CS342 OPERATING SYSTEMS

PROJECT #2

THREAD SUPPORT LIBRARY (TSL)

EXPERIMENTS REPORT

24.03.2024

Yasemin Akın Section 002 22101782

Göktuğ Serdar Yıldırım Section 002 22103111

Uygar Aras Section Section 002 22103277

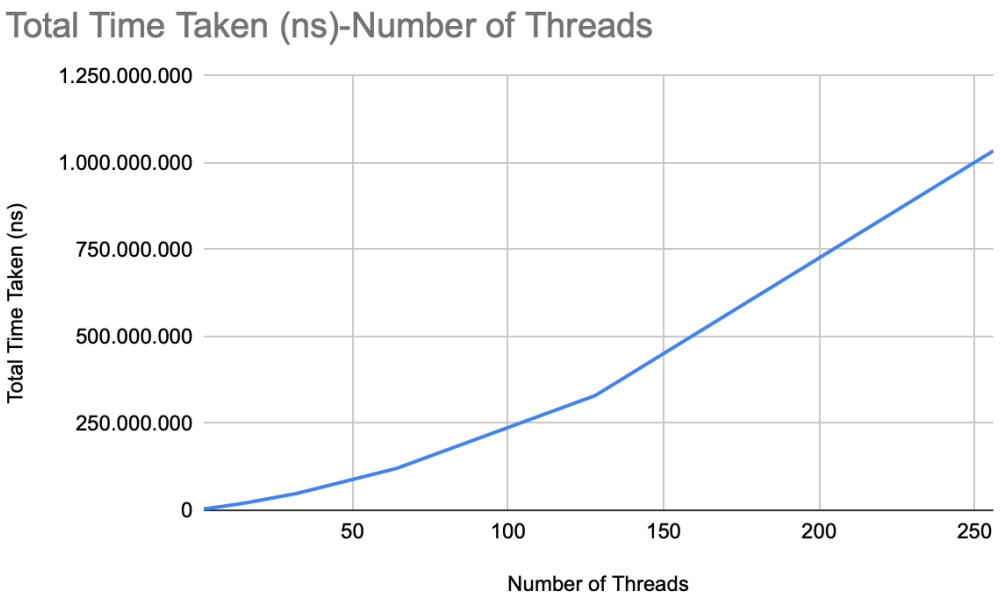
Deniz Can Özdemir Section 001 22003854

Experiment 1 – Baseline Performance Measurement

Objective of this experiment is to establish a baseline performance for simple thread operations like creation, yield, join, and exit. Expectation is to identify the overhead associated with thread management operations in our library. Method of the program written is the following:

1. Initializes the TSL library with a First-Come, First-Served (FCFS) scheduling algorithm.
2. Creates a specified number of threads (NUM_THREADS), each incrementing a counter 1000 times and voluntarily yielding control after each increment.
3. Waits for all threads to complete using tsl_join.
4. Measures the total time taken for all threads to perform their work and terminate.

Number of Threads	Total Time Taken (s)
2	0.003395210
8	0.010818147
16	0.021734963
32	0.048212409
64	0.120112029
128	0.330079230
256	1.034406436



Experiment 2 – Context Switching Overhead

The intention is to make a calculation of the TSL overhead focus on context switching. The reference step (baseline performance focused program) is made by iterating the counter without any context switches until a loop finishes. In the second program, there is context swapping and threads alternate yielding after increasing. A correct comparison of the performance times from these two experiments will certainly yield the amount for the context-switching overhead.

Maximum Number Count	Without Context Switching (s) (Main 1)	With Context Switching (s) (Main 2)
100,000	0.000412030	0.168525980
10,000,000	0.041616926	16.904999598

Experiment 3 – Stack Usage Efficiency

We used Resident Set Size (RSS), a term used to describe the portion of memory occupied by a process that is held in RAM (Random Access Memory), to assess how our Thread Support Library (TSL) manages stack memory with varying numbers of threads and task complexities. We have learned from our research that the RSS is an important metric for understanding the actual memory footprint of a process at any given moment. We have placed the following method to the start and end of our main function and switched between the simple task (count until 10000) and the complex task (a depth-first search (DFS) on a binary tree to find if a path equals a given sum) with varying thread numbers:

```
void log_memory_usage() {
   FILE *f = fopen("/proc/self/status", "r");
   char line[256];
   while (fgets(line, sizeof(line), f)) {
       if (strncmp(line, "VmRSS:", 6) == 0) {
           printf("%s", line);
           break;
       }
   }
   fclose(f);
}
```

Number of Threads	Simple Task Max RSS (kB)		Complex Task Max RSS (kB)	
	Before Starting Thread	After All Threads Completed	Before Starting Thread	After All Threads Completed
32	632	632	648	648
64	656	1212	624	1360
128	640	1640	624	1620
256	668	2080	636	2368

Experiment 4 – Impact of Yield Frequency on Performance

This experiment aimed to understand how the frequency of `yield()` calls affects system performance. We hypothesized that too many yields could slow down the system due to high context-switching overhead, while too few might lead to unfair CPU time allocation among threads. To test this, we crafted a program where threads perform a set workload but yield at varying frequencies. Through observing the impact on performance and fairness, we sought to identify an optimal yield frequency that ensures efficient and equitable CPU usage across threads. The following screenshot is the output of this experiment.

```
[yasemin.akin@dijkstra osproject2]$ ./run
Thread with yield frequency 10000 completed in 61200 ns.
Thread with yield frequency 1000 completed in 207893 ns.
Thread with yield frequency 100 completed in 693635 ns.
Thread with yield frequency 10 completed in 3818983 ns.
Thread with yield frequency 1 completed in 24312816 ns.
All threads completed.
[yasemin.akin@dijkstra osproject2]$
```

Explanation of the behavior:

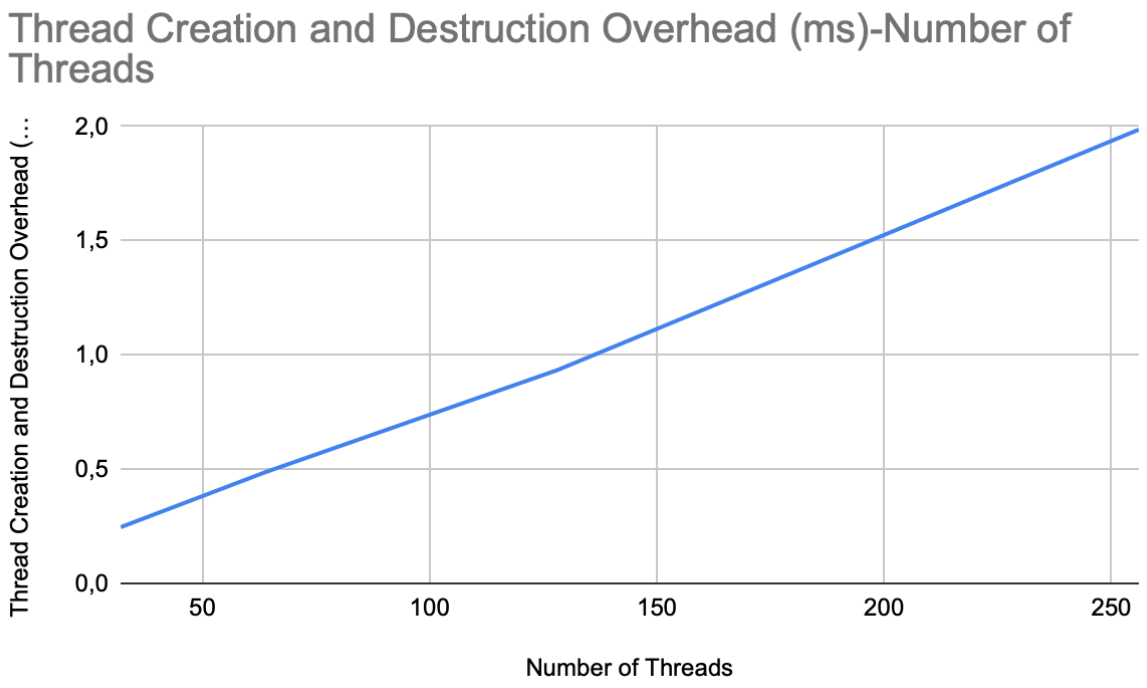
1. Context Switch Overhead: At a high yield frequency (e.g., yielding after every single operation or "Thread with yield frequency 1"), threads frequently leave control of the CPU, leading to numerous context switches. Each context switch undergoes overhead due to saving and restoring the execution context. This overhead accumulates, leading to longer total execution times for completing the thread's workload.

2. CPU Utilization Efficiency: When a thread yields less frequently (e.g., "Thread with yield frequency 10000"), it retains control of the CPU for longer periods, allowing it to progress further through its workload with fewer interruptions. This efficient use of CPU time reduces the total time needed to complete the thread's task because it minimizes the overhead associated with context switching.

Experiment 5 – Thread Creation and Destruction Overhead

The goal of this experiment is quantifying the overhead involved in constructing and destroying threads through our TSL, measured in terms of the amount of time it takes to process a large set of sequential threads where each thread carries out a minimum amount of work and then joins (destroys). Measuring the outcome allows to determine where the blockage may be or the ineffectiveness of the methods about the thread creation, scheduling, and termination algorithms.

Number of Threads	Thread Creation and Destruction Overhead (ms)
32	0.246
64	0.488
128	0.933
256	1.986



Conclusion

Based on the experimental results provided, we can draw several conclusions regarding the performance and efficiency of our Thread Support Library (TSL) under different scenarios. These experiments were conducted to evaluate various aspects such as baseline performance, context switching overhead, stack usage efficiency, impact of yield frequency on performance, and thread creation and destruction overhead. Here's a comprehensive conclusion covering all these aspects:

1. Baseline Performance Measurement: The test conditions used in this experiment were applied only to measure the baseline performances for the four major operations of the thread management, such as creation, yielding, joining, and exiting. In all likelihood, it uses total time that grows linearly with the number of threads, as one would anticipate given the overhead of managing more threads. This baseline performance sets the stage for understanding the overheads and efficiencies of our library.

2. Overhead of Context Switching: The massive difference in the execution time with and without context switching of the same workload indicates the overhead of context switching. That is, context switching is a necessary part of cooperative multitasking, but data suggests optimization of context switching operations could significantly increase performance, particularly under high loads.

3. Stack Usage Efficiency: The measurements of Resident Set Size (RSS) before and after the completion of the thread for tasks of different complexities suggest that our TSL is an efficient stack memory manager, which has modest increases in memory usage for increasing threads. These are especially noteworthy in more complex tasks and indicate the effect our library stack management mechanisms are having on minimizing memory footprint: the most critical factor in resource-constrained environments.

4. Effect of the number of yields on performance: The experiment on yields threw light upon the concept of balancing the number of yields in a way that it could achieve fair utilization of CPU among the executing threads at the same time, achieving less context-switching overhead and, in turn, better performance. On the other hand, overly low yield may imply that the CPU spends an unwarranted amount of time making allocations, so an optimal yield frequency has to be used to balance performance versus fairness. Such an optimum yield may be context-dependent and require dynamic adjustments based on workload characteristics.

5. Creation and Destruction of Threads: Overhead on creation and destruction of threads is directly proportional to the threads. Though the overhead is relatively small, it's one of the supporting factors that optimization of thread lifecycle management in applications where there is frequent occurrence of creating and destroying threads to avoid performance bottleneck.