# Lab Course: distributed data analytics

## 01. Threading and Parallelism

Nghia Duong-Trung, Mohsan Jameel

Information Systems and Machine Learning Lab (ISMLL)
University of Hildesheim, Germany

International Master's Program in Data Analytics
Summer Semester 2017

# Outline

1. Getting Started with Threading and Parallel Computing

2. Hands-On Multithreading Programming

3. Hands-On Multiprocessing Programming

# Outline

# Why Parallel Computing is Important ?

- A good tutorial on parallel computing is available at `https://computing.llnl.gov/tutorials/parallel_comp/`
- Get results in a short timespan.
  - process large amount of data in fraction of time,
  - run complex algorithm faster,
  - exploit underline hardware more efficiently.
- Parallel computing on multicore systems.
  - modern processor technology focus more on multi-cores rather than clock speed,
  - to get true performance, we need to write parallel program.
- What we need to do ?
  - Learn to develop parallel programs,
  - First step: write multi-threaded programs for multicore systems. i.e. learn driving mclaren 720S
  - Advance step: write GPU enable code i.e. learn driving F1 cars

# Parallel programming

- Parallel programming can be defined as an approach in which program data creates workers to run specific tasks simultaneously in a multicore environment without the need for concurrency amongst them to access a CPU [1].

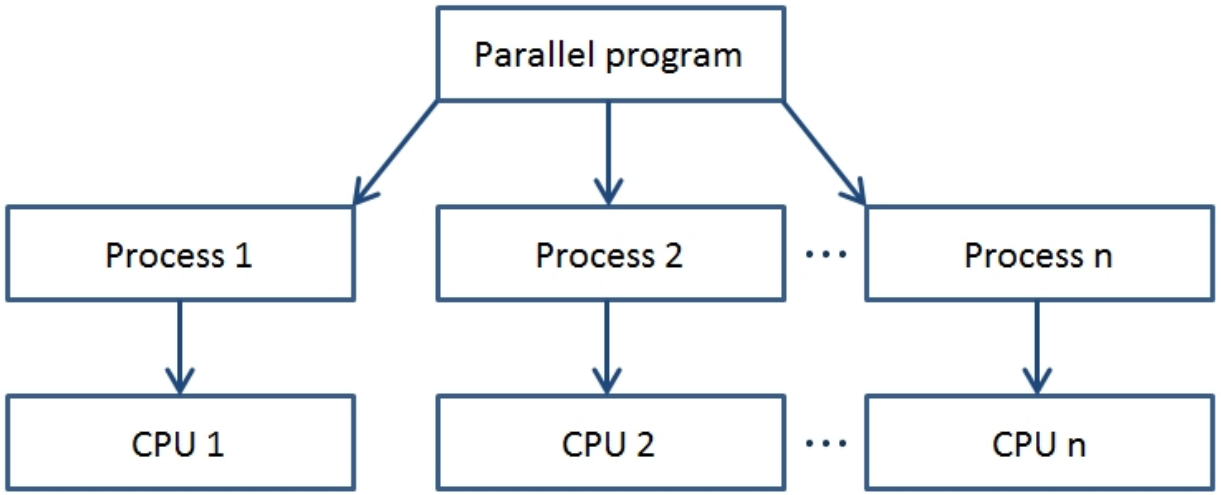


Figure: Parallel programming scheme

# Threads Basics

- Process: an instance of a running program such that
  - each process has its exclusive memory
  - managed by the OS
  - may have several threads
  - heavyweight
- Thread: a running subprogram such that
  - threads share memory allocated for their parent process
  - managed by the program which creates the threads
  - comprise a complete computing task that cannot be further sub-divided
  - lightweight
- Multiprocessing: execute several processes in parallel
- Multithreading: execute several threads in parallel

# Threads Basics

- Imagine the task to multiply a scalar with a matrix
- In a sequential system, one would perform each multiplication operation one after one and generate the final results at the end of all the instructions
- Apply the idea of parallelism for the same task:

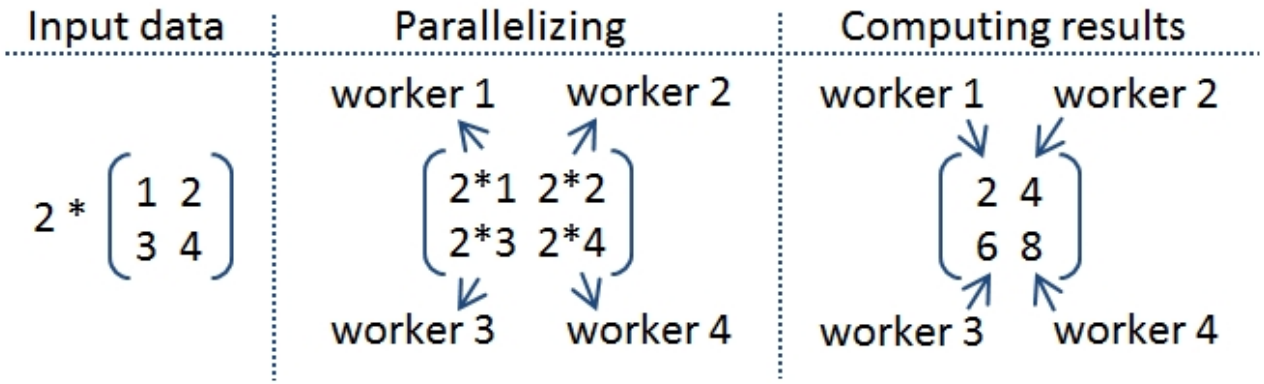


Figure: An example of parallelism in matrix multiplication

# States of a thread

- There are five possible states in a thread's life span:
  - Creation: This is the main process that creates a thread, and after its creation, it is sent to a line of threads ready for execution.
  - Ready: the thread is in a line of threads ready for execution and bounded to be executed.
  - Execution: the thread makes use of the CPU.
  - Blocked: the thread is blocked to wait for an I/O operation and it does not make use of the CPU.
  - Concluded: the thread's life span is end and resources are free from its execution.

# Parallel Programming Problems

- Deadlock
  - A situation in which two or more workers keep indefinitely waiting for the freeing of a required resource which is currently blocked by another worker of the same group.
- Starvation
  - A situation in which one or more processes run heavy tasks and has data processor priority while the others never get the chance.
- Race Conditions
  - A situation in which the computing result depends on a sequence of operations and this sequence is broken due to the lack of synchronization mechanism.

# Mechanism of Synchronization

- An example which contains a mechanism of synchronization

| Customer 1 | Customer 2 | A product (units) |
|---|---|---|
| | | 10 |
| How many left? | | 10 |
| Buys 5 | | 10 |
| Concludes operation | | 5 |
| | How many left? | 5 |
| | Buys 3 | 5 |
| | Concludes operation | 2 |

# Outline

# threading module

- Python manages a thread via the `threading` module that is provided by the Python standard library.
- The Python module `threading` has the `Thread()` method that is used to run functions in different threads:

  `class threading.Thread(group=None, target=None, name=None, args=(), kwargs={})`

  In the preceding constructor, arguments are:

  - `group` should be `None`; this is reserved for future implementations.
  - `target` is the callable object to be invoked by the `run()` method. Defaults to `None`, meaning nothing is called. `name` is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

    `args` is the argument tuple for the target invocation. Defaults to `()`.

    `kwargs` is a dictionary of keyword arguments for the target invocation. Defaults to {}.

# threading module

- To import the `threading` module, we simply use the Python command:

```
import threading
```

- In the main program, we instantiate a thread, pass an argument to the function:

```
t = threading.Thread(target=function , args=(i,))
```

- The thread does not start running until the `start()` method is called, and the `join()` method makes the calling thread wait until the thread has finished the execution.

```
t.start()
t.join()
```

# threading module

```
1  import threading
2
3  def function(i):
4    print ("function called by thread {}".format(i))
5    return
6
7  threads = []
8  for i in range(10):
9    t = threading.Thread(target=function, args=(i,))
10   threads.append(t)
11   t.start()
12   t.join()
13
```

Listing 1: threading example

# currentThread function

```
1  import threading
2  import time
3
4  def first_function():
5    print(threading.currentThread().getName() + " is starting \n")
6    time.sleep(2) #seconds
7    print(threading.currentThread().getName() + " is exiting \n")
8
9  def second_function():
10   print(threading.currentThread().getName() + " is starting \n")
11   time.sleep(4) #seconds
12   print(threading.currentThread().getName() + " is exiting \n")
13
14 def third_function():
15   print(threading.currentThread().getName() + " is starting \n")
16   time.sleep(2) #seconds
17   print(threading.currentThread().getName() + " is exiting \n")
18
19 if __name__ == "__main__":
20   t1 = threading.Thread(name='first_function', target=first_function)
21   t2 = threading.Thread(name='second_function', target=second_function)
22   t3 = threading.Thread(name='third_function',target=third_function)
23   t1.start()
24   t2.start()
25   t3.start()
26   t1.join()
27   t2.join()
28   t3.join()
29
```

Listing 2: currentThread function

# Why join()?

```
without join:
+---+---+-------------------                    main-thread
|   |
|   +.........                                 child-thread(short)
+.................................             child-thread(long)

with join
+---+---+----------------------**********+###    main-thread
|   |                                    |
|   +.........join()                     |      child-thread(short)
+....................join().....          |      child-thread(long)

with join and demon thread
+-+---+---+----------------------**********+###    parent-thread
|   |   |                                  |
|   |   +.........join()                   |        child-thread(short)
|   +....................join().....       |        child-thread(long)
+,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,      child-thread(long+demonized)

'-' main-thread/parent-thread/main-program execution
'.' child-thread execution
'#' optional parent-thread execution after join()-blocked parent-thread could continue
'*' main-thread 'sleeping' in join-method, waiting for child-thread to finish
',' demonized thread - 'ignores' lifetime of other threads; terminates when main-
    programs exits; is normally meant for join-independent tasks
```

Listing 3: A simple explanation from stackoverflow

# Inheritance of `Thread` class

- To implement a new `thread` that inherits the `Thread` class, one has to do:
  - Define a new class that inherits the `Thread` class.
  - Override the `__init__(self [,args])` and `run(self [,args])` to redefine some properties and instructions of the thread if necessary.
- Once one have created the new `Thread` subclass, one can create an instance of it and then start by invoking the `start()` method, which will in turn call the `run()` method.

# Inheritance of Thread class - continued

```
1  import threading, time
2
3  class myThread(threading.Thread):
4    def __init__(self, thread_name, counter):
5      self.thread_name = thread_name
6      self.counter = counter
7      threading.Thread.__init__(self)
8
9    def run(self):
10     display_time(self.name, self.counter)
11
12 def display_time(thread_name, counter):
13   while counter:
14     time.sleep(counter)
15     print("{}: {}".format(thread_name,time.ctime(time.time())))
16     counter -= 1
17
18
19 thread1 = myThread("thread-1",5)
20 thread2 = myThread("thread-2",3)
21
22 thread1.start()
23 thread2.start()
24 thread1.join()
25 thread2.join()
26
```

Listing 4: Inheritance of Thread class

# Thread synchronization

- Determinacy race

```
1  counter = 0 # globally defined
2  def process_item(item):
3      global counter
4      ... do something with item ...
5      counter += 1
6
```

- **Definition**: A determinacy race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write
- **Types of Races**: Read or Write
- Tutorial on Python threads synchronization: `http://www.laurentluce.com/posts/python-threads-synchronization-locks-rlocks-semaphores-co`

# Thread synchronization

- Atomic operations/ Thread-Safe Operations
- Lock and RLock
- Semaphore
- Condition
- Event
- `with` statement

# Atomic Operations

- reading or replacing a single instance attribute
- reading or replacing a single global variable
- fetching an item from a list
- modifying a list in place (e.g. adding an item using append)
- fetching an item from a dictionary
- modifying a dictionary in place (e.g. adding an item, or calling the clear method)

# Lock

- When a thread wants to access a portion of shared memory, it must necessarily acquire a lock on that portion prior to using it.
- After completing its operation, the thread must release the lock previously obtained.
- The portion of shared memory is available for any other threads.
- This approach may lead to a bad situation of deadlock.

# Without Lock

```
1  import threading, logging, time
2  logging.basicConfig(level=logging.INFO)
3
4  shared_resource_with_no_lock, COUNT = 0, 5
5
6  def increment_without_lock():
7    global shared_resource_with_no_lock
8    for i in range(COUNT):
9      time.sleep(2)
10     shared_resource_with_no_lock += 1
11     logging.info("resource is increasing ... " + str(shared_resource_with_no_lock))
12
13 def decrement_without_lock():
14   global shared_resource_with_no_lock
15   for i in range(COUNT):
16     time.sleep(2)
17     shared_resource_with_no_lock -= 1
18     logging.info("resource is decreasing ... " + str(shared_resource_with_no_lock))
19
20 if __name__ == "__main__":
21   t1 = threading.Thread(target = increment_without_lock)
22   t2 = threading.Thread(target = decrement_without_lock)
23
24   t1.start()
25   t2.start()
26   t1.join()
27   t2.join()
28
```

# Lock - continued

```
1  import threading, logging, time
2  logging.basicConfig(level=logging.INFO)
3  shared_resource_with_lock, COUNT = 0, 5
4  lock = threading.Lock()
5
6  def increment_with_lock():
7    global shared_resource_with_lock
8    for i in range(COUNT,0,-1):
9      lock.acquire()
10     time.sleep(2)
11     shared_resource_with_lock += 1
12     logging.info("resource is increasing ... " + str(shared_resource_with_lock))
13     lock.release()
14
15 def decrement_with_lock():
16   global shared_resource_with_lock
17   for i in range(COUNT,0,-1):
18     lock.acquire()
19     time.sleep(2)
20     shared_resource_with_lock -= 1
21     logging.info("resource is decreasing ... " + str(shared_resource_with_lock))
22     lock.release()
23
24 if __name__ == "__main__":
25   t1 = threading.Thread(target = increment_with_lock)
26   t2 = threading.Thread(target = decrement_with_lock)
27   t1.start()
28   t2.start()
29   t1.join()
30   t2.join()
31
```

# RLock

```
1  import threading, time
2
3  class Box(object):
4    def __init__(self):
5      self.total_items = 5
6
7    def execute(self,n):
8      lock.acquire()
9      self.total_items += n
10     lock.release()
11
12   def add(self):
13     lock.acquire()
14     self.execute(1)
15     lock.release()
16
17   def remove(self):
18     lock.acquire()
19     self.execute(-1)
20     lock.release()
21
22 def adder(box,items):
23   while items > 0:
24   lock.acquire()
25   box.add()
26   lock.release()
27   print("1 item added to the box ... \t \t " + str(box.total_items))
28   time.sleep(2)
29   items -= 1
30
```

# RLock - continued

```
1  def remover(box,items):
2    while items > 0 and box.total_items > 0:
3      lock.acquire()
4      box.remove()
5      lock.release()
6      print("1 item removed from the box ... \t " + str(box.total_items))
7      time.sleep(2)
8
9  if __name__ == "__main__":
10   lock = threading.Lock()
11   items = 5
12   box = Box()
13   t1 = threading.Thread(target=adder,args=(box,items))
14   t2 = threading.Thread(target=remover,args=(box,items))
15   t1.start()
16   t2.start()
17   t1.join()
18   t2.join()
19
```

- Try to replace

  `lock = threading.Lock()`

  by

  `lock = threading.RLock()`

# Semaphore

- A semaphore manages an internal counter which is decremented by each `acquire()` operation and incremented by each `release()` operation, as explained:
  - Whenever a thread wants to access a resource that is associated with a `semaphore`, it must invoke the `acquire()` operation, which decreases the internal counter of the `semaphore`. If the value is negative, the thread would be suspended.
  - Whenever a thread has finished using the resource, it must release the resource through the `release()` operation. the internal counter of the semaphore is incremented.
- There are many cases we may want to allow more than one worker access to a resource while still limiting the overall number of accesses.
  - One might use `semaphore` in a situation where we need to support concurrent connections/downloads.

# Condition

- A condition identifies a change of state in the application.
- A thread waits for a specific condition and another thread notifies that this condition has taken place.
- Once the condition takes place, the thread acquires the lock to get exclusive access to the shared resource.

# Condition - continued

```
1  from threading import Thread, Condition
2  import time
3
4  items = []
5  condition = Condition()
6
7  class consumer(Thread):
8    def __init__(self):
9      Thread.__init__(self)
10
11   def consume(self):
12     global condition
13     global items
14
15     condition.acquire()
16
17     if len(items) == 0:
18       print("Consumer notify : no item to consume")
19       condition.wait()
20
21     items.pop()
22     print("Consumer notify : consumed 1 item")
23     print("Consumer notify : items to consume are " + str(len(items)))
24     condition.notify()
25     condition.release()
26
27   def run(self):
28     for i in range(0,10):
29     time.sleep(5)
30     self.consume()
31
```

# Condition - continued

```
1  class producer(Thread):
2    def __init__(self):
3    Thread.__init__(self)
4
5    def produce(self):
6      global condition
7      global items
8
9      condition.acquire()
10     if len(items) == 5:
11       print("Producer notify : items producted are " + str(len(items)))
12       print("Producer notify : stop the production")
13       condition.wait()
14
15     items.append(1)
16     print("Producer notify : total items producted " + str(len(items)))
17     condition.notify()
18     condition.release()
19
20   def run(self):
21     for i in range(0,10):
22     time.sleep(2)
23     self.produce()
24
25 if __name__ == "__main__":
26   p = producer()
27   c = consumer()
28   p.start()
29   c.start()
30   p.join()
31   c.join()
32
```

# Event

- One thread signals an event and other threads wait for it.
- An event object manages an internal flag that can be set to true with the `set()` method and reset to false with the `clear()` method.
- The `wait()` method blocks until the flag is true.

# Event - continued

```
1  from threading import Thread, Event
2  import random, time
3
4  class consumer(Thread):
5    def __init__(self, items, event):
6      Thread.__init__(self)
7      self.items = items
8      self.event = event
9
10   def run(self):
11     self.event.wait()
12     print("Consumer notify : {} popped from list by {}".format(self.items.pop(), self
     .name))
13     self.event.clear()
14
15 class producer(Thread):
16   def __init__(self, integers, event):
17     Thread.__init__(self)
18     self.items = items
19     self.event = event
20
21   def run(self):
22     time.sleep(2)
23     global item
24     item = random.randint(0, 100)
25     self.items.append(item)
26     print("Producer notify : item {} appended to list by {}".format(item, self.name))
27     self.event.set()
28
```

# Event - continued

```
1  if __name__ == '__main__':
2      items = []
3      event = Event()
4      t1 = producer(items, event)
5      t2 = consumer(items, event)
6      t1.start()
7      t2.start()
8      t1.join()
9      t2.join()
10
```

# `with` statement

- `with` statement is called a context manager.
- All objects provided by the `acquire()` and `release()` methods may be used in a `with` statement.
- The following objects can be used within a `with` statement.
  - `Lock`
  - `RLock`
  - `Condition`
  - `Semaphore`

# with statement - continued

```
1  import threading
2
3  def threading_with(statement):
4    with statement:
5      print(str(statement) + "acquired via with")
6
7  def threading_not_with(statement):
8    statement.acquire()
9    try:
10     print(str(statement) + "acquired directly")
11   finally:
12     statement.release()
13
14 if __name__ == '__main__':
15   lock = threading.Lock()
16   rlock = threading.RLock()
17   condition = threading.Condition()
18   semaphore = threading.Semaphore()
19   threading_synchronization_list = [lock ,rlock , condition , semaphore]
20
21   for statement in threading_synchronization_list :
22     t1 = threading.Thread(target=threading_with , args=(statement ,))
23     t2 = threading.Thread(target=threading_not_with , args=(statement ,))
24     t1.start()
25     t2.start()
26     t1.join()
27     t2.join()
28
```

# Thread communication using a queue

- We implement a version of Producer and Consumer with queue.
  - The Producer thread is responsible for putting items into the queue.
  - Consumer thread consumes items if there are any in the queue.
- We simply consider these queue methods:
  - `put()`: this puts an item into the queue.
  - `get()`: this removes an returns an item from the queue.
  - `task_done()`: this needs to be called each time an item has been processed.

# Thread communication using a queue - continued

```
1  from threading import Thread
2  from queue import Queue
3  import time, random
4
5  class producer(Thread):
6    def __init__(self, queue):
7      Thread.__init__(self)
8      self.queue = queue
9      self.name= "Producer"
10
11   def run(self) :
12     for i in range(10):
13       item = random.randint(0, 100)
14       self.queue.put(item)
15       print("Producer notify : item {} appended to queue by {}".format(item, self.name)
       )
16       time.sleep(1)
17
18 class consumer(Thread):
19   def __init__(self, queue):
20     Thread.__init__(self)
21     self.queue = queue
22     self.name= "Consumer"
23
24   def run(self):
25     while True:
26       item = self.queue.get()
27       print("Consumer notify : {} popped from queue by {}".format(item, self.name))
28       self.queue.task_done()
29
```

# Thread communication using a queue - continued

```
1  if __name__ == '__main__':
2    queue = Queue()
3    t1 = producer(queue)
4    t2 = consumer(queue)
5    t3 = consumer(queue)
6    t4 = consumer(queue)
7    t1.start()
8    t2.start()
9    t3.start()
10   t4.start()
11   t1.join()
12   t2.join()
13   t3.join()
14   t4.join()
15
```

# Outline

# Create a process

- This section will examine the process-based approach using the standard library `multiprocessing`.
- `multiprocessing` implements the shared memory programming paradigm. That is the programming consists of one or more processors that have access to a common memory.

```
1   import multiprocessing
2
3   def function(i):
4     print("called function in process: {}".format(i))
5
6   if __name__ == '__main__':
7     Process_jobs = []
8     for i in range(5):
9       p = multiprocessing.Process(target=function, args=(i,))
10      Process_jobs.append(p)
11      p.start()
12      p.join()
13
```

# Name a process

- It is very useful to associate a name to the processes as debugging an application requires the processes to be well marked and identifiable.

```
1   import multiprocessing, time
2
3   def foo():
4     name = multiprocessing.current_process().name
5     print("Starting {}".format(name))
6     time.sleep(1)
7     print("Exiting {}".format(name))
8
9   if __name__ == '__main__':
10    process_with_name = multiprocessing.Process(name='foo_process', target=foo)
11    process_with_default_name = multiprocessing.Process(target=foo)
12    process_with_name.start()
13    process_with_default_name.start()
14    process_with_name.join()
15    process_with_default_name.join()
16
```

# Kill a process

- `terminate()` method kills a process immediately.
- `is_alive()` method keeps track of whether a process is alive or not.

```
1  import multiprocessing, time
2
3  def foo():
4    print ('Starting function')
5    time.sleep(1)
6    print ('Finished function')
7
8  if __name__ == '__main__':
9    p = multiprocessing.Process(target=foo)
10   print ('Process before execution:', p, p.is_alive())
11
12   p.start()
13   print ('Process running:', p, p.is_alive())
14
15   p.terminate()
16   print ('Process terminated:', p, p.is_alive())
17
18   p.join()
19   print ('Process joined:', p, p.is_alive())
20
```

# Inheritance of `Process` class

- To implement a new `process` that inherits the `Process` class, one has to do:
  - Define a new class that inherits the `Process` class.
  - Override the `__init__(self [,args])` and `run(self [,args])` to redefine some properties and instructions of the process if necessary.
- Once one have created the new `Process` subclass, one can create an instance of it and then start by invoking the `start()` method, which will in turn call the `run()` method.

# Inheritance of Process class - continued

```
1  import multiprocessing, time
2
3  class myProcess(multiprocessing.Process):
4    def __init__(self, process_name, counter):
5      self.process_name = process_name
6      self.counter = counter
7      multiprocessing.Process.__init__(self)
8
9    def run(self):
10     display_time(self.name, self.counter)
11
12 def display_time(process_name, counter):
13   while counter:
14     time.sleep(counter)
15     print("{}: {}".format(process_name,time.ctime(time.time())))
16     counter -= 1
17
18 if __name__ == '__main__':
19   process1 = myProcess("process-1",5)
20   process2 = myProcess("process-2",3)
21
22   process1.start()
23   process2.start()
24   process1.join()
25   process2.join()
26
```

Listing 5: Inheritance of Process class

# Process communication using a queue

- We implement a version of Producer and Consumer with queue.
  - The Producer process is responsible for putting items into the queue.
  - Consumer process consumes items if there are any in the queue.
- We simply consider these queue methods:
  - `put()`: this puts an item into the queue.
  - `get()`: this removes an returns an item from the queue.
  - `task_done()`: this needs to be called each time an item has been processed.

# Process communication using a queue - continued

```
1  from multiprocessing import Process, Queue
2  import time, random
3
4  class producer(Process):
5    def __init__(self, queue):
6      Process.__init__(self)
7      self.queue = queue
8      self.name= "Producer"
9
10   def run(self) :
11     for i in range(5):
12       item = random.randint(0, 100)
13       self.queue.put(item)
14       print("Producer notify : item {} appended to queue by {}".format(item, self.name)
       )
15       print("The size of queue is {}".format(self.queue.qsize()))
16       time.sleep(1)
17
18 class consumer(Process):
19   def __init__(self, queue):
20     Process.__init__(self)
21     self.queue = queue
22     self.name= "Consumer"
23
```

# Process communication using a queue - continued

```
      def run(self):
      while True:
        if (self.queue.empty()):
          print("The queue is empty")
          break
        else:
          time.sleep(1)
          item = self.queue.get()
          print("Consumer notify: item {} popped from by {}".format(item, self.name))
          print("The size of queue is {}".format(self.queue.qsize()))
          time.sleep(1)

if __name__ == '__main__':
  queue = Queue()
  t1 = producer(queue)
  t2 = consumer(queue)
  t3 = consumer(queue)
  t4 = consumer(queue)
  t1.start()
  t2.start()
  t3.start()
  t4.start()
  t1.join()
  t2.join()
  t3.join()
  t4.join()

```

# Process synchronization

- Timer
- Lock and RLock
- Semaphore
- Condition
- Event
- `with` statement
- Barrier

# Barrier

- `Barrier` divides a program into phases in which it requires all of the processes to reach a barrier before any of the processes proceeds.
- Code that is executed after a barrier cannot be concurrent with the code executed before the barrier.

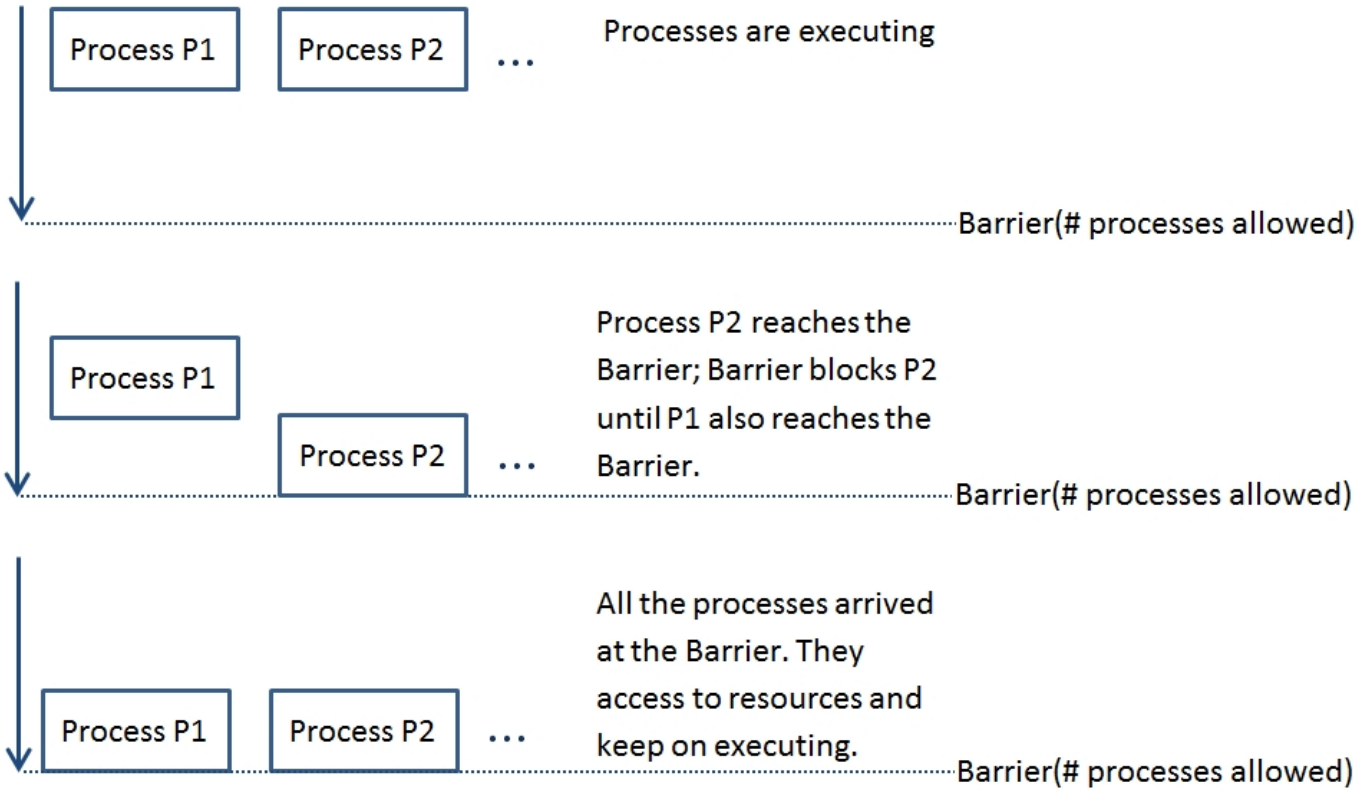


Figure: Process management with a `Barrier`.

# Barrier - continued

```
1  import multiprocessing, time, datetime
2  from multiprocessing import Barrier, Lock, Process
3
4  def with_barrier(synchronizer, lock):
5    name = multiprocessing.current_process().name
6    synchronizer.wait()
7    now = time.time()
8    with lock:
9      print("process {} ---> {}".format(name,datetime.datetime.fromtimestamp(now)))
10
11 def without_barrier():
12   name = multiprocessing.current_process().name
13   now = time.time()
14   print("process {} ---> {}".format(name ,datetime.datetime.fromtimestamp(now)))
15
16 if __name__ == '__main__':
17   synchronizer = Barrier(3)
18   lock = Lock()
19   Process(name="p1 - with_barrier", target=with_barrier, args=(synchronizer,lock)).
       start()
20   Process(name="p2 - with_barrier", target=with_barrier, args=(synchronizer,lock)).
       start()
21   Process(name="p3 - with_barrier", target=with_barrier, args=(synchronizer,lock)).
       start()
22   Process(name="p4 - without_barrier", target=without_barrier).start()
23   Process(name="p5 - without_barrier", target=without_barrier).start()
24   Process(name="p6 - without_barrier", target=without_barrier).start()
25
```

# Parallel processes with `Pool` class

- The multiprocessing library provides the `Pool` class for simple parallel processing tasks.

```
1   import multiprocessing
2
3   def function_square(data):
4     return data*data
5
6   if __name__ == '__main__':
7     inputs = list(range(0,100))
8     pool = multiprocessing.Pool(processes=4)
9     pool_outputs = pool.map(function_square, inputs)
10    pool.close()
11    pool.join()
12    print(pool_outputs)
13
```

- It is important to note that the result of the `pool.map()` method is equivalent to Python's built-in function `map()`, except that the processes run parallelly.

# Further Reading


1. Palach, Jan. Parallel Programming with Python. Packt Publishing Ltd, 2014.
2. Threads in Python `http://www.python-course.eu/threads.php`
3. The Python threading module
   `https://docs.python.org/3/library/threading.html?highlight=threading#module-threading`
4. The Python multi-processing module
   `https://docs.python.org/3/library/multiprocessing.html?highlight=multiprocessing#multiprocessing`
5. The parallel Python module `http://www.parallelpython.com/`
6. Celery: Distributed Task Queue `http://www.celeryproject.org/`
7. http://www.bogotobogo.com/python/pytut.php
8. Lutz, M. (2013). Learning python. " O'Reilly Media, Inc.".
9. Beazley, D., & Jones, B. K. (2013). Python cookbook. " O'Reilly Media, Inc.".
10. Zaccone, G. (2015). Python Parallel Programming Cookbook. Packt Publishing Ltd.
11. Lanaro, G. (2013). Python High Performance Programming. Packt Publishing Ltd.