ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ PYTHON

(Python Beginning)

Понятие процесс (тяжеловесный процесс) связано с исполняемой под управление ОС программой, это данные, команды и связанные с программой ресурсы (созданные, организованные и поддерживаемые ОС). Поток (легковесный поток) - это частью процесса…
(просто о сложном)

Урок 11. (Lesson 11) Системные инструменты

Потоки выполнения представляют собой один из механизмов одновременного исполнения нескольких операций. При этом создаваемые потоки являются «дочерними» по отношению к породившей их программе, но самостоятельными программами, то есть тяжеловесными процессами, не являются.

Порождение нового процесса – ветвление может занимать порядка 2000 машинных операций, переключение между процессами тоже «обходиться» системе где-то 200 машинных операций. Однако механизм параллельных процесса позволяет запускать на исполнение несколько параллельно работающих программ, в то время как параллельные потоки являются частью одного процесса. Они разделяют адресное пространство породившего их процесса, конкурируют за доступ к его глобальным переменным.

К сожалению, в `Python` выделяются переносимые и непереносимые механизмы ветвления процессов. В рамках нашего курса непереносимые механизмы, основанные на использовании вызова `os.fork()`, не рассматриваются.

Модуль subprocess

Модуль `subprocess` предоставляет переносимый механизм запуска параллельных процессов. Данный модуль был включен в состав языка `Python` начиная с версии 2.4 в целях замены таких средств модуля `os`, как `os.popen()`, `os.spawn()` и `os.system()`, и для замены других несовместимых методов. Это не значит, что от использования выше озвученных методов необходимо совсем отказаться, просто для них предложено альтернативное и переносимое решение.

Рассмотрим метод `subprocess.call`, предназначенный для блокирующего вызова другой программы. Фактически, он является альтернативой для `os.spawn()`. Данный метод имеет все необходимые средства для задания среды исполнения порождаемого процесса. Формат вызова:

```
call(args, *, stdin=None, stdout=None, stderr=None, shell=False,
     cwd=None, timeout=None)
```

Параметр args позволяет задавать список аргументов для вызываемого процесса. Кроме этого, как видно из формата объявления метода, имеются формальные параметры, позволяющие определить потоки ввода-вывода.

Рассмотрим следующий пример.

```
import subprocess

# Для Windows
subprocess.call("notepad.exe")

# Для linux
subprocess.call("gedit")
```

Данный сценарий всего лишь запускает в порожденном процессе текстовый редактор, ожидает его завершения. В этом примере не реализованы средства передачи параметров запускаемой программе, ведь это не всегда и нужно. Также вы можете запустить на исполнение *Windows* команду *shutdown*, позволяющую выключать компьютер, имя которого передано в качестве параметра. В случае, когда необходимо выключить компьютер, на котором исполняется сценарий, имя компьютера указывать не нужно. Функция *shutdown* поддерживает следующие ключи:

/L – выполнить немедленный выход текущего польхователя;

/f – принудительно закрыть все работающие приложения без оповещения других пользователей, работающих в системе;

/t – задать величину «задержки», то есть величину временного интервала, через который будет выполнена процедура завершения работы системы.

Таким образом, команда `'shutdown /s /t 60 /L'` приведет к завершению работы системы через 60 секунд. Но данная команда состоит не только из имени команды, но и из набора параметров – аргументов, которые нужно правильно передать методу *call()*. Получается примерно следующая инструкция.

```
call(('shutdown', '/s', '/t 60', '/L'), shell=True)
```

Также можно указать имя или IP-адрес машины, которую необходимо выключить.

```
'shutdown /s /t 60 /f /L /m \\192.168.xxx.yyy'
'shutdown /s /t 60 /f /L /m \\user-5412'
```

Замечание. *Результат работы данной инструкции можно проверить дома, на занятии для этого просто нет времени.*

Рассмотрим следующий пример.

```
from subprocess import call
import os
```

```
res = call(('dir', r'c:\Windows\System32'), shell=True)
print(res)
```

Вызывается системная программа *dir*, используется для вывода списка файлов и каталогов в указанном в качестве параметра каталоге. Ввиду того, что время работы порожденного процесса очень мало, вы даже не успеете увидеть выводимый на экран результат. Поэтому воспользуемся параметром *stdout*, позволяющим определить файловый поток для вывода порожденного процесса. Получается следующий код.

```
from subprocess import call
import os

fp = open("result.txt", "w")
res = call(('dir', r'c:\Windows\System32'), shell=True, stdout=fp)
fp.close()
print(res)
```

Посмотрите созданный в текущем по отношению к вашему сценарию каталоге файл. Первые строчки, связанные с выводом сообщения в формате кириллицы, содержат «нечто». Выберете необходимые параметры для метода *open()*, обеспечивающие правильную кодировку кириллицы.

На платформе Linux Вам доступен следующий код:

```
from subprocess import call
import os
res = call(('cat', '/etc/motd'))
```

Как Вы помните, сервисная программа *cat* предназначена для работы с файлами, она позволяет выводить их содержимое в стандартный поток. На платформе *MS Windows* она не поддерживается.

Рассмотрим еще два примера использования метода *call*.

Пример 1.

```
import subprocess

# Для Windows
subprocess.call("notepad.exe")

# Для linux
subprocess.call("gedit")
```

Пример 2.

```
import subprocess
code = subprocess.call(["ping", "www.yahoo.com"])
```

Сценарий первого примера просто запускает текстовый редактор в порожденном процессе и ожидает завершения его работы. Сценарий второго

примера запускает в порожденном процессе системную программу – функцию *ping*. С *ping* связан свой файл стандартного вывода, который при необходимости можно перенаправить в файл.

Вывод – метод *subprocess.call()* обеспечивает блокирующий вызов указанной программы в порожденном процессе (используется системный вызов *wait()*). После завершения работы программы данный процесс будет разрушен и метод *call()* возвращает код завершения процесса. Ни о какой параллельной работе процессов при этом речи не идет – родительский процесс переводится в режим ожидания.

Модуль *subprocess* содержит набор средств для запуска новых процессов, работающих параллельно с родительским по отношению к ним потоком.

Рассмотрим класс *subprocess.Popen*. Данный класс предоставляет средства для запуска дочерней программы в новом процессе. При этом дочерний процесс исполняется параллельно относительно к родительскому процессу. Важно, что он реализует ***не блокирующий вызов*** нового процесса. При этом родительский процесс, если не указать явно обратное, не ожидает завершения дочернего процесса.

Рассмотрим следующий пример.

```
import subprocess
subprocess.Popen('C:\\Windows\\System32\\calc.exe')
```

Сценарий вызывает в новом процессе встроенный калькулятор и, не дожидаясь завершения работы дочернего процесса, завершает свою работы. Чтобы перевести родительский процесс в режим ожидания, необходимо воспользоваться вызовом метода *wait()*.

```
import subprocess
cal = subprocess.Popen('C:\\Windows\\System32\\calc.exe')
cal.wait()
```

Также имеется возможность для чтения результата, возвращаемого дочерним процессом, для этого используется метод *poll()*.

```
import subprocess
cal = subprocess.Popen('C:\\Windows\\System32\\calc.exe')
print(cal.poll())  # При запущенном калькуляторе
cal.wait()         # Перевод родительского процесса в ожидание
print(cal.poll())  # После завершения работы калькулятора
```

При этом у родительского процесса есть средства для принудительного завершения дочернего процесса. Метод Popen.kill() 'убивает' дочерний процесс.

```
import subprocess
cal = subprocess.Popen('C:\\Windows\\System32\\calc.exe')
print(cal.poll())
```

```
choice = input("Kill? ")
if choice == 'y' or choice == 'Y':
    cal.kill()
print(cal.poll())
```

В явном использовании метода *wait()* нет необходимости, так как родительский процесс и так находится в режиме ожидания пользовательского ввода.

Рассмотрим методы класса subprocess.Popen.

Метод *Popen.wait(timeout=None)* – переводит родительский процесс в режим ожидания завершения дочернего процесса. Если в течение интервала времени *timeout* дочерний процесс не завершится, будет вызвано исключение *TimeoutExpired*. Данное исключение может быть перехвачено, после чего родительский процесс ещё раз может вызвать метод *wait()*.

Метод *Popen.communicate(input=None, timeout=None)* – обеспечивает взаимодействие с дочерним процессом: посылает данные, содержащиеся в *input* в *stdin* дочернего процесса, ожидает завершения его работы, возвращает кортеж данных потока вывода и ошибок. При этом в *Popen* необходимо задать значение *PIPE* для *stdin* (если вы хотите посылать в *stdin*), *stdout*, *stderr* (если необходимо прочитать вывод дочернего процесса). Если в течение *timeout* процесс не завершился, будет вызвано исключение *TimeoutExpired* (которое можно перехватить, после чего сделать ещё раз *communicate*, либо «убить» дочерний процесс). Прочитанные данные буферизируются в память, поэтому не стоит применять этот метод в случае «огромных» выходных данных.

Кроме того, метод *Popen.terminate()* позволят остановить дочерний процесс, а метод *Popen.send_signal()* – посылает дочернему процессу сигнал.

Рассмотрим пример, позволяющий обрабатывать данные, возвращаемые дочерним процессом.

```
import subprocess

args = ["ping", "www.yahoo.com"]
process = subprocess.Popen(args, stdout=subprocess.PIPE)
data = process.communicate()
print(data)
```

В данном примере родительский процесс взаимодействует с дочерним посредством вызова метода process.communicate() . Однако данные выводятся в unicode. Измените данный пример таким образом, чтобы результат был «читаем».

Хотя бы так.