Выполнением кода Python управляет виртуальная машина Python (называемая также главным циклом интерпретатора). Язык Python разработан таким способом, чтобы в этом главном цикле мог выполняться только один поток управления по аналогии с тем, как организовано совместное использование одного процессора несколькими процессами в системе. В памяти может находиться много программ, но в любой конкретный момент времени процессор занимает только одна из них. Аналогичным образом, при том что в интерпретаторе Python могут эксплуатироваться несколько потоков, в любой момент времени интерпретатором выполняется только один поток.

Для управления доступом к виртуальной машине Python применяется глобальная блокировка интерпретатора (global interpreter lock — GIL). Именно эта блокировка обеспечивает то, что выполняется один и только один поток. Виртуальная машина Python функционирует в многопоточной среде следующим образом.

1. Задание глобальной блокировки интерпретатора.
2. Переключение на поток для его выполнения.
3. Должно быть выполнено одно из следующего:
   а) заданное количество команд в байт-коде;
   б) проверка способности потока самостоятельно возвращать управление (для чего может служить вызов функции time.sleep(0)).
4. Перевести поток назад в приостановленное состояние (выйти из потока).
5. Разблокировать глобальную блокировку интерпретатора.
6. Снова проделать все эти действия (lather, rinse, repeat).

Если сделан вызов внешнего кода (допустим, любой встроенной функции расширения C/C++), то глобальная блокировка интерпретатора будет заблокирована до завершения этого вызова (поскольку в языке Python невозможно задать интервал с помощью байт-кода). Тем не менее при программировании расширений не исключена возможность разблокирования глобальной блокировки интерпретатора, что позволяет избавить разработчика Python от необходимости брать на себя управление блокировками в коде Python в подобных ситуациях.

Например, в любых процедурах Python, основанных на использовании ввода-вывода (в которых вызывается встроенный код С операционной системы), предусмотрено освобождение глобальной блокировки интерпретатора до вызова функции ввода-вывода, что позволяет продолжить выполнение других потоков, в то время как происходит ввод-вывод. Если же в коде не осуществляется большой объем ввода-вывода, то, как правило, процессор (и глобальная блокировка интерпретатора) блокируется на полный интервал времени, предоставленный потоку, пока он не вернет управление.

После того как поток завершает выполнение задачи, для которой он был создан, происходит выход из потока. Выход из потока может осуществляться путем вызова одной из функций выхода, такой как thread.exit(), с применением любого из стандартных способов выхода из процесса Python, например sys.exit(), или с помощью генерирования исключения SystemExit. Однако возможность непосредственно уничтожить поток отсутствует.

Мы будем рассматривать два модуля Python, применяемых для работы с потоками, но один из них, модуль thread, не рекомендуется для использования. Для этого есть много причин, но наиболее важной из них является то, что применение этого модуля приводит к завершению работы всех прочих потоков после выхода из основного потока, и при этом очистка памяти не осуществляется должным образом. Второй модуль, threading, гарантирует, что весь процесс будет оставаться действующим до тех пор, пока не произойдет выход из всех важных дочерних потоков.

Тем не менее в основные потоки всегда следует закладывать такие алгоритмы, чтобы они качественно выполняли функции диспетчера и при осуществлении этой задачи учитывали, какое назначение имеют отдельные потоки, какие данные или параметры требуются для каждого из порожденных потоков, когда эти потоки завершат выполнение и какие результаты предоставят. В ходе выполнения этой работы основные потоки могут дополнительно формировать отдельные результаты в виде окончательного, значимого вывода.

Язык Python поддерживает многопоточное программирование с учетом особенностей операционной системы, под управлением которой он функционирует. Он поддерживается на большинстве платформ на основе Unix, таких как Linux, Solaris, Mac OS X, *BSD, а также на персональных компьютерах под управлением Windows. В языке Python используются потоки, совместимые со стандартом POSIX, которые иногда называют пи-потоками (pthreads).

По умолчанию поддержка потоков включается при построении интерпретатора Python из исходного кода (начиная с версии Python 2.0) или при установке исполняемой программы интерпретатора в среде Win32. Чтобы определить, предусмотрено ли применение потоков на конкретном установленном интерпретаторе, достаточно просто попытаться импортировать модуль thread из интерактивного интерпретатора (если потоки недоступны, то появится сообщение об ошибке).

```
>>> import thread
```

В языке Python предусмотрено несколько модулей, позволяющих упростить задачу многопоточного программирования, включая модули thread, threading и Queue. Для создания потоков и управления ими программисты могут использовать модули thread и threading. В модуле thread предусмотрены простые средства управления потоками и блокировками, а модуль threading обеспечивает высокоуровневое, полноценное управление потоками. С помощью модуля Queue пользователи могут создать структуру данных очереди, совместно используемую несколькими потоками.

Однако, как уже было сказано, по многим причинам рекомендуется использовать высокоуровневый модуль threading вместо thread. Модуль threading имеет более широкий набор функций по сравнению с модулем thread, обеспечивает лучшую поддержку потоков, и в нем исключены некоторые конфликты атрибутов, обнаруживаемые в модуле thread. Еще одна причина отказаться от использования модуля thread состоит в том, что thread — это модуль более низкого уровня и имеет мало примитивов синхронизации (фактически только один), в то время как модуль threading обеспечивает более широкую поддержку синхронизации.

Еще одна причина отказа от работы с модулем thread состоит в том, что этот модуль не позволяет взять под свое управление выход из процесса. После завершения основного потока происходит также уничтожение всех прочих потоков без предупреждения или надлежащей очистки памяти. Как было указано выше, модуль threading позволяет по меньшей мере дождаться завершения работы важных дочерних потоков и только после этого выйти из программы.

Использование модуля thread рекомендуется только для экспертов, которым требуется получить доступ к потоку на более низком уровне. Для того чтобы эта особенность модуля стала более очевидной, в Python 3 он был переименован в _thread. В любом создаваемом многопоточном приложении следует использовать threading, а также, возможно, другие высокоуровневые модули.

Тем не менее, всё равно вначале рассмотрим, какие задачи возлагались на модуль thread. От модуля thread требовалось не только порождать потоки, но и обеспечивать работу с основной структурой синхронизации данных, называемой объектом блокировки (таковыми являются примитивная блокировка, простая блокировка, блокировка со взаимным исключением, мьютекс и двоичный семафор). Как было указано выше, без подобных примитивов синхронизации сложно обойтись при управлении потоками.

Основные функции модуля thread:

`start_new_thread(function, args, kwargs=None)` – порождает новый поток и вызывает ему на выполнение фукнцию function с заданными параметрами args и необязательными параметрами kwargs

`allocate_lock()` – распределяет объект блокировки LockType

`exit()` – даёт указание о выходе из потока

Методы объекта LockType:

`acquire(wait=None)` – попытка захватить объект блокировки

`locked()` – возвращает true или false в зависимости от того, захвачена ли блокировка

`release()` – освобождает блокировку.

Ключевой функцией модуля thread является start_new_thread(). Эта функция получает предназначенную для вызова функцию (объект) с позиционными параметрами и (необязательно) с ключевыми параметрами. Специально для вызова функции создается новый поток.

```
import thread
from time import sleep, ctime

def loop0():
    print 'start loop 0 at:', ctime()
    sleep(4)
    print 'loop 0 done at:', ctime()

def loop1():
    print 'start loop 1 at:', ctime()
    sleep(2)
    print 'loop 1 done at:', ctime()

def main():
    print 'starting at:', ctime()
    thread.start_new_thread(loop0, ())
    thread.start_new_thread(loop1, ())
    sleep(6)
    print 'all DONE at:', ctime()

if __name__ == '__main__':
    main()
```

Важным изменением в сравнении с последовательным кодом, помимо создания потоков, является добавление вызова sleep(6). Причина необходимости такого добавления состоит в том, что если не будет установлен запрет на продолжение основного потока, то в нем произойдет переход к следующей инструкции, появится сообщение “all done” (работа закончена) и работа программы завершится после уничтожения обоих потоков, в которых выполняются функции loop0() и loop1().

В сценарии отсутствует какой-либо код, который бы указывал основному потоку, что следует ожидать завершения дочерних потоков, прежде чем продолжить выполнение инструкций. Это — одна из ситуаций, которая показывает, что подразумевается под утверждением, согласно которому для потоков требуется определенная синхронизация. В данном случае в качестве механизма синхронизации применяется еще один вызов sleep(). При этом используется значение продолжительности приостановки, равное 6 секундам, поскольку известно, что оба потока (которые занимают 4 и 2 секунды) должны были завершиться до того, как в основном потоке будет отсчитан интервал времени 6 секунд.

Напрашивается вывод, что должен быть какой-то более удобный способ управления потоками по сравнению с созданием дополнительной задержки в 6 секунд в основном потоке. Дело в том, что из-за этой задержки общее время прогона ненамного лучше по сравнению с однопоточной версией. К тому же применение функции sleep() для синхронизации потоков, как в данном примере, не позволяет обеспечить полную надежность. Например, может оказаться, что синхронизируемые потоки являются независимыми друг от друга, а значения времени их выполнения изменяются. В таком случае выход из основного потока может произойти слишком рано или слишком поздно. Как оказалось, гораздо лучшим способом синхронизации является применение блокировок.

```
import thread
from time import sleep, ctime

loops = [4,2]

def loop(nloop, nsec, lock):
    print 'start loop', nloop, 'at:', ctime()
     sleep(nsec)
     print 'loop', nloop, 'done at:', ctime()
     lock.release()

    def main():
        print 'starting at:', ctime()
        locks = []
        nloops = range(len(loops))

     for i in nloops:
         lock = thread.allocate_lock()
         lock.acquire()
         locks.append(lock)

  for i in nloops:
      thread.start_new_thread(loop,
          (i, loops[i], locks[i]))

     for i in nloops:
         while locks[i].locked(): pass

     print 'all DONE at:', ctime()

 if __name__ == '__main__':
     main()
```

В этом сценарии значительная часть работы выполняется в функции main(), для чего применяются три отдельных цикла for. Вначале создается список блокировок, для получения которых используется функция thread.allocate_lock(), затем происходит захват каждой блокировки (отдельно) с помощью метода acquire(). Захват блокировки приводит к тому, что блокировка становится недоступной для дальнейшего манипулирования ею. После того как блокировка становится заблокированной, она добавляется к списку блокировок locks. Операция порождения потоков осуществляется в следующем цикле, после чего для каждого потока вызывается функция loop(), потоку присваивается номер цикла, задается продолжительность приостановки, и, наконец, для этого потока захватывается блокировка. Почему в данном случае не происходит запуск потоков в цикле захвата блокировок? На это есть две причины. Во-первых, необходимо обеспечить синхронизацию потоков, чтобы все они были запущены примерно в одно и то же время, и, во-вторых, приходится учитывать, что захват блокировок связан с определенными затратами времени. Если поток выполняет свою задачу слишком быстро, то может завершиться еще до того, как появится шанс захватить блокировку.
Задача разблокирования своего объекта блокировки после завершения выполнения возлагается на сам поток. В последнем цикле осуществляются лишь ожидание и возврат в начало (тем самым обеспечивается приостановка основного потока), и это происходит до тех пор, пока не будут освобождены обе блокировки.

Перейдем к описанию высокоуровневого модуля threading, который не только предоставляет класс Thread, но и дает возможность воспользоваться широким разнообразием механизмов синхронизации, позволяющих успешно решать многие важные задачи.
Основные объекты модуля threading:
Thread - Объект, который представляет отдельный поток выполнения
Lock - Примитивный объект блокировки (такая же блокировка, как и в модуле thread)
Rlock - Реентерабельный объект блокировки предоставляет возможность в отдельном потоке (повторно) захватывать уже захваченную блокировку (это рекурсивная блокировка)

Condition - Объект условной переменной вынуждает один поток ожидать, пока определенное условие не будет выполнено другим потоком. Таким условием может быть изменение состояния или задание определенного значения данных
Event - Обобщенная версия условных переменных, которая позволяет обеспечить ожидание некоторого события любым количеством потоков, так что после обнаружения этого события происходит активизация всех потоков
Semaphore - Предоставляет счетчик конечных ресурсов, совместно используемый потоками; если ни один из ресурсов не доступен, происходит блокировка
BoundedSemaphore - Аналогично Semaphore, но гарантируется, что превышение начального значения никогда не произойдет
Timer - Аналогично Thread, за исключением того, что происходит ожидание в течение выделенного промежутка времени перед выполнением
Barrier - Создает барьер, которого должно достичь определенное количество потоков, прежде чем всем этим потокам будет разрешено продолжить работу (новое в Python 3.2)

Еще одна причина, по которой следует избегать использование модуля thread, состоит в том, что он не поддерживает принцип организации работы программы на основе демонов (потоков, работающих в фоновом режиме). Модуль thread действует так, что после выхода из основного потока все дочерние потоки уничтожаются, без учета того, должны ли они продолжить выполнение определенной работы. Если это нежелательно, то можно организовать функционирование потоков в качестве демонов.
Поддержка демонов предусмотрена в модуле threading. Ниже описано, как они функционируют. Обычно демон применяется в качестве сервера, который ожидает поступления клиентских запросов, подлежащих выполнению. Если нет никакой работы, поступившей от клиентов, которая должна быть сделана, то демон простаивает. Для потока может быть установлен флаг, указывающий, что этот поток может выполнять роль демона. Такое указание равносильно обозначению родительского потока как не требующего после своего завершения, чтобы был завершен дочерний поток. Потоки сервера функционируют в виде бесконечных циклов и в обычных ситуациях не завершают свою работу.
Дочерние потоки могут быть также обозначены флагами как демоны, если в основном потоке могут складываться условия готовности к выходу, но нет необходимости ожидать завершения работы дочерних потоков, чтобы выйти из основной программы. Значение true указывает, что дочерний поток не приносит результатов, от которых зависит возможность завершения всей программы, и в основном рассматривается как указание, что единственным назначением потока является ожидание запросов от клиентов и их обслуживание.
Чтобы обозначить поток как выполняющий функции демона, необходимо применить оператор присваивания thread.daemon = True, прежде чем запустить этот поток. То же является справедливым в отношении проверки того, выполняет ли поток функции демона; достаточно проверить значение соответствующей переменной (а не вызывать функцию thread.isDaemon()). Новый дочерний поток наследует свой флаг, обозначающий его в качестве демона, от родительского потока. Вся программа Python (рассматриваемая как основной поток) продолжает функционировать до тех пор, пока не произойдет выход из всех потоков, не обозначенных как демоны.

В программе с многопоточной организацией главным инструментом является класс Thread модуля threading. Модуль threading поддерживает целый ряд функций, отсутствующих в модуле thread.

Атрибуты данных объекта потока
name - Имя потока
Ident - Идентификатор потока
Daemon - Булев флаг, указывающий, выполняет ли поток функции демона

Методы объекта потока
__init__(group=None,target=None, name=None,args=(), kwargs={},verbose=None,daemon=None) -
Порождение объекта Thread с использованием целевого параметра callable и набора параметров args или kwargs. Может быть также передан параметр name или group, но обработка последнего не реализована. Принимается также флаг verbose. Любое ненулевое значение daemon задает атрибут/флаг thread.daemon

start() - Запуск выполнения потока
run() - Метод, определяющий функционирование потока (обычно перекрывается разработчиком приложения в подклассе)
join(timeout=None) - Приостановка до завершения запущенного потока; блокировка, если не задан параметр timeout (в секундах)
getName() - Возвращаемое имя потока
setName(name) - Заданное имя потока
isAlive/is_alive() - Булев флаг, указывающий, продолжает ли поток работать
isDaemon() - Возвращает True, если поток выполняет функции демона, в противном случае возвращает False
setDaemon(daemonic) - Задание флага работы в режиме демона равным указанному булеву значению daemonic (вызов должен осуществляться перед выполнением функции start() для потока)

Предусмотрен целый ряд способов, с помощью которых могут создаваться потоки на основе класса Thread. В данной главе рассматриваются три из этих способов, которые мало отличаются друг от друга. Программист может выбрать способ, который является для него наиболее удобным, не говоря уже о том, что выбранный способ должен быть наиболее подходящим с точки зрения приложения и масштабирования в будущем (предпочтительным является последний из приведенных способов).
• Создание экземпляра Thread с передачей функции.
• Создание экземпляра Thread и передача вызываемого экземпляра класса.
• Формирование подкласса Thread и создание экземпляра подкласса.
Как правило, используется первый или третий вариант. Третий становится предпочтительным, если требуется создать в большей степени объектно-ориентированный интерфейс, а первый — в противном случае. Второй вариант, откровенно говоря, является немного более громоздким, и, как показывает практика, его применение приводит к созданию программ, более сложных для восприятия.

Первый вариант.

```
import threading
from time import sleep, ctime

loops = [4,2]

def loop(nloop, nsec):
    print 'start loop', nloop, 'at:', ctime()
     sleep(nsec)
     print 'loop', nloop, 'done at:', ctime()

 def main():
     print 'starting at:', ctime()
     threads = []
     nloops = range(len(loops))

     for i in nloops:
         t = threading.Thread(target=loop,
             args=(i, loops[i]))
         threads.append(t)

     for i in nloops:          # запуск потоков
         threads[i].start()

     for i in nloops:          # ожидание завершения
         threads[i].join()     # всех потоков

     print 'all DONE at:', ctime()
```

Удалось избавиться от блокировок, которые приходилось реализовывать при использовании модуля thread. Вместо этого создается ряд объектов Thread. После создания экземпляра каждого объекта Thread остается лишь передать функцию (target) и параметры (args) и получить взамен экземпляр Thread. Наибольшее различие между созданием экземпляра Thread (путем вызова Thread()) и вызовом thread.start_new_thread() состоит в том, что в первом случае запуск нового потока не происходит

немедленно. Это удобно с точки зрения синхронизации, особенно если не требуется, чтобы потоки запускались сразу после их создания.
Иными словами, появляется возможность почти одновременно запустить все потоки по окончании их распределения, но не раньше, для чего остается лишь вызвать метод start() каждого потока. Кроме того, отпадает необходимость заниматься управлением целым рядом блокировок (выделением, захватом, освобождением, проверкой состояния блокировки и т.д.), поскольку достаточно лишь вызвать метод join() для каждого потока. Метод join() обеспечивает переход в состояние ожидания до завершения работы потока или до истечения тайм-аута, если он предусмотрен. Использование метода join() открывает путь к созданию гораздо более наглядных программ по сравнению с применением бесконечного цикла, в котором происходит ожидание освобождения блокировок (такие блокировки иногда именуются спин-блокировками, или “крутящимися” блокировками, именно по той причине, что применяются в бесконечном цикле).
Еще одной важной отличительной особенностью метода join() является то, что он вообще не требует вызова. После запуска потока его выполнение происходит до завершения переданной ему функции, после чего осуществляется выход из потока. Если в основном потоке должны быть выполнены какие-то другие действия, кроме ожидания завершения потоков (такие как дополнительная обработка или ожидание новых клиентских запросов), организовать это совсем несложно. Метод join() становится удобным, только если требуется обеспечить ожидание завершения потока.

Второй способ.

```
import threading
from time import sleep, ctime

loops = [4,2]

class ThreadFunc(object):

    def __init__(self, func, args, name=''):
        self.name = name
        self.func = func
        self.args = args

    def __call__(self):
        self.func(*self.args)

def loop(nloop, nsec):
    print 'start loop', nloop, 'at:', ctime()
    sleep(nsec)
    print 'loop', nloop, 'done at:', ctime()

def main():
    print 'starting at:', ctime()
    threads = []
    nloops = range(len(loops))

    for i in nloops:  # создание всех потоков
        t = threading.Thread(
            target=ThreadFunc(loop, (i, loops[i]),
            loop.__name__))
        threads.append(t)

    for i in nloops:  # запуск всех потоков
        threads[i].start()

    for i in nloops:  # ожидание завершения
        threads[i].join()

    print 'all DONE at:', ctime()
```

Произошло добавление класса ThreadFunc и внесены небольшие изменения в процедуру создания экземпляра объекта Thread, в которой также порождается ThreadFunc, новый вызываемый класс. Фактически в данном примере применяется процедура создания не одного, а двух экземпляров.
Рассмотрим класс ThreadFunc более подробно.
Этот класс должен быть достаточно общим, для того чтобы его можно было использовать не только с функцией loop(), но и с другими функциями, поэтому была добавлена некоторая новая инфраструктура, которая обеспечивает хранение этим классом параметров для функции, самой функции, а также строки с именем функции. Конструктор __init__() лишь задает все необходимые значения.
При вызове в коде Thread объекта ThreadFunc в связи с созданием нового потока вызывается специальный метод __call__(). Необходимый набор параметров уже задан, поэтому его не обязвтельно передавать конструктору Thread() и можно вызывать функцию непосредственно.

Третий вариант.

```
import threading
from time import sleep, ctime

loops = (4, 2)

class MyThread(threading.Thread):
    def __init__(self, func, args, name=''):
        threading.Thread.__init__(self)
        self.name = name
        self.func = func
        self.args = args

    def run(self):
        self.func(*self.args)

def loop(nloop, nsec):
    print 'start loop', nloop, 'at:', ctime()
    sleep(nsec)
    print 'loop', nloop, 'done at:', ctime()

def main():
    print 'starting at:', ctime()
    threads = []
    nloops = range(len(loops))

    for i in nloops:
        t = MyThread(loop, (i, loops[i]),
            loop.__name__)
        threads.append(t)

    for i in nloops:
        threads[i].start()

    for i in nloops:
        threads[i].join()

    print 'all DONE at:', ctime()'
```

Необходимо подчеркнуть наиболее значительные отличия: во-первых, в конструкторе подкласса MyThread приходится вначале вызывать конструктор базового класса (threading.Thread.__init__(self)), и, во-вторых, применявшийся ранее специальный метод __call__() должен получить в подклассе имя run().

В модуле Threading, кроме различных объектов синхронизации и обеспечения многопоточной поддержки, предусмотрены также некоторые поддерживающие функции.
activeCount/active_count() - Возвращает количество активных в настоящее время объектов Thread
currentThread()/current_thread - Возвращает текущий объект Thread

enumerate() - Возвращает список всех активных в настоящее время объектов Thread.

Мы представили лишь упрощенные, применяемые в качестве примеров фрагменты кода, которые весьма далеки от того, что должно применяться в реальном приложении. Фактически единственное назначение этих примеров состоит лишь в том, чтобы показать потоки в работе и продемонстрировать различные способы их создания. При этом во всех примерах запуск потоков и ожидание их завершения происходит почти одинаково, а действия, выполняемые потоками, главным образом сводятся к приостановке.
Кроме того, как уже было сказано в разделе 4.3.1, виртуальная машина Python в действительности работает в однопоточном режиме (с применением глобальной блокировки интерпретатора), поэтому достижение большего распараллеливания в программе Python возможно, только если многопоточная организация применяется в приложении, ограничиваемом пропускной способностью ввода-вывода, а не пропускной способностью процессора, в котором так или иначе происходит лишь циклическая передача управления от одного процесса к другому.

Мы рассмотрели основные концепции многопоточной организации и было показали, как использовать многопоточность в приложениях Python. Однако при этом мы не затронули один очень важный аспект многопоточного программирования: синхронизация. Довольно часто в многопоточном коде содержатся определенные функции или блоки, в которых необходимо (или желательно) ограничить количество выполняемых потоков до одного. Обычно такие ситуации обнаруживаются при внесении изменений в базу данных, обновлении файла или выполнении подобных действий, при которых может возникнуть состояние состязания. Такое состояние проявляется, если код допускает появление нескольких путей выполнения или вариантов поведения либо формирование несогласованных данных, если один поток будет запущен раньше другого, или наоборот.
В таких случаях возникает необходимость обеспечения синхронизации. Синхронизация должна использоваться, если к какому-то из критических участков кода могут подойти одновременно несколько потоков, но в каждый конкретный момент времени должно быть разрешено дальнейшее выполнение только одного потока. Программист регламентирует прохождение потоков и для управления ими выбирает подходящие примитивы синхронизации, или механизмы управления потоками, с помощью которых вводит в действие синхронизацию. Предусмотрено несколько различных методов синхронизации процессов, часть которых поддерживается языком Python. Эта поддержка предоставляет достаточно возможностей для выбора метода, наиболее подходящего для конкретной задачи.
Методы синхронизации уже были представлены ранее, в начале этого раздела, поэтому перейдем к рассмотрению нескольких примеров сценариев, в которых используются примитивы синхронизации двух типов: блокировки/мьютексы и семафоры. Блокировка относится к числу самых простых среди всех механизмов синхронизации и находится на самом низком уровне, а семафоры предназначены для применения в таких ситуациях, в которых несколько потоков конкурируют друг с другом, стремясь получить доступ к ограниченным ресурсам. Понять назначение блокировок проще, поэтому начнем рассмотрение примитивов синхронизации с них, а затем перейдем к семафорам.

Блокировки, как и следовало ожидать, имеют два состояния: заблокированное и разблокированное. Блокировки поддерживают только две функции: acquire и release. Эти функции действуют в полном соответствии с их именами — захват и освобождение.
Иногда необходимость пройти критический участок кода возникает в нескольких потоках. В таком случае можно организовать конкуренцию между потоками за блокировку, и первый поток, который сможет ее захватить, получит разрешение войти в критический участок и выполнить содержащийся в нем код. Все остальные одновременно поступающие потоки блокируются до того времени, когда первый поток завершит свою работу, выйдет из критического участка и освободит блокировку. С этого момента возможность захватить блокировку и войти в критический участок получает любой из оставшихся ожидающих потоков. Заслуживает внимания то, что отсутствует какое-либо упорядочение потоков, работа которых организована с помощью блокировок (т.е. применяется принцип простой очереди — “первым пришел, первым обслуживается”); процесс выбора потока-победителя не детерминирован и может зависеть даже от применяемой реализации Python.

Как уже было сказано, блокировки являются довольно простыми для понимания и могут быть легко реализованы. Кроме того, можно довольно легко определить, в каком случае они действительно необходимы. Однако ситуация может оказаться сложнее, и тогда вместо блокировки потребуется более мощный примитив синхронизации. Если в приложении приходится иметь дело с ограниченными ресурсами, то семафоры могут оказаться более приемлемыми.
Семафоры относятся к числу примитивов синхронизации, которые были введены в действие раньше других. Семафор, по существу, представляет собой счетчик, значение которого уменьшается после захвата ресурса (и снова увеличивается после освобождения ресурса). Семафоры, представляющие закрепленные за ними ресурсы, можно рассматривать как доступные или недоступные. Действие по захвату ресурса и уменьшению значения счетчика принято обозначать как P(), но для обозначения этого действия применяются также термины “переход в состояние ожидания”, “осуществление попытки”, “захват”, “приостановка” или “получение”. И наоборот, после завершения работы потока с ресурсом должен быть произведен возврат ресурса в пул ресурсов. Для этого применяется действие, по традиции обозначаемое V(). Это действие обозначается также как “сигнализация”, “наращивание”, “отпускание”, “отправка”, “освобождение”. В языке Python вместо всех этих вариантов именования применяются упрощенные обозначения, согласно которым функции и (или) методы обозначаются как те, что служат для работы с блокировками: acquire и release. Семафоры являются более гибкими, чем блокировки, поскольку обеспечивают работу с несколькими потоками, в каждом из которых используется один из экземпляров конечного ресурса.
В модуле threading предусмотрены два класса семафоров, Semaphore и BoundedSemaphore. Как уже было сказано, в действительности семафоры — просто счетчики; при запуске семафора задается некоторое постоянное число, которое определяет конечное количество единиц ресурса.
Значение этого счетчика уменьшается на 1 после потребления одной единицы из конечного количества единиц ресурса, а после возврата этой единицы в пул значение счетчика увеличивается. Объект BoundedSemaphore предоставляет дополнительную возможность, которая состоит в том, что значение счетчика не может быть увеличено свыше установленного для него начального значения; иными словами, позволяет предотвратить возникновение такой абсурдной ситуации, при которой количество операций освобождения семафора превышает количество операций его захвата.
Методы работы с семафорами:
acquire(blocking=True, timeout=-1) – захват семафора (уменьшение счётчика). Параметры позволяют блокировать или не блокировать поток при неудавшемся захвате и выставлять таймаут операции.
release() – освобождение семафора. Генерирует исключение ValueError для BoundedSemaphore, если значение после освобождения превысит максимально установленное.

В заключительном примере иллюстрируется принцип работы “производитель–потребитель”, согласно которому производитель товаров или поставщик услуг производит товары или подготавливает услуги и размещает их в структуре данных наподобие очереди. Интервалы между отдельными событиями передачи произведенных товаров в очередь не детерминированы, как и интервалы потребления товаров.
Модуль Queue (который носит это имя в версии Python 2.x, но переименован в queue в версии 3.x) предоставляет механизм связи между потоками, с помощью которого отдельные потоки могут совместно использовать данные. В данном случае создается очередь, в которую производитель (один поток) помещает новые товары, а потребитель (другой поток) их расходует.

Общие атрибуты модуля Queue/queue:
Классы модуля Queue/queue
Queue(maxsize=0) - Создает очередь с последовательной организацией, имеющую указанный размер maxsize, которая не позволяет вставлять новые блоки после достижения этого размера. Если размер не указан, то длина очереди становится неограниченной
LifoQueue(maxsize=0) - Создает стек, имеющий указанный размер maxsize, который не позволяет вставлять новые блоки после достижения этого размера. Если размер не указан, то длина стека становится неограниченной
PriorityQueue(maxsize=0) - Создает очередь по приоритету, имеющую указанный размер maxsize, которая не позволяет вставлять новые блоки после достижения этого размера. Если размер не указан, то длина очереди становится неограниченной

Исключения модуля Queue/queue
Empty - Активизируется при вызове метода get*() применительно к пустой очереди
Full - Активизируется при вызове метода put*() применительно к заполненной очереди
Методы объекта Queue/queue
qsize() - Возвращает размер очереди (это — приблизительное значение, поскольку при выполнении этого метода может происходить обновление очереди другими потоками)
empty() - Возвращает True, если очередь пуста; в противном случае возвращает False
full() - Возвращает True, если очередь заполнена; в противном случае возвращает False
put(item, block=True,timeout=None) - Помещает элемент item в очередь; если значение block равно True (по умолчанию) и значение timeout равно None, устанавливает блокировку до тех пор, пока в очереди не появится свободное место. Если значение timeout является положительным, блокирует очередь самое больше на timeout секунд, а если значение block равно False, активизирует исключение Empty
put_nowait(item) - То же, что и put(item, False)
get(block=True, timeout=None) - Получает элемент из очереди, если задано значение block (отличное от 0); устанавливает блокировку до того времени, пока элемент не станет доступным
get_nowait() - То же, что и get(False)
task_done() - Используется для указания на то, что работа по постановке элемента в очередь завершена, в сочетании с описанным ниже методом join()
join() - Устанавливает блокировку до того времени, пока не будут обработаны все элементы в очереди; сигнал об этом вырабатывается путем вызова описанного выше метода task_done().

```
from random import randint
from time import sleep
from Queue import Queue
from myThread import MyThread

def writeQ(queue):
    print 'producing object for Q...',
     queue.put('xxx', 1)
     print "size now", queue.qsize()

 def readQ(queue):
     val = queue.get(1)
     print 'consumed object from Q... size now', \
             queue.qsize()

 def writer(queue, loops):
     for i in range(loops):
         writeQ(queue)
         sleep(randint(1, 3))

 def reader(queue, loops):
     for i in range(loops):
         readQ(queue)
         sleep(randint(2, 5))

 funcs = [writer, reader]
 nfuncs = range(len(funcs))

 def main():
     nloops = randint(2, 5)
     q = Queue(32)

     threads = []
     for i in nfuncs:
         t = MyThread(funcs[i], (q, nloops),
             funcs[i].__name__)
         threads.append(t)

     for i in nfuncs:
         threads[i].start()

     for i in nfuncs:
          threads[i].join()

     print 'all DONE'
```

В этом модуле используется объект Queue.Queue, а также потоки, сформированные с помощью класса myThread.MyThread, как было описано ранее. Метод random.randint() применяется для внесения элемента случайности в операции производства и потребления. (Заслуживает внимания то, что метод random.randint() действует точно так же, как и метод random.randrange(), но предусматривает включение в интервал вырабатываемых случайных чисел начального и конечного значений.)
Функции writeQ() и readQ() выполняют следующие операции: первая из них помещает объект в очередь (в качестве объекта может использоваться, например, строка 'xxx'), а вторая извлекает объект из очереди. Следует учитывать, что операции постановки в очередь и изъятия из очереди осуществляются одновременно по отношению только к одному объекту.
Метод writer() выполняется как отдельный поток, единственным назначением которого является выработка одного элемента для постановки в очередь, переход на время в состояние ожидания, а затем повтор этого цикла указанное количество раз, причем количество повторов устанавливается при выполнении сценария случайным образом. Метод reader() действует аналогично, если не считать того, что он не ставит, а извлекает элементы из очереди.
Необходимо отметить, что устанавливаемая случайным образом продолжительность приостановки метода-производителя в секундах, как правило, меньше по сравнению с той продолжительностью, на

которую приостанавливается метод-потребитель. Это сделано для того, чтобы метод-потребитель не мог предпринять попытки извлечения элементов из пустой очереди. Сокращение продолжительности приостановки метода-производителя способствует повышению вероятности того, что в распоряжении метода-потребителя всегда будет пригодный для извлечения элемент, когда настанет время очередного выполнения этой операции.
Наконец, предусмотрена функция main(), которая должна выглядеть весьма подобной функциям main() из всех прочих сценариев, приведенных в этой главе. Создаются необходимые потоки и осуществляется их запуск, а окончание работы наступает после того, как оба потока завершают свое выполнение.
На основании этого примера можно сделать вывод, что программа, предназначенная для выполнения нескольких задач, может быть организована так, чтобы для реализации каждой из задач применялись отдельные потоки. Результатом может стать получение гораздо более наглядного проекта программы по сравнению с однопоточной программой, в которой предпринимается попытка обеспечить выполнение всех задач.
Мы показали, что применение однопоточного процесса может стать препятствием к повышению производительности приложения. Особенно значительно может быть повышена производительность программ, основанных на последовательном выполнении независимых, недетерминированных и не имеющих причинных зависимостей задач, в результате их разбиения на отдельные задачи, выполняемые отдельными потоками. Существенный выигрыш от перехода к многопоточной обработке может быть достигнут не во всех приложениях. Причинами этого могут стать дополнительные издержки, а также тот факт, что сам интерпретатор Python представляет собой однопоточное приложение. Тем не менее овладение функциональными возможностями многопоточной организации Python позволяет взять этот инструмент на вооружение, когда это оправдано.

Прежде чем приступать к повсеместному применению средств поддержки многопоточности, следует провести краткий обзор особенностей такой организации программирования. Вообще говоря, применение нескольких потоков в программе может способствовать ее улучшению. Однако в интерпретаторе Python применяется глобальная блокировка, которая накладывает свои ограничения, поэтому многопоточная организация является более подходящей для приложений, ограничиваемых пропускной способностью ввода-вывода (при вводе-выводе происходит освобождение глобальной блокировки интерпретатора, что способствует повышению степени распараллеливания), а не приложений, ограничиваемых пропускной способностью процессора. В последнем случае для достижения более высокой степени распараллеливания необходимо иметь возможность параллельного выполнения процессов несколькими ядрами или процессорами.
Не вдаваясь в дополнительные подробности, перечислим основные альтернативы модулю threading, касающиеся поддержки нескольких потоков или процессов.
В первую очередь вместо модуля threading можно применить модуль subprocess, когда возникает необходимость запуска новых процессов, либо для выполнения кода, либо для обеспечения обмена данными с другими процессами через стандартные файлы ввода-вывода (stdin, stdout, stderr). Этот модуль был введен в версии Python 2.4.
Модуль multiprocessing, впервые введенный в Python 2.6, позволяет запускать процессы для нескольких ядер или процессоров, но с интерфейсом, весьма напоминающим интерфейс модуля threading. Он также поддерживает различные механизмы передачи данных между процессами, применяемыми для выполнения совместной работы.
Модуль concurrent.futures – это новая высокоуровневая библиотека, которая работает только на уровне заданий. Это означает, что при использовании модуля concurrent.futures исключается необходимость заботиться о синхронизации либо управлять потоками или процессами. Достаточно лишь указать поток или пул процесса с определенным количеством рабочих потоков, передать задания на выполнение и собрать полученные результаты. Этот модуль впервые появился в версии Python 3.2, но перенесен также в версию Python 2.6 и последующие версии.