Multithreaded client-server

DNP lab 2
Innopolis University
Fall 2022

Outline

- Processes and threads
- Threading in Python
- Producer-Consumer model
- A synchronized queue class

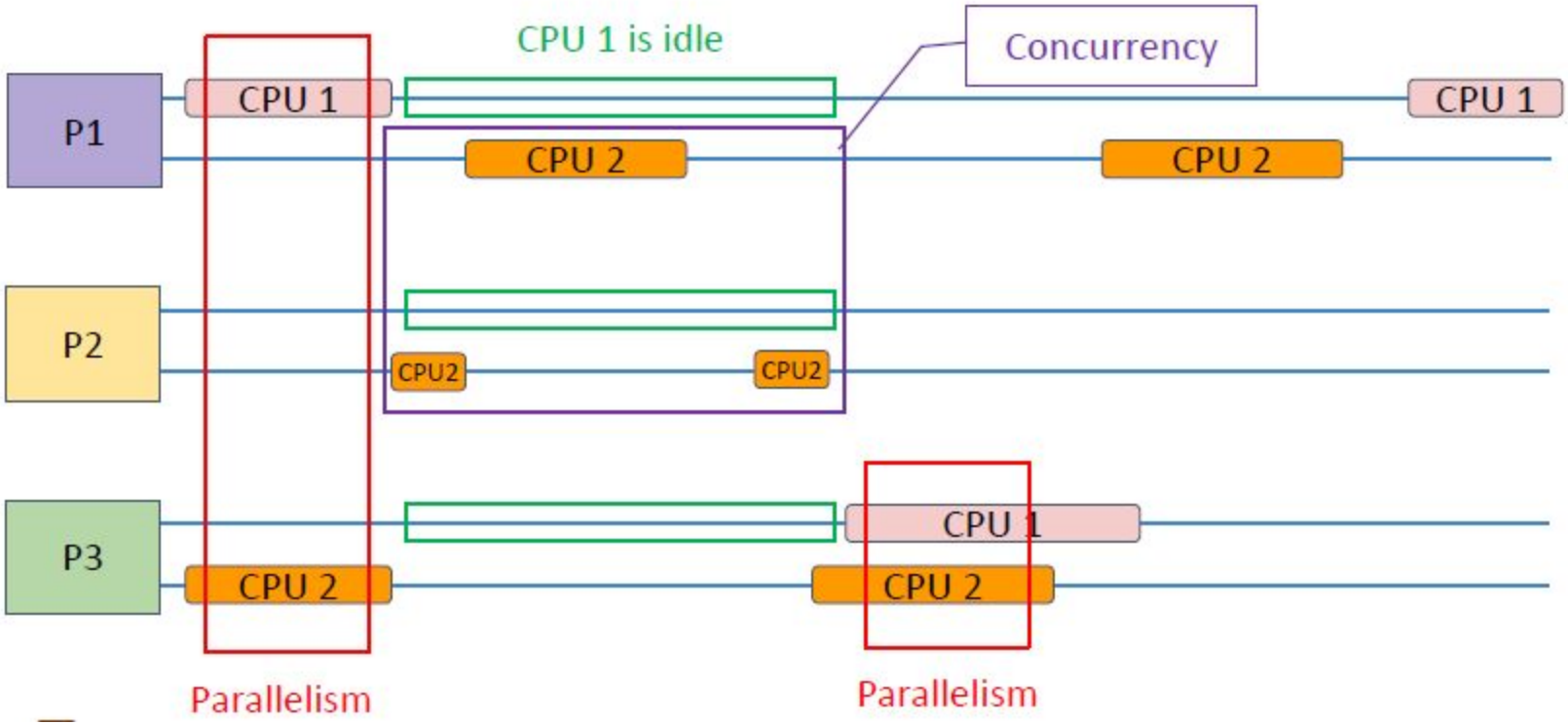
Process

Process is an **instance of a computer program that is being executed**.

Process has three basic components:

- An executable program
- The associated resources/data needed by the program (variables, workspace, buffers, etc.)
- The execution context (state of process)

Concurrency and Parallelism



- **Concurrency** - Multiple processes share the same CPU
- **Parallelism** - Multiple processes are executed simultaneously on different CPUs on parallel

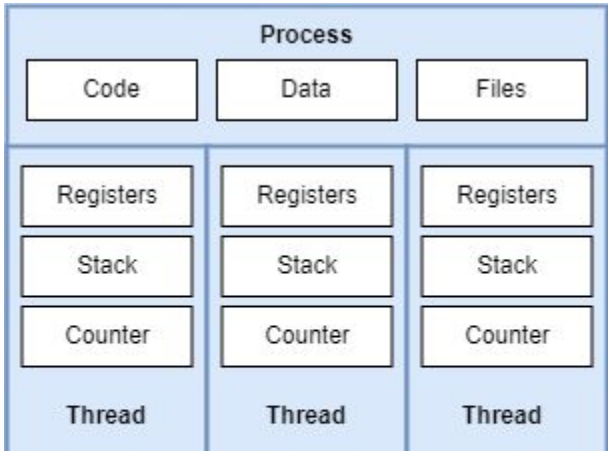
Processes and Threads

Process

- Independent
- Don’t share memory

Thread

- Dependent (child of a process)
- Shares part of process memory
- Lightweight (easier to create and destroy)



Threading in Python

- The threading package contains functionality for creating threads.
- **Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)**
  - **group** - should be None (reserved for future extension)
  - **target** - the callable object to be invoked by the run() method
  - **name** - the thread name
  - **args** - the argument tuple for the target invocation
  - **kwargs** - a dictionary of keyword arguments for the target invocation
  - **daemon** - If not None, daemon explicitly sets whether the thread is daemonic

https://docs.python.org/3/library/threading.html#threading.Thread

Thread class methods

- **start()**
  Start the thread’s activity.

- **run()**
  The standard run() method invokes the callable object passed to the object’s constructor as the **target** argument (you can override it).

- **join(timeout=None)**
  Wait until the thread terminates. This blocks the calling thread until the thread whose join() method is called terminates.

Creating the threads

Creating a single thread

```
1  from threading import Thread
2  from time import sleep
3
4  def greet(name):
5      print(f"Hi {name}")
6      sleep(3)
7      print(f"Bye {name}")
8
9  t = Thread(target=greet, args=("Alexey",))
10 t.start()
11 t.join()
```

✓ 3.7s

```
Hi Alexey
Bye Alexey
```

Creating multiple threads

```
9  threads = [Thread(target=greet, args=(name,))
10             for name in ["Alexey", "Shinnazar", "Vladimir"]]
11 [t.start() for t in threads]
12 [t.join() for t in threads]
```

✓ 3.3s

```
Hi Alexey
Hi Shinnazar
Hi Vladimir
Bye Alexey
Bye Shinnazar
Bye Vladimir
```

Effect of join() call

- **join()** blocks the thread from which it was called.

```
1  t = Thread(target=greet, args=("Alexey",))
2  t.start()
3  print("Main thread is not blocked")
```

✓

```
Hi Alexey
Main thread is not blocked
Bye Alexey
```

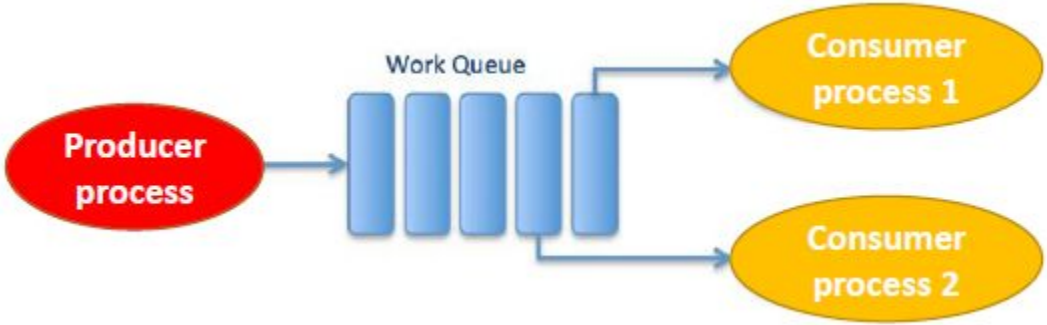
```
1  t = Thread(target=greet, args=("Alexey",))
2  t.start()
3  t.join()
4  print("Main thread is blocked")
```

✓ 3.6s

```
Hi Alexey
Bye Alexey
Main thread is blocked
```

Producer-Consumer model

- some processes **produce** tasks to do and put them in a shared data structure.
- while some other processes **consume** the tasks from shared data structure and do the work.



A shared data structure: Queue class

We can use the **Queue** class from **multiprocessing** module, as our shared data structure:

- A thread safe FIFO (first in first out) queue.
- The queue module implements multi-producer, multi-consumer queues.
- It is especially useful in threaded programming when information must be exchanged safely between multiple processes or threads.
- The Queue class in this module implements all the required locking semantics.

Queue class

Queue(maxsize=0)

- **Constructor** for a FIFO queue.
- **maxsize** is an integer that sets the upperbound limit on the number of items that can be placed in the queue.
- Insertion will **block** once this size has been reached, until queue items are consumed.
- If maxsize is less than or equal to **zero**, the queue size is **infinite**.

```
1 from multiprocessing import Queue
```

```
1 q = Queue()
```

```
1 q.qsize()
```

```
0
```

```
1 q.empty()
```

```
True
```

https://docs.python.org/3/library/queue.html#queue.Queue

Some methods of Queue class

put(item, block=True, timeout=None)

- Put item into the queue.
- If optional **block** is True and **timeout** is None (the default), blocks until a free slot is available.
- If **timeout** is a positive number, it blocks at most timeout seconds and raises the **Full** exception if no free slot was available within that time.
- If **block** is False, put an item on the queue if a free slot is immediately available, else raise the **Full** exception (timeout is ignored in that case).

```
q = Queue(maxsize=3)
```

```
q.put("A")
q.qsize()
```

```
1
```

```
q.empty()
```

```
False
```

```
q.full()
```

```
False
```

```
q.put("B")
q.put("C")
```

```
q.qsize()
```

```
3
```

```
q.full()
```

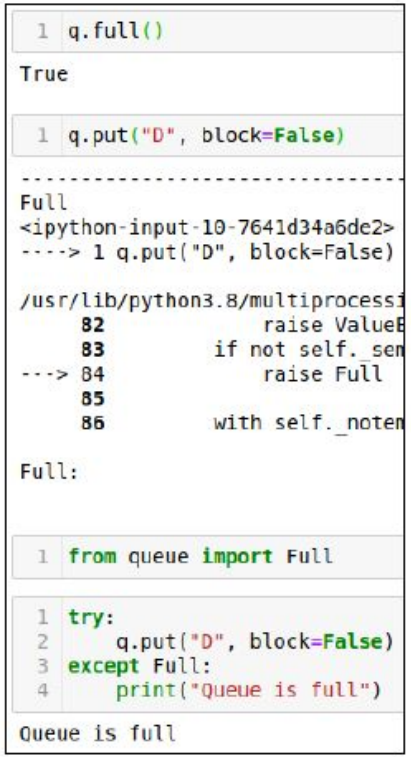
```
True
```

```
# blocks the interpreter
# when queue is full
# blocks until item is put
q.put("D")
```

Switching to non blocking mode

put() blocks the thread if the queue is full

- To overcome this, **disable** blocking mode.
- And **catch** the **Full exception** to prevent the program to fail.



```
1  q.full()
```

```
True
```

```
1  q.put("D", block=False)
```

```
---------------------------------
Full
<ipython-input-10-7641d34a6dc2>
----> 1 q.put("D", block=False)

/usr/lib/python3.8/multiprocessi
     82                 raise ValueE
     83             if not self._sem
---> 84                 raise Full
     85
     86             with self._notem

Full:
```

```
1  from queue import Full
```

```
1  try:
2      q.put("D", block=False)
3  except Full:
4      print("Queue is full")
```

```
Queue is full
```

Some methods of Queue class

get(block=True, timeout=None)

- Remove and return an item from the queue.
- If optional **block** is True and **timeout** is None (the default), block if necessary until an item is available.
- If **timeout** is a positive number, it **blocks** at most timeout seconds and raises the **Empty** exception if no item was available within that time.
- Otherwise (**block** is False), **return an item** if one is immediately available, else **raise** the **Empty exception** (timeout is ignored in that case).

```
q.get()
```
```
'A'
```
```
q.get()
```
```
'B'
```
```
q.qsize()
```
```
1
```
```
q.get()
```
```
'C'
```
```
q.empty()
```
```
True
```
```
# blocks the main program
q.get()
```
```
from queue import Empty

try:
    q.get(block=False)
except Empty:
    print("Queue is empty"
```
```
Queue is empty
```

Summary

- Use the **Thread** class from the **threading** module to create a thread.
- Use the **Queue** class from the **multiprocessing** module to exchange data between threads.
- Be aware of the blocks

The end

Any questions?