

# Multithreading dengan python (bagian 1)

**Amru Rosyada**

[amrupunya@gmail.com](mailto:amrupunya@gmail.com)

<http://amrurosyada.blogspot.com>

<http://technocratiz.wordpress.com>

## **Lisensi Dokumen:**

Copyright © 2003-2006 IlmuKomputer.Com

Seluruh dokumen di IlmuKomputer.Com dapat digunakan, dimodifikasi dan disebarkan secara bebas untuk tujuan bukan komersial (nonprofit), dengan syarat tidak menghapus atau merubah atribut penulis dan pernyataan copyright yang disertakan dalam setiap dokumen. Tidak diperbolehkan melakukan penulisan ulang, kecuali mendapatkan ijin terlebih dahulu dari IlmuKomputer.Com.

Kali ini kita akan memncoba untuk mengulas teknik multithreading pada python. Sebuah thread sering disebut juga "light-weight" process, karena sistem operasi umumnya menggunakan sedikit resources untuk menciptakan dan memanage thread.

Aplikasi multithread bisa dicontohkan seperti halnya pada web browser, kalo kita lihat dengan seksama browser adalah aplikasi multithreading di satu sisi digunakan surfing dari internet dan pada saat yang bersamman dapat digunakan untuk melakukan download, keduanya adalah proses yang terpisah.

## **Pendahuluan**

Multithreding dapat digunakan untuk mengoptimalkan kinerja komputer, karena dengan multithreading kita bisa memanfaatkan resource-resource yang sedang idle.

Intinya adalah membuat proses mempunyai subproses ataupun kita dapat membuat sharing data untuk proses-proses tersebut sehingga tidak terjadi deadlock saat threading tadi dijalankan. Pada artikel ini akan diulas bagaimana membuat program multithreading pada bahasa pemrograman python, meliputi :

- 1 Pengenalan
- 2 threading Module
- 3 Thread Scheduling
- 4 Thread States: Life Cycle of a Thread
- 5 Thread Synchronization
- 6 Hubungan antara Producer/Consumer Tanpa Sinkronisasi
- 7 Hubungan antara Producer/Consumer dengan Sinkronisasi
- 8 Hubungan antara Producer/Consumer : The Circular Buffer
- 9 Semaphores
- 10 Events
- 11 Daemon Threads

## Isi

Kali ini kita akan mencoba untuk mengulas teknik multithreading pada python. Sebuah thread sering disebut juga "light-weight" process, karena sistem operasi umumnya menggunakan sedikit resources untuk menciptakan dan manage thread.

Aplikasi multithread bisa dicontohkan seperti halnya pada web browser, kalo kita lihat dengan seksama browser adalah aplikasi multithreading di satu sisi digunakan surfing dari internet dan pada saat yang bersamaan dapat digunakan untuk melakukan download, keduanya adalah proses yang terpisah.

### 1. threading Module

Sekarang kita mulai dari membahas module threading, yang merupakan module dasar dari threading pada python. Thread tercipta dengan membuat objek dari class ***threading.Thread***. Biasanya, kita membuat subclass dari class Thread sehingga semua kemampuan dari kelas ini dapat digunakan. Semua kode yang akan dijalankan sebagai thread dimasukkan kedalam method ***run***. method ***run*** merupakan method yang di override dari class Thread.

Thread pada method ***run*** akan dijalankan dengan memanggil method ***start*** yang ada di class ***Thread***. Kemudian caller akan mengeksekusi thread yang dijalankan, jika thread sudah berjalan dan kita menjalankannya kembali maka start method akan mengeluarkan ***AssertionError*** exception.

Method ***isAlive*** akan mengembalikan nilai 1 jika thread masih dalam keadaan berjalan. Method ***setName*** digunakan untuk menset nama Thread. Method ***getName*** akan mengembalikan nama thread. Fungsi ***threading.currentThread*** mengembalikan referensi ke thread yang sedang berjalan saat ini. Fungsi ***threading.enumerate*** akan mengembalikan list dari semua thread yang sedang tereksekusi/berjalan termasuk Thread induk. Fungsi ***threading.activeCount*** akan mengembalikan panjang dari list yang diambil dari ***threading.enumerate***.

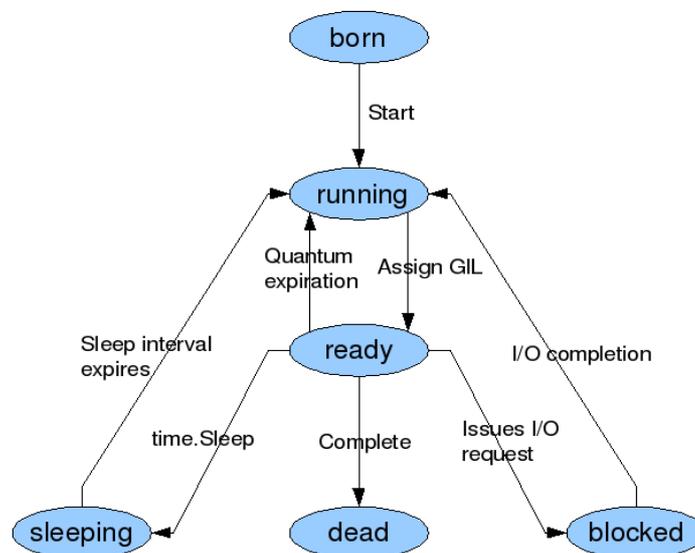
Method ***join*** pada Thread akan menunggu sampai thread yang memanggil method join untuk mati sebelum caller dapat diproses. Sebuah thread mungkin tidak akan memanggil method join-nya sendiri, tetapi oleh thread yang lain. Sebuah parameter yang bersifat optional dapat dilewatkan pada method join yang akan digunakan sebagai timeout, parameternya berupa bilangan floating-point yang akan diartikan sebagai waktu tunggu dalam detik. Jika method join dipanggil tanpa parameter maka akan mengindikasikan bahwa caller menunggu selamanya untuk thread target untuk mati sebelum caller diproses. Beberapa antrian bisa saja berbahaya, yang dapat mengakibatkan deadlock (satu atau lebih thread akan menunggu sampai waktu yang tidak dapat ditentukan bahkan selamanya) dan indefinite postponement (satu atau lebih thread akan didelay untuk waktu yang tidak dapat diprediksi).

## 2. Threading Scheduling

Python interpreter mengontrol semua thread yang ada di program. Ketika interpreter dijalankan, baik pada mode interaktif konsola maupun ketika dipanggil melalui file, thread induk akan dinyalakan. Thread induk ini merupakan caller untuk thread yang lain. Hanya satu thread yang diijinkan berjalan oleh interpreter pada suatu waktu. Interpreter akan mengontrol thread melalui global interpreter lock (GIL). Ketika program terdiri dari lebih dari satu thread, thread-thread tersebut akan di switch keluar masuk interpreter melalui GIL, pada suatu waktu yang telah dispesifikasikan.

## 3. Life Cycle of a Thread

Pada suatu waktu, sebuah thread akan berada pada salah satu state thread (Gambar 1). Dapat dikatakan thread dibuat pada state born. Thread akan berada di state ini sampai method start pada thread dipanggil, ketika method start dipanggil maka thread akan berada pada ready state (atau sering disebut sebagai runnable state). Ready state akan memasuki running state ketika thread dieksekusi (method run dieksekusi). Thread akan memasuki dead state ketika method run selesai tereksekusi atau dimatikan untuk beberapa alasan-interpreter kadangkala akan membuang dead thread.



(gambar 1)

Biasanya running thread memasuki blocked state adalah ketika thread itu mengeluarkan request input/output. Pada kasus ini, sebuah blocked thread akan menjadi ready state ketika I/O menunggu untuk diselesaikan. Interpreter tidak akan mengeksekusi blocked thread walaupun interpreter dalam keadaan free.

Ketika running thread memanggil fungsi `time.sleep`, maka thread akan memasuki thread state. Sebuah sleeping thread akan menjadi ready setelah melewati waktu sleep habis. Sebuah sleeping thread tidak dapat menggunakan interpreter. Sebuah thread memasuki dead state ketika method run telah selesai dijalankan atau melemparkan eksepsi `uncaught exception`.

Program 1 mendemonstrasikan teknik dasar thread, termasuk pembuatan sebuah class yang diturunkan dari `threading.Thread`, pembuatan sebuah thread dan menggunakan fungsi `time.sleep` pada sebuah thread. Setiap thread pada sebuah eksekusi yang dibuat akan menampilkan nama setelah melakukan sleep secara random antara selang waktu 1 dan 5 detik.

# Menunjukkan beberapa thread yang akan menampilkan nama pada interval yang berbeda.

```
import threading
import random
import time

class PrintThread(threading.Thread):
    """Subclass dari threading.Thread"""
    def __init__(self, threadName):
        """Inisialisasi thread, set sleep time, print data"""

        threading.Thread.__init__(self, name=threadName)
        self.sleepTime = random.randrange(1, 6)
        print "Name: %s; sleep: %d" % \
            (self.getName(), self.sleepTime)

    #overridden Thread run method
    def run(self):
        """Slee untuk 1-5 detik"""

        print self.getName(), "going to sleep"
        time.sleep(self.sleepTime)
        print self.getName(), "done sleeping"

thread1 = PrintThread("thread1")
thread2 = PrintThread("thread2")
thread3 = PrintThread("thread3")
thread4 = PrintThread("thread4")

print "\nStarting thread"

thread1.start() #menjalankan thread
thread2.start() #menjalankan thread
thread3.start() #menjalankan thread
thread4.start() #menjalankan thread

print "Thread started\n"
```

*(Program 1)*

```
## hasil eksekusi program
amru@icEcubE:~/Documents/multithread$ python program1.py
Name: thread1; sleep: 3
Name: thread2; sleep: 5
Name: thread3; sleep: 3
```

Name: thread4; sleep: 4

Starting thread

thread1 going to sleep

thread2 going to sleep

thread3 going to sleep

thread4 going to sleep

Thread started

thread1 done sleeping

thread3 done sleeping

thread4 done sleeping

thread2 done sleeping

Class PrintThread--yang diturunkan dari kelas threading.Thread jadi setiap object dari class dapat tereksekusi secara parallel--serta mempunyai atribut sleepTime, sebuah konstruktor dan sebuah method run. Atribut sleepTime menyimpan sebuah nilai random bertipe integer didefinisikan ketika PrintThread objek di buat. Ketika thread start, setiap PrintThread objek akan sleep untuk waktu yang telah ditentukan berdasarkan sleepTime, dan kemudian menampilkan nama thread.

Konstruktor PrintThread (baris 11-17) pertama memanggil base class konstruktor. Melewatkan class instance dan nama thread. Jika nama thread tidak diberikan, thread akan diberikan pengenal yang unik "Thread-n" dimana n adalah integer. Konstruktor kemudian menginisialisasi sleepTime dengan random integer antara 1 dan 5. Kemudian program akan menampilkan output berupa nama thread dan nilai sleepTime, untuk menunjukkan bahwa objek PrintThread telah dibentuk.

Ketika method start dari objek PrintThread di panggil (method start diturunkan dari objek Thread), maka PrintThread akan memasuki ready state. Ketika interpreter melakukan switch dalam PrintThread objek, maka PrintThread akan memasuki running state dan method run method akan mulai dieksekusi. Method run (baris 20-25) akan menampilkan pesan yang mengindikasikan bahwa thread berada dalam keadaan sleep dan kemudian memanggil fungsi time.sleep (baris 24) dengan segera thread akan berada dalam sleeping thread. Ketika thread aktif setelah sleepTime detik, thread akan ditempatkan lagi kedalam ready state sampai thread di switch kedalam prosesor. Ketika PrintThread objek masuk lagi ke running state, thread akan menampilkan nama thread (mengindikasikan bahwa thread selesai dari proses thread), kemudian setelah method run selesai dijalankan maka thread objek akan memasuki dead state.

Poin utama dari program adalah membuat empat objek PrintThread dan memanggil method start dari class thread untuk menempatkan PrintThread objek pada ready state. Setelah semuanya berjalan, program akan terhenti.

#### **4. Thread Synchronization**

Program Multithreading biasanya terdiri dari dua atau lebih thread yang mengakses

dan/atau memodifikasi data yang sama. Sebagai contoh, dua thread membaca dan mengupdate nilai dari sebuah variabel secara simultan. Jika sebuah program multithreading tidak melakukan proteksi akses pada sebuah variabel yang dipakai bersamaan, nilai dari variabel tersebut bisa saja tidak valid. Dimana banyak program yang mengakses resource yang sama dalam hal ini variabel sering disebut sebagai critical section. Untuk mencegah adanya banyak thread yang mengakses share data dan mengubahnya secara simultan, program multithreading biasanya membatasi berapa banyak thread yang dapat mengeksekusi kode dalam critical section pada suatu waktu. Batasan ini dapat dilakukan dengan menggunakan synchronization primitives.

threading module mempunyai banyak thread synchronization primitives. Primitives synchronization yang paling umum adalah lock. lock objek (dibuat dengan kelas `threading.Lock`) mendefinisikan dua method--`acquire` dan `release`. Ketika thread memanggil `acquire` method, thread akan di blok selama waktu yang tidak terbatas. Ketika thread memanggil method `release`, maka lock akan memasuki `unlocked` state dan thread yang diblok akan diaktifkan/notified. Pada bagian ini, thread yang diblok yang lain akan memasuki antrian. Jika thread lebih dari satu diblok pada lock, hanya satu thread yang notified/diaktifkan.

Lock dapat digunakan untuk membatasi akses pada critical section. Program harus dibuat agar thread harus `acquire` lock sebelum memasuki critical section dan melepaskan lock ketika keluar dari critical section. Sehingga, jika satu thread mengeksekusi critical section, thread yang lain akan diblok sebelum memasuki critical section sampai thread yang mengakses critical section tidak ada.

Beberapa prosedur tersebut diatas merupakan level dasar dari synchronization. Kadangkala, kita menginginkan untuk membuat thread lebih canggih saat mengakses critical section hanya ketika ada event yang terjadi (misal ketika nilai data berubah). Ini dapat dilakukan dengan menggunakan condition variable. Sebuah thread menggunakan condition variable ketika thread ingin memonitor state dari beberapa objek atau ingin mengaktifkan thread ketika suatu event terjadi. Ketika state objek berubah atau suatu event terjadi, thread yang diblok akan diaktifkan. Kita akan membahas condition variable pada bagian selanjutnya dan akan dicontohkan sebagai producer/consumer problem. Consumer akan memasuki critical section hanya ketika diaktifkan oleh produser, dan begitu juga sebaliknya.

Condition variable dibuat dengan class ***threading.Condition***. Karena condition variable terdiri dari underlying lock, condition variable mempunyai method `acquire` dan `release`. Condition variable mempunyai tambahan method `wait` dan `notify`. Ketika thread melakukan `acquire` pada underlying lock, memanggil method `wait` untuk melakukan lock dan menyebabkan thread terblok sampai diaktifkan kembali dengan memanggil method `notify` pada condition variable yang sama. Memanggil method `notify` akan mengaktifkan satu thread untuk menunggu pada condition variable. Semua thread yang menunggu dapat diaktifkan dengan memanggil method `notifyAll` pada condition variable.

Semaphore (dibuat dengan class *threading.Semaphore*) merupakan synchronization primitive yang memungkinkan beberapa thread untuk mengakses critical section. Semaphore objek menggunakan counter dan akan mencatat thread mana saja yang acquire dan release pada semaphore. Ketika thread memanggil method acquire, thread akan diblok jika counter bernilai 0. Dengan kata lain, thread acquire pada semaphore dan method acquire akan melakukan dekremen (mengurangi satu) pada counter. Memanggil method release akan membebaskan semaphore, inkremen (menambah satu) counter dan mengaktifkan thread yang menunggu. Inisialisasi nilai pada counter dapat dilewatkan sebagai argumen pada saat memuat objek Semaphore (default bernilai 1). Karena internal counter tidak dapat bernilai negatif, inisialisasi counter dengan nilai negatif akan memunculkan AssertionError exception.

Kadangkala, satu atau lebih thread ingin menunggu suatu event untuk terjadi sebelum mereka memulai eksekusi. Objek event (dibuat dengan class *threading.Event*) mempunyai internal flag yang diinisialisasi dengan nilai false (sebagai tanda bahwa event belum terjadi). Thread memanggil method wait pada Event untuk memblok sampai event terjadi. Ketika event terjadi, method set akan dipanggil untuk menset flag menjadi bernilai true dan mengaktifkan semua thread yang menunggu. Thread yang memanggil method wait setelah flag bernilai true tidak akan diblok keseluruhan. Method *isSet* mengembalikan nilai benar jika flag bernilai benar. Untuk mengembalikan nilai flag menjadi false dapat memanggil method clear.

Membuat program dengan menggunakan lock, condition variable atau synchronization primitive yang lain perlu memperhatikan dan meyakinkan bahwa program tidak deadlock. Program atau thread dikatakan deadlock ketika program atau thread terblok selamanya pada resource yang dibutuhkan. Sebagai contoh, pada saat thread memasuki critical section yang mencoba untuk membuka file. Jika file tidak ditemukan dan thread tidak menangkap suatu eksepsi, pada saat thread memasuki critical section yang mencoba untuk membuka file. Jika file tidak ditemukan dan thread tidak menangkap suatu eksepsi. Thread akan mati sebelum melepaskan lock. Sekarang semua thread akan deadlock, karena thread-thread tersebut terblok dalam waktu yang tak terbatas setelah memanggil method acquire.

## **5. Hubungan antara Producer/Consumer tanpa Sinkronisasi Thread**

Pada bagian ini, kita akan menggunakan hubungan antara producer/consumer untuk mendemonstrasikan method wait dan notify pada condition variable. Pada hubungan antara producer/consumer, sebuah thread producer akan memanggil method produce dan akan mengecek apakah pesan terakhir sudah dibaca oleh consumer, producer akan memanggil method wait pada condition variable. Ketika thread consumer membaca pesan, maka method notify akan dipanggil pada condition variable untuk mengizinkan producer memproduksi pesan kembali. Ketika thread consumer memanggil method consume dan menemui bahwa buffer sedang kosong (tidak ada pesan), maka method wait akan dipanggil. Kemudian producer akan

memanggil method produce dan menaruhnya pada buffer yang kosong, kemudian memanggil method notify sehingga consumer dapat mengaksesnya kembali.

Data yang diakses oleh banyak thread dapat menyebabkan data tersebut corrupt/rusak jika kita tidak melakukan sinkronisasi. Pada hubungan antara producer/consumer yang mana thread producer menyimpan urutan dalam angka (kita gunakan 1,2,3, ...) kedalam slot shared memory. Thread consumer akan membaca data dari shared memory dan menampilkannya ke layar. Program 2 mendemonstrasikan sebuah producer (program 3) dan sebuah consumer (program 4) yang akan mengakses sebuah shared memory tanpa adanya sinkronisasi (program 5). Program ini akan menampilkan apakah yang diproduksi oleh producer (menampilkan yang sedang diproduksi) dan apa yang dikonsumsi oleh consumer (yang sedang dikonsumsi saat ini).

```
# program 2
# menunjukkan multiple thread mengakses shared object

from UnsynchronizedInteger import UnsynchronizedInteger
from ProduceInteger import ProduceInteger
from ConsumeInteger import ConsumeInteger

# initialize integer and threads
number = UnsynchronizedInteger()
producer = ProduceInteger( "Producer", number )
consumer = ConsumeInteger( "Consumer", number )

print "Starting threads...\n"

# start threads
producer.start()
consumer.start()

# wait for threads to terminate
producer.join()
consumer.join()

print "\nAll threads have terminated."
```

*(Program2.py)*

```
## hasil eksekusi program
amru@icEcubE:~/Documents/multithread$ python program2.py
Starting threads...

Producer setting sharedNumber to 1
Consumer retrieving sharedNumber value 1
Producer setting sharedNumber to 2
Consumer retrieving sharedNumber value 2
Consumer retrieving sharedNumber value 2
Producer setting sharedNumber to 3
Consumer retrieving sharedNumber value 3
```

```
Producer setting sharedNumber to 4
Producer setting sharedNumber to 5
Consumer retrieving sharedNumber value 5
Producer setting sharedNumber to 6
Producer setting sharedNumber to 7
Producer setting sharedNumber to 8
Consumer retrieving sharedNumber value 8
Producer setting sharedNumber to 9
Producer setting sharedNumber to 10
Producer finished producing values
Terminating Producer
Consumer retrieving sharedNumber value 10
Consumer retrieved values totaling: 40
Terminating Consumer
```

All threads have terminated.

Karena thread tidak disinkronisasi, data dapat saja hilang jika producer menempatkan data yang baru kedalam slot sebelum consumer mengakses data sebelumnya, dan consumer dapat pula mengakses data yang sama berulang kali jika consumer mengakses data sebelum producer memproduksi item/pesan berikutnya. Untuk menunjukkan kemungkinan ini, thread consumer pada contoh berikut menambahkan semua nilai yang telah dibaca/diakses. Thread producer memproduksi nilai dari 1 sampai 10. Jika consumer mamapu untuk membaca setiap nilai yang diproduksi oleh producer, maka total nilai akan adalah 55. Tetapi, jika program dieksekusi pada waktu yang berdeda/ada tenggat waktu anantara producer dan consumer, maka jumlah total akan jarang mencapai nilai 55. Pada program 2 akan menginisialisasi objek *UnsynchronizedInteger*, kemudian akan dijadikan menjadi argumen/parameter pada waktu pembentukan *ProducerInteger* (objek producer) dan *ConsumerInteger* (objek consumer). Selanjutnya, program memanggil method start pada objek producer dan consumer untuk menempatkan keduanya pada ready state (baris 16-17). Baris 20-21 memanggil method join untuk meyakinkan bahwa main program menunggu sampai kedua thread selesai dieksekusi. Sebagai catatan baris 23 akan dijalankan setelah kedua thread selesai dijalankan.

Class *ProduceInteger* merupakan subclass dari *threading.Thread* terdiri dari atribut *shareObject*, sebuah konstruktor (baris 11-15) dan method run (baris 17-25). Konstruktor akan melakukan inisialisasi pada atribut *sharedObject* untuk menunjuk ke objek *UnsynchronizedInteger* yang dilewatkan sebagai argumen.

```
# Program 3: ProduceInteger.py
# Class that produces integers

import threading
import random
import time

class ProduceInteger( threading.Thread ):
    """Thread to produce integers"""
```

```
def __init__( self, threadName, sharedObject ):
    """Initialize thread, set shared object"""

    threading.Thread.__init__( self, name = threadName )
    self.sharedObject = sharedObject

def run( self ):
    """Produce integers in range 1-10 at random intervals"""

    for i in range( 1, 11 ):
        time.sleep( random.randrange( 4 ) )
        self.sharedObject.setSharedNumber( i )

    print self.getName(), "finished producing values"
    print "Terminating", self.getName()
```

(Program 3 ProduceInteger.py)

Class **ProduceInteger** mempunyai method run yang terdiri dari fro struktur yang melakukan iterasi sebanyak 10 kali. Setiap iterasi pertama-tama akan memanggil fungsi **time.sleep** untuk meletakkan objek **Produceinteger** kedalam sleeping state selama waktu random antara 0 dan 3 detik. Ketika thread diaktifkan, maka akan memanggil method share objek **setSharedNumber** (line 22) dengan nilai kontrol variabel i untuk menset share objek. Ketika iterasi selesai dijalankan, Thread **ProduceInteger** menampilkan pesan pada command prompt untuk menunjukkan bahwa proses produksi telah selesai.

Class **ConsumeInteger** merupakan subclass dari threading.Thread terdiri dari atribut **sharedObject**, sebuah konstruktor (baris 11-15) dan sebuah method **run** (baris 17-29). Konstruktor akan melakukan inisialisasi pada sribut **sharedObject** untuk menunjuk ke objek **UnsynchronizedInteger** yang dilewatkan sebagai argumen.

```
# Program 4: ConsumeInteger.py
# Class that consumes integers

import threading
import random
import time

class ConsumeInteger( threading.Thread ):
    """Thread to consume integers"""

    def __init__( self, threadName, sharedObject ):
        """Initialize thread, set shared object"""

        threading.Thread.__init__( self, name = threadName )
        self.sharedObject = sharedObject

    def run( self ):
        """Consume 10 values at random time intervals"""

        sum = 0          # total sum of consumed values
```

```
# consume 10 values
for i in range( 10 ):
    time.sleep( random.randrange( 4 ) )
    sum += self.sharedObject.getSharedNumber()

print "%s retrieved values totaling: %d" % \
    ( self.getName(), sum )
print "Terminating", self.getName()
```

(Program 4 ConsumeInteger.py)

Class *ConsumeInteger* mempunyai method run yang terdiridari for struktur yang melakukan iterasi sebanyak 10 kali untuk membaca nilai dari objek *UnsynchronizedInteger* yang ditunjuk oleh *sharedObject*. Setiap iterasi dari loop memanggil fungsi `time.sleep` untuk meletakkan objek *ConsumeInteger* kedalam sleeping state selama waktu random antara 0 dan 3 detik. Selanjutnya, Thread memanggil method *getSharedNumber* untuk mendapatkan nilai dari share objek. Kemudian, thread menambahkan variable sum keudian nilai dar variabel itu akan dikembalikan oleh method *getSharedNumber* (line 25). Ketika iterasi selesai dijalankan, thread *ConsumeInteger* menampilkan pesan ke layar command prompt yang menandakan bahwa data telah selesai dikonsumsi.

Class *UnsynchronizedInteger* mempunyai method *setSharedNumber* (line 14-19) dan *getSharedNumber* (21-28) tidak melakukan sinkronisasi dalam mengakses variabel *sharedNumber* (line 12). Idealnya, kita menginginkan setiap nilai yang dihasilkan oleh objek *ProduceInteger* untuk dikonsumsi tepat satu oleh objek *ConsumeInteger*. Tetapi Pada output program *ConsumeInteger* ada beberapa nilai yang lost/terlewat.

```
# Program 5: UnsynchronizedInteger.py
# Unsynchronized access to an integer

import threading

class UnsynchronizedInteger:
    """Class that provides unsynchronized access an integer"""

    def __init__( self ):
        """Initialize shared number to -1"""

        self.sharedNumber = -1

    def setSharedNumber( self, newNumber ):
        """Set value of integer"""

        print "%s setting sharedNumber to %d" % \
            ( threading.currentThread().getName(), newNumber )
        self.sharedNumber = newNumber

    def getSharedNumber( self ):
        """Get value of integer"""
```

```
tempNumber = self.sharedNumber
print "%s retrieving sharedNumber value %d" % \
      ( threading.currentThread().getName(), tempNumber )

return tempNumber
```

*(Program 5 UnsynchronizedInteger.py)*

Pada kenyatannya, method **getSharedNumber** harus melakukan beberapa trick agar bisa mengeluarkan output dengan tepat. Baris 24 variabel **tempNumber** diisi dengan **sharedNumber**. Pada baris 25-28 menggunakan nilai **tempNumber** untuk menampilkan pesan dan mengembalikan nilai. Jika kita tidak menggunakan temporary variabel, scenario diatas tidak akan berjalan. Consumer akan memanggil method **getShareNumber** dan menampilkan nilai data. Interpreter kemudian akan mengeluarkan thread consumer untuk kemudian memasukkan thread producer. Thread producer kemudian akan mengubah nilai dari **shareNumber** dengan memanggil method **setSharedNumber**. Begitu seterusnya, interpreter memindahkan consumer kembali dan method **getSharedNumber** mengembalikan nilai yang berbeda sebelum consumer dipindahkan dari interpreter.

## Penutup

Diharapkan dengan adanya artikel ini bisa membantu dalam meningkatkan kemajuan teknologi informasi di Indonesia dan mendukung suksesnya IGOS

## Referensi

<http://python.org>

Python How to Program, <http://www.deitel.com>

## Biografi Penulis



**Amru Rosyada.** Lahir pada tanggal 22 Mei 1986, menamatkan pendidikan dasar sampai pendidikan menengah akhir di kota Ngawi kemudian terdampar di Jogja mengambil program Diploma tiga Teknik Elektro Universitas Gadjah Mada dan Sekarang masih menamatkan Strata satu di Ilmukomputer Universitas Gadjah Mada.