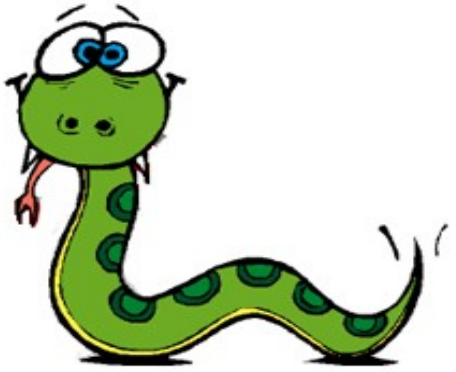
python

www.python.org



Pemrosesan parallel

Background

- Sebagian besar komputer menghabiskan banyak waktu untuk tidak melakukan apapun. Jika Anda mengaktifkan alat monitor sistem dan melihat utilisasi CPU, Anda akan melihat jarang sekali terjadi hit 100 persen, bahkan saat Anda menjalankan beberapa program.
- Ada terlalu banyak penundaan yang dibangun ke dalam perangkat lunak: akses disk, lalu lintas jaringan, query database, menunggu pengguna mengklik tombol, dan seterusnya.
- Faktanya, sebagian besar kapasitas CPU modern sering dihabiskan dalam keadaan menganggur; Chip yang lebih cepat membantu mempercepat kinerja pada puncak permintaan, namun daya mereka sebagian besar tidak digunakan.

Background

- Pada awal komputasi, programer menyadari bahwa mereka dapat memanfaatkan kekuatan pemrosesan yang tidak terpakai tersebut dengan menjalankan lebih dari satu program pada saat bersamaan. Teknik ini biasanya disebut **parallel processing** (dan kadang disebut "multiprocessing" atau bahkan "multitasking") karena banyak tugas yang tampaknya dilakukan sekaligus, tumpang tindih dan paralel pada waktu yang sama. Ini adalah inti dari sistem operasi modern.
- Dalam suatu program, membagi pemrosesan menjadi tugas yang berjalan secara paralel dapat membuat keseluruhan sistem lebih cepat.

Background

- Ada dua cara mendasar untuk mendapatkan tugas (task) yang berjalan bersamaan pada Python, yaitu:
  - process forks
  - spawned threads
- Secara fungsional, keduanya mengandalkan layanan sistem operasi yang mendasar untuk menjalankan bit kode Python secara paralel. Secara prosedural, keduanya sangat berbeda dalam hal interface, portabilitas, dan komunikasi.

Proses forking

- Proses forking adalah cara tradisional untuk menyusun tugas paralel, dan ini adalah bagian mendasar dari sistem Unix. Forking adalah cara mudah untuk memulai program mandiri.
- Forking didasarkan pada gagasan tentang program penyalinan: saat sebuah program memanggil rutin forking, sistem operasi membuat salinan baru dari program tersebut dan memprosesnya di memori dan mulai menjalankan salinan tersebut secara paralel dengan yang asli.

Proses forking

- Setelah operasi forking, salinan asli dari program ini disebut proses induk, dan salinan yang dibuat oleh **os.fork** disebut proses anak.
- Secara umum, induk dapat membuat sejumlah anak, dan anak-anak dapat menciptakan proses anaknya sendiri; semua proses forking berjalan secara independen dan paralel di bawah kendali sistem operasi, dan anak-anak dapat terus berjalan setelah induk mereka keluar (berakhir)

Contoh proses forking

```
"forks child processes until you type 'q'"
import os
def child():
      print('Hello from child', os.getpid())
      os._exit(0) # else goes back to parent loop
def parent():
   while True:
      newpid = os.fork()
      # newpid = 0 , we are in child process
      # newpid = positive value, we are in parent process
      # newpid = negative value, error occured
      if newpid == 0:
        child()
      else:
        print('Hello from parent', os.getpid(), newpid)
      if raw_input() == 'q':
        break
parent()
```

Contoh proses forking

```
import os, time
def counter(count):
    for i in range(count):
        time.sleep(1)
        print('[%s] => %s' % (os.getpid(), i))
# run in new process
# simulate real work
for i in range(5):
     pid = os.fork()
     if pid != 0:
        print('Process %d spawned' % pid)
     else:
        counter(5)
        os._exit(0) # in parent: continue
print('Main process exiting.')
```

Kombinasi fork/exec

- Dalam dua contoh sebelumnya, proses anak hanya menjalankan fungsi dalam program Python dan kemudian keluar.
- Pada platform mirip-Unix, forking sering menjadi dasar untuk memulai program yang dijalankan secara independen yang benar-benar berbeda dari program yang melakukan panggilan fork. Misalnya, Contoh berikut menghasilkan proses baru sampai kita mengetik q lagi, namun proses anak menjalankan program baru daripada memanggil fungsi dalam file yang sama.

Contoh kombinasi fork/exec

```
"starts programs until you type 'q'"
import os
parm = 0
while True:
    parm += 1
    pid = os.fork()
    if pid == 0:
        # copy process
        os.execlp('python', 'python', 'child.py', str(parm))
        assert False, 'error starting program'
    else:
        print('Child is', pid)
    if raw_input() == 'q':
        break
```

Contoh kombinasi fork/exec

```
#child.py

import os, sys

print('Hello from child', os.getpid(),
sys.argv[1])
```

Threads

- Thread adalah cara lain untuk memulai task (tugas) yang berjalan pada saat bersamaan. Singkatnya, mereka menjalankan panggilan ke fungsi (atau jenis objek panggilan lainnya) bersamaan dengan sisa program.
- Thread terkadang disebut "proses ringan," karena berjalan paralel seperti proses forking, namun semuanya berjalan dalam proses tunggal yang sama.
- Sementara proses biasanya digunakan untuk memulai program independen, threads biasanya digunakan untuk tugas seperti nonblocking input calls dan tugas yang berjalan lama di GUI. Mereka juga menyediakan model alami untuk algoritma yang dapat dinyatakan sebagai tugas yang berjalan secara independen.

Threads

- Untuk aplikasi yang bisa mendapatkan keuntungan dari pemrosesan paralel, beberapa pengembang mempertimbangkan thread untuk menawarkan sejumlah keuntungan:
  - Performance
  - Simplicity
  - Shared global memory
  - Portability

Threads

- Pertama, thread setidaknya bukanlah suatu cara langsung-untuk memulai program lain. Sebaliknya, thread dirancang untuk menjalankan panggilan ke fungsi (secara teknis, apapun, termasuk metode terikat dan tidak terikat) bersamaan dengan sisa program. Seperti yang kita lihat di bagian sebelumnya, sebaliknya, proses forking dapat memanggil fungsi atau memulai program baru.

Threads

- Kedua, fakta bahwa thread berbagi objek dan nama dalam memori proses global adalah kabar baik dan kabar buruk-ini menyediakan mekanisme komunikasi, namun kita harus berhati-hati untuk menyinkronkan berbagai operasi. Seperti yang akan kita lihat, bahkan operasi seperti percetakan adalah konflik potensial karena hanya ada satu **sys.stdout** per proses, yang dimiliki oleh semua thread

Modul _thread

```
"spawn threads until you type 'q'"
import _thread
def child(tid):
     print('Hello from thread', tid)
def parent():
     i = 0
     while True:
          i += 1
          _thread.start_new_thread(child, (i,))
          if raw_input() == 'q':
               break
parent()
```

Contohe thread

```
import _thread as thread, time
def counter(myId, count):
      for i in range(count):
            time.sleep(1)
            print('[%s] => %s' % (myId, i)) #
function run in threads
for i in range(5):
       thread.start_new_thread(counter, (i, 5)) #
spawn 5 threads
       # each thread loops 5 times
       time.sleep(5)
print('Main thread exiting.')
```

Modul threading

- Meskipun sangat efektif untuk threading tingkat rendah, namun modul thread dan _thread sangat terbatas dibandingkan dengan modul **threading** yang baru.
- Modul threading yang lebih baru disertakan dalam Python 2.4 yang memberikan dukungan kuat dan tingkat tinggi yang jauh lebih kuat untuk thread daripada modul thread yang dibahas di bagian sebelumnya.

Modul threading

- Modul threading mengekspose semua metode yang ada pada modul thread (_thread) dan menyediakan beberapa metode tambahan, seperti:
  - **threading.activeCount()** − Mengembalikan jumlah objek thread yang aktif.
  - **threading.currentThread()** - Mengembalikan jumlah objek thread pada kontrol thread pemanggil
  - **threading.enumerate()** − Mengembalikan daftar semua objek thread yang sedang aktif.

Modul threading

- Selain metode, modul threading memiliki kelas Thread yang mengimplementasikan threading. Metode yang diberikan oleh kelas Thread adalah sebagai berikut :
  - **run()** - Metode run() adalah titik masuk untuk sebuah thread.
  - **start()** - Metode start() memulai thread dengan memanggil metode run.
  - **join([time])** - join() menunggu thread untuk mengakhiri.
  - **isAlive()** - Metode isAlive() memeriksa apakah sebuah thread masih dijalankan.
  - **getName()** - Metode getName() mengembalikan nama thread.
  - **setName()** - Metode setName() menetapkan nama sebuah thread.

Membuat Thread Menggunakan Modul Threading

- Untuk menerapkan thread baru menggunakan modul threading, Anda harus melakukan hal berikut :
  - Buat subkelas baru dari kelas Thread.
  - Timpa (override) metode __**init** __ **(self [, args])** untuk menambahkan argumen tambahan.
  - Kemudian, timpa (override) metode **run(self [, args])** untuk mengimplementasikan apa yang seharusnya dilakukan thread saat dimulai.
- Setelah Anda membuat subclass Thread baru, Anda dapat membuat sebuah instance dari subclass tersebut dan kemudian memulai thread baru dengan memanggil start(), yang pada gilirannya memanggil metode run().

Contoh thread dengan modul Threading

```
import threading
import time
class myThread (threading.Thread):
   def __init__(self,threadName,counter):
      threading.Thread.__init__(self)
      self.name = threadName
      self.counter = counter
   def run(self):
      while self.counter >= 1:
        print("Starting %s - counter : %d "%(self.name,self.counter))
        self.counter -= 1
        time.sleep(1)
      print("Exiting %s"%(self.name))
# Create new threads
thread1 = myThread("Thread-1", 5)
thread2 = myThread("Thread-2", 4)
# Start new Threads
thread1.start()
thread2.start()
print "Exiting Main Thread"
```

Sinkronisasi Thread

- Modul threading yang disertakan dalam Python mencakup mekanisme penguncian sederhana yang dapat digunakan untuk menyinkronkan thread. Kunci baru dibuat dengan memanggil metode **Lock()**, yang mengembalikan kunci baru.
- Metode **acquire(blocking)** dari objek kunci baru digunakan untuk memaksa thread untuk berjalan serentak. Parameter pemblokiran opsional memungkinkan Anda mengontrol apakah thread menunggu untuk mendapatkan kunci.
- Jika pemblokiran disetel ke 0, thread dengan segera mengembalikan nilai 0 jika kunci tidak dapat diperoleh dan dengan 1 jika kunci diperoleh. Jika pemblokiran diatur ke 1, thread melakukan block dan menunggu kunci dilepaskan.
- Metode **release()** dari objek kunci baru digunakan untuk melepaskan kunci saat tidak diperlukan lagi.

Contoh Sinkronisasi Thread (1)

```
import threading
class myThread (threading.Thread):
   def __init__(self,threadName,counter):
      threading.Thread.__init__(self)
      self.name = threadName
      self.counter = counter
   def run(self):
      # Get lock to synchronize threads
      threadLock.acquire()
      while self.counter >= 1:
        print("Starting %s - counter : %d "%(self.name,self.counter))
        # Free lock to release next thread
        self.counter -= 1
      threadLock.release()
      print("Exiting %s"%(self.name))
# Create new threads
threadLock = threading.Lock()
```

Contoh Sinkronisasi Thread (2)

```
threads = []
thread1 = myThread("Thread-1", 5)
thread2 = myThread("Thread-2", 4)
# Start new Threads
thread1.start()
thread2.start()
# Add threads to thread list
threads.append(thread1)
threads.append(thread2)
# Wait for all threads to complete
for t in threads:
    t.join()
print "Exiting Main Thread"
```