Session 1 : Processes and a first taste of IPC

Razmig Kéchichian

Learning outcomes

The aim of this session is to familiarize you with:

- process creation tools in Python
- parent-child process relationships
- simple asynchronous interprocess communication via signals, and
- simple synchronous interprocess communication via ordinary pipes

I. Preliminaries (5-10 minutes)

Organizing your code and organizing your time

Always consider organizing your source files in a directory hierarchy. As you know, such organization is mandatory if you intend to organize your Python modules into packages. Also consider the use of a version-control system, such as `git` to consolidate your work instead of simple but often error-prone archiving. Organize your time taking into consideration duration estimates given for each section. The aim of practice sessions is to discover as many notions and tools as possible. Do not get stuck in some place for the duration of the session leaving out later sections, you can go into the problematic section in depth later.

Running Python programs

Remember that you can run your Python programs on the command line in 2 ways. Say your program is contained in the source file `myscript.py`:

- You can pass the script to the Python interpreter directly along with any arguments it may need:

```
login@hostame:~$ python3 myscript.py "bartzabel" 666
```

- Or, you can make your script executable and run it directly. In this case you need to place the following line at the very beginning of your script to instruct the bash shell on where to look for your Python interpreter, the location of which may vary from a system to another, hence the call to the `env` utility:

```
#!/usr/bin/env python3
```

  Then, give yourself and your group execution rights on the script and launch it:

```
login@hostame:~$ chmod ug+x myscript.py
login@hostame:~$ ./myscript.py "bartzabel" 666
```

Perusing documentation

Throughout the practice sessions, you will often be directed to documentation sections to read and code examples to examine. This material is often short and can be viewed and understood quickly, therefore you are strongly encouraged to take your time to do so. In addition to using the Python docs, remember that you can access the documentation of any Python function or object by passing its name to the built-in Python function `help()`.

Play with the code and elaborate solutions incrementally

Do take your time to test a new function or a class that you just discovered to make sure you understand how it works. Resist the urge to code your program in its entirety in one shot. Going about it incrementally with intermediate tests makes things easier and keeps bugs in control!

II. Processes (30-40 minutes)

Python process creation and associated tools of IPC and synchronization are defined in the standard module `multiprocessing`. Central to this module is the `Process` class defining the process abstraction along with its attributes and operations represented as methods. `Process` allows you to create a new process in your Python program via 2 methods:

1. Implementing the algorithm that you need to execute as a function and passing its name along with its arguments to the constructor of `Process`, or
2. subclassing `Process` and defining its `run()` method to implement your algorithm.

Here's a runnable code example illustrating the first method.

```
from multiprocessing import Process

def greet(name):
    print("Hello,", name, "!")

if __name__ == "__main__":
    p = Process(target=greet, args=("Kitty",))
    p.start()
    p.join()
```

And here's the same example, illustrating the second method.

```
from multiprocessing import Process

class Greet(Process):
    def __init__(self, name):
        super().__init__()
        self.name = name
    def run(self):
```

```
        print("Hello,", self.name, "!")

if __name__ == "__main__":
    p = Greet("Kitty")
    p.start()
    p.join()
```

Two things have to be taken into consideration:

- When you subclass the `Process` class, in your constructor you must first invoke the constructor of the parent class, hence the line `super().__init__()`.
- In either way of process creation, protecting the entry point of the program by `if __name__ == "__main__":` is **mandatory**. This is to make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects, such a starting a new process. Have a look at this description of how to execute Python modules as scripts if you're not comfortable with the idea.

Now's a great time to have a quick look on the documentation of the `Process` class to see if you guessed right what its methods `start()` and `join()` do, and to find out about its other members.

Before we tackle our first exercise, it is interesting to point out that the standard module `os`, which provides an interface to the underlying operating system, defines a host of useful methods for use with processes, such as `getpid()` and `getppid()` which return the PID of the calling process and the PID of its parent process respectively. However, not all of the facilities defined in this module are available on all platforms.

Exercise 1 : parent with worker child

The Fibonacci sequence is the series of numbers $0, 1, 1, 2, 3, 5, 8, \ldots$
Formally, it can be expressed as:

$$
\begin{aligned}
fib_0 &= 0 \\
fib_1 &= 1 \\
fib_n &= fib_{n-1} + fib_{n-2}
\end{aligned}
$$

- Write a Python program that that generates the Fibonacci sequence in the child process. The index of the sequence element will be provided at the command line. For example, if 5 is provided, the first six numbers in the Fibonacci sequence will be output by the child process. Because the parent and child processes have their own copies of data, it will be necessary for one of them, the child in this case, to output the sequence. Make sure that the parent waits for the child process to complete before exiting the program. You may either define a process class by inheriting from `multiprocessing.Process`, or define a function and provide it as the target to be executed along with its parameter to `multiprocessing.Process()`.
- Perform necessary error checking to ensure that a non-negative number is passed on the command line. Remember that the `argv` member of the standard module `sys` captures command-line arguments as `str` objects. You can convert a `str` object into an `int` by passing it to the `int()` constructor and checking for possible exceptions.
- Put the child process to sleep via `time.sleep(n)` for `n` seconds and locate the entries of both processes in the output of the command `ps ax`, issued in another terminal. Call `os.getpid()` and `os.getppid()` in the appropriate places to check the process and parent process PIDs respectively and verify with the output of the `ps` command.

III. Signals (20-30 minutes)

Signals are a useful means for asynchronous communication with processes, and often for interprocess communication as well, although more limited than usual message passing or shared memory IPC. Note that signals are available on Unix-like systems only. In essence, a signal is a notification sent to a process given its PID, to which the process responds by calling a signal handler function implementing a default or a user-defined behavior.

There are different types of signals, to glance through them, have a look on the `signal` manpage, `man 7 signal`, or consult this page if manpages aren't available on your system. Some signals have special meanings, for example `SIGTERM` is sent to a process to request its proper termination, `SIGKILL` kills a process immediately without allowing it to cleanup its resources, `SIGINT` delivers a keyboard interrupt, an so on. Some signals, such as `SIGUSR1` and `SIGUSR2`, are available for users to implement a desired behavior.

Let's begin with a rather simple example. Write this 2-liner program and have it run in a terminal.

```
while True:
  pass
```

In a separate terminal, issue the command `ps a` and take note of the PID of the process you just launched. Now send a `SIGKILL` to it by issuing `kill -s SIGKILL PID` where PID is the actual numeric PID of the process. Observe what happens in the first terminal. Re-run the the Python program, get its new PID, and this time send it a `SIGINT`. Re-run the program for the last time, and this time type `Ctrl+C` in the first terminal. You will notice that Python reports keyboard interrupts by raising a `KeyboardInterrupt` exception.

You can intercept a signal and decide what to upon its arrival by defining a signal handler function. You then install this signal handler for the given signal. Let's develop the above simple program to intercept `SIGINT` and print a smiley face instead of terminating the program, which is the default behavior upon receiving the signal. The code follows.

```
import signal

def handler(sig, frame):
    if sig == signal.SIGINT:
        print("}:-)")

signal.signal(signal.SIGINT, handler)

while True:
    pass
```

As you have guessed, Python signal handling routines are defined in the `signal` module. Two remarks are in order:

- A signal handler function can be installed for a given signal via the `signal.signal()` method. This method can also be used to ignore a signal or to redefine its default behavior. Find out how by looking at its documentation. In general, it is not a good idea to ignore or to redefine signals having to do with system-process interaction such as `SIGTERM`,

`SIGKILL`, etc.
- The signal handler is a normal Python function that is called when the signal is received by the process. The first argument passed is the signal number, which allows you to use the same handler for different signals. The `signal` module provides signals as uppercase constants. You can safely ignore the second argument to the handler function. Go ahead and test the new program sending it a `SIGINT`, then terminate it.

Before moving to the second exercise, let us stress on the fact that a process must be alive for it to be able to receive a signal. Without the `while` loop, the process would have terminated without having the chance of receiving a signal. An alternative, and a much better way, is to call the method `signal.pause()` to suspend the execution of the process until the signal is received making sure the appropriate handle is called.

Exercise 2 : signal exchange

Write a Python program to implement the following scenario. A parent process creates a child and waits for its termination. After sleeping for 5 seconds, the child process sends a signal to its parent. Upon intercepting the signal, the parent kills its child. Make sure to use a user-available signal for child-parent communication. To send a signal to a process given its PID, you can call the method `os.kill()`. Have a look on its documentation.

IV. Ordinary pipes (20-30 minutes)

The Python `multiprocessing` module provides ordinary (sometimes called unnamed) pipes to allow a pair of processes to communicate via the `multiprocessing.Pipe()` method. This method actually returns 2 `multiprocessing.Connection` objects connected through a pipe which is duplex (two-way) by default.

You are required to examine and understand this code example illustrating a simple one-way use of a pipe. Have a look at the documentation of the `Pipe()` method, then look into the documentation of the `Connection` class, paying attention to its methods `send()`, `recv()` and `close()`. Make sure to understand that a pair of processes have to use different ends of an ordinary

pipe at the same time, using the same end of the pipe at the same time causes data corruption.

Before moving to the last exercise, let us note here that even though ordinary pipes are often used by a pair of process in a parent-child configuration, they could also be used between two child processes created by the same parent. This is because file descriptors underlying `Connection` objects used to establish pipe communications are inherited in child processes. Indeed, the child process is an exact duplicate of the parent process except for few things. On a Unix-like system, you can find out what else a child process inherits from its parent by consulting the manpage for the `fork()` system call.

Exercise 3 : two-way ordinary pipe

Write a Python program to take advantage of the bidirectional nature of ordinary pipes for the following situation. A parent process prompts the user (you may use the built-in method `input()` for that) and reads a phrase from the terminal. It sends the phrase to its child process which reverses it and sends it back to its parent, the parent displays the reversed phrase. First, implement and test the program for a single round-trip communication, then extend it to accept an arbitrary number of communications with a stop phrase or word such as “end”. Always make sure to call the `close()` method on pipe connections once the program is done with them to release resources associated with `Connection` objects.

V. Going further: named pipes

We just studied communication among parent-child processes via ordinary pipes. However, sometimes it is necessary to perform communication between unrelated processes (processes with different parents). If the use of shared memory isn’t desirable, named pipes can be resorted to. These are communication mechanisms which use file descriptors associated with special files on the file system that implement a First-In First-Out (FIFO) scheme for writing and reading data. Recall that file descriptors for ordinary pipes are created in memory. You are autonomous in your exploration of named pipes. Python routines for creating and communicating via named pipes are defined in the `os` module. Note however that these routines are available on Unix-like

systems only.