Parallel System Tools

SMJE4383

Assoc Prof Ir Dr Zool Hilmi Ismail

Parallel System Tools

- Most computers spend a lot of time doing nothing.
- If you start a system monitor tool and watch the CPU utilization, you'll see what I mean—it's rare to see one hit 100 percent, even when you are running multiple programs.*
- There are just too many delays built into software: disk accesses, network traffic, database queries, waiting for users to click a button, and so on. In fact, the majority of a modern CPU's capacity is often spent in an idle state; faster chips help speed up performance demand peaks, but much of their power can go largely unused.

Forking Processes

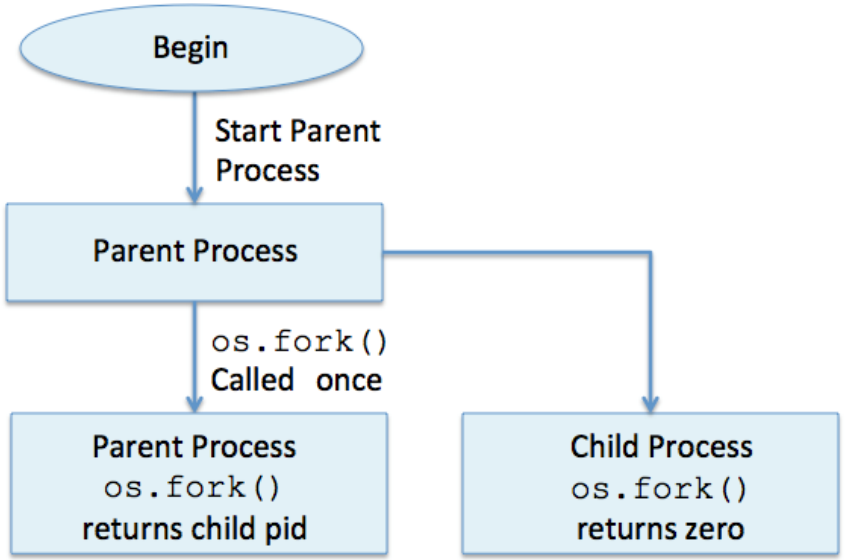
- A traditional way to structure parallel tasks, and they are a fundamental part of the Unix tool set.
- Forking is a straightforward way to start an independent program, whether it is different from the calling program or not.
- Forking is based on the notion of *copying* programs:
  - when a program calls the fork routine, the operating system makes a new copy of that program and its process in memory and starts running that copy in parallel with the original.

Forking Process

- After a fork operation, the **original** copy of the program is called the ***parent*** process, and the copy created by os.fork is called the *child* process.

Example 5-1. PP4E\System\Processes\fork1.py

```
"forks child processes until you type 'q'"

import os

def child():
    print('Hello from child',  os.getpid())
    os._exit(0)  # else goes back to parent loop

def parent():
    while True:
        newpid = os.fork()
        if newpid == 0:
            child()
        else:
            print('Hello from parent', os.getpid(), newpid)
        if input() == 'q': break

parent()
```



Output on terminal

```
[C:\...\PP4E\System\Processes]$ python fork1.py
Hello from parent 7296 7920
Hello from child 7920

Hello from parent 7296 3988
Hello from child 3988

Hello from parent 7296 6796
Hello from child 6796
q
```

Example 5-2. PP4E\System\Processes\fork-count.py

```
"""
fork basics: start 5 copies of this program running in parallel with
the original; each copy counts up to 5 on the same stdout stream--forks
copy process memory, including file descriptors; fork doesn't currently
work on Windows without Cygwin: use os.spawnv or multiprocessing on
Windows instead; spawnv is roughly like a fork+exec combination;
"""

import os, time

def counter(count):                                     # run in new process
    for i in range(count):
        time.sleep(1)                                   # simulate real work
        print('[%s] => %s' % (os.getpid(), i))

for i in range(5):
    pid = os.fork()
    if pid != 0:
        print('Process %d spawned' % pid)               # in parent: continue
    else:
        counter(5)                                      # else in child/new process
        os._exit(0)                                     # run function and exit

print('Main process exiting.')                          # parent need not wait
```

Output on terminal

```
[C:\...\PP4E\System\Processes]$ python fork-count.py
Process 4556 spawned
Process 3724 spawned
Process 6360 spawned
Process 6476 spawned
Process 6684 spawned
Main process exiting.
[4556] => 0
[3724] => 0
[6360] => 0
[6476] => 0
[6684] => 0
[4556] => 1
[3724] => 1
[6360] => 1
[6476] => 1
[6684] => 1
[4556] => 2
[3724] => 2
[6360] => 2
[6476] => 2
[6684] => 2
...more output omitted...
```

Combination of fork/exec

- It forks new processes until we type *q* again, but child processes run a **brand-new program** instead of calling a function in the same file.

Example 5-3. PP4E\System\Processes\fork-exec.py

```
"starts programs until you type 'q'"

import os

parm = 0
while True:
    parm += 1
    pid = os.fork()
    if pid == 0:                                                # copy process
        os.execlp('python', 'python', 'child.py', str(parm)) # overlay program
        assert False, 'error starting program'                  # shouldn't return
    else:
        print('Child is', pid)
        if input() == 'q': break
```

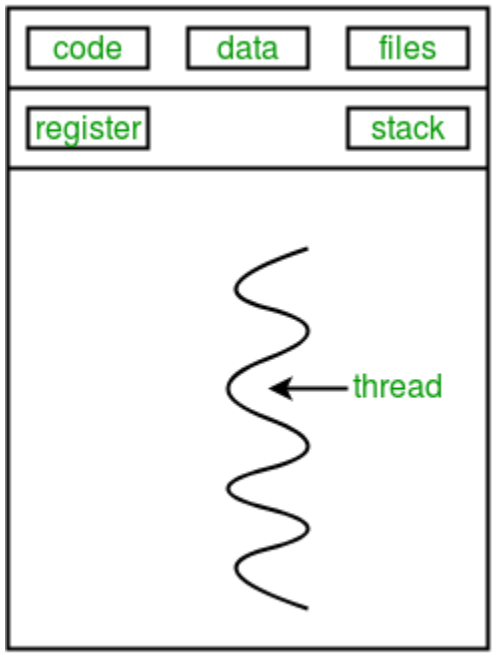
Output on Terminal

```
[C:\...\PP4E\System\Processes]$ python fork-exec.py
Child is 4556
Hello from child 4556 1

Child is 5920
Hello from child 5920 2

Child is 316
Hello from child 316 3
q
```

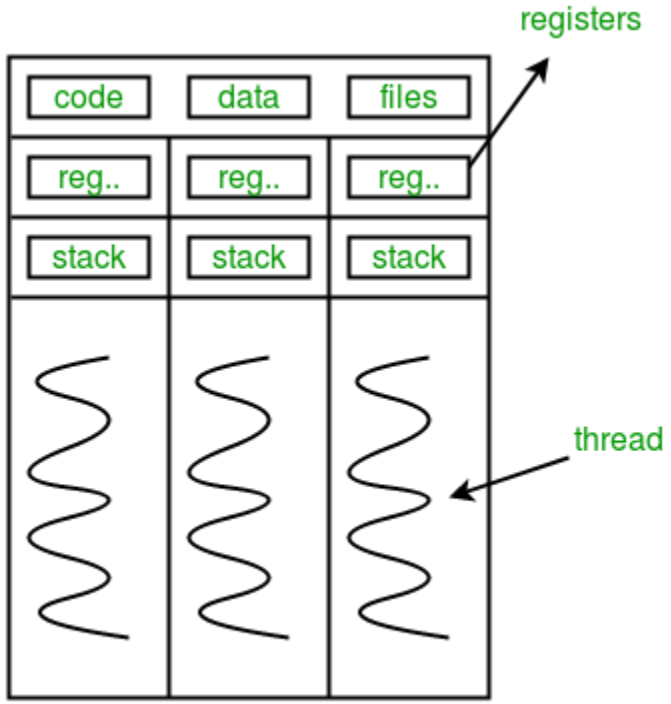
Threads in parallel processing

- Another way to **start activities running** at the **same time.**
- They run a call to a function (or any other type of callable object) in parallel with the rest of the program.
- Sometimes called “lightweight processes,” because they run in parallel like forked processes, but all of them run within the same single process.
- While processes are commonly used to start independent programs, threads are commonly used for tasks such as nonblocking input calls and long-running tasks in a GUI.
- They also provide a natural model for algorithms that can be expressed as independently running tasks.

Single vs Multi-Thread Process



single-threaded process



multithreaded process

Advantages

- *Performance*
  Because all threads run within the same process, they don't generally incur a big startup cost to copy the process itself. The costs of both copying forked processes and running threads can vary per platform, but threads are usually considered less expensive in terms of performance overhead.

- *Simplicity*
  To many observers, threads can be noticeably simpler to program, too, especially when some of the more complex aspects of processes enter the picture (e.g., process exits, communication schemes).

Advantages

- *Shared global memory*
  On a related note, because threads run in a single process, every thread shares the same global memory space of the process. This provides a natural and easy way for threads to communicate—by fetching and setting names or objects accessible to all the threads.

- *Portability*
  Perhaps most important is the fact that threads are more portable than forked processes. At this writing, os.fork is not supported by the standard version of Python on Windows, but threads are.

The _threads Module

Example 5-5. PP4E\System\Threads\thread1.py

```
"spawn threads until you type 'q'"

import _thread

def child(tid):
    print('Hello from thread', tid)

def parent():
    i = 0
    while True:
        i += 1
        _thread.start_new_thread(child, (i,))
        if input() == 'q': break

parent()
```

Output on Terminal

```
C:\...\PP4E\System\Threads> python thread1.py
Hello from thread 1

Hello from thread 2

Hello from thread 3

Hello from thread 4
q
```

Running Multiple Threads

Example 5-6. PP4E\System\Threads\thread-count.py

```
"""
thread basics: start 5 copies of a function running in parallel;

uses time.sleep so that the main thread doesn't die too early--
this kills all other threads on some platforms; stdout is shared:
thread outputs may be intermixed in this version arbitrarily.
"""

import _thread as thread, time

def counter(myId, count):                       # function run in threads
    for i in range(count):
        time.sleep(1)                           # simulate real work
        print('[%s] => %s' % (myId, i))

for i in range(5):                              # spawn 5 threads
    thread.start_new_thread(counter, (i, 5))    # each thread loops 5 times

time.sleep(6)
print('Main thread exiting.')                   # don't exit too early
```

```
C:\...\PP4E\System\Threads> python thread-count.py
[1] => 0
[1] => 0
[0] => 0
[1] => 0
[0] => 0
[2] => 0
[3] => 0

[3] => 0

[1] => 1
[3] => 1
[3] => 1
[0] => 1[2] => 1
[3] => 1
[0] => 1[2] => 1
[4] => 1

[1] => 2
[3] => 2[4] => 2
[3] => 2[4] => 2
[0] => 2
[3] => 2[4] => 2
[0] => 2
[2] => 2
[3] => 2[4] => 2
[0] => 2
[2] => 2

...more output omitted...
Main thread exiting.
```

How to synchronize the access to shared objects and names?