Processes and Threads

Kai Striega

2025-12-13

Who am I?

- Self-taught developer
- Work as a Senior Software Engineer at Endgame Analytics
- Volunteer in the Free and Open Source community
- Organised the Scientific Python Specialist Track at PyCon Australia
- SheCodes mentor since early 2022

Before I start

Before I start

Here be dragons

This talk introduces advanced topics, quickly. The content is designed to stretch your understanding and to challenge you.

Before I start

Here be dragons

This talk introduces advanced topics, quickly. The content is designed to stretch your understanding and to challenge you.

Questions

I am happy to take questions during the talk, feel free to ask if something doesn't make sense.

Expanding your technical toolbox

1. As a programmer, you're continually building up your technical toolbox
2. This toolbox is the sum total of all the tools that you have available to solve any given problem
3. Processes or threads provide another, extremely powerful tool to add to your toolbox
4. Helps you understand what happens when a program executes
5. "Process vs Thread" is also a common interview question

Today's Analogy

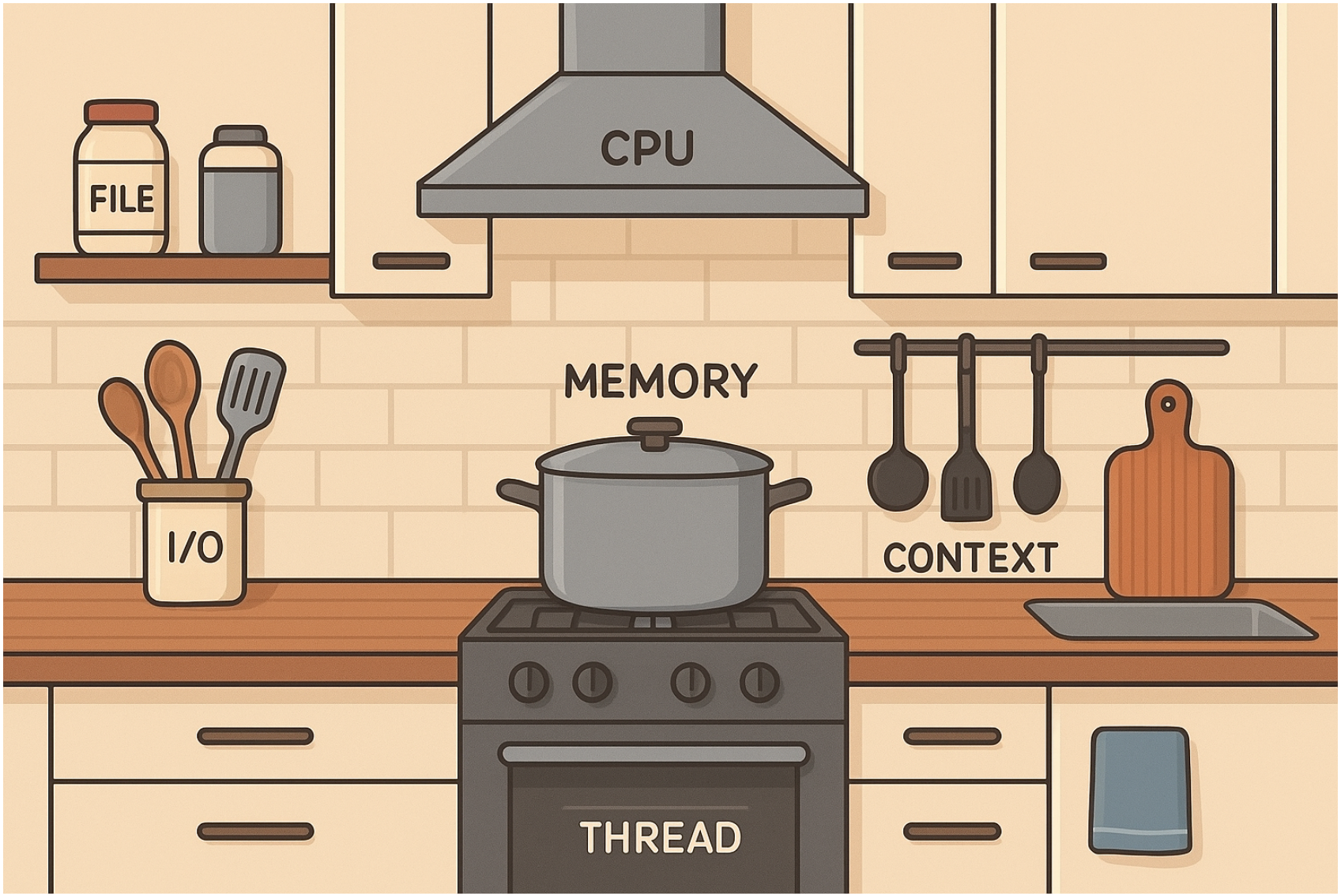
- Processes and Threads can be difficult to understand without context
- So I'll teach by using an analogy
- This analogy is cooking

Why is cooking relevant?

- So far you've learnt to write simple Python programs
- You've learnt to combine data (ingredients) using code (a recipe)
- a.k.a. you've learnt to cook small meals by yourself
- But you need more to cook a meal than just ingredients and a recipe

Processes

A process is the kitchen



A Processes

- A process provides **resources** with which you can cook your meal
  - Memory to store your variables and instructions
  - CPU Time The OS schedules CPU time by processes
  - System Resources give access to things like file handlers and network connections

A Processes

Process

An instance of a program in execution, together with its associated resources and execution context.

- A process provides **resources** with which you can cook your meal
  - Memory to store your variables and instructions
  - CPU Time The OS schedules CPU time by processes
  - System Resources give access to things like file handlers and network connections

Viewing processes

- On Linux/MacOS it is possible to view processes by using the *ps u* command
- This gives us a lot of information:

User	Who owns the processes?
PID	What is the process ID?
%CPU	How much CPU is the process using, as a percentage of total available?
%MEM	How much memory is the process using, as a percentage of total available?
RSS	resident set size, the non-swapped physical memory that a task has used
STAT	process state. What state is the process currently in?

Creating our own process

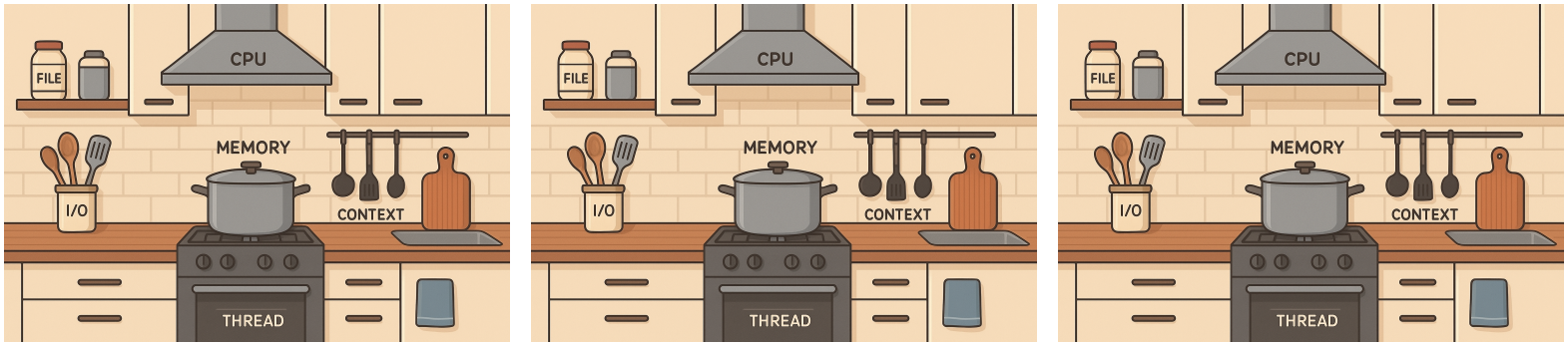
- Let's say we have a very simple Python program:

```
if __name__ == "__main__":
    name = input("Enter your name:")
```

- We can now launch it and see some things:
  - *ps u | grep python*

Processes for parallelism

- So far all your recipies do one thing at a time
- ...but computers can do many things at the same time
- This is analogous to having multiple kitchens to cook in



Launching multiple processes

- Naively we can launch Python multiple times by launching multiple terminals
- This works but is a hassle
- Instead we can use the inbuilt “multiprocessing” module
- The module allows us to launch a processes from another process

Creating processes from another process

```
from multiprocessing import Pool
from time import sleep
NAMES = ["Kai", "Tessa", "Jess", "Lucy"]

def say_hello(name):
    sleep(10)
    print(f"Hi {name}!")

def main():
    with Pool(4) as pool:
        pool.map(say_hello, NAMES)

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

Check-in quiz

1. What is the output of this program?
2. How long does this program take to execute?

Summary of Processes

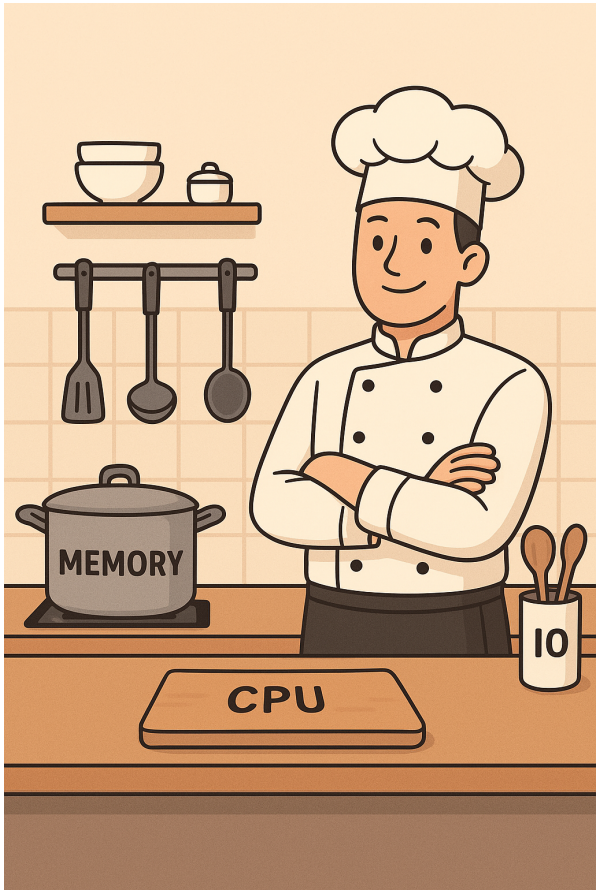
We've covered a lot so far. Let's take a second to revise. We have:

- defined processes as the operating system's representation of a running program
- seen that we can show processes using the ps command
- explained the output of (some) of the ps command's outputs
- shown how to launch multiple processes, both through the terminal and via Python
- shown that using multiple processes can introduce parallelism, allowing us to run many functions at the same time

Threads

Threads

- Processes build the whole kitchen
- ...**every time** we want to do parallelism
- This can be highly inefficient
- What about sharing resources within the kitchen?
- Threads are analogous to multiple cooks in the kitchen all utilising the same resources



Many Threads

Creating threads from a process

```
from concurrent.futures import ThreadPoolExecut
from time import sleep
NAMES = ["Kai", "Tessa", "Jess", "Lucy"]

def say_hello(name):
    sleep(10)
    print(f"Hi {name}!")

def main():
    with ThreadPoolExecutor(4) as p:
        p.map(say_hello, NAMES)

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

Seeing threads from a process

- Let's run the trusty *ps u* | *grep python* command again

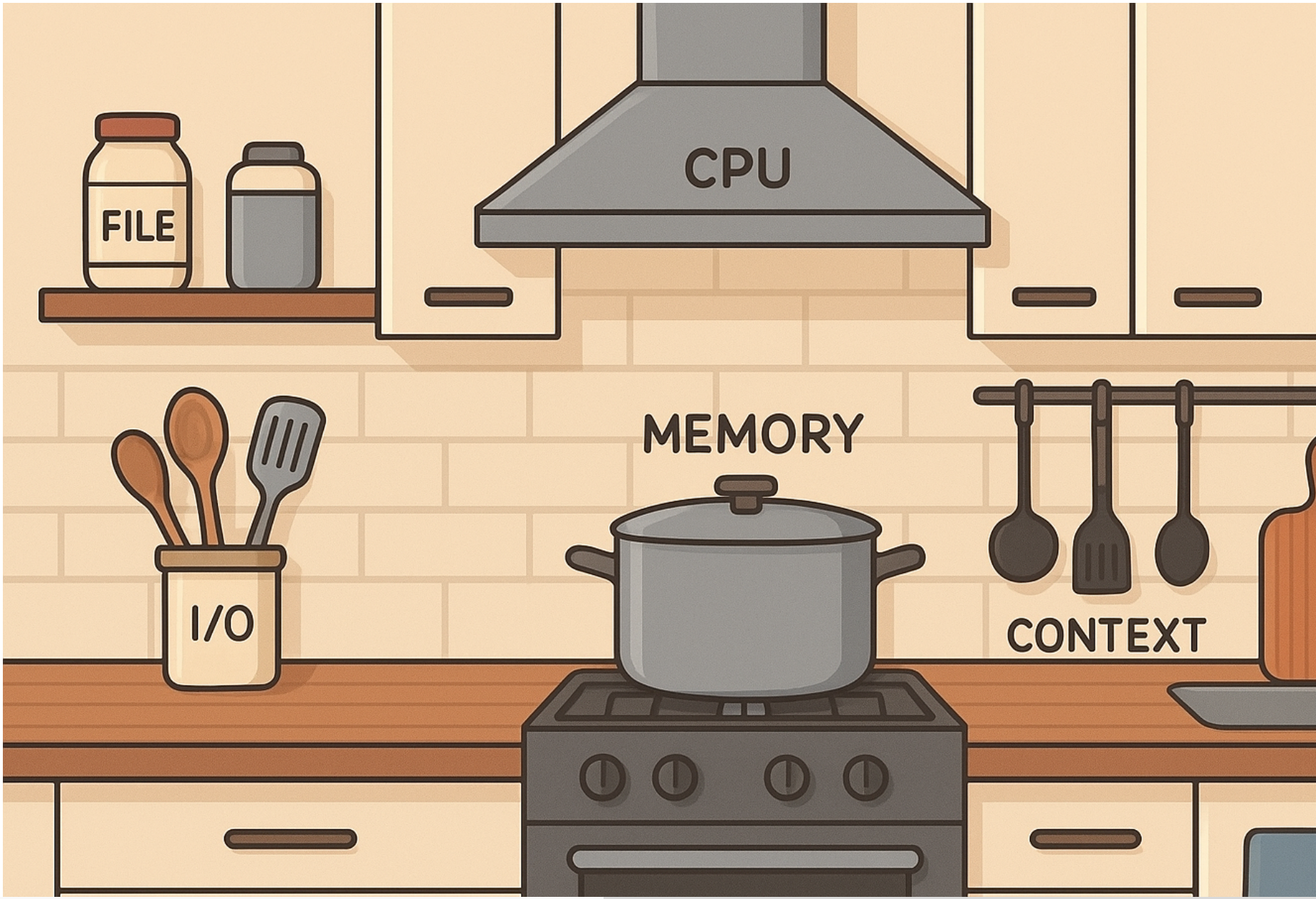
Seeing threads from a process

- Let's run the trusty *ps u | grep python* command again
- What do we notice here?
  - Only one process appears
  - The amount of virtual memory is much higher

What do we notice about threads

- We need to use the $-T$ flag to see threads
- The *PID* of each thread is the same
- This is because the thread runs *inside* the process
- This is significant, threads share the resources provided by the process

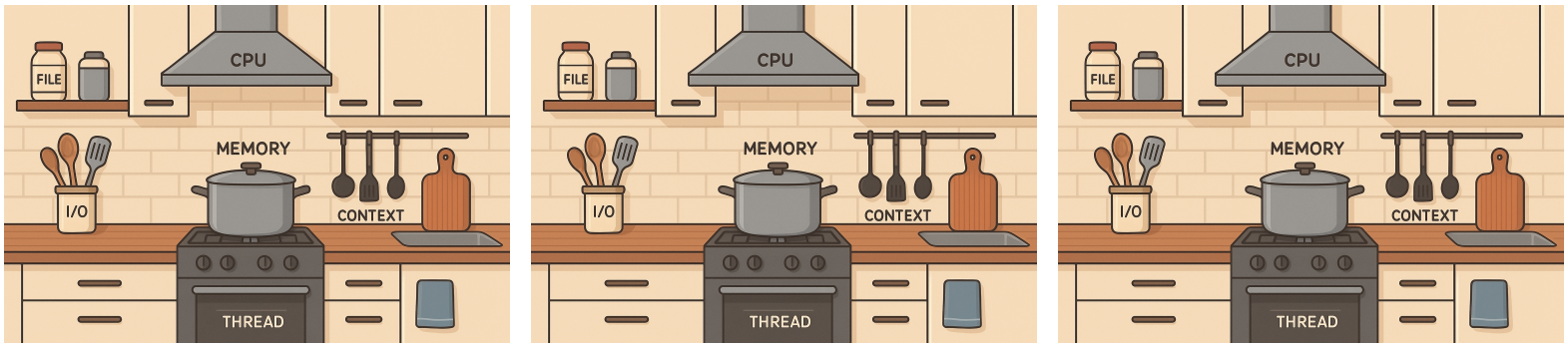
How do threads and processes relate?



A kitchen must always have a chef

You can have multiple chefs per kitchen

And you can have multiple kitchens



You can even have multiple chefs in multiple kitchens



Processes vs Threads

- We've talked about processes and threads individually
- Made the analogy of a process as the kitchen and threads as the chef in the kitchen
- But I don't think the example really demonstrates the tradeoffs:
  - Speed
  - Memory
  - Bugs
- I'm going to demonstrate the difference between processes and threads in practice

Digression: Measuring Performance

- Measuring is nuanced, difficult to get right and is a distinct area of study
- My measurements here are very approximate
- This works because the sizes are massive
- **Don't do this in the real world**

Threads are much quicker to create

- A process needs to manage its own memory and resources
- $\Longrightarrow$ the process of spawning (or creating) a process is much slower than spawning a thread
- Let's test this

Speed benchmark setup

```
from functools import wraps
from time import time


def timed(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        start = time() * 1_000
        result = f(*args, **kwargs)
        end = time() * 1_000
        time_ms = end - start
        print(f"{f.__name__}: {time_ms:.2f}ms")
        return result
    return wrapper
```

Speed benchmark setup

```
from threading import Thread
from timed import timed

@timed
def run_thread():
    t = Thread(target=lambda: None)
    t.start()
    t.join()

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

Speed benchmark setup

```
from multiprocessing import Process
from timed import timed

@timed
def run_process():
    p = Process(target=lambda: None)
    p.start()
    p.join()

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

Threads use much less memory

- Creating copies of memory requires more memory (*duh!*)
- Threads do not create copies of memory while processes do
- Threads therefore use much less memory, but how much?
- Let's find out!

Speed benchmark setup

```
from time import sleep
from concurrent.futures import ThreadPoolExecut
XS = [x for x in range(1_000_000)]

def dummy_work(x):
    sleep(3)
    return x

if __name__ == "__main__":
    with ThreadPoolExecutor(3) as pool:
        pool.map(dummy_work, [XS, XS, XS])
```

Speed benchmark setup

```
from time import sleep
from multiprocessing import Pool
XS = [x for x in range(1_000_000)]

def dummy_work(x):
    sleep(3)
    return x

if __name__ == "__main__":
    with Pool(3) as pool:
        pool.map(dummy_work, [XS, XS, XS])
```

Threads look great! Why would I ever want to use a process?

- If a thread crashes all threads in the process will (usually) also crash
- There are many, very difficult to debug bugs that arise from shared memory
  - Race Conditions
  - Deadlocks
  - Livelocks
  - Thread Starvation
  - Priority Starvation
  - Memory consistency errors
  - Atomicity violations

Conclusion

I hope that you:

- enjoyed this session
- learnt something about processes and threads
- remember that you won't be an expert (yet!)