William Turner
ENCE360: Operating Systems
October 9, 2025

Effect of Multi-Threaded Execution on Process Running Times

Serial Execution

Table 1 summarizes the execution times of the original `serial.c` program on the lab machines. **Real time** represents the total wall-clock time for program execution, **User time** is the CPU time spent during execution, and **Kernel time** (Sys) is the time spent on kernel/system calls. All values in this report are averaged over 10 runs.

	Time (s)	σ (s)
real	53.064	0.262
user	53.051	0.268
sys	0.003	0.007

TABLE 1. Process Execution Times for Single-Threaded Program

The test suite contained 50 integration problems that took 53.064 s in **Real time** to complete in serial execution. The findings show that the execution times for the `serial.c` program are highly consistent across trials. Note the low **Kernel time**, this is due to the low CPU overhead involved in running a single process. The `time` command reports to 2 decimal places, and many trials reported 0.00s.

Multi-Threaded Execution

The implementation of multi-threaded execution in the `thread.c` program had minimal impact on the execution speed of the program when run with a singular thread. **Real time** for execution of `thread.c` fell within 0.87% of the original `serial.c` program, with a lower standard deviation.

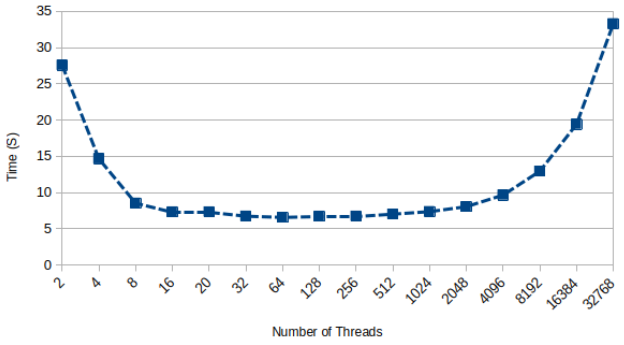

FIGURE 1. **Real time** for execution with various thread counts.

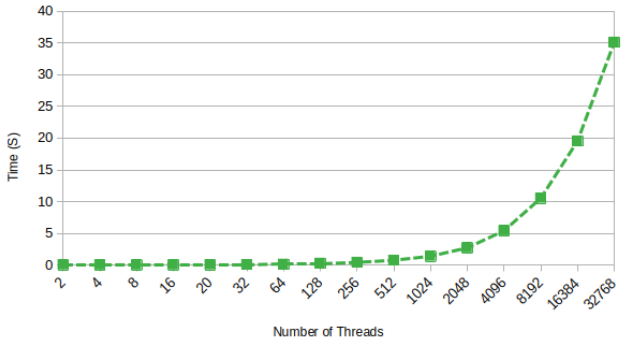

FIGURE 2. **Kernel time** for execution with various thread counts.

[Figure 1] shows the effect of execution with multiple threads for the program `thread.c`. We can see that execution with 2 threads yielded a 48.09% decrease in real execution time over serial execution, and a further 46.82% decrease when doubling the thread count again to 4. Real execution times decrease roughly logarithmically, indicating diminishing returns with increasing thread counts as explained in [1] (p 1). Execution with 64 threads yielded the lowest execution time (6.562 s), although it is possible that another untested number of threads is optimal. We also note an increase in **Kernel time** from serial

execution to multi-threaded execution with 64 threads of +3733.33% (0.003 s to 0.115 s). This is expected in multi-threaded execution as the CPU must perform additional system calls to manage context switches and parallel threads.

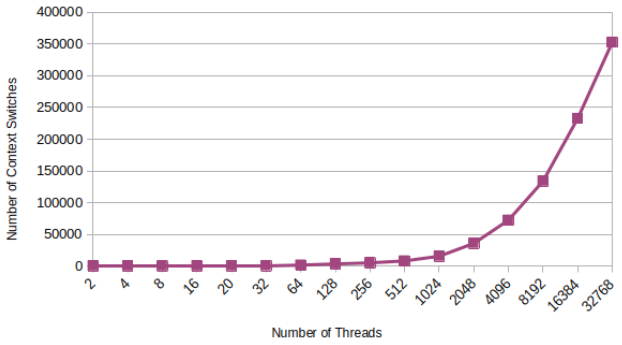

FIGURE 3. Number of voluntary context switches made during execution for various thread counts.

As a demonstration in **Cache thrashing**, the program was run with increasing numbers of threads. It is important to note that the Intel® Core™ i7-12700 CPU on which the trials were run has 20 threads. We can see from [Figure 1] that increasing the thread count past 64 results in an increase in real execution times. One of the reasons for this is the large number of **Sys calls** being made, resulting in a high **Kernel time**. At a thread count of 4096, the kernel time alone (5.408 s) is almost the length of the real execution time when run with 64 threads (6.562 s). This is due in part to **Cache thrashing** in which the values stored in the CPU cache are constantly being overwritten when the context is switched [2]. This negates the benefits of caching and becomes a negative influence on real execution time.

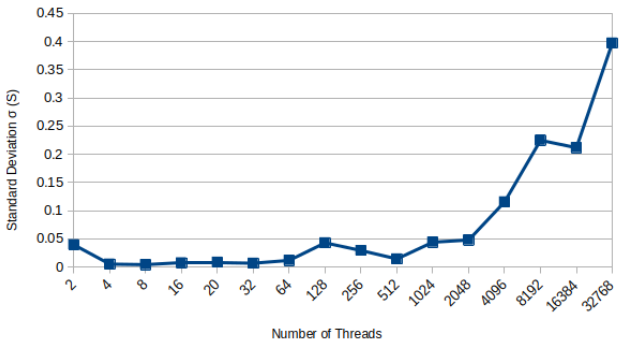

FIGURE 4. Standard deviation of **Kernel time** for execution with various thread counts. (S)

It is interesting to note that as the number of threads increases, the variability of the **Kernel time** between trials also increases [Figure 4]. As the program becomes more dependent on the OS scheduler, its **Kernel time** is increasingly affected by random external factors such as other background tasks that the program shares the CPU with [3].

Note the large increase in **User time** for program execution on 20 threads. In all honesty, I do not know why this is the case. My guess is that the operating system likely throttles the number of threads when run with 99999, meaning we don't see a dramatic spike like with 20 threads, and thus making our data appear how it does.

Process Execution

	Time (s)	σ (s)
real	14.802	0.362
user	53.841	0.975
sys	0.229	0.178

TABLE 2. 4 Child Processes

	Time (s)	σ (s)
real	14.539	0.082
user	52.975	0.173
sys	0.014	0.011

TABLE 3. 10 Child Processes

The implementation of multi-process execution decreased the **Real time** significantly, with a maximum of 4 child processes providing a 72.1% decrease over serial execution from 53.064 s to 14.802 s.

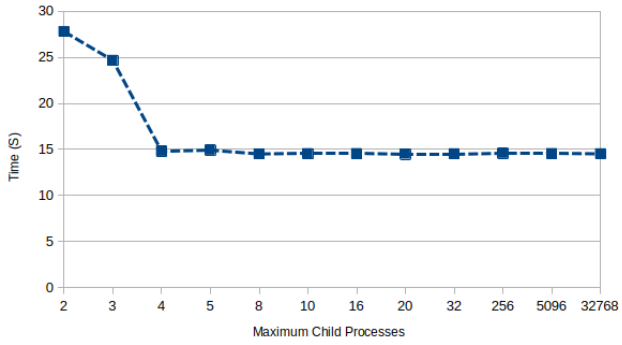

FIGURE 5. Real execution time for various maximum child counts.

Increasing the number of child processes past 4 did not have a significant effect on execution times [Figure 5].

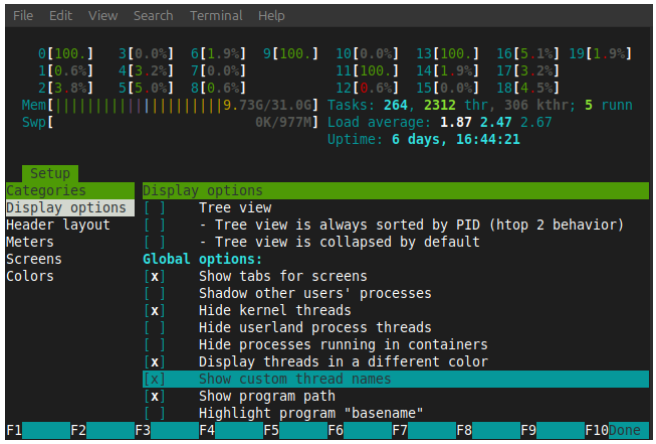

FIGURE 6. Active threads during Process Execution.

Running the `htop` command during execution with 10 maximum children produces the data shown in [Figure 6], which shows thread utilization. From this we can see that despite permitting up to 10 or more child processes in the program; there is a limit enforced by the OS or otherwise that restricts execution to 4 threads. This explains the behavior in [Figure 5].

Process-Thread Execution

Combining the programs `thread.c` and `process.c` to implement multi-threaded processes in `processThread.c` did not yield any significant benefit over the original `thread.c` program.

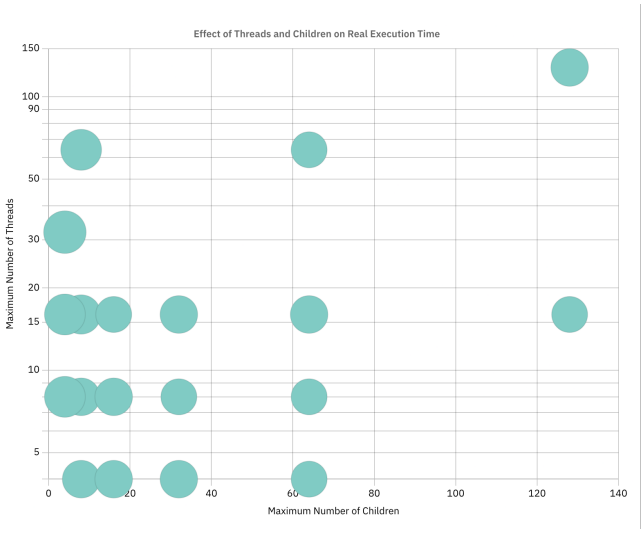

FIGURE 7. Active threads during Process Execution.

[Figure 7] shows the effect of varying both the maximum number of child processes, and maximum number of threads permitted during the execution of the `processThread.c` program. The radius correlates to the **Real time** to complete the test suite. The average across all trials was 6.726 s, with a standard deviation of 0.329 s, indicating varying the child and process counts had little effect.

While there was no significant decrease in **Real time** over 64-threaded execution, the addition of multiple processes allowed the thread count to be relaxed with performance maintained. Execution with 4 children and 8 threads yielded similar results as 64-threaded execution in the `thread.c` program. This outcome suggests that the program's workload is well-suited for parallelization across multiple processes, each handling a subset of tasks with fewer threads. By distributing the workload among multiple processes, the system can better utilize available CPU cores, reducing the overhead associated with managing a large number of threads within a single process. This approach can mitigate issues such as thread contention and excessive context switching, which often arise in highly threaded environments. Therefore, employing multiple processes with a moderate number of threads per process can achieve performance comparable to a single process with a high thread count, while potentially offering better scalability and resource utilization.

REFERENCES

[1] D. Dice and A. Kogan, "Avoiding scalability collapse by restricting concurrency," *arXiv preprint arXiv:1905.10818*, May 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1905.10818

[2] P. J. Denning, "Thrashing: Its causes and prevention," in *Proceedings of the AFIPS Fall Joint Computer Conference*, vol. 33, 1968, pp. 915–922, [Online]. Available: https://cs.uwaterloo.ca/~brecht/courses/702/Possible-Readings/vm-and-gc/thrashing-denning-afips-1968.pdf.

[3] A. R. Alameldeen and D. A. Wood, "Variability in architectural simulations of multi-threaded workloads," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA-9)*, Anaheim, CA, USA, Feb. 2003, pp. 7–18. [Online]. Available: https://research.cs.wisc.edu/multifacet/papers/hpca03_variability.pdf