Multithreading dengan python (bagian 3) – dengan sinkronisasi plus buffer

Amru Rosyada
amrupunya@gmail.com
http://amrurosyada.blogspot.com
http://technocratiz.wordpress.com

Pada artikel sebelumnya telah kita bahas tentang multithread asynchron dan synchron, sekarang kita akan membahas threading dengan menggunakan buffer. Fungsi dari buffer disini adalah sebagai shared data yang dapat menyimpan data dari producer sesuai dengan alokasi maksimal dari buffer, sehingga producer tidak perlu menunggu sampai consumer mengambil nilai yang ada pada shared memori untuk memproduksi kembali. Sebagai analogi producer memperluas gudang penyimpanan dari hasil produksinya, sehingga tiap kali producer mengghasilkan produk langsung dimasukkan ke gudang, hal ini akan mempermudah konsumer karena konsumer tinggal mengambil barang dari gudang. Pada kasus ini kita akan menggunakan buffer berupa list yang panjangnya lima data untuk menampung produk dari producer, kemudian consumer akan mengambil data dari list tersebut. Program threadbuffer.py di bawah sama persis dengan kode pada artikel sebelumnya yaitu sinkron.py.

Pendahuluan

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

Intinya adalah membuat prosess 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 Plus Buffer

Pada artikel sebelumnya telah kita bahas tentang multithread asinkron dan sinkron, sekarang kita akan membahas threading dengan menggunakan buffer. Fungsi dari buffer disini adalah sebagai shared data yang dapat menyimpan data dari producer sesuai dengan alokasi maksimal dari buffer, sehingga producer tidak perlu menunggu sampai consumer mengambil nilai yang ada pada shared memori untuk memproduksi kembali. Sebagai analogi producer memperluas gudang penyimpanan dari hasil produksinya, sehingga tiap kali producer mengghasilkan produk langsung dimasukkan ke gudang, hal ini akan mempermudah konsumer karena konsumer tinggal mengambil barang dari gudang. Pada kasus ini kita akan menggunakan buffer berupa list yang panjangnya lima data untuk menampung produk dari producer, kemudian consumer akan mengambil data dari list tersebut. Program ***threadbuffer.py*** di bawah sama persis dengan kode pada artikel sebelumnya yaitu ***sinkron.py***.

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

from SynchronizedCells import SynchronizedCells
from ProduceInteger import ProduceInteger
from ConsumeInteger import ConsumeInteger

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

(threadBuffer.py)

Class ***SynchronizedCells*** terdiri dari enam atribut yaitu ***sharedCells*** berupa lima elemen list yang merepresentasikan sebagai circular buffer, ***writeable*** sebagai flag penanda bahwa producer dapat memasukkan data kedalam buffer, ***readable*** sebagai flag penanda bahwa consumer dapat membaca data dari circular buffer, ***readLocation*** sebagai tanda lokasi pada buffer yang akan dibaca selanjutnya oleh consumer, ***writeLocation*** sebagai tanda lokasi selanjutnya tempat untuk menaruh nilai yang dihaslikan oleh consumer dan ***threadCondition*** adalah condition variable yang digunakan untuk melindungi akses ke buffer.

```
# SynchronizedCells.py
# Synchronized circular buffer of integer values

import threading

class SynchronizedCells:

    def __init__( self ):
    """Set cells, flags, locations and condition variable"""

        self.sharedCells = [ -1, -1, -1, -1, -1 ] # buffer
        self.writeable = 1          # buffer may be changed
        self.readable = 0           # buffer may not be read
        self.writeLocation = 0        # current writing index
        self.readLocation = 0         # current reading index

        self.threadCondition = threading.Condition()

    def setSharedNumber( self, newNumber ):
        """Set next buffer index value--blocks until lock acquired"""

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

        # while buffer is full, release lock and block
        while not self.writeable:
```

```
                print "WAITING TO PRODUCE", newNumber
                        self.threadCondition.wait()

                # buffer is not full, lock has been re-acquired

                # produce a number in shared cells, consumer may consume
                self.sharedCells[ self.writeLocation ] = newNumber
                self.readable = 1
                print "Produced %2d into cell %d" % \
                ( newNumber, self.writeLocation ),

                # set writing index to next place in buffer
                self.writeLocation = ( self.writeLocation + 1 ) % 5

                print "      write %d read %d " % \
                ( self.writeLocation, self.readLocation ),
                print self.sharedCells

                # if producer has caught up to consumer, buffer is full
                if self.writeLocation == self.readLocation:
                self.writeable = 0
                print "BUFFER FULL"

                self.threadCondition.notify() # wake up a waiting thread
                self.threadCondition.release() # allow lock to be acquired

        def getSharedNumber( self ):
                """Get next buffer index value--blocks until lock acquired"""

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

                # while buffer is empty, release lock and block
                while not self.readable:
                print "WAITING TO CONSUME"
                        self.threadCondition.wait()

                # buffer is not empty, lock has been re-acquired
```

```
# consume a number from shared cells, producer may produce
self.writeable = 1
tempNumber = self.sharedCells[ self.readLocation ]
self.sharedCells[ self.readLocation ] = -1

print "Consumed %2d from cell %d" % \
( tempNumber, self.readLocation ),

# move to next produced number
self.readLocation = ( self.readLocation + 1 ) % 5

print "   write %d read %d " % \
( self.writeLocation, self.readLocation ),
print self.sharedCells

# if consumer has caught up to producer, buffer is empty
if self.readLocation == self.writeLocation:
self.readable = 0
print "BUFFER EMPTY"

self.threadCondition.notify() # wake up a waiting thread
self.threadCondition.release() # allow lock to be acquired

return tempNumber
```

(SynchronizedCells.py)

Method ***setSharedNumber*** sama seperti pada class ***Synchronized*** pada artikel sebelumnya dengan sedikit modifikasi. Ketika eksekusi berlanjut setelah ***while*** loop, nilai yang dihasilkan producer akan ditempatkan di ***writeLocation***. Selanjutnya, ***readable*** diset bernilai 1 karena tidak lebih dari satu nilai yang tersimpan di buffer untuk dibaca. Kemudian ***writeLocation*** akan diupdate untuk selanjutnya memanggil ***setSharedNumber***. Sebagai catatan nilai dari ***writeLocation*** akan bernilai antara 0-4. Jika ***writeLocation*** bernilai sama dengan ***readLocation***, berarti buffer dalam kedaan penuh, jadi writeable akan diset bernilai 0 dan string "BUFFER FULL" akan ditampilkan oleh program. Selanjutnya, method ***notify*** dari condition variable akan dipanggil untuk menandakan bahwa thread yang menunggu seharusnya dipindahkan ke ***ready*** state. Akhirnya, method ***release*** pada condition variable akan dipanggil untuk melepaskan

condition variable yang berada pada kondisi lock.

Method ***getSharedNumber*** juga sama seperti pada class ***synchronized*** tetapi dengan sedikit modifikasi. Ketika eksekusi berlangsung setelah ***while*** loop, ***writeable*** akan diset bernilai satu karena kurang lebih akan terdapat tempat yang kosong di buffer. Selanjutnya, kemudian nilai dari ***readLocation*** pada circular buffer akan diisikan ke ***tempNumber***. untuk menandai tempat yang kosong maka ***readLocation*** akan diberi nilai -1. Kemudian atribut ***readLocation*** akan diupdate untuk selanjutnya memanggil ***getSharedNumber***. Jika ***readLocation*** bernilai sama dengan ***writeLocation*** menandakan bahwa bahwa circular buffer kosong, jadi ***readable*** akan diset bernilai 0 dan string "BUFFER EMPTY" akan ditampilakn. Selanjutnya, memanggil method ***notify*** pada condition variable untuk menempatkan thread yang menunggu kedalam ***ready*** state. Kemudian akan memanggil method ***release*** dari condition variable untuk melepaskan lock pada condition variable. Yang terakhir adalah mengembalikan nilai ***tempNumber***.

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.