multiprocessing and multithreading

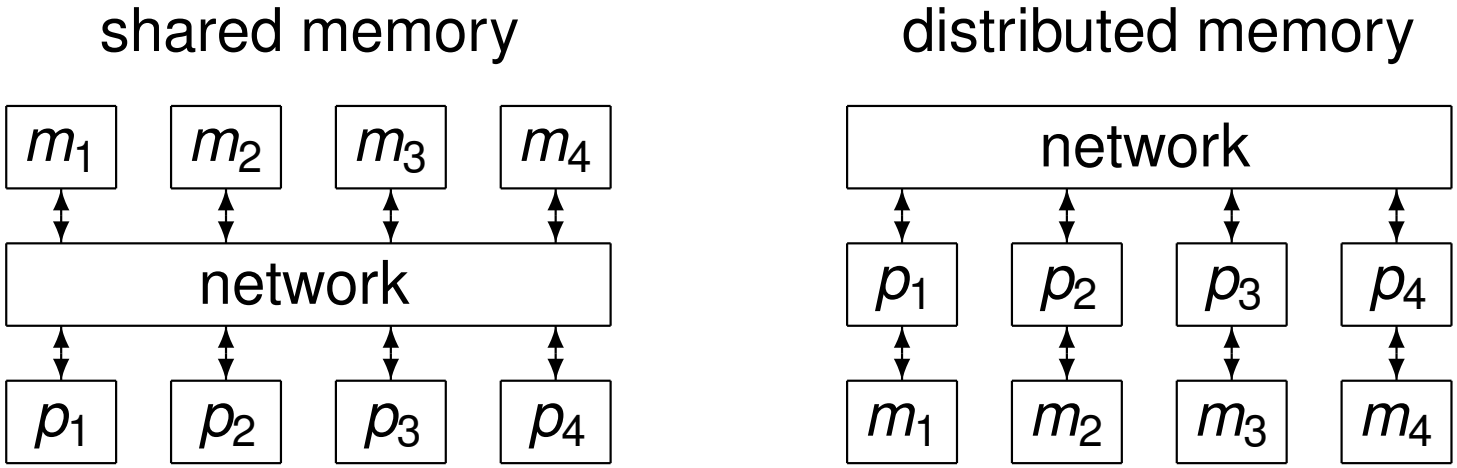
1. Types of Parallel Computing
   - shared and distributed memory multicomputers
2. Multiprocessing in Python
   - pleasingly parallel computation
   - the multiprocessing module
   - numerical integration with multiple processes
3. Multithreading in Python
   - processes and threads
   - life cycle of a thread
   - the Thread class
   - modeling a producer/consumer relation

MCS 507 Lecture 12
Mathematical, Statistical and Scientific Software
Jan Verschelde, 18 September 2023

multiprocessing and multithreading

1. Types of Parallel Computing
   - shared and distributed memory multicomputers
2. Multiprocessing in Python
   - pleasingly parallel computation
   - the multiprocessing module
   - numerical integration with multiple processes
3. Multithreading in Python
   - processes and threads
   - life cycle of a thread
   - the Thread class
   - modeling a producer/consumer relation

types of parallel computers



One crude distinction concerns memory, shared or distributed:

- A shared memory multicomputer has one single address space, accessible to every processor.
- In a distributed memory multicomputer, every processor has its own memory accessible via messages through that processor.

On distributed memory computers, we apply multiprocessing, on shared memory computers, we apply multithreading.

multiprocessing and multithreading

1. Types of Parallel Computing
   - shared and distributed memory multicomputers
2. Multiprocessing in Python
   - pleasingly parallel computation
   - the multiprocessing module
   - numerical integration with multiple processes
3. Multithreading in Python
   - processes and threads
   - life cycle of a thread
   - the Thread class
   - modeling a producer/consumer relation

pleasingly parallel computation

A computation is *pleasingly parallel* if there is no communication overhead between the processes and an optimal speedup is expected.

Our example: $\int_0^1 \sqrt{1-x^2}dx = \frac{\pi}{4}$.

We will use the Simpson rule (available in `scipy`) as a relatively computational intensive example.

The Simpson rule is available as `simps` in the `integrate` module of the `scipy` package.

interactive computing

We test `simps` on $\int_0^1 \sqrt{1-x^2}dx$, using 1,000 evaluations:

```
$ python
>>> from scipy.integrate import simps
>>> from scipy import sqrt, linspace
>>> x = linspace(0,1,1000)
>>> y = sqrt(1-x**2)
>>> I = simps(y,x)
>>> 4*I
3.1415703366671104
>>> from scipy import pi
>>> pi
3.141592653589793
```

the script `simpson4pi.py`

```
from scipy.integrate import simps
from scipy import sqrt, linspace, pi
print('10**k       approximation          error')
for k in range(2, 9):
    x = linspace(0, 1, 10**k)
    y = sqrt(1-x**2)
    I = 4*simps(y, x)
    print('10**%d: %.15e, %.3e' % (k, I, abs(I-pi)))
```

running the script `simpson4pi.py`

```
$ time python simpson4pi.py
10**k       approximation          error
10**2: 3.140876361334483e+00, 7.163e-04
10**3: 3.141570336667110e+00, 2.232e-05
10**4: 3.141591948898189e+00, 7.047e-07
10**5: 3.141592631308735e+00, 2.228e-08
10**6: 3.141592652885214e+00, 7.046e-10
10**7: 3.141592653567513e+00, 2.228e-11
10**8: 3.141592653589095e+00, 6.986e-13

real    0m8.053s
user    0m8.719s
sys     0m3.945s
$
```

ran on 3.1 GHz Intel Core i7

multiprocessing and multithreading

1. Types of Parallel Computing
   - shared and distributed memory multicomputers
2. Multiprocessing in Python
   - pleasingly parallel computation
   - the multiprocessing module
   - numerical integration with multiple processes
3. Multithreading in Python
   - processes and threads
   - life cycle of a thread
   - the Thread class
   - modeling a producer/consumer relation

say hello to the multiprocessing module

```
from multiprocessing import Process
import os
from time import sleep

def say_hello(name, t):
    """
    Process with name says hello.
    """
```

- The function `say_hello()` is executed by every process.
- Each process sleeps a bit to avoid instantaneous termination.
- We print also the process identification number.

the function `say_hello`

```
def say_hello(name, t):
    """
    Process with name says hello.
    """
    print('hello from', name)
    print('parent process :', os.getppid())
    print('process id :', os.getpid())
    print(name, 'sleeps', t, 'seconds')
    sleep(t)
    print(name, 'wakes up')
```

To verify that every process runs indeed on a different process, we check the process identification number (pid) of the process with `getpid()`. With `getppid()`, we get the pid of the parent process.

creating the processes

The script continues:

```
pA = Process(target=say_hello, args = ('A', 2,))
pB = Process(target=say_hello, args = ('B', 1,))
pA.start(); pB.start()
print('waiting for processes to wake up...')
pA.join(); pB.join()
print('processes are done')
```

There are three processes:

- the parent process forks two child processes, and
- two child processes, called `A` and `B` above.

The parent process waits for the child processes to finish.
This waiting is executed with the `join()` applied to a `Process` object.

running the script `multiprocess.py`

```
$ python multiprocess.py
waiting for processes to wake up...
hello from A
parent process : 1131
process id : 1132
A sleeps 2 seconds
hello from B
parent process : 1131
process id : 1133
B sleeps 1 seconds
B wakes up
A wakes up
processes are done
```

multiprocessing and multithreading

1. Types of Parallel Computing
   - shared and distributed memory multicomputers
2. Multiprocessing in Python
   - pleasingly parallel computation
   - the multiprocessing module
   - numerical integration with multiple processes
3. Multithreading in Python
   - processes and threads
   - life cycle of a thread
   - the Thread class
   - modeling a producer/consumer relation

the script simpson4pi1.py

```
from scipy.integrate import simps
from scipy import sqrt, linspace, pi
x = linspace(0, 1, 10**8)
y = sqrt(1-x**2)
I = 4*simps(y, x)
print(I, abs(I - pi))
```

We time the above code on one process:

```
$ time python simpson4pi1.py
3.1415926535890946 6.985523270941485e-13

real    0m7.852s
user    0m7.690s
sys     0m3.905s
$
```

the script simpson4pi2.py

```
from multiprocessing import Process, Queue
from scipy import linspace, sqrt, pi
from scipy.integrate import simps

def call_simpson(fun, a, b, n, q):
    """
    Calls Simpson rule to integrate fun
    over [a, b] using n intervals.
    Adds the result to the queue q.
    """
    x = linspace(a, b, n)
    y = fun(x)
    I = simps(y, x)
    q.put(I)
```

the main program

```
def main():
    """
    The number of processes is given at the command line.
    """
    from sys import argv
    if len(argv) < 2:
        print('Enter the number of processes',)
        print(' at the command line.')
        return
    npr = int(argv[1])
```

We want to run the script as

```
$ time python simpson4pi2.py 3
```

to time the running of the script with 3 processes.

defining processes and queues

```
crc = lambda x: sqrt(1-x**2)
nbr = 10**8
nbrsam = nbr//npr  # integer division
intlen = 1.0/npr
queues = [Queue() for _ in range(npr)]
procs = []
(left, right) = (0, intlen)
for k in range(1, npr+1):
    procs.append(Process(target=call_simpson, \
        args = (crc, left, right, nbrsam, queues[k-1]))
    (left, right) = (right, right+intlen)
```

starting processes and collecting results

```
for process in procs:
    process.start()
for process in procs:
    process.join()
app = 4*sum([q.get() for q in queues])
print(app, abs(app - pi))
```

checking for speedup

```
$ time python simpson4pi2.py 2
3.141592653589087 7.061018436615996e-13

real    0m5.632s
user    0m10.152s
sys     0m5.317s
$ time python simpson4pi2.py 1
3.1415926535890946 6.985523270941485e-13

real    0m7.679s
user    0m7.540s
sys     0m4.013s
$ time python simpson4pi2.py 3
3.1415926535893393 4.53859172466764e-13

real    0m3.288s
user    0m6.214s
sys     0m3.753s
$
```

comparing wall clock times

```
$ time python simpson4pi2.py 1
3.1415926535890946 6.985523270941485e-13

real    0m7.679s
$ time python simpson4pi2.py 3
3.1415926535893393 4.538591724667 64e-13

real    0m3.288s
```

We have 7.679/3.288 = 2.335 with 4 processes.

a first exercise

1. A Monte Carlo method to estimate $\pi/4$ generates random tuples (x, y), with x and y uniformly distributed in $[0, 1]$. The ratio of the number of tuples inside the unit circle over the total number of samples approximates $\pi/4$.

```
>>> from random import seed
>>> seed(2019)  # for reproducible runs
>>> from random import uniform as u
>>> X = [u(0, 1) for _ in range(1000)]
>>> Y = [u(0, 1) for _ in range(1000)]
>>> Z = list(zip(X, Y))
>>> F = [1 for (x, y) in Z if x**2 + y**2 <= 1]
>>> 4*sum(F)/1000
3.188
```

   Use the multiprocessing module to write a parallel version, letting processes take samples independently. Compute the speedup. The seed must be passed to the child processes to ensure that all random sequences are different from each other.

multiprocessing and multithreading

1. Types of Parallel Computing
   - shared and distributed memory multicomputers
2. Multiprocessing in Python
   - pleasingly parallel computation
   - the multiprocessing module
   - numerical integration with multiple processes
3. Multithreading in Python
   - processes and threads
   - life cycle of a thread
   - the Thread class
   - modeling a producer/consumer relation

concurrency and parallelism

First some terminology:

- **concurrency**
  Concurrent programs execute multiple tasks independently.

  For example, a drawing application, with tasks:
  - receiving user input from the mouse pointer,
  - updating the displayed image.

- **parallelism**
  A parallel program executes two or more tasks in parallel with the explicit goal of increasing the overall performance.

  For example: a parallel Monte Carlo simulation for π, written with the multiprocessing module of Python.

Every parallel program is concurrent,
but not every concurrent program executes in parallel.

Parallel Processing

processes and threads

At any given time, many processes are running simultaneously on a computer.
The operating system employs *time sharing* to allocate a percentage of the CPU time to each process.

Consider for example the downloading of an audio file.
Instead of having to wait till the download is complete, we would like to listen sooner.

Processes have their own memory space, whereas threads share memory and other data.
Threads are often called lightweight processes.

A thread is short for *a thread of execution*, it typically consists of one function.
A program with more than one thread is *multithreaded*.

the global interpreter lock

In CPython, the *Global Interpreter Lock*, or GIL, prevents multiple native threads from executing Python bytecodes in parallel.

Why is the GIL necessary?
$\rightarrow$ The memory management in CPython is not thread safe.

Consequence: degrading performance on multicore processors.

For parallel code in Python, use

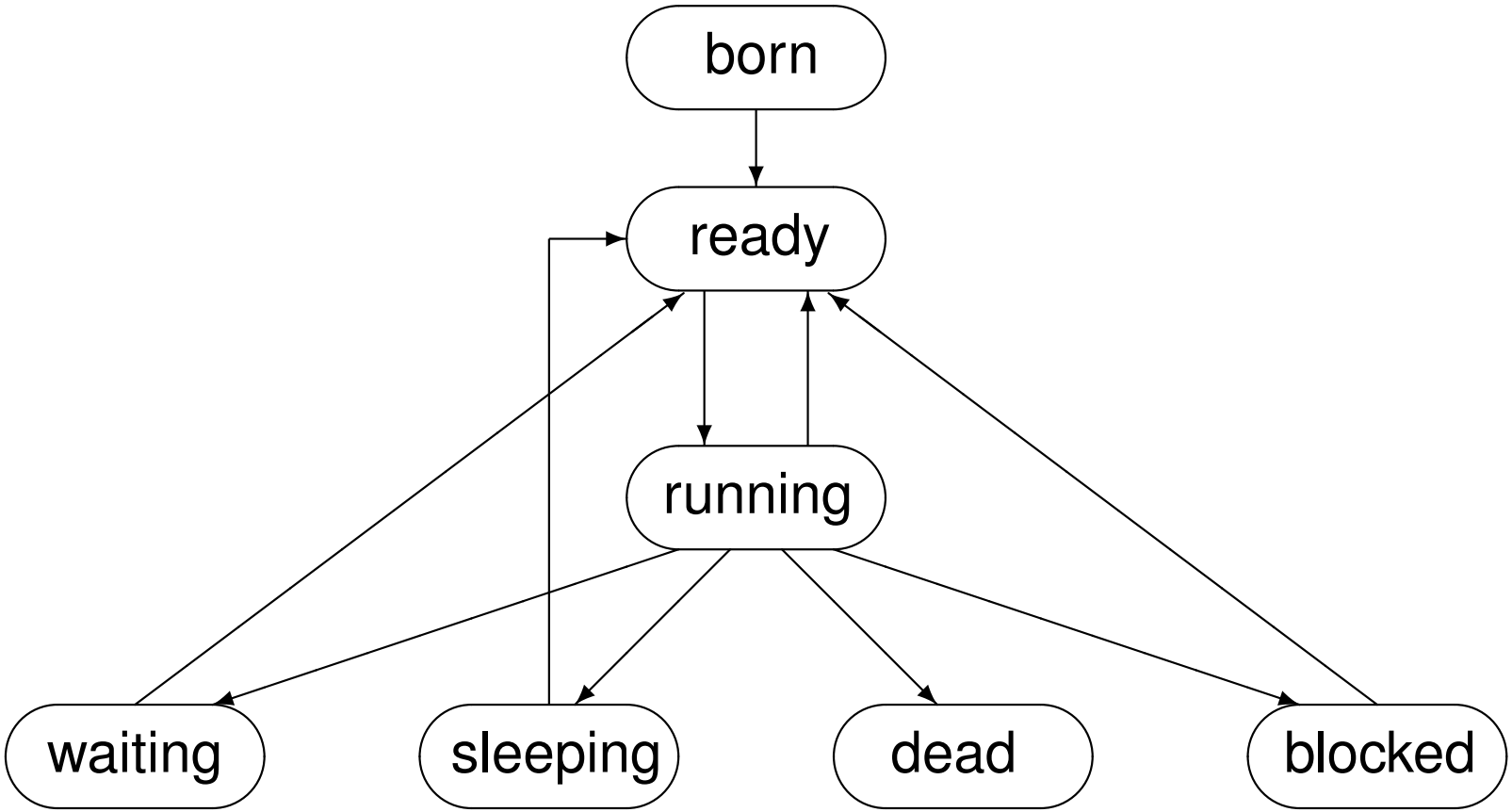
- the multiprocessing module (if sufficient memory available); or
- depend on multithreaded libraries that run outside the GIL.

multiprocessing and multithreading

1. Types of Parallel Computing
   - shared and distributed memory multicomputers
2. Multiprocessing in Python
   - pleasingly parallel computation
   - the multiprocessing module
   - numerical integration with multiple processes
3. Multithreading in Python
   - processes and threads
   - life cycle of a thread
   - the Thread class
   - modeling a producer/consumer relation

the Life Cycle of a Thread

a state diagram



multiprocessing and multithreading

1. Types of Parallel Computing
   - shared and distributed memory multicomputers
2. Multiprocessing in Python
   - pleasingly parallel computation
   - the multiprocessing module
   - numerical integration with multiple processes
3. Multithreading in Python
   - processes and threads
   - life cycle of a thread
   - the Thread class
   - modeling a producer/consumer relation

using the Thread class

an object oriented approach

The `threading` module exports the `Thread` class.

We create new threads by inheriting from `threading.Thread`, overriding the __init__ and `run`.

After creating a thread object, a new thread is born.

With `run`, we start the thread.

Threads are good to model independent actors.

running hello_threading

At the command prompt `$`:

```
$ python hello_threading.py
first thread is born
second thread is born
third thread is born
starting threads
hello from first thread
hello from second thread
hello from third thread
threads started
third thread slept 1 seconds
second thread slept 4 seconds
first thread slept 5 seconds
$
```

the class `HelloThread`

```
import threading

class HelloThread(threading.Thread):
    """
    hello world with threads
    """
    def __init__(self, t):
        """
        initializes thread with name t"
        """
    def run(self):
        """
        says hello and sleeps awhile"
        """
def main():
    """
    Starts three threads.
    """
```

the constructor and run method

```
def __init__(self, t):
    """
    initializes thread with name t"
    """
    threading.Thread.__init__(self, name=t)
    print(t + " is born ")

def run(self):
    """
    says hello and sleeps awhile"
    """
    name = self.getName()
    print("hello from " + name)
    nbr = randint(1, 6)
    sleep(nbr)
    print(name + " slept %d seconds" % nbr)
```

the `main()` in `HelloThread`

```
def main():
    """
    Starts three threads.
    """
    first = HelloThread("first thread")
    second = HelloThread("second thread")
    third = HelloThread("third thread")
    print("starting threads")
    first.start()
    second.start()
    third.start()
    print("threads started")

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

multiprocessing and multithreading

1. Types of Parallel Computing
   - shared and distributed memory multicomputers
2. Multiprocessing in Python
   - pleasingly parallel computation
   - the multiprocessing module
   - numerical integration with multiple processes
3. Multithreading in Python
   - processes and threads
   - life cycle of a thread
   - the Thread class
   - modeling a producer/consumer relation

Producer/Consumer Relation

with threads

A very common relation between two threads is that of producer and consumer. For example, the downloading of an audio file is production, while listening is consumption.

Our producer/consumer relation with threads uses

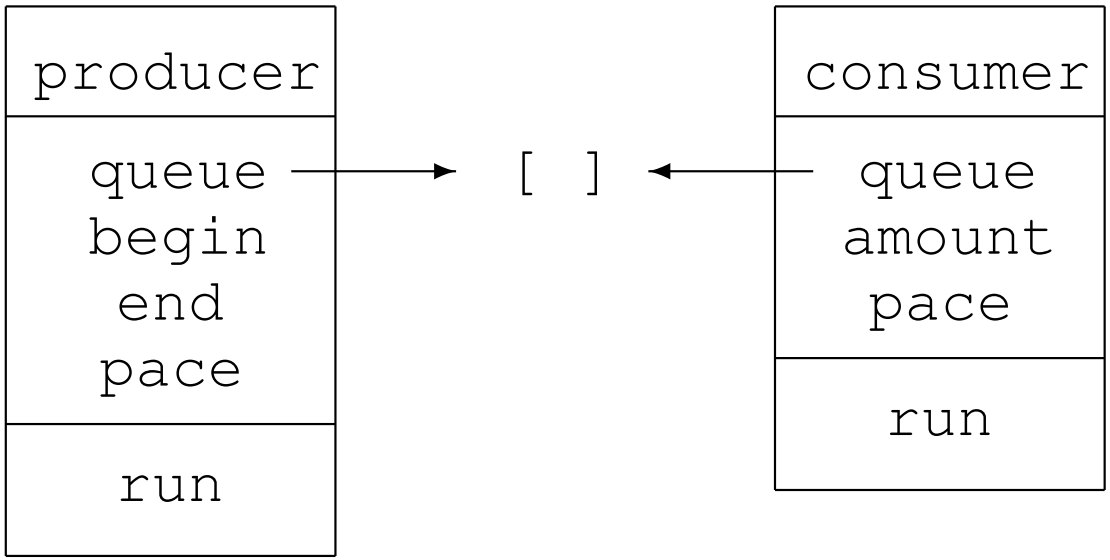
- an object of the class Producer is a thread
  that will append to a queue consecutive integers
  in a given range and at a given pace;
- an object of the class Consumer is a thread
  that will pop integers from the queue and print them,
  at a given pace.

If the pace of the produces is slower than the pace of the consumer, then the consumer will wait.

running the code

```
$ python prodcons.py
producer starts...
producer sleeps 1 seconds
consumption starts...
consumer sleeps 1 seconds
appending 1 to queue
producer sleeps 4 seconds
popped 1 from queue
consumer sleeps 1 seconds
wait a second...
wait a second...
wait a second...
appending 2 to queue
producer sleeps 2 seconds
popped 2 from queue
consumer sleeps 1 seconds
wait a second...
appending 3 to queue
production terminated
popped 3 from queue
consumption terminated
```

UML class diagrams



The `queue` in each class refer to the same list:

- The producer appends to the queue.
- The consumer pops from the queue.

structure of the class `Producer`

```
import threading

class Producer(threading.Thread):
    """
    Appends integers to a queue.
    """
    def __init__(self, t, q, a, b, p):
        """
        Thread t to add integers in [a, b] to q,
        sleeping between 1 and p seconds.
        """
    def run(self):
        """
        Produces integers at some pace.
        """
```

the constructor method

```
def __init__(self, t, q, a, b, p):
    """
    Thread t to add integers in [a, b] to q,
    sleeping between 1 and p seconds.
    """
    threading.Thread.__init__(self, name=t)
    self.queue = q
    self.begin = a
    self.end = b
    self.pace = p
```

the production method

```
def run(self):
    """
    Produces integers at some pace.
    """
    print(self.getName() + " starts...")
    for i in range(self.begin, self.end+1):
        nbr = randint(1, self.pace)
        print(self.getName() + \
            " sleeps %d seconds" % nbr)
        sleep(nbr)
        print("appending %d to queue" % i)
        self.queue.append(i)
    print("production terminated")
```

constructor and run method of the class `Consumer`

```
import threading

class Consumer(threading.Thread):
    """
    Pops integers from a queue.
    """
    def __init__(self, t, q, n, p):
        """
        Thread t to pop n integers from q.
        """
    def run(self):
        """
        Pops integers at some pace.
        """
```

the constructor of the class `Consumer`

```
def __init__(self, t, q, n, p):
    """
    Thread t to pop n integers from q.
    """
    threading.Thread.__init__(self, name=t)
    self.queue = q
    self.amount = n
    self.pace = p
```

consuming elements

```
def run(self):
    """
    Pops integers at some pace.
    """
    print("consumption starts...")
    for i in range(0, self.amount):
        nbr = randint(1, self.pace)
        print(self.getName() + \
            " sleeps %d seconds" % nbr)
        sleep(nbr)
        # code to pop from the queue
    print("consumption terminated")
```

code to pop from the queue

Popping from an empty list raises an `IndexError`.
Placing the code in a `try/except` block
lets us write code to handle the exception.

```
while True:
    try:
        i = self.queue.pop(0)
        print("popped %d from queue" % i)
        break
    except IndexError:
        print("wait a second...")
        sleep(1)
```

the main program in `prodcons.py`

Code for the class Producer and Consumer in modules `classproducer` and `classconsumer` respectively.

```
from classproducer import Producer
from classconsumer import Consumer

QUE = []      # queue is shared list
PROD = Producer("producer", QUE, 1, 3, 4)
CONS = Consumer("consumer", QUE, 3, 1)
PROD.start()  # start threads
CONS.start()
PROD.join()   # wait for thread to finish
CONS.join()
```

a second exercise

2. A Monte Carlo method to estimate $\pi/4$ generates random tuples (x, y), with x and y uniformly distributed in $[0, 1]$. The ratio of the number of tuples inside the unit circle over the total number of samples approximates $\pi/4$.

```
>>> from random import seed
>>> seed(2019)  # for reproducible runs
>>> from random import uniform as u
>>> X = [u(0, 1) for _ in range(1000)]
>>> Y = [u(0, 1) for _ in range(1000)]
>>> Z = list(zip(X, Y))
>>> F = [1 for (x, y) in Z if x**2 + y**2 <= 1]
>>> 4*sum(F)/1000
3.188
```

Use the Thread class to write a parallel version, letting processes take samples independently. Compute the speedup.
The seed must be passed to the threads
to ensure that all random sequences are different from each other.