Christopher Procum

Operating Systems CS305

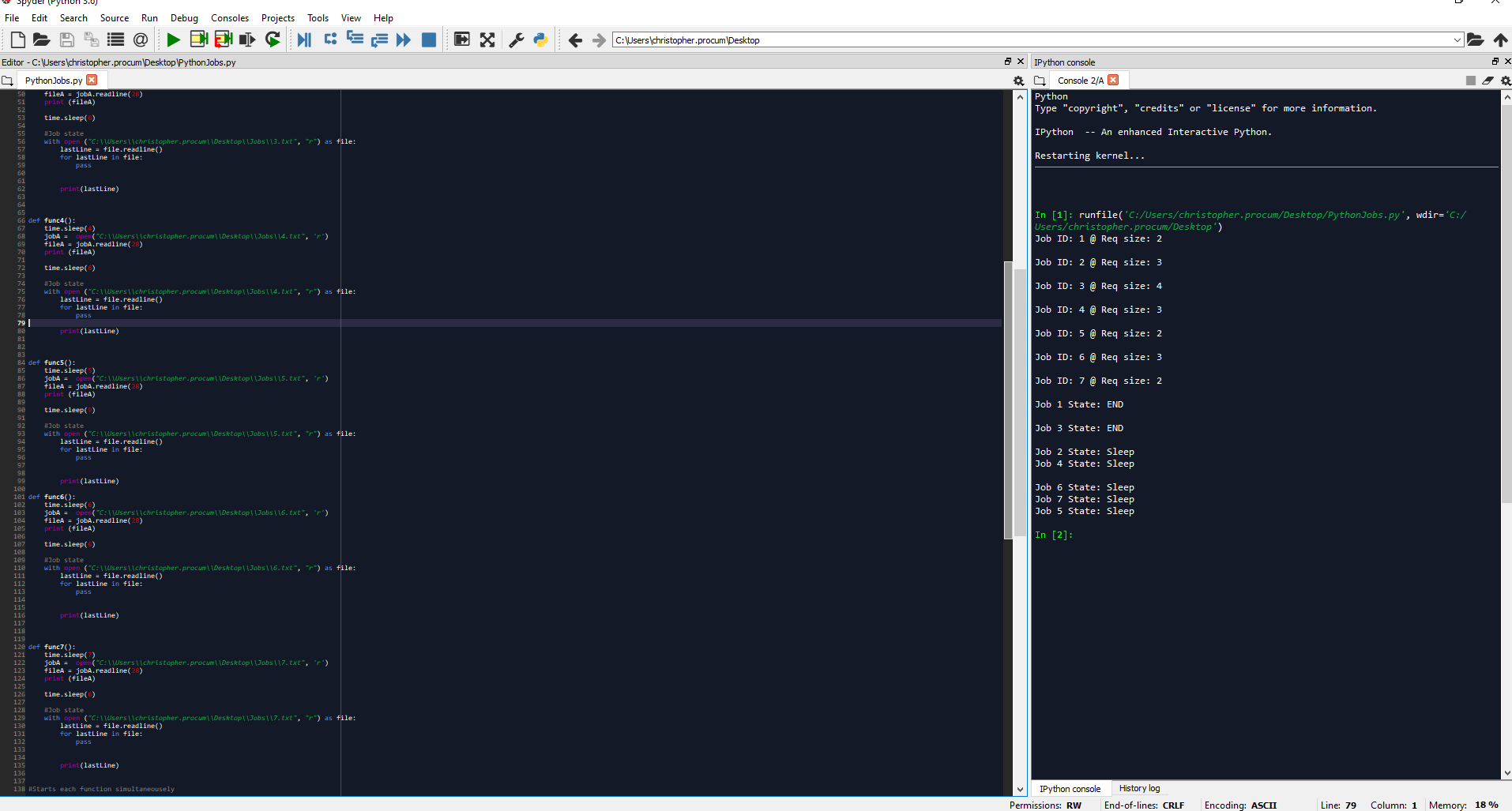
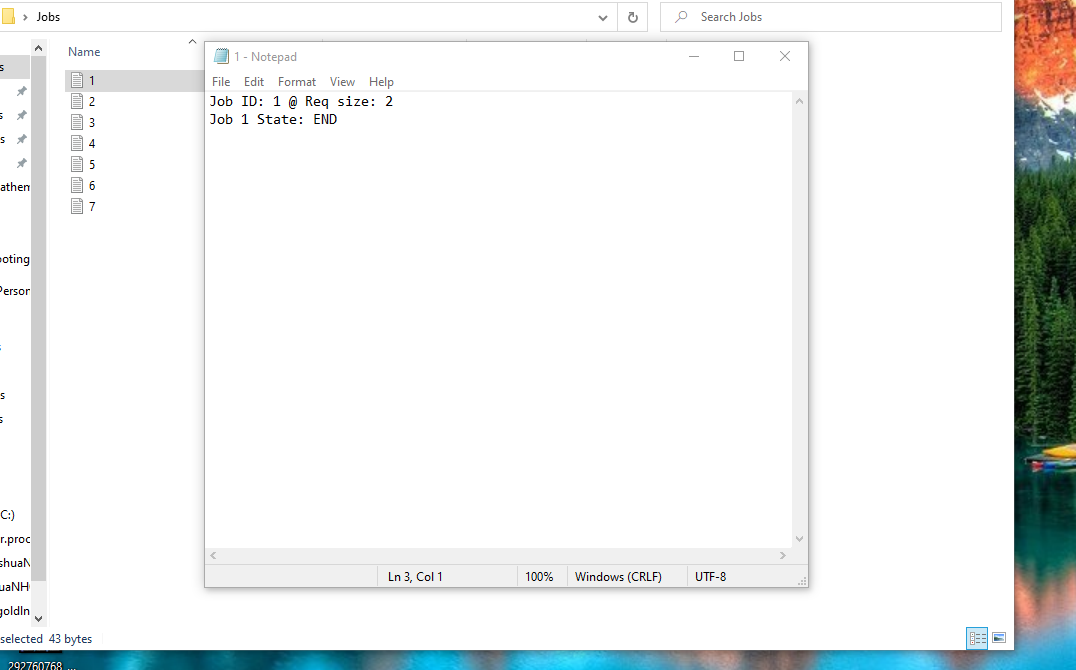
In this code I have created, I have multiple definitions for each job to be done. Each definition that is run through will count to a specific time dependent upon the given *start time* using time.sleep function. Once this delay is over, the file will open to the desired text file path and read the first 28 characters to output “Job ID” and “Required size”. The execution interval is then simulated using the sleep function once again that’s dependent upon the given information. Following this I have each definition print the last line of the text file to print the jobs state. In the screen shot of my files, you can see how I set it up so I could easily extract data from each file and simulate a multiprocessing job.

To further simulate the multiprocessing, at the bottom of the code I have threading functions which allow each definition to be started simultaneously. Once all the jobs have started, it goes through the wait times given to properly simulate a multiprocessing program. As you can see, all the jobs start in order, but because each “job” has different processing times, they all do not finish In that same order. In the code with the given information, the jobs finished in the order of: *1, 3, 2, 4, 6, 7, and 5.*

As far as how this code is set up, it simulates the FIFO, or First in First Out, method. While it may not be the most efficient, I believe with such a small amount of data in this example, it does just fine. Additionally, it’s easy to implement which is one of its main pros. As we start scaling to larger sets of data though, it would be necessary to start implementing other strategies as the performance of FIFO isn’t necessarily the quickest. For this particular data set, using the random method could work as well. However, everything seemed to finish within seconds of being started all around the same time on its FIFO method. Random could give you a run through of something slightly more efficient or something slightly less efficient, it all would depend on the order given and luck of the draw. With the thought of these different methods and all the options we have to use, I prefer the FIFO method due to the lack of data and its easy implementation. Analyzing it at NRU or NFU and even Random, I would say that would be most efficient the more data we have to process.

When looking at what OS will impact the processing better or worse, and the job state, it’s important to know some differences. Windows uses a micro kernel, while Linux has a monolithic kernel and renders Linux more efficient to run operations. If linux were the OS for this job, I would say more jobs would be at the END state versus the sleeping state due to its efficiency in operations. *“****Linux is an open source operating system whereas Windows OS is commercial****. Linux has access to source code and alters the code as per user need whereas Windows does not have access to the source code. In Linux, the user has access to the source code of the kernel and alter the code according to his need” –* <https://www.edureka.co>. As this quote states, you have much more availability to source code allowing you to alter things when needed to ensure your job is running optimally.

Screenshots of work:



Full code below:

import threading

import time

#job run

def func1():

time.sleep(1)

jobA = open("C:\\Users\\christopher.procum\\Desktop\\Jobs\\1.txt", 'r')

fileA = jobA.readline(28)

print (fileA)

time.sleep(7)

#Job state

with open ("C:\\Users\\christopher.procum\\Desktop\\Jobs\\1.txt", "r") as file:

lastLine = file.readline()

for lastLine in file:

pass

print(lastLine)

jobA.close()

def func2():

time.sleep(2)

jobA = open("C:\\Users\\christopher.procum\\Desktop\\Jobs\\2.txt", 'r')

fileA = jobA.readline(28)

print (fileA)

time.sleep(8)

#Job state

with open ("C:\\Users\\christopher.procum\\Desktop\\Jobs\\2.txt", "r") as file:

lastLine = file.readline()

for lastLine in file:

pass

print(lastLine)

def func3():

time.sleep(3)

jobA = open("C:\\Users\\christopher.procum\\Desktop\\Jobs\\3.txt", 'r')

fileA = jobA.readline(28)

print (fileA)

time.sleep(6)

#Job state

with open ("C:\\Users\\christopher.procum\\Desktop\\Jobs\\3.txt", "r") as file:

lastLine = file.readline()

for lastLine in file:

pass

print(lastLine)

def func4():

time.sleep(4)

jobA = open("C:\\Users\\christopher.procum\\Desktop\\Jobs\\4.txt", 'r')

fileA = jobA.readline(28)

print (fileA)

time.sleep(6)

#Job state

with open ("C:\\Users\\christopher.procum\\Desktop\\Jobs\\4.txt", "r") as file:

lastLine = file.readline()

for lastLine in file:

pass

print(lastLine)

def func5():

time.sleep(5)

jobA = open("C:\\Users\\christopher.procum\\Desktop\\Jobs\\5.txt", 'r')

fileA = jobA.readline(28)

print (fileA)

time.sleep(9)

#Job state

with open ("C:\\Users\\christopher.procum\\Desktop\\Jobs\\5.txt", "r") as file:

lastLine = file.readline()

for lastLine in file:

pass

print(lastLine)

def func6():

time.sleep(6)

jobA = open("C:\\Users\\christopher.procum\\Desktop\\Jobs\\6.txt", 'r')

fileA = jobA.readline(28)

print (fileA)

time.sleep(6)

#Job state

with open ("C:\\Users\\christopher.procum\\Desktop\\Jobs\\6.txt", "r") as file:

lastLine = file.readline()

for lastLine in file:

pass

print(lastLine)

def func7():

time.sleep(7)

jobA = open("C:\\Users\\christopher.procum\\Desktop\\Jobs\\7.txt", 'r')

fileA = jobA.readline(28)

print (fileA)

time.sleep(6)

#Job state

with open ("C:\\Users\\christopher.procum\\Desktop\\Jobs\\7.txt", "r") as file:

lastLine = file.readline()

for lastLine in file:

pass

print(lastLine)

#Starts each function simultaneousely

Thread1 = threading.Thread(target = func1)

Thread2 = threading.Thread(target = func2)

Thread3 = threading.Thread(target = func3)

Thread4 = threading.Thread(target = func4)

Thread5 = threading.Thread(target = func5)

Thread6 = threading.Thread(target = func6)

Thread7 = threading.Thread(target = func7)

Thread1.start()

Thread2.start()

Thread3.start()

Thread4.start()

Thread5.start()

Thread6.start()

Thread7.start()