Using More Processes

Dan Tofan

Software Engineer, PhD

@dan_tofan | www.programmingwithdan.com

Module Overview

Process-based parallelism

Processes communication

When to use more processes

Scaling from one to more machines

Limitations of Multithreaded and Asynchronous Code

Potential for bugs

Learning curve

CPU-intensive tasks

More Processes

Separate memory spaces

No GIL for processes

Utilize all CPU cores

Increased memory overhead

Harder to share resources

```
from multiprocessing import Process
```

◀ **Import the multiprocessing module**

```
class OrderProcessing(Process):
    def run(self):
        print(f"Processing")
```

◀ **Create a subclass of Process**

```
process = OrderProcessing()
process.start()
```

◀ **Create new instance**

◀ **Start new process**

```
def process_order():
    print(f"Processing...")
```

◀ **Python function**

```
p = Process(target=process_order)
p.start()
```

◀ **Create new process**

◀ **Start new process**

Demo

Process orders with multiprocessing

Run a CPU-intensive task

- One process
- Two processes

Applications with More Processes

Processes communication

Overhead of processes communication

Communication between Processes

Handle exceptions

Synchronize processes

Balance workloads

How to Communicate

Pipes

Queues

```
import multiprocessing
from multiprocessing import Process
from random import randint


def producer(queue):
    for i in range(10):
        queue.put(randint(1, 100))
    queue.put(None)


def consumer(queue):
    while True:
        item = queue.get()
        if item:
            print(f'Processed {item}')
        else:
            break
```

◂ **Producer function**

◂ **Consumer function**

```
if __name__ == '__main__':

    queue = multiprocessing.Queue()

    consumer_process =
        Process(target=consumer,
                args=(queue,))

    producer_process =
        Process(target=producer,
                args=(queue,))



    consumer_process.start()
    producer_process.start()

    producer_process.join()
    consumer_process.join()
```

- Dedicated queue
- Send queue as argument
- Send queue as argument
- Start processes
- Wait for processes to finish

Multiprocessing vs Threading/Asyncio

Multiprocessing	VS	Threading/asyncio
More processes		One or more threads in a process
Can use more CPU cores		Can use only one CPU core
CPU-intensive tasks		I/O-intensive tasks
More isolated		Less isolated

Use Cases

Data pipeline

Producer-consumer applications

Parallelizable workloads

Implementing Multi-process Applications

Use logging, monitoring

Terminate cleanly

Access to shared resources

Limit the number of processes

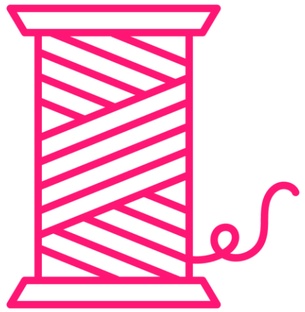
Demo

Clean orders

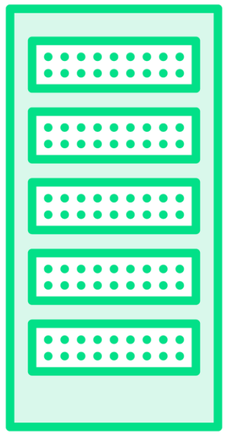
Compare performance

- Processes
- Threads

Scaling



Threading and
asyncio on a CPU core



Multiprocessing on
more CPU cores

Use
more machines

Celery

Task queue

Data processing

Run on many worker machines

Dask

Integrates with NumPy, Pandas

Data science

Scaling from laptop to cluster

Ray

Framework for scaling Python applications

Scaling from laptop to cluster

Designed to be general purpose

Machine learning workloads

Kubernetes

General purpose

Autoscaling

Cloud managed service

Demo

Play with the Dask library

- Install Dask
- Clean orders

Module Summary

- Process-based parallelism
- Processes communication
- When to use more processes
- Scaling from one to more machines

Course Summary

- Measuring performance
- Using the right data structures
- Optimizing Python code
- Using more threads
- Using asynchronous code
- Using more processes

Up Next:

Your Turn