Comp 8005 – Assignment #1

PROCESSES VS THREADS (CALCULATING N DIGIS OF PI)

DANIEL KLEIN, JAN 20 2018

Introduction

The purpose of this assignment is to shed light on which is better, processes or threads. This is a topic hotly debated across forums and in classrooms alike. Traditionally, Linux relied on processes, and was optimized as such. Although processes are fast in Linux, they have drawbacks. Threads were introduced as a form of lightweight processes that allowed easier access to data across processes. Threads also have the advantage of not being synchronous, which allows for versatility in processing within an application.

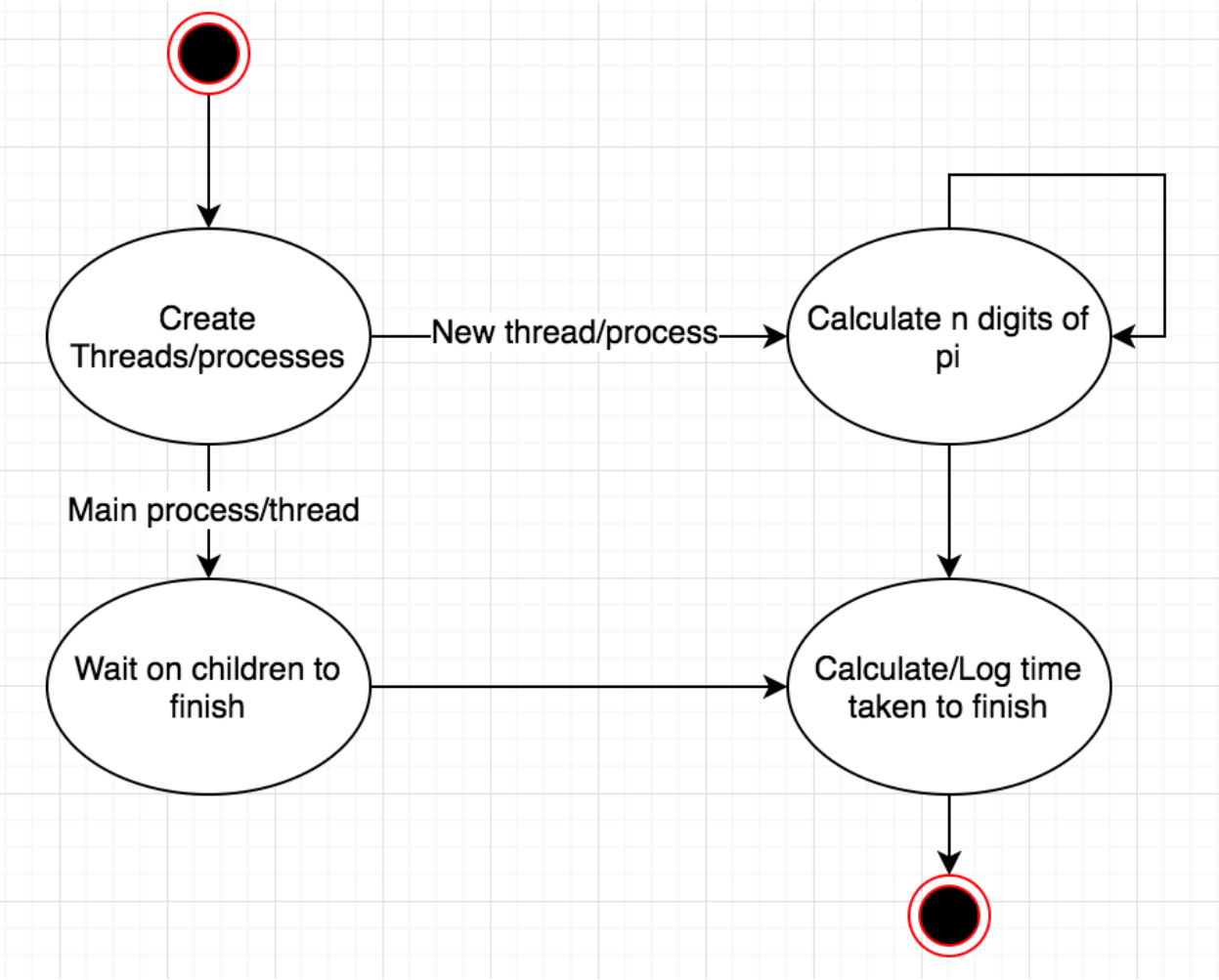
At first, I tried to fall back on my knowledge of Java for this assignment, but I was faced with the reason I had never used processes inside java before: Java doesn't support sub processes. You can spawn Java processes, but it involves spawning a fresh copy of the JVM, and an entirely new copy of the program. Obviously, the data collected in this fashion wouldn't suit the original purpose of this project, because threads are clearly going to perform better in this situation.

Instead, I decided to use a language that is very popular/trendy at the moment. I chose Python. I have little to no experience with Python outside of minor scripting, so I was unsure how the language handled processes and threads internally.

Design

I chose an algorithm that is very processor heavy and scales up in time/difficulty as parameters are increased. I found an algorithm for calculating prime numbers to n digits, and used it in both versions of my program. Instead of dividing the task into tasks and spreading those tasks among the processes/threads, I had each process/thread do its own identical calculation. I chose this approach because it guaranteed each thread/process would do an even amount of work. The load being even makes comparisons much easier.

Both versions off the app use the following design:



Testing/Data

Baseline

In order to test performance of the two applications, I first ran a benchmark of a single threaded application calculating pi to 30,000 digits. These are the results:

```
[Daniels-MBP:ass1 danielklein$ python3 assignment1-threads.py
 (MainThread) starting execution
 (MainThread) time taken 6.175374181009829 seconds
 (MainThread) program time taken 6.175679731008131 seconds
```

Threads

When I ran the version of the program that used 4 threads simultaneously calculating pi to 30,000 digits, I got the following results:

```
[Daniels-MBP:ass1 danielklein$ python3 assignment1-threads.py
 (Thread-1) starting execution
 (Thread-2) starting execution
 (Thread-3) starting execution
 (Thread-4) starting execution
 (Thread-1) time taken 24.68379732297035 seconds
 (Thread-4) time taken 25.021936840959825 seconds
 (Thread-2) time taken 25.085215830011293 seconds
 (Thread-3) time taken 25.145489252987318 seconds
 (MainThread) program time taken 25.17538518900983 seconds
```

Processes

When I ran the version of the program that used 4 processes to calculate pi to the 30,000th digit, I got the following results:

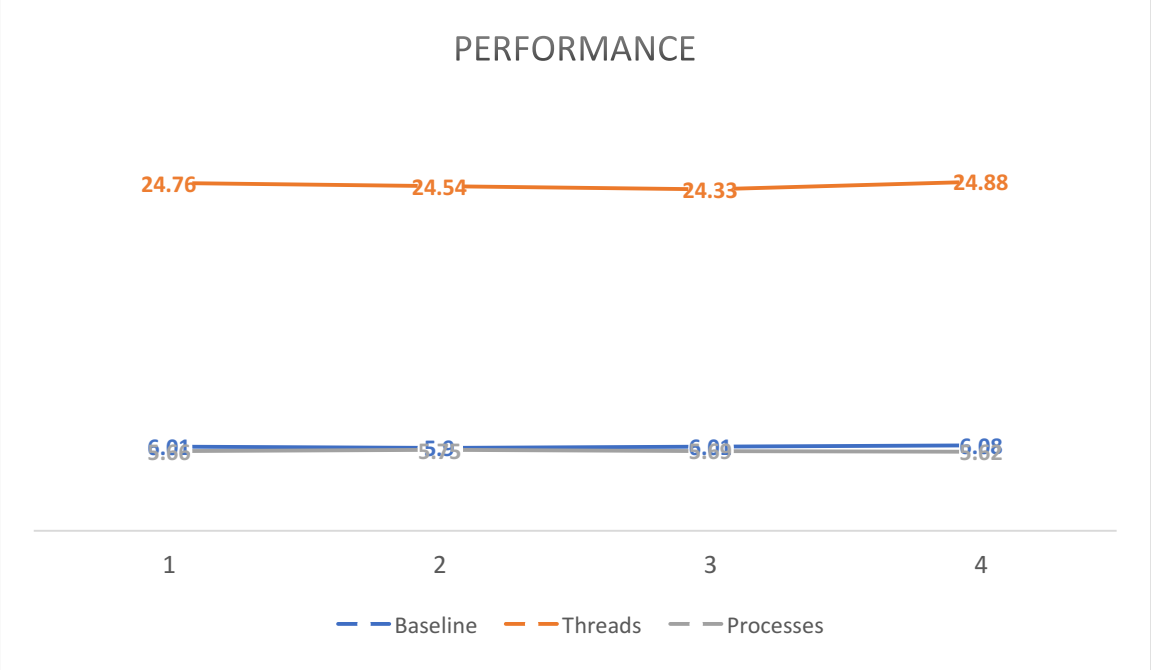
```
[Daniels-MBP:ass1 danielklein$ python3 assignment1-processes.py
 (MainThread) starting execution 18774
 (MainThread) starting execution 18773
 (MainThread) starting execution 18776
 (MainThread) starting execution 18775
 (MainThread) time taken 5.58889283600729 seconds 18774
 (MainThread) time taken 5.590934341016691 seconds 18776
 (MainThread) time taken 5.591956897987984 seconds 18775
 (MainThread) time taken 5.6000760609749705 seconds 18773
 (MainThread) program time taken 5.602256193989888 seconds
```

Figure 1 Note: the number after 'starting execution' is the process number

Trends

I ran each version, including the baseline 5 times and got the following results:

Run	Baseline	Threads	Processes
1	6.01	24.76	5.66
2	5.90	24.54	5.75
3	6.01	24.33	5.69
4	6.08	24.88	5.62
5	5.73	24.56	5.66



Analysis

When I first saw the data, I assumed I had done something wrong. The multithreaded application performed worse than the single threaded application! But, no matter how I created the threads, I was unable to overcome this problem. After some investigation online, I discovered that the core of Python is single threaded! Python supports multiple threads, but only one executes on the processor at a time, so they are essentially running in sequence, as far as performance is concerned.

The processes on the other hand were very fast. They performed at very similar rates to the single threaded application, but were able to calculate 4x as much in the same timeframe. This is mostly because my laptop, being an Apple, is *nix based, so it benefits from the optimized performance of processes.

Conclusion

I didn't answer the problem of which is better, threads or processes. But, I did learn something valuable: Python doesn't handle threads well, and Java doesn't handle processes well. The only conclusion I can draw from this experiment is that sometimes threads are better, and other times processes are better. It's important to know how each language handles each in different situations. A one size fits all approach isn't going to work.