Python:How os.fork() works?

https://stackoverflow.com/a/33561147

tudouyatudouya 36922 gold badges55 silver badges1111 bronze badges Thu Jul, 29 12:55

First of all, remove that `print '******...'` line. It just confuses everyone. Instead, let's try this code...

```
import os
import time

for i in range(2):
    print('I'm about to be a dad!')
    time.sleep(5)
    pid = os.fork()
    if pid == 0:
        print('I'm {}, a newborn that knows to write to the terminal!'.format(os.getpid()))
    else:
        print('I'm the dad of {}, and he knows to use the terminal!'.format(pid))
        os.waitpid(pid, 0)
```

Okay, first of all, what is 'fork'? *Fork* is a feature of modern and standard-compliant operating systems (except of M$ Windows: that joke of an OS is all but modern and standard-compliant) that allows a process (a.k.a: 'program', and that includes the Python interpreter!) to literally make an exact duplicate of itself, effectively creating a new process (another instance of the 'program'). Once that magic is done, both processes are independent. Changing anything in one of them does not affect the other one.

The process responsible for spelling out this dark and ancient incantation is known as the parent process. The soulless result of this immoral abomination towards life itself is known as the child process.

As shall be obvious to all, including those for which it isn't, you can become a member of that select group of programmers who have sold their soul by means of `os.fork()`. This function performs a fork operation, and thus results in a second process being created out of thin air.

Now, what does this function return, or more importantly, *how* does it even return? If you want not to become insane, **please** don't go and read the Linux kernel's `/kernel/fork.c` file! Once the kernel does what we know it has to do, but we don't want to accept it, `os.fork()` returns in *the two* processes! Yes, even the call stack is copied on!

So, if they are exact copies, how does one differentiate between parent and child? Simple. If the result of `os.fork()` is zero, then you're working in the child. Otherwise, you're working in the parent, and the return value is the PID (Process IDentifier) of the child. Anyway, the child can get its own PID from `os.getpid()`, no?

Now, taking this into account, and the fact that doing `fork()` inside a loop is the recipe for mess, this is what happens. Let's call the original process the 'master' process...

- Master: `i = 0`, forks into child-#1-of-master
    - Child-#1-of-master: `i = 1` forks into child-#1-of-child-#1-of-master
    - Child-#1-of-child-#1-of-master: `for` loop over, exits
    - Child-#1-of-master: `for` loop over, exits
- Master: `i = 1`, forks into child-#2-of-master
    - Child-#2-of-master: `i = 1` forks into child-#1-of-child-#2-of-master
    - Child-#1-of-child-#2-of-master: `for` loop over, exits
    - Child-#2-of-master: `for` loop over, exits
- Master: `for` loop over, exits

As you can see, there are a total of 6 parent/child prints coming from 4 unique processes, resulting in 6 lines of output, something like...

```
I'm the dad of 12120, and he knows to use the terminal!

I'm 12120, a newborn that knows to write to the terminal!

I'm the dad of 12121, and he knows to use the terminal!

I'm 12121, a newborn that knows to write to the terminal!

I'm the dad of 12122, and he knows to use the terminal!

I'm 12122, a newborn that knows to write to the terminal!
```

But that's just arbitrary, it could have output this out instead...

```
I'm 12120, a newborn that knows to write to the terminal!

I'm the dad of 12120, and he knows to use the terminal!

I'm 12121, a newborn that knows to write to the terminal!

I'm the dad of 12121, and he knows to use the terminal!

I'm 12122, a newborn that knows to write to the terminal!

I'm the dad of 12122, and he knows to use the terminal!
```

Or anything other than that. The OS (and your motherboard's funky clocks) is solely responsible for the order in which processes get timeslices, so go blame on Torvalds (and expect no self-steem when back) if you dislike how the kernel manages to organize your processes ;).

I hope this has led some light on you!