

Multithreading, Networks, and Client/ Server Programming

After completing this chapter, you will be able to

- ⦿ Describe what threads do and how they are manipulated in an application
- ⦿ Code an algorithm to run as a thread
- ⦿ Use conditions to solve a simple synchronization problem with threads
- ⦿ Use IP addresses, ports, and sockets to create a simple client/server application on a network
- ⦿ Decompose a server application with threads to handle client requests efficiently
- ⦿ Restructure existing applications for deployment as client/server applications on a network

Thus far in this book we have explored ways of solving problems by using multiple cooperating algorithms and data structures. Another commonly used strategy for problem solving involves the use of multiple threads. Threads describe processes that can run concurrently to solve a problem. They can also be organized in a system of **clients** and **servers**. For example, a Web browser runs in a client thread and allows a user to view Web pages that are sent by a Web server, which runs in a server thread. Client and server threads can run concurrently on a single computer or can be distributed across several computers that are linked in a **network**. The technique of using multiple threads in a program is known as multithreading. This chapter offers an introduction to multithreading, networks, and client/server programming. We provide just enough material to get you started with these topics; more complete surveys are available in advanced computer science courses.

Threads and Processes

You are well aware that an algorithm describes a computational process that runs to completion. You are also aware that a process consumes resources, such as CPU (central processing unit) cycles and memory. Until now, we have associated an algorithm or a program with a single process, and we have assumed that this process runs on a single computer. However, your program's process is not the only one that runs on your computer, and a single program could describe several processes that could run concurrently on your computer or on several networked computers. The following historical summary shows how this is the case.

Time-sharing operating systems: In the late 1950s and early 1960s, computer scientists developed the first time-sharing operating systems. These systems allow several programs to run concurrently on a single computer. Instead of giving their programs to a human scheduler to run one after the other on a single machine, users log in to the computer via remote terminals. They then run their programs and have the illusion, if the system performs well, of having sole possession of the machine's resources (CPU, disk drives, printer, etc.). Behind the scenes, the operating system creates separate processes for these programs. The system gives each process a turn at the CPU and other resources, and it performs all the work of scheduling. When a process is about to be swapped out of the CPU, the system saves its state (the values of variables currently in play and the call stack for any active subroutines) and then restores the state of the process about to execute. If this procedure, called a **context switch**, happens very rapidly, the illusion of concurrency is maintained. Time-sharing systems are still in widespread use in the form of Web servers, e-mail servers, print servers, and other kinds of servers on networked systems.

Multiprocessing systems: Most time-sharing systems allow a single user to run one program and then return to the operating system to run another program before the first program is finished. The concept of a single user running several programs at once was extended to desktop microcomputers in the late 1980s, when these machines became more powerful. For example, the Macintosh MultiFinder allowed a user to run a word processor, a spreadsheet, and the Finder (the file browser) concurrently and to switch from one application to another by selecting an application's window. Users of stand-alone PCs now

take this capability for granted. A related development was the ability of a program to start another program by “forking,” or creating a new process. For example, a word processor might create another process to print a document in the background, while the user is staring at the window thinking about the next words to type.

Networked or distributed systems: The late 1980s and early 1990s saw the rise of networked systems. At that time, the processes associated with a single program or with several programs began to be distributed across several CPUs linked by high-speed communication lines. Thus, for example, the Web browser that appears to be running on my machine is actually making requests as a client to a Web server application that runs on a multiuser machine at a remote location on the Internet. The problems of scheduling and running processes are more complex on a networked system, but the basic ideas are the same.

Parallel systems: As CPUs became less expensive and smaller, it became feasible to run a single program on several CPUs at once. **Parallel computing** is the discipline of building the hardware architectures, operating systems, and specialized algorithms for running a program on a cluster of processors. The multi-core technology now found in all new PCs can be used to run a single program or multiple programs on several processors simultaneously.

Threads

Most modern computers, whether they are networked or stand-alone machines, represent some processes as **threads**. For example, a Web browser uses one thread to load an image from the Internet while using another thread to format and display text. The Python virtual machine runs several threads that you have already used without realizing it. For example, the IDLE editor runs as a separate thread, as does your main Python application program. The garbage collector that recycles objects in your Python programs runs as a separate thread in the Python virtual machine.

In Python, a thread is an object like any other in that it can hold data, be stored in data structures, and be passed as parameters to methods. However, some code defined in a thread can also be executed as a process. To execute this code, a thread’s class must implement a **run** method.

During its lifetime, a thread can be in various states. Figure 10-1 shows some of the states in the lifetime of a Python thread. In this diagram, the box labeled “The ready queue” is a data structure, whereas the box labeled “The CPU” is a hardware resource. The thread states are the labeled ovals.

After it is created, a thread remains newborn and inactive until someone runs its **start** method. Running this method also makes the thread “ready” and places a reference to it in the **ready queue**. A queue is a data structure that enforces first-come, first-served access to a single resource. The resource in this case is the CPU, which can execute the instructions of just one thread at a time. A newly started thread’s **run** method is also activated. However, before its first instruction can be executed, the thread must wait its turn in the ready queue

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

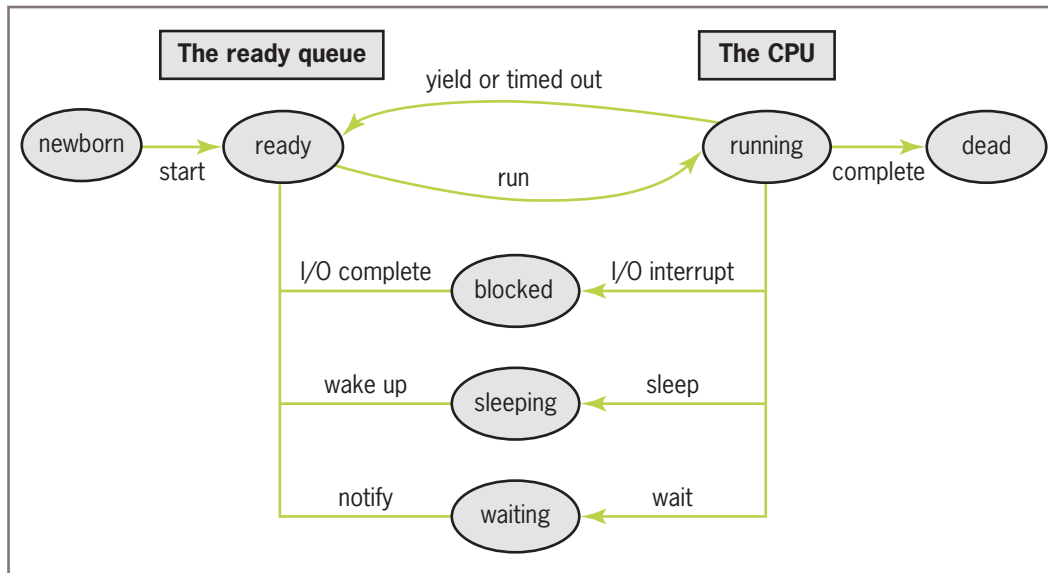


Figure 10-1 States in the life of a thread

for access to the CPU. After the thread gets access to the CPU and executes some instructions in its **run** method, the thread can lose access to the CPU in several ways:

- **Time-out**—Most computers running Python programs automatically time-out a running thread every few milliseconds. The process of automatically timing-out, also known as **time slicing**, has the effect of pausing the running thread's execution and sending it to the rear of the ready queue. The thread at the front of the ready queue is then given access to the CPU.
- **Sleep**—A thread can be put to sleep for a given number of milliseconds. When the thread wakes up, it goes to the rear of the ready queue.
- **Block**—A thread can wait for some event, such as user input, to occur. When a blocked thread is notified that an event has occurred, it goes to the rear of the ready queue.
- **Wait**—A thread can voluntarily relinquish the CPU to wait for some condition to become true. A waiting thread can be notified when the condition becomes true and move again to the rear of the ready queue.

When a thread gives up the CPU, the computer saves its state (the values of its instance variables and data on its call stack), so that when the thread returns to the CPU, its **run** method can pick up where it left off. As mentioned earlier, the process of saving or restoring a thread's state is called a context switch.

When a thread's **run** method has executed its last instruction, the thread dies as a process but continues to exist as an object. A thread object can also die if it raises an exception that is not handled.

Python's **threading** module includes resources for creating threads and managing multi-threaded applications. The most common way to create a thread is to define a class that

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

extends the class `threading.Thread`. The new class should include a `run` method that executes the algorithm in the new thread. The `start` method places a thread at the rear of the ready queue. The next code segment defines a simple thread class that prints its name.

```
from threading import Thread
```

356

```
class MyThread (Thread):
    """A thread that prints its name."""

    def __init__(self, name):
        Thread.__init__(self, name = name)

    def run(self):
        print("Hello, my name is %s" % self.getName())
```

The session that follows instantiates this class and starts up the thread.

```
>>> process = MyThread("Ken")
>>> process.start()
Hello, my name is Ken
```

The thread's `start` method automatically invokes its `run` method. When you run this code in the IDLE shell, your new thread runs to completion but does not appear to quit and return you to another shell prompt. To do so, you must press Control+C to interrupt the process. Because IDLE itself runs in a thread, it is not generally a good idea to test a multithreaded application in that environment. From now on, we will launch Python programs containing threads from a terminal prompt rather than from an IDLE window. Here is the code for a `main` function that starts up a thread and runs to a normal termination at the terminal:

```
def main():
    MyThread("Ken").start()
if __name__ == "__main__":
    main()
```

The `Thread` class maintains an instance variable for the thread's name and includes the associated methods `getName` and `setName`. Table 10-1 lists some important `Thread` methods.

Thread Method	What It Does
<code>__init__(name = None)</code>	Initializes the thread's name.
<code>getName()</code>	Returns the thread's name.
<code>setName(newName)</code>	Sets the thread's name to <code>newName</code> .
<code>run()</code>	Executes when the thread acquires the CPU.
<code>start()</code>	Makes the new thread ready. Raises an exception if run more than once.
<code>isAlive()</code>	Returns <code>True</code> if the thread is alive or <code>False</code> otherwise.

Table 10-1 Some `Thread` Methods

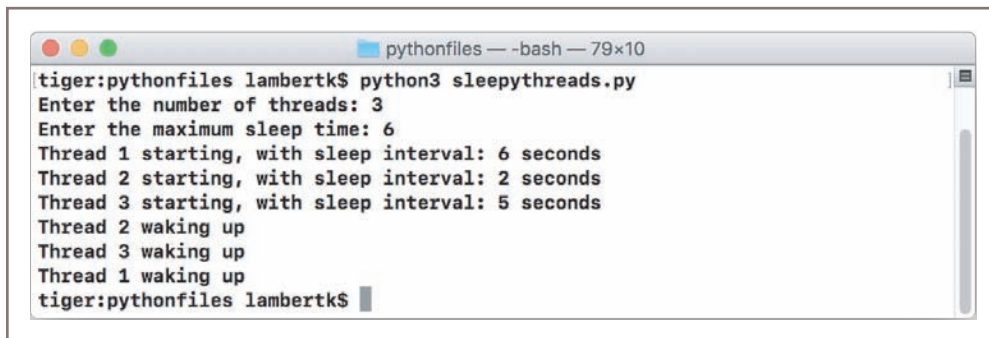
Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Other important resources used with threads include the function `time.sleep` and the class `threading.Condition`. We now consider some example programs that illustrate the use of these resources.

Sleeping Threads

In our first example, we develop a program that allows the user to start several threads. Each thread does not do much when started; it simply prints a message, goes to sleep for a random number of seconds, and then prints a message and terminates on waking up. The program allows the user to specify the number of threads to run and the maximum sleep time. When a thread is started, it prints a message identifying itself and its sleep time and then goes to sleep. When a thread wakes up, it prints another message identifying itself. A session with this program is shown in Figure 10-2.



```
tiger:pythonfiles lambertk$ python3 sleepythreads.py
Enter the number of threads: 3
Enter the maximum sleep time: 6
Thread 1 starting, with sleep interval: 6 seconds
Thread 2 starting, with sleep interval: 2 seconds
Thread 3 starting, with sleep interval: 5 seconds
Thread 2 waking up
Thread 3 waking up
Thread 1 waking up
tiger:pythonfiles lambertk$
```

Figure 10-2 A run of the sleeping threads program

Note the following points about the example in Figure 10-2:

- When a thread goes to sleep, the next thread has an opportunity to acquire the CPU and display its information in the view.
- Threads with random sleep times do not necessarily wake up in the order in which they were started. The size of the sleep interval determines this order. In Figure 10-2, thread 2 has the shortest sleep time, so it wakes up first. Thread 3 wakes up before thread 1, because thread 1 has the longest sleep time.

The program consists of the class `SleepyThread`, a subclass of `Thread`, and a `main` function. When called within a thread's `run` method, the function `time.sleep` puts that thread to sleep for the specified number of seconds. Here is the code:

```
"""
File: sleepythreads.py
Illustrates concurrency with multiple threads.
"""
```

```
import random, time
from threading import Thread
```

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```

class SleepyThread(Thread):
    """Represents a sleepy thread."""

    def __init__(self, number, sleepMax):
        """Create a thread with the given name
        and a random sleep interval less than the
        maximum."""
        Thread.__init__(self, name = "Thread " + str(number))
        self.sleepInterval = random.randint(1, sleepMax)

    def run(self):
        """Print the thread's name and sleep interval
        and sleep for that interval. Print the name
        again at wake-up."""
        print("%s starting, with sleep interval: %d seconds" % \
              (self.getName(), self.sleepInterval))
        time.sleep(self.sleepInterval)
        print("%s waking up" % self.getName())

    def main():
        """Create the user's number of threads with sleep
        intervals less than the user's maximum. Then start
        the threads."""
        numThreads = int(input("Enter the number of threads: "))
        sleepMax = int(input("Enter the maximum sleep time: "))
        threadList = []
        for count in range(numThreads):
            threadList.append(SleepyThread(count + 1, sleepMax))
        for thread in threadList: thread.start()

if __name__ == "__main__":
    main()

```

Producer, Consumer, and Synchronization

In the previous example, the threads ran independently and did not interact. However, in many applications, threads interact by sharing data. One such interaction is the **producer/consumer relationship**. Think of an assembly line in a factory. Worker A, at the beginning of the line, produces an item that is then ready for access by the next person on the line, Worker B. In this case, Worker A is the producer, and Worker B is the consumer. Worker B then becomes the producer, processing the item in some way until it is ready for Worker C, and so on.

Three requirements must be met for the assembly line to function properly:

1. A producer must produce each item before a consumer consumes it.
2. Each item must be consumed before the producer produces the next item.
3. A consumer must consume each item just once.

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Let us now consider a computer simulation of the producer/consumer relationship. In its simplest form, the relationship has only two threads: a producer and a consumer. They share a single data cell that contains an integer. The producer sleeps for a random interval, writes an integer to the shared cell, and generates the next integer to be written, until the integer reaches an upper bound. The consumer sleeps for a random interval and reads the integer from the shared cell, until the integer reaches the upper bound. Figure 10-3 shows two runs of this program. The user enters the number of accesses (data items produced and consumed). The output announces that the producer and consumer threads have started up and shows when each thread accesses the shared data.

```

pythonfiles --bash-- 56x19
tiger:pythonfiles lambertk$ python3 producerconsumer1.py
Enter the number of accesses: 4
Starting the threads
Producer starting up
Consumer starting up

Producer setting data to 1
Consumer accessing data 1
Producer setting data to 2
Consumer accessing data 2
Producer setting data to 3
Consumer accessing data 3
Producer setting data to 4
Producer is done producing

Consumer accessing data 4
Consumer is done consuming

tiger:pythonfiles lambertk$

pythonfiles --bash-- 56x19
tiger:pythonfiles lambertk$ python3 producerconsumer1.py
Enter the number of accesses: 4
Starting the threads
Producer starting up
Consumer starting up

Consumer accessing data -1
Producer setting data to 1
Producer setting data to 2
Consumer accessing data 2
Producer setting data to 3
Consumer accessing data 3
Producer setting data to 4
Producer is done producing

Consumer accessing data 4
Consumer is done consuming

tiger:pythonfiles lambertk$

```

Figure 10-3 Two runs of the producer/consumer program

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

In the first run of the program, the producer happens to update the shared data each time before the consumer accesses it. However, some bad things happen in the second run of the program:

1. The consumer accesses the shared cell before the producer has written its first datum.
2. The producer then writes two consecutive data (1 and 2) before the consumer has accessed the cell again.
3. The consumer accesses data 2 but misses data 1.

Although the producer always produces all of its data, the consumer can access data that are not there, can miss data, and can access the same data more than once. These are known as **synchronization problems**. Before we explain why they occur, we present the essential parts of the program itself (**producerconsumer1.py**), which consists of the four resources in Table 10-2.

Class or Function	Role and Responsibility
main	Manages the user interface. Creates the shared cell and producer and consumer threads and starts the threads.
SharedCell	Represents the shared data, which is an integer (initially -1).
Producer	Represents the producer process. Repeatedly writes an integer to the cell and increments the integer, until it reaches an upper bound.
Consumer	Represents the consumer process. Repeatedly reads an integer from the cell, until it reaches an upper bound.

Table 10-2 The classes and **main** function in the producer/consumer program

The code for the **main** function is similar to the one in the previous example:

```
def main():
    """Get the number of accesses from the user, create a
    shared cell, and create and start up a producer and a
    consumer."""
    accessCount = int(input("Enter the number of accesses: "))
    sleepMax = 4
    cell = SharedCell()
    producer = Producer(cell, accessCount, sleepMax)
    consumer = Consumer(cell, accessCount, sleepMax)
    print("Starting the threads")
    producer.start()
    consumer.start()
```

Here is the code for the classes **SharedCell**, **Producer**, and **Consumer**:

```
import time, random
from threading import Thread, currentThread
```

```

class SharedCell(object):
    """Shared data for the producer/consumer problem."""

    def __init__(self):
        """Data undefined at startup."""
        self.data = -1

    def setData(self, data):
        """Producer's method to write to shared data."""
        print("%s setting data to %d" % \
              (currentThread().getName(), data))
        self.data = data

    def getData(self):
        """Consumer's method to read from shared data."""
        print("%s accessing data %d" % \
              (currentThread().getName(), self.data))
        return self.data

class Producer(Thread):
    """A producer of data in a shared cell."""

    def __init__(self, cell, accessCount, sleepMax):
        """Create a producer with the given shared cell,
        number of accesses, and maximum sleep interval."""
        Thread.__init__(self, name = "Producer")
        self.accessCount = accessCount
        self.cell = cell
        self.sleepMax = sleepMax

    def run(self):
        """Announce start-up, sleep and write to shared
        cell the given number of times, and announce
        completion."""
        print("%s starting up" % self.getName())
        for count in range(self.accessCount):
            time.sleep(random.randint(1, self.sleepMax))
            self.cell.setData(count + 1)
        print("%s is done producing\n" % self.getName())

class Consumer(Thread):
    """A consumer of data in a shared cell."""

    def __init__(self, cell, accessCount, sleepMax):
        """Create a consumer with the given shared cell,
        number of accesses, and maximum sleep interval."""
        Thread.__init__(self, name = "Consumer")
        self.accessCount = accessCount
        self.cell = cell
        self.sleepMax = sleepMax

```

```
def run(self):
    """Announce start-up, sleep, and read from shared
    cell the given number of times, and announce completion."""
    print("%s starting up" % self.getName())
    for count in range(self.accessCount):
        time.sleep(random.randint(1, self.sleepMax))
        value = self.cell.getData()
        print("%s is done consuming\n" % self.getName())
```

The cause of the synchronization problems is not hard to spot in this code. On each pass through their main loops, the threads sleep for a random interval of time. Thus, if the consumer thread has a shorter interval than the producer thread on a given cycle, the consumer wakes up sooner and accesses the shared cell before the producer has a chance to write the next datum. Conversely, if the producer thread wakes up sooner, it accesses the shared data and writes the next datum before the consumer has a chance to read the previous datum.

To solve this problem, we need to synchronize the actions of the producer and consumer threads. In addition to holding data, the shared cell must be in one of two states: writeable or not writeable. The cell is writeable if it has not yet been written to (at start-up) or if it has just been read from. The cell is not writeable if it has just been written to. These two conditions can now control the callers of the **setData** and **getData** methods in the **SharedCell** class as follows:

1. While the cell is writeable, the caller of **getData** (the consumer) must wait or suspend activity, until the producer writes a datum. When this happens, the cell becomes not writeable, the other thread (the producer) is notified to resume activity, and the data are returned (to the consumer).
2. While the cell is not writeable, the caller of **setData** (the producer) must wait or suspend activity, until the consumer reads a datum. When this happens, the cell becomes writeable, the other thread (the consumer) is notified to resume activity, and the data are modified (by the producer).

To implement these restrictions, the **SharedCell** class now includes two additional instance variables:

1. A Boolean flag named **writeable**. If this flag is **True**, only writing to the cell is allowed; if it is **False**, only reading from the cell is allowed.
2. An instance of the **threading.Condition** class. This object allows each thread to block until the Boolean flag is in the appropriate state to write to or read from the cell.

A **Condition** object is like a **lock** on a resource. When a thread acquires this lock, no other thread can access the resource, even if the acquiring thread is timed-out. After a thread successfully acquires the lock, it can do its work or relinquish the lock in one of two ways:

1. By calling the condition's **wait** method. This method causes the thread to block until it is notified that it can continue its work.
2. By calling the condition's **release** method. This method unlocks the resource and allows it to be acquired by other threads.

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

When other threads attempt to acquire a locked resource, they block until the thread is released or a thread holding the lock calls the condition's **notify** method. To summarize, the pattern for a thread accessing a resource with a lock is the following:

```
Run acquire on the condition
While it's not OK to do the work
    Run wait on the condition
Do the work with the resource
Run notify on the condition
Run release on the condition
```

Computer scientists call the step labeled **Do the work with the resource** a **critical section**. The code in a critical section must be run in a **thread-safe** manner, meaning that the thread executing this code must be able to finish it before another thread accesses the same resource. Table 10-3 lists the methods of the **Condition** class.

Condition Method	What It Does
acquire()	Attempts to acquire the lock. Blocks if the lock is already taken.
release()	Relinquishes the lock, leaving it to be acquired by others.
wait()	Releases the lock, blocks the current thread until another thread calls notify or notifyAll on the same condition, and then reacquires the lock. If multiple threads are waiting, the notify method wakes up only one of the threads, while notifyAll always wakes up all of the threads.
notify()	Lets the next thread waiting on the lock know that it's available.
notifyAll()	Lets all threads waiting on the lock know that it's available.

Table 10-3 The methods of the **Condition** class

Here is the code that shows the addition of synchronization to the **SharedCell** class (**producerconsumer2.py**):

```
import time, random
from threading import Thread, currentThread, Condition

class SharedCell(object):
    """Shared data that sequences writing before reading."""

    def __init__(self):
        """Can produce but not consume at startup."""
        self.data = -1
        self.writeable = True
        self.condition = Condition()

    def setData(self, data):
        """Second caller must wait until someone has
        consumed the data before resetting it."""
```

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```

self.condition.acquire()
while not self.writeable:
    self.condition.wait()
print("%s setting data to %d" % \
      (currentThread().getName(), data))
self.data = data
self.writeable = False
self.condition.notify()
self.condition.release()

def getData(self):
    """Caller must wait until someone has produced the
    data before accessing it."""
    self.condition.acquire()
    while self.writeable:
        self.condition.wait()
    print("%s accessing data %d" % \
          (currentThread().getName(), self.data))
    self.writeable = True
    self.condition.notify()
    self.condition.release()
    return self.data

```

Exercises

1. What does a thread's **run** method do?
2. What is time slicing?
3. What is a synchronization problem?
4. What is the difference between a sleeping thread and a waiting thread?
5. Give two real-world examples of the producer-consumer problem.

The Readers and Writers Problem

In many applications, threads may share data as readers and writers in a looser manner than producers and consumers. Unlike producers and consumers, readers and writers may access the shared data in any order, and there may be multiple readers and writers. For example different threads may access a database, either in primary memory or secondary file storage, to access or modify the state of the data. In this situation, also known as the **readers and writers problem**,

- readers access the data to observe it
- writers access the data to modify it
- only one writer can be writing at a given time, and that writer must be able to finish before other writers or readers can begin writing or reading

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

- multiple readers can read the shared data concurrently without waiting for each other to finish, but all active readers must finish before a writer starts writing

Obviously, reader and writer threads must be synchronized around the shared data, to avoid having a reader or writer access the data at an inappropriate moment. For example, although it's okay for two readers to access the shared data at the same time, we would not want two writers to do so. Moreover, we would not want a writer and a reader to access the shared data at the same time.

Some Python data structures, such as lists and dictionaries, are already thread-safe, because they provide automatic support for synchronizing multiple readers and writers. Thus, in the case of a dictionary, Python guarantees that multiple threads may access the data to read from it (using any of the operations such as **get**, the subscript, or **len**). But if a thread is writing to a dictionary (using the subscript or **pop**), no other thread may read or write until the current writer completes its operation.

By contrast, many other objects, including those that might be contained in a list or a dictionary, are not themselves thread-safe. Examples include many of the new types of objects you defined in Chapter 9, such as **SavingsAccount** objects. In these cases, you would need to include extra machinery to ensure thread-safety, when using such objects in a multithreaded program.

A solution to the readers and writers problem is to encase the shared data in a shared cell object, with a locking mechanism to synchronize access for multiple readers and writers. We next develop an abstraction of a shared cell (**sharedcell.py**) that can be used in any application to synchronize readers and writers. The interface for this resource is listed in Table 10-4.

SharedCell Method	What It Does
SharedCell(data)	Constructor, creates a shared cell containing data .
read(readerFunction)	Applies readerFunction to the cell's shared data in a critical section. readerFunction must be a function of one argument, which is of the same type as the shared data. The function's code should only observe, not modify, the data. Returns the result of this function.
write(writerFunction)	Applies writerFunction to the cell's shared data in a critical section. writerFunction must be a function of one argument, which is of the same type as the shared data. The function's code can observe or modify the data. Returns the result of this function.

Table 10-4 SharedCell methods

Using the SharedCell Class

To see how a shared cell is used, suppose that readers and writers must access a common **SavingsAccount** object, of the type discussed in Chapter 9. Readers can use the **getBalance** method to observe the account's balance, while writers can use the **deposit**, **withdraw**, or **computeInterest** methods to make changes to the account's balance. But they must use

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

these methods in a thread-safe manner, and that's where our **SharedCell** resource comes into play. Let's assume that we create a shared cell containing a **SavingsAccount** object for multiple readers and writers, as follows:

```
account = SavingsAccount(name = "Ken", balance = 100.00)
cell = SharedCell(account)
```

Then at some point, a reader could run the code

```
print("The account balance is ",
      cell.read(lambda account: account.getBalance()))
```

to display the account's balance. A writer could run the code

```
amount = 200.00
cell.write(lambda account: account.deposit(amount))
```

to deposit \$200.00 into the account.

Note the use of Python's **lambda** expression, introduced in Chapter 6. The syntax of the **lambda** expressions used here is

lambda <parameter name>: <expression>

When Python sees a **lambda** expression, it creates a function to be applied later. When this function is called, in the **read** or **write** method, the function's single parameter becomes the data object encased in the shared cell. The **lambda**'s expression is then evaluated in a critical section. This expression should contain an operation on the encased object. The operation's value is then returned. Although the construction and use of **lambda** expressions might seem challenging at first, they provide a very clean and powerful way to structure the shared cell abstraction for any readers and writers.

Implementing the Interface of the SharedCell Class

Two locks or conditions are needed to synchronize multiple readers and writers: one on which the readers wait and the other on which the writers wait. Two other data values belong to the shared cell's state: a Boolean value to indicate whether a writer is currently writing, and a counter to track the number of readers currently reading (remember that only one writer can be writing, but many readers can be concurrently reading).

Consequently, the instance variables of a **SharedCell** object include the shared data object named **data**, two conditions named **okToRead** and **okToWrite**, a Boolean variable named **writing**, and an integer variable named **readerCount**. Here is the code for the **__init__** method:

```
class SharedCell(object):
    """Synchronizes readers and writers around shared data,
    to support thread-safe reading and writing."""

    def __init__(self, data):
        """Sets up the conditions and the count of
        active readers."""
```



```

self.data = data
self.writing = False
self.readerCount = 0
self.okToRead = Condition()
self.okToWrite = Condition()

```

Note that the user of a shared cell object will pass the shared data to the cell when it is instantiated. This will allow the shared cell to be used for any kind of data that we want to make thread-safe for reading and writing.

The next step is to develop the code for accessing the shared cell for reading and writing. Recall that the **SharedCell** class for the producer-consumer problem includes the methods **setData** and **getData** for the use of the producer and consumer threads, respectively. For the readers and writers problem, there are two similar methods, named **read** and **write**, for the use of reader and writer threads (see Table 10-4). You'll also recall that the methods **getData** and **setData** have a similar structure. They each acquire access to a lock on the shared data, run a critical section of code, and then release the lock. The design of the methods **read** and **write** also has this pattern, as shown in the following pseudocode:

```

Acquire access to the two locks on the shared data
Perform actions on the data in the critical section
Release access to the two locks on the shared data

```

Because readers and writers have different mechanisms for acquiring and releasing the locks, we package this code in the helper methods **beginRead**, **endRead**, **beginWrite**, and **endWrite**. Likewise, the code to be executed in the critical section will vary with the application, as one can read or write in many different ways. Therefore, we package this code in a function that gets passed as an argument to the **read** and **write** methods. This function expects one argument, a data object of the type encased within the shared cell. The code of the function runs an accessor method on its argument for a reader, or runs a mutator method on this argument for a writer. In either case, the **read** or **write** method returns the result. Here is the code for the **SharedCell** methods **read** and **write**:

```

def read(self, readerFunction):
    """Observe the data in the shared cell."""
    self.beginRead()
    # Enter the reader's critical section
    result = readerFunction(self.data)
    # Exit the reader's critical section
    self.endRead()
    return result

def write(self, writerFunction):
    """Modify the data in the shared cell."""
    self.beginWrite()
    # Enter the writer's critical section
    result = writerFunction(self.data)
    # Exit the writer's critical section
    self.endWrite()
    return result

```

The beauty of these operations is their abstract and general character: they will work with any type of data we want to share among threads, and with any operations can observe (read) or modify (write) the shared data.

368

Implementing the Helper Methods of the `SharedCell` Class

Our final task is to tackle the implementation of the methods that acquire and release the locks for reading and writing. As in the producer-consumer problem, a thread will be either executing in the CPU (the current thread), active on the ready queue (ready), or asleep or waiting on a condition (blocked). When multiple threads wait on a condition, they go onto a queue associated with that condition. Python's `Condition` class has an instance variable, named `_waiters`, which refers to a condition's queue. Armed with this information, we can now consider the code for the methods `beginRead` and `endRead`, which acquire and release access to the critical section for readers.

In `beginRead`, the reader thread must wait on its condition if a writer is currently writing or writers are waiting on their condition. Otherwise, the reader is free to increment the count of active readers, notify the next reader waiting on its condition, and enter the critical section. Here is the code for method `beginRead`:

```
def beginRead(self):
    """Waits until a writer is not writing or the writers
    condition queue is empty. Then increments the reader
    count and notifies the next waiting reader."""
    self.okToRead.acquire()
    self.okToWrite.acquire()
    while self.writing or len(self.okToWrite._waiters) > 0:
        self.okToRead.wait()
    self.readerCount += 1
    self.okToRead.notify()
```

When a reader is finished in its critical section, the method `endRead` decrements the count of active readers. It then notifies the next waiting writer, if there are no active readers:

```
def endRead(self):
    """Notifies a waiting writer if there are
    no active readers."""
    self.readerCount -= 1
    if self.readerCount == 0:
        self.okToWrite.notify()
    self.okToWrite.release()
    self.okToRead.release()
```

Note that `beginRead` acquires both locks and `endRead` releases both locks.

The methods `beginWrite` and `endWrite` show a similar pattern. A writer can enter its critical section if there is no current writer and there are no active readers. When leaving its critical section, a writer notifies the next reader waiting on its condition, if there are any such readers. Otherwise, it notifies the next waiting writer. Here is the code for these two methods:

```

def beginWrite(self):
    """Can write only when someone else is not
    writing and there are no readers ready."""
    self.okToWrite.acquire()
    self.okToRead.acquire()
    while self.writing or self.readerCount != 0:
        self.okToWrite.wait()
    self.writing = True

def endWrite(self):
    """Notify the next waiting writer if the readers
    condition queue is empty. Otherwise, notify the
    next waiting reader."""
    self.writing = False
    if len(self.okToRead._waiters) > 0:
        self.okToRead.notify()
    else:
        self.okToWrite.notify()
self.okToRead.release()
self.okToWrite.release()

```

Testing the SharedCell Class with a Counter Object

Figure 10-4 shows a run of a tester program that creates a shared cell on a **Counter** object (discussed in Chapter 9).

```

tiger:pythonfiles lambertk$ python3 readersandwriters.py
Creating reader threads.
Creating writer threads.
Starting the threads.
Reader1 starting up
Reader2 starting up
Reader3 starting up
Reader4 starting up
Writer1 starting up
Writer2 starting up
Reader1 is done getting 0
Reader3 is done getting 0
Writer2 is done incrementing to 1
Reader2 is done getting 1
Reader4 is done getting 1
Writer1 is done incrementing to 2
tiger:pythonfiles lambertk$

```

Figure 10-4 A run of the readers and writers program

At start-up, the program wraps a shared cell around a new **Counter** object. The program then starts several reader and writer threads that access the shared cell. The readers print the current value of the counter, whereas the writers increment and print the updated value. As in our producer-consumer example, threads begin by sleeping a random interval, so they arrive at the shared cell in a random order. As you can see, all of the threads obtain access to the shared counter, and the counter retains its integrity throughout the process. The coding of this program, which is similar in structure to the producer-consumer program, is left as an exercise for you.

Defining a Thread-Safe Class

We mentioned earlier that Python data structures such as lists and dictionaries are thread-safe, but the data objects contained therein might not be. For example, the dictionary that contains the accounts in the **Bank** class of Chapter 9 is thread-safe, but the individual accounts, of type **SavingsAccount**, are not. How can we use the technology of our shared cell to fix this problem?

The solution is to apply a design pattern known as the **decorator pattern**. In this strategy, we define a new class that has the same interface or set of methods as the class that it “decorates.” Thus, programmers can substitute objects of this new class wherever they have used objects of the decorated class. Figure 10-5 shows the decorator relationship between two classes, **ThreadSafeSavingsAccount** and the class it decorates, **SavingsAccount**.

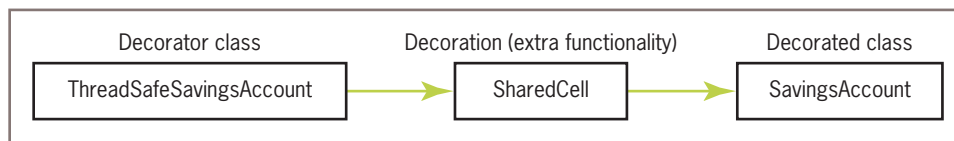


Figure 10-5 Using the decorator pattern

The new class encases an object of the decorated class, as well as other information necessary to accomplish its decoration. When the unsuspecting programmer calls a method on an object of the new class, the object behaves just as it did before, but with extra functionality—in this case, thread-safety. The beauty of this solution is that none of the code in the application must change, except for the name of the class being decorated. For example, applications that create instances of **SavingsAccount** need only change this name to **ThreadSafeSavingsAccount**, and they can make thread-safe accounts available to multiple readers and writers.

The code for the **ThreadSafeSavingsAccount** class (**threadsafesavingsaccount.py**) contains a **SharedCell** object, which in turn contains a **SavingsAccount** object. The constructor for **ThreadSafeSavingsAccount** creates a new **SavingsAccount** object and passes this to a new **SharedCell** object, as follows:

```

from savingsaccount import SavingsAccount
from sharedcell import SharedCell

class ThreadSafeSavingsAccount(object):
    """This class represents a thread-safe savings account
    with the owner's name, PIN, and balance."""
  
```

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```
def __init__(self, name, pin, balance = 0.0):
    """Wrap a new account in a shared cell for
    thread-safety."""
    account = SavingsAccount(name, pin, balance)
    self.cell = SharedCell(account)
```

The other methods in **ThreadSafeSavingsAccount** observe or modify the data in the account by running the **read** or **write** methods on the shared cell. For example, here is the code for the **getBalance** and **deposit** methods:

```
def getBalance(self):
    """Returns the current balance."""
    return self.cell.read(lambda account: account.getBalance())

def deposit(self, amount):
    """If the amount is valid, adds it
    to the balance and returns None;
    otherwise, returns an error message."""
    return self.cell.write(lambda account: account.deposit(amount))
```

The remaining methods in **ThreadSafeSavingsAccount** follow a similar pattern. The only change you need to make in the **bank** module is where you create accounts to test the module. For example, the function **createBank** now adds the new type of account with the statement **bank.add(ThreadSafeSavingsAccount(name, str(pinNumber), balance))**

Exercises

1. Give two real-world examples of the readers and writers problem.
2. State two ways in which the readers and writers problem is different from the producer-consumer problem.
3. Describe how you would make the **Student** class from Chapter 9 thread-safe for readers and writers.
4. Define a new class called **PCCell**. This class provides an abstraction of a shared cell for the producer-consumer problem. The design pattern should be similar for the one presented for the shared cell for readers and writers, but it should use the mechanism specific to the producer-consumer situation.

Networks, Clients, and Servers

Clients and servers are applications or processes that can run locally on a single computer or remotely across a **network** of computers. As explained in the following sections, the resources required for this type of application are IP addresses, sockets, and threads.

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

IP Addresses

Every computer on a network has a unique identifier called an **IP address** (IP stands for Internet Protocol). This address can be specified either as an **IP number** or as an **IP name**. An IP number typically has the form *ddd.ddd.ddd.ddd*, where each *d* is a digit. The number of digits to the right or the left of a decimal point may vary but does not exceed three. For example, the IP number of the author's office computer might be 137.112.194.77. Because IP numbers can be difficult to remember, people customarily use an IP name to specify an IP address. For example, the IP name of the author's computer might be **lambertk**.

Python's **socket** module includes two functions that can look up these items of information. These functions are listed in Table 10-5, followed by a short session showing their use.

socket Function	What It Does
gethostname()	Returns the IP name of the host computer running the Python interpreter. Raises an exception if the computer does not have an IP address.
gethostbyname(ipName)	Returns the IP number of the computer whose IP name is ipName . Raises an exception if ipName cannot be found.

Table 10-5 **socket** functions for IP addresses

```
>>> from socket import *
>>> gethostname()
'kenneth-lamberts-powerbook-g4-15.local'
>>> gethostbyname(gethostname())
'193.169.1.209'
>>> gethostbyname("Ken")
Traceback (most recent call last):
File "<pyshell#7>", line 1, in <module>
gethostbyname('Ken')
gaierror: (7, 'No address associated with nodename')
```

Note that these functions raise exceptions if they cannot locate the information. To handle this problem, one can embed these function calls in a **try-except** statement. The next code segment recovers from an unknown IP address error by printing the exception's error message:

```
try:
    print(gethostbyname('Ken'))
except Exception as exception:
    print(exception)
```

When developing a network application, the programmer can first try it out on a **local host**—that is, on a standalone computer that may or may not be connected to the Internet.

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

The computer's IP name in this case is "**localhost**", a name that is standard for any computer. The IP number of a computer that acts as a local host is distinct from its IP number as an **Internet host**, as shown in the next session:

```
>>> gethostbyname(gethostname())
'196.128.1.159'
>>> gethostbyname("localhost")
'127.0.0.1'
```

373

When the programmer is satisfied that the application is working correctly on a local host, the application can then be deployed on the Internet host simply by changing the IP address. In the discussion that follows, we use a local host to develop network applications.

Ports, Servers, and Clients

Clients connect to servers via objects known as **ports**. A port serves as a channel through which several clients can exchange data with the same server or with different servers. Ports are usually specified by numbers. Some ports are dedicated to special servers or tasks. For example, almost every computer reserves port number 13 for the day/time server, which allows clients to obtain the date and time. Port number 80 is reserved for a Web server, and so forth. Most computers also have hundreds or even thousands of free ports available for use by any network applications.

Sockets and a Day/Time Client Script

You can write a Python script that is a client to a server. To do this, you need to use a **socket**. A socket is an object that serves as a communication link between a single server process and a single client process. You can create and open several sockets on the same port of a host computer. Figure 10-6 shows the relationships between a host computer, ports, servers, clients, and sockets.

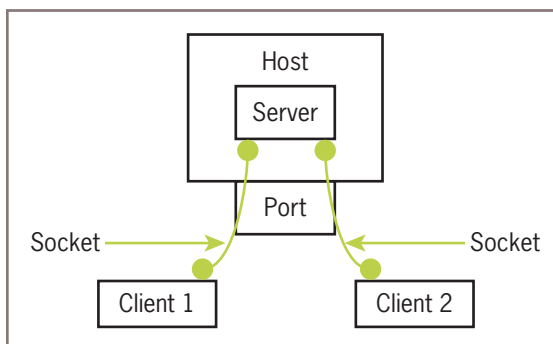


Figure 10-6 Setup of day/time host and clients

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

A Python day/time client script uses the **socket** module introduced earlier. This script does the following:

- Creates a socket object.
- Opens the socket on a free port of the local host. We use a large number, 5000, for this port.
- Reads and decodes the day/time from the socket.
- Displays the day/time.

Here is a Python script that performs these tasks:

```
"""
Client for obtaining the day and time.
"""
from socket import *
from codecs import decode

HOST = "localhost"
PORT = 5000
BUFSIZE = 1024
ADDRESS = (HOST, PORT)

server = socket(AF_INET, SOCK_STREAM)
server.connect(ADDRESS)
dayAndTime = decode(server.recv(BUFSIZE), "ascii")
print(dayAndTime)
server.close()
```

Although we cannot run this script until we write and launch the server program, Figure 10-7 shows the client's anticipated output.

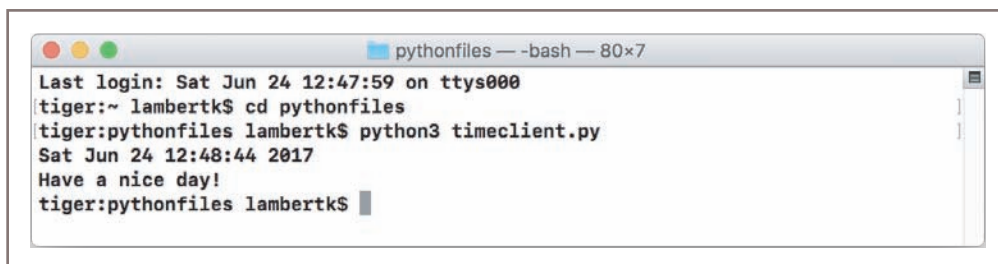


Figure 10-7 The user interface of the day/time client script

As you can see, a Python socket is fairly easy to set up and use. A socket resembles a file object, in that the programmer opens it, receives data from it, and closes it when finished. We now explain these steps in our client script in more detail.

The script creates a socket by running the function **socket** in the **socket** module. This function returns a new socket object, when given a socket family and a socket type as arguments. We use the family **AF_INET** and the type **SOCK_STREAM**, both **socket** module constants, in all of our examples.

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

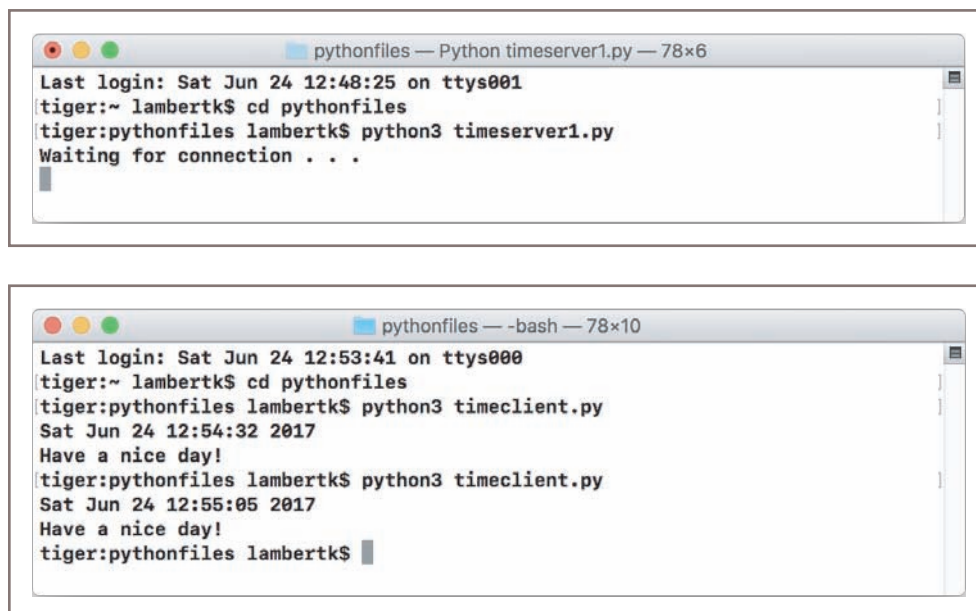
To connect the socket to a host computer, one runs the socket's **connect** method. This method expects as an argument a tuple containing the host's IP address and a port number. In this case, these values are **"localhost"** and 5000, respectively. These two values should be the same as the ones used in the server script.

To obtain information sent by the server, the client script runs the socket's **recv** method. This method expects as an argument the maximum size in bytes of the data to be read from the socket. The **recv** method returns an object of type **bytes**. You convert this to a string by calling the **codecs** function **decode**, with the encoding **"ascii"** as the second argument.

After the client script has printed the string read from the socket, the script closes the connection to the server by running the socket's **close** method.

A Day/Time Server Script

You can also write a day/time server script in Python to handle requests from many clients. Figure 10-8 shows the interaction between a day/time server and two clients in a series of screenshots. In the first shot, the day/time server script is launched in a terminal window, and it's waiting for a connection. In the second shot, two successive clients are launched in a separate terminal window (you can open several terminal windows at once). They have connected to the server and have received the day/time. The third shot shows the updates to the server's window after it has served these two clients. Note that the two clients terminate execution after they print their results, whereas the server appears to continue waiting for another client.



The figure consists of two terminal window screenshots. The top window, titled 'pythonfiles — Python timeserver1.py — 78x6', shows the server script running. It displays the last login time, the current directory, and the command to run the script. The script is now waiting for a connection. The bottom window, titled 'pythonfiles — -bash — 78x10', shows two separate client sessions. Each session runs the 'timeclient.py' script, which receives the day and time from the server and prints it. The first client session shows the time as 'Sat Jun 24 12:54:32 2017' and 'Have a nice day!'. The second client session shows the time as 'Sat Jun 24 12:55:05 2017' and 'Have a nice day!'. Both client sessions terminate after printing the results.

```

pythonfiles — Python timeserver1.py — 78x6
Last login: Sat Jun 24 12:48:25 on ttys001
tiger:~ lambertk$ cd pythonfiles
tiger:pythonfiles lambertk$ python3 timeserver1.py
Waiting for connection . . .

pythonfiles — -bash — 78x10
Last login: Sat Jun 24 12:53:41 on ttys000
tiger:~ lambertk$ cd pythonfiles
tiger:pythonfiles lambertk$ python3 timeclient.py
Sat Jun 24 12:54:32 2017
Have a nice day!
tiger:pythonfiles lambertk$ python3 timeclient.py
Sat Jun 24 12:55:05 2017
Have a nice day!
tiger:pythonfiles lambertk$

```

Figure 10-8 A day/time server and two clients

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```

pythonfiles — Python timeserver1.py — 78x10
Last login: Sat Jun 24 12:48:25 on ttys001
tiger:~ lambertk$ cd pythonfiles
tiger:pythonfiles lambertk$ python3 timeserver1.py
Waiting for connection . . .
... connected from: ('127.0.0.1', 50243)
Waiting for connection . . .
... connected from: ('127.0.0.1', 50244)
Waiting for connection . . .

```

Figure 10-8 (Continued)

A Python day/time server script also uses the resources of the **socket** module. The basic sequence of operations for a simple day/time server script is the following:

```

Create a socket and open it on port 5000 of the local host
While true
    Wait for a connection from a client
    When the connection is made,
        send the date to the client

```

Our script also displays information about the host, the port, and the client. Here is the code, followed by a brief explanation:

```

"""
Server for providing the day and time.
"""
from socket import *
from time import ctime

HOST = "localhost"
PORT = 5000
ADDRESS = (HOST, PORT)

server = socket(AF_INET, SOCK_STREAM)
server.bind(ADDRESS)
server.listen(5)
while True:
    print("Waiting for connection ...")
    (client, address) = server.accept()
    print("... connected from: ", address)
    client.send(bytes(ctime() + "\nHave a nice day!",
                     "ascii"))
    client.close()

```

The server script uses the same information to create a socket object as the client script presented earlier. In particular, the IP address and port number must be *exactly* the same as they are in the client's code.

However, connecting the socket to the host and to the port so as to become a server socket is done differently. First, the socket is bound to this address by running its **bind** method.

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Second, the socket then is made to listen for up to five requests at a time from clients by running its **listen** method. If you want the server to handle more concurrent requests before rejecting additional ones, you can increase this number.

After the script enters its main loop, it prints a message indicating that it is waiting for a connection. The socket's **accept** method then pauses execution of the script, in a manner similar to Python's **input** function, to wait for a request from a client.

When a client connects to this server, **accept** returns a tuple containing the client's socket and its address information. Our script binds the variables **client** and **address** to these values and uses them in the next steps.

The script prints the client's address, and then sends the current day/time to the client by running the **send** method with the client's socket. The **send** method expects a **bytes** object as an argument. You create a **bytes** object from a string by calling the built-in **bytes** function, with the string and an encoding, in this case, **"ascii"**, as arguments. The Python function **time.ctime** returns a string representing the day/time.

Finally, the script closes the connection to the client by running the client socket's **close** method. The script then returns in its infinite loop to accept another client connection.

A Two-Way Chat Script

The communication between the day/time server and its client is one-way. The client simply receives a message from the server and then quits. In a two-way chat, the client connects to the server, and the two programs engage in a continuous communication until one of them, usually the client, decides to quit.

Once again, there are two distinct Python scripts, one for the server and one for the client. The setup of a two-way chat server is similar to that of the day/time server discussed earlier. The server script creates a socket with a given IP address and port and then enters an infinite loop to accept and handle clients. When a client connects to the server, the server sends the client a greeting.

Instead of closing the client's socket and listening for another client connection, the server then enters a second, nested loop. This loop engages the server in a continuous conversation with the client. The server receives a message from the client. If the message is an empty string, the server displays a message that the client has disconnected, closes the client's socket, and breaks out of the nested loop. Otherwise, the server prints the client's message and prompts the user for a reply to send to the client.

Here is the code for the two loops in the server script:

```
CODE = "ascii"

while True:
    print("Waiting for connection ...")
    client, address = server.accept()
    print("... connected from: ", address)
    client.send(bytes("Welcome to my chat room!", CODE))
```

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

```

while True:
    message = decode(client.recv(BUFSIZE), CODE)
    if not message:
        print("Client disconnected")
        client.close()
        break
    else:
        print(message)
        client.send(bytes(input("> "), CODE))

```

The client script for the two-way chat sets up a socket in a similar manner to the day/time client. After the client has connected to the server, it receives and displays the server's initial greeting message.

Instead of closing the server's socket, the client then enters a loop to engage in a continuous conversation with the server. This loop mirrors the loop that is running in the server script. The client's loop prompts the user for a message to send to the server. If this string is empty, the loop breaks. Otherwise, the client sends the message to the server's socket and receives the server's reply. If this reply is the empty string, the loop also breaks. Otherwise, the server's reply is displayed. The server's socket is closed after the loop has terminated. Here is the code for the part of the client script following the client's connection to the server:

```

print(decode(server.recv(BUFSIZE), CODE))
while True:
    message = input("> ")
    if not message:
        break
    server.send(bytes(message, CODE))
    reply = decode(server.recv(BUFSIZE), CODE)
    if not reply:
        print("Server disconnected")
        break
    print(reply)
server.close()

```

As you can see, it is important to synchronize the sending and the receiving of messages between the client and the server. If you get this right, the conversation can proceed, usually without a hitch.

Handling Multiple Clients Concurrently

The client/server programs that we have discussed thus far are rather simple and limited. First, the server handles a client's request and then returns to wait for another client. In the case of the day/time server, the processing of each request happens so quickly that clients will never notice a delay. However, when a server provides extensive processing, other clients will have to wait until the currently connected client is finished.

To solve the problem of giving many clients timely access to the server, we relieve the server of the task of handling the client's request and assign it instead to a separate client-handler thread. Thus, the server simply listens for client connections and dispatches these to new client-handler objects. The structure of this system is shown in Figure 10-9.

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

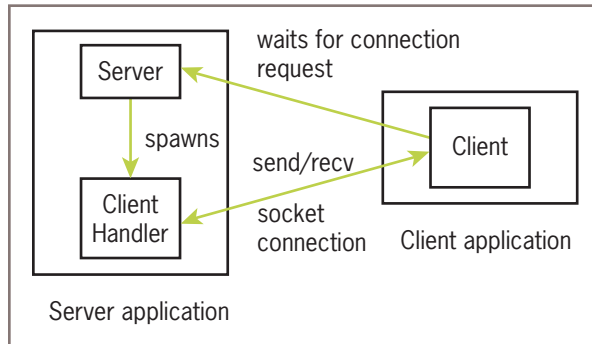


Figure 10-9 A day/time server with a client handler

The use of separate server and client handler objects accomplishes two things in this design:

1. The details of fielding a request for service are separated from the details of performing that service, making the design of each task simpler and more maintainable.
2. Because the server object and the client handler objects run in separate threads, their processes can run concurrently. This means that new clients will not have to wait for service until a connected client has been served (think of a busy server running for Google or Amazon, with hundreds of millions of clients being served simultaneously).

Returning to the day/time server script, we now add a client handler to improve efficiency. This handler is an instance of a new class, **TimeClientHandler**, which is defined in its own module. This class extends the **Thread** class. Its constructor receives the client's socket from the server and assigns it to an instance variable. The **run** method includes the code to send the date to the client and close its socket. Here is the code for the **TimeClientHandler** class:

```

"""
File: timeclienthandler.py
Client handler for providing the day and time.
"""
from time import ctime
from threading import Thread

class TimeClientHandler(Thread):
    """Handles a client request."""

    def __init__(self, client):
        Thread.__init__(self)
        self.client = client

    def run(self):
        self.client.send(bytes(ctime() + \
                               "\nHave a nice day!",
                               "ascii"))
        self.client.close()
  
```

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

The code for the server's script now imports the `TimeClientHandler` class. The server creates a socket and listens for requests, as before. However, when a request comes in, the server creates a client socket and passes it to a new instance of `TimeClientHandler` for processing. The server then immediately returns to listen for new requests. Here is the code for the modified day/time server:

```
"""
File: timeserver2.py
Server for providing the day and time. Uses client
handlers to handle clients' requests.
"""
from socket import socket
from timeClientHandler import TimeClientHandler

HOST = "localhost"
PORT = 5000
ADDRESS = (HOST, PORT)

server = socket(AF_INET, SOCK_STREAM)
server.bind(ADDRESS)
server.listen(5)
# The server now just waits for connections from clients
# and hands sockets off to client handlers
while True:
    print("Waiting for connection ... ")
    client, address = server.accept()
    print("... connected from: ", address)
    handler = TimeClientHandler(client)
    handler.start()
```

The code for the day/time client's script does not change at all. Moreover, to create a new server for different kind of service, you just define a new type of client handler and use it in the code for the server just presented.

Exercises

1. Explain the role that ports and IP addresses play in a client/server program.
2. What is a local host, and how is it used to develop networked applications?
3. Why is it a good idea for a server to create threads to handle clients' requests?
4. Describe how a menu-driven command processor of the type developed for an ATM application in Chapter 9 could be run on a network.
5. The ATM application discussed in Chapter 9 has a single user. Will there be a synchronization problem if we deploy that application with threads for multiple users? Justify your answer.

6. The servers discussed in this section all contain infinite loops. Thus, the applications running them cannot do anything else while the server is waiting for a client's request, and they cannot even gracefully be shut down. Suggest a way to restructure these applications so that the applications can do other things, including performing a graceful shutdown.

CASE STUDY: Setting Up Conversations between Doctors and Patients

Now that we have modified the day/time server to handle multiple clients, can we also modify the two-way chat program to support chats among multiple clients? Let us consider first the problem of supporting multiple two-way chats. We don't want to involve the server in the chat, much less the human user who is running the server. Can we first set up a chat between a human user and an automated agent? The doctor program developed in a Case Study in Chapter 5 is a good example of an automated agent or **bot** that chats with its client, who is a human user.

Request

Write a program that allows multiple clients to be served by doctors who provide non-directive psychotherapy.

Analysis

A doctor server program listens for requests from clients for doctors. Upon receiving a request, the server dispatches the client's socket to a new handler thread. This thread creates a new **Doctor** object (see Programming Project 5 in Chapter 9) and then manages the conversation between the doctor and the client. The server returns to field more requests from clients for sessions with their doctors. Figure 10-10 shows the structure of this program.

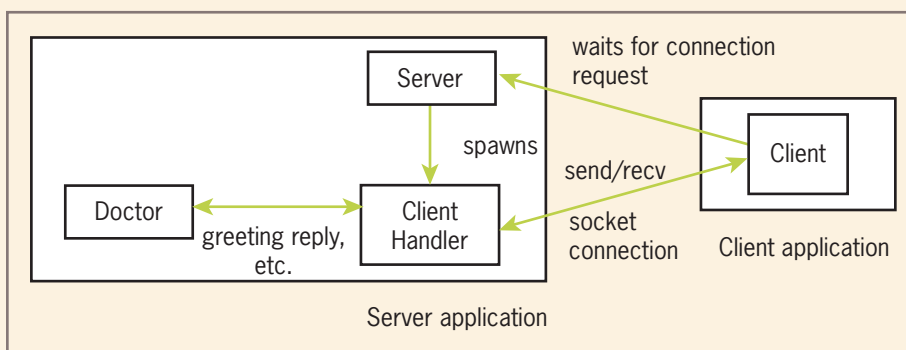


Figure 10-10 The structure of a client/server program for patients and doctors

(continues)

(continued)

382

The user interface for the server script is terminal-based, as you have seen in our other examples. The client script provides a GUI for clients, as shown in Figure 10-11. The GUI provides widgets for the user's inputs and the doctor's replies, and a button to connect or disconnect to the server.

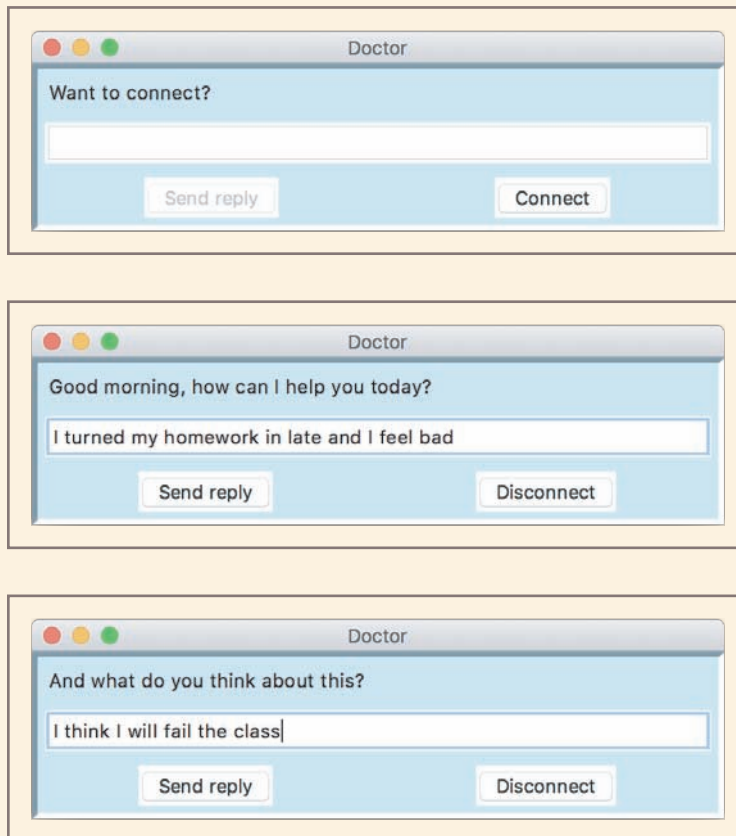


Figure 10-11 The user interface of clients in the doctor program

Design and Implementation

The design of the server script is the same as that of the multithreaded day/time server, but it now uses a **DoctorClientHandler** class to be developed shortly.

In the code that follows, we assume that a **Doctor** class is defined in the module **doctor.py**. This class includes two methods. The method **greeting** returns a string representing the doctor's welcome. The method **reply** expects the patient's string as an argument and returns the doctor's response string.

(continues)

(continued)

The client handler resembles the day/time client handler, but it includes the following changes:

- The client handler's `__init__` method creates a **Doctor** object and assigns it to an extra instance variable.
- The client handler's `run` method includes a conversation management loop similar to the one in the chat server. However, when the client handler receives a message from the client socket, this message is sent to the **Doctor** object rather than being displayed in the server's terminal window. Then, instead of taking input from the server's keyboard and sending it to the client, the client handler obtains this reply from the **Doctor** object.

Here is the code for the client handler:

```

"""
File: doctorclienthandler.py
Client handler for a therapy session. Handles multiple clients
concurrently.
"""
from codecs import decode
from threading import Thread
from doctor import Doctor

BUFSIZE = 1024
CODE = "ascii"

class DoctorClientHandler(Thread):
    """Handles a session between a doctor and a patient."""

    def __init__(self, client):
        Thread.__init__(self)
        self.client = client
        self.dr = Doctor()

    def run(self):
        self.client.send(bytes(self.dr.greeting(), CODE))
        while True:
            message = decode(self.client.recv(BUFSIZE), CODE)
            if not message:
                print("Client disconnected")
                self.client.close()
                break
            else:
                self.client.send(bytes(self.dr.reply(message),
                                         CODE))

```

The `doctorclient` module includes the code for the GUI and the code for managing the connection to the server. When the user clicks the **Connect** button, the program

(continues)

(continued)

384

connects to the server, as in previous examples. It then receives and displays the doctor's greeting and waits for the user's input. The user replies by entering text in an input field and clicking the **Send** button. The user signals the end of a session by clicking the **Disconnect** button, which closes the server's socket.

Here is the code for the client, which includes the class **DoctorClient**

```

"""
File: doctorclient.py
GUI-based view for client for nondirective psychotherapy.
"""

from socket import *
from codecs import decode
from breezypythongui import EasyFrame

HOST = "localhost"
PORT = 5000
BUFSIZE = 1024
ADDRESS = (HOST, PORT)
CODE = "ascii"

class DoctorClient(EasyFrame):
    """Represents the client's window."""

    COLOR = "#CCEEFF"          # Light blue

    def __init__(self):
        """Initialize the window and widgets."""
        EasyFrame.__init__(self, title = "Doctor",
                           background = DoctorClient.COLOR)
        # Add the labels, fields, and buttons
        self.drLabel = self.addLabel("Want to connect?",
                                     row = 0, column = 0,
                                     colspan = 2,
                                     background = DoctorClient.COLOR)
        self.ptField = self.addTextField(text = "",
                                         row = 1,
                                         column = 0,
                                         colspan = 2,
                                         width = 50)
        self.sendBtn = self.addButton(row = 2, column = 0,
                                     text = "Send",
                                     command = self.sendReply,
                                     state = "disabled")
        self.connectBtn = self.addButton(row = 2,
                                         column = 1,
                                         text = "Connect",
                                         command = self.connect)

```

(continues)

(continued)

```

# Support the return key in the input field
self.ptField.bind("<Return>",
                  lambda event: self.sendReply())

def sendReply(self):
    """Sends patient input to doctor, receives
    and outputs the doctor's reply."""
    ptInput = self.ptField.getText()
    if ptInput != "":
        self.server.send(bytes(ptInput, CODE))
        drReply = decode(self.server.recv(BUFSIZE),
                           CODE)

        if not drReply:
            self.messageBox(message = "Doctor disconnected")
            self.disconnect()
        else:
            self.drLabel["text"] = drReply
            self.ptField.setText("")

def connect(self):
    """Starts a new session with the doctor."""
    self.server = socket(AF_INET, SOCK_STREAM)
    self.server.connect(ADDRESS)
    self.drLabel["text"] = decode(self.server.recv(BUFSIZE),
                                   CODE)

    self.connectBtn["text"] = "Disconnect"
    self.connectBtn["command"] = self.disconnect
    self.sendBtn["state"] = "normal"

def disconnect(self):
    """Ends the session with the doctor."""
    self.server.close()
    self.ptField.setText("")
    self.drLabel["text"] = ""
    self.connectBtn["text"] = "Connect"
    self.connectBtn["command"] = self.connect
    self.sendBtn["state"] = "disabled"

def main():
    """Instantiate and pop up the window."""
    DoctorClient().mainloop()

if __name__ == "__main__":
    main()

```

You might have noticed that each client interacts with its own **Doctor** object. Thus, no synchronization problems arise when a patient's replies are added to the doctor's history list, because the client threads do not access any shared data.

(continues)

(continued)

386

However, in other applications, such as the ATM developed in Chapter 9, concurrent users would be accessing shared data. In the case of the ATM application, the server would create the common **Bank** object and pass it to the client handlers. Because this object holds the accounts in a dictionary and dictionaries are thread-safe, additions or removals of accounts pose no synchronization problems. However, the **SavingsAccount** objects within the **Bank** object's dictionary are not themselves thread-safe, and thus they could cause synchronization problems when two or more users access a joint account. The solution is to provide a lock and condition mechanism for a **SavingsAccount** object, to allow concurrent access to readers and writers of that shared object. You will work with shared data in client server applications in the programming projects.

Summary

- Threads allow the work of a single program to be distributed among several computational processes. These processes may be run concurrently on the same computer or may collaborate by running on separate computers.
- A thread can have several states during its lifetime, such as newborn, ready, executing (in the CPU), sleeping, and waiting. A ready queue schedules the threads for access to the CPU in first-come, first-served order.
- After a thread is started, it goes to the end of the ready queue to be scheduled for a turn in the CPU.
- A thread may give up the CPU when it is timed-out, goes to sleep, waits on a condition, or finishes its **run** method.
- When a thread wakes up, is timed-out, or is notified that it can stop waiting, it returns to the rear of the ready queue.
- Thread synchronization problems can occur when two or more threads share data. These threads can be synchronized by waiting on conditions that control access to the data.
- Each computer on a network has a unique IP address that allows other computers to locate it. An IP address contains an IP number but can also be labeled with an IP name.
- Servers and clients can communicate on a network by means of sockets. A socket is created with a port number and an IP address of the server on the client's computer and on the server's computer.
- Clients and servers communicate by sending and receiving bytes through their socket connections. A string is converted to bytes before being sent, and the bytes are converted to a string after receipt.
- A server can handle several clients concurrently by assigning each client request to a separate handler thread.

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

Review Questions

1. Multiple threads can run on the same desktop computer by means of
 - a. time-sharing
 - b. multiprocessing
 - c. distributed computing
2. A **Thread** object moves to the ready queue when
 - a. its **wait** method is called
 - b. its **sleep** method is called
 - c. its **start** method is called
3. The method that executes a thread's code is called
 - a. the **start** method
 - b. the **run** method
 - c. the **execute** method
4. A lock on a resource is provided by an instance of the
 - a. **Thread** class
 - b. **Condition** class
 - c. **Lock** class
5. If multiple threads share data, they can have
 - a. total cooperation
 - b. synchronization problems
6. The object that uniquely identifies a host computer on a network is a(n)
 - a. port
 - b. socket
 - c. IP address
7. The object that allows several clients to access a server on a host computer is a(n)
 - a. port
 - b. socket
 - c. IP address
8. The object that effects a connection between an individual client and a server is a(n)
 - a. port
 - b. socket
 - c. IP address

9. The data that are transmitted between client and server are
 - a. of any type
 - b. strings
10. The best way for a server to handle requests from multiple clients is to
 - a. directly handle each client's request
 - b. create a separate client-handler thread for each client

Projects

1. Redo the producer/consumer program so that it allows multiple consumers. Each consumer must be able to consume the same data before the producer produces more data.
2. Sometimes servers are down, so clients cannot connect to them. Python raises an exception of type **ConnectionRefusedError** in a client program when a network connection is refused. Add code to the day/time client program to catch and recover from this kind of exception.
3. Modify the code in the day/time server application so that the user on the server side can shut the server down. That user should be able to press the return or enter key at the terminal to do this.
4. Modify the doctor application discussed in this chapter so that it tracks clients by name and history. A **Doctor** object has its own history list of a patient's inputs for generating replies that refer to earlier conversations, as discussed in Chapter 5. A **Doctor** object is now associated with a patient's name. The client application takes this name as input and sends it to the client handler when the patient connects. The client handler checks for a pickled file with the patient's name as its filename ("**<patient name>.dat**"). If that file exists, it will contain the patient's history, and the client handler loads the file to create the **Doctor** object. Otherwise, the patient is visiting the doctor for the first time, so the client handler creates a brand-new **Doctor** object. When the client disconnects, the client handler pickles the **Doctor** object in a file with the patient's name.
5. Design, implement, and test a network application that maintains an online phone book. The data model for the phone book is saved in a file on the server's computer. Clients should be able to look up a person's phone number or add a name and number to the phone book. The server should handle multiple clients without delays. Unlike the doctor program, there should be just one phone book that all clients share. The server creates this object at start-up and passes it to the client handlers.
6. Convert the ATM application presented in Chapter 9 to a networked application. The client manages the user interface, whereas the server and client handler manage connecting to and the transactions with the bank. Do not be concerned about synchronization problems in this project.

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

7. Write the tester program for readers and writers of a shared **Counter** object. A sample run is shown in Figure 10-4.
8. Add synchronization to the ATM program of Project 6. You will need to give concurrent readers access to a single account, as long as a writer is not writing to it, and give a single writer access, as long as other writers and readers are not accessing the account. *Hint:* just complete the **ThreadSafeSavingsAccount** class discussed in this chapter, and use it to create account objects in the **Bank** class.
9. Jack has been working on the shared cell classes for the producer-consumer problem and the readers and writers problem, and he notices some serious redundancy in the code. The **read** and **write** methods are the same in both classes, and both classes include an instance variable for the data. Jill, his team manager, advises him to place this redundant code in a parent class named **SharedCell**. Then two subclasses, named **PCSharedCell** and **RWSharedCell**, can inherit this code and define the methods **beginRead**, **endRead**, **beginWrite**, and **endWrite**, to enforce their specific synchronization protocols. Also, the **__init__** method in each subclass first calls the **__init__** method in the **SharedCell** class to set up the data, and then adds the condition(s) and other instance variables for its specific situation. Jack has called in sick, so you must complete this hierarchy of classes and redo the demo programs so that they use them.
10. A crude multi-client chat room allows two or more users to converse by sending and receiving messages. On the client side, a user connects to the chat room as in the ATM application, by clicking a **Connect** button. At that point, a transcript of the conversation thus far appears in a text area. At any time, the user can send a message to the chat room by entering it as input and clicking a **Send** button. When the user sends a message, the chat room returns another transcript of the entire conversation to display in the text area. The user disconnects by clicking the **Disconnect** button.

On the server side, there are five resources: a server, a client handler, a transcript, a thread-safe transcript, and a shared cell. Their roles are much the same as they are in the ATM application of Project 8. The server creates a thread-safe transcript at start-up, listens for client connections, and passes a client's socket and the thread-safe transcript to a client handler when a client connects. The client handler receives the client's name from the client socket, adds this name and the connection time to the thread-safe transcript, sends the thread-safe transcript's string to the client, and waits for a reply. When the client's reply comes in, the client handler adds the client's name and time to it, adds the result to the thread-safe transcript, and sends the thread-safe transcript's string back to the client. When the client disconnects, her name and a message to that effect are added to the thread-safe transcript.

The **SharedCell** class includes the usual **read** and **write** methods for a readers and writers protocol, and the **SharedTranscript** and **Transcript** classes include an **add** method and an **__str__** method. The **add** method adds a string to a list of strings, while **__str__** returns the **join** of this list, separated by newlines.

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. WCN 02-200-203

Copyright 2019 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.