Chapter 4: Distributed and Parallel Computing

Contents

4.1 Introduction

So far, we have focused on how to create, interpret, and execute programs. In Chapter 1, we learned to use functions as a means for combination and abstraction. Chapter 2 showed us how to represent data and manipulate it with data structures and objects, and introduced us to the concept of data abstraction. In Chapter 3, we learned how computer programs are interpreted and executed. The result is that we understand how to design programs for a single processor to run.

In this chapter, we turn to the problem of coordinating multiple computers and processors. First, we will look at distributed systems. These are interconnected groups of independent computers that need to communicate with each other to get a job done. They may need to coordinate to provide a service, share data, or even store data sets that are too large to fit on a single machine. We will look at different roles computers can play in distributed systems and learn about the kinds of information that computers need to exchange in order to work together.

Next, we will consider concurrent computation, also known as parallel computation. Concurrent computation is when a single program is executed by multiple processors with a shared memory, all working together in parallel in order to get work done faster. Concurrency introduces new challenges, and so we will develop new techniques to manage the complexity of concurrent programs.

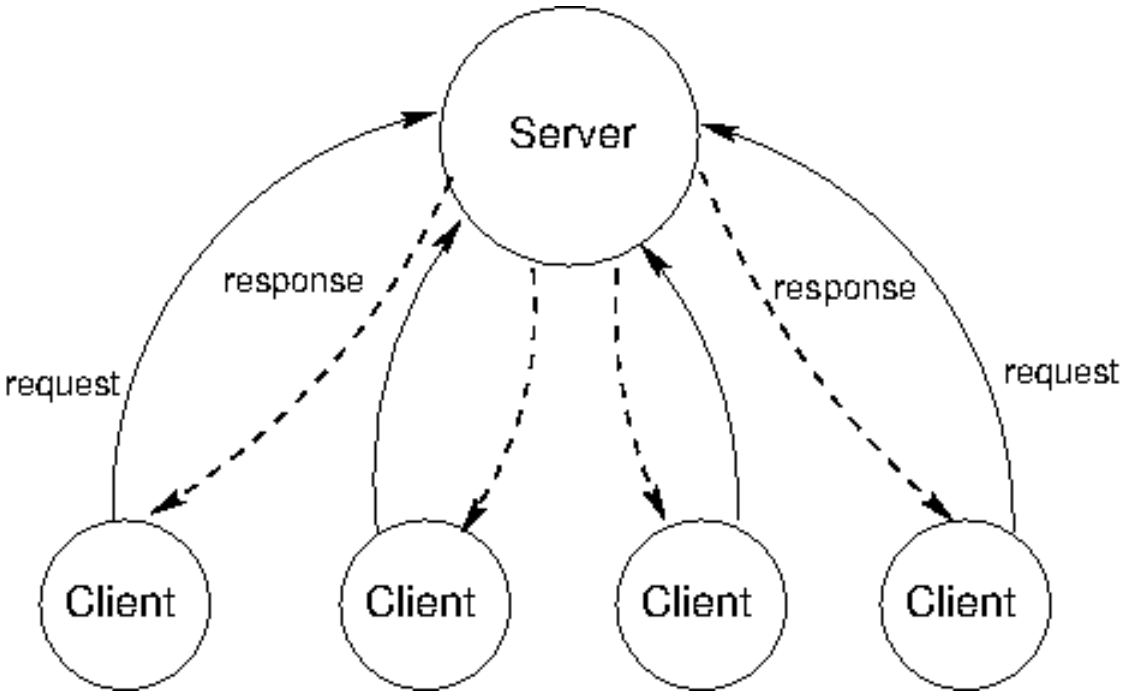
4.2 Distributed Computing

A distributed system is a network of autonomous computers that communicate with each other in order to achieve a goal. The computers in a distributed system are independent and do not physically share memory or processors. They communicate with each other using *messages*, pieces of information transferred from one computer to another over a network. Messages can communicate many things: computers can tell other computers to execute a procedures with particular arguments, they can send and receive packets of data, or send signals that tell other computers to behave a certain way.

Computers in a distributed system can have different roles. A computer's role depends on the goal of the system and the computer's own hardware and software properties. There are two predominant ways of organizing computers in a distributed system. The first is the client-server architecture, and the second is the peer-to-peer architecture.

4.2.1 Client/Server Systems

The client-server architecture is a way to dispense a service from a central source. There is a single *server* that provides a service, and multiple *clients* that communicate with the server to consume its products. In this architecture, clients and servers have different jobs. The server's job is to respond to service requests from clients, while a client's job is to use the data provided in response in order to perform some task.



The client-server model of communication can be traced back to the introduction of UNIX in the 1970's, but perhaps the most influential use of the model is the modern World Wide Web. An example of a client-server interaction is reading the New York Times online. When the web server at www.nytimes.com is contacted by a web browsing client (like Firefox), its job is to send back the HTML of the New York Times main page. This could involve calculating personalized content based on user account information sent by the client, and fetching appropriate advertisements. The job of the web browsing client is to render the HTML code sent by the server. This means displaying the images, arranging the content visually, showing different colors, fonts, and shapes and allowing users to interact with the rendered web page.

The concepts of *client* and *server* are powerful functional abstractions. A server is simply a unit that provides a service, possibly to multiple clients simultaneously, and a client is a unit that consumes the service. The clients do not need to know the details of how the service is provided, or how the data they are receiving is stored or calculated, and the server does not need to know how the data is going to be used.

On the web, we think of clients and servers as being on different machines, but even systems on a single machine can have client/server architectures. For example, signals from input devices on a computer need to be generally available to programs running on the computer. The programs are clients, consuming mouse and keyboard input data. The operating system's device drivers are the servers, taking in physical signals and serving them up as usable input.

A drawback of client-server systems is that the server is a single point of failure. It is the only component with the ability to dispense the service. There can be any number of clients, which are interchangeable and can come and go as necessary. If the server goes down, however, the system stops working. Thus, the functional abstraction created by the client-server architecture also makes it vulnerable to failure.

Another drawback of client-server systems is that resources become scarce if there are too many clients. Clients increase the demand on the system without contributing any computing resources. Client-server systems cannot shrink and grow with changing demand.

4.2.2 Peer-to-peer Systems

The client-server model is appropriate for service-oriented situations. However, there are other computational goals for which a more equal division of labor is a better choice. The term *peer-to-peer* is used to describe distributed systems in which labor is divided among all the components of the system. All the computers send and receive data, and they all contribute some processing power and memory. As a distributed system increases in size, its capacity of computational resources increases. In a peer-to-peer system, all components of the system contribute some processing power and memory to a distributed computation.

Division of labor among *all* participants is the identifying characteristic of a peer-to-peer system. This means that peers need to be able to communicate with each other reliably. In order to make sure that messages reach their intended destinations, peer-to-peer systems need to have an organized network structure. The components in these systems cooperate to maintain enough information about the locations of other components to send messages to intended destinations.

In some peer-to-peer systems, the job of maintaining the health of the network is taken on by a set of specialized components. Such systems are not pure peer-to-peer systems, because they have different types of components that serve different functions. The components that support a peer-to-peer network act like scaffolding: they help the network stay connected, they maintain information about the locations of different computers, and they help newcomers take their place within their neighborhood.

The most common applications of peer-to-peer systems are data transfer and data storage. For data transfer, each computer in the system contributes to send data over the network. If the destination computer is in a particular computer's neighborhood, that computer helps send data along. For data storage, the data set may be too large to fit on any single computer, or too valuable to store on just a single computer. Each computer stores a small portion of the data, and there may be multiple copies of the same data spread over different computers. When a computer fails, the data that was on it can be restored from other copies and put back when a replacement arrives.

Skype, the voice- and video-chat service, is an example of a data transfer application with a peer-to-peer architecture. When two people on different computers are having a Skype conversation, their communications are broken up into packets of 1s and 0s and transmitted through a peer-to-peer network. This network is composed of other people whose computers are signed into Skype. Each computer knows the location of a few other computers in its neighborhood. A computer helps send a packet to its destination by passing it on a neighbor, which passes it on to some other neighbor, and so on, until the packet reaches its intended destination. Skype is not a pure peer-to-peer system. A scaffolding network of *supernodes* is responsible for logging-in and logging-out users, maintaining information about the locations of their computers, and modifying the network structure to deal with users entering and leaving.

4.2.3 Modularity

The two architectures we have just considered -- peer-to-peer and client-server -- are designed to enforce *modularity*. Modularity is the idea that the components of a system should be black boxes with respect to each other. It should not matter how a component implements its behavior, as long as it upholds an *interface*: a specification for what outputs will result from inputs.

In chapter 2, we encountered interfaces in the context of dispatch functions and object-oriented programming. There, interfaces took the form of specifying the messages that objects should take, and how they should behave in response to them. For example, in order to uphold the "representable as strings" interface, an object must be able to respond to the `__repr__` and `__str__` messages, and output appropriate strings in response. How the generation of those strings is implemented is not part of the interface.

In distributed systems, we must consider program design that involves multiple computers, and so we extend this notion of an interface from objects and messages to full programs. An interface specifies the inputs that should be accepted and the outputs that should be returned in response to inputs. Interfaces are everywhere in the real world, and we often take them for granted. A familiar example is TV remotes. You can buy many different brands of remote for a modern TV, and they will all work. The only commonality between them is the "TV remote" interface. A piece of electronics obeys the "TV remote" interface as long as it sends the correct signals to your TV (the output) in response to when you press the power, volume, channel, or whatever other buttons (the input).

Modularity gives a system many advantages, and is a property of thoughtful system design. First, a modular system is easy to understand. This makes it easier to change and expand. Second, if something goes wrong with the system, only the defective components need to be replaced. Third, bugs or malfunctions are easy to localize. If the output of a component doesn't match the specifications of its interface, even though the inputs are correct, then that component is the source of the malfunction.

4.2.4 Message Passing

In distributed systems, components communicate with each other using message passing. A message has three essential parts: the sender, the recipient, and the content. The sender needs to be specified so that the recipient knows which component sent the message, and where to send replies. The recipient needs to be specified so that any computers who are helping send the message know where to direct it. The content of the message is the most variable. Depending on the function of the overall system, the content can be a piece of data, a signal, or instructions for the remote computer to evaluate a function with some arguments.

This notion of message passing is closely related to the message passing technique from Chapter 2, in which dispatch functions or dictionaries responded to string-valued messages. Within a program, the sender and receiver are identified by the rules of evaluation. In a distributed system however, the sender and receiver must be explicitly encoded in the message. Within a program, it is convenient to use strings to control the behavior of the dispatch function. In a distributed system, messages may need to be sent over a network, and may need to hold many different kinds of signals as 'data', so they are not always encoded as strings. In both cases, however, messages serve the same function. Different components (dispatch functions or computers) exchange them in order to achieve a goal that requires coordinating multiple modular components.

At a high level, message contents can be complex data structures, but at a low level, messages are simply streams of 1s and 0s sent over a network. In order to be usable, all messages sent over a network must be formatted according to a consistent *message protocol*.

A **message protocol** is a set of rules for encoding and decoding messages. Many message protocols specify that a message conform to a particular format, in which certain bits have a consistent meaning. A fixed format implies fixed encoding and decoding rules to generate and read that format. All the components in the distributed system must understand the protocol in order to communicate with each other. That way, they know which part of the message corresponds to which information.

Message protocols are not particular programs or software libraries. Instead, they are rules that can be applied by a variety of programs, even written in different programming languages. As a result, computers with vastly different software systems can participate in the same distributed system, simply by conforming to the message protocols that govern the system.

4.2.5 Messages on the World Wide Web

HTTP (short for Hypertext Transfer Protocol) is the message protocol that supports the world wide web. It specifies the format of messages exchanged between a web browser and a web server. All web browsers use the HTTP format to request pages from a web server, and all web servers use the HTTP format to send back their responses.

When you type in a URL into your web browser, say http://en.wikipedia.org/wiki/UC_Berkeley , you are in fact telling your browser that it must request the page "wiki/UC_Berkeley" from the server called "en.wikipedia.org" using the "http" protocol. The sender of the message is your computer, the recipient is en.wikipedia.org, and the format of the message content is:

```
GET /wiki/UC_Berkeley HTTP/1.1
```

The first word is the type of the request, the next word is the resource that is requested, and after that is the name of the protocol (HTTP) and the version (1.1). (There are another types of requests, such as PUT, POST, and HEAD, that web browsers can also use).

The server sends back a reply. This time, the sender is en.wikipedia.org, the recipient is your computer, and the format of the message content is a header, followed by data:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2011 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
```

```
Last-Modified: Wed, 08 Jan 2011 23:11:55 GMT
Content-Type: text/html; charset=UTF-8

... web page content ...
```

On the first line, the words "200 OK" mean that there were no errors. The subsequent lines of the header give information about the server, the date, and the type of content being sent back. The header is separated from the actual content of the web page by a blank line.

If you have typed in a wrong web address, or clicked on a broken link, you may have seen a message like this error:

```
404 Error File Not Found
```

It means that the server sent back an HTTP header that started like this:

```
HTTP/1.1 404 Not Found
```

A fixed set of response codes is a common feature of a message protocol. Designers of protocols attempt to anticipate common messages that will be sent via the protocol and assign fixed codes to reduce transmission size and establish a common message semantics. In the HTTP protocol, the 200 response code indicates success, while 404 indicates an error that a resource was not found. A variety of other response codes exist in the HTTP 1.1 standard as well.

HTTP is a fixed format for communication, but it allows arbitrary web pages to be transmitted. Other protocols like this on the internet are XMPP, a popular protocol for instant messages, and FTP, a protocol for downloading and uploading files between client and server.

4.3 Parallel Computing

Computers get faster and faster every year. In 1965, Intel co-founder Gordon Moore made a prediction about how much faster computers would get with time. Based on only five data points, he extrapolated that the number of transistors that could inexpensively be fit onto a chip would double every two years. Almost 50 years later, his prediction, now called Moore's law, remains startlingly accurate.

Despite this explosion in speed, computers aren't able to keep up with the scale of data becoming available. By some estimates, advances in gene sequencing technology will make gene-sequence data available more quickly than processors are getting faster. In other words, for genetic data, computers are become less and less able to cope with the scale of processing problems each year, even though the computers themselves are getting faster.

To circumvent physical and mechanical constraints on individual processor speed, manufacturers are turning to another solution: multiple processors. If two, or three, or more processors are available, then many programs can be executed more quickly. While one processor is doing one aspect of some computation, others can work on another. All of them can share the same data, but the work will proceed in parallel.

In order to be able to work together, multiple processors need to be able to share information with each other. This is accomplished using a shared-memory environment. The variables, objects, and data structures in that environment are accessible to all the processes.The role of a processor in computation is to carry out the evaluation and execution rules of a programming language. In a shared memory model, different processes may execute different statements, but any statement can affect the shared environment.

4.3.1 The Problem with Shared State

Sharing state between multiple processes creates problems that a single-process environments do not have. To understand why, let us look the following simple calculation:

```
x = 5
x = square(x)
x = x + 1
```

The value of x is time-dependent. At first, it is 5, then, some time later, it is 25, and then finally, it is 26. In a single-process environment, this time-dependence is is not a problem. The value of `x` at the end is always 26. The same cannot be said, however, if multiple processes exist. Suppose we executed the last 2 lines of above code in

parallel: one processor executes `x = square(x)` and the other executes `x = x+1`. Each of these assignment statements involves looking up the value currently bound to `x`, then updating that binding with a new value. Let us assume that since `x` is shared, only a single process will read or write it at a time. Even so, the order of the reads and writes may vary. For instance, the example below shows a series of steps for each of two processes, `P1` and `P2`. Each step is a part of the evaluation process described briefly, and time proceeds from top to bottom:

```
P1                      P2
read x: 5
                        read x: 5
calculate 5*5: 25       calculate 5+1: 6
write 25 -> x
                        write x-> 6
```

In this order, the final value of `x` is 6. If we do not coordinate the two processes, we could have another order with a different result:

```
P1                      P2
                        read x: 5
read x: 5               calculate 5+1: 6
calculate 5*5: 25       write x->6
write 25 -> x
```

In this ordering, `x` would be 25. In fact, there are multiple possibilities depending on the order in which the processes execute their lines. The final value of `x` could end up being 5, 25, or the intended value, 26.

The preceding example is trivial. `square(x)` and `x = x + 1` are simple calculations that are fast. We don't lose much time by forcing one to go after the other. But what about situations in which parallelization is essential? An example of such a situation is banking. At any given time, there may be thousands of people wanting to make transactions with their bank accounts: they may want to swipe their cards at shops, deposit checks, transfer money, or pay bills. Even a single account may have multiple transactions active at the same time.

Let us look at how the `make_withdraw` function from Chapter 2, modified below to print the balance after updating it rather than return it. We are interested in how this function will perform in a concurrent situation.

```
>>> def make_withdraw(balance):
        def withdraw(amount):
            nonlocal balance
            if amount > balance:
                print('Insufficient funds')
            else:
                balance = balance - amount
                print(balance)
        return withdraw
```

Now imagine that we create an account with $10 in it. Let us think about what happens if we withdraw too much money from the account. If we do these transactions in order, we receive an insufficient funds message.

```
>>> w = make_withdraw(10)
>>> w(8)
2
>>> w(7)
'Insufficient funds'
```

In parallel, however, there can be many different outcomes. One possibility appears below:

```
P1: w(8)                        P2: w(7)
read balance: 10
read amount: 8                  read balance: 10
8 > 10: False                   read amount: 7
if False                        7 > 10: False
10 - 8: 2                       if False
```

```
write balance -> 2              10 - 7: 3
read balance: 2                 write balance -> 3
print 2                         read balance: 3
                                print 3
```

This particular example gives an incorrect outcome of 3. It is as if the `w(8)` transaction never happened! Other possible outcomes are 2, and `'Insufficient funds'`. The source of the problems are the following: if `P2` reads `balance` before `P1` has written to `balance` (or vice versa), `P2`'s state is *inconsistent*. The value of `balance` that `P2` has read is obsolete, and `P1` is going to change it. `P2` doesn't know that and will overwrite it with an inconsistent value.

These example shows that parallelizing code is not as easy as dividing up the lines between multiple processors and having them be executed. The order in which variables are read and written matters.

A tempting way to enforce correctness is to stipulate that no two programs that modify shared data can run at the same time. For banking, unfortunately, this would mean that only one transaction could proceed at a time, since all transactions modify shared data. Intuitively, we understand that there should be no problem allowing 2 different people to perform transactions on completely separate accounts simultaneously. Somehow, those two operations do not interfere with each other the same way that simultaneous operations on the same account interfere with each other. Moreover, there is no harm in letting processes run concurrently when they are not reading or writing.

4.3.2 Correctness in Parallel Computation

There are two criteria for correctness in parallel computation environments. The first is that the outcome should always be the same. The second is that the outcome should be the same as if the code was executed in serial.

The first condition says that we must avoid the variability shown in the previous section, in which interleaving the reads and writes in different ways produces different results. In the example in which we withdrew `w(8)` and `w(7)` from a $10 account, this condition says that we must always return the same answer independent of the order in which `P1`'s and `P2`'s instructions are executed. Somehow, we must write our programs in such a way that, no matter how they are interleaved with each other, they should always produce the same result.

The second condition pins down which of the many possible outcomes is correct. In the example in which we evaluated `w(7)` and `w(8)` from a $10 account, this condition says that the result must always come out to be `Insufficient funds`, and not 2 or 3.

Problems arise in parallel computation when one process influences another during **critical sections** of a program. These are sections of code that need to be executed as if they were a single instruction, but are actually made up of smaller statements. A program's execution is conducted as a series of **atomic** hardware instructions, which are instructions that cannot be broken in to smaller units or interrupted because of the design of the processor. In order to behave correctly in concurrent situations, the critical sections in a programs code need to be have *atomicity* -- a guarantee that they will not be interrupted by any other code.

To enforce the atomicity of critical sections in a program's code under concurrency , there need to be ways to force processes to either *serialize* or *synchronize* with each other at important times. Serialization means that only one process runs at a time -- that they temporarily act as if they were being executed in serial. Synchronization takes two forms. The first is **mutual exclusion**, processes taking turns to access a variable, and the second is **conditional synchronization**, processes waiting until a condition is satisfied (such as other processes having finished their task) before continuing. This way, when one program is about to enter a critical section, the other processes can wait until it finishes, and then proceed safely.

4.3.3 Protecting Shared State: Locks and Semaphores

All the methods for synchronization and serialization that we will discuss in this section use the same underlying idea. They use variables in shared state as *signals* that all the processes understand and respect. This is the same philosophy that allows computers in a distributed system to work together -- they coordinate with each other by passing messages according to a protocol that every participant understands and respects.

These mechanisms are not physical barriers that come down to protect shared state. Instead they are based on mutual understanding. It is the same sort of mutual understanding that allows traffic going in multiple directions to safely use an intersection. There are no physical walls that stop cars from crashing into each other, just respect for rules that say red means "stop", and green means "go". Similarly, there is really nothing protecting those shared

variables except that the processes are programmed only to access them when a particular signal indicates that it is their turn.

Locks. Locks, also known as *mutexes* (short for mutual exclusions), are shared objects that are commonly used to signal that shared state is being read or modified. Different programming languages implement locks in different ways, but in Python, a process can try to acquire "ownership" of a lock using the `acquire()` method, and then `release()` it some time later when it is done using the shared variables. While a lock is acquired by a process, any other process that tries to perform the `acquire()` action will automatically be made to wait until the lock becomes free. This way, only one process can acquire a lock at a time.

For a lock to protect a particular set of variables, all the processes need to be programmed to follow a rule: no process will access any of the shared variables unless it owns that particular lock. In effect, all the processes need to "wrap" their manipulation of the shared variables in `acquire()` and `release()` statements for that lock.

We can apply this concept to the bank balance example. The critical section in that example was the set of operations starting when `balance` was read to when `balance` was written. We saw that problems occurred if more than one process was in this section at the same time. To protect the critical section, we will use a lock. We will call this lock `balance_lock` (although we could call it anything we liked). In order for the lock to actually protect the section, we must make sure to `acquire()` the lock before trying to entering the section, and `release()` the lock afterwards, so that others can have their turn.

```
>>> from threading import Lock
>>> def make_withdraw(balance):
        balance_lock = Lock()
        def withdraw(amount):
            nonlocal balance
            # try to acquire the lock
            balance_lock.acquire()
            # once successful, enter the critical section
            if amount > balance:
                print("Insufficient funds")
            else:
                balance = balance - amount
                print(balance)
            # upon exiting the critical section, release the lock
            balance_lock.release()
```

If we set up the same situation as before:

```
w = make_withdraw(10)
```

And now execute `w(8)` and `w(7)` in parallel:

```
P1                                  P2
acquire balance_lock: ok
read balance: 10                    acquire balance_lock: wait
read amount: 8                      wait
8 > 10: False                       wait
if False                            wait
10 - 8: 2                           wait
write balance -> 2                  wait
read balance: 2                     wait
print 2                             wait
release balance_lock                wait
                                    acquire balance_lock:ok
                                    read balance: 2
                                    read amount: 7
                                    7 > 2: True
                                    if True
                                    print 'Insufficient funds'
                                    release balance_lock
```

We see that it is impossible for two processes to be in the critical section at the same time. The instant one process acquires balance_lock, the other one has to wait until that processes *finishes* its critical section before it can even start.

Note that the program will not terminate unless `P1` releases `balance_lock`. If it does not release `balance_lock`, `P2` will never be able to acquire it and will be stuck waiting forever. Forgetting to release acquired locks is a common error in parallel programming.

Semaphores. Semaphores are signals used to protect access to limited resources. They are similar to locks, except that they can be acquired multiple times up to a limit. They are like elevators that can only carry a certain number of people. Once the limit has been reached, a process must wait to use the resource until another process releases the semaphore and it can acquire it.

For example, suppose there are many processes that need to read data from a central database server. The server may crash if too many processes access it at once, so it is a good idea to limit the number of connections. If the database can only support *N=2* connections at once, we can set up a semaphore with value *N=2*.

```
>>> from threading import Semaphore
>>> db_semaphore = Semaphore(2) # set up the semaphore
>>> database = []
>>> def insert(data):
        db_semaphore.acquire() # try to acquire the semaphore
        database.append(data)  # if successful, proceed
        db_semaphore.release() # release the semaphore

>>> insert(7)
>>> insert(8)
>>> insert(9)
```

The semaphore will work as intended if all the processes are programmed to only access the database if they can acquire the semaphore. Once *N=2* processes have acquired the semaphore, any other processes will wait until one of them has released the semaphore, and then try to acquire it before accessing the database:

```
P1                          P2                          P3
acquire db_semaphore: ok    acquire db_semaphore: wait  acquire db_semaphore: ok
read data: 7                wait                        read data: 9
append 7 to database        wait                        append 9 to database
release db_semaphore: ok    acquire db_semaphore: ok    release db_semaphore: ok
                            read data: 8
                            append 8 to database
                            release db_semaphore: ok
```

A semaphore with value 1 behaves like a lock.

4.3.4 Staying Synchronized: Condition variables

Condition variables are useful when a parallel computation is composed of a series of steps. A process can use a condition variable to signal it has finished its particular step. Then, the other processes that were waiting for the signal can start their work. An example of a computation that needs to proceed in steps a sequence of large-scale vector computations. In computational biology, web-scale computations, and image processing and graphics, it is common to have very large (million-element) vectors and matrices. Imagine the following computation:

$$A = B + C$$

$$V = MA$$

We may choose to parallelize each step by breaking up the matrices and vectors into range of rows, and assigning each range to a separate thread. As an example of the above computation, imagine the following simple values:

$$B = \begin{pmatrix} 2 \\ 0 \end{pmatrix} C = \begin{pmatrix} 0 \\ 5 \end{pmatrix} M = \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$$

We will assign first half (in this case the first row) to one thread, and the second half (second row) to another thread:

$$\begin{aligned} P1: A_1 &= B_1 + C_1 \\ P1: V_1 &= M_1 \cdot A \\ P2: A_2 &= B_2 + C_2 \\ P2: V_2 &= M_2 \cdot A \end{aligned}$$

In pseudocode, the computation is:

```
def do_step_1(index):
  A[index] = B[index] + C[index]

def do_step_2(index):
        V[index] = M[index] . A
```

Process 1 does:

```
do_step_1(1)
do_step_2(1)
```

And process 2 does:

```
do_step_1(2)
do_step_2(2)
```

If allowed to proceed without synchronization, the following inconsistencies could result:

```
P1                              P2
read B1: 2
read C1: 0
calculate 2+0: 2
write 2 -> A1                   read B2: 0
read M1: (1 2)                  read C2: 5
read A: (2 0)                   calculate 5+0: 5
calculate (1 2).(2 0): 2        write 5 -> A2
write 2 -> V1                   read M2: (1 2)
                                read A: (2 5)
                                calculate (1 2).(2 5):12
                                write 12 -> V2
```

The problem is that V should not be computed until all the elements of A have been computed. However, P1 finishes `A = B+C` and moves on to `V = MA` before all the elements of A have been computed. It therefore uses an inconsistent value of A when multiplying by M.

We can use a condition variable to solve this problem.

Condition variables are objects that act as signals that a condition has been satisfied. They are commonly used to coordinate processes that need to wait for something to happen before continuing. Processes that need the condition to be satisfied can make themselves wait on a condition variable until some other process modifies it to tell them to proceed.

In Python, any number of processes can signal that they are waiting for a condition using the `condition.wait()` method. After calling this method, they automatically wait until some other process calls the `condition.notify()` or `condition.notifyAll()` function. The `notify()` method wakes up just one process, and leaves the others waiting. The `notifyAll()` method wakes up all the waiting processes. Each of these is useful in different situations.

Since condition variables are usually associated with shared variables that determine whether or not the condition is true, they are offer `acquire()` and `release()` methods. These methods should be used when modifying variables that could change the status of the condition. Any process wishing to signal a change in the condition must first get access to it using `acquire()`.

In our example, the condition that must be met before advancing to the second step is that both processes must have finished the first step. We can keep track of the number of processes that have finished a step, and whether or not the condition has been met, by introducing the following 2 variables:

```
step1_finished = 0
start_step2 = Condition()
```

We will insert a `start_step_2().wait()` at the beginning of `do_step_2`. Each process will increment `step1_finished` when it finishes Step 1, but we will only signal the condition when `step_1_finished = 2`. The following pseudocode illustrates this:

```
step1_finished = 0
start_step2 = Condition()

def do_step_1(index):
  A[index] = B[index] + C[index]
  # access the shared state that determines the condition status
  start_step2.acquire()
  step1_finished += 1
  if(step1_finished == 2): # if the condition is met
        start_step2.notifyAll() # send the signal
  #release access to shared state
  start_step2.release()

def do_step_2(index):
  # wait for the condition
  start_step2.wait()
  V[index] = M[index] . A
```

With the introduction of this condition, both processes enter Step 2 together as follows::

```
P1                              P2
read B1: 2
read C1: 0
calculate 2+0: 2
write 2 -> A1                   read B2: 0
acquire start_step2: ok         read C2: 5
write 1 -> step1_finished       calculate 5+0: 5
step1_finished == 2: false      write 5-> A2
release start_step2: ok         acquire start_step2: ok
start_step2: wait               write 2-> step1_finished
```

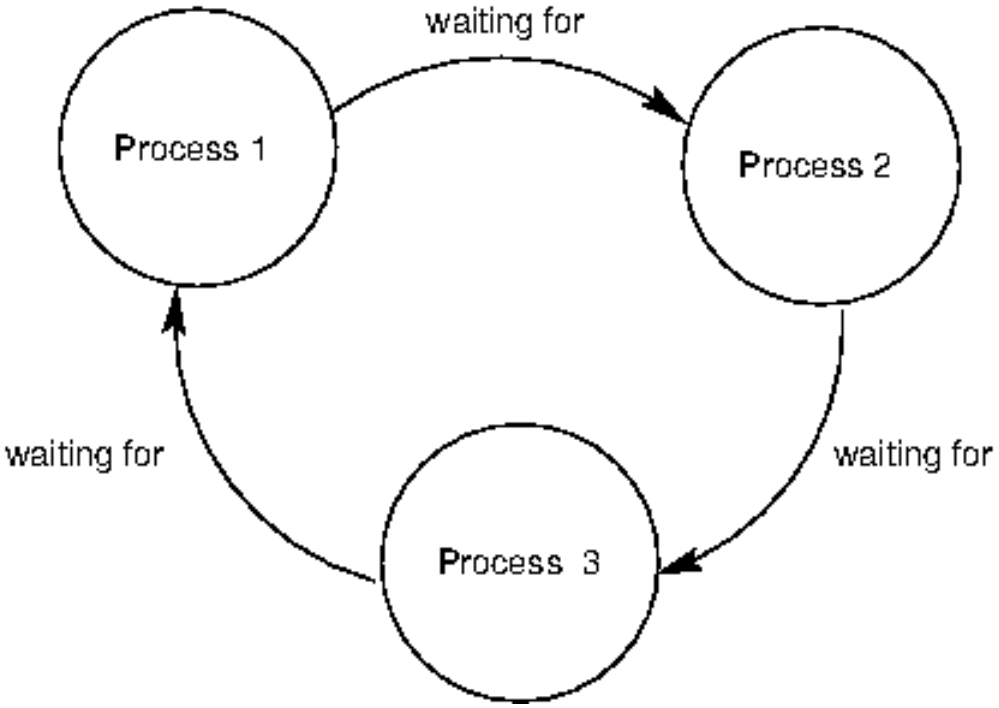
```
wait                              step1_finished == 2: true
wait                              notifyAll start_step_2: ok
start_step2: ok                   start_step2:ok
read M1: (1 2)                    read M2: (1 2)
read A:(2 5)
calculate (1 2). (2 5): 12        read A:(2 5)
write 12->V1                      calculate (1 2). (2 5): 12
                                  write 12->V2
```

Upon entering `do_step_2`, P1 has to wait on `start_step_2` until P2 increments `step1_finished`, finds that it equals 2, and signals the condition.

4.3.5 Deadlock

While synchronization methods are effective for protecting shared state, they come with a catch. Because they cause processes to wait on each other, they are vulnerable to **deadlock**, a situation in which two or more processes are stuck, waiting for each other to finish. We have already mentioned how forgetting to release a lock can cause a process to get stuck indefinitely. But even if there are the correct number of `acquire()` and `release()` calls, programs can still reach deadlock.

The source of deadlock is a **circular wait**, illustrated below. No process can continue because it is waiting for other processes that are waiting for it to complete.



As an example, we will set up a deadlock with two processes. Suppose there are two locks, `x_lock` and `y_lock`, and they are used as follows:

```
>>> x_lock = Lock()
>>> y_lock = Lock()
>>> x = 1
>>> y = 0
>>> def compute():
        x_lock.acquire()
        y_lock.acquire()
        y = x + y
```

```
        x = x * x
        y_lock.release()
        x_lock.release()

>>> def anti_compute():
        y_lock.acquire()
        x_lock.acquire()
        y = y - x
        x = sqrt(x)
        x_lock.release()
        y_lock.release()
```

If `compute()` and `anti_compute()` are executed in parallel, and happen to interleave with each other as follows:

```
P1                          P2
acquire x_lock: ok          acquire y_lock: ok
acquire y_lock: wait        acquire x_lock: wait
wait                        wait
wait                        wait
wait                        wait
...                         ...
```

the resulting situation is a deadlock. `P1` and `P2` are each holding on to one lock, but they need both in order to proceed. `P1` is waiting for `P2` to release `y_lock`, and `P2` is waiting for `P1` to release `x_lock`. As a result, neither can proceed.