

Multithreading dengan python (bagian 2) – dengan sinkronisasi

Amru Rosyada

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

Hubungan antara Producer/Consumer dengan Sinkronisasi

Program sinkron.py akan mendemonstrasikan antara producer dengan consumer yang akan mengakses share data dengan sinkronisasi, jadi bisa dikatakan bahwa consumer akan tepat mengkonsumsi satu kali setelah producer memproduksi. Yang berbeda pada bagian ini adalah kode program untuk *UnsynchronizedInteger* pada bagian artikel bagian pertama diganti dengan *SinchronizedInteger*, sedangkan untuk class *ProduceInteger* dan *ConsumeInteger* identik/sama dengan kode program pada bagian multithreading tak sinkron pada artikel pertama, oiya ada lagi yang berbeda adalah pada program inti kalo yang pada artikel pertama menggunakan nama *program1.py* sekarang menggunakan nama *sinkron.py*.

```
# sinkron.py
# Show multiple threads modifying shared object.

from SynchronizedInteger import SynchronizedInteger
from ProduceInteger import ProduceInteger
from ConsumeInteger import ConsumeInteger

# initialize number and threads
number = SynchronizedInteger()
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."
```

(*sinkron.py*)

output program setelah dijalankan adalah sebagai berikut :

```
Starting threads...

Producer setting sharedNumber to 1
Consumer retrieving sharedNumber value 1
Producer setting sharedNumber to 2
Consumer retrieving sharedNumber value 2
Producer setting sharedNumber to 3
Consumer retrieving sharedNumber value 3
Producer setting sharedNumber to 4
Consumer retrieving sharedNumber value 4
```

```
Producer setting sharedNumber to 5
Consumer retrieving sharedNumber value 5
Producer setting sharedNumber to 6
Consumer retrieving sharedNumber value 6
Producer setting sharedNumber to 7
Consumer retrieving sharedNumber value 7
Producer setting sharedNumber to 8
Consumer retrieving sharedNumber value 8
Producer setting sharedNumber to 9
Consumer retrieving sharedNumber value 9
Producer setting sharedNumber to 10
Producer finished producing values
Terminating Producer
Consumer retrieving sharedNumber value 10
Consumer retrieved values totaling: 55
Terminating Consumer
```

All threads have terminated.

(output program sinkron.py)

Class *SynchronizedInteger* terdiri dari 3 atribut yaitu *sharedNumber*, *writable* dan *threadCondition*. Method *setSharedNumber* menggunakan condition variable untuk menentukan bahwa method yang memanggil dapat mengubah nilai dari share memori. Method *getShareNumber* menggunakan condition variable untuk menentukan bahwa method yang memanggil hanya dapat membaca nilai dari share memori. Baris 14 adalah sintak yang digunakan untuk membuat thread condition variable dengan memanggil konstruktor *threading.Condition*. Karena tidak ada argumen yang dilewatkan pada pembuatan conditional variable, maka lock baru akan dibuat untuk condition variable.

```
# SynchronizedInteger.py
# Synchronized access to an integer with condition variable

import threading

class SynchronizedInteger:
    """Class that provides synchronized access an integer"""

    def __init__( self ):
        """Set shared number, write flag and condition variable"""

        self.sharedNumber = -1
        self.writable = 1 # the value can be changed
        self.threadCondition = threading.Condition()

    def setSharedNumber( self, newNumber ):
        """Set value of integer--blocks until lock acquired"""

        # block until lock released then acquire lock
        self.threadCondition.acquire()
```

```
# while not producer's turn, release lock and block
while not self.writeable:
    self.threadCondition.wait()

# (lock has now been re-acquired)

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

self.sharedNumber = newNumber
self.writeable = 0      # allow consumer to consume
self.threadCondition.notify() # wake up a waiting thread
self.threadCondition.release() # allow lock to be acquired

def getSharedNumber( self ):
    """Get value of integer--blocks until lock acquired"""

    # block until lock released then acquire lock
    self.threadCondition.acquire()

    # while producer's turn, release lock and block
    while self.writeable:
        self.threadCondition.wait()

    # (lock has now been re-acquired)

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

    self.writeable = 1      # allow producer to produce
    self.threadCondition.notify() # wake up a waiting thread
    self.threadCondition.release() # allow lock to be acquired

    return tempNumber
```

(SynchronizedInteger.py)

Konstruktor pada baris 9-14 membuat atribut **writeable** dan diinisialisasi dengan nilai 1. Class condition variable **threadCondition** akan memproteksi akses ke atribut **writeable**. Jika writeable bernilai 1, maka producer akan menempatkan sebuah nilai pada variable **sharedNumber**, ini berarti consumer tidak dapat membaca nilai dari sharedNumber. Jika writeable bernilai 0, maka konsumer dapat membaca nilai dari variable sharedNumber, ini berarti saat ini producer tidak dapat menempatkan nilai pada sharedNumber.

Ketika objek thread ProduceInteger memanggil method **setSharedNumber** (baris 16-34), lock akan diacquire pada condition variable (baris 20). strukture while pada baris 23-24 akan melakukan test pada variable writeable. Jika writeable bernilai 0, baris 24 akan memanggil method method. Pemanggilan method ini ditempatkan pada thread ProduceInteger yang memanggil method setSharedNumber kedalam waiting state dan melepaskan lock pada SynchronizedInteger jadi objek yang lain bisa mengaksesnya.

Objek `ProduceInteger` akan kembali memasuki `waiting state` sampai diaktifkan kembali untuk melakukan proses, yang mana akan memasuki `redy state` dan menunggu sampai interpreter mengeksekusinya. Ketika objek `ProduceInteger` kembali memasuki `running state`, maka secara implisit akan melakukan `acquire` pada `condition variable`, dan method `setSharedNumber` akan kembali ke struktur `while` dan akan mengeksekusi statemen selanjutnya setelah `wait`. Setelah tidak ada statemen lagi untuk dieksekusi, jadi `while` statemen akan di test lagi. Jika kondisi bernilai `true` (`writable` bernilai 0), maka program akan menampilkan pesan yang mengindikasikan bahwa `producer` telah diseting dengan nilai yang baru yaitu `newNumber` (sebagai argumen yang dilewatkan pada `setSharedNumber`). `writable` diset dengan nilai 0 mengindikasikan bahwa `share memori` telah penuh (`consumer` dapat membaca tetapi `producer` tidak dapat merubah nilai tersebut) dan kemudian method `notify` dari `condition variable` akan dipanggil. Jika disana ada `waiting thread`, satu `thread` pada `waiting state` akan ditempatkan pada `ready state`, yang mengindikasikan bahwa `thread` dapat dijalankan lagi. Baris 34 kemudian memanggil method `release` pada `condition variable`.

Method `getSharedNumber` dan `setSharedNumber` implementasinya hampir sama. method melakukan `acquire lock` pada objek `condition variable`. `while` struktur pada baris 43-44 akan melakukan test pada variable `writable`. Jika `writable` bernilai 1 (belum dikonsumsi), method `wait` pada objek `condition variable` akan dipanggil pada `thread ConsumeInteger` yang memanggil `getSharedNumber` kedalam `waiting state` dan melepaskan `lock` pada objek `SynchronizedInteger` jadi objek yang lain bisa mengaksesnya. `ConsumeInteger` akan kembali memasuki `waiting state` sampai diaktifkan kembali dan memasuki `ready state` dan menunggu interpreter untuk mengeksekusinya. Ketika `ConsumeInteger` masuk kembali pada `running state`, method `setSharedNumber` akan melakukan `acquire lock` pada `condition variable` dan method akan melanjutkan untuk mengeksekusi statemen selanjutnya pada struktur `while` setelah `wait`. Jika tidak ada lagi statemen yang dieksekusi, maka `while` akan melakukan pengecekan kembali. Jika kondisi bernilai 0, nilai dari `sharedNumber` akan diinisialisasikan pada variable `tempNumber` (baris 48) dan method akan mengeluarkan pesan yang menandakan bahwa `consumer` telah mengkonsumsi nilai `sharedNumber`. Perlu dicatat nilai dari `shareNumber` hanya akan dikonsumsi sekali oleh `consumer` dan akan disimpan pada `tempNumber` (ketika berada pada `critical section`). Baris 49 dan 56 (dik luar `critical section`) menggunakan nilai dari `tempNumber` selain dari `sharedNumber` untuk memastikan nilai yang digunakan adalah nilai yang sama.

Selanjutnya, `writable` akan diset dengan nilai 1 untuk menandakan bahwa `share memory` telah kosong, dan method `notify` dari `condition variable` akan dipanggil. Jika ada `thread` yang menunggu, satu `thread` pada antrian `waiting state` akan di tempatkan pada `ready state`, yang menandakan bahwa `thread` dapat menjalankan `task` kembali. Baris 54 melepaskan `lock` pada `condition variable`, dan baris 56 akan mengembalikan nilai nilai dari `tempNumber` pada method

getSharedNumber.

Output program sinkron.py menunjukkan bahwa setiap integer yang diproduksi akan dikonsumsi tepat sekali, tidak ada nilai yang terlewat dan tidak ada nilai yang dobel. Consumer tidak akan bisa membaca sebelum producer memproduksi. Pada bahasan artikel selanjutnya akan dibahas cara untuk Producer/Consumer membaca dan mengubah banyak nilai secara simultan.

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.