Project 2 Report - Trevor Olson, Tobias DiRito

System Information

OS - Windows 11
CPU - 12th Gen Intel(R) Core(TM) i7-1260P (12 core)
RAM - 16 GB DDR4
Language/Runtime: C (GCC 11.4.0) utilizing POSIX pthreads
Build Instructions: Run in WSL, execute “cc -O2 -Wall -Wextra -pedantic -pthread cs440project2.c -o project2” to compile and “./project2” to execute.

Methodology & Implementation

The goal of this program was to create and destroy 5000 threads using 3 different organizational structures, utilizing the POSIX pthreads library in C. To safely manage memory and arguments across so many operations, we used malloc() to create arrays for the thread handles “pthread_t.” In order to ensure we created and destroyed exactly 5,000 threads, we used atomic variables which incremented creation of global variable counters without running into race conditions as discussed in class.

- Experiment A: Flat- The main thread creates 5000 worker threads in a single for loop, calling the pthread_create() function. It then entered a second for loop which iterated backwards calling the pthread_join() function on each thread, effectively pausing the main thread until the worker thread increments our atomic “destruction counter” and terminates the thread.
- Experiment B: Two level - The main thread creates 50 parent threads, and then each parent subsequently executes parent_worker_2b_no_batching containing a loop that calls pthread_create() 99 times to create the children. To destroy the threads, each parent executes a loop called pthread_join(), letting its 99 children finish. Once all children have been joined and destroyed, it increments the destruction counter and terminates itself. At the same time, the main thread is joining and destroying all 50 parents.
- Experiment C: Three level - The main thread creates 20 initial threads, each creates 3 children, and each child creates 82 grandchildren, the system then stores their ids and joins them. To maintain the format of parent->child->grandchild, the code uses custom structs (initial_arg_t and child_arg_t), and passes them down the generations of threads. The main creates 20 initial parents in a loop, which then calls “initial_worker_2c_no_batching” for every parent to create 3 children. The loop that creates the 3 children also calls “grandchild_worker_2c” for each child to

create 82 grandchildren. This all runs simultaneously simulating concurrent multithreading as seen in our output. To destroy the threads, each higher level thread must wait until all descendants have been terminated before terminating themselves (which would otherwise create orphans), so the child threads join and terminate their grandchildren, and so on up the chain until the parents have been joined and terminated.

For our implementation we used unbatched logic, both of our machines were more than capable of completing the process unbatched so batching was unnecessary.

	Trial 1	Trial 2	Trial 3	Average
Flat	235.107 ms	216.505 ms	213.531 ms	221.714 ms
Two Level	278.997 ms	273.365 ms	275.587 ms	275.983 ms
Three Level	268.731 ms	277.773 ms	253.128 ms	266.544 ms

Analysis:

The results of this experiment show that the organizational structure of a multithreaded application significantly impacts performance even when the workload remains constant. Comparing the performance of the different methods, flat, two level and three level, you can see that flat was the fastest at 221.714 ms on average. This makes sense because it involves minimal management overhead; it just creates threads and destroys them. The main thread communicates directly with the kernel to spawn workers ensuring that the CPU's instruction cache remains full with thread creation logic. Interestingly, the three level hierarchy outperformed the two levels with averages of 266.544 and 275.983 respectively. While the deeper hierarchy usually implies more overhead these results highlight the impact of thread contention. The slower performance of the two level method can be attributed to the higher number of managing threads active at once. With 50 parent threads all attempting to call pthread_create at the same time to spawn 99 children the system likely experienced significant contention within the OS kernel and the C library's memory allocator. Alternatively, the three level tests with only 20 initial threads performed better in reaching 5000 threads because the creation process was more staggered. By reducing the number of top level threads competing for cpu time the system could manage the burst of grandchildren creation more efficiently leading to the improvement over the two level method. This experiment's output demonstrated the non deterministic nature of the POSIX thread scheduler. Threads were rarely joined in the same order they were created. This occurs because the Linux Completely Fair Scheduler assigns threads to

different cores based on load and priority. The distribution of the 5000 threads across the hardware means that a threads completion time is influenced by many factors such as cache hits, core switching and other system processes. We can see that from run to run the time differs and that can be for a multitude of reasons. As previously mentioned the scheduler is balancing thousands of tasks at the same time which are different from run to run. There is also the overhead of context switching which varies as well as kernel level contention with functions like pthread_create and malloc often requiring internal locks within the kernel. Analyzing the data we can draw the conclusion that for a fixed number of tasks the flat structure is far superior at throughput for independent tasks. The hierarchical structure introduces management complexity that uses resources due to the fork and join dependencies meaning a parent cannot terminate until all of its children are joined. We also learned that a deeper narrower tree can sometimes outperform a wider tree due the the initial spike in resource needed with a wide tree.