

Inter-Process Communication

<https://youtu.be/xuaoQGXSxM>

Inter-process Communication

Review:

Previously we've looked at threads and processes as methods of creating concurrent solutions.

What were the differences between threads and processes?

Inter-process Communication

Threads vs. Processes:

Threading: All threads share the same process space which means that they share: memory for code, memory for data, opened files. Each thread has it's own execution context (stack & registers). One interpreter manages the creation of & switching between threads.

This is true for a single threaded solution too, but with only one thread there is no switching.

Inter-process Communication

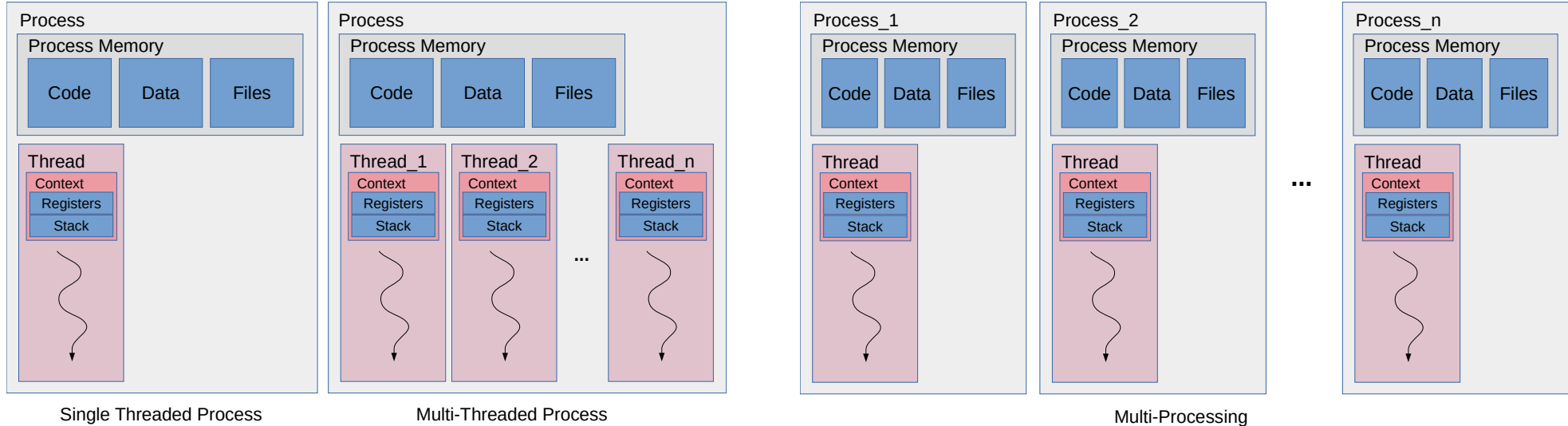
Threads vs. Processes:

Multi-Processing: Each process has its own process space which means that they have independent: memory for code, memory for data, opened files. Each process is managed by the OS and has its own independent interpreter. Each process could, however, have multiple threads.

Note: Processes are totally independent.

Inter-process Communication

Threads vs Processes:



Note: Threads can easily share data, processes are isolated and independent.

Inter-process Communication

Threads vs Processes:

Implications:

- Threads:
- Directly can share data making it easy to work cooperatively.
 - Low overhead for creating and managing threads.
 - Must share single interpreter.
 - Execution is concurrent but not truly parallel.
- Processes:
- Can't directly share data. Must use other mechanisms to work cooperatively.
 - Higher overhead for creation and management.
 - Each has an independent interpreter.
 - Execution can be truly parallel.
 - Can be more easily scaled across multiple systems.

Inter-process Communication

Communicating Between Processes:

Processes have some tremendous advantages, but for cooperative solutions we need to overcome the lack of shared memory.

To do this we need a way to get multiple processes to communicate and/or share data without directly sharing memory. There are several options available:

Queue Objects – We've seen these before with processes. Typically a FIFO.

Pipes – A two-ended communication mechanism that allows inter-process communication.

Value / Array Objects – Simple data objects that can be shared between processes.

Manager Objects – Special data objects that can be accessed by multiple processes.

Ctypes – Objects that utilized shared memory that child processes can access.

Sockets – These allow the sending / receiving of data between processes. Can communicate between processes on the same or on different machines.

Other – There are frameworks and protocols that can simplify and standardize this. (Ex: MPI)

We'll look at a couple of these today. (We'll look at the others later.)

Inter-process Communication

Communicating Between Processes:

Review of Queues:

```
import multiprocessing
main_queue=multiprocessing.Queue()
def myprocess(shared_queue):
    #do stuff here
    shared_queue.put(result)
if __name__ == "__main__":
    process_list=[]
    for i in range(10):
        process_list.append(multiprocessing.Process(target=myprocess,args=(main_queue,)))
    for p in process_list:
        p.start()
    for p in process_list:
        p.join()
    for i in range(10):
        result=main_queue.get()
        print("Result:",result)
    input("enter to exit")
```

Queue object will be passed to all processes

Each process puts data into the same queue

Queue passed to all processes

Start all processes

Wait for all processes to finish

Main process gets all data out of the queue

Inter-process Communication

Communicating Between Processes:

Review of Queues:

Queues can be accessed by any of the processes that have access to the object.

Multiple queue objects can be used.

All entities interacting with the queue can put items in a queue or get items from a queue.

Similarly, other familiar objects for synchronization and communication can be used in this way:

Lock(), Rlock(), Condition(), Semaphore(), BoundedSemaphore(), Barrier(), Event()

Inter-process Communication

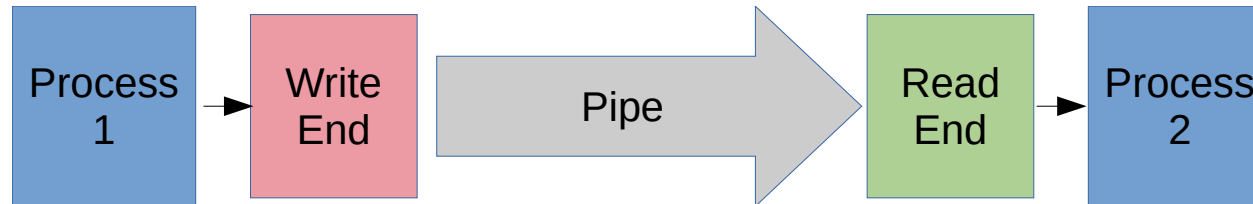
Communicating Between Processes:

Pipes

Pipes are used within operating systems as a way to connect processes together.

The general concept is that a pipe has two ends: a read end and a write end.

One process puts data into the write end and the other process pulls data out of the read end. Pipes are a uni-directional mechanism.



Inter-process Communication

Pipes:

Most operating systems allow this to be done on a command line using sometime like this:

In UNIX: `$ cat file.txt | grep "Yost" | wc -l`

Will launch 3 processes:

P1: `cat` – `cat` copies the contents of the specified file to stdout.

P2: `grep` – `grep` filters the input stream (stdin) for the specified pattern “Yost”.

P3: `wc` – word count counts the number of lines.

The command interpreter starts the programs in separate processes, creates the pipes, and connects each pipe-end to the processes’ stdin or stdout as specified.

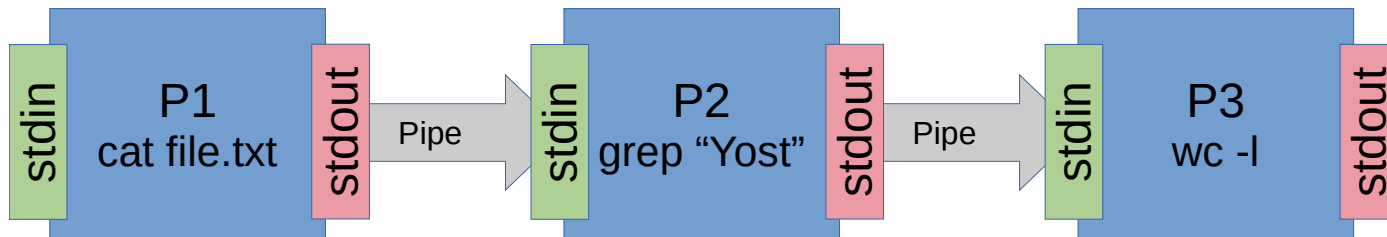
Inter-process Communication

Pipes:

So, the UNIX command line:

```
$ cat file.txt | grep "Yost" | wc -l
```

Results in this:



The command interpreter launches the processes and creates and connects the pipes. The programs only have to read from stdin and write to stdout for this to work. The programs don't even have to know that they are talking to each-other.

Inter-process Communication

Pipes:

Pipes also work in Linux, Windows, Mac, and other operating systems.

While the OS can manage these, we can also create our own and manage them in code.

The easiest way to do this in Python is by using `multiprocessing.Pipe()`.

`Pipe()` creates an object similar to a typical OS pipe, but is duplex by default. Meaning that both ends can read/write through to the other end. When invoked, `Pipe()` returns two connection objects.

```
con1,con2 = multiprocessing.Pipe()
```

`Pipe()` supports an optional parameter: `duplex=True`

If `duplex` is set to `False`, only `con1` can only receive and `con2` can only send.

Inter-process Communication

Example:

```
import multiprocessing
import time, random

def producer(pipe):
    for message in ['hello', 'are you there?', 'hey', 'LOL', 'K', 'THX', 'Bye!', 'END']:
        time.sleep(random.uniform(1,2))
        pipe.send(message)
    pipe.close()

def consumer(pipe):
    while True:
        data=pipe.recv()
        print("Message Received: "+data)
        if data=="END":
            break
    pipe.close()

if __name__ == '__main__':
    pipeCon1,pipeCon2=multiprocessing.Pipe()
    p1=multiprocessing.Process(target=producer,args=(pipeCon1,))
    p2=multiprocessing.Process(target=consumer,args=(pipeCon2,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print("done")
    input("enter to exit")
```

Write to pipe

Read from pipe

Pipe with two connections

Inter-process Communication

Pipes vs Queues:

Pipes: Only allow 2 end-points
By default allow duplex communication
Not safe with more than one process reading or writing to the same connection.

Queues: Allow multiple producers and consumers
Safe to use with any number of processes reading and/or writing
Can emulate a pipe by only using one writer and one reader process

Pipes are faster because they are built directly upon O.S. pipes.

Note that we could even use O.S. pipes directly, but this is less portable:

```
import os

# Create a pipe
readEnd, writeEnd = os.pipe()
```

Inter-process Communication

Managers:

A manager is a special object that acts as a guard and guarantees thread-safe and process-safe access to other data and objects.

This is similar to the “monitor” concept we implemented in earlier classes, but the Manager handles much of the implementation for us.

To create a new Manager:

```
import multiprocessing  
myManager=multiprocessing.Manager()
```

Managers support the use of namespaces, which can help organize the protected data:

```
myNamespace = myManager.Namespace()
```

Once a namespace has been created, we can add attributes to it.

Inter-process Communication

Managers:

Adding attributes to a namespace is easy:

```
myNamespace.count = 0  
myNamespace.item_list=[]
```

Once the namespace has been created, it can be passed to each of the processes that will need to access it.

Since each process can access the attributes in the namespace, all data items are effectively shared in a thread-safe and process-safe manner.

Let's see a full example...

Inter-process Communication

```
import multiprocessing
import random,time


def enter_proc(ns):
    for i in range(10):
        time.sleep(random.uniform(.1,.2))
        ns.count+=1
        ns.log.append("Entered")

def exit_proc(ns):
    for i in range(10):
        time.sleep(random.uniform(.1,.2))
        ns.count-=1
        ns.log.append("Left")

if __name__ == '__main__':
    myManager=multiprocessing.Manager()
    myNamespace=myManager.Namespace()
    myNamespace.count=0
    myNamespace.log=[]
    p1=multiprocessing.Process(target=enter_proc,args=(myNamespace,))
    p2=multiprocessing.Process(target=exit_proc,args=(myNamespace,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print("count:",myNamespace.count)
    print("log:",myNamespace.log)
    input("enter to exit")
```

Output

```
count: 0
log: []
enter to exit
```



Note Error!
Log is empty?!

Inter-process Communication

Managers:

What is the actual problem?

Manager contained objects are unable to propagate changes made to mutable objects inside a container.

In other words, if you have a normal mutable Python list then any changes are not propagated because the manager has no way of detecting the change.

In order to propagate the changes, you have to use a `manager.list()` type.

This is also true for other mutable python object types such as dictionaries.

Immutable object types such as strings and numbers don't have this requirement.

Inter-process Communication

```
import multiprocessing
import random,time

def enter_proc(ns):
    for i in range(10):
        time.sleep(random.uniform(.1,.2))
        ns.count+=1
        ns.log.append("Entered")

def exit_proc(ns):
    for i in range(10):
        time.sleep(random.uniform(.1,.2))
        ns.count-=1
        ns.log.append("Left")

if __name__ == '__main__':
    myManager=multiprocessing.Manager()
    myNamespace=myManager.Namespace()
    myNamespace.count=0
    myNamespace.log=myManager.list()
    p1=multiprocessing.Process(target=enter_proc,args=(myNamespace,))
    p2=multiprocessing.Process(target=exit_proc,args=(myNamespace,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print("count:",myNamespace.count)
    print("log:",myNamespace.log)
    input("enter to exit")
```

Output

```
count: 0
log: ['Entered', 'Left', 'Entered',
'Left', 'Entered', 'Left', 'Entered',
'Left', 'Entered', 'Left', 'Entered',
'Left', 'Entered', 'Left', 'Entered',
'Left', 'Entered', 'Left', 'Entered',
'Left']
enter to exit
```

This fixes the problem.

Inter-process Communication

That's all for today. Stay safe!