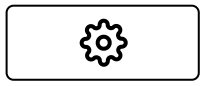

This lesson discusses the caveats to remember when forking processes.

Fork

Fork is the default method Python uses to create processes on Unix based systems.

Fork is one of the options a developer can choose to create processes. We'll need to examine the underlying operating system's fork system call in order to understand how the fork option works in Python.

Fork system call

There are two families of system calls, **fork** and **exec** that can be invoked by a process to create subprocesses on Unix based systems. A process can invoke this system call and get an *almost* clone of itself. We qualify the statement with *almost* because not everything is copied when a fork happens. The exact list of what isn't copied can be found by checking out the manpage of fork on your platform (type "man fork" in terminal). The child process gets an identical memory image so any open file descriptors are copied. However, multiple threads of a process don't get copied. Any threads running in the parent process do not exist in the child process. Additionally, fork isn't available on a Windows platform. Under the hood, Python uses `os.fork()` to create the child process.

Often times we don't want to fork processes because the newly created process (called the child process) comes with identical copies of data-structures and file descriptors of the parent process, which can be problematic.

To illustrate the differences with other methods of creating processes, consider the snippet below:

```
from multiprocessing import Process
import multiprocessing

class Test:
    value = 777


def process_task():
    print(Test.value)


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

    # 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()
```

In the snippet above, the forked child process executes the method **`process_task()`**. The main process changes the value of a static variable **`Test.value`** from 777 to 999 just before it forks the child process. The child process gets a copy of all the resources of the parent, including the static variable and prints a value of 999.

```
from multiprocessing import Process
import multiprocessing

class Test:
    value = 777


def process_task():
    print(Test.value)


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

    # 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()
```

Example

Every data structure, open file, and database connection that exists in the parent process is copied over, open and ready to use, in the child process. Consider the below program, we open a file for writing in the parent process and then subsequently write to it from the child process. The file descriptor is a global variable, a copy of which is also received by the child process. This demonstrates that the child process is an identical copy of the parent 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()

    multiprocessing.set_start_method('fork')

    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")
```

Forking in Python

When we fork, the entire Python process is duplicated in memory including the Python interpreter, code, libraries, current stack, etc. This creates a new copy of the python interpreter. The implication is that forking creates two python interpreters each with its own GIL. A single GIL is no more a blocker and we can have true multi-processing on a

multi-core system. In contrast, threads in one process share the same GIL, meaning only one thread runs at a given moment, giving only the illusion of parallelism.

Problems with fork

- To illustrate what problems fork can create, consider the below snippet. The main process created a **Lock** object and acquires it just before forking a child process. The child process receives a copy of all the global data-structures of the parent process. It attempts to acquire the lock but when the lock object was copied it was already in the locked state so the child ends up waiting on a lock object that was inherited in the locked state and will never be unlocked because there's no other thread in the process that will unlock it. The child process will hang. If you run the executable code, the execution will time out. This problem can manifest itself and be very hard to debug when you import a third-party module/library that uses threads behind the scenes. The moment when the fork happens, synchronization primitives can be copied in a state that causes the child process to hang.

Child process hangs on fork

```
lock = Lock()


def process_task():
    lock.acquire()
    print("I am child process")
    lock.release()


if __name__ == '__main__':
    multiprocessing.set_start_method('fork')
    process = Process(target=process_task)

    # acquire the lock just before starting a new process
    lock.acquire()

    process.start()

    # release the lock after starting the child process
    lock.release()

    # wait for child process to be done
    print("Parent process waiting for child process to fin
ish")
    process.join()
    print("done")
```

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

lock = Lock()


def process_task():
    lock.acquire()
    print("I am child process")
    lock.release()


if __name__ == '__main__':
    multiprocessing.set_start_method('fork')
    process = Process(target=process_task)

    # acquire the lock just before starting a new process
    lock.acquire()

    process.start()

    # release the lock after starting the child process
    lock.release()

    # wait for child process to be done
    print("Parent process waiting for child process to finish")
    process.join()
    print("done")
```

- Forking and multithreading don't mix well. Imagine a situation where a process has some threads waiting in systems calls or for IO and a different thread attempts a fork. The other threads don't get copied and it may not even make sense to copy them.
- Some libraries assume proper initialization for each process but this isn't true when a process is a result of a fork. The assumptions of the underlying libraries may not hold anymore causing the

program to fail in myriad ways.

- Portability can be an issue because of differences amongst platforms. For instance, how open file descriptors get inherited by a child process or what system calls can be executed after a fork vary across platforms.
- The fork method can't be used on a Windows platform.
- Forking is prone to security holes. For instance, if a random number generator were seeded in the main process and then forked, the child would receive the same pseudo-random number generator (PRNG) state and produce the same sequence of random numbers. This issue is now fixed and was tracked by this issue (https://bugs.python.org/issue18747). Security protocols such as SSL make use of random numbers and can get compromised. For the uninitiated, random number generation isn't truly random. The `random.seed()` is passed in a seed value and a sequence of numbers is generated thereafter. The same seed generates the same sequence of random numbers.

← Back

Process

Next →

Spawn

Mark as Completed

Report an Issue

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