Spawn

This lesson discusses the alternative to forking processes, which is spawning them.

Spawn

In the previous section, we discussed how we could create processes using fork as the start method. However, fork comes with shortcomings of its own and may not be what we want. The next option available to create a process is the **spawn** method.

exec system call

In order to understand spawn, we'll examine the exec family of system calls. When you issue an **ls** or a **find** command in your terminal, the shell first forks itself and then invokes one of the variants of the **exec** family of system calls. Briefly, an exec call transforms the calling process into another. The program in the calling process is replaced with another program and is run from the entry point. **Realize the distinction between a program and a process as it matters in this context.** The program is just a set of instructions and data that is used to initialize a process. Using **exec**, the running process loads a program (instructions and data) and replaces its own program with the loaded one and starts execution.

Both fork and exec can be called independently and need not be called in succession. For instance, a process that is ending can simply call exec and start another program rather than forking itself. Similarly, a process

listening on a socket may want to fork itself to let the child process deal with a received request while it goes back to listening. But the usual way to create a new process in the Unix world is to first fork in the parent process and then exec in the child process. The child process's PID doesn't change and the parent process can wait for the child process to finish before resuming execution.

Remember, forking produces two processes, whereas exec loads an executable in the existing process's address space. The current executable image is replaced with another one loaded from an executable file.

spawn call

With the above background, it is easier to explain the "spawn" start method. Spawn is essentially a combination of fork followed by an exec (one of its variants) system call. Depending on the platform, posix_spawn system call may also be used. posix_spawn is roughly a combination of fork and exec with some optional housekeeping steps between the two calls.

This solves the problem we encountered with fork start method. The module state isn't inherited by a child process, rather it starts from scratch. A new python interpreter process is created and the child doesn't inherit any resources from the parent process other than those required to execute the specified callable target.

Note that spawning a process is slower than forking a process. When a child process is spawned anything imported at the module level in the __main__ module of the parent process gets reimported in the child. Consider the example below. If you change the start method to "spawn" on **line 14**, you'll see that the print messages in the imported modules

are printed twice. Once when the main process runs and once when the child process runs. However, running the snippet with "fork" as the start method will only have the modules imported once for the main process.

main.py

python_course.py

educative.py

```
from multiprocessing import Process
import multiprocessing
import educative

def process_task():
    print("I am child process")


if __name__ == '__main__':

    # Change the method to 'spawn' and verify
    # that the modules are reimported in the child
    # process
    multiprocessing.set_start_method('spawn')
    process = Process(target=process_task)
    process.start()
    process.join()
    print("I am parent process")
```

Example

Let's revisit our program from the previous section where a child process inherits a file descriptor from the parent process and is able to write to it. This would not be possible under the spawn start method.

Running the below code will result in an error because the value of the variable **`file_desc`** isn't copied over to the child process.

```
from multiprocessing import Process
import multiprocessing
import os

file_desc = None


def process_task():
    # write to the file in a child process
    file_desc.write("\nline written by child process with id {0}".format(os.getpid(
    file_desc.flush()


if __name__ == '__main__':
    # create a file descriptor in the parent process
    file_desc = open("sample.txt", "w")
    file_desc.write("\nline written by parent process with id {0}".format(os.getpid
    file_desc.flush()

    # changed the start method to spawn
    multiprocessing.set_start_method('spawn')

    process = Process(target=process_task)
    process.start()
    process.join()
    file_desc.close()

    # read and print the contents of the file
    file_desc = open("sample.txt", "r")
    print(file_desc.read())

    os.remove("sample.txt")
```

Yet another Example

Examine the other program from the previous section, where the change in the **`Test`** class's variable isn't reflected in the child process if the start method is specified to be spawn.

```
from multiprocessing import Process
import multiprocessing

class Test:
    value = 777


def process_task():
    print(Test.value)


if __name__ == '__main__':
    multiprocessing.set_start_method('spawn')

    # change the value of Test.value before creating
    # a new process
    Test.value = 999
    process = Process(target=process_task, name="process-1")
    process.start()
    process.join()
```

Note that if we pass any arguments to the process's callable target, the child process does receive them. This is in line with the documentation that says the child process inherits just enough resources from the parent process to execute the run method of the process object.

```
from multiprocessing import Process
import multiprocessing

global_arg = "this is a global arg"

def process_task(garg, larg):
    print(garg + " - " + larg)


if __name__ == '__main__':

    multiprocessing.set_start_method('spawn')
    local_arg = "this is a global arg"

    process = Process(target=process_task, name="process-1", args=(global_arg, loca
    process.start()
    process.join()
```

Use of Pickling

Arguments to the child process are sent using pickle. **`Pickle`** is a module used for serializing and deserializing an object. The object is serialized into a stream of bytes by the sender and then reconstructed into a different incarnation of the same object from the byte stream by the receiver. Multiprocessing uses pickle to transport data between processes. This mandates that the arguments being passed to the child process be picklable i.e. they can be serialized. Consider the below snippet, where we attempt to pass a **`Lock`** object from the **`threading`** module to a process and get an error saying the argument isn't picklable.

```
from multiprocessing import Process
from threading import Lock
import multiprocessing

def process_task(lock):
    print("child process")


if __name__ == '__main__':
    lock = Lock()
    multiprocessing.set_start_method('spawn')
    process = Process(target=process_task,
                      args=(lock,))
    process.start()
    process.join()
```

Back

Fork

Next

Forkserver

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/spawn__multiprocessing__python-concurrency-for-senior-engineering-interviews)