THEORY OF INFORMATION, ARCHITECTURE OF COMPUTERS AND OPERATING SYSTEMS (TIACOS)
Bioinformatics, ESCI
DAC, UPC

ProcLab **Spring 2023**

Objectives

The objectives of this session are practicing with the basic system calls to manage processes, and the basic commands and mechanisms to monitor kernel information associated to the active processes of the system. You'll do this by creating processes that execute functions. These functions are quite artificial, but you'll find yourself thinking much more about processes in working your way through them.

Skills

- BASIC user level:
  - Be able to do a concurrent program using process management system calls: `fork, execlp, getpid, exit, wait, waitpid, signal, kill.`
  - Understand the process inheritance and the parent/child relation.
- BASIC admin level
  - Be able to see the process list of a user and some details of his/her status using commands (ps, top).
  - Get information through the /proc pseudo-filesystem.

Handout Instructions

> This wording has a companion file, `proclab-files.tar`, with two Python scripts `proclab.py` and `testproclab.py`. You must edit and deliver `proclab.py`. The other script, `testproclab.py` will help you during the lab.

Start by copying `proclab-files.tar` to a (protected) directory on a Linux machine in which you plan to do your work. Then give the command

```
$ tar xvf proclab-files.tar
```

This will cause the two files to be unpacked in the directory. The only file you will be modifying and turning in is `proclab.py`.

The `proclab.py` file contains a skeleton for each of the 8 proclab functions. Your assignment is to complete each function skeleton using a restricted number of system calls. See the comments in `proclab.py` for detailed rules and a discussion of the desired coding style.

You can use `testproclab.py` to check your functions. The program `testproclab.py` uses `proclab.py` as a module and call your functions in the proper way. Note, if your functions do not pass all the test, you know for sure that more work is needed. On the contrary, your functions can pass the test and still not work properly. Pass `testproclab.py` is necessary but not sufficient condition. The program `testproclab.py` has one argument, the number of processes to create.

```
jfornes@ginjol ProcLab % ./testproclab.py -n 3
*** [000] calling createChildren(n,f) ***
--- [000] createChildren(n,f) passed ---
*** [001] calling waitNChildren(n) ***
2632
2633
2634
--- [001] waitNChildren(n) passed ---
*** [010] calling check_status(status) ***
--- [010] check_status(status) passed ---
*** [011] calling launchCmd(f,a) ***
...
```

The functions

Name	Description	Rating
`createChildren(n,f)`	Create `n` processes that execute function `f`.	3
`check_status(status)`	Checks `os.wait()` output. Returns pid, exit value and cause of death.	2
`waitNChildren(n)`	Waits for `n` children using `os.wait()` syscall.	2
`waitChildren()`	Waits for **all** already dead children using `os.waitpid(pid, options)` syscall.	4
`chgProcImg (path,args)`	replaces the current process image with a new process image.	1
`launchCmd (f,a)`	Creates a process and replaces the child process image with program named `f`.	2
`sendSignal(p,s)`	Sends signal `s` to process `p`.	1
`sigAction(s,f)`	Set the handler for signal `s` to the function handler `f`.	1

Table 1: Proclab functions.

Process management system calls

Read the manual pages of the `getpid/fork/exit/waitpid/execlp` system calls. Understand the parameters, return values and basic functionality associated to the theory explained at the lectures. Notice the necessary includes, error cases and return values. Consult the description and indicated options of the command `ps` and the `/proc` pseudo-filesystem. See table 2.

Manual page	Basic description	Chapter
getpid	Return the PID of the executing process	2
fork	Creates a new process, child of the executing process	2
exit	Ends the executing process that calls it	3
waitpid	Waits until the finalization of a child process	2
execlp	Executes a program in the context of the same process	3
signal	Simplified software signal facilities	2 and 7
kill	Send signal to a process	2

Table 2: Process management Linux system calls

System calls from Python

The os module provides a portable way of using operating system dependent functionality. Some of its functions have different behavior in differents systems. In this course, we always assume that the underlaying operating system of the Python virtual machine is Linux. The sys module provides functions and environment variables close related with the shell that launched the program. See table 3 and read the Python library web pages of all three modules (os, signal and sys).

C system call	Python interface
pid_t getpid(void);	os.getpid()
pid_t fork(void);	os.fork()
void exit(int status);	sys.exit(n)
pid_t waitpid(pid_t pid, int *wstatus, int options);	os.waitpid(pid, options)
int execlp(const char *file, const char *arg, .../* (char *) NULL */);	os.execlp(file, arg0, arg1, ...)
sighandler_t signal(int signum, sighandler_t handler);	signal.signal(signalnum, handler)
int kill(pid_t pid, int sig);	os.kill(pid, sig)

Table 3: Python process management interface

Creating processes

In order to create a process in Python, we are using the system call `os.fork()`. It forks a child process. It returns 0 in the child and the child's process id in the parent. If an error occurs OSError is raised. You can use the Linux commands `ps` and `top` to check that your processes have been create. Besides, remember that a directory is created for each process in `/proc/<pid>/` (replace `<pid>` with the child's pid).

Error checking

As you can see, all the manual pages related to system calls have a similar structure. Towards the end there is a section titled RETURN VALUE. A negative value indicates failure. In this section the manual explains the reasons why it may fail. Although in Python we could use the negative return value to handle errors, it is much clearer and more elegant to use exceptions.

For instance, `fork` returns -1 if it could not create the child process. Returns 0 to the child process and returns the pid of the child to its parent. The list shows the treatment of the three alternatives

```
1 import os, sys
2 try:
3     pid = os.fork()
4 except OSError as err:
5     print("OS error: {0}".format(err))
```

```
else:
    if pid == 0:
        print("I'm the child")
    else:
        print("I'm the parent")
```

Print a message is ok if you are inside the main function, that acts as interface with the user. However, inside the rest of functions is not a good idea to print messages. You also have the option to raise exceptions until the main function deals with them.

For example:

```
def this_fails():
        x = 1/0
try:
        this_fails()
except ZeroDivisionError:
        raise
```

All the functions in the `os` module raise exception `OSError(errno, strerror,...)` in case of -1 returned by the syscall. You can check the error number (as it appears in the manual page) o just print the string error message. Therefore, you can be more especific when raise your exceptions. Or even create your own exceptions. Imagine we allow the creation of three processes only. We can raise and exception if that threshold is reached. Take a look:

```

try:
        i+=1
    pid = os.fork()
    if pid == 0:
        print("I'm the child\n")
    else:
        if i>2:
                raise Exception("too many")
        print("I'm the parent\n")
     except OSError:
            raise OSError("no child")
```

See https://docs.python.org/3/tutorial/errors.html.

Terminating processes

Children processes, created with `os.fork()`, can use `os._exit(n)` to exit. However, the general form is `sys.exit(n)` that performs several cleanup actions. The argument `n` is sent to the parent process. Parent process has to call some wait function, such as `os.wait()` or `waitpid(pid,options)` in order to wait the completion of a child process and to obtain the value `n`. See theory slide number 47 to know how.

Interprocess communication

In this laboratory, we will utilize a minimal portion of the signal's potential. Signal transmission and reception serve as a means of communication between processes. In the upcoming laboratory sessions, we will explore this concept further, along with other communication mechanisms, including message passing. Use `os.kill(pid, signum)` to send the signal `signum` to the process with identifier `pid`. You can also send signals from the shell:

```
jfornes@tiacos:~/proclab$ ps -ef | grep hello
jfornes    33784   29191 99 20:08 pts/1    00:00:15 /usr/bin/python3 ./hello.py -l
jfornes    33786    3429  0 20:08 pts/0    00:00:00 grep --color=auto hello
jfornes@tiacos:~/proclab$ kill -KILL 33784
```

Note that in this case, `signum` goes first (you can use the actual number, 9 for `SIGKILL`) and the `pid` is the second argument.

`SIGKILL` and `SIGSTOP` cannot be reprogrammed (caught), but for the rest you can use `signal.signal(signum, handler)` to define customs handlers to be executed when signal `signum` is received.

Evaluation

Your score will be computed out of a maximum of 32 points based on the following distribution:

16 Correctness points.

8 Performance points.

4 Method points.

2 Style points.

Correctness points. The functions you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 16. You will get full credit for a function if it works, and no credit otherwise.

Performance points. We want to instill in you a sense of keeping things as short and simple as you can. For each function we've established a set of system calls and functions you are allowed to use. You will receive two points for each correct function that satisfies that.

Programming points You can earn 4 extra points for good programming. You must check for syscall errors and keep things as short as possible.

Style points. Finally, we've reserved 2 points for a subjective evaluation of the style of your solutions and your commenting. Your solutions should be as clean and straightforward as possible. Your comments should be informative, but they need not be extensive.

Deliverable

In this lab, you only have to upload `proclab.py` via Moodle.

References

[Python 3.9] *The Python Language Reference*
https://docs.python.org/3/reference/index.html

[Randal E. Bryant, David R. O'Hallaron] *Computer systems: a programmer's perspective*
Prentice Hall, 2015. Chapter 2.
https://upfinder.upf.edu/iii/encore/record/C__Rb1318766