Multiprocessing using Python

CPEN333 - 2021 W2

University of British Columbia

Introduction

- We have already studied some fundamental concepts related to processes.

- Here we look at how we can implement a program that can use two or more processes. This is one fundamental way in which we can implement concurrency or parallelism.

- In this set of slides, we focus on the Python's multiprocessing module to achieve parallel execution using processes.
  - This will allow truly parallel execution (given capable hardware), working around the GIL limitation.

Objectives

- Learn the multiprocessing module of Python's standard library
- Use the API to implement programs that can create new processes to multi-task, join and terminate
- Understand CPython's GIL

Let’s start with an example

- First a simple single-process program:

```
def sayHello():
    print("Hello from the function sayHello")

if __name__ == "__main__":
    print("This is the main process: beginning")
    sayHello()
    print("This is the main process: end")
```

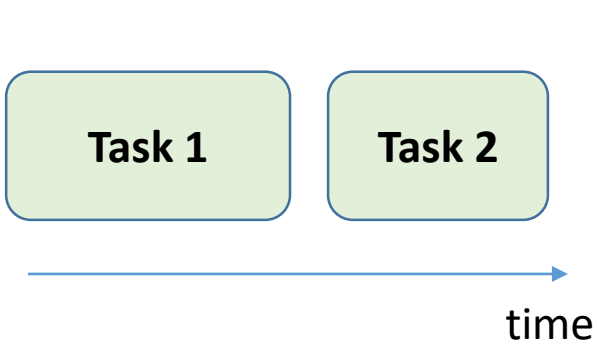
Output:

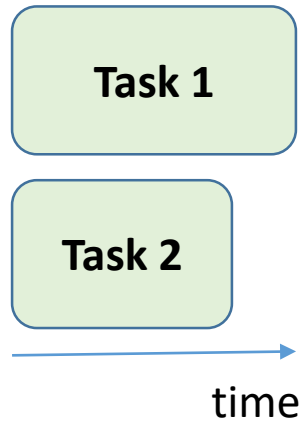
```
This is the main process: beginning
Hello from the function sayHello
This is the main process: end
```

Using a second process

- In the previous example, what if we want to create and delegate the execution of the sayHello method to another process (a worker).
- Why would we want to do that?
  - maybe we want to use the capabilities of the hardware (e.g. CPU cores) and run two things at the same time.
  - maybe we want to make our program more responsive (for example one process dealing with user interface and another with another task)
  - …
- For example:

(hardware and context dependant)





Example: using a second process

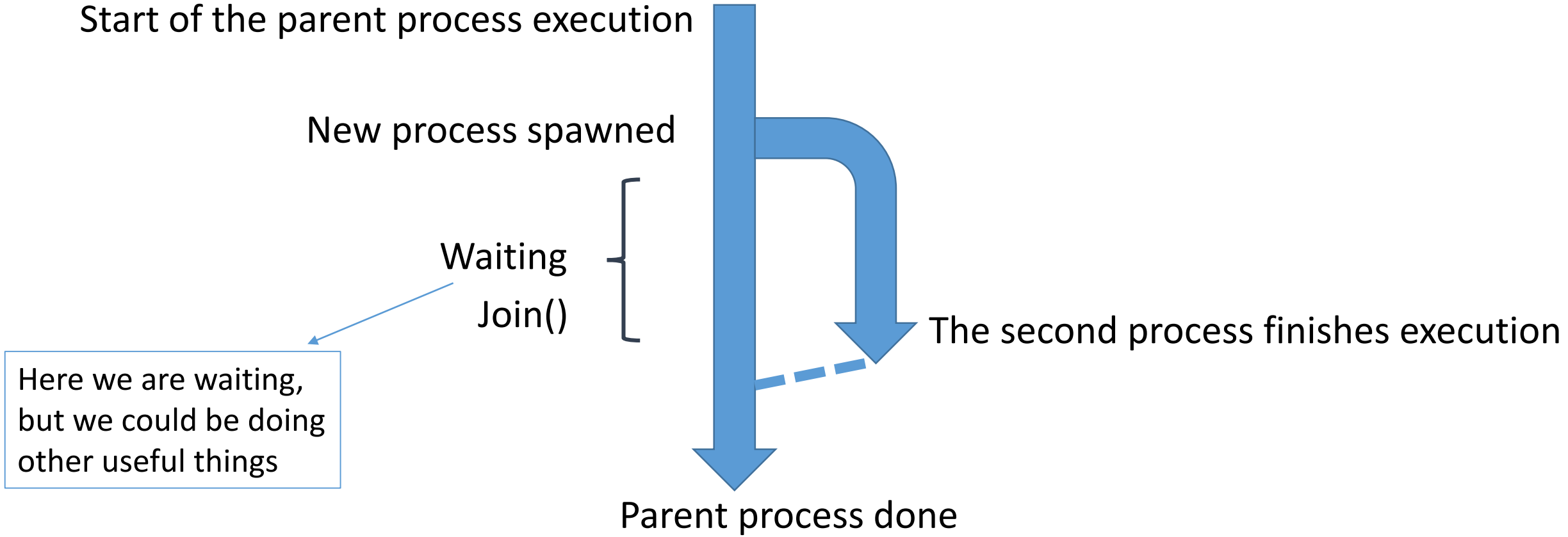
```
import multiprocessing as mp

def sayHello():
    print("Hello from the child process")

if __name__ == "__main__":
    print("This is the main process, before starting a new process")
    p1 = mp.Process(target=sayHello)
    p1.start()
    p1.join()
    print("This is the main process after joining")
```

Output:

```
This is the main process, before starting a new process
Hello from the child process
This is the main process after joining
```

A simple visualization

- The following simple graph shows the flow of execution in our very simple code for our parent process and the newly spawned process



multiprocessing module

- multiprocessing is a package from Python's standard library.

- It allows us to fully leverage multiple processors (e.g. CPU cores) of a given machine.
  - To use it:

```
import multiprocessing
```

- The following simple program will print the number of CPU's in a given system:

```
import multiprocessing

# cpu_count() returns the number of available CPUs
print("CPU count: ", multiprocessing.cpu_count())
```

The multiprocessing.Process class

- We can use the Process class to create a process object.
- Then we use the start method to spawn the new process.
- In our previous example, we did not pass any parameters to the new process.
- Here is an example that we pass a string:

```
from multiprocessing import Process

def sayHello(name):
    print("Hello", name)

if __name__ == '__main__':
    p = Process(target=sayHello, args=("Victor",))
    p.start()
    p.join()
```

This comma is needed to indicate that this is a tuple

multiprocessing.Process documentation

- https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process

class multiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, *, *daemon=None*)

Process objects represent activity that is run in a separate process. The `Process` class has equivalents of all the methods of `threading.Thread`.

The constructor should always be called with keyword arguments. *group* should always be `None`; it exists solely for compatibility with `threading.Thread`. *target* is the callable object to be invoked by the `run()` method. It defaults to `None`, meaning nothing is called. *name* is the process name (see `name` for more details). *args* is the argument tuple for the target invocation. *kwargs* is a dictionary of keyword arguments for the target invocation. If provided, the keyword-only *daemon* argument sets the process `daemon` flag to `True` or `False`. If `None` (the default), this flag will be inherited from the creating process.

By default, no arguments are passed to *target*.

If a subclass overrides the constructor, it must make sure it invokes the base class constructor (`Process.__init__()`) before doing anything else to the process.

Changed in version 3.3: Added the *daemon* argument.

start() and join()

- start(): Start the process's activity.
  - This must be called at most once per process object.

- join([timeout]):
  - If the optional argument timeout is None (the default), the method blocks until the process whose join() method is called terminates.
  - If timeout is a positive number, it blocks at most timeout seconds.

Method of starting processes

- There are three distinct methods of starting processes: spawn, fork and forkserver

- Fork: the mechanism available and default on Unix only.
  - The fork()+exec() is historically the execution model in Unix.
  - Since python version 3.8, macOS does not use fork (considered unsafe).
- Forkserver: This method is only available for select Unix platform.

- So we need to mainly focus on the spawn method.

Spawn

- Spawning creates a second distinct python interpreter process, including its own GIL (we will discussed GIL soon).
  - Starting the process activity by calling start()
  - Spawn is default on Windows and macOS (since version 3.8)

- Why is this important?
  - A new process created using spawn will begin execution from the top of the program.
  - So we need to use a `if __name__=="__main__":` block to specify the code intended for the main process.
  - Failing to do so causes *RuntimeError* or may cause ad infinitum creation of process recursively.

Daemon Processes

- A daemon process continues to run until done or until the main process is terminated (whichever occurs first).
  - A note from the documentation: "… these are **not** Unix daemons or services, they are normal processes that will be terminated (and not joined) if non-daemonic processes have exited."

- We can use the daemon flag to daemonize a process (flag set to True)

```
dp = multiprocessing.Process(target = daemonProcess)
dp.daemon = True
```

  - The flag must be set before `start()` is called. Alternatively, we can use the `daemon` argument of the `Process` constructor.

Example d1

```
import multiprocessing as mp
import time

def myProcess():
    print("Starting the child Process")
    print(f"Child process started: {mp.current_process()} pid = {mp.current_process().pid}")
    time.sleep(3)
    print("Child process terminating ...")

if __name__ == "__main__":
    print(f"Main process: {mp.current_process()} pid = {mp.current_process().pid}")
    p = mp.Process(target=myProcess, name="child")
    p.daemon = True
    p.start()
    print("Let's see if the child process continues to execute")
    time.sleep(6)
    print("Main process terminating ...")
```

Notes:

- We are just waiting using `sleep` in main (e.g. instead of `join`) in order to be able to examine these different scenarios.
- Instead of multiprocessing.current_process().pid, we could use os.getpid() (from the os module)

Example d1 output

- Note that:
  - The child process is a daemon process
  - Due to sleep, the main process waits long enough for the child process to complete and terminate, before it itself terminating.
- A sample output from the program is:

```
Main process: <_MainProcess name='MainProcess' parent=None started> pid = 7780
Let's see if the child process continues to execute
Starting the child Process
Child process started: <Process name='child' parent=7780 started daemon>  pid = 4176
Child process terminating ...
Main process terminating ...
```

Example d2

- Example d2 is similar to Example d1, except for the duration of `sleep` in the main process (marked below):

```
import multiprocessing as mp
import time

def myProcess():
    print("Starting the child Process")
    print(f"Child process started: {mp.current_process()} pid = {mp.current_process().pid}")
    time.sleep(3)
    print("Child process terminating ...")

if __name__ == "__main__":
    print(f"Main process: {mp.current_process()} pid = {mp.current_process().pid}")
    p = mp.Process(target=myProcess, name="child")
    p.daemon = True
    p.start()
    print("Let's see if the child process continues to execute")
    time.sleep(1)
    print("Main process terminating ...")
```

Example d2 output

- Note that:
  - The child process is a daemon process
  - The main process does not wait long enough for the child process to complete and terminate.
  - So the main process terminates before the child complete and the child process is terminated prematurely.
- A sample output from the program is:

```
Main process: <_MainProcess name='MainProcess' parent=None started> pid = 9056
Let's see if the child process continues to execute
Starting the child Process
Child process started: <Process name='child' parent=9056 started daemon>  pid = 2444
Main process terminating ...
```

Example d3

- Now let's compare example d2 with a non-daemonic child process:

```
import multiprocessing as mp
import time

def myProcess():
    print("Starting the child Process")
    print(f"Child process started: {mp.current_process()} pid = {mp.current_process().pid}")
    time.sleep(3)
    print(f"Child process (parent pid = {mp.parent_process().pid}) terminating ...")

if __name__ == "__main__":
    print(f"Main process: {mp.current_process()} pid = {mp.current_process().pid}")
    p = mp.Process(target=myProcess, name="child")
    p.daemon = False
    p.start()
    print("Let's see if the child process continues to execute")
    time.sleep(1)
    print("Main process terminating ...")
```

Example d3 output

- Note that:
  - The child process is not a daemon process (non-daemonic)
  - So even though the main process is done before the child process completes, the child process continues and completes before terminating.
  - This is the default for non-daemonic child processes, where the main program will not exit until all the children have exited (`multiprocessing` tries to make sure that programs using it behave well).
- A sample output from the program is:

```
Main process: <_MainProcess name='MainProcess' parent=None started> pid = 15020
Let's see if the child process continues to execute
Starting the child Process
Child process started: <Process name='child' parent=15020 started>  pid = 7292
Main process terminating ...
Child process (parent pid = 15020) terminating ...
```

Terminating a process

- We can use the *terminate()* method on our process object to terminate it.

- https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process.terminate

terminate()

Terminate the process. On Unix this is done using the `SIGTERM` signal; on Windows `TerminateProcess()` is used. Note that exit handlers and finally clauses, etc., will not be executed.

Note that descendant processes of the process will *not* be terminated – they will simply become orphaned.

Example: terminate()

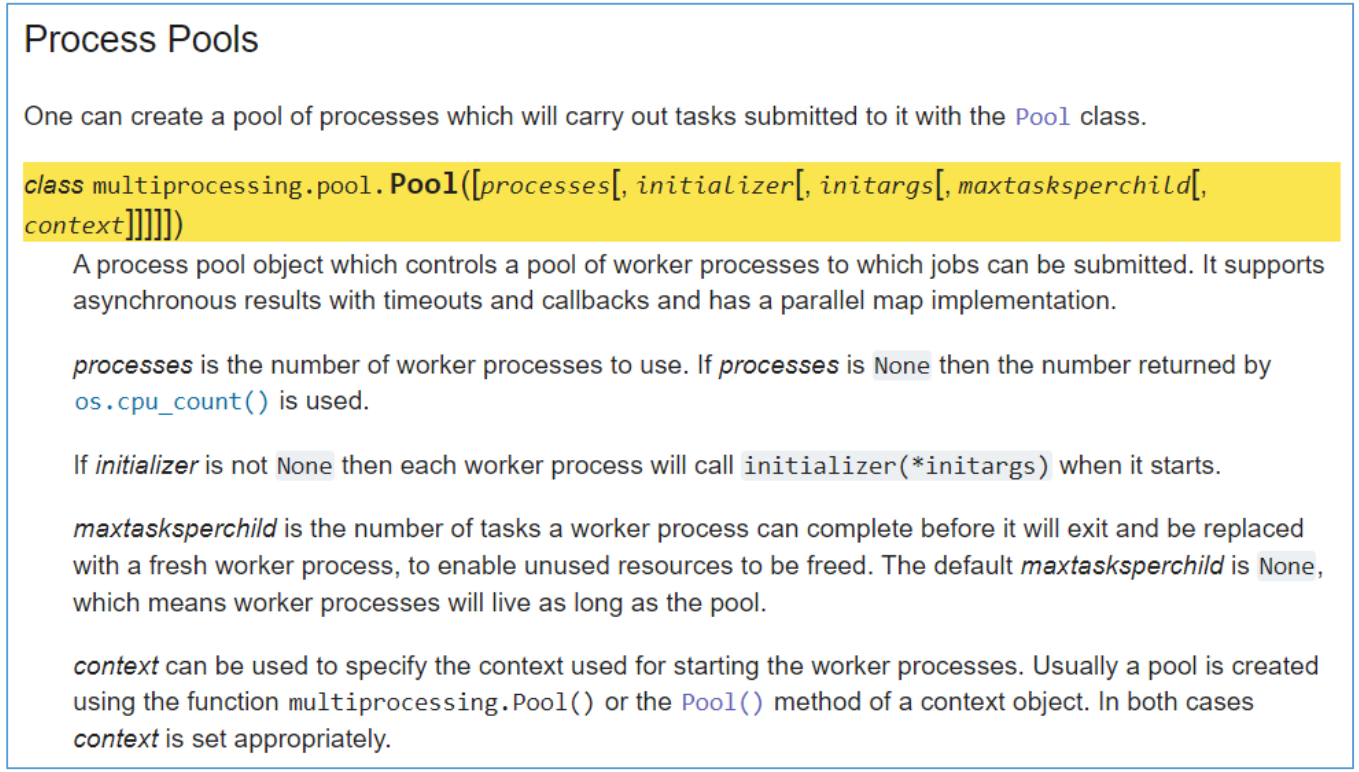
```
import multiprocessing
import time

def myProcess():
    print(f"Starting the child Process with pid: {multiprocessing.current_process().pid}")
    time.sleep(10)
    print("Child process terminating normally")

if __name__ == "__main__":
    p = multiprocessing.Process(target=myProcess, name="child")
    p.start()
    time.sleep(1)
    print("Main is to terminate the child process")
    p.terminate()
    print("Child process killed by main")
```

Output:

```
Starting the child Process with pid: 23344
Main is to terminate the child process
Child process killed by main
```

Process Pool

- The **`Pool`** class can be used to mange a fixed number of child processes for simple cases where the work to be done can be broken up and distrusted independently.
  - https://docs.python.org/3/library/multiprocessing.html#multiprocessing.pool.Pool



Process Pools

One can create a pool of processes which will carry out tasks submitted to it with the `Pool` class.

class `multiprocessing.pool.`**`Pool`**([*processes*[, *initializer*[, *initargs*[, *maxtasksperchild*[, *context*]]]]])

A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

processes is the number of worker processes to use. If *processes* is `None` then the number returned by `os.cpu_count()` is used.

If *initializer* is not `None` then each worker process will call `initializer(*initargs)` when it starts.

maxtasksperchild is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default *maxtasksperchild* is `None`, which means worker processes will live as long as the pool.

context can be used to specify the context used for starting the worker processes. Usually a pool is created using the function `multiprocessing.Pool()` or the `Pool()` method of a context object. In both cases *context* is set appropriately.

Example

- In this example, a pool of 4 processes are created and jobs are passed to them until there are no more jobs.
- map() is used here on the iterable range(10). It chops the iterable into a number of chunks which it submits to the process pool as separate tasks.

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # start 4 worker processes
        print(pool.map(f, range(10)))        # prints "[0, 1, 4,..., 81]
```

The `with` statement

- The `with` statement is used for resource management and exception handling.
- This generic `with` statement:

```
with some_resource:
    # do something...
```

- is equivalent to the following code:

```
some_resource.acquire()
try:
    # do something...
finally:
    some_resrouce.release()
```

- https://docs.python.org/3/library/threading.html#using-locks-conditions-and-semaphores-in-the-with-statement

Related topics

- We postpone discussions on some related topics and modules to near future, for example:
  - Topics:
    - Inter-process communication
    - Synchronization
    - Threads, …
  - Python library:
    - threading
    - os
    - subprocess, …

Working around GIL

- The very popular CPython implementation of Python uses GIL (global interpreter lock).
  - We will talk about GIL and its limitation when we talk about threading next in more details.

- For now, just a note that by using multiprocessing we effectively work around GIL.
  - This way we can create multiple processes that would allow us to fully leverage multiple processors power on a machine.
  - This though will cause other challenges such as dealing with inter-process communications ...

References

- Python documentation
  - https://docs.python.org/3/library/multiprocessing.html

- *The Python Standard Library*, D. Hellmann

- *Learning Concurrency in Python*
  - Note that this book seems to have errata. Use it with a grain of salt.