Concurrent Programming: Stackless Python

สุธี สุดประเสริฐ

เนื้อทั้งหมดลอกมาจาก
Introduction to Concurrent Programming with Stackless Python - Grant Olson

ต้นฉบับ: http://www.grant-olson.net/python/intro-to-stackless-python

Why Stackless?

- เพราะมันคือ Python
  - มีโครงสร้างภาษาที่เรียบง่ายและเป็นที่นิยมมากกว่า Mozart/Oz หรือ Erlang
- ง่ายต่อการใช้งานและมีประสิทธิภาพกว่าการใช้วิธีการปกติที่ Python เตรียมไว้

```
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.
```

Why concurrency?

- The real world is '**concurrent**'
    - ในโลกความเป็นจริงจะประกอบขึ้นจากผู้กระทำ (actors) ที่มีปฏิสัมพันธ์กัน โดยที่แต่ละคนมีความเป็นตัวของตัวเอง และแต่ละคนจะรู้จักกับคนอื่นๆ ได้ในขอบเขตที่จำกัด

- OOP เป็นแบบจำลองที่ดีในการจำลองวัตถุหนึ่ง แต่เป็นแบบจำลองที่ไม่ดีนักสำหรับการจำลอง**ความสัมพันธ์ระหว่างวัตถุ** (interaction) ตามความเป็นจริง

The real world is concurrent

- พิจารณาตัวอย่างของโค้ดต่อไปนี้

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

ไม่อะไรผิดปกติและไม่สมเหตุสมผลหรือไม่?

Installing Stackless

- http://zope.stackless.com/download/sdocument_view
- Stackless ไม่ใช่โมดูล แต่คือ Python ทั้งตัวที่มีโมดูลพิเศษสำหรับการทำงานแบบ concurrent
  - หากเลือกติดตั้งเวอร์ชั่นเดียวกันกับ Python เดิมที่มีอยู่ในเครื่อง มันไปแทนที่ Python ตัวเดิม (โมดูลอื่นๆ ได้ติดไว้กับ Python เดิม ก็ยังสามารถใช้ได้ตามปกติ)

Stackless's primitives

- Tasklets
- The Scheduler
- Channels

Tasklets

- Tasklets คือ ส่วนหลักประกอบหลักของ stackless การสร้าง tasklet หนึ่งจะต้องป้อน callable object (function or class method) ให้มัน
  - เมื่อ tasklet ถูกสร้างขึ้น มันจะถูกใส่เข้าไปใน scheduler โดยทันที

```
import stackless

def print_x(x):
  print x

stackless.tasklet(print_x)('one')
stackless.tasklet(print_x)('two')
stackless.tasklet(print_x)('three')

stackless.run()
```

The Scheduler

- scheduler เป็นตัวที่ใช้ควบคุมลำดับการทำงานของ tasklet ซึ่งโดยปกติ tasklet จะเรียงลำดับตามลำดับการสั่ง แต่ว่าแต่ละ tasklet สามารถสั่งให้ตัวมันเองเปลี่ยนลำดับการทำงานในระหว่างที่ทำงานได้ เมื่อเรียกฟังก์ชั่น schedule

```
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(print_three_times)(‘second’)
stackless.tasklet(print_three_times)(‘third’)

stackless.run()
```

เมื่อเรียก schedule แล้ว tasklet นั้นจะหยุดทำงาน และจะถูกนำไปใส่ไว้ในลำดับท้ายสุดใน scheduler

Channels

- channel เป็นส่วนประกอบที่ทำให้ แต่ละ tasklet สามารถส่งข้อมูลหากันได้ ซึ่งมี 2 ประเด็นที่เราสามารถให้ประโยชน์ได้จาก channel คือ
  - สามารถทำให้เราแลกเปลี่ยนข้อมูลระหว่าง tasklets
  - สามารถทำให้เราควบคุมลำดับการทำงานได้

Channel

```
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(sending_tasklet)()
stackless.tasklet(another_tasklet)()
stackless.run()
```

หยุดทำงานถ้าไม่มีผู้ส่ง
เมื่อมีผู้ส่ง ผู้รับจะเริ่มทำงานทันที

หยุดทำงานถ้าไม่มีผู้รับ
เมื่อมีผู้รับข้อมูลจะถูกส่ง
และผู้ส่งจะย้ายไปอยู่ท้ายสุด

Summary

- objects
    - stackless.channel, stackless.tasklet
- functions
    - stackless.run, stackless.schedule
    - channel.send, channel.receive
- ทั้งหมดคือส่วนประกอบหลักของ stackless ซึ่งเพียงพอในการสร้างระบบที่น่าสนใจได้

Coroutines

The problem with to subroutines

- โปรแกรมภาษาทั่วไปจะให้แนวคิดของ subroutines เป็นหลักในการทำงาน
  - subroutine จะถูกจาก routine อื่น ด้วยที่ subroutine ที่ถูกเรียกนั้นจะอยู่ใต้อำนาจ (subordinate) ของผู้เรียก

```
def ping():
  print “ping”
  pong()

def pong():
  print “pong”
  ping()

ping()
```

โค้ดนี้มีปัญหาอะไร?

The stack

- ทุกๆ ครั้งที่ subroutine ถูกเรียก stack frame จะถูกสร้างขึ้นมา ซึ่งเป็นส่วนที่ใช้เก็บค่าตัวแปรและข้อมูลต่างๆ ที่อยู่ใน subroutine

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 ...

- หากจำนวนข้อมูลที่ถูกเก็บมีมากเกินกว่าหน่วยความค่าสูงสุดที่ stack จะรองรับได้ สิ่งที่ตามมาคือปัญหา stack overflow

Why do we use stacks?

- โดยปกติ stack frame จะถูกลบทิ้งทันทีเมื่อ subroutine จบการทำงาน (return ค่าออกมา) ซึ่งการทำงานส่วนใหญ่จะอยู่ในรูปแบบนี้

- ในภาษา C หน่วยความจำส่วนของ stack จะถูกจัดการเองโดยอัตโนมัติ ด้วยเหตุซึ่ง ตัวแปลงภาษา Python เขียนโดย C ดังนั้น การใช้งาน stack ของ Python จึงเหมือน C

- หากเราเรียกใช้ function วนไปวนมา โดยที่ไม่มีการ return ค่า (เช่นในตัวอย่าง ping-pong) จะนำพามาซึ่งปัญหา stack overflow ดังนั้นการทำเช่นนี้จึงเป็นข้อห้ามสำหรับโปรแกรมภาษาส่วนใหญ่

Enter Coroutines

- ในความเป็นจริง การทำงานแบบในตัวอย่าง ping-pong ไม่ได้เป็นการทำงานแบบ subroutines เพราะแต่ละฟังก์ชั่นไม่ได้ขึ้นต่อกัน

- ลักษณะการทำงานแบบนี้เรียกว่า coroutines นั้นคือทั้งสอง routine มีสิทธิเสมอกัน และสามารถทั้งสองสามารถติดต่อสื่อสารกันได้อย่างไม่มีข้อจำกัด

Enter Coroutines

- stackless ใช้ channel ในการสร้าง coroutine เพราะ channel เปิดโอกาสให้ tasklet แต่ละตัว ติดต่อสื่อสารกันและเราสามารถควบคุมลำดับการทำงานของ tasklets ได้

- การใช้ channel ทำให้เราไม่ต้องใช้ stack ดังนั้นการสลับการทำงานระหว่าง pong-ping จึงไม่ทำให้เกิดปัญหา stack overflow

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

Ping-Pong (stackless version)

```
import stackless

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

def ping():
    while 1:
        ping_channel.receive()
        print "Ping"
        pong_channel.send("from ping")

def pong():
    while 1:
        pong_channel.receive()
        print "Pong"
        ping_channel.send("from pong")

stackless.tasklet(ping)()
stackless.tasklet(pong)()
stackless.tasklet(ping_channel.send)('startup')
stackless.run()
```

Summary

- บางครั้งขอบเขตของการโปรแกรมได้ถูกจำกัดด้วยตัวโปรแกรมภาษา เช่น ตัวอย่าง ping-pong (recursive version) จนโปรแกรมเมอร์คิดว่าการกระทำแบบนั้นไม่สามารถทำได้ แต่ในทางทฤษฎีไม่เป็นเช่นนั้น

Lightweight Threads

Threads in stackless

- threadlets คือ thread ที่มีค่าใช้จ่าย (เวลาและทรัพยากร) ในการใช้งาน น้อยกว่า thread แบบปกติที่ Python ใช้ง่ายอยู่เป็นมาตรฐาน

- การสลับการทำงานระหว่าง threadlet นั้นใช้หน่วยความจำน้อยกว่า thread แบบปกติมาก ซึ่งทำให้ใน stackless เราสามารถสร้าง threadlet ได้เป็นจำนวนมาก

The hackysack simulation

1. รอจนกว่าจะมีใครแตะลูกบอกมาให้
2. เมื่อได้ลูกบอกมา จะสุ่มแตะต่อไปให้คนอื่น
3. นับว่าทั้งกลุ่มแตะไปแล้วกี่ครั้ง เมื่อครบกำหนดให้แยกย้ายไปกับบ้าน

- ในการโปรแกรม เราจะให้แต่ละ thread แทน hackysacker แต่ละคน ใน thread จะมีการวนรอบที่มีการรอรับการติดต่อเข้ามาจาก thread อื่น

- การติดต่อมีสองแบบคือ 1) เพื่อบอกว่าแตะลูกบอกไปให้ และ 2) เพื่อบอกว่ามีการแตะลูกบอลครบตามกำหนดแล้ว

The hackysack simulation

- Traditional OS threaded version (hackysackthreaded.py)
  - ใช้เวลามากในทำงาน
  - แตก thread มากๆ ไม่ได้
- Stackless version (hackysackstackless.py)
  - ใช้เวลาในการทำงานเร็วกว่า (ที่ 10,000 thread ใช้เวลาน้อยกว่าแบบปกติ)
  - แตก thread ได้เยอะกว่า

Summary

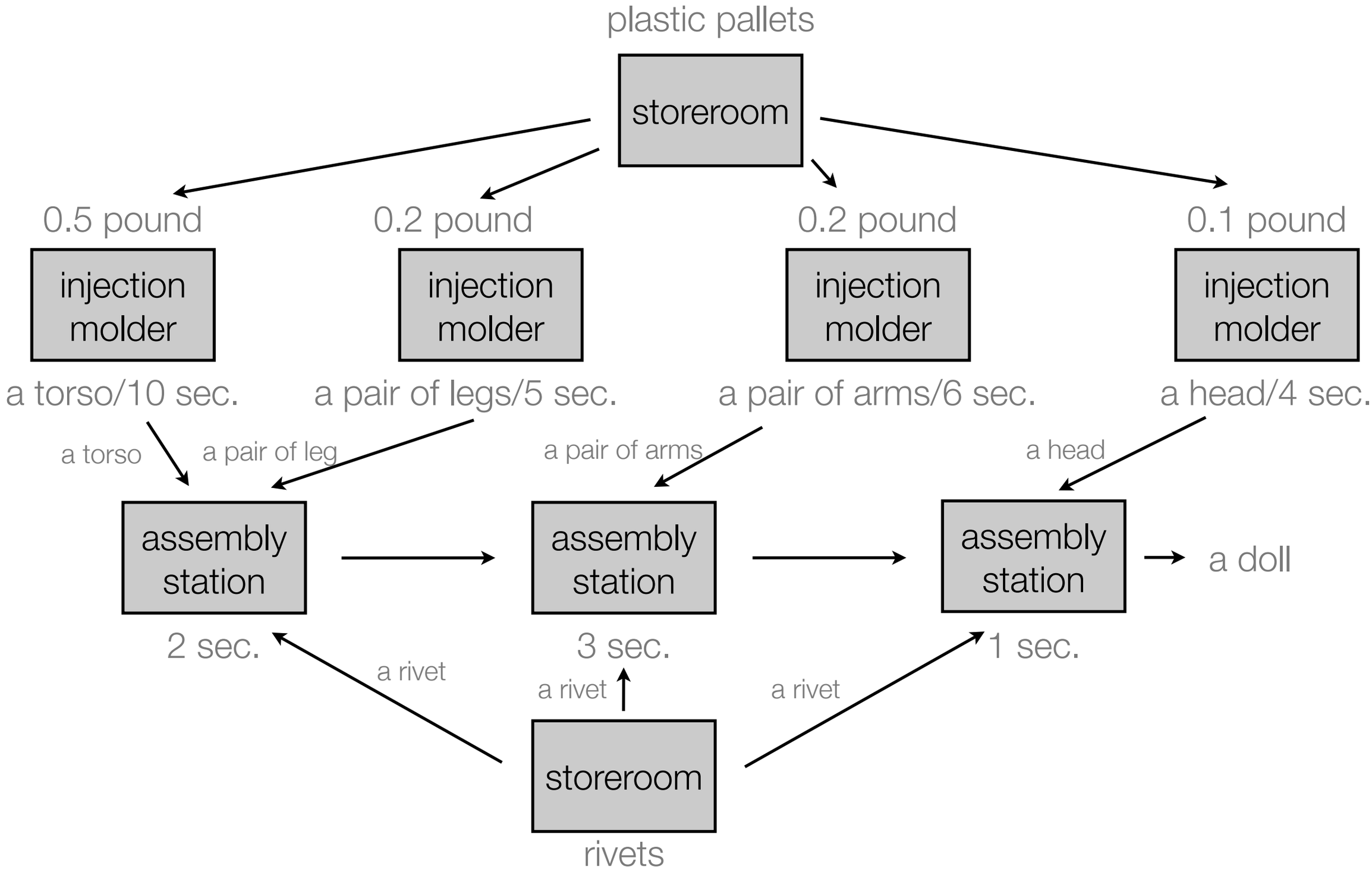
- การที่ OS thread ทำงานได้ช้าเพราะในทุกๆ รอบ จะต้องคอยตรวจสอบอยู่เสมอว่ามี message ส่งเข้ามาใน Queue หรือเปล่า

- แต่ thread ใน stackless จะหยุดการทำงาน เมื่อมีการส่ง message มาถึงจะถูกเรียกเข้ามาทำงานอีกครั้ง

- ดังนั้นเพื่อประสิทธิภาพของโปรแกรมควรใช้งาน OS thread ให้น้อยที่สุดเท่าที่เป็นไปได้

Dataflow

The Factory


plastic pallets
storeroom
0.5 pound
0.2 pound
0.2 pound
0.1 pound
injection molder
injection molder
injection molder
injection molder
a torso/10 sec.
a pair of legs/5 sec.
a pair of arms/6 sec.
a head/4 sec.
a torso
a pair of leg
a pair of arms
a head
assembly station
assembly station
assembly station
a doll
2 sec.
3 sec.
1 sec.
a rivet
a rivet
a rivet
storeroom
rivets

Normal version (assemblyline.py)

- ทุกๆ คลาสที่เป็นส่วนประกอบของระบบจะประกอบด้วย 3 เมดเทดหลักคือ
  - get: ดึงของออกไปใช้
  - put: ใส่ของเข้ามา
  - run: ปฏิบัติงาน
    - ในคลาส storeroom จะประกาศ run ไว้เฉยๆ โดยไม่มีการทำงานใดๆ เพราะ storeroom ไม่ได้มีคำนวณอะไร เพียงทำหน้าที่แค่ส่งของให้กับส่วนอื่นๆ เท่านั้น

Normal version

- ในเมดทอด run ของส่วนประกอบที่มีการปฏิบัติงาน จะมีการตรวจสอบสถานะของปฏิบัติงานว่าดำเนินการไปถึงไหนแล้ว
    - การเรียก run หนึ่งครั้ง ถือว่าเวลาผ่านไป 1 วินาที
- การทำงานหลักของระบบทั้งหมดจะอยู่ในฟังก์ชั่น run ซึ่งจะวนเรียกเมดทอด run ของแต่ละส่วนประกอบไปเรื่อยๆ

Stackless version (assemblyline-stackless.py)

- การโปรแกรมจะใช้เทคนิกที่เรียกว่า dataflow ซึ่งเป็นวิธีการที่ใช้กันแพร่หลายในการใช้งานคำสั่งบนระบบปฏิบัติการตระกูล Unix เช่น
  - `cat README | less`
- ข้อมูลจะถูกส่งผ่านของคำสั่งหนึ่งไปอีกคำส่งตาม ความต้องการของฝ่ายรับ
- ใน stackless เราสามารถใช้ schedule และ channel เพื่อทำงานแบบ dataflow ได้

Stackless version

- การสั่งให้แต่ละส่วนประกอบหยุดเพื่อจำลองการปฏิบัติงาน จะใช้ Sleep ฟังก์ชั่น (ซึ่งเป็น idioms อันหนึ่งของ stackless: http://www.stackless.com/wiki/Idioms)
    - ในกรณีของตัวอย่างนี้ การนับเวลาจะไม่ใช้เวลาจริงๆ แต่จะจำลองผ่านทางตัวแปร sleepingTasks แทน

- รายละเอียดในแต่ละคลาสจะคล้ายกับ normal version โดยมีสิ่งที่แตกต่างดังนี้
    - ไม่จำเป็นต้องใช่ฟังก์ชั่น run เพื่อจำลองการทำงานในแต่ละรอบ
    - การหยุดรอจะจำลองผ่านทาง Sleep แทนการใช้ self.time
    - ถ้าของที่ต้องการมีไม่พอ เมื่อเรียก get แล้ว tasklet นั้น จะถูก schedule ใหม่

Pushing data

- ในตัวอย่างที่แล้ว การทำงานจะเป็นแบบ pulling data คือ แต่ละส่วนประกอบจะได้ดึงทรัพยากรมาจากส่วนประกอบอื่น
    - lazy data flow
- เราสามารถออกแบบระบบให้เป็นแบบ pushing data ได้ คือ ส่วนประกอบหนึ่งจะส่งทรัพยากรไปให้ส่วนประกอบอื่น
    - eager data flow

Digital circuit simulator

- การจำลองการทำงานของ digital circuit สามารถทำได้ด้วยการมองว่า digital circuit ประกอบขึ้นมาจากชิ้นส่วนต่างๆ ที่มาค่าสถานะเป็น 0 หรือ 1 และส่วนประกอบต่างๆ สามารถเชื่อมต่อกันได้หลากหลายรูปแบบ

- เราจะใช้หลักการ OOP โดยให้คลาส EventHandler เป็นคลาสหลักที่แต่ละส่วนประกอบจะต้องสืบทอดมาจากคลาสนี้ หลักการทำงานของ EventHandler มีดังนี้
  - รอรับข้อความที่จะถูกส่งเข้ามาทาง channel ของตัวเอง
  - ประมวลผลข้อความผ่านทางเมดทอด processMessage
  - ส่งผลลัพธ์ไปยังส่วนประกอบที่ถูกลงทะเบียนไว้ใน self.outputs

Digital circuit simulator

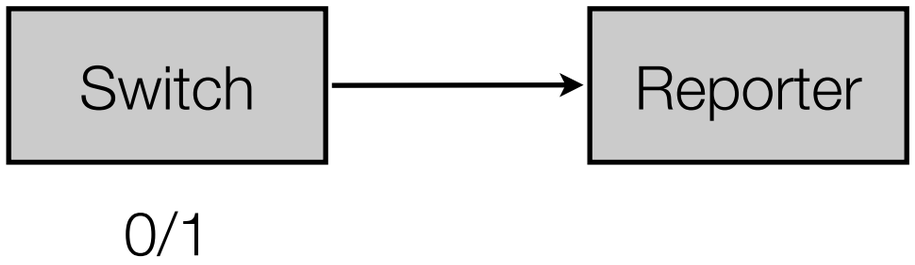
Switch

```
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))
```

Reporter

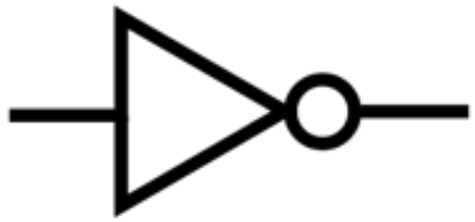
```
class Reporter(EventHandler):
    def __init__(self,msg="%(sender)s send message %(value)s"):
        EventHandler.__init__(self)
        self.msg = msg
    def processMessage(self,msg):
        sender,value=msg
        print self.msg % {'sender':sender,'value':value}
```

Digital circuit simulator




```
>>> 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
>>>
```

Digital circuit simulator



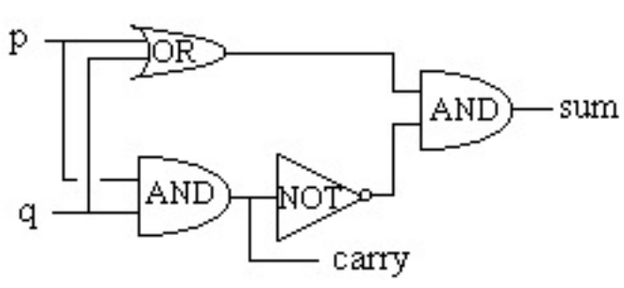
Inverter
(Not gate)

And gate

Or gate

Half Adder

```
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)
```



p	q	sum	carry
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

Actors

The actor model

- ใน actor model เราจะให้ทุกอย่างคือ actor ซึ่งคุณสมบัติของ actor มีดังนี้
    - รับข้อความจาก actor ตัวอื่นๆ ได้
    - ประมวลข้อความที่ได้รับมาตามที่เห็นสมควร
    - สร้าง actor ใหม่ได้
- actor จะไม่สามารถเข้าถึง actor ตัวอื่นโดยตรงได้ แต่จะใช้วิธีการส่งข้อความหากันแทน
- แบบจำลองนี้สามารถจำลองการทำงานของ real-world objects ได้อย่างดี

Killers Robots: Actor base class

- สำหรับตัวอย่างนี้ เราจะสร้างโลกจำลองสำหรับการต่อสู้กันของหุ่นยนต์ โดยที่ actor ทุกตัวจะสืบทอดมาจาก base class นี้

```
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
```

- ทุกๆ actor สร้าง channel สำหรับรับข้อความ และ กำหนดว่าจะใช้เมดทอดใดในการประมวลข้อความที่ได้รับเข้ามา

- การทำงานหลักคือการวนลูปเพื่อรองรับข้อความที่จะถูกส่งเข้ามา

Killers Robots: Message format

- ข้อความทั้งหมดที่ส่งกันในระบบจะอยู่ในรูปแบบคือ
  - channel ของ ผู้ส่ง : self.channel
  - ชื่อของคำส่ง : “JOIN”, “COLLISION”, ...
    - ข้อมูลเพิ่มเติม (optional)
- ตัวอย่างเช่น : (self.channel, “JOIN”, (1,1))
- การส่งเฉพาะ channel จะป้องกันไม่ให้ actor หนึ่งเข้าถึงข้อมูลส่วนตัวของ actor อื่นๆ ดังนั้นการติดต่อกลับจะใช้การส่งผ่านทาง channel ที่ได้รับมาเท่านั้น

Killer robots: World class

- world actor ทำหน้าที่เป็นศูนย์กลางสำหรับการสร้างปฏิสัมพันธ์กันระหว่าง actor
- โดยที่ actor ตัวอื่นๆ จะส่งข้อความ “JOIN” มายัง world actor เพื่อระบุถึงความมีตัวตนในระบบ และ เพื่อให้ world actor สามารถติดตามกระทำของ actor นั้นได้
- world actor จะส่งข้อความ “WORLD_STATE” อย่างต่อเนื่องตามช่วงเวลา เพื่อบอกถึงข้อมูลภายในของตัวเองสู่ actor ตัวอื่นๆ ที่เป็นสมาชิกของระบบ
- ดูตัวอย่างโค้ดในไฟล์ `actors.py`

Killer robots: World class

- เมดทอดหลักของ world actor คือ `sendStateToActors()` ซึ่งทำหน้าที่ในการสร้างสถานะปัจจุบันของ world actor (รวบรวมข้อมูลจาก actor ตัวอื่นๆ) และส่งกลับไปยัง actor ทุกๆ ตัว
    - world actor จะปรับตำแหน่งของ actor ตามมุมและความเร็วของ actor นอกจากนั้นยังตรวจสอบด้วยว่าตำแหน่งของ actor เกิดการชนขอบหรือไม่
- world actor จะสนใจข้อความสองอย่างคือ JOIN และ UPDATE_VECTOR
    - JOIN: เพิ่ม actor โดยจะมีข้อมูลที่รับมาคือ ตำแหน่ง มุม และ ความเร็วของ actor
    - UPDATE_VECTOR: ปรับปรุงมุมและความเร็วของ actor

Killer robots: Simple robot

- simple robot คือหุ่นยนต์ที่เดินด้วยความเร็วคงที่ และหมุนตัวตามเข็มนาฬิกา 1 องศา เหมือนได้รับ WOLRD_STATE ถ้ามีการขนเกิดขึ้น มันจะหมุนตัว 73 องศาและพยายามเดินต่อไป

- ใน constructor เริ่มต้น จะเป็นการส่งข้อความ JOIN ไปยัง world actor

Killer robots: Display (pyGame)

- เป็น actor พิเศษแบบมองไม่เห็น (ตำแหน่ง = (-1,-1)) ทำหน้าที่ในการแสดงข้อมูลทั้งหมดของระบบ
- display จะปรับปรุงข้อมูลทุกครั้งเมื่อได้รับ WORLD_STATE

Simulate mechanics: Actor properties

- การส่งข้อมูลแบบเดิมนั้นไม่สะดวกในกรณีที่เรามีข้อมูลจำนวนมากที่ต้องการส่ง ในกรณีนี้เราควรสร้างคลาสเพื่อใช้ในการแทนข้อมูลเหล่านั้น
    - เป็นวิธีการหนึ่งในการทำ refactor ที่เรียกว่า Introduce Parameter Object (Martin Fowler, 1999 - *Refactoring: Improving the design of existing code*)

```
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
```

Simulate mechanics: Collision Detection

- ในเวอร์ชั่นก่อนหน้า หุ่นยนต์จะไม่ชนกันเองและตำแหน่งในการชนกำแพงไม่ตรงตามความเป็นจริง เพราะ ไม่ได้นำขนาดของหุ่นยนต์มาคำนวณ

- ดังนั้นใน property จึงจำเป็นต้องเก็บกรอบของหุ่นยนต์เอาไว้ โดยใช้ค่า location ในการระบบตำแหน่งจุดขวาบนของหุ่นยนต์ และ height กับ width ในการบอกขนาดความสูงและความกว้าง

- การตรวจสอบการชนจะใช้วิธีคือ ถ้ามุมของวัตถุมุมใดมุมหนึ่งไปทับไปยังวัตถุอื่น ให้ถือว่ามีการชนกันเกิดขึ้น

- ดูตัวอย่างโค้ดในไฟล์ `actors2.py`

Simulate mechanics: Constant time

- อีกปัญหาหนึ่งคือการที่ world actor ส่ง WORLD_STATE ตามรอบของการวนลูป จะส่งผลให้การเดินของหุ่นยนต์ในแต่ละเครื่องมีความเร็วไม่เท่ากัน

- วิธีการแก้ไขปัญหานี้คือเพิ่มค่าอัตราการปรับปรุงเข้าไปในข้อมูลของ WORLD_STATE โดยปกติจะให้เป็น 30 เฟรมต่อวินาที

- แต่ในความเป็นจริง เราไม่สามารถมั่นใจได้ว่าคอมพิวเตอร์แต่ละเครื่องจะสามารถปรับปรุงข้อมูลได้ตามที่กำหนดหรือไม่ ดังนั้นเพื่อให้แต่ละเครื่องทำงานได้ผลเหมือนกัน จึงต้องมีการปรับอัตราการปรับปรุง ให้เหมาะสม สำหรับการทำงานในครั้งนั้น

Simulate mechanics: Constant time

- หากเราใช้เวลาการปรับความถี่ในการปรับปรุง เช่น ถ้ามีเครื่องหนึ่งทำงานได้เร็วมาก ทำให้เมื่อคำนวณเสร็จแล้ว เหลือเวลาเกินกว่า 40% ของเวลาที่ควรใช้ ให้ลดอัตราการปรับปรุงลง 1 วินาที จนกว่าจะเหลือเวลาต่ำกว่า 40% และในทางกลับกันก็ให้เพิ่มอัตราการปรับปรุง

- ผลที่ตามมาคือในระยะเวลาเท่ากัน เครื่องที่มีความเร็วต่างกันจะมีจำนวนการปรับปรุงไม่เท่ากัน คือ เครื่องที่เร็วกว่าจะมีจำนวนการปรับปรุงน้อยกว่าเครื่องที่ช้า

    - ถ้า simple robot หมุนตัว 1 องศาและเคลื่อนด้วยความเร็วคงที่ ดังนั้น simple robot ทั้งสองเครื่องจะอยู่ในตำแหน่งที่แตกต่างกัน

Simulate mechanics: Constant time

- วิธีการที่ถูกต้องคือการปรับปรุงค่าการหมุนและอัตราการเคลื่อนที่ให้สัมพันธ์กับความเร็วของเครื่อง
    - เครื่องเร็ว: ใน 1 รอบจะหมุนน้อยกว่าและเคลื่อนที่ช้ากว่า
    - เครื่องช้า: ใน 1 รอบจะหมุนมากกว่าและเคลื่อนที่ช้ากว่า
- ตัวอย่างเช่น : ถ้า เครื่อง A ทำงานได้เร็วกว่าเครื่อง B 2 เท่า ซึ่งหมายความว่า B ทำงานได้หนึ่งรอบ A จะทำงานได้ 2 รอบ
    - ดังนั้น simple robot บนเครื่อง A ต้องหมุนตัวน้อยกว่าและเดินช้ากว่า B 2 เท่าตัว เพื่อที่จะทำให้ simple robot ของทั้งสองเครื่องอยู่ในตำแหน่งเดียวกัน

Simulate mechanics: Constant time

- ดังนั้น เป้าหมายของเราคือเมื่อเวลาผ่านไปเท่ากันหุ่นยนต์ที่อยู่ในเครื่องช้าหรือเครื่องเร็วควรอยู่ในตำแหน่งเดียวกัน หลักการในการคำนวณมีดังนี้
  - คำนวนเวลาที่ควรใช้ในหนึ่งรอบ และ เปรียบเทียบกับเวลาที่ใช้จริง ซึ่งเวลาที่ใช้จริงควรเป็นหนึ่งส่วนของ frame rate
  - ถ้าเร็วไปให้เพิ่ม frame rate แล้วรอจนกว่าจะใช้เวลาครบ
  - ถ้าช้าไปให้ลด frame rate แล้วทำงานต่อทันที

Simulate mechanics: Damage, hit-points and dying

- ของเดิมหุ่นยนต์จะเป็นอมตะไม่มีวันตาย เพื่อเพิ่มความสนุกเราจะให้หุ่นยนต์มีพลังชีวิต ถ้าพลังชีวิตลดลงเหลือศูนย์ หุ่นยนต์จะตายแล้วถูกเอาออกจากระบบ
  - world actor สามารถส่งข้อความ DAMAGE พร้อมกับ hit-point เพื่อไปลดค่าเพิ่มชีวิตของหุ่นยนต์
    - DAMAGE จะเกิดขึ้นเมื่อหุ่นยนต์ชนกับเองหรือชนกำแพง
  - หุ่นยนต์จะส่งข้อความ KILLME กลับไปยัง world actor เมื่อพลังชีวิตเหลือ 0 เพื่อให้ world actor ลบตัวเองออกจากระบบ