Session 2 : Interprocess communication

Razmig Kéchichian

Learning outcomes

The aim of this session is to familiarize you with:

- sharing state between processes (shared memory), and
- System V-style message passing.

I. Sharing state

The Python concurrent programming guidelines advice avoiding shared state between processes as much as possible. They suggest using pipes and queues (not to be confused with message queues both of which we'll cover later) instead. The reason is that such shared state often has to be manually synchronized to guard against *race conditions*, such as simultaneous writes and reads which lead to data corruption. Nevertheless, shared sate can sometimes be useful for faster and more convenient interprocess communication, especially when larger chunks of data are involved.

Python provides a number of ways to create and use shared data, the simplest of those are the classes `Value` and `Array`, defined in the `multiprocessing` module, which respectively represent scalar and fixed-size array data of predefined type, allocated from shared memory. The following program illustrates their use.

```python
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    number = Value('d', 0.0)
    vector = Array('i', range(10))

    p = Process(target=f, args=(number, vector))
    p.start()
    p.join()

    print(number.value)
    print(vector[:])
```

Take note of the following points:

- Objects returned by `Value()` and `Array()` are merely wrappers for the underlying data objects which have to be retrieved via the `value` attribute for the former and by an indexing or slicing operator for the latter.
- The `'d'` and `'i'` arguments to `Value()` and `Array()` are typecodes of the kind used by the `array` module: `'d'` indicates a double precision float and `'i'` indicates a signed integer. The `array` module documentation gives the full list of typecodes.
- Shared objects created by `Value()` and `Array()` are process and thread-safe by default, meaning that Python automatically synchronizes concurrent access by multiple processes or threads so that race conditions do not occur. Take a moment to look at the documentation of `Value` and `Array` to understand how this process and thread-safety is achieved.

A second, more advanced way to create shared data is via Python *managers*, defined in the standard module `multiprocessing`. Managers allow creating data which can be shared between different processes, including sharing over a network between processes running on different machines. A manager actually runs a background process which manages shared objects. Other processes can access the shared objects by using proxies. Managers support a host of process and thread-safe types such as `list`, `dict` and many others. They can also be extended to support arbitrary object types. However, this mechanism is slower than `Value` and `Array` direct shared memory approach discussed above. The following program illustrates the use of a manager.

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = 'one'
    d['two'] = 2
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        dct = manager.dict()
        lst = manager.list(range(10))

        p = Process(target=f, args=(dct, lst))
        p.start()
        p.join()

        print(dct)
        print(lst)
```

Notice how the shared dictionary and list (`dct` and `lst`) objects are accessed in both parent and child processes. If you are unfamiliar with the `with ... as ...` statement, then you might want to have a look on this description of Python *context management*. In the case of `multiprocessing.Manager`, the context manager ensures the proper termination of the process managing the shared objects, even if an exception occurs in the `with` block. Otherwise, `manager.shutdown()` must be explicitly called after finishing work with the manager. A more common example of Python context management is with file objects created via the built-in call `open()`.

Because shared memory segments created in a process are "inherited" by its children, it is most natural to use shared state for interprocess communication between processes in parent-child configuration. This is the paradigm supported by `Value` and `Array` direct shared memory objects and those created and managed by `Manager`. As mentioned in the above, it is still possible to define managers sharing objects between unrelated processes. We won't be addressing this mechanism in this session.

Exercise 1 : The Fibonaccis return

In Exercise 1 from Session 1, the child process had to output the Fibonacci sequence it generated because the parent and child have their own copies of data. Another approach to designing this program is to establish a shared-memory object between the parent and child processes. This technique allows the child to write the contents of the sequence to the shared-memory object.

The parent can then output the sequence when the child completes. Write a Python program to implement this scenario using either the direct shared memory approach or the manager approach. You can reuse parts of your solution for the previous exercise.

II. Message passing

The Python standard library doesn't provide process-independent many-to-many message passing facilities. Recall that pipes involve only a pair of processes. The `multiprocessing.Queue` class implements a First-In-First-Out process-safe queue intended for exchanging objects between multiple processes. `multiprocessing.Queue` objects are typically used to feed data and tasks (often represented as callables) to a collection of worker processes and to collect results, such as in the last example on this page of the Python library reference. We shall make use of queues when we address process and thread pools.

Powerful third-party Python messaging libraries do exist nevertheless, such as ZeroMQ. However, their use is out of the scope of this course. We shall study message passing mechanisms through `sysv_ipc`, a third-party Python interface of System V interprocess communication facilities which are available on all Unix-like operating systems, including Mac OS X.

Let's first proceed to install the `sysv_ipc` module. A local installation is preferable, and is the only choice if you don't have root access to your machine. Visit the homepage of `sysv_ipc`, retrieve the compressed tar file of the latest release, uncompress it and issue the following command in the root directory of the release:

```
login@hostame:~$ python3 setup.py install --user
```

To make sure the installation is successful, import `sysv_ipc` in an interactive Python session, and examine the attribute `sysv_ipc.__version__`.

Let us turn our attention now to message queues. To illustrate their use, we shall use the following minimal example. An independent process, called the server, creates a message queue then reads integers from the terminal sending them into the message queue until zero (0) is read, at which point the server stops input, removes the message queue, and terminates. A second independent process, called the client, connects to the message queue created by the server, receives integers sent by the latter displaying them on the terminal until it encounters zero (0), at which point it terminates.

The source code of the server program follows.

```
import sysv_ipc

key = 128

mq = sysv_ipc.MessageQueue(key, sysv_ipc.IPC_CREAT)

value = 1
while value:
    try:
        value = int(input())
    except:
        print("Input error, try again!")
    message = str(value).encode()
    mq.send(message)

mq.remove()
```

And that of the client program.

```
import sysv_ipc

key = 128

mq = sysv_ipc.MessageQueue(key)

while True:
    message, t = mq.receive()
    value = message.decode()
    value = int(value)
    if value:
        print("received:", value)
    else:
        print("exiting.")
        break
```

Before running these programs, examine them carefully and make sure you understand how they work by referring to the documentation of the `sysv_ipc.MessageQueue` class and taking the following remarks into account.

- Messages passed to `MessageQueue.send()` and retrieved by `MessageQueue.receive()` are objects of type `bytes`. Objects of this type can be created literally, e.g. `b"Razmig"`, or by calling the `encode()` method on `str` objects to convert from `str` to `bytes`. A `bytes` object can be converted to `str` by calling its `decode()` method.
- We can associate integer identifiers with messages, referred to as message types. This allows us to retrieve specific messages from the queue. If unspecified upon sending, the message type defaults to 1. If unspecified upon receiving, the message type defaults to 0, which translates to retrieving the first message on the queue. This is the exact behavior of our programs.
- In order for queue operations send and receive to succeed, the queue must be alive. Algorithms must be designed so that these operations aren't attempted after the removal of the queue, which should be systematically carried out at the end of the communication.

Go ahead and run the two programs in 2 separate terminals. Launch a third terminal and issue the command `ipcs`. Examine its output and find the section that lists message queues. You should be able to locate the active message queue by its key. If something goes wrong and the server program crashes before it had the opportunity to remove the queue, you can do that manually by issuing the command `ipcrm -Q key`, where `key` is the key that was used to create the message queue in question.

Exercise 2 : Is it time yet?

We'd like to implement a time server that creates a message queue and listens on it. It understands 2 types of messages, 1 and 2. Messages of type 1 are time requests. Upon receiving a time request, the server reads the system clock (see `time.asctime()`), and sends it into the queue as message type 3. A message of type 2 is a termination request. Upon receiving it, the server removes the message queue and terminates. The client program connects to the message queue, prompts the user for the message type to send to the server and communicates with the latter accordingly. Upon receiving a message from the server in response to a time request, it displays the message on the terminal. Implement the server and the client programs using the message passing facilities of the `sysv_ipc` module. Test with multiple clients.