Лабораторная работа №7. Многопоточность в Python

С появлением многоядерных процессоров стала общеупотребительной практика распространять нагрузку на все доступные ядра. Существует два основных подхода в распределении нагрузки: использование процессов и потоков. Использование нескольких процессов фактически означает использование нескольких программ, которые выполняются независимо друг от друга. Программно это решается с помощью системных вызовов exec и fork. Такой подход создает большие неудобства в управлении обмена данными между этими программами.
В качестве альтернативы существует другой подход – создание многопоточных программ. Обмен данными между потоками существенно упрощается. Но управление такими программами усложняется, и вся ответственность ложится на программиста.

1. Как работают процессы

В питоне есть стандартный модуль subprocess, который упрощает управление другими программами, передавая им опции командной строки и организуя обмен данными через каналы (pipe). Мы рассмотрим пример, в котором пользователь запускает программу из командной строки, которая в свою очередь запустит несколько дочерних программ. В данном примере два скрипта – parent.py и child.py. Запускается parent.py. Child.py выступает в роли аргумента command, который передается в запускаемый процесс. У этого процесса есть стандартный вход, куда мы передаем два аргумента – поисковое слово и имя файла. Мы запустим два экземпляра программы child.py, каждый экземпляр будет искать слово word в своем файле – это будут файлы исходников самих программ. Запись на стандартный вход осуществляет модуль subprocess. Каждый процесс пишет результат своего поиска в консоль. В главном процессе мы ждем, пока все child не закончат свою работу.
Код parent.py:

```
1
2     import os
3     import subprocess
4     import sys
5
6
7     child = os.path.join(os.path.dirname(__file__), "./child.py")
8     word  = 'word'
9     file = ['./parent.py','./child.py']
10
11    pipes = []
12    for i in range(0,2):
13      command = [sys.executable, child]
14      pipe = subprocess.Popen(command, stdin=subprocess.PIPE)
15      pipes.append(pipe)
16      pipe.stdin.write(word.encode("utf8") + b"\n")
17      pipe.stdin.write(file[i].encode("utf8") + b"\n")
18      pipe.stdin.close()
19
20    while pipes:
21        pipe = pipes.pop()
          pipe.wait()
```

Код child.py:

```
1
2    import sys
3
4    word = sys.stdin.readline().rstrip()
5    filename = sys.stdin.readline().rstrip()
6
7    try:
8      with open(filename, "rb") as fh:
9        while True:
10         current = fh.readline()
11         if not current:
12             break
13         if (word in current ):
14             print("find: {0} {1}".format(filename,word))
15   except :
16       pass
17
```

2. Как работают потоки в питоне

Если нужно, чтобы ваше приложение выполняло несколько задач в одно и то же время, то можете воспользоваться потоками (threads). Потоки позволяют приложениям выполнять в одно и то же время множество задач. Многопоточность (multi-threading) важна во множестве приложений, от примитивных серверов до современных сложных и ресурсоёмких игр.

Когда в одной программе работают несколько потоков, возникает проблема разграничения доступа потоков к общим данным. Предположим, что есть два потока, имеющих доступ к общему списку. Первый поток может делать итерацию по этому списку:

```
1   for x in L
```

а второй в этот момент начнет удалять значения из этого списка. Тут может произойти все что угодно: программа может упасть, или мы просто получим неверные данные.

Решением в этом случае является применение блокировки. При этом доступ к заблокированному списку будет иметь только один поток, второй будет ждать, пока блокировка не будет снята.

Применение блокировки порождает другую проблему – дедлок (deadlock) – мертвая блокировка. Пример дедлока: имеется два потока и два списка. Первый поток блокирует первый список, второй поток блокирует второй список. Первый поток изнутри первой блокировки пытается получить доступ к уже заблокированному второму списку, второй поток пытается проделать то же самое с первым списком. Получается неопределенная ситуация с бесконечным ожиданием. Эту ситуации легко описать, на практике все гораздо сложнее.

Вариантом решения проблемы дедлоков является политика определения очередности блокировок. Например, в предыдущем примере мы должны определить, что блокировка первого списка идет всегда первой, а уже потом идет блокировка второго списка.

Другая проблема с блокировками – в том, что несколько потоков могут одновременно ждать доступа к уже заблокированному ресурсу и при этом ничего не делать. Каждая питоновская программа всегда имеет главный управляющий поток.

Питоновская реализация многопоточности ограниченная. Интерпретатор питона использует внутренний глобальный блокировщик (GIL), который позволяет выполняться только одному потоку. Это сводит на нет преимущества многоядерной архитектуры процессоров. Для многопоточных приложений, которые работают в основном на дисковые операции чтения/записи, это не имеет

особого значения, а для приложений, которые делят процессорное время между потоками, это является серьезным ограничением.

3. Создание потока

Для создания потоков мы будем использовать стандартный модуль threading. Есть два варианта создания потоков:

вызов функции

```
1   threading.Thread()
```

вызов класса

```
1   threading.Thread
```

Следующий пример показывает, как к потоку приаттачить функцию через вызов функции:

```
1
2   import threading
3   import time
4   def clock(interval):
5       while True:
6           print("The time is %s" % time.ctime())
7           time.sleep(interval)
8   t = threading.Thread(target=clock, args=(15,))
9   t.daemon = True
10  t.start()
11
```

Пример на создание потока через вызов класса: в конструкторе обязательно нужно вызвать конструктор базового класса. Для запуска потока нужно выполнить метод `start()` объекта-потока, что приведет к выполнению действий в методе `run()`:

```
1
2    import threading
3    import time
4    class ClockThread(threading.Thread):
5        def __init__(self,interval):
6            threading.Thread.__init__(self)
7            self.daemon = True
8            self.interval = interval
9        def run(self):
10           while True:
11               print("The time is %s" % time.ctime())
12               time.sleep(self.interval)
13  t = ClockThread(15)
14  t.start()
15
```

Для управления потоками существуют методы:

`start()` – дает потоку жизнь.

`run()` –этот метод представляет действия, которые должны быть выполнены в потоке.

`join([timeout])` – поток, который вызывает этот метод, приостанавливается, ожидая завершения потока, чей метод вызван. Параметр `timeout` (число с плавающей точкой) позволяет указать время ожидания (в секундах), по истечении которого приостановленный поток продолжает свою работу независимо от завершения потока, чей метод `join` был вызван. Вызывать `join()`некоторого потока можно много раз. Поток не может вызвать метод `join()` самого себя. Также нельзя ожидать завершения еще не запущенного потока.

`getName()` – возвращает имя потока.

`setName(name)` – присваивает потоку имя `name`.

`isAlive()` – возвращает истину, если поток работает (метод `run()` уже вызван).

`isDaemon()` – возвращает истину, если поток имеет признак демона.

`setDaemon(daemonic)` – устанавливает признак `daemonic` того, что поток является демоном.
`activeCount()` – возвращает количество активных в настоящий момент экземпляров класса Thread. Фактически это `len(threading.enumerate())`.
`currentThread()` – возвращает текущий объект-поток, т.е. соответствующий потоку управления, который вызвал эту функцию.
`enumerate()` – возвращает список активных потоков.

Задание

Главный поток программы должен генерировать строки случайного содержания и помещать их в конец списка. Дочерний поток №1 должен выдавать текущее состояние списка на экран. Дочерний поток №2 пробуждается каждые пять секунд и сортирует список в лексикографическом порядке и сохраняет в файл.