ГЛАВА 4

Многопоточное программирование

В этой главе...

- Введение/общее назначение
- Потоки и процессы
- Поддержка потоков в языке Python
- Модуль `thread`
- Модуль `threading`
- Сравнение однопоточного и многопоточного выполнения
- Практическое применение многопоточной обработки
- Проблема “производитель–потребитель” и модуль `Queue/queue`
- Дополнительные сведения об использовании потоков
- Связанные модули

> С помощью Python можно запустить поток, но нельзя его остановить.
> Вернее, приходится ожидать, пока он не достигнет конца выполнения.
> Это означает, что все происходит, как в группе новостей [comp.lang.python]?
Обмен сообщениями между Клиффом Уэллсом (Cliff Wells)
и Стивом Холденом (Steve Holden) с участием Тимоти Делани
(Timothy Delaney), февраль 2002 г.

В настоящей главе рассматриваются различные способы обеспечения параллельного выполнения в коде. В первых нескольких разделах этой главы показано, в чем состоят различия между процессами и потоками. После этого будет дано определение понятия многопоточного программирования и представлены некоторые средства многопоточного программирования, предусмотренные в языке Python. (Читатели, уже знакомые с многопоточным программированием, могут перейти непосредственно к разделу 4.3.5.) В заключительных разделах этой главы приведены некоторые примеры того, как использовать модули `threading` и `Queue` для реализации многопоточного программирования с помощью языка Python.

4.1. Введение/общее назначение

До появления средств многопоточного (multithreaded — MT) программирования выполнение компьютерных программ состояло из единой последовательности шагов, которые выполнялись процессором компьютера от начала до конца, т.е. *синхронно*. Такая организация выполнения применялась независимо от того, требовала ли сама задача последовательного упорядочения шагов или допускала разбиение на подзадачи и отдельное их выполнение в программе. В последнем случае подзадачи вообще могли быть независимыми, не связанными никакими причинно-следственными отношениями (а это означает, что результаты одних подзадач не влияют на выполнение других подзадач). Из этого следует вывод, что такие независимые задачи могут выполняться не последовательно, а одновременно. Подобная параллельная организация работы позволяет существенно повысить эффективность решения всей задачи. Изложенные выше соображения лежат в основе многопоточного программирования.

Многопоточное программирование идеально подходит для решения задач, *асинхронных* по своему характеру (т.е. допускающих прерывание работы), требующих выполнения нескольких параллельных действий, в которых реализация каждого действия может быть *недетерминированной*, иными словами, происходящей в случайные и непредсказуемые моменты времени. Такие задачи программирования могут быть организованы в виде нескольких потоков выполнения или разделены на несколько потоков, в каждом из которых осуществляется конкретная подзадача. В зависимости от приложения в этих подзадачах могут вычисляться промежуточные результаты для последующего слияния и формирования заключительной части вывода.

Задачи, выполнение которых ограничено пропускной способностью процессора, довольно легко разделить на подзадачи, выполняемые в последовательном или многопоточном режиме. С другой стороны, организовать выполнение однопоточного процесса с несколькими внешними источниками ввода не столь просто. Для решения этой задачи без применения многопоточного режима в последовательной программе необходимо предусмотреть один или несколько таймеров и реализовать схему мультиплексирования.

В последовательной программе потребуется опрашивать каждый терминальный канал ввода-вывода для проверки наличия данных, введенных пользователем. При этом необходимо добиться того, чтобы в программе не происходило блокирование при чтении из терминального канала ввода-вывода, поскольку сам характер поступления введенных пользователем данных является недетерминированным, а блокировка привела бы к нарушению обработки данных из других каналов ввода-вывода. В такой последовательной программе приходится использовать неблокирующий ввод-вывод или блокирующий ввод-вывод с таймером (чтобы блокировка устанавливалась лишь на время).

Последовательная программа представляет собой единый поток выполнения, поэтому ей приходится манипулировать отдельными подзадачами, чтобы на любую отдельно взятую подзадачу не затрачивалось слишком много времени, а также следить за тем, чтобы длительность формирования ответов пользователям соответствовала установленным критериям. Применение последовательной программы для решения задачи такого типа часто требует организации сложной системы передачи управления, которую трудно понять и сопровождать.

Если же для решения подобной задачи программирования применяется многопоточная программа с общей структурой данных, такой как `Queue` (многопоточная структура данных очереди, рассматриваемая ниже в этой главе), то весь ход работы можно организовать с помощью нескольких потоков, каждый из которых выполняет конкретные функции, например, как показано ниже.

- `UserRequestThread`. Обеспечивает чтение данных, введенных пользователем, возможно, из канала ввода-вывода. В программе может быть создан целый ряд потоков, по одному для каждого из одновременно работающих клиентов, запросы которых могут быть поставлены в очередь.
- `RequestProcessor`. Поток, который отвечает за выборку запросов из очереди и их обработку с предоставлением полученных выходных данных для еще одного потока.
- `ReplyThread`. Поток, обеспечивающий получение выходных данных, предназначенных для пользователя, и их отправку в ответ на запрос (если приложение является сетевым) или запись данных в локальной файловой системе или базе данных.

Если для решения подобной задачи программирования применяется несколько потоков, то сложность программы сокращается и появляется возможность обеспечить простую, эффективную и хорошо организованную реализацию. Программная реализация каждого потока, как правило, становится проще, поскольку поток предназначен для выполнения не всего задания, а лишь его части. Например, поток `UserRequestThread` просто считывает данные, введенные пользователем, и помещает их в очередь для дальнейшей обработки другим потоком и т.д. Каждый поток решает собственную подзадачу; программисту остается лишь тщательно спроектировать потоки каждого из применяемых типов, чтобы они выполняли то, что от них требуется, наилучшим образом. Принцип использования потоков для решения конкретных задач мало чем отличается от предложенной Генри Фордом модели сборочной линии для производства автомобилей.

4.2. Потоки и процессы

4.2.1. Общее определение понятия процесса

Компьютерные программы — это просто исполняемые объекты в двоичной (или другой) форме, которые находятся на диске. Программы начинают действовать лишь после их загрузки в память и вызова операционной системой. Процесс — это программа в ходе ее выполнения (в такой форме процессы иногда называют *тяжеловесными процессами*). Каждый процесс имеет собственное адресное пространство, память и стек данных, а также может использовать другие вспомогательные данные для контроля над выполнением. Операционная система управляет выполнением всех процессов в системе, выделяя каждому процессу процессорное время по определенному принципу. В ходе выполнения процесса может также происходить ветвление или порождение новых процессов для осуществления других задач, но каждый новый процесс имеет собственную память, стек данных и т.д. Вообще говоря, отдельные процессы не могут иметь доступ к общей информации, если не реализовано *межпроцессное взаимодействие* (interprocess communication — IPC) в той или иной форме.

4.2.2. Общее определение понятия потока

Потоки (иногда называемые *легковесными процессами*) подобны процессам, за исключением того, что все они выполняются в пределах одного и того же процесса, следовательно, используют один и тот же контекст. Потоки можно рассматривать как своего рода "мини-процессы", работающие параллельно в рамках основного процесса или основного потока.

Поток запускается, проходит определенную последовательность выполнения и завершается. В потоке ведется указатель команд, позволяющий следить за тем, где в настоящее время происходит его выполнение в текущем контексте. Поток может быть прерван и переведен на время в состояние ожидания (это состояние принято также называть *приостановкой* (sleeping)), в то время как другие потоки продолжают работать. Такая операция называется *возвратом управления* (yielding).

Все потоки, организованные в одном процессе, используют общее пространство данных с основным потоком, поэтому могут обмениваться информацией или взаимодействовать друг с другом с меньшими сложностями по сравнению с отдельными процессами. Потоки, как правило, выполняются параллельно. Именно распараллеливание и совместное использование данных становятся предпосылками обеспечения координации выполнения нескольких задач. Вполне естественно, что в системе с одним процессором невозможно в полном смысле слова организовать параллельное выполнение, поэтому планирование потоков происходит таким образом, чтобы каждый из них выполнялся в течение какого-то короткого промежутка времени, а затем возвращал управление другим потокам (образно говоря, снова становился в очередь на получение следующей порции процессорного времени). В ходе выполнения всего процесса каждый поток осуществляет свои собственные, отдельные задачи и передает полученные результаты другим потокам по мере необходимости.

Разумеется, переход от последовательной организации работы к параллельной связан с возникновением дополнительных сложностей. В частности, если два или несколько потоков получают доступ к одному и тому же фрагменту данных, то в зависимости от того, в какой последовательности происходит доступ, могут возникать несогласованные результаты. Неопределенность в отношении последовательности

доступа принято называть *состоянием состязания* (race condition). К счастью, в большинстве библиотек поддержки потоков предусмотрены примитивы синхронизации того или иного типа, которые позволяют диспетчеру потоков управлять выполнением и доступом.

Еще одна сложность обусловлена тем, что невозможно предоставлять всем потокам равную и справедливую долю времени выполнения. Это связано с тем, что некоторые функции устанавливают блокировки и снимают их только после завершения своего выполнения. Если функция не разработана специально для использования в потоке, то ее применение может привести к перераспределению процессорного времени в ее пользу. Такие функции принято называть *жадными* (greedy).

4.3. Поддержка потоков в языке Python

В разделе описывается использование потоков в программе на языке Python. В частности, рассматриваются ограничения потоков, обусловленные применением глобальной блокировки интерпретатора, и приводится небольшой демонстрационный сценарий.

4.3.1. Глобальная блокировка интерпретатора

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, притом 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. Иными словами, больше шансов воспользоваться преимуществами многопоточной среды имеют программы Python, ограничиваемые пропускной способностью ввода-вывода, чем программы, ограничиваемые пропускной способностью процессора.

Читатели, желающие ознакомиться с исходным кодом, а также изучить организацию главного цикла интерпретатора и глобальной блокировки интерпретатора, могут просмотреть файл `Python/ceval.c`.

4.3.2. Выход из потока

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.

В следующем разделе будут рассматриваться 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. (Чтобы ознакомиться со сведениями о том, почему это так важно, прочитайте врезку “Избегайте использования модуля thread”.)

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.

4.3.3. Доступ к потокам из программы Python

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, как показано ниже (если потоки доступны, то не появляется сообщение об ошибке).

```
>>> import thread
>>>
```

Если интерпретатор Python не был откомпилирован с включенными потоками, то попытка импорта модуля оканчивается неудачей:

```
>>> import thread
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ImportError: No module named thread
```

В таких случаях может потребоваться повторно откомпилировать интерпретатор Python, чтобы получить доступ к потокам. Для этого обычно достаточно вызвать сценарий `configure` с опцией `--with-thread`. Прочитайте файл `README` для применяемого вами дистрибутива, чтобы ознакомиться с инструкциями, касающимися того, как откомпилировать исполняемую программу интерпретатора Python с поддержкой потоков для своей системы.

4.3.4. Организация программы без применения потоков

В первом ряде примеров для демонстрации работы потоков воспользуемся функцией `time.sleep()`. Функция `time.sleep()` принимает параметр в формате с плавающей запятой и приостанавливается ("засыпает") на указанное количество секунд; иными словами, выполнение программы временно прекращается на заданное время.

Создадим два цикла во времени: приостанавливающийся на 4 секунды (функция `loop0()`) и на 2 секунды (функция `loop1()`) соответственно. (В данной программе имена `loop0` и `loop1` используются в качестве указания на то, что в конечном итоге будет создана последовательность циклов.) Если бы задача состояла в том, чтобы функции `loop0()` и `loop1()` выполнялись последовательно в однопроцессной или однопоточной программе, по аналогии со сценарием `onethr.py` в примере 4.1, то общее время выполнения составляло бы по меньшей мере 6 секунд. Между завершением работы `loop0()` и запуском `loop1()` может быть предусмотрен промежуток в 1 секунду, кроме того, в ходе выполнения могут возникнуть другие задержки, поэтому общая продолжительность работы программы может достичь 7 секунд.

Пример 4.1. Выполнение циклов в одном потоке (`onethr.py`)

В этом сценарии два цикла выполняются последовательно в однопоточной программе. Вначале должен быть завершен один цикл, чтобы мог начаться другой. Общее истекшее время представляет собой сумму значений времени, затраченных в каждом цикле.

```
1   #!/usr/bin/env python
2
3   from time import sleep, ctime
4
```

```
5    def loop0():
6        print 'start loop 0 at:', ctime()
7        sleep(4)
8        print 'loop 0 done at:', ctime()
9
10   def loop1():
11       print 'start loop 1 at:', ctime()
12       sleep(2)
13       print 'loop 1 done at:', ctime()
14
15   def main():
16       print 'starting at:', ctime()
17       loop0()
18       loop1()
19       print 'all DONE at:', ctime()
20
21   if __name__ == '__main__':
22       main()
```

В этом можно убедиться, выполнив сценарий `onethr.py` и ознакомившись со следующим выводом:

```
$ onethr.py
starting at: Sun Aug 13 05:03:34 2006
start loop 0 at: Sun Aug 13 05:03:34 2006
loop 0 done at: Sun Aug 13 05:03:38 2006
start loop 1 at: Sun Aug 13 05:03:38 2006
loop 1 done at: Sun Aug 13 05:03:40 2006
all DONE at: Sun Aug 13 05:03:40 2006
```

Теперь предположим, что работа функций `loop0()` и `loop1()` не организована по принципу приостановки, а предусматривает выполнение отдельных и независимых вычислений, предназначенных для выработки общего решения. При этом не исключена такая возможность, что выполнение этих функций можно осуществлять параллельно в целях сокращения общей продолжительности работы программы. В этом состоит идея, лежащая в основе многопоточного программирования, к рассмотрению которого мы теперь приступим.

4.3.5. Многопоточные модули Python

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. Рассмотрим эти модули отдельно и представим примеры и более крупные приложения.

Избегайте использования модуля `thread`

Мы рекомендуем использовать высокоуровневый модуль `threading` вместо модуля `thread` по многим причинам. 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.

Тем не менее мы представим некоторые примеры кода, в которых используется модуль thread, поскольку это будет способствовать изучению языка Python и многопоточной организации программ в целом. Но эти примеры представлены исключительно в учебных целях, в надежде на то, что они позволят гораздо лучше понять обоснованность рекомендации, касающейся отказа от использования модуля thread. Мы также покажем, как использовать более удобные инструменты, предусмотренные в модулях Queue и threading.

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.

3.x 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.

4.4. Модуль thread

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.

В табл. 4.1 приведен список наиболее широко используемых функций потока и методов объекта блокировки LockType.

Таблица 4.1. Модуль **thread** и объекты блокировки

Функция/метод	Описание
Функции модуля thread	
start_new_thread(*function*, *args*, *kwargs=None*)	Порождает новый поток и вызывает на выполнение функцию function с заданными параметрами args и необязательными параметрами *kwargs*
allocate_lock()	Распределяет объект блокировки LockType
exit()	Дает указание о выходе из потока
Методы объекта LockType Lock	
acquire(*wait=None*)	Предпринимает попытки захватить объект блокировки

Окончание табл. 4.1

Функция/метод	Описание
`locked()`	Возвращает `True`, если блокировка захвачена, в противном случае возвращает `False`
`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.

Возвратимся к примеру `onethr.py`, чтобы встроить в него многопоточную поддержку. В примере 4.2 представлен сценарий `mtsleepA.py`, в котором внесены некоторые изменения в функции `loop*()`:

Пример 4.2. Использование модуля `thread` (`mtsleepA.py`)

Выполняются те же циклы, что и в сценарии `onethr.py`, но на этот раз с использованием простого многопоточного механизма, предоставленного модулем `thread`. Эти два цикла выполняются одновременно (разумеется, не считая того, что менее продолжительный цикл завершается раньше), поэтому общие затраты времени определяются продолжительностью работы самого длительного потока, а не представляют собой сумму значений времени выполнения отдельно каждого цикла.

```
1   #!/usr/bin/env python
2
3   import thread
4   from time import sleep, ctime
5
6   def loop0():
7       print 'start loop 0 at:', ctime()
8       sleep(4)
9       print 'loop 0 done at:', ctime()
10
11   def loop1():
12       print 'start loop 1 at:', ctime()
13       sleep(2)
14       print 'loop 1 done at:', ctime()
15
16   def main():
17       print 'starting at:', ctime()
18       thread.start_new_thread(loop0, ())
19       thread.start_new_thread(loop1, ())
20       sleep(6)
21       print 'all DONE at:', ctime()
22
23   if __name__ == '__main__':
24       main()
```

Для функции `start_new_thread()` должны быть представлены по крайней мере первые два параметра, поэтому при ее вызове задан пустой кортеж, несмотря на то, что вызываемая на выполнение функция не требует параметров.

Выполнение этой программы показывает, что данные на выходе существенно изменились. Вместо полных затрат времени, составлявших 6 или 7 секунд, новый сценарий завершается в течение 4 секунд, что представляет собой продолжительность самого длинного цикла с добавлением небольших издержек.

```
$ mtsleepA.py
starting at: Sun Aug 13 05:04:50 2006
start loop 0 at: Sun Aug 13 05:04:50 2006
start loop 1 at: Sun Aug 13 05:04:50 2006
loop 1 done at: Sun Aug 13 05:04:52 2006
loop 0 done at: Sun Aug 13 05:04:54 2006
all DONE at: Sun Aug 13 05:04:56 2006
```

Фрагменты кода, в которых происходит приостановка на 4 с и на 2 секунды, теперь начинают выполняться одновременно, внося свой вклад в отсчет минимального значения полного времени прогона. Можно даже наблюдать за тем, как цикл 1 завершается перед циклом 0.

Еще одним важным изменением в приложении является добавление вызова `sleep(6)`. С чем связана необходимость такого добавления? Причина этого 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.

В примере 4.3 показан сценарий `mtsleepB.py`, полученный в результате следующего обновления кода, в котором добавляются блокировки и исключается дополнительная функция установки задержки. Выполнение этого сценария показывает, что полученный вывод аналогичен выводу сценария `mtsleepA.py`. Единственное различие состоит в том, что не пришлось устанавливать дополнительное время ожидания завершения работы, как в сценарии `mtsleepA.py`. С использованием блокировок мы получили возможность выйти из программы сразу после того, как оба потока завершили выполнение. При этом был получен следующий вывод:

```
$ mtsleepB.py
starting at: Sun Aug 13 16:34:41 2006
start loop 0 at: Sun Aug 13 16:34:41 2006
start loop 1 at: Sun Aug 13 16:34:41 2006
loop 1 done at: Sun Aug 13 16:34:43 2006
loop 0 done at: Sun Aug 13 16:34:45 2006
all DONE at: Sun Aug 13 16:34:45 2006
```

Пример 4.3. Использование блокировок и модуля `thread` (`mtsleepB.py`)

Очевидно, что гораздо удобнее использовать блокировки по сравнению с вызовом `sleep()` для задержки выполнения основного потока, как в сценарии `mtsleepA.py`.

```
#!/usr/bin/env python

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()
```

Рассмотрим, как в данном случае организовано применение блокировок. Обратимся к исходному коду.

Построчное объяснение

Строки 1–6

После начальной строки Unix располагаются инструкции импорта модуля `thread` и задания нескольких знакомых атрибутов модуля `time`. Вместо жесткого задания в коде отдельных функций для отсчета 4 и 2 секунд используется единственная функция `loop()`, а константы, применяемые в ее вызове, задаются в списке `loops`.

Строки 8–12

Эта функция `loop()` действует в качестве замены исключенных из кода функций `loop*()`, которые были предусмотрены в предыдущих примерах. В функцию `loop()` пришлось внести несколько небольших изменений и дополнений, чтобы обеспечить выполнение этой функцией своего назначения с помощью блокировок. Одно из очевидных изменений состоит в том, что циклы обозначены номерами, с которыми связана продолжительность приостановки. Еще одним дополнением стало применение самой блокировки. Для каждого потока распределяется и захватывается блокировка. По истечении времени, установленного функцией `sleep()`, соответствующая блокировка освобождается, тем самым основному потоку передается указание, что данный дочерний поток завершен.

Строки 14–34

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 наши лошадки выскочили из ворот на беговую дорожку примерно в одно и то же время, и, во-вторых, приходится учитывать, что захват блокировок связано 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. Затем выполнение продолжается. Проверка каждой блокировки происходит последовательно, поэтому может оказаться, что при размещении всех продолжительных циклов ближе к началу списка циклов весь ход программы будет определяться задержками в медленных циклах. В таких случаях основная часть времени ожидания будет затрачиваться в первом (или первых) цикле. К моменту освобождения этой блокировки все остальные блокировки могут уже быть разблокированы (иными словами, соответствующие потоки могут быть завершены). В результате в основном потоке выполнение

операций проверки остальных, освобожденных блокировок произойдет мгновенно, без пауз. Наконец, необходимо полностью гарантировать, чтобы в заключительной паре строк выполнение функции `main()` происходило лишь при непосредственном вызове этого сценария.

Как было указано в предыдущем основном примечании, в данной главе модуль `thread` представлен исключительно для того, чтобы ознакомить читателя с начальными сведениями о многопоточном программировании. В многопоточном приложении, как правило, следует использовать высокоуровневые модули, такие как модуль `threading`, который рассматривается в следующем разделе.

4.5. Модуль `threading`

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. В табл. 4.2 представлен список всех объектов, имеющихся в модуле `threading`.

Таблица 4.2. Объекты модуля **`threading`**

Объект	Описание
`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`[a]	Создает барьер, которого должно достичь определенное количество потоков, прежде чем всем этим потокам будет разрешено продолжить работу

3.2 [a] Новое в Python 3.2.

В этом разделе описано, как использовать класс `Thread` для реализации многопоточного режима работы. Выше в данной главе уже были приведены основные сведения о блокировках, поэтому в данном разделе не будут рассматриваться примитивы

блокировки. Кроме того, сам класс `Thread()` позволяет решать некоторые задачи синхронизации, поэтому нет необходимости явно использовать примитивы блокировки.

Потоки, функционирующие в качестве демонов

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. Как было описано в главе 2, 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. (Устаревший способ осуществления этой задачи, заключавшийся в вызове оператора `thread.setDaemon(True)`, теперь запрещен.) 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, иными словами, до тех пор, пока остаются активными какие-либо потоки, не действующие как демоны.

4.5.1. Класс `Thread`

В программе с многопоточной организацией главным инструментом является класс `Thread` модуля `threading`. Модуль `threading` поддерживает целый ряд функций, отсутствующих в модуле `thread`. В табл. 4.3 представлен список атрибутов и методов модуля `threading`.

Таблица 4.3. Атрибуты и методы объекта класса Thread

Атрибут	Описание
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)`[c]	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()`[a]	Возвращаемое имя потока
`setName(name)`[a]	Заданное имя потока
`isAlive/is_alive()`[b]	Булев флаг, указывающий, продолжает ли поток работать
`isDaemon()`[c]	Возвращает `True`, если поток выполняет функции демона, в противном случае возвращает `False`
`setDaemon(daemonic)`[c]	Задание флага работы в режиме демона равным указанному булеву значению *daemonic* (вызов должен осуществляться перед выполнением функции `start()` для потока)

[a] Обозначается как устаревший путем задания (или получения) атрибута `thread.name` или передачи его во время порождения экземпляра.

[b] Имена в так называемом *ВерблюжьемСтиле* (CamelCase) рассматриваются как устаревшие и заменяются, начиная с версии Python 2.6.

[c] Метод `is/setDaemon()` обозначается как устаревший путем задания атрибута `thread.daemon`; значение `thread.daemon` может быть также задано во время порождения экземпляра путем указания необязательного значения для демона; новое в версии Python 3.3.

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, программисты выбирают первый или третий вариант. Последний 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.

Создание экземпляра `Thread` с передачей функции

В первом примере будет лишь создан экземпляр `Thread` с передачей функции (с ее параметрами) в форме, аналогичной предыдущим примерам. Именно эта функция должна быть выполнена после передачи потоку указания, что он должен начать выполнение. Взяв за основу сценарий `mtsleepB.py` из примера 4.3 и внеся в него корректировки, необходимые для использования объектов `Thread`, получим сценарий `mtsleepC.py`, как показано в примере 4.4.

Пример 4.4. Использование модуля `threading` (`mtsleepC.py`)

В классе `Thread` из модуля `threading` предусмотрен метод `join()`, который позволяет обеспечить ожидание в основном потоке завершения текущего потока.

```
#!/usr/bin/env python

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()

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

После выполнения сценария, приведенного в примере 4.4, формируется примерно такой же вывод, как и при вызове предыдущих сценариев:

```
$ mtsleepC.py
starting at: Sun Aug 13 18:16:38 2006
start loop 0 at: Sun Aug 13 18:16:38 2006
start loop 1 at: Sun Aug 13 18:16:38 2006
loop 1 done at: Sun Aug 13 18:16:40 2006
loop 0 done at: Sun Aug 13 18:16:42 2006
all DONE at: Sun Aug 13 18:16:42 2006
```

Так что же фактически изменилось? 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.

Создание экземпляра `Thread` и передача вызываемого экземпляра класса

Подход, аналогичный передаче функции при создании потока, состоит в применении вызываемого класса и передаче его экземпляра на выполнение; в этом состоит в большей степени объектно-ориентированный способ многопоточного программирования. Такой вызываемый класс воплощает в себе среду выполнения, а это открывает намного больше возможностей по сравнению с применением функции или выбором из ряда функций. Теперь в руках у программиста оказывается вся мощь объекта класса, а не просто единственной функции или даже ряда функций, определяемого списком или кортежем.

После введения в код нового класса ThreadFunc и внесения других небольших изменений в сценарий mtsleepC.py был создан сценарий mtsleepD.py, показанный в примере 4.5.

Пример 4.5. Использование вызываемых классов (mtsleepD.py)

В этом примере передается вызываемый класс (экземпляр), в отличие от отдельной функции. Такой подход является в большей степени объектно-ориентированным по сравнению с применяемым в mtsleepC.py.

```
#!/usr/bin/env python

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()

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

После вызова на выполнение сценария mtsleepD.py формируется ожидаемый вывод:

```
$ mtsleepD.py
starting at: Sun Aug 13 18:49:17 2006
start loop 0 at: Sun Aug 13 18:49:17 2006
start loop 1 at: Sun Aug 13 18:49:17 2006
loop 1 done at: Sun Aug 13 18:49:19 2006
loop 0 done at: Sun Aug 13 18:49:21 2006
all DONE at: Sun Aug 13 18:49:21 2006
```

Так что же изменилось на этот раз? 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.

Подкласс `Thread` и создание экземпляра подкласса

В последнем вводном примере рассмотрим создание подкласса `Thread()`. Как оказалось, применяемая при этом последовательность действий весьма напоминает то, что происходит при создании вызываемого класса, как в предыдущем примере. Код создания подкласса является немного более легким для восприятия, если дело касается создания потоков (строки 29-30). Код сценария `mtsleepE.py` представлен в примере 4.6, затем показан вывод, полученный в результате выполнения этого сценария, а сравнение сценариев `mtsleepE.py` и `mtsleepD.py` оставлено в качестве упражнения для читателя.

Пример 4.6. Создание подкласса `Thread` (`mtsleepE.py`)

Вместо создания экземпляра класса `Thread` создается его подкласс. Благодаря этому открываются более широкие возможности настройки объектов многопоточной поддержки и упрощается осуществление действий по созданию потока.

```
1   #!/usr/bin/env python
2
3   import threading
4   from time import sleep, ctime
5
6   loops = (4, 2)
7
8   class MyThread(threading.Thread):
9       def __init__(self, func, args, name=''):
10          threading.Thread.__init__(self)
11          self.name = name
12          self.func = func
13          self.args = args
```

```
14
15          def run(self):
16              self.func(*self.args)
17
18      def loop(nloop, nsec):
19          print 'start loop', nloop, 'at:', ctime()
20          sleep(nsec)
21          print 'loop', nloop, 'done at:', ctime()
22
23      def main():
24          print 'starting at:', ctime()
25          threads = []
26          nloops = range(len(loops))
27
28          for i in nloops:
29              t = MyThread(loop, (i, loops[i]),
30                  loop.__name__)
31              threads.append(t)
32
33          for i in nloops:
34              threads[i].start()
35
36          for i in nloops:
37              threads[i].join()
38
39          print 'all DONE at:', ctime()'
40
41      if __name__ == '__main__':
42          main()
```

Ниже приведен вывод сценария `mtsleepE.py`. В данном случае полученные результаты вполне соответствуют ожиданию:

```
$ mtsleepE.py
starting at: Sun Aug 13 19:14:26 2006
start loop 0 at: Sun Aug 13 19:14:26 2006
start loop 1 at: Sun Aug 13 19:14:26 2006
loop 1 done at: Sun Aug 13 19:14:28 2006
loop 0 done at: Sun Aug 13 19:14:30 2006
all DONE at: Sun Aug 13 19:14:30 2006
```

Сравнивая исходный код модулей `mtsleep4` и `mtsleep5`, необходимо подчеркнуть наиболее значительные отличия: во-первых, в конструкторе подкласса `MyThread` приходится вначале вызывать конструктор базового класса (строка 9), и, во-вторых, применявшийся ранее специальный метод `__call__()` должен получить в подклассе имя `run()`.

После этого дополним класс `MyThread` некоторыми средствами формирования диагностического вывода и сохраним его в отдельном модуле `myThread` (как показано в примере 4.7). В следующих примерах этот класс будет применяться для импорта. Вместо того чтобы просто вызывать применяемые функции, сохраним результат в атрибуте экземпляра `self.res` и создадим новый метод для получения этого значения, `getResult()`.

Пример 4.7. Подкласс `MyThread` потока (`myThread.py`)

Для того чтобы повысить общность подкласса `Thread` из сценария `mtsleepE.py`, переместим этот подкласс в отдельный модуль и добавим метод `getResult()` для вызова функций, которые формируют возвращаемые значения.

```
#!/usr/bin/env python

import threading
from time import ctime

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

   def getResult(self):
      return self.res

   def run(self):
      print 'starting', self.name, 'at:', \
      ctime()
      self.res = self.func(*self.args)
      print self.name, 'finished at:', \
      ctime()
```

4.5.2. Другие функции модуля `Threading`

В модуле `Threading`, кроме различных объектов синхронизации и обеспечения многопоточной поддержки, предусмотрены также некоторые поддерживающие функции, представленные в табл. 4.4.

Таблица 4.4. Функции модуля **`threading`**

Функция	Описание
`activeCount/active_count()`[a]	Возвращает количество активных в настоящее время объектов `Thread`
`currentThread()/current_thread`[a]	Возвращает текущий объект `Thread`
`enumerate()`	Возвращает список всех активных в настоящее время объектов `Thread`
`settrace(`*`func`*`)`[b]	Задает функцию трассировки для всех потоков
`setprofile(`*`func`*`)`[b]	Задает профиль *`function`* для всех потоков
`stack_size(`*`size=0`*`)`[c]	Возвращает размер стека вновь созданных потоков; с учетом потоков, которые будут создаваться в дальнейшем, может быть задан необязательный параметр *`size`*

[a] Имена в так называемом ВерблюжьемСтиле рассматриваются как устаревшие и заменяются, начиная с версии Python 2.6.

[b] Новое в версии Python 2,3.

[c] Псевдоним метода `thread.stack_size()`. Метод и его псевдоним впервые введены в версии Python 2.5.

4.6. Сравнение однопоточного и многопоточного выполнения

В сценарии `mtfacfib.py`, представленном в примере 4.8, происходит сравнение хода выполнения рекурсивных функций, включая функции вычисления числа Фибоначчи, факториала и суммы. В этом сценарии все три функции выполняются в однопоточном режиме. После этого та же задача решается с использованием потоков, что позволяет показать одно из преимуществ применения среды многопоточной поддержки.

Пример 4.8. Функции вычисления числа Фибоначчи, факториала и суммы (`mtfacfib.py`)

В этом многопоточном приложении выполняются три отдельные рекурсивные функции, сначала в однопоточном режиме, а затем с применением альтернативной организации работы с несколькими потоками.

```
#!/usr/bin/env python

from myThread import MyThread
from time import ctime, sleep

def fib(x):
    sleep(0.005)
    if x < 2: return 1
    return (fib(x-2) + fib(x-1))

def fac(x):
    sleep(0.1)
    if x < 2: return 1
    return (x * fac(x-1))

def sum(x):
    sleep(0.1)
    if x < 2: return 1
    return (x + sum(x-1))

funcs = [fib, fac, sum]
n = 12

def main():
    nfuncs = range(len(funcs))

    print '*** SINGLE THREAD'
    for i in nfuncs:
        print 'starting', funcs[i].__name__, 'at:', \
            ctime()
        print funcs[i](n)
        print funcs[i].__name__, 'finished at:', \
            ctime()

    print '\n*** MULTIPLE THREADS'
    threads = []
    for i in nfuncs:
        t = MyThread(funcs[i], (n,),
            funcs[i].__name__)
        threads.append(t)
```

```
41
42          for i in nfuncs:
43              threads[i].start()
44
45          for i in nfuncs:
46              threads[i].join()
47              print threads[i].getResult()
48
49          print 'all DONE'
50
51      if __name__ == '__main__':
52          main()
```

Выполнение в однопоточном режиме сводится к тому, что функции вызываются одна за другой и после завершения их работы сразу же отображаются полученные результаты вызова.

Если же выполнение функций происходит в многопоточном режиме, то результат не отображается немедленно. Желательно, чтобы класс `MyThread` был настолько общим, насколько это возможно (способным выполнять вызываемые функции, которые формируют и не формируют вывод), поэтому вызов метода `getResult()` откладывается до самого конца, что позволяет показать значения, возвращенные в каждом вызове функции, после того, как все будет сделано.

В данном примере вызовы всех функций выполняются очень быстро (вернее, исключением может стать вызов функции вычисления числа Фибоначчи), поэтому, как можно заметить, мы были вынуждены добавить вызовы `sleep()` к каждой функции для замедления процесса выполнения, чтобы можно было убедиться в том, что многопоточная организация действительно способствует повышению производительности, если в разных потоках применяются функции с различным временем выполнения. Безусловно, на практике дополнять функции вызовами `sleep()` нет необходимости. Так или иначе, получим такой вывод:

```
$ mtfacfib.py
*** SINGLE THREAD
starting fib at: Wed Nov 16 18:52:20 2011
233
fib finished at: Wed Nov 16 18:52:24 2011
starting fac at: Wed Nov 16 18:52:24 2011
479001600
fac finished at: Wed Nov 16 18:52:26 2011
starting sum at: Wed Nov 16 18:52:26 2011
78
sum finished at: Wed Nov 16 18:52:27 2011

*** MULTIPLE THREADS
starting fib at: Wed Nov 16 18:52:27 2011
starting fac at: Wed Nov 16 18:52:27 2011
starting sum at: Wed Nov 16 18:52:27 2011

fac finished at: Wed Nov 16 18:52:28 2011
sum finished at: Wed Nov 16 18:52:28 2011
fib finished at: Wed Nov 16 18:52:31 2011
233
479001600
78
all DONE
```

4.7. Практическое применение многопоточной обработки

До сих пор были представлены 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. По этой причине мы рассмотрим пример приложения первого типа и в качестве дальнейшего упражнения попытаемся перенести его в версию Python 3, чтобы можно было понять, что с этим связано.

4.7.1. Пример ранжирования книг

Сценарий `bookrank.py`, показанный в примере 4.9, весьма прост. Он выполняет переход на сайт интернет-торговли Amazon и запрашивает текущее ранжирование книг, написанных вашим покорным слугой. В рассматриваемом примере кода представлены функция `getRanking()`, в которой используется регулярное выражение для извлечения и возврата текущего ранжирования, и функция `showRanking()`, которая отображает результаты для пользователя.

Следует отметить, что согласно условиям использования "*компания Amazon предоставляет пользователю сайта ограниченные права доступа к сайту и использования его в личных целях, но не позволяет загружать содержимое сайта (исключая кэширование страниц) либо изменять его или любую его часть без явно выраженного письменного согласия Amazon*." Задача рассматриваемого приложения состоит в том, чтобы выбрать данные о текущем ранжировании конкретной книги, а затем отбросить остальные сведения о ранжировании; в данном случае даже кеширование страницы не применяется.

В примере 4.9 представлена первая (и почти последняя) попытка создания сценария `bookrank.py`, который не относится к многопоточной версии.

Пример 4.9. Программа извлечения с веб-страницы данных о ранжировании книг (`bookrank.py`)

В этом сценарии выполняются вызовы, обеспечивающие загрузку информации о ранжировании книг через отдельные потоки.

```
1   #!/usr/bin/env python
2
3   from atexit import register
4   from re import compile
5   from threading import Thread
```

```
6   from time import ctime
7   from urllib2 import urlopen as uopen
8
9   REGEX = compile('#([\d,]+) in Books ')
10  AMZN = 'http://amazon.com/dp/'
11  ISBNs = {
12       '0132269937': 'Core Python Programming',
13       '0132356139': 'Python Web Development with Django',
14       '0137143419': 'Python Fundamentals',
15: }
16:
17: def getRanking(isbn):
18:     page = uopen('%s%s' % (AMZN, isbn)) #или str.format()
19:     data = page.read()
20:     page.close()
21:     return REGEX.findall(data)[0]
22:
23: def _showRanking(isbn):
24:     print '- %r ranked %s' % (
25:         ISBNs[isbn], getRanking(isbn))
26:
27: def _main():
28:     print 'At', ctime(), 'on Amazon...'
29:     for isbn in ISBNs:
30:         _showRanking(isbn)
31:
32: @register
33: def _atexit():
34:     print 'all DONE at:', ctime()
35:
36: if __name__ == '__main__':
37:     main()
```

Построчное объяснение

Строки 1–7

Это строки запуска и импорта. Для определения того, когда будет завершено выполнение сценария, используется функция `atexit.register()` (почему это сделано, будет описано ниже). Кроме того, для работы с шаблоном, с помощью которого извлекаются сведения о ранжировании книг со страниц с описанием товаров на сайте Amazon, применяется функция поддержки регулярных выражений `re.compile()`. Затем результаты импорта `threading.Thread` сохраняются для использования в намеченном на будущее усовершенствованном варианте сценария (об этом немного позже), вызывается метод `time.ctime()` для получения строки с текущей отметкой времени, а для получения доступа к каждой ссылке применяется метод `urllib2.urlopen()`.

Строки 9–15

В этом сценарии используются три константы. Первой из них является `REGEX`, объект регулярного выражения (полученный путем компиляции шаблона регулярного выражения, который согласуется с данными по ранжированию книги); вторая константа — `AMZN`, префикс каждой ссылки на товары Amazon. За этим префиксом следует ISBN (International Standard Book Number) искомой книги; ISBN служит для

книги уникальным обозначением и позволяет отличить ее от других. Предусмотрены два стандарта ISBN: ISBN-10, с десятисимвольным обозначением, и ISBN-13, принятый позднее стандарт, в котором применяются тринадцать символов. В настоящее время поисковая система Amazon распознает ISBN обоих типов, поэтому для краткости будем использовать только ISBN-10. Искомые ISBN хранятся в словаре `ISBNs` (который представляет собой третью константу) наряду с соответствующими названиями книг.

Строки 17–21

Функция `getRanking()` предназначена для получения ISBN, создания конечного значения URL, которое применяется для доступа к серверам Amazon, а затем вызова для этого URL метода `urllib2.urlopen()`. Для соединения воедино компонентов значения URL используется оператор форматирования строки (в строке 18), но если работа ведется с версией Python 2.6 или последующей версией, то можно также попытаться воспользоваться методом `str.format()`, например `'{0}{1}'.format(AMZN,isbn)`.

После получения полного URL вызывается метод `urllib2.urlopen()` (в данном сценарии в качестве него применяется сокращение `uopen()`), который в случае успеха возвращает файловый объект после ответа веб-сервера. Затем происходит вызов функции `read()` для загрузки всей веб-страницы, и файловый объект закрывается. Если регулярное выражение составлено правильно, то при его использовании должно происходить одно и только одно сопоставление, поэтому достаточно извлечь необходимый результат из сформированного списка (все остальные результаты будут пропущены) и возвратить его в вызывающую функцию.

Строки 23–25

Функция `_showRanking()` представляет собой всего лишь короткий фрагмент кода, в котором берется ISBN, осуществляется поиск названия книги, которую представляет этот ISBN, вызывается метод `getRanking()` для получения текущего ранга книги на веб-сайте Amazon, после чего ISBN и название передаются пользователю. В имени этого метода применяется префикс в виде одного знака подчеркивания. Этот префикс служит в качестве указания, что данная функция является специальной, предназначена для использования только в данном модуле и не должна быть импортирована в каком-либо другом приложении в составе библиотечного или вспомогательного модуля.

Строки 27–30

Функция `_main()` также относится к категории специальных и вызывается на выполнение, только если данный модуль вызван непосредственно из командной строки (а не импортируется для использования другим модулем). Происходит отображение времени начала и окончания (чтобы пользователь мог определить, сколько времени потребовалось для выполнения всего сценария), затем вызывается функции `_showRanking()` применительно к каждому ISBN для поиска и отображения данных о текущем ранжировании каждой книги на сайте Amazon.

Строки 32–37

В этих строках выполняются действия, которые прежде нами не рассматривались. Рассмотрим назначение функции `atexit.register()`. Такие функции принято называть декораторами. Декораторы — одна из разновидностей служебных функций.

В данном случае с помощью указанной функции происходит регистрация функции выхода в интерпретаторе Python. Это равносильно запросу, чтобы интерпретатор вызвал некоторую специальную функцию непосредственно перед завершением сценария. (Вместо вызова декоратора можно также применить конструкцию `register(_atexit())`).

Рассмотрим причины использования декоратора в данном коде. Прежде всего необходимо отметить, что можно было бы вполне обойтись без него. Оператор вывода может быть размещен в последней части функции `_main()`, в строках 27–31, но в действительности при этом организация программ оставляла бы желать лучшего. Кроме того, здесь демонстрируется пример применения функционального средства, без которого при определенных обстоятельствах нельзя было бы обойтись в приложении, применяемом на производстве. Предполагается, что читатель сам сможет определить назначение строк 36-37, поэтому перейдем к описанию полученного вывода:

```
$ python bookrank.py
At Wed Mar 30 22:11:19 2011 PDT on Amazon...
- 'Core Python Programming' ranked 87,118
- 'Python Fundamentals' ranked 851,816
- 'Python Web Development with Django' ranked 184,735
all DONE at: Wed Mar 30 22:11:25 2011
```

Заслуживает внимания то, что в данном примере разделены процессы получения данных (`getRanking()`) и их отображения (`_showRanking()` и `_main()`), что позволяет при желании вместо отображения результатов для пользователя с помощью терминала предусмотреть какой-то другой способ обработки вывода. На практике может потребоваться отправить эти данные назад с помощью веб-шаблона, сохранить в базе данных, вывести в виде текста на экран мобильного телефона и т.д. Если бы весь этот код был помещен в одну функцию, то было бы сложнее обеспечить его повторное использование и (или) отправить по другому назначению.

Кроме того, если компания Amazon изменит компоновку своих страниц с описанием товаров, то может потребоваться лишь изменить регулярное выражение, предназначенное для выборки данных с веб-страниц, чтобы по-прежнему иметь возможность извлекать сведения о книгах. Следует также отметить, что в этом простом примере вполне оправдывает себя способ обработки данных с помощью регулярного выражения (который может быть даже заменен простыми традиционными операциями работы со строками), но в какое-то время может потребоваться более мощный синтаксический анализатор разметки, такой как HTMLParser из стандартной библиотеки, а возможно, нельзя будет обойтись без таких инструментов сторонних производителей, как BeautifulSoup, html5lib или lxml. (Некоторые из этих инструментов будут продемонстрированы в главе 9.)

Добавление в программу средств многопоточной поддержки

Очевидно, что приведенный выше пример все еще относится к категории несложных однопоточных программ. Теперь перейдем к внесению изменений в приложение, чтобы в нем вместо этого использовались потоки. Это приложение, ограничиваемое пропускной способностью ввода-вывода, поэтому вполне подходит для применения в нем многопоточной организации. Для упрощения на первых порах мы не будем использовать классы и объектно-ориентированное программирование; вместо этого в программе будет применяться непосредственно метод `threading.Thread`, поэтому

данный пример в большей степени должен напоминать сценарий mtsleepC.py, чем любой из последующих примеров. Дополнением является лишь то, что в приложении создаются потоки, которые немедленно запускаются.

Возьмем за основу ранее разработанное приложение и изменим вызов _showRanking(isbn) следующим образом:

```
Thread(target=_showRanking, args=(isbn,)).start().
```

Получен именно такой результат, который требуется! Теперь в нашем распоряжении имеется окончательная версия сценария bookrank.py, которая показывает, что это приложение (как правило) выполняется быстрее, чем предыдущее, благодаря дополнительному распараллеливанию. Тем не менее быстродействие приложения ограничивается тем, насколько быстро будет получен ответ, обработка которого потребовала больше всего времени.

```
$ python bookrank.py
At Thu Mar 31 10:11:32 2011 on Amazon...
- 'Python Fundamentals' ranked 869,010
- 'Core Python Programming' ranked 36,481
- 'Python Web Development with Django' ranked 219,228
all DONE at: Thu Mar 31 10:11:35 2011
```

Как показывает полученный вывод, вместо шести секунд, в течение которых выполнялась однопоточная версия, для многопоточной достаточно трех. Кроме того, важно отметить, что в окончательно полученном выводе последовательность результатов может изменяться в зависимости от времени завершения работы потоков, в отличие от однопоточной версии. В версии, в которой не применялась многопоточная организация, последовательность расположения результатов всегда определяется ключами словаря, а теперь все запросы выполняются параллельно и вывод формируется в той последовательности, которая определяется значениями времени завершения отдельных потоков.

В предыдущих примерах (в сценариях mtsleep*X.py*) применительно ко всем потокам вызывался метод Thread.join() для блокирования выполнения до завершения каждого потока. Это равносильно блокированию дальнейшей работы основного потока до завершения всех потоков, поэтому инструкция вывода на печать “all DONE at” вызывается после того, как действительно закончится вся работа.

В указанных примерах не было необходимости применять метод join() ко всем потокам, поскольку ни один из потоков не функционировал в качестве демона. В основном потоке не происходит выход из сценария до тех пор, пока не произойдет успешное или неудачное завершение всех порожденных потоков. Опираясь на эти рассуждения, мы удалили все вызовы join() из сценария mtsleepF.py. Тем не менее следует учитывать, что неправильно было бы отображать строку “all done” (все сделано) на том же этапе выполнения сценария, как и прежде.

Строка “all done” должна быть выведена в основном потоке, т.е. до завершения дочерних потоков, чтобы не было необходимости вызывать оператор печати, выше функции _main(). Этот оператор print можно поместить в одно из двух мест в сценарии: после строки 37, где происходит возврат из функции _main() (самая последняя строка, выполняемая в сценарии), или в месте, которое определяется в связи с использованием метода atexit.register() для регистрации функции выхода. Эта тема в настоящей книге еще не рассматривалась, и в дальнейшем мы к ней обязательно вернемся, но именно здесь удобно впервые затронуть вопрос регистрации

функций. Важно также, что для регистрации функций применяется интерфейс, который останется неизменным после перехода от Python 2 к Python 3 (это — тема следующего раздела).

Перенос приложения в версию Python 3

3.x

Теперь перейдем к рассмотрению еще одного варианта данного сценария, который предназначен для работы с версией Python 3. С этой темой необходимо ознакомиться, изучая пути переноса проектов и приложений из текущей версии интерпретатора Python в последующую версию. К счастью, эту работу не требуется выполнять вручную, поскольку уже предусмотрены необходимые инструменты, одним из которых является инструмент `2to3`. Вообще говоря, предусмотрены два способа его использования:

```
$ 2to3 foo.py    # в выводе показаны только различия
$ 2to3 -w foo.py # переопределяет с помощью кода версии 3.x
```

В первой команде инструмент `2to3` лишь отображает различия между исходным сценарием в версии 2.x и сформированным с его помощью эквивалентом для версии 3.x. Флаг `-w` служит для инструмента `2to3` указанием, что исходный сценарий должен быть перезаписан вновь полученным сценарием для версии 3.x, а сценарий для версии 2.x переименован в `foo.py.bak`.

Вызовем на выполнение инструмент `2to3` применительно к файлу сценария `bookrank.py` с перезаписью существующего файла. Предусмотрен не только вывод различий; сохраняется также новая версия, как уже было сказано:

```
$ 2to3 -w bookrank.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
--- bookrank.py (original)
+++ bookrank.py (refactored)
@@ -4,7 +4,7 @@
 from re import compile
 from threading import Thread
 from time import ctime
-from urllib2 import urlopen as uopen
+from urllib.request import urlopen as uopen

 REGEX = compile('#([\d,]+) in Books ')
 AMZN = 'http://amazon.com/dp/'
@@ -21,17 +21,17 @@
     return REGEX.findall(data)[0]

 def _showRanking(isbn):
-    print '- %r ranked %s' % (
-        ISBNs[isbn], getRanking(isbn))
+    print('- %r ranked %s' % (
+        ISBNs[isbn], getRanking(isbn)))


 def _main():
-    print 'At', ctime(), 'on Amazon...'
```

```
+    print('At', ctime(), 'on Amazon...')
     for isbn in ISBNs:
         Thread(target=_showRanking, args=(isbn,)).start()#_showRanking(isbn)

 @register
 def _atexit():
-    print 'all DONE at:', ctime()
+    print('all DONE at:', ctime())

 if __name__ == '__main__':
     _main()
RefactoringTool: Files that were modified:
RefactoringTool: bookrank.py
```

Следующий шаг читатели могут рассматривать как необязательный. Достаточно лишь отметить, что в нем рассматриваемые файлы были переименованы в `bookrank.py` и `bookrank3.py` с использованием команд POSIX (пользователи компьютеров с операционной системой Windows должны использовать команду `ren`):

```
$ mv bookrank.py bookrank3.py
$ mv bookrank.py.bak bookrank.py
```

Разумеется, было бы желательно, чтобы преобразование сценария для использования в новой версии интерпретатора прошло идеально, чтобы не пришлось ни о чем заботиться, приступая к работе со сценарием нового поколения. Однако в данном случае произошло нечто непредвиденное и в каждом потоке возникает исключение (приведенный вывод относится только к одному потоку; нет смысла показывать результаты для других потоков, поскольку они являются такими же):

```
$ python3 bookrank3.py
Exception in thread Thread-1:
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/
      3.2/lib/python3.2/threading.py", line 736, in
      _bootstrap_inner
    self.run()
  File "/Library/Frameworks/Python.framework/Versions/
      3.2/lib/python3.2/threading.py", line 689, in run
    self._target(*self._args, **self._kwargs)
  File "bookrank3.py", line 25, in _showRanking
    ISBNs[isbn], getRanking(isbn)))
  File "bookrank3.py", line 21, in getRanking
    return REGEX.findall(data)[0]
TypeError: can't use a string pattern on a bytes-like object
        :
```

Что же случилось? По-видимому, проблема заключается в том, что регулярное выражение представлено в виде строки Юникода, тогда как данные, полученные с помощью метода `read()` файлового объекта (возвращенного функцией `urlopen()`), имеют вид строки `ASCII/bytes`. Чтобы исправить эту ошибку, откомпилируем вместо текстовой строки объект `bytes`. Для этого внесем изменения в строку 9, чтобы в методе `re.compile()` производилась компиляция строки `bytes` (добавим строку `bytes`). Для этого добавим обозначение `b` строки `bytes` непосредственно перед открывающей кавычкой следующим образом:

```
REGEX = compile(b'#([\d,]+) in Books ')
Now let's try it again:
$ python3 bookrank3.py
At Sun Apr  3 00:45:46 2011 on Amazon...
- 'Core Python Programming' ranked b'108,796'
- 'Python Web Development with Django' ranked b'268,660'
- 'Python Fundamentals' ranked b'969,149'
all DONE at: Sun Apr  3 00:45:49 2011
```

Опять что-то не так! Что же случилось теперь? Безусловно, результат стал немного лучше (нет ошибок), но выглядит странно. В данных ранжирования, полученных с помощью регулярных выражений, после передачи в функцию `str()` отображаются символы `b` и кавычки. Для устранения этого недостатка первым побуждением может стать попытка применить операцию получения среза строки, которая также выглядит довольно неуклюже:

```
>>> x = b'xxx'
>>> repr(x)
"b'xxx'"
>>> str(x)
"b'xxx'"
>>> str(x)[2:-1]
'xxx'
```

Тем не менее более подходящий вариант состоит в применении операции преобразования данных в действительное значение (строка в Юникоде, возможно, с использованием UTF-8):

```
>>> str(x, 'utf-8')
'xxx'
```

Для реализации этого решения в текущем сценарии внесем аналогичное изменение в строку 53, чтобы она выглядела следующим образом:

```
return str(REGEX.findall(data)[0], 'utf-8')
```

После этого вывод сценария для версии Python 3 полностью совпадает с тем, что получен в сценарии Python 2:

```
$ python3 bookrank3.py
At Sun Apr  3 00:47:31 2011 on Amazon...
- 'Python Fundamentals' ranked 969,149
- 'Python Web Development with Django' ranked 268,660
- 'Core Python Programming' ranked 108,796
all DONE at: Sun Apr  3 00:47:34 2011
```

Вообще говоря, практика показывает, что перенос сценария из версии 2.x в версию 3.x осуществляется по аналогичному принципу: необходимо убедиться, что код проходит все тесты модульности и интеграции, провести основное преобразование с использованием инструмента `2to3` (и других инструментов), а затем устранить возможные расхождения, добиваясь того, чтобы код успешно выполнялся и проходил такие же проверки, как и исходный сценарий. Попробуем повторить это упражнение снова на следующем примере, в котором демонстрируется использование синхронизации с помощью потоков.

4.7.2. Примитивы синхронизации

В основной части этой главы рассматривались основные концепции многопоточной организации и было показано, как использовать многопоточность в приложениях Python. Однако в этом изложении не затрагивался 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. Как уже было сказано в этой главе, 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. (С дополнительными сведениями о состояниях состязания можно ознакомиться на странице `http://en.wikipedia.org/wiki/Race_condition`.)

В таких случаях возникает необходимость обеспечения синхронизации. Синхронизация должна использоваться, если к какому-то из критических участков кода могут подойти одновременно несколько потоков (см. `http://en.wikipedia.org/wiki/Critical_section`), 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 (см. `http://en.wikipedia.org/wiki/Synchronization_(computer_science)`), 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.

4.7.3. Пример применения блокировки

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.

Рассмотрим, с чем связана необходимость применения блокировок. Сценарий `mtsleepF.py` представляет собой приложение, в котором происходит порождение случайным образом выбранного количества потоков, в каждом из которых осуществляется выход после завершения работы. Рассмотрим следующий базовый фрагмент исходного кода (*для* версии Python 2):

```
from atexit import register
from random import randrange
from threading import Thread, currentThread
from time import sleep, ctime

class CleanOutputSet(set):
    def __str__(self):
        return ', '.join(x for x in self)

loops = (randrange(2,5) for x in xrange(randrange(3,7)))
remaining = CleanOutputSet()

def loop(nsec):
    myname = currentThread().name
    remaining.add(myname)
    print '[%s] Started %s' % (ctime(), myname)
    sleep(nsec)
    remaining.remove(myname)
    print '[%s] Completed %s (%d secs)' % (
        ctime(), myname, nsec)
    print '    (remaining: %s)' % (remaining or 'NONE')

def _main():
    for pause in loops:
        Thread(target=loop, args=(pause,)).start()

@register
def _atexit():
    print 'all DONE at:', ctime()
```

Более подробное построчное описание кода мы приведем вслед за окончательным вариантом сценария, в котором применяются блокировки, но вкратце можно отметить, что сценарий `mtsleepF.py` по сути *лишь* дополняет приведенные ранее примеры. Как и в примере сценария `bookrank.py`, немного упростим код. *Для* этого отложим на время применение средств объектно-ориентированного программирования, исключим список объектов потока и операции `join()` с потоками и снова введем в действие метод `atexit.register()` (по тем же причинам, как и в коде `bookrank.py`).

Проведем еще одно небольшое изменение по отношению к приведенным ранее примерам `mtsleep`*X*`.py`. Вместо жесткого задания пары операций приостановки циклов/потоков на 4 и 2 секунды соответственно, внесем некую неопределенность, создавая случайным образом от 3 до 6 потоков, каждый из которых может приостанавливаться на какой-то промежуток времени от 2 до 4 секунд.

В этом сценарии применяются также некоторые новые средства, причем наиболее заметным среди них является использование множества для хранения имен оставшихся потоков, которые все еще функционируют. Причина, по которой создается подкласс объекта множества вместо непосредственного использования самого класса, состоит в том, что это позволяет продемонстрировать еще один вариант использования множества, в котором изменяется применяемое по умолчанию для вывода строковое представление множества.

При использовании операции вывода содержимого множества формируются примерно такие результаты: `set([X, Y, Z,...])`. Однако это не очень удобно, поскольку потенциальные пользователи нашего приложения не знают (и не должны знать) о том, что такое множества и для чего они используются в программе. Таким образом, необходимо вместо этого вывести данные, которые выглядят примерно как `X, Y, Z, ...`. Именно по этой причине мы создали подкласс класса `set` и реализовали его метод `__str__()`.

После этого изменения, при условии, что вся остальная часть сценария будет работать правильно, должен сформироваться аккуратный вывод, который будет иметь подходящее выравнивание:

```
$ python mtsleepF.py
[Sat Apr  2 11:37:26 2011] Started Thread-1
[Sat Apr  2 11:37:26 2011] Started Thread-2
[Sat Apr  2 11:37:26 2011] Started Thread-3
[Sat Apr  2 11:37:29 2011] Completed Thread-2 (3 secs)
    (remaining: Thread-3, Thread-1)
[Sat Apr  2 11:37:30 2011] Completed Thread-1 (4 secs)
    (remaining: Thread-3)
[Sat Apr  2 11:37:30 2011] Completed Thread-3 (4 secs)
    (remaining: NONE)
all DONE at: Sat Apr  2 11:37:30 2011
However, if you're unlucky, you might get strange output such as this pair of example
executions:
$ python mtsleepF.py
[Sat Apr  2 11:37:09 2011] Started Thread-1
 [Sat Apr  2 11:37:09 2011] Started Thread-2
[Sat Apr  2 11:37:09 2011] Started Thread-3
[Sat Apr  2 11:37:12 2011] Completed Thread-1 (3 secs)
 [Sat Apr  2 11:37:12 2011] Completed Thread-2 (3 secs)
    (remaining: Thread-3)
    (remaining: Thread-3)
[Sat Apr  2 11:37:12 2011] Completed Thread-3 (3 secs)
    (remaining: NONE)
all DONE at: Sat Apr  2 11:37:12 2011

$ python mtsleepF.py
[Sat Apr  2 11:37:56 2011] Started Thread-1
[Sat Apr  2 11:37:56 2011] Started Thread-2
 [Sat Apr  2 11:37:56 2011] Started Thread-3
[Sat Apr  2 11:37:56 2011] Started Thread-4

[Sat Apr  2 11:37:58 2011] Completed Thread-2 (2 secs)
 [Sat Apr  2 11:37:58 2011] Completed Thread-4 (2 secs)
    (remaining: Thread-3, Thread-1)
    (remaining: Thread-3, Thread-1)

[Sat Apr  2 11:38:00 2011] Completed Thread-1 (4 secs)
```

```
    (remaining: Thread-3)
[Sat Apr  2 11:38:00 2011] Completed Thread-3 (4 secs)
    (remaining: NONE)
all DONE at: Sat Apr  2 11:38:00 2011
```

Что же произошло? Во-первых, очевидно, что результаты далеко не однородны (поскольку возможность выполнять операции ввода-вывода параллельно предоставлена сразу нескольким потокам). Подобное чередование результирующих данных можно было также наблюдать и в некоторых примерах приведенного ранее кода. Во-вторых, обнаруживается такая проблема, что два потока изменяют значение одной и той же переменной (множества, содержащего имена оставшихся потоков).

Операции ввода-вывода и операции доступа к одной и той же структуре данных входят в состав критических разделов кода, поэтому необходимы блокировки, которые смогли бы воспрепятствовать одновременному вхождению в эти разделы нескольких потоков. Чтобы ввести в действие блокировки, необходимо добавить строку кода для импорта объекта `Lock` (или `RLock`), создать объект блокировки и внести дополнения или изменения в код, позволяющие применять блокировки в нужных местах:

```
from threading import Thread, Lock, currentThread
lock = Lock()
```

Теперь необходимо обеспечить установку и снятие блокировки. В следующем коде показаны вызовы `acquire()` и `release()`, которые должны быть введены в функцию `loop()`:

```
def loop(nsec):
    myname = currentThread().name
    lock.acquire()
    remaining.add(myname)
    print '[%s] Started %s' % (ctime(), myname)
    lock.release()
    sleep(nsec)
    lock.acquire()
    remaining.remove(myname)
    print '[%s] Completed %s (%d secs)' % (
        ctime(), myname, nsec)
    print '    (remaining: %s)' % (remaining or 'NONE')
    lock.release()
```

После внесения изменений полученный вывод уже не должен содержать прежних искажений:

```
$ python mtsleepF.py
[Sun Apr  3 23:16:59 2011] Started Thread-1
[Sun Apr  3 23:16:59 2011] Started Thread-2
[Sun Apr  3 23:16:59 2011] Started Thread-3
[Sun Apr  3 23:16:59 2011] Started Thread-4
[Sun Apr  3 23:17:01 2011] Completed Thread-3 (2 secs)
    (remaining: Thread-4, Thread-2, Thread-1)
[Sun Apr  3 23:17:01 2011] Completed Thread-4 (2 secs)
    (remaining: Thread-2, Thread-1)
```

```
[Sun Apr  3 23:17:02 2011] Completed Thread-1 (3 secs)
    (remaining: Thread-2)
[Sun Apr  3 23:17:03 2011] Completed Thread-2 (4 secs)
    (remaining: NONE)
all DONE at: Sun Apr  3 23:17:03 2011
```

Исправленная (и окончательная) версия `mtsleepF.py` показана в примере 4.10.

Пример 4.10. Сценарий, в котором предусмотрены блокировки и введены дополнительные средства случайного выбора (`mtsleepF.py`)

В этом примере демонстрируется использование блокировок и других инструментов обеспечения многопоточного функционирования.

```
#!/usr/bin/env python

from atexit import register
from random import randrange
from threading import Thread, Lock, currentThread
from time import sleep, ctime

class CleanOutputSet(set):
    def __str__(self):
        return ', '.join(x for x in self)

lock = Lock()
loops = (randrange(2,5) for x in xrange(randrange(3,7)))
remaining = CleanOutputSet()

def loop(nsec):
    myname = currentThread().name
    lock.acquire()
    remaining.add(myname)
    print '[%s] Started %s' % (ctime(), myname)
    lock.release()
    sleep(nsec)
    lock.acquire()
    remaining.remove(myname)
    print '[%s] Completed %s (%d secs)' % (
        ctime(), myname, nsec)
    print '    (remaining: %s)' % (remaining or 'NONE')
    lock.release()

def _main():
    for pause in loops:
        Thread(target=loop, args=(pause,)).start()

@register
def _atexit():
    print 'all DONE at:', ctime()

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

Построчное объяснение

Строки 1–6

2.6 Это обычные строки запуска и импорта. Следует учитывать, что начиная с версии 2.6 метод `threading.currentThread()` переименован в `threading.current_thread()`, но первый компонент имени метода остался неизменным в целях обеспечения обратной совместимости.

Строки 8–10

Это подкласс множества, о котором речь шла выше. Он содержит реализацию метода `__str__()`, который позволяет вместо формата вывода, применяемого по умолчанию, перейти к формату, представляющему собой строку элементов, разделенную запятыми.

Строки 12–14

В состав применяемых глобальных переменных входят блокировка, экземпляр позаимствованного из предыдущего кода откорректированного множества и случайно выбранного числа потоков (от трех до шести), каждый из которых останавливается (вернее, приостанавливается) на время от двух до четырех секунд.

Строки 16–28

Функция `loop()` сохраняет имя текущего потока, в котором она выполняется, затем захватывает блокировку, что позволяет сделать неразрывным (атомарным) следующий ряд операций: добавление имени к множеству `remaining` и формирование вывода с указанием на начало потока (с этого момента больше никакой другой поток не сможет войти в критический участок кода). После освобождения блокировки этот поток приостанавливается на время в секундах, выбранное случайным образом, затем повторно захватывает блокировку, чтобы сформировать свой окончательный вывод перед ее освобождением.

Строки 30–39

Функция `_main()` выполняется, только если этот сценарий не был импортирован для использования в другой программе. Назначение этой функции состоит в том, что она порождает и запускает каждый из потоков. Как было указано выше, применяется метод `atexit.register()` для регистрации функции `_atexit()`, которую интерпретатор может выполнить перед выходом из программы.

В качестве альтернативного по отношению к варианту, в котором поддерживается собственное множество выполняющихся в настоящее время потоков, можно применить вариант с использованием метода `threading.enumerate()`, который возвращает список всех потоков, работающих в настоящее время (этот список включает потоки, действующие в качестве демона, но, безусловно, в него не входят потоки, которые еще не запущены). В рассматриваемом примере этот вариант не используется, поскольку его применение приводит к созданию двух дополнительных потоков, которые приходится удалять для сокращения объема вывода, в том числе текущего потока (поскольку он еще не завершен) и основного потока (который так или иначе не должен быть показан).

Не следует также забывать, что вместо оператора форматирования строки можно использовать метод `str.format()`, при условии, что для работы применяется версия Python 2.6 или более новая (включая версии 3.x). Иными словами, следующая инструкция **`print`**:

```
print '[%s] Started %s' % (ctime(), myname)
```

2.6-2.7 может быть заменена в версии 2.6 и последующей таким вызовом:

```
print '[{0}] Started {1}'.format(ctime(), myname)
```

3.x или (в версии 3.x) таким вызовом `print()`:

```
print('[{0}] Started {1}'.format(ctime(), myname))
```

Если задача состоит лишь в том, чтобы подсчитать число потоков, выполняющихся в настоящее время, то вместо этого можно использовать метод `threading.activeCount()` (переименованный в `active_count()`, начиная с версии 2.6).

Применение управления контекстом

2.5 Программисты, работающие в версии Python 2.5 и более новой, могут воспользоваться еще одним вариантом, который вообще не требует вызова методов `acquire()` и `release()` применительно к блокировке, что способствует еще большему упрощению кода. С помощью инструкции `with` можно вводить в действие для любого объекта диспетчер контекста, который обеспечит вызов метода `acquire()` перед входом в критический участок и метода `release()`, когда этот блок завершит выполнение.

Диспетчеры контекста предусмотрены для всех объектов модуля `threading`, таких как `Lock`, `RLock`, `Condition`, `Semaphore` и `BoundedSemaphore`, а это означает, что для работы с этими объектами может применяться инструкция **`with`**. С помощью инструкции **`with`** можно еще больше упростить код функции `loop()` следующим образом:

```
from __future__ import with_statement # только в версии 2.5
def loop(nsec):
    myname = currentThread().name
    with lock:
        remaining.add(myname)
        print '[%s] Started %s' % (ctime(), myname)
    sleep(nsec)
    with lock:
        remaining.remove(myname)
        print '[%s] Completed %s (%d secs)' % (
            ctime(), myname, nsec)
        print '    (remaining: %s)' % (
            remaining or 'NONE',)
```

Перенос приложения в версию Python 3

3.x Теперь можно сравнительно легко перенести приведенный выше сценарий в версию Python 3.x, применив к нему инструмент `2to3` (следующий вывод приведен в сокращенном виде, поскольку полные результаты применения `diff` уже были показаны ранее):

```
$ 2to3 -w mtsleepF.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
     :
RefactoringTool: Files that were modified:
RefactoringTool: mtsleepF.py
```

После переименования `mtsleepF.py` в `mtsleepF3.py` и `mtsleep.py.bak` в `mtsleepF.py` обнаруживается, что этот сценарий относится к той замечательной категории программ, перенос которых в новую версию интерпретатора происходит идеально, без малейших проблем:

```
$ python3 mtsleepF3.py
[Sun Apr  3 23:29:39 2011] Started Thread-1
[Sun Apr  3 23:29:39 2011] Started Thread-2
[Sun Apr  3 23:29:39 2011] Started Thread-3
[Sun Apr  3 23:29:41 2011] Completed Thread-3 (2 secs)
    (remaining: Thread-2, Thread-1)
[Sun Apr  3 23:29:42 2011] Completed Thread-2 (3 secs)
    (remaining: Thread-1)
[Sun Apr  3 23:29:43 2011] Completed Thread-1 (4 secs)
    (remaining: NONE)
all DONE at: Sun Apr  3 23:29:43 2011
```

Теперь, после ознакомления с блокировками, перейдем к изучению семафоров и рассмотрим пример, в котором используется и то и другое.

4.7.4. Пример семафора

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()` (от голландского слова probeer/proberen), 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()` (от голландского слова verhogen/verhoog). 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 качестве следующего примера рассмотрим предельно упрощенную модель автомата для торговли конфетами. В данном конкретном автомате имеются в наличии только пять карманов, заполняемых запасом товара (шоколадными батончиками). Если заполнены все карманы, то в автомат больше нельзя заправить ни одной конфеты и, аналогично, при отсутствии в автомате шоколадных батончиков какого-то конкретного типа покупатели, желающие приобрести именно их, будут вынуждены возвратиться с пустыми руками. В данном случае ресурсы (карманы для конфет) являются конечными, и для отслеживания их состояния можно использовать семафор.

Исходный код сценария (`candy.py`) приведен в примере 4.11.

Пример 4.11. Автомат для торговли конфетами, управляемый с помощью семафоров (`candy.py`)

В этом сценарии блокировки и семафоры используются для моделирования автомата для торговли конфетами.

```
#!/usr/bin/env python

from atexit import register
from random import randrange
from threading import BoundedSemaphore, Lock, Thread
from time import sleep, ctime

lock = Lock()
MAX = 5
candytray = BoundedSemaphore(MAX)

def refill():
    lock.acquire()
    print 'Refilling candy...',
    try:
        candytray.release()
    except ValueError:
        print 'full, skipping'
    else:
        print 'OK'
    lock.release()

def buy():
    lock.acquire()
    print 'Buying candy...',
    if candytray.acquire(False):
        print 'OK'
    else:
        print 'empty, skipping'
```

```
30:      lock.release()
31:
32:  def producer(loops):
33:      for i in xrange(loops):
34:          refill()
35:          sleep(randrange(3))
36:
37:  def consumer(loops):
38:      for i in xrange(loops):
39:          buy()
40:          sleep(randrange(3))
41:
42:  def _main():
43:      print 'starting at:', ctime()
44:      nloops = randrange(2, 6)
45:      print 'THE CANDY MACHINE (full with %d bars)!' % MAX
46:      Thread(target=consumer, args=(randrange(
47:          nloops, nloops+MAX+2),)).start() # покупатель
48:      Thread(target=producer, args=(nloops,)).start() #продавец
49:
50:  @register
51:  def _atexit():
52:      print 'all DONE at:', ctime()
53:
54:  if __name__ == '__main__':
55:      _main()
```

Построчное объяснение

Строки 1–6

Строки запуска и импорта практически полностью совпадают с теми, которые были приведены в предыдущих примерах этой главы. Добавлено лишь объявление семафора. 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.

Строки 8–10

Глобальные переменные в этом сценарии определяют блокировку, константу, представляющую максимальное количество единиц ресурса, которые могут составить потребляемый запас, а также лоток с конфетами.

Строки 12–21

Оператор, обслуживающий эти вымышленные торговые автоматы, время от времени пополняет запасы товара. Для моделирования этого действия применяется функция `refill()`. Вся процедура представляет критический участок кода; именно поэтому захват блокировки становится обязательным условием, при котором могут быть беспрепятственно выполнены все строки от начала до конца. В коде предусмотрен вывод журнала со сведениями о выполненных действиях, с которым может ознакомиться пользователь. Кроме того, вырабатывается предупреждение при попытке превысить максимально допустимый объем запасов (строки 17-18).

Строки 23–30

Функция `buy()` противоположна по своему назначению функции `refill()`; она позволяет покупателю каждый раз получать по одной единице из хранимых запасов. Для обнаружения того, не исчерпаны ли уже конечные ресурсы, применяется условное выражение (строка 26). Значение счетчика ни в коем случае не может стать отрицательным, поэтому после достижения нулевого значения количества запасов вызов функции `buy()` блокируется до того момента, когда значение счетчика будет снова увеличено. Если этой функции передается значение неблокирующего флага, `False`, то вместо блокирования функция возвращает `False`, указывая на то, что ресурсы исчерпаны.

Строки 32–40

Функции `producer()` и `consumer()`, моделирующие пополнение и потребление запаса конфет, просто повторяются в цикле и обеспечивают выполнение соответствующих вызовов функций `refill()` и `buy()`, приостанавливаясь на короткое время между вызовами.

Строки 42–55

В последней части кода сценария содержится вызов функции `_main()`, выполняемый при запуске сценария из командной строки, предусмотрена регистрация функции выхода, а также приведено определение функции `_main()`, в которой предусмотрена инициализация вновь созданной пары потоков, которые моделируют пополнение и потребление конфет.

В коде создания потока, который представляет покупателя (потребителя), дополнительно предусмотрена возможность вводить установленное случайным образом положительное смещение, при котором покупатель фактически может попытаться получить больше шоколадных батончиков, чем заправлено в торговый автомат поставщиком/производителем (в противном случае в коде никогда не возникла бы ситуация, в которой покупателю разрешалось бы совершать попытку купить шоколадный батончик из пустого автомата).

Выполнение сценарием приводит к получению примерно такого вывода:

```
$ python candy.py
starting at: Mon Apr  4 00:56:02 2011
THE CANDY MACHINE (full with 5 bars)!
Buying candy... OK
Refilling candy... OK
Refilling candy... full, skipping
Buying candy... OK
```

```
Buying candy... OK
Refilling candy... OK
Buying candy... OK
Buying candy... OK
Buying candy... OK
all DONE at: Mon Apr  4 00:56:08 2011
```

В процессе отладки этого сценария может потребоваться вмешательство вручную

Если в какой-то момент нужно будет приступить к отладке сценария, в котором используются семафоры, прежде всего необходимо точно знать, какое значение находится в счетчике семафора в любое заданное время. В одном из упражнений в конце данной главы предлагается реализовать такое решение в сценарии `candy.py`, возможно, переименовав его в `candydebug.py`, и предусмотреть в нем возможность отображать значение счетчика. Для этого может потребоваться рассмотреть исходный код модуля `threading.py` (причем, вероятно, и в версии Python 2, и в версии Python 3).

3.x

При этом следует учитывать, что имена примитивов синхронизации модуля `threading` не являются именами классов, даже притом, что в них используется выделение прописными буквами, как в ВерблюжьемСтиле, что делает их похожими на классы. В действительности эти примитивы представляют собой всего лишь однострочные функции, применяемые для порождения необходимых объектов. При работе с модулем `threading` необходимо учитывать две сложности: во-первых, невозможно порождать подклассы создаваемых объектов (поскольку они представляют собой функции), а во-вторых, при переходе от версий 2.x к версиям 3.x изменились имена переменных.

Обе эти сложности можно было бы легко преодолеть, если бы в обеих версиях объекты предоставляли свободный и единообразный доступ к счетчику, что в действительности не обеспечивается. Непосредственный доступ к значению счетчика может быть получен, поскольку он представляет собой всего лишь атрибут класса, но, как уже было сказано, имя переменной `self.__value` изменилось. Это означает, что переменная `self._Semaphore__value` в Python 2 была переименована в `self._value` в Python 3.

Что касается разработчиков, то наиболее удобный вариант применения интерфейса прикладного программирования API (по крайней мере, по мнению автора) состоит в создании подкласса класса `threading._BoundedSemaphore` и реализации метода `__len__()`. Но если планируется поддерживать сценарий и в версии 2.x, и в версии 3.x, то следует использовать правильное значение счетчика, как было только что описано.

Перенос приложения в версию Python 3

По аналогии с `mtsleepF.py`, `candy.py` представляет собой еще один пример того, что достаточно применить инструмент `2to3`, чтобы сформировать работоспособную версию для Python 3 (для которой должно использоваться имя `candy3.py`). Оставляем проверку этого утверждения в качестве упражнения для читателя.

Резюме

В настоящей главе было продемонстрировано использование только некоторых примитивов синхронизации, которые входят в состав модуля `threading`. Поэтому при желании читатель может найти для себя широкие возможности по исследованию этой тематики. Однако следует учитывать, что объекты, представленные в

указанном модуле, полностью соответствуют своему определению, т.е. являются примитивами. Это означает, что разработчик вполне может заняться созданием на их основе собственных классов и структур данных, обеспечив, в частности, их потоковую безопасность. Для этого в стандартной библиотеке Python предусмотрен еще один объект, `Queue`.

4.8. Проблема "производитель–потребитель" и модуль `Queue/queue`

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.

3.x 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. В табл. 4.5 перечислены различные атрибуты, которые представлены в указанном модуле.

Таблица 4.5. Общие атрибуты модуля **`Queue/queue`**

Атрибут	Описание
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)

Окончание табл. 4.5

Атрибут	Описание
`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()`

Для демонстрации того, как реализуется принцип "производитель–потребитель" с помощью модуля `Queue`/`queue`, воспользуемся примером 4.12 (`prodcons.py`). Ниже приведен вывод, полученный при одном запуске на выполнение этого сценария.

```
$ prodcons.py
starting writer at: Sun Jun 18 20:27:07 2006
producing object for Q... size now 1
starting reader at: Sun Jun 18 20:27:07 2006
consumed object from Q... size now 0
producing object for Q... size now 1
consumed object from Q... size now 0
producing object for Q... size now 1
producing object for Q... size now 2
producing object for Q... size now 3
consumed object from Q... size now 2
consumed object from Q... size now 1
writer finished at: Sun Jun 18 20:27:17 2006
consumed object from Q... size now 0
reader finished at: Sun Jun 18 20:27:25 2006
all DONE
```

Пример 4.12. Пример реализации принципа "производитель–потребитель" (`prodcons.py`)

В этой реализации принципа "производитель–потребитель" используются объекты `Queue` и вырабатывается случайным образом количество произведенных (и потребленных) товаров. Производитель и потребитель моделируются с помощью потоков, действующих отдельно и одновременно.

```
#!/usr/bin/env python

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'

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

Вполне очевидно, что операции, выполняемые производителем и потребителем, не всегда чередуются, поскольку образуют две независимые последовательности. (Весьма удачно то, что в нашем распоряжении имеется готовый механизм выработки случайных чисел!) Если же говорить серьезно, то события, происходящие в действительности, как правило, подчиняются законам случайности и недетерминированы.

Построчное объяснение

Строки 1–6

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.)

Строки 8–16

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.

Строки 18–26

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.

Строки 28-29

Это всего лишь подготовительные строки, с помощью которых задается общее количество потоков, подлежащих порождению и запуску.

Строки 31–47

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.

В настоящей главе было показано, 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.

4.9. Дополнительные сведения об использовании потоков

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 (поскольку некоторые из соответствующих тем уже рассматривались в главе "Среда выполнения" книги "Core Python Programming" или "Core Python Language Fundamentals"), перечислим основные альтернативы модулю `threading`, касающиеся поддержки нескольких потоков или процессов.

4.9.1. Модуль `subprocess`

2.4

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.

4.9.2. Модуль `multiprocessing`

2.6

Этот модуль, 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.

4.9.3. Модуль concurrent.futures

3.2 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. Модуль можно получить по адресу `http://code.google.com/p/pythonfutures`.

Рассмотрим вариант сценария `bookrank3.py` с указанными изменениями. При условии, что все прочее остается таким, как прежде, рассмотрим новые операторы импорта и изменившуюся часть сценария `_main()`:

```
from concurrent.futures import ThreadPoolExecutor
        . . .
def _main():
    print('At', ctime(), 'on Amazon...')
    with ThreadPoolExecutor(3) as executor:
        for isbn in ISBNs:
            executor.submit(_showRanking, isbn)
    print('all DONE at:', ctime())
```

Методу `concurrent.futures.ThreadPoolExecutor` передается параметр, представляющий собой размер пула потоков, а приложение применяется для определения рангов трех книг. Безусловно, это — приложение, ограничиваемое пропускной способностью ввода-вывода, для которого применение потоков оказывает наибольшую пользу. Что касается приложений, ограничиваемых пропускной способностью процессора, то вместо указанного метода целесообразно было бы использовать `concurrent.futures.ProcessPoolExecutor`.

После создания управляющего объекта (действие которого распространяется на потоки или процессы), отвечающего за планирование заданий и сбор результатов, можно вызвать его метод `submit()` для выполнения намеченной ранее задачи порождения потока.

После полного переноса в версию Python 3 путем замены оператора форматирования строки методом `str.format()`, повсеместного введения инструкции **`with`** и использования метода `map()` управляющего объекта появляется возможность полностью удалить метод `_showRanking()` и передать его функции в программу `_main()`. Заключительная версия сценария `bookrank3CF.py` приведена в примере 4.13.

Пример 4.13. Применение средств управления заданиями высокого уровня (`bookrank3CF.py`)

В этом участке кода, как и в предыдущих примерах, осуществляется сбор с экрана данных о рангах книг, но на этот раз с помощью модуля `concurrent.futures`.

```
1   #!/usr/bin/env python
2
3   from concurrent.futures import ThreadPoolExecutor
4   from re import compile
```

```
5    from time import ctime
6    from urllib.request import urlopen as uopen
7
8    REGEX = compile(b'#([\d,]+) in Books ')
9    AMZN = 'http://amazon.com/dp/'
10   ISBNs = {
11        '0132269937': 'Core Python Programming',
12        '0132356139': 'Python Web Development with Django',
13        '0137143419': 'Python Fundamentals',
14   }
15
16   def getRanking(isbn):
17       with uopen('{0}{1}'.format(AMZN, isbn)) as page:
18           return str(REGEX.findall(page.read())[0], 'utf-8')
19:
20:  def _main():
21:       print('At', ctime(), 'on Amazon...')
22:       with ThreadPoolExecutor(3) as executor:
23:           for isbn, ranking in zip(
24:                   ISBNs, executor.map(getRanking, ISBNs)):
25:               print('- %r ranked %s' % (ISBNs[isbn], ranking)
26:        print('all DONE at:', ctime())
27:
28:   if __name__ == '__main__':
29:       main()
```

Построчное объяснение

Строки 1–14

Если не считать новой инструкции **`import`**, то вся первая половина этого сценария полностью идентична той, что приведена в файле `bookrank3.py`, который рассматривался выше в главе.

Строки 16–18

В новой функции `getRanking()` используются инструкция **`with`** и функция `str.format()`. Аналогичные изменения можно внести в сценарий `bookrank.py`, поскольку оба указанных средства доступны также в версии 2.6 и последующих (а не предусмотрены исключительно в версиях 3.x).

Строки 20–26

В предыдущем примере кода использовался метод `executor.submit()` для формирования заданий. В данном примере предусмотрены некоторые изменения в связи с использованием метода `executor.map()`, поскольку он позволяет реализовать функции из `_showRanking()` и полностью исключать их поддержку из нашего кода.

Полученный вывод почти аналогичен тому, который рассматривался ранее:

```
$ python3 bookrank3CF.py
At Wed Apr  6 00:21:50 2011 on Amazon...
- 'Core Python Programming' ranked 43,992
- 'Python Fundamentals' ranked 1,018,454
- 'Python Web Development with Django' ranked 502,566
all DONE at: Wed Apr  6 00:21:55 2011
```

Дополнительные сведения об истории создания модуля `concurrent.futures` можно найти с помощью приведенных ниже ссылок.

```
http://docs.python.org/dev/py3k/library/concurrent.futures.html
http://code.google.com/p/pythonfutures/
http://www.python.org/dev/peps/pep-3148/
```

Краткое описание этих параметров, а также другая информация, касающаяся модулей и пакетов для многопоточной организации программы, приведена в следующем разделе.

4.10. Связанные модули

В табл. 4.6 перечислены некоторые модули, которые можно использовать при программировании многопоточных приложений.

Таблица 4.6. Стандартные библиотечные модули, связанные с многопоточной поддержкой

Модуль	Описание
`thread`[a]	Основной, находящийся на низком уровне модуль поддержки потоков
`threading`	Объекты для многопоточной организации работы и синхронизации высокого уровня
`multiprocessing`[b]	Запуск/использование подпроцессов с помощью интерфейса threading
`subprocess`[c]	Полный отказ от потоков и выполнение вместо них процессов
`Queue`	Синхронизированная очередь с последовательной организацией для нескольких потоков
`mutex`[d]	Объекты мьютексов
`concurrent.futures`[e]	Библиотека высокого уровня для асинхронного выполнения
`SocketServer`	Создание/управление многопоточными серверами TCP или UDP

[a] Переименован в `_thread` в Python 3.0.
[b] Новое в версии Python 2,6.
[c] Новое в версии Python 2.4.
[d] Обозначен как устаревший в Python 2.6 и удален в версии 3.0.
[e] Впервые введенный в Python 3.2, но предоставляемый вне стандартной библиотеки для версии 2.6 и последующих версий.

4.11. Упражнения

4.1. *Сопоставление процессов с потоками.* В чем состоят различия между процессами и потоками?

4.2. *Потоки Python.* Какие типы многопоточных приложений обеспечивают наибольшую производительность в Python, ограничиваемые пропускной способностью ввода-вывода или пропускной способностью процессора?

4.3. *Потоки.* Какие наиболее заметные отличия будут обнаружены после запуска многопоточного приложения в системе не с одним, а с несколькими процессорами? Как, по вашему мнению, будут выполняться несколько потоков в этих системах?

4.4. *Потоки и файлы.*

а) Создайте функцию, которая получает байтовое значение и имя файла (в виде параметров или данных, введенных пользователем) и определяет, сколько раз байтовое значение появляется в файле.

б) Теперь предположим, что входной файл является чрезвычайно большим. Допускается применение для чтения файла одновременно нескольких функций, поэтому измените полученное ранее решение в целях создания многочисленных потоков, обрабатывающих разные части файла, чтобы каждый поток отвечал лишь за определенную часть файла. Соберите данные, полученные каждым потоком, и рассчитайте суммарное значение. Воспользуйтесь модулем `timeit` для измерения продолжительности работы обоих решений, однопоточного и многопоточного, и прокомментируйте разницу в производительности, если таковая будет обнаружена.

4.5. *Потоки, файлы и регулярные выражения*. Для выполнения этого задания требуется очень большой файл почтового ящика. Если таковой отсутствует, соедините все свои сообщения электронной почты в общий текстовый файл. Задание заключается в следующем. Воспользуйтесь регулярными выражениями, предназначенными для распознавания адресов электронной почты и URL веб-сайтов, которые рассматривались ранее в этой книге, для преобразования всех адресов электронной почты и URL из указанного большого файла в актуальные ссылки. Эти ссылки необходимо сохранить в отдельном файле с расширением `.html` (или `.htm`), который можно открыть в веб-браузере для получения возможности переходить по ним с помощью щелчка мышью. Используйте потоки для проведения процесса преобразования одновременно по всему большому текстовому файлу, после чего соберите полученные результаты в отдельный новый файл `.html`. Откройте этот файл в веб-браузере, чтобы проверить, действительно ли работают ссылки.

4.6. *Потоки и сети.* В приложении службы интерактивной переписки, разработанном в качестве упражнения в предыдущей главе, требовалось использование в составе решения так называемых тяжеловесных потоков, или процессов. Преобразуйте это решение в многопоточное.

4.7. **Потоки и программирование для веб.* Приложение `Crawler`, которое представлено в главе 10 "Программирование для веб. CGI и WSGI", является однопоточным приложением, предназначенным для загрузки веб-страниц. Его усовершенствованию способствовало бы применение многопоточного программирования. Обновите сценарий `crawl.py` (переименовав его в `mtcrawl.py`), чтобы для загрузки страниц использовались независимые потоки. Обязательно воспользуйтесь механизмом блокировки того или иного типа для предотвращения конфликтов доступа к очереди ссылок.

4.8. *Пулы потоков.* Внесите изменения в код сценария `prodcons.py`, который рассматривается в примере 4.12, чтобы в нем вместо потока производителя и одного потока потребителя могло применяться любое количество потоков потребителя (пул потоков) для обработки или выборки в любой момент времени нескольких элементов из очереди `Queue`.

4.9. *Файлы*. Создайте ряд потоков для подсчета количества строк в наборе текстовых файлов (который может иметь большой суммарный объем). Может

быть предусмотрена возможность выбирать количество используемых потоков. Сравните производительность многопоточной и однопоточной версий этого кода. Совет. Ознакомьтесь с упражнениями в конце главы 9 в книге *Core Python Programming* или *Core Python Language Fundamentals*.

4.10. *Параллельная обработка.* Возьмите за основу свое решение упражнения 4.9 и примите к выбранной вами произвольной задаче, такой как обработка набора сообщений электронной почты, загрузка веб-страниц, обработка веб-каналов RSS или Atom, усовершенствование обработки сообщений сервером интерактивной переписки, поиск решения головоломки и т.д.

4.11. *Примитивы синхронизации.* Изучите каждый из примитивов синхронизации в модуле threading. Опишите их назначение, укажите возможную область их применения и подготовьте примеры рабочего кода для каждого из них.

Следующий ряд упражнений касается сценария `candy.py`, который рассматривался в примере 4.11.

4.12. *Перенос в версию Python 3.* Возьмите за основу сценарий `candy.py` и примените к нему инструмент `2to3` для создания версии Python 3 с именем `candy3.py`.

4.13. *Модуль `threading`.* Добавьте к сценарию средства отладки. В частности, для приложений, в которых используются семафоры (с начальным значением, как правило, больше 1), можно предусмотреть точное определение значения счетчика в любое конкретное время. Подготовьте вариант сценария `candy.py` (который можно назвать `candydebug.py`) и реализуйте в нем возможность отображать значение счетчика. Вам может потребоваться изучить исходный код сценария `threading.py`, как было указано выше в одном из советов. После внесения этих изменений можно откорректировать вывод сценария, чтобы он выглядел примерно так:

```
$ python candydebug.py
starting at: Mon Apr  4 00:24:28 2011
THE CANDY MACHINE (full with 5 bars)!
Buying candy... inventory: 4
Refilling candy... inventory: 5
Refilling candy... full, skipping
Buying candy... inventory: 4
Buying candy... inventory: 3
Refilling candy... inventory: 4
Buying candy... inventory: 3
Buying candy... inventory: 2
Buying candy... inventory: 1
Buying candy... inventory: 0
Buying candy... empty, skipping
all DONE at: Mon Apr  4 00:24:36 2011
```