Distributed & network programming (2021 Fall)

Lab 01: Sequential (serial) vs Parallel processing

Aug 16, 2021

Prof. Shinnazar Seytnazarov

Faculty of Computer Science & Engineering

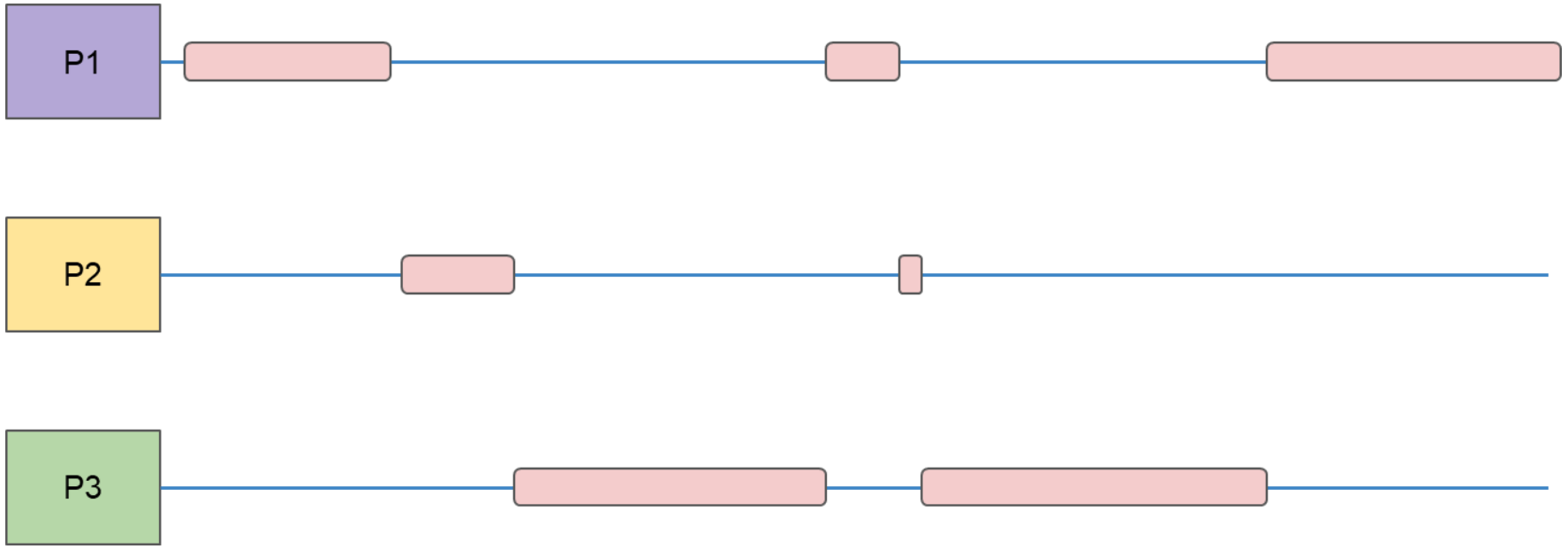
Outline

- **Introduction to process and multiprocessing in Python**
- **Exercise #1: Sequential processing (20pts)**
- **Exercise #2: Parallel processing (40pts)**
  - Using multiprocessing library in Python
- **Exercise #3: Comparison of Sequential and Parallel (40pts)**
  - Drawing plot for different number of processes
  - Explain the results

Traditional processes

- **A process is**
  - an instance of a computer program that is being executed.
- **A 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)

The process scheduling in single core CPU

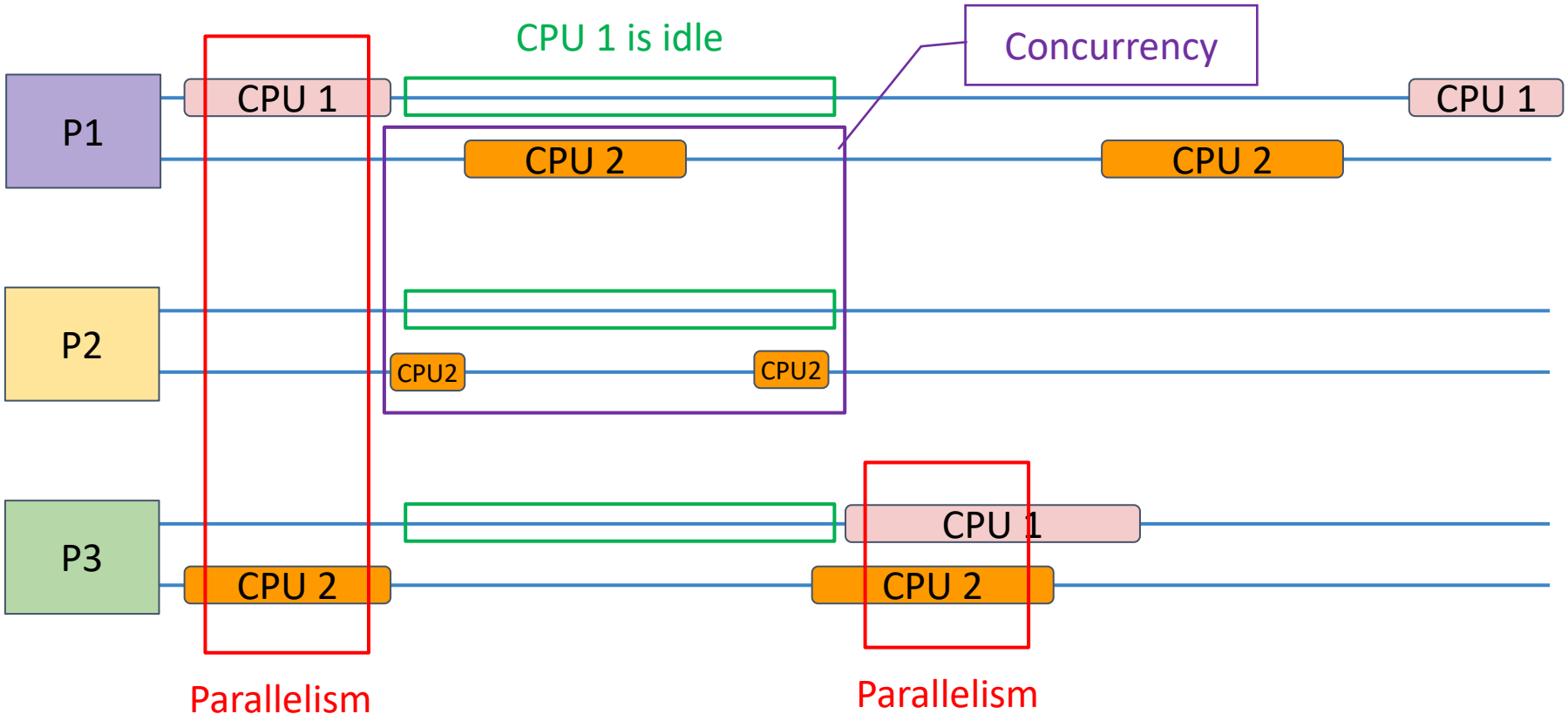


This is what we call:

Concurrency

Several processes are “running” but they share the same CPU.

A two-core CPU



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

Parallel processing in Python

- **Supported by *multiprocessing* module**
- **Process(group=None, target=None, name=None, args=(), kwargs={}, daemon=None)**
  - This constructor should always be called with **keyword arguments**.
  - **group** should be None; reserved for future extension.
  - **target** is the callable object to be invoked by the run() method.
  - **name** is the process name. By default, a unique name is constructed of the form “process-N” where N is a small decimal number.
  - **args** is the argument tuple for the target invocation.
  - **kwargs** is a dictionary of keyword arguments for the target invocation.
  - If not None, **daemon** explicitly sets whether the thread is daemonic. If None (the default), the daemonic property is inherited from the current process.

Process class methods

- **start()**
  - Start the process' activity
- **is_alive()**
  - Return whether the process is alive.
- **join(timeout=None)**
  - Wait until the process terminates. This blocks the calling process until the process whose join() method is called terminates
- **run()**
  - Method representing the thread's activity.
  - You may override this method in a subclass. The standard **run()** method invokes the **target** argument.

Creating the processes

❑ Creating the processes

- Need to call the constructor of Process class with **keyword arguments**

- Creating a single thread
- Need to call **start()** method to run the thread

```
1  from multiprocessing import Process
2  from time import sleep, time
3
4  def greet(name):
5      print(f"{time()}: Hi {name}")
6      sleep(5)
7      print(f"{time()}: Bye {name}")
8
9  p = Process(target=greet, args=("Ivan",))
10 p.start() # starts the process's activity
11 p.join() # blocks the interpreter until t
12 p.close() # closes the Process object, re
```

```
1628930659.8606088: Hi Ivan
1628930664.9100642: Bye Ivan
```

- Creating multiple threads

```
1  processes = [Process(target=greet, args=(name,))
2               for name in ["Ivan", "Alexey", "Vladimir"]]
```

```
1  [p.start() for p in processes]
2  [p.join() for p in processes]
3  [p.close() for p in processes]
```

```
1628930793.4104717: Hi Ivan
1628930793.4286308: Hi Alexey
1628930793.449318: Hi Vladimir
1628930798.4274025: Bye Ivan
1628930798.452116: Bye Alexey
1628930798.4601228: Bye Vladimir
```

Effect of join() call

- **join() blocks the main program**

```
1  p = Process(target=greet, args=("Ivan",))
2  p.start()
3  sleep(1)
4  print("main program isn't blocked")
```

```
1628933949.9598112: Hi Ivan
main program isn't blocked
1628933954.9769063: Bye Ivan
```

```
1  p.close()
```

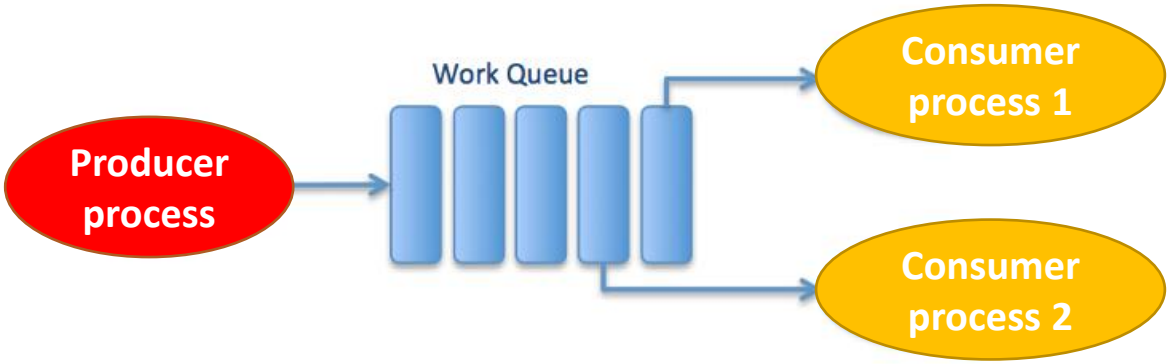
```
1  p = Process(target=greet, args=("Ivan",))
2  p.start()
3  p.join()
4  print("main program is blocked")
5  p.close()
```

```
1628933998.7069: Hi Ivan
1628934003.7225947: Bye Ivan
main program is blocked
```

Producer-Consumer model

❑ 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.
  - 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
```

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).

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

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

```
1
```

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

```
False
```

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

```
False
```

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

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

```
3
```

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

```
True
```

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

Switching to non-blocking mode

- **put() blocks the main program if the queue is full**
  - To overcome this, disable blocking mode
  - And catch the Full exception to prevent the program to stop

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

```
True
```

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

```
---------------------------------
Full
<ipython-input-10-7641d34a6de2>
----> 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).

```
1 q.get()
```

```
'A'
```

```
1 q.get()
```

```
'B'
```

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

```
1
```

```
1 q.get()
```

```
'C'
```

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

```
True
```

```
1 # blocks the main program
2 q.get()
```

```
1 from queue import Empty
2
3 try:
4     q.get(block=False)
5 except Empty:
6     print("Queue is empty"
```

```
Queue is empty
```

Practice session

- **Download the Lab01_assignment.ipynb file and solve the problems there**
  - You can also use .pdf file
- **Submit your solution in jupyter-notebook (.ipynb) format**
  - Jupyter-notebook on Windows doesn't fully support Python's multiprocessing module
    - but it does fully support multiprocessing on Linux
    - Not sure about MacOS
  - If you don't really want to work on Linux for some reason
    - solve the problems on one of the Windows IDEs: ex) vscode fully support multiprocessing module on Windows
    - then copy the code into jupyter-notebook and submit.
- **Deadline 2021.08.22 PM11:59**
  - For each day of late submission, 5% will be cut
    - Ex) You submitted the next day after deadline, i.e., a day later, suppose you scored 70%, then your final score will be 65%.

Resources

- https://docs.python.org/3/library/multiprocessing.html
- https://docs.python.org/3/library/multiprocessing.html#the-process-class
- https://docs.python.org/3/library/multiprocessing.html#pipes-and-queues

Any questions?