CS2506 Operating Systems II Lab 4 14, 16/03/2017
Application Programming Interface (API)

Laboratory Goals / Objectives

Students will learn about Python os module by testing some of the functions one by one, or as part of programs.

.

1. Introduction

An **application programming interface** (**API**), such as Python *os module*, enables applications' interaction with computer system resources. We can consider os API as an abstraction that describes an interface for the interaction with a set of platform functions. The software that provides the functions described by an API is said to be an *implementation* of the API.

The os module provides a portable way of using operating system dependent functionality. This module has numerous tools to deal with filenames, paths, directories, processes, however not all implemented on all operating systems.

1.1 Process parameters

This set of functions and data items provide information and operate on the current process and user context data. Following are three examples of functions that give access to context data.

*os.**environ***, for example, is a mapping object representing the string environment. environ['HOME'] is the pathname of your home directory (on some platforms). This mapping is captured the first time the os module is imported, typically during Python startup. Changes to the environment made after this time are not reflected in os.environ, except for changes made by modifying os.environ directly.

os.**getenv**(*varname*[, *value*]) returns the value of the environment variable *varname* if it exists, or *value* if it doesn't.

os.**getpid**() returns the current process id.

1.2 Process management

These functions may be used to create and manage processes. The various **exec*** functions take a list of arguments for the new program loaded into the process. In each case, the first of these arguments is passed to the new program as its own name rather than as an argument a user may have typed on a command line.

os.**execl**(*path, arg0, arg1, ...*)
os.**execle**(*path, arg0, arg1, ..., env*)
os.**execlp**(*file, arg0, arg1, ...*)

os.**execlpe**(*file, arg0, arg1, ..., env*)
os.**execv**(*path, args*)
os.**execve**(*path, args, env*)
os.**execvp**(*file, args*)
os.**execvpe**(*file, args, env*)

All the above calls execute a new program, replacing the current one; they do not return. On Unix, the new executable is loaded into the current process, and will have the same process id as the caller. The current process is replaced immediately. Open file objects and descriptors are not flushed, so if there may be data buffered on these open files, you should flush them using sys.stdout.flush() or os.fsync() before calling an exec* function. The "l" variants are used with a fixed number of parameters in the command-line; the individual parameters simply become additional parameters to the execl*() functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced. The variants which include a "p" near the end (execlp(), execlpe(), execvp(), and execvpe()) will use the PATH environment variable to locate the program *file*. When the environment is being replaced, the new environment is used as the source of the PATH variable. The other variants, execl(), execle(), execv(), and execve(), will not use the PATH variable to locate the executable; *path* must contain an appropriate absolute or relative path. Following are other important functions.

os.**fork**() forks a child process. Return 0 in the child and the child's process id in the parent. If an error occurs OSError is raised.

os.**forkpty**() forks a child process, using a new pseudo-terminal as the child's controlling terminal. Return a pair of (pid, fd), where *pid* is 0 in the child, the new child's process id in the parent, and *fd* is the file descriptor of the master end of the pseudo-terminal. For a more portable approach, use the **pty** module.

os.**kill**(*pid, sig*) sends signal *sig* to the process *pid*. Constants for the specific signals available on the host platform are defined in the **signal** module. In Windows, the signal.CTRL_C_EVENT and signal.CTRL_BREAK_EVENT signals are special signals which can only be sent to console processes which share a common console window, e.g., some subprocesses. Any other value for *sig* will cause the process to be unconditionally killed by the TerminateProcess API, and the exit code will be set to *sig*. The Windows version of kill() additionally takes process handles to be killed.

os.**times**() returns a 5-tuple of floating point numbers indicating accumulated (processor or other) times, in seconds. The items are: user time, system time, children's user time, children's system time, and elapsed real time since a fixed point in the past, in that order. On Windows, only the first two items are filled, the others are zero.

Lab Work

Task 1 Read the documentation pointed by the link https://docs.python.org/3/library/os.html

Task 2 Execute the following commands:

import os	os.getcwd()	os.listdir('.')	os.mkdir('./dir5')	os chdir('./dir5')

Create a python batch file of your choice that operates with directories and files. For example, it creates few directories, copies an existing file into a new directory and then reads the file. Then show the benefits of using this batch file with some applications.

Task 3 Run the following two code samples on Kubuntu:

```
import os
pid = os.fork()
if pid == 0: # the child
    print "this is the child"
elif pid > 0:
    print "the child is pid %d" % pid
else:
    print("An error occured")
```

In the next code you need to insert the new program commanded by execl:

```
import os
pid = os.fork()
# fork and exec together
print "second test"
if pid == 0: # This is the child
    print "this is the child"
    print "I'm going to exec another program now"
    os.execl() # insert the new program here
else:
    print "the child is pid %d" % pid
    os.wait()
```

Task 4 Write your own Python program that does the following operations:

- detect if the platform is Linux or Windows;
- for the platform detected, include in your program the execution of a specific command/new program at your choice (for example, fork() in Linux and spawn() in Windows);
- introduce comments in the program.

Return your results with code and execution screenshots, in a pdf file on moodle, by the deadline. The total for this lab is 5 marks.