Python System Programming

Operating System Practice

João Vicente Ferreira Lima

Universidade Federal de Santa Maria
jvlima@inf.ufsm.br
http://www.inf.ufsm.br/~jvlima

2021/2

Outline

Outline

System calls

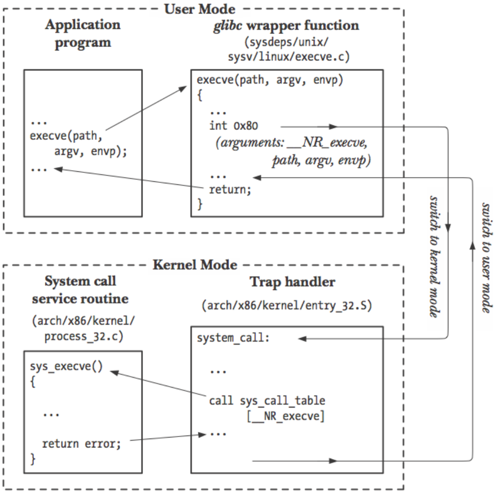
- A system call changes the processor state from user mode to kernel mode.
- The number of system calls is fixed, and each system call has an unique number.
- Each system call may have a set of arguments that has information to transfer to be transfered from user space to kernel space.

The system call steps are:

1. Calls a wrapper function in the C library.
2. The wrapper function copies the arguments to these registers.
3. The wrapper function copies the system call number into a specific CPU register (%`eax`).
4. The wrapper function executes a *trap* instruction (`int 0x80`).
5. The kernel invokes its *system_call()*.
6. If the return value indicates an error, the wrapper function sets the global variable *errno*.

System calls

The Figure below illustrates the above steps for the *execve()* system call.



Outline

Python system modules

Most system-level calls are shipped in just two modules: `sys` and `os`. `sys` exports components related to the Python *interpreter* and `os` contains variables and functions that map to the operating system. Other related modules are:

- **glob** filename expansion.
- **socket** network connections and IPC.
- **threading, queue** running and synchronizing concurrent threads.
- **time, timeit** system time details.
- **subprocess, multiprocessing** launching and controlling parallel processes.
- **signal, select, shutil, tempfile,** ***etc*** system-related tasks

Outline

Time

```python
import time
from datetime import date

today = date.today()
print( "Today is: ", today)

timestamp = time.time()
print( "Timestamp now: ", timestamp )

time_today = date.fromtimestamp(timestamp)
print( "From timestamp:", time_today )
```

```
Today is:  2019-05-13
Timestamp now:  1557773061.429696
From timestamp: 2019-05-13
```

Time

```python
import time
from datetime import date, datetime

t1 = time.time()
print("Sleeping 2 seconds...")
time.sleep(2)
t2 = time.time()

delta = datetime.fromtimestamp(t2-t1)
print("Seconds:", delta.second,
    "Microseconds:", delta.microsecond)
```

```
Sleeping 2 seconds...
Seconds: 2 Microseconds: 7270
```

Outline

Introduction

The `os` module contains basic functions to run shell commands from within Python scripts. Two `os` functions allow scripts to run any command line:

- `os.system` runs shell command
- `os.popen` runs a shell command and connects to its input or output.
- `subprocess` intends to replace `os.system` and `os.spawn*`.

Current Working Directory

The CWD is the directory you where in when you typed this command, not where the script resides. On the other hand, Python automatically adds the identity of the script's home directory to the front of the module search path in order to import any other files.

```
import os
import sys

print('My os.getcwd: ' + os.getcwd())
print('My sys.path : ' + str(sys.path))
```

```
My os.getcwd: /Users/jvlima/pso/lectures
My sys.path : ['', '/usr/local/Cellar/python3/3.5.2_3/Frameworks/Python.framework/Ver
```

Shell environment variables

Shell variables are available as `os.environ`, a Python dictionary-like object with one entry per variable in the shell.

```
import os

print(os.environ.keys())
print(list(os.environ.keys()))
print('Variable TMPDIR is: ' + os.environ['TMPDIR'])
```

```
KeysView(environ({'XPC_FLAGS': '0x0', 'PWD': '/Users/jvlima/pso/lectures', 'SCRATCH':
['XPC_FLAGS', 'PWD', 'SCRATCH', 'TERM', 'TERM_SESSION_ID', '_', 'SECURITYSESSIONID',
Variable TMPDIR is: /var/folders/m6/d0jhl9fs19j82w4fck9qxs7m0000gn/T/
```

Shell environment variables

To change or create variables, we can just assign a value like normal dictionaries. However, this new value is only visible to the enclosing shell environment.

```python
#!/usr/bin/env python3
import os

print('Old value: ' + os.environ['USER'])
os.environ['USER'] = 'thing'
print('New value: ' + os.environ['USER'])
```

```
Old value: jvlima
New value: thing
```

Outline

Running a shell command

The `os.system` call lets Python scripts run any sort of command line program.

```
import os

ret = os.system('ls ..')
print('Return value: ' + str(ret))
```

```
LICENSE
README.org
assignments
lectures
scripts
Return value: 0
```

The `os.system` returns the exit status, and redirects the command's output in the session or standard output.

Communicating with shell commands

os.popen connects to the standard output or input of the command. If we pass a w mode flag to popen, we connect to the command's input stream.

```python
import os

text = os.popen('ls ..').read()
print(text)

lines = os.popen('ls ..').readlines()
print(lines)
```

```
LICENSE
README.org
assignments
lectures
scripts

['LICENSE\n', 'README.org\n', 'assignments\n', 'lectures\n', 'scripts\n']
```

Outline

subprocess module

As we mentioned before, the `subprocess` module intends to replate several older modules and functions such as `os.system` and `os.spawn*`.
Running a shell command can be done using `run()` (recommended) or `call()`.

```
import subprocess

subprocess.run('date')
subprocess.run(['ls', '..'])
subprocess.run('hello.py', shell=True)
```

```
Wed Nov 16 14:39:46 BRST 2016
LICENSE
README.org
assignments
lectures
scripts
```

Two things must be noted here:

1. The second command received a list in which the first element is the command and the second its arguments.
2. The `shell=True` argument. On Unix-like platforms, when `shell` is `False`, the program command line is run directly by `os.execvp`. If this

argument is `True`, the command is run through a shell instead.

Outline

Forking processes

```python
import os

def child():
  print('Hello from child', os.getpid())
  os._exit(0)

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

Forking processes

```python
import os, time

def counter(count):
  for i in range(count):
    time.sleep(1)
    print('[%s] => %s' % (os.getpid(), i))

for i in range(5):
  pid = os.fork()
  if pid != 0:
    print('Process %s spawned' % pid)
  else:
    counter(5)
    os._exit(0)

print('Main process exiting.')
```

Fork

```python
import os

parm = 0
while True:
  parm += 1
  pid = os.fork()
  if pid == 0:
    os.execlp('python', 'python', 'child.py', str(parm))
    assert False, 'error starting program'
  else:
    print('Child is', pid)
    if input() == 'q':
      break
```

Exec

```
import os
import sys

print('Hello from child', os.getpid(), sys.argv[1])
```

A list of `os.exec` variants are:

```
os.execv(program, commandlinesequence)
os.execl(program, cmdarg1, cmdarg2, ... cmdargN)
os.execlp
os.execvp
os.execvpe
os.execlpe
```

Outline

Threading

The `threading` module uses the `_thread` module (lower-level interface) to implement a higher-level interface based on objects and classes.

```
import threading
```

Threading

This example demostrates a threading class (`MyThead`):

```
import threading

class MyThread(threading.Thread):
  def __init__(self, myId, count, mutex):
    self.myId = myId
    self.count = count
    self.mutex = mutex
    threading.Thread.__init__(self)

  def run(self):
    for i in range(self.count):
      with self.mutex:
        print('[%s] => %s' % (self.myId, i))
```

```
stdoutmutex = threading.Lock()
threads = []

for i in range(10):
  thread = MyThread(i, 10, stdoutmutex)
  thread.start()
  threads.append(thread)

for thread in threads:
  thread.join()

print('Main thread exiting.')
```

Threading

```
[0] => 0
[0] => 1
[0] => 2
[0] => 3
[0] => 4
[0] => 5
[0] => 6
[0] => 7
[0] => 8
[0] => 9
[1] => 0
[1] => 1
[1] => 2
[1] => 3
[1] => 4
[1] => 5
[1] => 6
[1] => 7
```

Threading

Your thread class do not necessarily have to subclass `Thread`. The thread's target in `threading` may be any type of *callable object*.

```
import threading

class Power:
    def __init__(self, i):
        self.i = i
    def action(self):
        print(self.i ** 32)

obj = Power(2)
threading.Thread(target=obj.action).start()
```

4294967296

Threading

Global variables can require coordination if concurrent updates are possible, such as:

```
import threading
import time

count = 0

def adder():
    global count
    count = count + 1
    time.sleep(0.5)
    count = count + 1
```

```python
threads = []
for i in range(100):
    thread = threading.Thread(target=adder, args=())
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()
print(count)
```

200

Threading

This code clearly has a race condition on the update of `count` global variable. To avoid this race, we need to add a lock to synchronize the updates:

```
import threading
import time

count = 0

def adder(addlock):
    global count
    with addlock:
        count = count + 1
    time.sleep(0.5)
    with addlock:
        count = count + 1
```

Threading

```
addlock = threading.Lock()
threads = []
for i in range(100):
    thread = threading.Thread(target=adder, args=(addlock,))
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()
print(count)
```

200

Queue

The queue module provides a standard queue data structure (FIFO), in which items are added on one end and removed from the other. The queue object is automatically controlled with thread lock acquire and release operations.

In this example, the program runs two producers and two consumers (five threads including the main one). Note that consumers threads are set to be *daemon* threads. The entire program exits when only deamon threads are left. Producer threads end with a *join* at the end.

```
import threading
import queue
import time

nconsumers = 2
nproducers = 2
nmessages = 4

safeprint = threading.Lock()
dataQueue = queue.Queue()
```

Producer/consumer

```
def producer(idnum):
    for msg in range(nmessages):
        time.sleep(idnum)
        dataQueue.put('[producer id=%d, count=%d]' %
                      (idnum, msg))

def consumer(idnum):
    while True:
        time.sleep(0.1)
        try:
            data = dataQueue.get(block=False)
        except queue.Empty:
            pass
        else:
            with safeprint:
                print('consumer', idnum,
                      'got =>', data)
```

Producer/consumer

```python
if __name__ == '__main__':
    for i in range(nconsumers):
        thread = threading.Thread(target=consumer,
                                  args=(i,))
        thread.daemon = True # else cannot exit
        thread.start()

    threads = []
    for i in range(nproducers):
        thread = threading.Thread(target=producer,
                                  args=(i,))
        thread.start()
        threads.append(thread)

    for thread in threads:
        thread.join()
```

Producer/consumer

```
consumer 0 got => [producer id=0, count=0]
consumer 1 got => [producer id=0, count=1]
consumer 0 got => [producer id=0, count=2]
consumer 1 got => [producer id=0, count=3]
consumer 0 got => [producer id=1, count=0]
consumer 0 got => [producer id=1, count=1]
consumer 0 got => [producer id=1, count=2]
```

Other synchronization objects

`threading.RLock` reentrant lock.

`threading.Condition(lock=None)` condition variable.

`threading.Semaphore(value=1)` semaphore.

`threading.Event` one thread signals an event and other threads wait for it.

Outline

Pipes

Pipes are implemented by the operating system and made available in the Python standard library.
Pipes are unidirectional channels.

Anonymous and named pipes

There are *anonymous* and *named* pipes. Named pipes (or fifos) are external files. By contrast, anonymous pipes exist only within processes and are tipically used in conjunction with process *forks*.

Anonymous Pipes

This example forks itself and creates a pipe. The `os.pipe` call returns a tuple of two file descriptors, representing the input and output sides of the pipe.

```
import os, time

def child(pipeout):
    zzz = 0
    while True:
        # make parent wait
        time.sleep(zzz)
        # pipes are binary bytes
        msg = ('Spam %03d' % zzz).encode()
        # send to parent
        os.write(pipeout, msg)
        # goto 0 after 4
        zzz = (zzz+1) % 5
```

Anonymous Pipes

```python
def parent():
    # make 2-ended pipe
    pipein, pipeout = os.pipe()
    # copy this process
    if os.fork() == 0:
        child(pipeout)
    else:
        # in parent, listen to pipe
        while True:
            # blocks until data sent
            line = os.read(pipein, 32)
            print('Parent %d got [%s] at %s' %
                  (os.getpid(), line, time.time()))

parent()
```

Anonymous Pipes

```
Parent 79486 got [b'Spam 000'] at 1479696342.063272
Parent 79486 got [b'Spam 001'] at 1479696343.063545
Parent 79486 got [b'Spam 002'] at 1479696345.065001
Parent 79486 got [b'Spam 003'] at 1479696348.066442
Parent 79486 got [b'Spam 004'] at 1479696352.067562
Parent 79486 got [b'Spam 000'] at 1479696352.067668
Parent 79486 got [b'Spam 001'] at 1479696353.068904
Parent 79486 got [b'Spam 002'] at 1479696355.070184
Parent 79486 got [b'Spam 003'] at 1479696358.071641
Parent 79486 got [b'Spam 004Spam 000'] at 1479696362.073227
Parent 79486 got [b'Spam 001'] at 1479696363.074522
Parent 79486 got [b'Spam 002'] at 1479696365.075826
Parent 79486 got [b'Spam 003'] at 1479696368.077408
Parent 79486 got [b'Spam 004Spam 000'] at 1479696372.078836
Parent 79486 got [b'Spam 001'] at 1479696373.079927
```

Named pipes (Fifos)

Named pipes

Named pipes are external files to any particular program. Once a named pipe file is create, clients open it by name and read and write data using normal file operations.

In this example, a named pipe is created with the `os.mkfifo` call. Because the fifo exists independently of both parent and child, there is no reason to fork here.

Named pipes (Fifos)

```python
import os, time, sys
fifoname = '/tmp/pipefifo'

def child():
    # open fifo pipe file as fd
    pipeout = os.open(fifoname, os.O_WRONLY)
    zzz = 0
    while True:
        time.sleep(zzz)
        # binary as opened here
        msg = ('Spam %03d\n' % zzz).encode()
        os.write(pipeout, msg)
        zzz = (zzz+1) % 5
```

Named pipes (fifos)

```
def parent():
    # open fifo as text file object
    pipein = open(fifoname, 'r')
    while True:
        # blocks until data sent
        line = pipein.readline()[:-1]
        print('Parent %d got "%s" at %s' %
                (os.getpid(), line, time.time()))

if __name__ == '__main__':
    if not os.path.exists(fifoname):
        os.mkfifo(fifoname)
    if len(sys.argv) == 1:
        parent()
    else:
        child()
```

Named pipes (Fifos)

Execute the parent typing:

```
python pipefifo.py
Parent 80553 got "Spam 000" at 1479700554.835515
Parent 80553 got "Spam 001" at 1479700555.840781
Parent 80553 got "Spam 002" at 1479700557.845987
Parent 80553 got "Spam 003" at 1479700560.84998
Parent 80553 got "Spam 004" at 1479700564.855003
Parent 80553 got "Spam 000" at 1479700564.855088
Parent 80553 got "Spam 001" at 1479700565.859777
.....
```

Execute the child:

```
python pipefifo.py -child
```

Outline

Other os module exports

`os.environ` manipulates environment variables.

`os.fork` spawns a new child process.

`os.pipe` communicates between programs.

`os.execlp` starts new programs.

`os.spawnv` starts new programs with lower-level control.

`os.open` opens a low-level descriptor file.

`os.mkdir` creates a new directory.

`os.mkfifo` creates a new named pipe.

`os.stat` fetches low-level file information.

`os.remove` deletes a file by its pathname.

`os.walk` applies a function or loop body to all parts of an entire directory tree.