

# Multithreading dengan python (ending) – dengan semaphore dan event

**Amru Rosyada**

[amrupunya@gmail.com](mailto:amrupunya@gmail.com)  
<http://amrurosya.blogspot.com>  
<http://technocratiz.wordpress.com>

#### **Lisensi Dokumen:**

Copyright © 2003-2006 IlmuKomputer.Com

Seluruh dokumen di IlmuKomputer.Com dapat digunakan, dimodifikasi dan disebarluaskan 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.

Alhamdulillah ini merupakan artikel seri terakhir dari *multithreading* di python, pada bagian akhir ini akan dibahas tentang *threading* menggunakan *semaphore* dan *event*. Tidak jauh beda dengan teknik *threading* yang telah dipaparkan pada artikel-artikel sebelumnya, tetapi kali ini akan lebih simple dan mudah untuk diimplementasikan. Teknik *threading* yang telah diulas pada artikel-artikel ini akan sangat bergantung sekali pada kebutuhan dan masing-masing mempunyai kelebihan masing-masing.

## 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

## Semaphore dan Event

### Semaphore

*Semaphore* adalah sebuah variabel yang mengontrol *resource* yang sedang digunakan/*critical section*. Semaphore akan melakukan *counter* yang menunjukkan jumlah *thread* yang dapat menggunakan *resource* atau memasuki *critical section*. Counter akan dikurangi dengan satu/dekremen setiap kali *thread* melakukan *acquire* pada *semaphore*. Ketika *counter* bernilai 0, *semaphore* melakukan blok pada *thread* yang lain sampai *semaphore* dilepaskan oleh *thread* yang lain. Pada program *semaphore.py* berikut akan menunjukkan contoh *semaphore*.

```
# semaphore.py
# Using a semaphore to control access to a critical section

import threading
import random
import time

class SemaphoreThread( threading.Thread ):
    """Class using semaphores"""

    availableTables = [ "A", "B", "C", "D", "E" ]

    def __init__( self, threadName, semaphore ):
        """Initialize thread"""

        threading.Thread.__init__( self, name = threadName )
        self.sleepTime = random.randrange( 1, 6 )

        # set the semaphore as a data attribute of the class
        self.threadSemaphore = semaphore

    def run( self ):
        """Print message and release semaphore"""

        # acquire the semaphore
        self.threadSemaphore.acquire()
```

```
# remove a table from the list
table = SemaphoreThread.availableTables.pop()
print "%s entered; seated at table %s." % \
    ( self.getName(), table ),
print SemaphoreThread.availableTables
time.sleep( self.sleepTime ) # enjoy a meal

# free a table
print " %s exiting; freeing table %s." % \
    ( self.getName(), table ),
SemaphoreThread.availableTables.append( table )
print SemaphoreThread.availableTables

# release the semaphore after execution finishes
self.threadSemaphore.release()

threads = [] # list of threads

# semaphore allows five threads to enter critical section
threadSemaphore = threading.Semaphore(
    len( SemaphoreThread.availableTables ) )

# create ten threads
for i in range( 1, 11 ):
    threads.append( SemaphoreThread( "thread" + str( i ),
        threadSemaphore ) )

# start each thread
for thread in threads:
    thread.start()
```

*(semaphore.py)*

## OUTPUT PROGRAM

```
amru@icEcubE:~/Documents/multithread$ python semaphore.py
thread1 entered; seated at table E. ['A', 'B', 'C', 'D']
thread2 entered; seated at table D. ['A', 'B', 'C']
thread3 entered; seated at table C. ['A', 'B']
thread4 entered; seated at table B. ['A']
```

```
thread5 entered; seated at table A. []
thread2 exiting; freeing table D. ['D']
thread5 exiting; freeing table A. ['D', 'A']
thread6 entered; seated at table A. ['D']
thread7 entered; seated at table D. []
thread1 exiting; freeing table E. ['E']
thread8 entered; seated at table E. []
thread8 exiting; freeing table E. ['E']
thread9 entered; seated at table E. []
thread3 exiting; freeing table C. ['C']
thread4 exiting; freeing table B. ['C', 'B']
thread10 entered; seated at table B. ['C']
thread6 exiting; freeing table A. ['C', 'A']
thread7 exiting; freeing table D. ['C', 'A', 'D']
thread9 exiting; freeing table E. ['C', 'A', 'D', 'E']
thread10 exiting; freeing table B. ['C', 'A', 'D', 'E', 'B']
```

pada program *semaphore.py* terdapat baris untuk membuat objek *threading.Semaphore* yang parameternya (*len( SemaphoreThread.availableTables )*) berupa banyaknya *thread* yang boleh mengakses *critical section* pada suatu waktu. Baris selanjutnya akan memasuki iterasi *for* yang akan membuat objek *thread* (*SemaphoreThread*) sebanyak 10, kemudian akan dilanjutkan ke iterasi *loop* yang kedua untuk menjalankan masing-masing *thread* dengan memanggil method *start*.

Class *SemaphoreThread* dapat dianalogikan sebagai *single customer* pada restoran, kemudian atribut *availableTables* akan mencatat meja-meja yang tersedia di restoran.

*Semaphore* objek mempunyai *built-in counter* untuk mencatat jumlah pemanggilan method *acquire* dan *release*. Jika *counter* lebih besar dari 0, method *acquire* akan mendapatkan *semaphore* untuk *thread* dan kemudian mengurangi *coounter*. Jika *counter* 0, *thread* akan diblok sampai ada *thread* lain yang melepaskan *semaphore*.

Method *pop* akan mengambil item terakhir dari *availableTables* yang berarti *thread* lain akan mulai untuk memasuki *critical section*. Program *semaphore.py* akan menampilkan *thread* mana yang sedang memasuki *critical section*. Kemudian item yang diambil akan dimasukkan kembali/appends ke *availableTables* yang menandakan bahwa *thread* telah siap untuk keluar dari *critical section*.

method *release* pada objek *semaphore* akan melepaskan *semaphore* ketika *thread* selesai memasuki *critical section*. Method *release* akan melakukan inkremen pada *counter* dan akan mengaktifkan *thread* yang sedang menunggu.

## Events

Module *threading* mempunyai class ***Event***, yang sangat berguna untuk komunikasi *thread*. Sebuah objek ***Event*** mempunyai *flag internal*, yang bisa benilai *true/false*. Satu atau lebih *thread* mungkin akan memanggil *method wait* dari objek ***Event*** sampai suatu ***event*** terjadi. Ketika suatu *event* terjadi, *thread-thread* akan dibangkitkan. Program *events.py* berikut akan mendemonstrasikan situasi dimana *traffic light* akan menyala hijau setiap 3 detik.

```
# events.py
# Demonstrating Event objects

import threading
import random
import time

class VehicleThread( threading.Thread ):
    """Class representing a motor vehicle at an intersection"""

    def __init__( self, threadName, event ):
        """Initializes thread"""

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

        # ensures that each vehicle waits for a green light
        self.threadEvent = event

    def run( self ):
        """Vehicle waits unless/until light is green"""

        # stagger arrival times
        time.sleep( random.randrange( 1, 10 ) )

        # prints arrival time of car at intersection
        print "%s arrived at %s" % \
            ( self.getName(), time.ctime( time.time() ) )

        # flag is false until two vehicles are queued
        self.threadEvent.wait()
```

```
# displays time that car departs intersection
print "%s passes through intersection at %s" \% \
( self.getName(), time.ctime( time.time() ) )

greenLight = threading.Event()
vehicleThreads = []

# creates and starts ten Vehicle threads
for i in range( 1, 11 ):
    vehicleThreads.append( VehicleThread( "Vehicle" + str( i ),
greenLight ) )

for vehicle in vehicleThreads:
    vehicle.start()

while threading.activeCount() > 1:

    # sets the Event object's flag to false
    greenLight.clear()
    print "RED LIGHT!"
    time.sleep( 3 )

    # sets the Event object's flag to true
    print "GREEN LIGHT!"
    greenLight.set()
```

*(events.py)*

## OUTPUT PROGRAM

```
amru@icEcubE:~/Documents/multithread$ python events.py
RED LIGHT!
Vehicle10 arrived at Thu Sep 25 13:24:23 2008
Vehicle7 arrived at Thu Sep 25 13:24:24 2008
GREEN LIGHT!
RED LIGHT!
Vehicle10 passes through intersection at Thu Sep 25 13:24:25 2008
Vehicle7 passes through intersection at Thu Sep 25 13:24:25 2008
Vehicle5 arrived at Thu Sep 25 13:24:26 2008
```

```
Vehicle1 arrived at Thu Sep 25 13:24:28 2008
Vehicle2 arrived at Thu Sep 25 13:24:28 2008
Vehicle6 arrived at Thu Sep 25 13:24:28 2008
Vehicle8 arrived at Thu Sep 25 13:24:28 2008
GREEN LIGHT!
RED LIGHT!
Vehicle5 passes through intersection at Thu Sep 25 13:24:28 2008
Vehicle8 passes through intersection at Thu Sep 25 13:24:28 2008
Vehicle2 passes through intersection at Thu Sep 25 13:24:28 2008
Vehicle6 passes through intersection at Thu Sep 25 13:24:28 2008
Vehicle1 passes through intersection at Thu Sep 25 13:24:28 2008
Vehicle9 arrived at Thu Sep 25 13:24:29 2008
Vehicle3 arrived at Thu Sep 25 13:24:30 2008
Vehicle4 arrived at Thu Sep 25 13:24:31 2008
GREEN LIGHT!
RED LIGHT!
Vehicle9 passes through intersection at Thu Sep 25 13:24:31 2008
Vehicle3 passes through intersection at Thu Sep 25 13:24:31 2008
Vehicle4 passes through intersection at Thu Sep 25 13:24:31 2008
GREEN LIGHT!
```

Pada program *events.py* didalamnya terdapat class *VehicleThread* yang merupakan gambaran sebagai sebuah kendaraan pada persimpangan, jadi setiap kendaraan di analogikan sebagai sebuah *thread*. Tiap *thread* akan *sleep* selama waktu acak/random, kemudian akan menampilkan pesan berapa kendaraan yang datang/ngantri di lampu merah, menunggu sampai *traffic light* berwarna hijau (dengan kata lain lampu hijau menandakan *internal flag* bernilai *true*) dan kemudian akan menampilkan berapa kendaraan yang lewat selama lampu hijau menyala.

Struktur *while* akan *looping* sampai hanya *main thread* yang tersisa (dalam artian semua *thread* kendaraan telah selesai). Setiap *iterasi* memanggil *method clear* dari objek *Event*, kemudian *sleep* selama 3 detik dan memanggil *method set* dari objek *Event*. *Method clear (false)* dan *set (true)* akan mengubah nilai dari *internal flag*.

## 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.