3
Process-based Parallelism

In this chapter, we will cover the following recipes:

- Using the `multiprocessing` Python module
- How to spawn a process
- How to name a process
- How to run a process in the background
- How to kill a process
- How to use a process in a subclass
- How to exchange objects between processes
- Using a queue to exchange objects
- Using pipes to exchange objects
- How to synchronize processes
- How to manage a state between processes
- How to use a process pool
- Using the `mpi4py` Python module
- Point-to-point communication
- Avoiding deadlock problems
- Collective communication using `broadcast`
- Collective communication using a `scatter` function

- Collective communication using a `gather` function
- Collective communication using `AlltoAll`
- Reduction operation
- How to optimize the communication

Introduction

In the previous chapter, we saw how to use threads to implement concurrent applications. This section will examine the process-based approach. In particular, the focus is on two libraries: the Python `multiprocessing` module and the Python `mpi4py` module.

The Python `multiprocessing` library, which is part of the standard library of the language, implements the shared memory programming paradigm, that is, the programming of a system that consists of one or more processors that have access to a common memory.

The Python library `mpi4py` implements the programming paradigm called message passing. It is expected that there are no shared resources (and this is also called shared nothing) and that all communications take place through the messages that are exchanged between the processes.

For these features, it is in contrast with the techniques of communication that provide memory sharing and the use of lock or similar mechanisms to achieve mutual exclusion. In a message passing code, the processes are connected via the communication primitives of the types `send()` and `receive()`.

In the introduction of the Python multiprocessing docs, it is clearly mentioned that all the functionality within this package requires the main module to be importable to the children (`https://docs.python.org/3.3/library/multiprocessing.html`).

The `__main__` module is not importable to the children in IDLE, even if you run the script as a file with IDLE. To get the correct result, we will run all the examples from the Command Prompt:

```
python multiprocessing_example.py
```

Here, `multiprocessing_example.py` is the script's name. For the examples described in this chapter, we will refer to the Python distribution 3.3 (even though Python 2.7 could be used).

How to spawn a process

The term "spawn" means the creation of a process by a parent process. The parent process can of course continue its execution asynchronously or wait until the child process ends its execution. The multiprocessing library of Python allows the spawning of a process through the following steps:

1. Build the object process.
2. Call its `start()` method. This method starts the process's activity.
3. Call its `join()` method. It waits until the process has completed its work and exited.

How to do it...

This example shows you how to create a series (five) of processes. Each process is associated with the function `foo(i)`, where `i` is the ID associated with the process that contains it:

```
#Spawn a Process: Chapter 3: Process Based Parallelism
import multiprocessing

def foo(i):
    print ('called function in process: %s' %i)
    return

if __name__ == '__main__':
    Process_jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=foo, args=(i,))
        Process_jobs.append(p)
        p.start()
        p.join()
```

To run the process and display the results, let's open the Command Prompt, preferably in the folder containing the example file (named `spawn_a_process.py`), and then type the following command:

```
python spawn_a_process.py
```

We obtain the following output using this command:

```
C:\Python CookBook\ Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python spawn_a_process.py
called function in process: 0
called function in process: 1
called function in process: 2
called function in process: 3
called function in process: 4
```

How it works...

As explained in the introduction section of this recipe, to create the object process, we must first import the multiprocessing module with the following command:

```
import multiprocessing
```

Then, we create the object process in the main program:

```
p = multiprocessing.Process(target=foo, args=(i,))
```

Further, we call the `start()` method:

```
p.start()
```

The object process has for argument the function to which the child process is associated (in our case, the function is called `foo()`). We also pass an argument to the function that takes into account the process in which the associated function is situated. Finally, we call the `join()` method on the process created:

```
p.join()
```

Without `p.join()`, the child process will sit idle and not be terminated, and then, you must manually kill it.

There's more...

This reminds us once again of the importance of instantiating the `Process` object within the main section:

```
if __name__ == '__main__':
```

This is because the child process created imports the script file where the target function is contained. Then, by instantiating the process object within this block, we prevent an infinite recursive call of such instantiations. A valid workaround is used to define the target function in a different script, and then imports it to the namespace. So for our first example, we could have:

```
import multiprocessing
import target_function

if __name__ == '__main__':
    Process_jobs = []
    for i in range(5):
        p = multiprocessing.Process   \
            (target=target_function.function,args=(i,))
         Process_jobs.append(p)
         p.start()
         p.join()
```

Here, `target_function.py` is as shown:

```
#target_function.py

def function(i):
    print ('called function in process: %s' %i)
    return
```

The output is always similar to that shown in the preceding example.

How to name a process

In the previous example, we identified the processes and how to pass a variable to the target function. However, it is very useful to associate a name to the processes as debugging an application requires the processes to be well marked and identifiable.

How to do it...

The procedure to name a process is similar to that described for the threading library (see the recipe *How to determine the current thread* in *Chapter 2*, *Thread-based Parallelism*, of the present book.)

In the main program, we create a process with a name and a process without a name. Here, the common target is the `foo()` function:

```
#Naming a Process: Chapter 3: Process Based Parallelism
import multiprocessing
import time

def foo():
    name = multiprocessing.current_process().name
    print ("Starting %s \n" %name)
    time.sleep(3)
    print ("Exiting %s \n" %name)

if __name__ == '__main__':
    process_with_name = \
                      multiprocessing.Process\
                      (name='foo_process',\
                       target=foo)
    process_with_name.daemon = True
    process_with_default_name = \
                              multiprocessing.Process\
                              (target=foo)
    process_with_name.start()
    process_with_default_name.start()
```

To run the process, open the Command Prompt and type the following command:

```
python naming_process.py
```

This is the result that we get after using the preceding command:

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python naming_process.py
Starting foo_process
Starting Process-2
Exiting foo_process
Exiting Process-2
```

How it works...

The operation is similar to the procedure used for naming a thread. To name a process, we should provide an argument with the object's name:

```
process_with_name = multiprocessing.Process
          (name='foo_function', target=foo)
```

In this case, we called the `foo_function` process. If the process child wants to know which its parent process is, it must use the following statement:

```
name = multiprocessing.current_process().name
```

This statement will provide the name of the parent process.

How to run a process in the background

Running a process in background is a typical mode of execution of laborious processes that do not require your presence or intervention, and this course may be concurrent to the execution of other programs. The Python multiprocessing module allows us, through the daemonic option, to run background processes.

How to do it...

To run a background process, simply follow the given code:

```
import multiprocessing
import time

def foo():
    name = multiprocessing.current_process().name
    print ("Starting %s \n" %name)
    time.sleep(3)
    print ("Exiting %s \n" %name)


if __name__ == '__main__':
    background_process = multiprocessing.Process\
                         (name='background_process',\
                          target=foo)
    background_process.daemon = True
```

```
    NO_background_process = multiprocessing.Process\
                            (name='NO_background_process',\
                             target=foo)

    NO_background_process.daemon = False

    background_process.start()
    NO_background_process.start()
```

To run the script from the Command Prompt, type the following command:

```
python background_process.py
```

The final output of this command is as follows:

```
C:\Python CookBook\ Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python background_process.py


Starting NO_background_process
Exiting NO_background_process
```

How it works...

To execute the process in background, we set the `daemon` parameter:

```
background_process.daemon = True
```

The processes in the no-background mode have an output, so the daemonic process ends automatically after the main program ends to avoid the persistence of running processes.

There's more...

Note that a daemonic process is not allowed to create child processes. Otherwise, a daemonic process would leave its children orphaned if it gets terminated when its parent process exits. Additionally, these are not Unix daemons or services, they are normal processes that will be terminated (and not joined) if non-daemonic processes have exited.

How to kill a process

It's possible to kill a process immediately using the `terminate()` method. Also, we use the `is_alive()` method to keep track of whether the process is alive or not.

How to do it...

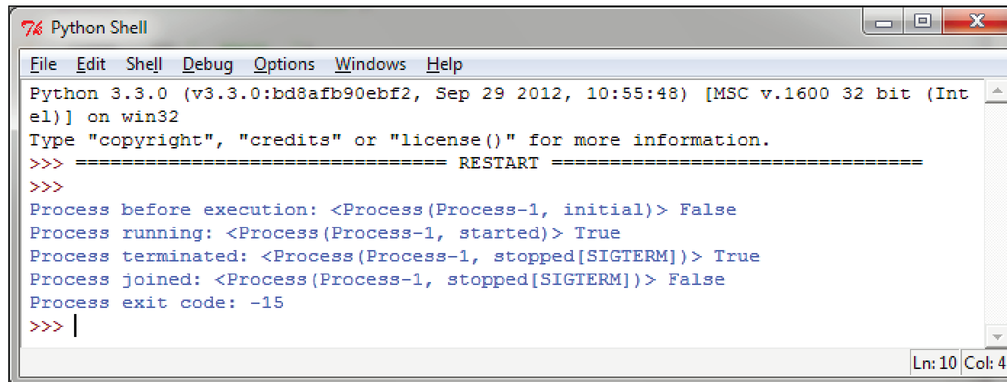
In this example, a process is created with the target function `foo()`. After the start, we kill it with the `terminate()` function:

```
#kill a Process: Chapter 3: Process Based Parallelism
import multiprocessing
import time

def foo():
    print ('Starting function')
    time.sleep(0.1)
    print ('Finished function')

if __name__ == '__main__':
    p = multiprocessing.Process(target=foo)
    print ('Process before execution:', p, p.is_alive())
    p.start()
    print ('Process running:', p, p.is_alive())
    p.terminate()
    print ('Process terminated:', p, p.is_alive())
    p.join()
    print ('Process joined:', p, p.is_alive())
    print ('Process exit code:', p.exitcode)
```

The following is the output we get when we use the preceding command:



How it works...

We create the process and then monitor its lifetime by the `is_alive()` method. Then, we finish it with a call to `terminate()`:

```
p.terminate()
```

Finally, we verify the status code when the process is finished, and read the attribute of the `ExitCode` process. The possible values of `ExitCode` are, as follows:

- `== 0`: This means that no error was produced
- `> 0`: This means that the process had an error and exited that code
- `< 0`: This means that the process was killed with a signal of *-1 * ExitCode*

For our example, the output value of the `ExitCode` code is equal to `-15`. The negative value `-15` indicates that the child was terminated by an interrupt signal identified by the number 15.

How to use a process in a subclass

To implement a custom subclass and process, we must:

- Define a new subclass of the `Process` class
- Override the `_init__(self [,args])` method to add additional arguments
- Override the `run(self [,args])` method to implement what `Process` should when it is started

Once you have created the new `Process` subclass, you can create an instance of it and then start by invoking the `start()` method, which will in turn call the `run()` method.