8005 Assignment 1

Design & Documentation

Eric Wu, A00961904

BTech 2020 Winter

Table of contents

Objectives

Use multiple processes and threads on either the Windows or Linux operating systems and measure the performance and efficiency of each mechanism.

Approach

Two programs will be used. Both programs will have the same number of workers to perform a series of basic prime number factorization.

One program uses fork() only to create workers, while the other one uses threading only. Both programs will use the exact same algorithm and accepts the exact same argument. The required computation for each of the program will be output to the console to see the overall differences in performance.

Both programs will have an option to write result of factorization and total computation time to a file, which helps to see if IO activity has any significant influence over the performance of any of the two programs.

Testing platform

OS : Linux Fedora
CPU(s): 4

Program usage

Name: p_main
$ ./p_main <number of workers> <number of tasks per worker> <starting number to factorize> [-w output filename]

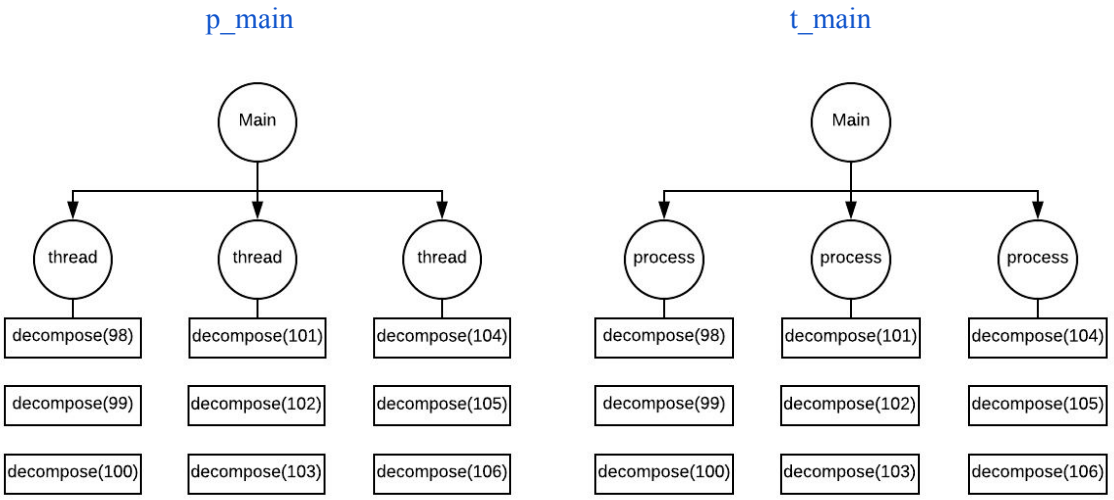
Sample:
./p_main 5 10 1000 -w result.txt
(program will evenly distribute range of numbers from 1000 to 10 to 5 workers, the result of factorization will be written to a file named result.txt)

Name: t_main
$ ./p_main <number of workers> <number of tasks per worker> <starting number to factorize> [-w output filename]

Sample:
./t_main 5 10 1000 -w result.txt
(program will evenly distribute range of numbers from 1000 to 1049 to 5 workers, the result of factorization will be written to a file named result.txt)

Program Design

Both programs (p_main, t_main) will spawn a number of workers in a fan type of architecture.
In p_main, the workers are spawned with the fork().
In t_main, the workers are spawned with the pthread_create().
Below figure is a simple demonstration of the program architecture.



Test cases

Test set for both program

Case 1 ~ 5 is to test the scalability of the two programs. Each case will have the same number of tasks per worker. IO is disable for now.

Case 6 ~ 10 is to test the rate of deterioration in efficiency per each worker in response to an increase in workload.. IO is disable for now.

Case 11 ~ 15 is the same as 1 ~ 5, except that the IO is enabled now. This is see change of efficiency in response to IO activity for both programs.

Case 16 ~ 20 is the same as 6 ~ 10, except that the IO is enabled now. This is see change of efficiency in response to IO activity for both programs.

	Worker number of workers	**Task per worker**	**Range of number To factorize**	**IO**
Case 1	5	10	87654321 ~ 87654370	disable
Case 2	10	10	87654321 ~ 87654420	disable
Case 3	15	10	87654321 ~ 87654470	disable

Case 4	20	10	87654321 ~ 87654520	disable
Case 5	25	10	87654321 ~ 87654570	disable

	Worker number of workers	**Task per worker**	**Range of number To factorize**	**IO**
Case 6	5	10	87654321 ~ 87654370	disable
Case 7	5	20	87654321 ~ 87654520	disable
Case 8	5	30	87654321 ~ 87654470	disable
Case 9	5	40	87654321 ~ 87654520	disable
Case 10	5	50	87654321 ~ 87654570	disable

	Worker number of workers	**Task per worker**	**Range of number To factorize**	**IO**
Case 11	5	10	87654321 ~ 87654370	enable
Case 12	10	10	87654321 ~ 87654420	enable
Case 13	15	10	87654321 ~ 87654470	enable
Case 14	20	10	87654321 ~ 87654520	enable
Case 15	25	10	87654321 ~ 87654570	enable

	Worker number of workers	**Task per worker**	**Range of number To factorize**	**IO**
Case 16	5	10	87654321 ~ 87654370	enable
Case 17	5	20	87654321 ~ 87654520	enable
Case 18	5	30	87654321 ~ 87654470	enable
Case 19	5	40	87654321 ~ 87654520	enable
Case 20	5	50	87654321 ~ 87654570	enable

Test result

*IO disabled

	p_main (msec)	t_main (msec)	ratio (p_main/t_main)
Case 1	17141036	17306268	0.99
Case 2	82823107	84284740	0.98
Case 3	112568948	111975338	1.00
Case 4	216510397	217420986	0.99
Case 5	424890936	430843820	0.98
avg			**0.98**

	t_main (msec)	t_main (msec)	Ratio (p_main/t_main)
Case 6	17171371	17299659	0.99
Case 7	49588032	49922916	0.99
Case 8	51290374	51451773	0.99
Case 9	84808656	84562471	1.00
Case 10	109042084	108274505	1.00
avg			**0.99**

*IO enable

	p_main (msec)	t_main (msec)	ratio (p_main/t_main)
Case 11	17869452	17863002	1.00
Case 12	87013741	87133341	0.99
Case 13	124870321	110318826	1.13
Case 14	238019712	240969910	0.98
Case 15	459233128	431148434	1.06
avg			**1.03**

	p_main (msec)	t_main (msec)	ratio (p_main/t_main)
Case 16	18482233	18652154	0.99
Case 17	52210372	44523421	1.17
Case 18	55354918	55048418	1.00
Case 19	90675881	90535726	1.00
Case 20	115966087	109189037	1.06
avg			**1.04**

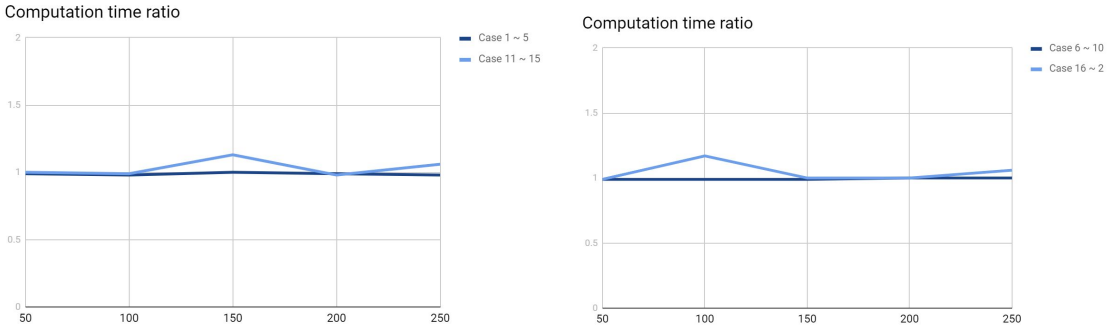
Analysis

The testing platform we are using have 4 cores. We can see that the performance is much better when we keep the number of workers close to 4 (5 in this case). Consider case 5 and 10, both cases requires the program to factorize number from *87654321 ~ 87654570*, the performance for case 5 is 4 times better than case 10. This is true for both p_main and t_main.

Both p_main and t_main yield similar efficiency after computing same data. From the computation time ratio between p_main and t_main, we see that the number is fairly close to **1**, which indicates that both programs used up almost the same amount of time to process the numbers.

The computation time ratio remains close to **1** even if we increase the number of workers, increase the number of tasks per worker, or even introduce IO activity. Below figure shows the computation time ratio between both programs in each of the cases.

*For case **1 ~5** (without IO) & 11 ~ 15 (with IO) * For case **6 ~ 10** (without IO) & 16 ~ 20 (with IO)



In both figures, the ratio remain within the range of 0.25 and 1.25.

If we look at the computation time closely, p_main does use a little more time than t_main overall. This is because creation of process is little bit longer due to copying table and copy-on-write memory mapping in memory. However, the difference is negligible.

Conclusion

Based on the analysis, the behaviour is expected, since in linux, threads are implemented as standard processes. Even if creation of process is longer than the creation of thread, the difference is almost negligible. One thing to consider is that the program we use does not consider IPC, which is the major difference between thread and process. The rule of thumb is that, if we are choosing process over thread, our primary focus should be data isolation/protection between the workers. If we are choosing thread over process, our focus should be on communication between the workers. If all these factors disregarded, picking process or thread is pretty much just personal preference.