Analysis of scheduler designs in multithreaded workloads

Researcher: Darsh Manoj . *Supervised by* Dr Susmit Sarkar, University of St Andrews School of Computer Science

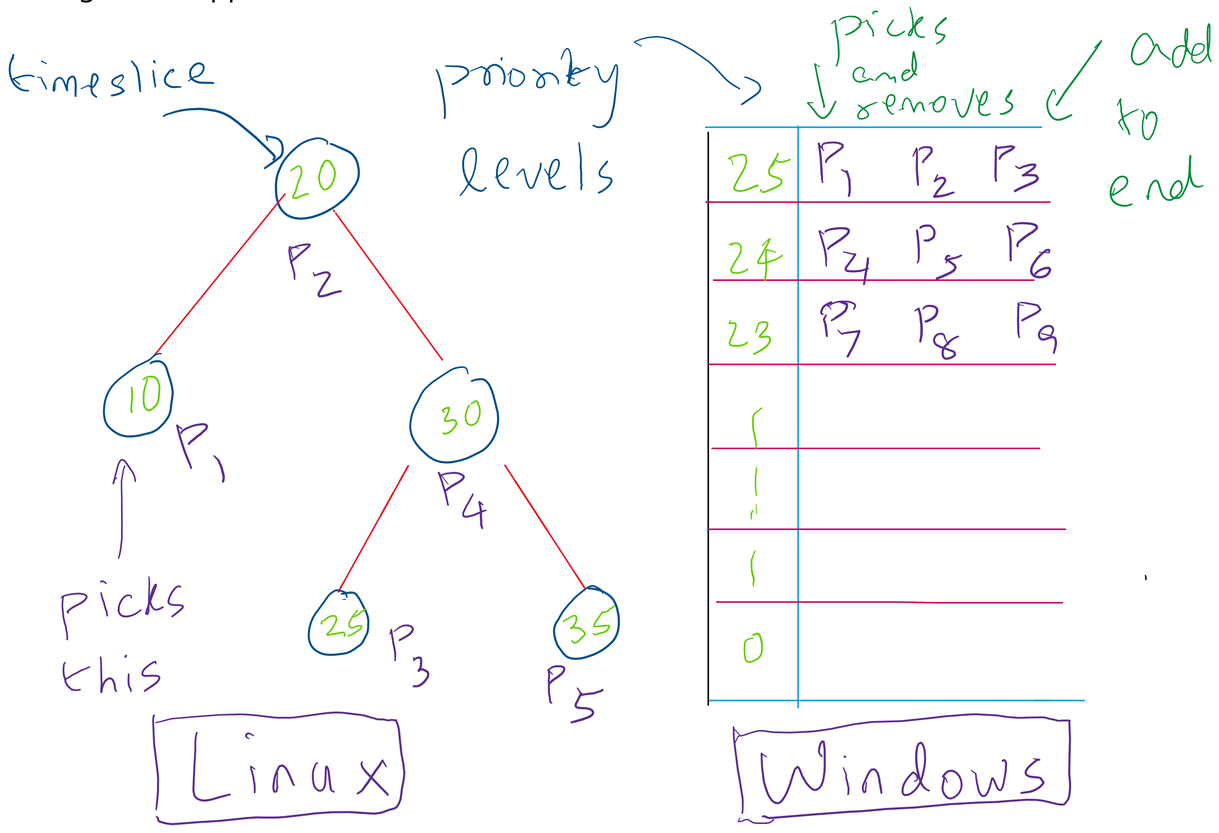
Introduction

The scheduler is a crucial part of any modern operating system that decides *what* runs in the computer, *when* and how long. With today's CPUs having multiple cores, and the fact that thousands of applications could be running in the system at any given time, the role of the scheduler would only become more important. It is perhaps even more striking that two major operating systems – Linux and Windows – employ disparate scheduling designs. Here, we test the performance of these operating systems under various conditions to establish how they respond and whether any differences can be seen, and where.

The scheduler designs

Since v2.6.21, Linux uses a CFS (completely fair scheduler) design, and its aim is to keep things *fair* – that is, each process should get what it needs and no process should be unfairly advantaged (that is, get too much CPU time), or disadvantaged (that is, get kicked out quickly). This is achieved through the use of *red-black trees* [1][2] – a data structure that sorts values automatically (here by timeslice). However, this means that the time complexity of this design is $O(\log n)$ compared to the constant-time dequeue design that was prevalent in older v2.6 releases (due to the need to sort every time).

Windows, since the 1990s, use a multi-level feedback queue – essentially it keeps an array of possible processor priorities, and wires a dequeue (double ended queue) to each of the processor priorities, with the dequeues only performing constant-time add/remove operations. Hence the theoretical time complexity is $O(1)$ – that is, no matter the number of processes, it should take around the same amount of time. macOS also uses a similar high-level approach.



Methodology

We used a program created by the author that finds the first *k* Pythagorean triplets in a loop, and run it five times for accuracy. The program was modified to support multiple-threads, and a pair of virtual machines (one using Ubuntu 18.04, other with Windows Server 2016) were used to run the benchmarks under various, controlled test cases, and the results were then generated by the program and averaged over three separate runs, with the main variable being the time taken to complete the entire benchmark.

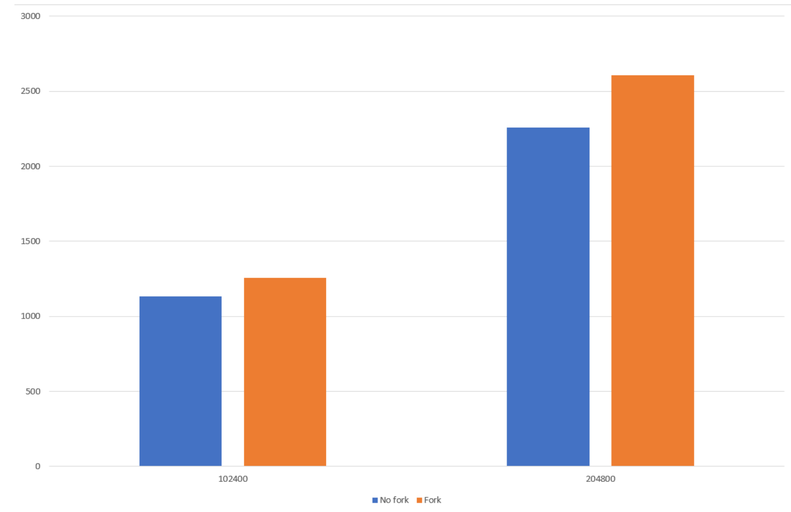
Many types of test cases were constructed. Some examples include:

- Simulating a thread pool so that each thread will run non-stop rather than having to context-switch (stop and restart) every time (which is expensive)
- Modifying the benchmark so that it runs for more threads but with each run taking far fewer – this is because it would otherwise take way too long to complete
- Forking: essentially rather than using one process to run all *N* threads, we are using *N* processes that run one instance. This is typically costlier due to the extra start-up cost, as each program has to start up and from scratch.
- Testing a miniature I-O bound scenario by making the processes sleep for a defined period of time – this also stresses the runqueue because more threads will be in the queue (that is, have not exited).
- (For Linux only) Switching the schedulers [3] (from CFS to FIFO -first in first out - to more). Windows does not support changing the entire scheduler model. This is important because some scheduler options (`SCHED_IDLE` for example) are meant specifically for low-priority tasks (i.e, being slower) – will that be shown in the results?

Results and analysis

For a standard benchmark run of up to 3200 threads, we see that, both operating systems take around the same amount of time. When we simulate a *thread pool* – that is reusing the same thread rather than creating new ones all the time - to reduce the effects of context-switching, the two operating systems still perform similarly.

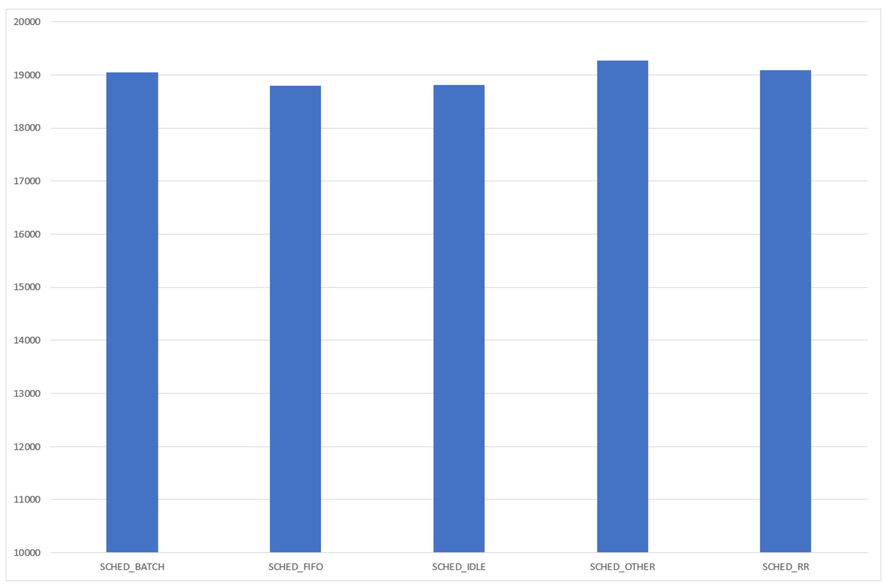
However, we start to see a performance difference when *forking* – that is creating many copies of the program itself – is done instead of using the same program many times – is done – with a up to 15% loss observed. However the differences between operating systems is still negligible.



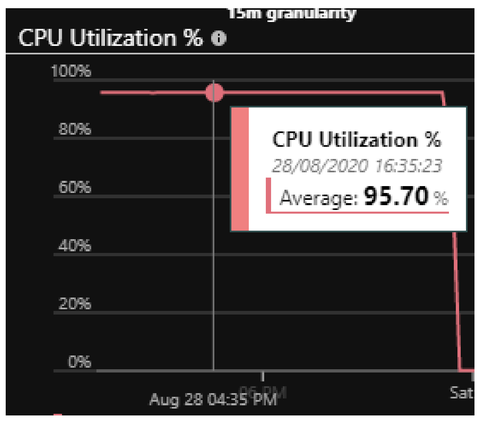
Time taken to run 102400 threads – blue = without forking – orange = with forking

When the threads are made to go to sleep (so that they do not exit quickly), we can see a measurable difference of up to 10%. This suggests that the CFS runqueue, under very large cases, could take longer to process compared to a constant-time variant. However, the performance did not vary otherwise.

In the final case, where the scheduler was manually changed, there were only minor differences in the time taken between each of the schedulers (the below is for 204800 threads):



However, we can see an odd phenomenon with some scheduler options (like FIFO) – that is – about 4% of CPU space is unused as in the below example. While minor, it could suggest that we could optimise it to become even faster (by making it use all 100% of the CPU)



Conclusions and next steps

The similarly in performance between the two schedulers under most test cases suggest that the overhead by the scheduler itself is minimal. However, when the scheduler is stressed much beyond what a normal home user would run, we can see that CFS is a bit slower in some cases, and we can also see that the effect of forking causes a noticeable performance penalty. When we change the schedulers on Linux, the difference in performance is minimal despite the fact that they employ vastly different mechanisms, though some options do not make full use of the CPU, which suggests that improvements are still possible. This research just scratches the research involved in scheduler design and it is expected that its importance will only increase as the world gets more multithreaded.

There is a lot yet to be explored – for instance how the use of NUMA (non-uniform memory access) modes in processors with high core count can impact performance – even artificially. And there are a lot of other test cases that need to be investigated – for instance the research focused on an artificial benchmark, not real applications.

LAIDLAW SCHOLARS

University of St Andrews

References: https://www.eit.lth.se/fileadmin/eit/courses/eitf60/Rapporter/Ludwig_Hellgren_Winblad_4547_assignsubmission_file_ludwig.hellgren.winblad.pdf (1), https://opensource.com/article/19/2/fair-scheduling-linux (2) and https://man7.org/linux/man-pages/man7/sched.7.html (3). Picture, images and graphs are from the researcher.

Acknowledgements: The researcher would like to thank their supervisor Dr Susmit Sarkar for their help in the research, Microsoft for graciously providing the Azure credits used for this research, and the Laidlaw Foundation (and Lord Laidlaw) for the unique opportunity to conduct two useful summers of research.

V1.0, RTM. 13/9/2020