ETEC3702 – Concurrency

# Inter-Process Communication
# Lab: Pipes

https://youtu.be/rXA5sRioDb4

# IPC Lab: Pipes

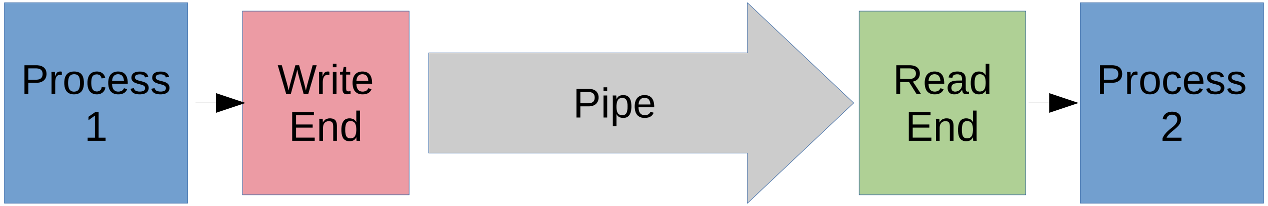
## Communicating Between Processes:

**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.



# IPC Lab: Pipes

## 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.

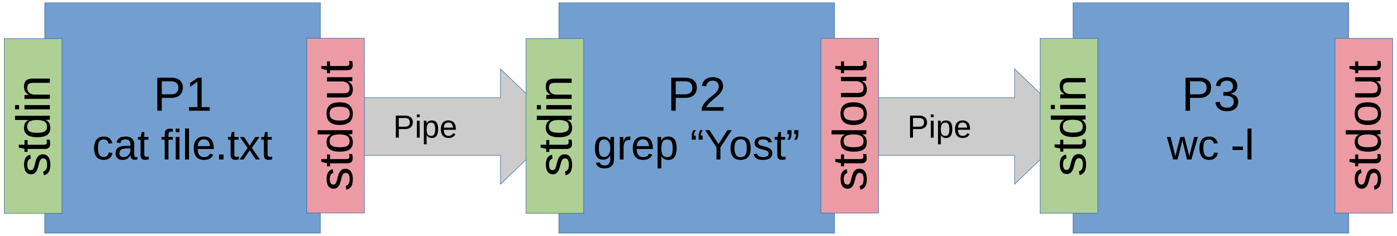
# IPC Lab: Pipes

## 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.

# IPC Lab: Pipes

## 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.

# IPC Lab: Pipes

## 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

# IPC Lab: Pipes

## 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()
```

# IPC Lab: Pipes

## Pipes Lab:

**Let’s look at the Lab Assignment!**

# IPC Lab: Pipes

That’s all for today.

Work on the lab assignment!
(Seriously!)

Stay safe!