Источник: http://python-3.ru/page/multiprocessing

Модуль **multiprocessing** был добавлен в Python версии 2.6. Изначально он был определен в PEP 371 Джесси Ноллером и Ричардом Одкерком. Модуль [multiprocessing](http://python-3.ru/page/multiprocessing) позволяет вам создавать процессы таким же образом, как при создании потоков при помощи модуля [threading](http://python-3.ru/page/import-threading). Суть в том, что, в связи с тем, что мы теперь создаем процессы, вы можете обойти GIL (Global Interpreter Lock) и воспользоваться возможностью использования нескольких процессоров на компьютере. Пакет **multiprocessing** также включает ряд API, которых вообще нет в модуле threading. Например, есть очень удобный **класс Pool**, который вы можете использовать для параллельного выполнения функции между несколькими входами. Мы рассмотрим Pool немного позже. Мы начнем с класса **Process** модуля multiprocessing.

# Приступим к работе с Multiprocessing

Класс **Process** очень похож на класс Thread модуля threading. Давайте попробуем создать несколько процессов, которые вызывают одну и ту же функцию, и посмотрим, как это сработает:

import os

from multiprocessing import Process

def doubler(number):

"""

Функция умножитель на два

"""

result = number \* 2

proc = os.getpid()

print('{0} doubled to {1} by process id: {2}'.format(

number, result, proc))

if \_\_name\_\_ == '\_\_main\_\_':

numbers = [5, 10, 15, 20, 25]

procs = []

for index, number in enumerate(numbers):

proc = Process(target=doubler, args=(number,))

procs.append(proc)

proc.start()

for proc in procs:

proc.join()

Для этого примера мы импортируем **Process** и создаем функцию doubler. Внутри функции, мы дублируем число, которое мы ей передали. Мы также используем [модуль os](https://python-scripts.com/import-os-example), чтобы получить **ID нынешнего процесса**. Это скажет нам, какой именно процесс вызывает функцию. Далее, в нижнем блоке кода, мы создаем несколько Процессов и начинаем их.

Самый последний цикл только вызывает метод **join**() для каждого из процессов, что говорит Python подождать, пока процесс завершится. Если вам нужно остановить процесс, вы можете вызвать метод **terminate**(). Когда вы запустите этот код, вы получите выдачу, на подобие этой:

5 doubled to 10 by process id: 10468

10 doubled to 20 by process id: 10469

15 doubled to 30 by process id: 10470

20 doubled to 40 by process id: 10471

25 doubled to 50 by process id: 10472

Все же иногда приятно иметь читабельное название процессов. К счастью, класс **Process** дает возможность вам получить доступ к названию вашего процесса. Давайте посмотрим:

import os

from multiprocessing import Process, current\_process

def doubler(number):

result = number \* 2

proc\_name = current\_process().name

print('{0} doubled to {1} by: {2}'.format(

number, result, proc\_name))

if \_\_name\_\_ == '\_\_main\_\_':

numbers = [5, 10, 15, 20, 25]

procs = []

proc = Process(target=doubler, args=(5,))

for index, number in enumerate(numbers):

proc = Process(target=doubler, args=(number,))

procs.append(proc)

proc.start()

proc = Process(target=doubler, name='Test', args=(2,))

proc.start()

procs.append(proc)

for proc in procs:

proc.join()

На этот раз мы импортируем кое-что дополнительно: **current\_process**. Это примерно то же самое, что и **current\_thread**модуля [threading](http://python-3.ru/page/import-threading). Мы используем его для того, чтобы получить имя потока, который вызывает нашу функцию. Обратите внимание на то, что мы не указывали название первых пяти процессов. И только шестой мы назвали Test. Давайте посмотрим, какую выдачу мы получим:

5 doubled to 10 by: Process-2

10 doubled to 20 by: Process-3

15 doubled to 30 by: Process-4

20 doubled to 40 by: Process-5

25 doubled to 50 by: Process-6

2 doubled to 4 by: Test

Выдача показывает, что [модуль multiprocessing](http://python-3.ru/page/multiprocessing) назначает номер каждому процессу, как часть его названия по умолчанию. Конечно, когда мы лично определяем название, модуль не будет добавлять число к нашему названию

# Замки (Locks)

Модуль **multiprocessing** поддерживает замки так же, как и модуль threading. Все что вам нужно, это импортировать Lock, повесить его, сделать что-нибудь и снять его. Давайте посмотрим:

from multiprocessing import Process, Lock

def printer(item, lock):

"""

Выводим то что передали

"""

lock.acquire()

try:

print(item)

finally:

lock.release()

if \_\_name\_\_ == '\_\_main\_\_':

lock = Lock()

items = ['tango', 'foxtrot', 10]

for item in items:

p = Process(target=printer, args=(item, lock))

p.start()

Здесь мы создали простую функцию вывода, которая выводит все, что вы ей передаете. Чтобы не дать процессам **конфликтовать друг с другом**, мы используем объект **Lock**. Этот код зациклится над нашим списком трех объектов и создаст процесс для каждого из них. Каждый процесс будет вызывать нашу функцию, и передавать её одному из объектов. Так как мы используем замки, следующий процесс в строке **будет ждать, пока замок не снимается**, после чего он сможет продолжить.

# Логирование (Logging)

Логирование процессов немного отличается от **логирования потоков**. Причина в том, что пакет logging не использует замки, предназначенные для процессов, так что в итоге вы можете получить результат, который состоит из кучи перемешанных между собой процессов. Давайте попробуем добавить базовый логгинг к предыдущему примеру. Вот код:

import logging

import multiprocessing

from multiprocessing import Process, Lock

def printer(item, lock):

"""

Выводим то что передали

"""

lock.acquire()

try:

print(item)

finally:

lock.release()

if \_\_name\_\_ == '\_\_main\_\_':

lock = Lock()

items = ['tango', 'foxtrot', 10]

multiprocessing.log\_to\_stderr()

logger = multiprocessing.get\_logger()

logger.setLevel(logging.INFO)

for item in items:

p = Process(target=printer, args=(item, lock))

p.start()

Простейший способ вести журнал, это отправить все на **stderr**. Мы можем сделать это, вызвав функцию **log\_to\_stderr**(). Далее мы вызываем функцию get\_logger для получения доступа к логгеру и настраиваем его [уровень логгинга на INFO](https://python-scripts.com/logging-python). Остальная часть кода остается такой же, какой и была. Обратите внимание на то, что я не вызываю метод **join**() здесь. Вместо этого, поток parent (другими словами, ваш скрипт) вызовет join() лично. Когда вы сделаете это, вы получите что-то на подобие:

[INFO/Process-1] child process calling self.run()

tango

[INFO/Process-1] process shutting down

[INFO/Process-1] process exiting with exitcode 0

[INFO/Process-2] child process calling self.run()

[INFO/MainProcess] process shutting down

foxtrot

[INFO/Process-2] process shutting down

[INFO/Process-3] child process calling self.run()

[INFO/Process-2] process exiting with exitcode 0

10

[INFO/MainProcess] calling join() for process Process-3

[INFO/Process-3] process shutting down

[INFO/Process-3] process exiting with exitcode 0

[INFO/MainProcess] calling join() for process Process-2

Давайте пойдем дальше, и рассмотрим **класс Pool** поближе

# Класс Pool

Класс Pool используется для показа пула рабочих процессов. Он включает в себя методы, которые позволяют вам разгружать задачи к рабочим процессам. Давайте посмотрим на простейший пример:

from multiprocessing import Pool

def doubler(number):

return number \* 2

if \_\_name\_\_ == '\_\_main\_\_':

numbers = [5, 10, 20]

pool = Pool(processes=3)

print(pool.map(doubler, numbers))

Здесь мы создали экземпляр **Pool** и указали ему создать три рабочих процесса. Далее мы используем метод **map** для отображения функции для каждого процесса. Наконец мы выводим результат, что в нашем случае является списком: [10, 20, 40]. Вы также можете получить результат вашего процесса в пуле, используя метод **apply\_async**:

from multiprocessing import Pool

def doubler(number):

return number \* 2

if \_\_name\_\_ == '\_\_main\_\_':

pool = Pool(processes=3)

result = pool.apply\_async(doubler, (25,))

print(result.get(timeout=1))

Так мы можем запросить результат процесса. В этом суть работы функции **get**. Она пытается получить наши результаты. Обратите внимание на то, что мы также настроили обратный отсчет, на тот случай, если что-нибудь произойдет с вызываемой нами функцией. Мы не хотим, чтобы она была заблокирована.

# Связь между процессами

Когда речь заходит о **связи между процессами**, модули нашего multiprocessing включают в себя два главных метода: [Queue](https://python-scripts.com/queues) и **Pipe**. Работа Queue защищена как от процессов, так и от потоков. Давайте взглянем на достаточно простой пример:

from multiprocessing import Process, Queue

sentinel = -1

def creator(data, q):

"""

Creates data to be consumed and waits for the consumer

to finish processing

"""

print('Creating data and putting it on the queue')

for item in data:

q.put(item)

def my\_consumer(q):

"""

Consumes some data and works on it

In this case, all it does is double the input

"""

while True:

data = q.get()

print('data found to be processed: {}'.format(data))

processed = data \* 2

print(processed)

if data is sentinel:

break

if \_\_name\_\_ == '\_\_main\_\_':

q = Queue()

data = [5, 10, 13, -1]

process\_one = Process(target=creator, args=(data, q))

process\_two = Process(target=my\_consumer, args=(q,))

process\_one.start()

process\_two.start()

q.close()

q.join\_thread()

process\_one.join()

process\_two.join()

Здесь нам только и нужно, что импортировать **Process** и **Queue**. Далее мы создаем две функции, одна для создания данных и добавления их в очередь, и вторая для использования данных и обработки их. Добавление данных в **Queue** выполняется при помощи метода **put**(), в то время как получение данных из Queue выполняется через метод get. Последний кусок кода только создает объект Queue и несколько экземпляров **Process**, после чего возвращает их. Обратите внимание на то, что мы вызываем join() в наших объектах process больше, чем Queue.

# Подведем итоги

Здесь мы прошли через достаточно большое количество материала. Вы узнали много чего нового о модуле **multiprocessing** для направления обычных функций, связи между процессами при помощи Queue, наименований потоков и многого другого. Разумеется, в документации Python предоставлено намного больше развернутой информации, которую я даже не начинал затрагивать в данной статье, так что настоятельно рекомендую с ней ознакомиться. Тем не менее, вы все-таки узнали много чего о том, как усилить мощность обработки вашего компьютера при помощи Python!

Tонкости использования языка Python: Часть 5. Мульти-платформенные многопоточные приложения

<Источник: https://www.ibm.com/developerworks/ru/library/l-python_details_05/index.html>

Решение, показанное в предыдущей статье и **использующее вызов**fork() из импортированного модуля os, обладает многими преимуществами, но и существенным недостатком. Этой возможностью нельзя воспользоваться в операционных системах семейства Microsoft Windows, так как в ОС Windows никогда не было вызова fork(), и разработчики модуля os для этого вызова оставили только заглушку. Но из-за этого ограничения теряется одно из главных достоинств Python — переносимость проектов и их независимость от платформы.

Поэтому создатели Python предоставляют для реализации параллельных процессов другой модуль –multiprocessing, использующий особые приёмы для клонирования процессов, обсуждению которого, в частности, и будет посвящена данная статья.

Мульти-платформенный код

В документации говорится, что из-за отсутствия вызова fork() в ОС Windows он эмулируется путём создания нового процесса для исполнения кода, который в OC Linux выполнялся бы в дочернем процессе. Так как исполняемый код технически не связан с процессом, то он должен быть помещён в процесс перед запуском. Для этого код форматируется и передаётся по каналу из оригинального процесса во вновь созданный. Также новый процесс получает инструкцию запустить код, полученный по каналу, через переданный аргумент командной строки --multiprocessing-fork. Если посмотреть на реализацию метода freeze\_support(), то его задачей является проверка того, должен ли исполняемый процесс запускать код, полученный по каналу, или нет.

При подобном подходе запуск параллельных процессов может производиться, как показано в листинге 1. Полный код можно найти в файле **child.py** в архиве **python\_parallel.tgz** в разделе "Материалы для скачивания".

Листинг 1. Мульти-платформенный код для запуска нескольких дочерних процессов

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | #!/usr/bin/python3 -O  # -\*- coding: utf-8 -\*-  import time  import sys  import os  from multiprocessing import Process, freeze\_support    def info( title ):      if hasattr( os, 'getppid' ):  # only available on Unix          print( '{0}:\tPID={1} PPID={2}'.format( title, os.getpid(), os.getppid() ) )      else:          print( '{0}:\tPID={1}'.format( title, os.getpid() ) )    def fun( name ):      info( 'порождённый процесс' )      print( 'процесс {0} выполняет функцию с параметром {1}'.format( os.getpid(), name ) )      time.sleep( 0.5 )    if \_\_name\_\_ == '\_\_main\_\_':      freeze\_support()      nproc = len( sys.argv ) > 1 and int( sys.argv[ 1 ] ) or 3      print( 'число дочерних процессов ', nproc )      info( 'родительский процесс' )      procs = []      for i in range( nproc ):          procs.append( Process( target = fun, args = ( i, ) ) )      for i in range( nproc ):          procs[ i ].start()      for i in range( nproc ):          procs[ i ].join()      print( 'завершается родительский процесс' ) |

Как уже объяснялось выше, в Windows в качестве кода процесса используется уже компилированный байт-код приложения, поэтому использование конструкции: if \_\_name\_\_ == '\_\_main\_\_' - становится **обязательным**! Без этого фрагмента код порождённого дочернего процесса начнёт снова выполнять код главной ветви приложения, что породит бесконечную рекурсию из-за "размножения" процессов. Использование этой конструкции в операционных системах, реализующих вызов fork() не обязательно, но приветствуется, так как такой код становится независимым от платформы исполнения:

Попробуем запустить данное приложение в OC Windows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | $ python child.py  число дочерних процессов  3  родительский процесс:   PID=14562 PPID=2089  порождённый процесс:    PID=14563 PPID=14562  процесс 14563 выполняет функцию с параметром 0  порождённый процесс:    PID=14564 PPID=14562  процесс 14564 выполняет функцию с параметром 1  порождённый процесс:    PID=14565 PPID=14562  процесс 14565 выполняет функцию с параметром 2  завершается родительский процесс |

Новый процесс создаётся конструктором класса Process(), а целевым кодом для него указывается функция (target=...), как это имеет место при создании потока, после чего процесс должен быть запущен вызовом метода start().

Модуль multiprocessing предоставляет различные механизмы для взаимодействия созданных процессов, например:

* механизмы взаимодействия IPC: Queue, Pipe;
* механизмы взаимодействия через разделяемую процессами память Value, Array;
* специфичные механизмы, такие как Manager и Pool — пул потоков;

Примеры кода, использующие эти механизмы, включены в архив **python\_parallel.tgz** (файлы: **ipc.py**, **mgr.py**, **pool.py**), но мы не будем подробно разбирать их.

Многопроцессорное выполнение

Параллельные ветви исполнения (потоков или процессов) в коде программы могут применяться для различных целей:

1. квазипараллельный код (попеременно переключающийся с одной ветви на другую) в прикладных системах, где логика системы описывается естественным образом в терминах параллелизма (например, это задачи "производитель-потребитель");
2. параллельное совмещение ветвей кода, имеющих различный характер загрузки процессора: активный ввод-вывод в сочетании с большой вычислительной нагрузкой;
3. распараллеливание процессорной нагрузки между несколькими процессорами в многопроцессорных SMP системах.

Ещё не так давно последняя категория приложений была скорее экзотикой, чем практикой. Но за это время произошло массовое внедрение многоядерных процессоров и процессоров с **гиперпоточностью** (hyper-threading), и сегодня рядовой офисный компьютер, с большой вероятностью, является многопроцессорным.

В листинге 2 представлен простой пример для динамического определения числа процессоров в системе. Этот пример можно найти в файле **num\_proc.py** в архиве **python\_parallel.tgz**:

Листинг 2. Диагностика числа процессоров

|  |  |
| --- | --- |
| 1  2  3  4 | #!/usr/bin/python -O  # -\*- coding: utf-8 -\*-  from multiprocessing import cpu\_count  print( 'число процессоров = {}'.format( cpu\_count() ) ) |

Пример запуска этого сценария:

|  |  |
| --- | --- |
| 1  2 | $ python3 num\_proc.py  число процессоров = 2 |

Известно, что модель потоков, принятая в Python, **непригодна** к многопроцессорному выполнению. Это связано с блокировкой GIL, с которой мы уже встречались, а наиболее детально этот вопрос анализируется в известной статье Дэвида Бизли (см. раздел "Ресурсы"):

Принцип работы прост. Потоки удерживают GIL, пока выполняются. Однако они освобождают его при блокировании для операций ввода-вывода. Каждый раз, когда поток вынужден ждать, другие, готовые к выполнению, потоки используют свой шанс запуститься. ... При работе с CPU-зависимыми потоками, которые никогда не производят операции ввода-вывода, интерпретатор периодически проводит проверку. Интервал проверки — глобальный счетчик, абсолютно независимый от порядка переключения потоков. ... Ожидающий поток при этом может сделать сотни безуспешных попыток захватить GIL. Мы видим, что происходит битва за две взаимоисключающие цели. Python просто хочет запускать не больше одного потока в один момент. А операционная система щедро переключает потоки, пытаясь извлечь максимальную выгоду из всех ядер.

Подтверждение этого утверждения и его последствия можно увидеть на примере листинга 3. В этом листинге содержится файл **mthrs.py** из архива **python\_parallel.tgz**:

Листинг 3. Сравнение способов параллельного исполнения

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86 | #!/usr/bin/python  # -\*- coding: utf-8 -\*-    import time  import sys  import getopt  import threading  import os  import multiprocessing    def ncount( n ) : # тестовая CPU-загружающая функция      while n > 0 : n -= 1    if \_\_name\_\_ == '\_\_main\_\_':      repnum = 10000000      thrnum = 2      mode = 'stpm' # варианты запуска        try :          opts, args = getopt.getopt( sys.argv[1:], "t:n:m:" )      except getopt.GetoptError :          print ( "недопустимая опция команды или её значение" )        for opt, arg in opts :          if opt[ 1: ] == 't' : thrnum = int( arg )          if opt[ 1: ] == 'n' : repnum = int( arg )          if opt[ 1: ] == 'm' : mode = arg        print( "число процессоров (ядер) = {0:d}".format( multiprocessing.cpu\_count() ) )      print( "исполнение в Python версия {0:s}".format( sys.version ) )      print( "число ветвей выполнения {0:d}".format( thrnum ) )      print( "число циклов в ветви {0:d}".format( repnum ) )        if 's' in mode :          print( "============ последовательное выполнение ============" )          clc = time.time()          for i in range( thrnum ) : ncount( repnum )          clc = time.time() - clc          print( "время {0:.2f} секунд".format( clc ) )        if 't' in mode :          print( "================ параллельные потоки ================" )          threads = []          for n in range( thrnum ) :              tid = threading.Thread( target = ncount, args=( repnum, ) )              threads.append( tid )              tid.setDaemon( 1 )          clc = time.time()          for n in range( thrnum ) : threads[ n ].start()          for n in range( thrnum ) : threads[ n ].join()          clc = time.time() - clc          print( "время {0:.2f} секунд".format( clc ) )        if 'p' in mode :          print( "=============== параллельные процессы ===============" )          threads = []; fork = True          clc = time.time()          for n in range( thrnum ) :              try : pid = os.fork();              except :                  print( "ошибка создания дочернего процесса" )                  fork = False                  break              else :                  if pid == 0 : # дочерний процесс                      ncount( repnum )                      sys.exit( 0 )                  if pid > 0 :  # родительский процесс                      threads.append( pid )          if fork :              for p in threads :                  pid, status = os.wait()              clc = time.time() - clc              print( "время {0:.2f} секунд".format( clc ) )        if 'm' in mode :          print( "=============== модуль multiprocessing ==============" )          parms = []          for n in range( thrnum ) :              parms.append( repnum )          multiprocessing.freeze\_support()          pool = multiprocessing.Pool( processes = thrnum, )          clc = time.time()          pool.map( ncount, parms )          clc = time.time() - clc          print( "время {0:.2f} секунд".format( clc ) ) |

Приложение тестирует время выполнения большого числа (опция -n) циклов п ростого декремента целочисленной переменной, выполняемого в несколько (опция -t) параллельных ветвей исполнения для 4-х вариантов выполнения этой нагрузки:

1. без ветвления, весь объём работы выполняется последовательно;
2. работа распределяется на **N потоков**;
3. работа распределяется на **N процессов**, разветвлённых fork();
4. работа распределяется на **N процессов**, разветвлённых с помощью API модуля multiprocessing;

Поскольку тестирование может занять весьма продолжительное время, то опцией запуска -m можно указать только тот режим тестирования из 4-х, который следует выполнять (соответственно, значения для -m будут 's', 't', 'p', 'm'). Для запуска приложения в определённых режимах используется подобная команда:

|  |  |
| --- | --- |
| 1  2 | $ python mthrs.py -n 5000000 -t 4 -m tm  ... |

Приложение единообразно выполняется как в Linux, так и в Windows, и под версиями Python 2 и 3. Теперь можно проанализировать все возможные варианты исполнения:

На платформе Linux для Python 2 мы получим:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | $ python mthrs.py  число процессоров (ядер) = 2  исполнение в Python версия 2.7.3 (default, Jul 24 2012, 10:05:39)  [GCC 4.7.0 20120507 (Red Hat 4.7.0-5)]  число ветвей выполнения 2  число циклов в ветви 10000000  ============ последовательное выполнение ============  время 2.89 секунд  ================ параллельные потоки ================  время 3.55 секунд  =============== параллельные процессы ===============  время 1.78 секунд  =============== модуль multiprocessing ==============  время 1.75 секунд |

Вот главный результат, из-за которого была написана статья Дэвида Бизли и который может привести в недоумение: выполнение нагрузки на 2-х процессорах в 2 **потока** в Python требует на 23% **больше** времени, чем, если ту же нагрузку просто выполнить последовательно, вообще не создавая никаких потоков! А время выполнения для 2-х независимых **процессов** составляет только 60%.

Также на платформе Linux, но уже для Python 3 мы получим:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | $ python3 mthrs.py  число процессоров (ядер) = 2  исполнение в Python версия 3.2.3 (default, Jun  8 2012, 05:37:15)  [GCC 4.7.0 20120507 (Red Hat 4.7.0-5)]  число ветвей выполнения 2  число циклов в ветви 10000000  ============ последовательное выполнение ============  время 6.57 секунд  ================ параллельные потоки ================  время 9.74 секунд  =============== параллельные процессы ===============  время 3.93 секунд  =============== модуль multiprocessing ==============  время 3.66 секунд |

В данном сценарии картина становится ещё радикальнее: при использовании потоков замедление увеличивается до 48%, а для параллельных процессов время исполнения сокращается до 55% от случая последовательного исполнения. Отметим также, что несмотря на исполнение кода на той же системе, что и в предыдущем случае, общее время выполнения того же объёма работы увеличилось более чем в 2 раза, по сравнению с Python 2. За гибкость новых синтаксических возможностей языка приходится расплачиваться увеличением времени, затрачиваемого интерпретатором на исполнение!

Перейдём к Windows XP и Python 2 (выполнение в Python-терминале IDLE):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win32  Type "copyright", "credits" or "license()" for more information.  >>> ================================ RESTART ================================  >>>  число процессоров (ядер) = 2  исполнение в Python версия 2.7.5 (default, May 15 2013, 22:43:36) ...  число ветвей выполнения 2  число циклов в ветви 10000000  ============ последовательное выполнение ============  время 1.19 секунд  ================ параллельные потоки ================  время 14.05 секунд  =============== параллельные процессы ===============  ошибка создания дочернего процесса  =============== модуль multiprocessing ==============  время 0.72 секунд  >>> |

Картина становится ещё хуже, так как выполнение в 2 **потока** занимает в 11.8 раз больше времени по сравнению с простым последовательным выполнением. Но модуль multiprocessing при использовании параллельных **процессов**, создаваемых под Windows, обеспечивает прирост производительности на те же 60%

Запуск в Windows XP для Python 3 (код выполняется на той же системе, но на этот раз уже Windows XP работает в виртуальной машине VirtualBox 4.2.6, приложение выполняется в IDLE):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) ...  Type "copyright", "credits" or "license()" for more information.  >>> ================================ RESTART ================================  >>>  число процессоров (ядер) = 2  исполнение в Python версия 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) ...  число ветвей выполнения 2  число циклов в ветви 10000000  ============ последовательное выполнение ============  время 2.30 секунд  ================ параллельные потоки ================  время 2.33 секунд  =============== параллельные процессы ===============  ошибка создания дочернего процесса  =============== модуль multiprocessing ==============  время 1.31 секунд  >>> |

Похоже, что в версии Python 3.3 (в Linux сценарий выполнялся в Python 3.2) произошли значительные улучшения в области управления потоками, так при использовании 2-х потоков наблюдалось замедление всего на 1.5%.

Отметим также скорость выполнения Python-приложений в Windows под VirtualBox. Можно сказать, что скорость исполнения кода в VirtualBox практически не уступает "родной" среде.

В конце запустим наш сценарий также на ОС Linux, но на достаточно старом дистрибутиве Ubuntu 10.4 и таком процессоре:

|  |  |
| --- | --- |
| 1  2  3  4  5 | $ cat /proc/cpuinfo | grep 'model name'  model name      : Intel(R) Atom(TM) CPU  330   @ 1.60GHz  model name      : Intel(R) Atom(TM) CPU  330   @ 1.60GHz  model name      : Intel(R) Atom(TM) CPU  330   @ 1.60GHz  model name      : Intel(R) Atom(TM) CPU  330   @ 1.60GHz |

Но, как известно, процессоров Atom с 4-мя ядрами не бывает, это 2 достаточно медленных ядра с hyper-threading. Выполняем приложение на подобной конфигурации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | $ python mthrs.py -t4  число процессоров (ядер) = 4  исполнение в Python версия 2.6.5 (r265:79063, Oct  1 2012, 22:07:21) [GCC 4.4.3]  число ветвей выполнения 4  число циклов в ветви 10000000  ============ последовательное выполнение ============  время 12.90 секунд  ================ параллельные потоки ================  время 19.14 секунд  =============== параллельные процессы ===============  время 4.59 секунд  =============== модуль multiprocessing ==============  время 4.58 секунд |

В данном случае мы наблюдаем уже знакомую картину: выполнение в 4 потока только замедляет работу (+48%), но вот выполнение в 4 процесса ускоряет её почти в 3 раза. Попробуем сделать выводы из полученных результатов.

Потоки или процессы?

Так, использование потоков Python в многопроцессорных конфигурациях просто не имеет смысла. Если не учесть это обстоятельство, то можно реализовать проект как многопоточный, рассчитывая ускорить выполнение, а в итоге замедлить его, и, возможно, очень существенно.

Означает ли это, что использование потоков Python нецелесообразно вообще? Нет, не означает. Потоки будут уместны для сценариев, когда параллельные ветви (или некоторые из них) достаточно часто переходят в блокированные состояния, например, ожидая результатов операций ввода-вывода.

В многопроцессорной конфигурации целесообразно будет использовать параллельные процессы. В этом случае в каждом процессе будет выполняться отдельная копия интерпретатора Python, и это может дать существенный выигрыш в производительности.

Параллельные процессы предпочтительнее создавать не средствами операционной системы (вызов fork()), а используя API модуля multiprocessing из стандартной библиотеки Python.

Означает ли это, что параллельные процессы всегда предпочтительнее потоков, и обладают такой же "лёгкостью" (в смысле скорости) как и потоки? Нет, не означает. Медлительность параллельных процессов будет проявляться в их взаимодействиях между собой, будь это использование механизмов IPC, или использование разделяемой (shared) памяти. Оба эти способа требуют значительных процессорных ресурсов, поскольку для обмена информацией приходится каждый раз преодолевать границы защищённых адресных пространств процессов, что невозможно без вмешательства кода супервизорного режима (ядра операционной системы) и вовлечения MMU (устройства управления памятью).

Данные выводы не зависят от используемой операционной системы. Хотя в примерах были показаны результаты для ОС Linux и Windows, но такие же эффекты наблюдаются и в MacOS. Численные значения могут увеличиваться / уменьшаться, но общие принципы останутся неизменными.

Заключение

В этой и предыдущей статьях были описаны способы организации параллельных вычислений, доступные в Python. Для каждого способа были представлены практические примеры кода, которые могут использоваться в качестве отправной точки при построении собственных приложений.

Также мы проанализировали производительность работы параллельных механизмов в различных сценариях. Как было показано, результаты, полученные на практике, могут существенно отличаться от ожидаемых, но все наблюдаемые эффекты имеют обоснованное объяснение.