Assignment #1

Processes vs. Threads – Calculating Prime Numbers

Steffen L. Norgren
A00683006

COMP 8005 - Network & Security Applications in Software Dev • BCIT • January 26, 2010

Table of Contents

Overview 3

Design Work 3

- Mathematical & I/O Tasks 3
- Processes 4
- Threads 5

Testing 6

- Processes 7
- Threads 8

Findings 9

Overview

This project served to answer the age old question of which is better, processes or threads. The Linux philosophy towards threads is as such: threads are not meant to be light weight processes, instead they are simply a means in which multiple processes can communicate with each other through shared resources. In the end all the threading libraries end up calling clone() which splits the program in two according to the number of shared resources specified through it's parameters.

However, given this assumption, shouldn't threads be equally as fast as processes? This assignment aims to answer this question through the creation of two identical applications that perform an intense mathematical task. The only difference between both applications is that one will use processes and the other will use threads.

Design Work

Mathematical & I/O Tasks

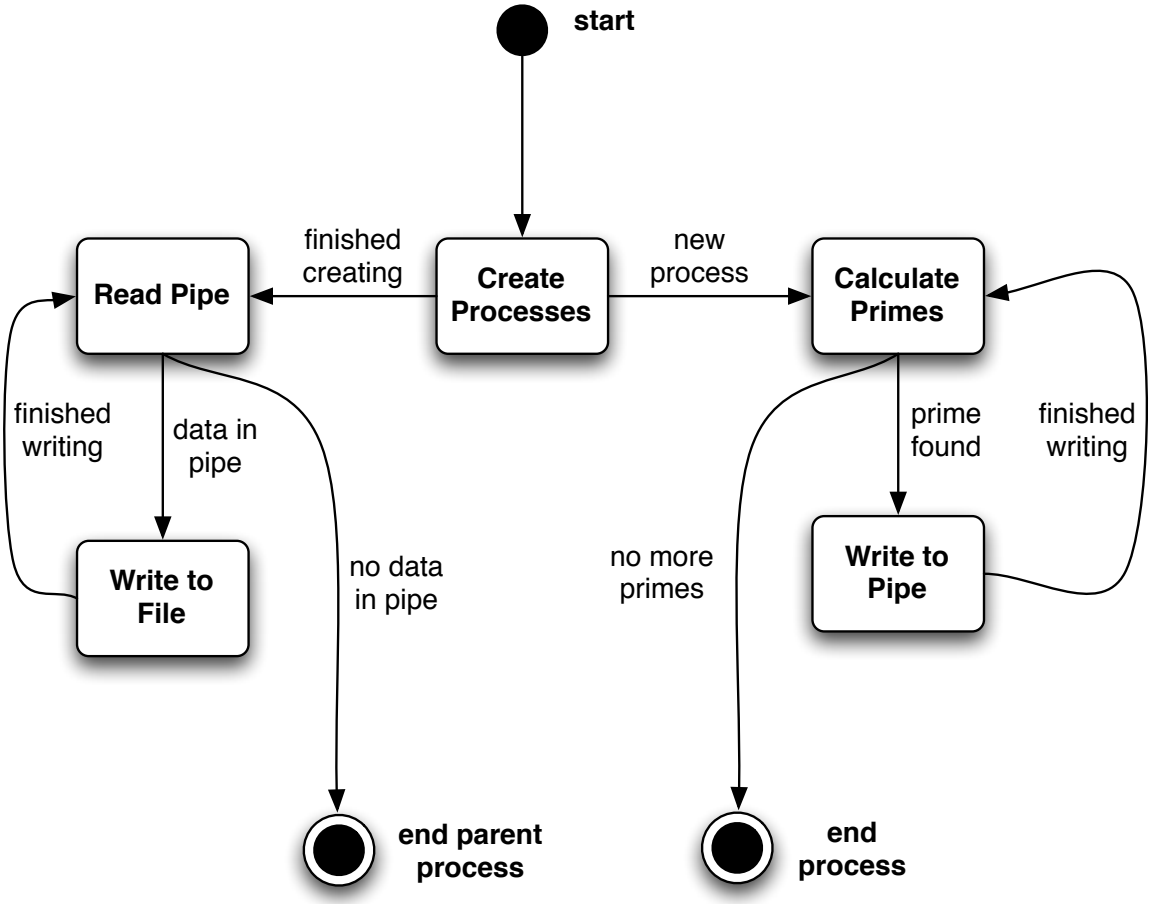
In both the process and threaded versions of the application, they will be calculating and producing lists of prime numbers.

In both cases the parent process/thread will be responsible for all the file I/O along with reading from the named pipe that each process/thread uses to communicate with the parent process/thread. Every time data is read from the named pipe, the results are output to a file.

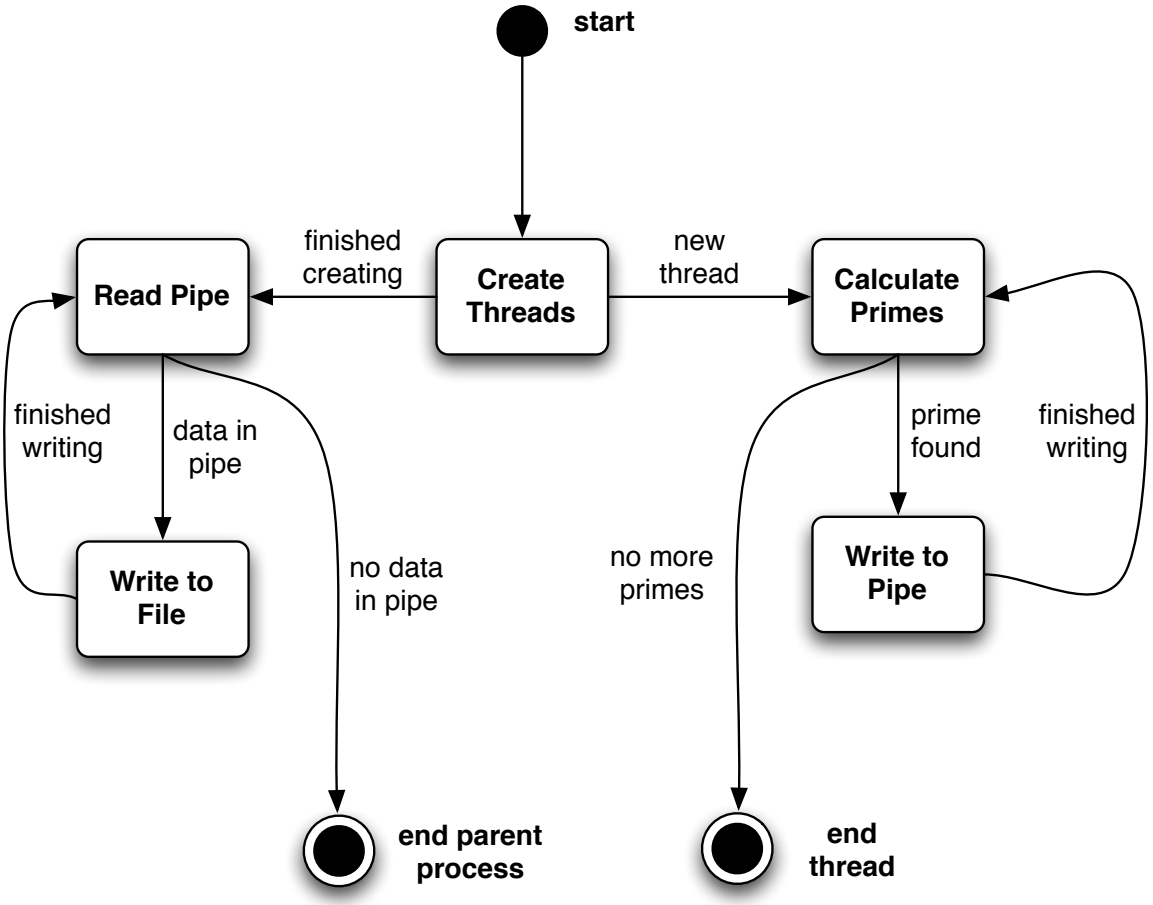
The reason I chose calculating prime numbers was because this task could easily be subdivided into chunks for each process/thread to handle. For example, we could assign blocks of 1,000,000 numbers for each process/thread to calculate the primes for. The first process/thread would deal with 1 – 1,000,000 and the second would deal with 1,000,001 – 2,000,000 and so on. This method guarantees that each process/thread have generally even work loads, making comparison between each process and thread valid.

Processes

The diagrams for both the processes and threads versions of the application are pretty much identical, but I will include them nonetheless.



Threads


start
Create Threads
finished creating
new thread
Read Pipe
Calculate Primes
finished writing
data in pipe
no data in pipe
prime found
finished writing
no more primes
Write to File
Write to Pipe
end parent process
end thread

TESTING

Testing was done using various different process/thread counts as well as varying loads (processing blocks) for each process/thread. While testing each method, I verified that the correct number of processes/threads appeared in *top*. Additionally, when analyzing the data for each case, I verified that the same number of results were produced using each method.

Considering the large number of prime numbers (I used processing blocks of 10,000,000), I had to limit the programs output to the disk, otherwise each file would approach nearly 1GB of data. As such, each process/thread only writes each 1,000th result to the named pipe. This way I could ensure a fairly long running time, but also have a manageable data set to analyse.

Additionally, because there are more prime numbers between 1 and 10,000,000, I set the initial conditions such that only primes from 10,000,001 and up are searched for. That is, each process/thread got a range of 10,000,000 to search through.

Processes

Below is an example test run for using processes.

```
> ./process -o process.csv -p 5 -b 10000000 -s 10000001
Using the Following Options: (For help use "process -h")
  Number of Child Processes:   5
  Output to File:              process.csv
  Start Listing Primes From:   10000001
  Block Size for each Process: 10000000

Child pid 1081 starting on range 10000001 - 20000001
Child pid 1082 starting on range 20000001 - 30000001
Child pid 1083 starting on range 30000001 - 40000001
Parent pid 1080 writing output to file process.csv
Child pid 1084 starting on range 40000001 - 50000001
Child pid 1085 starting on range 50000001 - 60000001
Child pid 1081 finished range 10000001 - 20000001
Child pid 1082 finished range 20000001 - 30000001
Child pid 1083 finished range 30000001 - 40000001
Child pid 1084 finished range 40000001 - 50000001
Child pid 1085 finished range 50000001 - 60000001
Parent pid 1080 finished writing to file process.csv
```

PID 1061–1064 shows that 6 processes are running, 5 worker processes and 1 parent.

```
PID   COMMAND      %CPU TIME     #TH
1068  screencaptur 0.5  00:00.07 2
1064  process      28.2 00:07.98 1/1
1063  process      28.5 00:08.06 1/1
1062  process      32.1 00:07.87 1/1
1061  process      30.2 00:08.00 1/1
1060  process      27.6 00:08.12 1/1
1059  process      23.9 00:06.18 1/1
```

Threads

Below is an example test run for using threads.

```
> ./thread -o thread.csv -t 5 -b 10000000 -s 10000001
Using the Following Options: (For help use "thread -h")
  Number of Threads:  5
  Output to File:  thread.csv
  Start Listing Primes From:  10000001
  Block Size for each Thread:  10000000

Writing output to file thread.csv
Thread 0 starting on range 10000001 - 20000001
Thread 1 starting on range 20000001 - 30000001
Thread 2 starting on range 30000001 - 40000001
Thread 3 starting on range 40000001 - 50000001
Thread 4 starting on range 50000001 - 60000001
Thread 0 finished range 10000001 - 20000001
Thread 1 finished range 20000001 - 30000001
Thread 2 finished range 30000001 - 40000001
Thread 3 finished range 40000001 - 50000001
Thread 4 finished range 50000001 - 60000001
Finished writing to file thread.csv
```

PID 1110 shows that 6 threads are running, 5 worker threads and 1 parent.

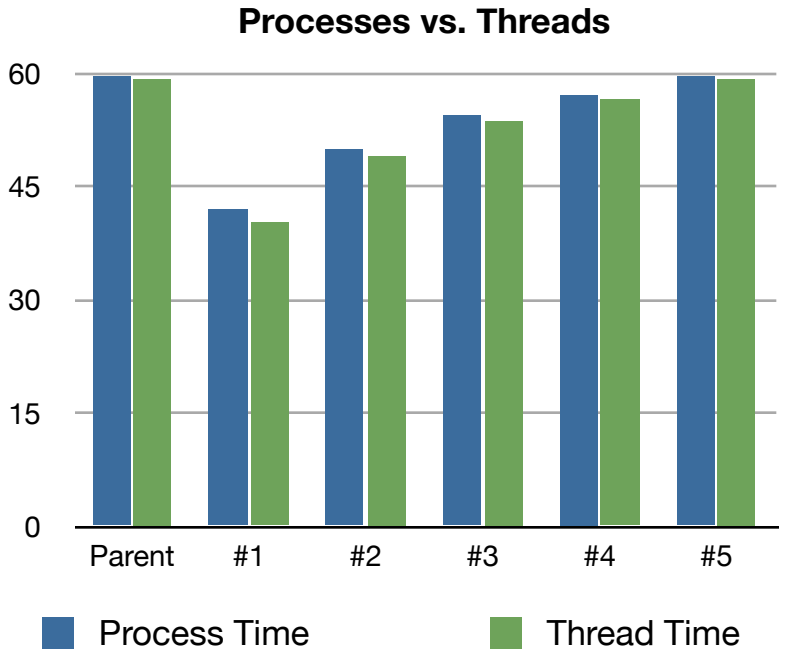
```
PID   COMMAND      %CPU  TIME     #TH
1113  screencaptur 0.0   00:00.07 2
1111  taskgated    0.0   00:00.00 2
1110  thread       186.3 00:29.13 6/6
1091  Xcode        0.0   00:09.87 5
1049  top          4.4   00:16.73 1/1
```

FINDINGS

Through analyzing the output data, I found that there is a very marginal difference between the performance of threads and processes. Threads managed to beat out processes with a 1.43% improvement in speed overall.

Process / Thread	Process Time	Thread Time	Process Memory	Thread Memory
Parent	59.55429983	59.24739981	360448	450560
#1	41.96093011	40.2499299	192512	450560
#2	49.92145014	49.04545999	192512	450560
#3	54.43161011	53.65022016	192512	450560
#4	57.10788012	56.49676991	192512	450560
#5	59.56673002	59.24427986	192512	450560

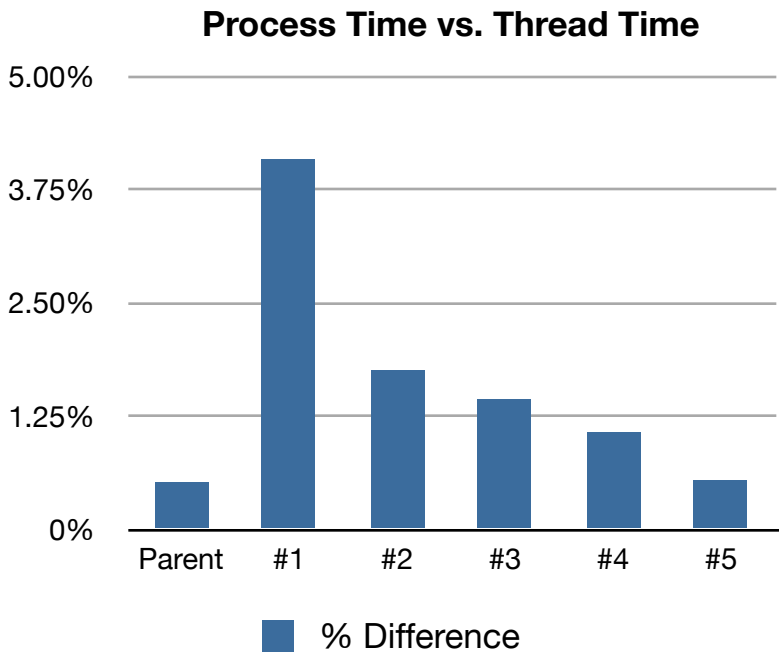
One major difference between the processes and threads in this example is memory usage. Where the threaded version only used 450K total, the process method used a total of 1,323K of memory. This is more than twice as much as the memory usage for the threaded version as the process version makes copies of the entire application including variables whereas the threaded version does not.



One can see that the completion time between each process and thread is very slight, with the biggest difference being with the process/thread that was created first.

Process / Thread	Time Difference	% Difference
Parent	0.30690002	0.52%
#1	1.71100021	4.08%
#2	0.87599015	1.75%
#3	0.78138995	1.44%
#4	0.61111021	1.07%
#5	0.32245016	0.54%
Total	**4.6088407**	**1.43%**

Here we can see the overall difference in completion times between each process and thread. The differences are slight, but with the first thread beating out the first process by a larger margin. This is likely due to how a dual core CPU schedules the thread and process priority.



Overall, it appears that (at least in this example) threads beat out processes in performance, albeit by a small margin. However, where threads do best is in reducing the overall memory footprint of an application. Therefor if memory is of a concern, using threads instead of processes would be a better option.