.height = height
        self.width = width
        self.public = public
        self.hitpoints = hitpoints
        self.physical = physical

class worldState:
    def __init__(self,updateRate,time):
        self.updateRate = updateRate
        self.time = time
        self.actors = []

class world(actor):
    def __init__(self):
        actor.__init__(self)
        self.registeredActors = {}
        self.updateRate = 30
        self.maxupdateRate = 30
        stackless.tasklet(self.runFrame)()

    def testForCollision(self,x,y,item,otherItems=[]):
        if x < 0 or x + item.width > 496:
            return self.channel
        elif y < 0 or y+ item.height > 496:
            return self.channel
        else:
            ax1,ax2,ay1,ay2 = x, x+item.width, y,y+item.height
            for item,bx1,bx2,by1,by2 in otherItems:
                if self.registeredActors[item].physical == False: continue
                for x,y in [(ax1,ay1),(ax1,ay2),(ax2,ay1),(ax2,ay2)]:
                    if x >= bx1 and x <= bx2 and y >= by1 and y <= by2:
                        return item
                for x,y in [(bx1,by1),(bx1,by2),(bx2,by1),(bx2,by2)]:
                    if x >= ax1 and x <= ax2 and y >= ay1 and y <= ay2:
                        return item
            return None

    def killDeadActors(self):
        for actor in self.registeredActors.keys():
            if self.registeredActors[actor].hitpoints <= 0:
                print "ACTOR DIED", self.registeredActors[actor].hitpoints
                actor.send_exception(TaskletExit)
                del self.registeredActors[actor]

    def updateActorPositions(self):
```

```
        actorPositions = []
        for actor in self.registeredActors.keys():
            actorInfo = self.registeredActors[actor]
            if actorInfo.public and actorInfo.physical:
                x,y = actorInfo.location
                angle = actorInfo.angle
                velocity = actorInfo.velocity
                VectorX,VectorY = (math.sin(math.radians(angle)) * velocity,
                                   math.cos(math.radians(angle)) * velocity)
                x += VectorX/self.updateRate
                y -= VectorY/self.updateRate
                collision = self.testForCollision(x,y,actorInfo,actorPositions)
                if collision:
                    #don't move
                    actor.send((self.channel,"COLLISION",actor,collision))
                    if collision and collision is not self.channel:
                        collision.send((self.channel,"COLLISION",actor,collision))
                else:
                    actorInfo.location = (x,y)
                actorPositions.append( (actor,
                                        actorInfo.location[0],
                                        actorInfo.location[0] + actorInfo.height,
                                        actorInfo.location[1],
                                        actorInfo.location[1] + actorInfo.width))

    def sendStateToActors(self,starttime):
        WorldState = worldState(self.updateRate,starttime)
        for actor in self.registeredActors.keys():
            if self.registeredActors[actor].public:
                WorldState.actors.append( (actor, self.registeredActors[actor]) )
        for actor in self.registeredActors.keys():
            actor.send( (self.channel,"WORLD_STATE",WorldState) )

    def runFrame(self):
        initialStartTime = time.clock()
        startTime = time.clock()
        while 1:
            self.killDeadActors()
            self.updateActorPositions()
            self.sendStateToActors(startTime)
            #wait
            calculatedEndTime = startTime + 1.0/self.updateRate

            doneProcessingTime = time.clock()
            percentUtilized =  (doneProcessingTime - startTime) / (1.0/self.updateRate)
            if percentUtilized >= 1:
                self.updateRate -= 1
                print "TOO MUCH LOWERING FRAME RATE: " , self.updateRate
            elif percentUtilized <= 0.6 and self.updateRate < self.maxupdateRate:
                self.updateRate += 1
                print "TOO MUCH FREETIME, RAISING FRAME RATE: " , self.updateRate

            while time.clock() < calculatedEndTime:
                stackless.schedule()
            startTime = calculatedEndTime

            stackless.schedule()
```

```
    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "JOIN":
            print 'ADDING ' , msgArgs
            self.registeredActors[sentFrom] = msgArgs[0]
        elif msg == "UPDATE_VECTOR":
            self.registeredActors[sentFrom].angle = msgArgs[0]
            self.registeredActors[sentFrom].velocity = msgArgs[1]
        elif msg == "COLLISION":
            pass # known, but we don't do anything
        elif msg == "KILLME":
            self.registeredActors[sentFrom].hitpoints = 0
        else:
            print '!!!! WORLD GOT UNKNOWN MESSAGE ' , args

World = world().channel

class display(actor):
    def __init__(self,world=World):
        actor.__init__(self)

        self.world = World
        self.icons = {}
        pygame.init()

        window = pygame.display.set_mode((496,496))
        pygame.display.set_caption("Actor Demo")

        self.world.send((self.channel,"JOIN",
                         properties(self.__class__.__name__,
                                    public=False)))

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            self.updateDisplay(msgArgs)
        else:
            print "UNKNOWN MESSAGE", args

    def getIcon(self, iconName):
        if self.icons.has_key(iconName):
            return self.icons[iconName]
        else:
            iconFile = os.path.join("data","%s.bmp" % iconName)
            surface = pygame.image.load(iconFile)
            surface.set_colorkey((0xf3,0x0a,0x0a))
            self.icons[iconName] = surface
            return surface

    def updateDisplay(self,msgArgs):

        for event in pygame.event.get():
            if event.type == pygame.QUIT: sys.exit()

        screen = pygame.display.get_surface()
```

```
        background = pygame.Surface(screen.get_size())
        background = background.convert()
        background.fill((200, 200, 200))

        screen.blit(background, (0,0))

        WorldState = msgArgs[0]

        for channel,item in WorldState.actors:
            screen.blit(pygame.transform.rotate(self.getIcon(item.name),-item.angle), item.location)
        pygame.display.flip()

display()

class basicRobot(actor):
    def __init__(self,location=(0,0),angle=135,velocity=1,
                 hitpoints=20,world=World):
        actor.__init__(self)
        self.location = location
        self.angle = angle
        self.velocity = velocity
        self.hitpoints = hitpoints
        self.world = world
        self.world.send((self.channel,"JOIN",
                         properties(self.__class__.__name__,
                                    location=self.location,
                                    angle=self.angle,
                                    velocity=self.velocity,
                                    height=32,width=32,
                                    hitpoints=self.hitpoints)))

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            for actor in msgArgs[0].actors:
                if actor[0] is self: break
            self.location = actor[1].location
            self.angle += 30.0 * (1.0 / msgArgs[0].updateRate)
            if self.angle >= 360:
                self.angle -= 360

            updateMsg = (self.channel, "UPDATE_VECTOR", self.angle,
                         self.velocity)
            self.world.send(updateMsg)
        elif msg == "COLLISION":
            self.angle += 73.0
            if self.angle >= 360:
                self.angle -= 360
            self.hitpoints -= 1
            if self.hitpoints <= 0:
                self.world.send((self.channel, "KILLME"))
        elif msg == "DAMAGE":
            self.hitpoints -= msgArgs[0]
            if self.hitpoints <= 0:
                self.world.send( (self.channel,"KILLME") )
        else:
            print "UNKNOWN MESSAGE", args
```

```
basicRobot(angle=135,velocity=150)
basicRobot((464,0),angle=225,velocity=300)
basicRobot((100,200),angle=78,velocity=500)
basicRobot((400,300),angle=298,velocity=5)
basicRobot((55,55),angle=135,velocity=150)
basicRobot((464,123),angle=225,velocity=300)
basicRobot((180,200),angle=78,velocity=500)
basicRobot((400,380),angle=298,velocity=5)

stackless.run()
```

7.10 actors3.py - third actor example

```
import pygame
import pygame.locals
import os, sys
import stackless
import math
import time
import random

class actor:
    def __init__(self):
        self.channel = stackless.channel()
        self.processMessageMethod = self.defaultMessageAction
        stackless.tasklet(self.processMessage)()

    def processMessage(self):
        while 1:
            self.processMessageMethod(self.channel.receive())

    def defaultMessageAction(self,args):
        print args

class properties:
    def __init__(self,name,location=(-1,-1),angle=0,
                 velocity=0,height=-1,width=-1,hitpoints=1,physical=True,
                 public=True):
        self.name = name
        self.location = location
        self.angle = angle
        self.velocity = velocity
        self.height = height
        self.width = width
        self.public = public
        self.hitpoints = hitpoints
        self.physical = physical

class worldState:
    def __init__(self,updateRate,time):
        self.updateRate = updateRate
```

```
        self.time = time
        self.actors = []

class world(actor):
    def __init__(self):
        actor.__init__(self)
        self.registeredActors = {}
        self.updateRate = 30
        self.maxupdateRate = 30
        stackless.tasklet(self.runFrame)()

    def testForCollision(self,x,y,item,otherItems=[]):
        if x < 0 or x + item.width > 496:
            return self.channel
        elif y < 0 or y+ item.height > 496:
            return self.channel
        else:
            ax1,ax2,ay1,ay2 = x, x+item.width, y,y+item.height
            for item,bx1,bx2,by1,by2 in otherItems:
                if self.registeredActors[item].physical == False: continue
                for x,y in [(ax1,ay1),(ax1,ay2),(ax2,ay1),(ax2,ay2)]:
                    if x >= bx1 and x <= bx2 and y >= by1 and y <= by2:
                        return item
                for x,y in [(bx1,by1),(bx1,by2),(bx2,by1),(bx2,by2)]:
                    if x >= ax1 and x <= ax2 and y >= ay1 and y <= ay2:
                        return item
            return None

    def killDeadActors(self):
        for actor in self.registeredActors.keys():
            if self.registeredActors[actor].hitpoints <= 0:
                print "ACTOR DIED", self.registeredActors[actor].hitpoints
                actor.send_exception(TaskletExit)
                del self.registeredActors[actor]

    def updateActorPositions(self):
        actorPositions = []
        for actor in self.registeredActors.keys():
            actorInfo = self.registeredActors[actor]
            if actorInfo.public and actorInfo.physical:
                x,y = actorInfo.location
                angle = actorInfo.angle
                velocity = actorInfo.velocity
                VectorX,VectorY = (math.sin(math.radians(angle)) * velocity,
                                   math.cos(math.radians(angle)) * velocity)
                x += VectorX/self.updateRate
                y -= VectorY/self.updateRate
                collision = self.testForCollision(x,y,actorInfo,actorPositions)
                if collision:
                    #don't move
                    actor.send((self.channel,"COLLISION",actor,collision))
                    if collision and collision is not self.channel:
                        collision.send((self.channel,"COLLISION",actor,collision))
                else:
                    actorInfo.location = (x,y)
                actorPositions.append( (actor,
                                        actorInfo.location[0],
```

```
                                        actorInfo.location[0] + actorInfo.height,
                                        actorInfo.location[1],
                                        actorInfo.location[1] + actorInfo.width))

    def sendStateToActors(self,starttime):
        WorldState = worldState(self.updateRate,starttime)
        for actor in self.registeredActors.keys():
            if self.registeredActors[actor].public:
                WorldState.actors.append( (actor, self.registeredActors[actor]) )
        for actor in self.registeredActors.keys():
            actor.send( (self.channel,"WORLD_STATE",WorldState) )

    def runFrame(self):
        initialStartTime = time.clock()
        startTime = time.clock()
        while 1:
            self.killDeadActors()
            self.updateActorPositions()
            self.sendStateToActors(startTime)
            #wait
            calculatedEndTime = startTime + 1.0/self.updateRate

            doneProcessingTime = time.clock()
            percentUtilized =  (doneProcessingTime - startTime) / (1.0/self.updateRate)
            if percentUtilized >= 1:
                self.updateRate -= 1
                print "TOO MUCH LOWERING FRAME RATE: " , self.updateRate
            elif percentUtilized <= 0.6 and self.updateRate < self.maxupdateRate:
                self.updateRate += 1
                print "TOO MUCH FREETIME, RAISING FRAME RATE: " , self.updateRate

            while time.clock() < calculatedEndTime:
                stackless.schedule()
            startTime = calculatedEndTime

            stackless.schedule()

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "JOIN":
            self.registeredActors[sentFrom] = msgArgs[0]
        elif msg == "UPDATE_VECTOR":
            self.registeredActors[sentFrom].angle = msgArgs[0]
            self.registeredActors[sentFrom].velocity = msgArgs[1]
        elif msg == "COLLISION":
            pass # known, but we don't do anything
        elif msg == "KILLME":
            self.registeredActors[sentFrom].hitpoints = 0
        else:
            print '!!!! WORLD GOT UNKNOWN MESSAGE ' , msg, msgArgs

World = world().channel

class display(actor):
    def __init__(self,world=World):
        actor.__init__(self)
```

```
        self.world = World
        self.icons = {}
        pygame.init()

        window = pygame.display.set_mode((496,496))
        pygame.display.set_caption("Actor Demo")

        self.world.send((self.channel,"JOIN",
                         properties(self.__class__.__name__,
                                    public=False)))

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            self.updateDisplay(msgArgs)
        else:
            print "DISPLAY UNKNOWN MESSAGE", args

    def getIcon(self, iconName):
        if self.icons.has_key(iconName):
            return self.icons[iconName]
        else:
            iconFile = os.path.join("data","%s.bmp" % iconName)
            surface = pygame.image.load(iconFile)
            surface.set_colorkey((0xf3,0x0a,0x0a))
            self.icons[iconName] = surface
            return surface

    def updateDisplay(self,msgArgs):

        for event in pygame.event.get():
            if event.type == pygame.QUIT: sys.exit()

        screen = pygame.display.get_surface()

        background = pygame.Surface(screen.get_size())
        background = background.convert()
        background.fill((200, 200, 200))

        screen.blit(background, (0,0))

        WorldState = msgArgs[0]

        for channel,item in WorldState.actors:
            itemImage = self.getIcon(item.name)
            itemImage = pygame.transform.rotate(itemImage,-item.angle)
            screen.blit(itemImage, item.location)
        pygame.display.flip()

display()

class basicRobot(actor):
    def __init__(self,location=(0,0),angle=135,velocity=1,
                 hitpoints=20,world=World):
        actor.__init__(self)
        self.location = location
        self.angle = angle
```

```
        self.velocity = velocity
        self.hitpoints = hitpoints
        self.world = world
        self.world.send((self.channel,"JOIN",
                         properties(self.__class__.__name__,
                                    location=self.location,
                                    angle=self.angle,
                                    velocity=self.velocity,
                                    height=32,width=32,hitpoints=self.hitpoints)))

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            for actor in msgArgs[0].actors:
                if actor[0] is self: break
            self.location = actor[1].location
            self.angle += 30.0 * (1.0 / msgArgs[0].updateRate)
            if self.angle >= 360:
                self.angle -= 360

            updateMsg = (self.channel, "UPDATE_VECTOR",
                         self.angle,self.velocity)
            self.world.send(updateMsg)
        elif msg == "COLLISION":
            self.angle += 73.0
            if self.angle >= 360:
                self.angle -= 360
            self.hitpoints -= 1
            if self.hitpoints <= 0:
                explosion(self.location,self.angle)
                self.world.send((self.channel, "KILLME"))
        elif msg == "DAMAGE":
            self.hitpoints -= msgArgs[0]
            if self.hitpoints <= 0:
                explosion(self.location,self.angle)
                self.world.send( (self.channel,"KILLME") )

        else:
            print "BASIC ROBOT UNKNOWN MESSAGE", args

class explosion(actor):
    def __init__(self,location=(0,0),angle=0,world=World):
        actor.__init__(self)
        self.time = 0.0
        self.world = world
        self.world.send((self.channel,"JOIN",
                         properties(self.__class__.__name__,
                                    location = location,
                                    angle = angle,
                                    velocity=0,
                                    height=32.0,width=32.0,hitpoints=1,
                                    physical=False)))

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            WorldState = msgArgs[0]
```

```
            if self.time == 0.0:
                self.time = WorldState.time
            elif WorldState.time >= self.time + 3.0:
                self.world.send( (self.channel, "KILLME") )

class mine(actor):
    def __init__(self,location=(0,0),world=World):
        actor.__init__(self)
        self.world = world
        self.world.send((self.channel,"JOIN",
                         properties(self.__class__.__name__,
                                    location=location,
                                    angle=0,
                                    velocity=0,
                                    height=2.0,width=2.0,hitpoints=1)))

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            pass
        elif msg == "COLLISION":
            if msgArgs[0] is self.channel:
                other = msgArgs[1]
            else:
                other = msgArgs[0]
            other.send( (self.channel,"DAMAGE",25) )
            self.world.send( (self.channel,"KILLME"))
            print "MINE COLLISION"
        else:
            print "UNKNOWN MESSAGE", args

class minedropperRobot(actor):
    def __init__(self,location=(0,0),angle=135,velocity=1,
                 hitpoints=20,world=World):
        actor.__init__(self)
        self.location = location
        self.angle = angle
        self.delta = 0.0
        self.height=32.0
        self.width=32.0
        self.deltaDirection = "up"
        self.nextMine = 0.0
        self.velocity = velocity
        self.hitpoints = hitpoints
        self.world = world
        self.world.send((self.channel,"JOIN",
                         properties(self.__class__.__name__,
                                    location=self.location,
                                    angle=self.angle,
                                    velocity=self.velocity,
                                    height=self.height,width=self.width,
                                    hitpoints=self.hitpoints)))

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            for actor in msgArgs[0].actors:
```

```
                if actor[0] is self.channel:
                    break
            self.location = actor[1].location
            if self.deltaDirection == "up":
                self.delta += 60.0 * (1.0 / msgArgs[0].updateRate)
                if self.delta > 15.0:
                    self.delta = 15.0
                    self.deltaDirection = "down"
            else:
                self.delta -= 60.0 * (1.0 / msgArgs[0].updateRate)
                if self.delta < -15.0:
                    self.delta = -15.0
                    self.deltaDirection = "up"
            if self.nextMine <= msgArgs[0].time:
                self.nextMine = msgArgs[0].time + 1.0
                mineX,mineY = (self.location[0] + (self.width / 2.0) ,
                               self.location[1] + (self.width / 2.0))

                mineDistance = (self.width / 2.0 ) ** 2
                mineDistance += (self.height / 2.0) ** 2
                mineDistance = math.sqrt(mineDistance)

                VectorX,VectorY = (math.sin(math.radians(self.angle + self.delta)),
                                   math.cos(math.radians(self.angle + self.delta)))
                VectorX,VectorY = VectorX * mineDistance,VectorY * mineDistance
                x,y = self.location
                x += self.width / 2.0
                y += self.height / 2.0
                x -= VectorX
                y += VectorY
                mine( (x,y))

            updateMsg = (self.channel, "UPDATE_VECTOR",
                         self.angle + self.delta ,self.velocity)
            self.world.send(updateMsg)
        elif msg == "COLLISION":
            self.angle += 73.0
            if self.angle >= 360:
                self.angle -= 360
            self.hitpoints -= 1
            if self.hitpoints <= 0:
                explosion(self.location,self.angle)
                self.world.send((self.channel,"KILLME"))
        elif msg == "DAMAGE":
            self.hitpoints -= msgArgs[0]
            if self.hitpoints <= 0:
                explosion(self.location,self.angle)
                self.world.send((self.channel, "KILLME"))
        else:
            print "UNKNOWN MESSAGE", args

class spawner(actor):
    def __init__(self,location=(0,0),world=World):
        actor.__init__(self)
        self.location = location
        self.time = 0.0
        self.world = world
```

```
        self.robots = []
        for name,klass in globals().iteritems():
            if name.endswith("Robot"):
                self.robots.append(klass)

        self.world.send((self.channel,"JOIN",
                         properties(self.__class__.__name__,
                                    location = location,
                                    angle=0,
                                    velocity=0,
                                    height=32.0,width=32.0,hitpoints=1,
                                    physical=False)))

    def defaultMessageAction(self,args):
        sentFrom, msg, msgArgs = args[0],args[1],args[2:]
        if msg == "WORLD_STATE":
            WorldState = msgArgs[0]
            if self.time == 0.0:
                self.time = WorldState.time + 0.5 # wait 1/2 second on start
            elif WorldState.time >= self.time: # every five seconds
                self.time = WorldState.time + 5.0
                angle = random.random() * 360.0
                velocity = random.random() * 1000.0
                newRobot = random.choice(self.robots)
                newRobot(self.location,angle,velocity)

spawner( (32,32) )
spawner( (432,32) )
spawner( (32,432) )
spawner( (432,432) )
spawner( (232,232) )

stackless.run()
```