Final Report

Kernel-Level Thread Implementation and Analysis

Authors

Team 3

Su Zhekai (Student ID: 121090492)
Zhao Yiheng (Student ID: 225040146)

April 30, 2026

Contents

List of Figures

List of Tables

Abstract

This report studies kernel-level thread implementation and performance analysis for this project. We implement a Linux kernel module that creates kernel threads, binds them to CPUs, measures context switches and timing information, and exports statistics through `/proc/kthread_perf/stats`. A user-space pthread benchmark is implemented as the required ULT baseline, and a `ucontext`-based coroutine library is added as an advanced comparison. The experiments compare Kernel-Level Threads (KLT), User-Level Threads (ULT), and Coroutines under normal atomic workloads, high-frequency yielding, CPU-affinity settings, workload scaling, and thread-count scaling. The results show that pthreads outperform the instrumented kernel-thread workload in the tested atomic benchmarks, while coroutines achieve the highest throughput in the aligned cooperative-scheduling comparison. Overall, the project demonstrates that kernel-level execution provides direct scheduler integration and observability, but it is not automatically faster than user-space concurrency mechanisms.

Keywords: Linux kernel threads; pthreads; coroutines; CPU affinity; context switch; `/proc`; scheduling; performance evaluation.

1 Introduction

1.1 Background

Threads are the basic scheduling units used by modern operating systems to express concurrent execution. Compared with a process, a thread shares most process-level resources, such as address space, file descriptors, and signal-related state, while maintaining its own execution context. This difference makes threads cheaper to create and switch than processes, but it also introduces synchronization, scheduling, and measurement challenges.

This project focuses on kernel-level thread implementation and analysis. It requires a Linux kernel module that creates kernel threads, binds them to selected CPUs, collects context-switch statistics, and compares the resulting behavior with a user-space pthread program. This report also includes a user-space coroutine implementation as an advanced comparison model. The subject is not only about writing threads. It is about understanding where concurrency management happens, how Linux exposes scheduling state, and how kernel-level execution differs from user-level execution in measurable experiments.

1.2 Problem statement

The goal of this project is to implement and evaluate kernel-level threads (KLTs) under controlled workloads. Specifically, the project addresses three questions:

1. How can kernel threads be created, controlled, stopped, and cleaned up safely inside a loadable kernel module?
2. How can CPU affinity, context switches, runtime, CPU time, and throughput be measured for kernel threads?
3. How does the behavior of kernel threads compare with user-space pthreads and coroutines under normal computation and high-frequency yielding?

To answer these questions, the project implements a kernel module named `kthread_perf.ko`, a user-space comparison program based on pthreads, a small coroutine library based on `ucontext`, and analysis scripts that collect and summarize the experimental results.

1.3 Structure of this report

Section 2 reviews the required operating-system concepts. Section 3 describes the implementation. Section 4 explains the experimental design. Section 5 presents the results. Section 6 analyzes the main findings. Sections 7 and 8 discuss the project value, limitations, and future work. Section 9 concludes the report.

---

2 Background and Key Concepts

2.1 Process vs thread

A process is usually treated as the unit of resource ownership, while a thread is treated as the unit of scheduling. A process has its own address space and resource tables. Multiple threads inside the same process share the same address space but have separate stacks, register states, and scheduling states.
This project follows the standard distinction: a process is the basic unit of resource allocation, while a thread is the basic unit of scheduling.

Table 1: Process vs thread comparison

Feature	Process	Thread
Definition	Basic unit of resource allocation	Basic unit of scheduling
Memory	Independent address space	Shared address space
Resources	Independent file descriptors and signal handling	Shared file descriptors and signal handling
Overhead	Higher creation and switching overhead	Lower creation and switching overhead

This distinction explains why threads are useful for concurrency. Creating a new process requires duplicating or setting up a larger resource context. Creating a thread is lighter because most resources can be shared. However, shared resources also mean that threads must coordinate access to common data structures, especially when several execution contexts update the same variable.

2.2 User-level threads and kernel-level threads

User-level threads are managed by user-space libraries. The kernel may not be aware of every scheduling decision made by the user-level runtime. Such threads can be lightweight, but they can suffer from blocking and parallelism limitations depending on the mapping model.
Kernel-level threads are managed directly by the Linux scheduler. Each kernel thread has a `task_struct` and participates in kernel scheduling. Kernel-level threads can run on different CPUs and can be assigned affinity constraints. The cost is that scheduling and lifecycle control involve kernel mechanisms, and errors in implementation can affect system stability.
The project uses pthreads as the user-space baseline and Linux kernel threads as the kernel-space implementation. This creates a direct comparison between user-mode and kernel-mode execution paths. The trade-off can be summarized as follows:

Table 2: User-level and kernel-level thread comparison

Thread type	Advantages	Limitations
User-level threads	Fast creation and switching; can be managed without frequent kernel involvement	Blocking system calls may block the whole process in some models; parallelism depends on mapping model; scheduling is library-controlled
Kernel-level threads	True parallel execution on multiple cores; blocking in one thread does not stop other kernel-scheduled threads; better visibility to the OS scheduler	Higher creation and switching overhead; kernel scheduling involvement; more expensive context switches

The common thread-mapping models are:

Table 3: Thread mapping models

Model	Description	Advantages	Disadvantages
Many-to-One	Multiple user-level threads map to one kernel thread	Fast user-space switching	No true parallelism and blocking problems
One-to-One	Each user-level thread maps to one kernel thread	True parallelism	Higher thread creation overhead
Many-to-Many	Multiple user-level threads map to multiple kernel threads	Balances parallelism and overhead	More complex implementation

Modern Linux pthreads use the one-to-one style through NPTL, which is why pthreads are a meaningful baseline for kernel-scheduled thread behavior.
This report also evaluates coroutines as a third concurrency model. Coroutines are scheduled cooperatively in user space. Unlike pthreads, coroutine switching does not require the kernel scheduler for every logical task switch. In this project, the coroutine implementation uses `ucontext` to allocate stacks, create coroutine contexts, resume them, yield back to the scheduler, and destroy scheduler state. This gives a useful contrast: KLTs represent kernel-managed concurrency, pthreads represent kernel-scheduled user threads, and coroutines represent user-space cooperative scheduling.

2.3 Linux thread evolution

Linux threading has evolved from earlier LinuxThreads-style designs to the Native POSIX Thread Library (NPTL). The modern Linux model uses `task_struct` for both processes and threads. Threads in the same process share a thread group ID (TGID), while each schedulable entity still has its own kernel task identity.
This model is important for the project because it shows that Linux does not treat threads as a completely separate object type. Instead, Linux represents processes and threads through task structures with different sharing properties.
From the user-space perspective, the process ID is the TGID, while the individual thread ID corresponds to the real kernel PID of that schedulable task. From the kernel perspective, each thread is still represented as an independent `task_struct`:

```
struct task_struct {
    pid_t pid;   // Thread ID
    pid_t tgid;  // Thread group ID, usually seen as process ID
};
```

2.4 The clone() system call

The `clone()` system call is the core mechanism behind Linux task creation. Unlike `fork()`, which creates a child process with mostly separate resources, `clone()` allows selective sharing through flags such as `CLONE_VM`, `CLONE_FS`, `CLONE_FILES`, `CLONE_SIGHAND`, and `CLONE_THREAD`.
The lecture notes emphasize that omitting `CLONE_THREAD` creates a lightweight process with separate PID/TGID behavior rather than a normal POSIX-style thread. This is useful background for understanding pthreads and Linux kernel task creation.

Table 4: Task-creation system calls

System call	Semantics	Resource sharing	Typical use
`fork()`	Create a child process	Mostly copied resources	Process creation
`vfork()`	Create a child process that temporarily shares memory	Shared address space before `exec()`	Fast creation before `exec()`

System call	Semantics	Resource sharing	Typical use
`clone()`	Create a lightweight task with selectable sharing	Controlled by clone flags	Thread creation and low-level task creation

2.5 Kernel thread APIs

Linux provides kernel APIs for kernel-thread creation and lifecycle management. Two common approaches are:

```
struct task_struct *task = kthread_run(thread_fn, data, "thread_name");
```

and:

```
struct task_struct *task = kthread_create(thread_fn, data, "thread_name");
kthread_bind(task, cpu_id);
wake_up_process(task);
```

The second approach is used in this project because it allows CPU affinity to be set before the thread starts. This is important for controlled performance experiments.

Table 5: Kernel-thread API comparison

API style	Main idea	Strength	Limitation
`kthread_run()`	Create and start the thread immediately	Simple	Less control before the thread starts
`kthread_create()` + `wake_up_process()`	Create first, configure, then wake	Can bind CPU before execution	More code and more lifecycle responsibility

Kernel threads must stop voluntarily. A thread function should periodically call `kthread_should_stop()`, and module cleanup should call `kthread_stop()` to request termination and wait for the thread to exit. This avoids unsafe forced termination and prevents module-unload races.

2.6 CPU affinity and context-switch statistics

CPU affinity restricts where a thread is allowed to run. In the kernel module, CPU binding is implemented with `kthread_bind()`. In the pthread baseline, affinity is implemented with `pthread_setaffinity_np()`. Binding threads makes experiments more reproducible and makes it possible to study oversubscription when multiple threads share the same CPU.

Context-switch statistics are collected from `task_struct` fields for kernel threads:

```
task->nvcsw;   // voluntary context switches
task->nivcsw;  // involuntary context switches
```

The user-space baseline reads similar statistics from `/proc/[tid]/status`, using the `voluntary_ctxt_switches` and `nonvoluntary_ctxt_switches` fields.

3 Implementation

3.1 System architecture

The implemented system has three main parts:

1. A kernel module, `kthread_perf.ko`, that creates and measures kernel threads.
2. A user-space pthread program, `user_threads`, that provides a comparable baseline.
3. Analysis scripts and result files that automate testing and summarize performance data.

The kernel module exports its statistics through `/proc/kthread_perf/stats`. The user-space program prints a JSON-style result summary that can be written back into the proc file, allowing the kernel-side report to display KLT and ULT results together.

3.2 Kernel module design

The kernel module defines configurable parameters for the experiment:

- `thread_count`: number of kernel threads, bounded by `MAX_THREADS`.
- `work_target`: number of operations per thread.
- `work_mode`: normal computation or high-frequency yield.
- `yield_interval`: how often a thread yields in yield mode.
- `yield_sleep_jiffies`: sleep duration for each voluntary yield.
- `bind_cpu_count`: number of CPUs used for affinity binding.

The module maintains per-thread statistics, global atomic counters, and procfs entries. The shared workload counter is implemented with `atomic_t`, which allows multiple kernel threads to update the same counter safely.

3.3 Kernel thread creation and lifecycle control

The kernel module uses `kthread_create()` instead of `kthread_run()`. This design makes it possible to bind each thread to a CPU before it begins execution:

```
thread_stats[thread_id].task = kthread_create(
    thread_function,
    &thread_stats[thread_id],
    "perf_thread_%d",
    thread_id
);

kthread_bind(thread_stats[thread_id].task, cpu_id);
wake_up_process(thread_stats[thread_id].task);
```

The thread function runs until either `kthread_should_stop()` becomes true or the configured per-thread `work_target` is reached. During module exit, all live kernel threads are stopped and completion objects are used to reduce unload races.

3.4 CPU affinity control

Each kernel thread is assigned a CPU according to:

```
cpu_affinity[i] = i % active_bind_cpus;
```

This supports both normal 4-core experiments and oversubscription experiments. For example, with four threads and one bound CPU, all four threads compete on one core. With four bound CPUs, the threads can run on separate cores.

The user-space pthread program mirrors this behavior by setting `pthread_setaffinity_np()` inside the worker thread function.

3.5 Workload implementation

The main workload is an atomic increment loop. Each worker updates a shared counter, updates a local counter, and increases the total work count:

```
atomic_inc(&shared_counter);
atomic_inc(&stats->local_counter);
```

```
stats->work_count++;
atomic_inc(&total_work_done);
```

Two modes are supported:

- **Normal mode**: threads execute the atomic workload continuously until the target count is reached.
- **Yield mode**: threads periodically enter an interruptible state and call `schedule_timeout()`, producing voluntary context switches.

The pthread baseline uses a comparable atomic workload and calls `sched_yield()` in its yield mode.
This matches the intended measurable payload: the worker threads should do something measurable, such as incrementing a shared atomic counter or repeatedly sleeping and waking. The atomic counter was chosen because it gives a simple correctness check: the final counter should equal `thread_count * work_target`.

3.6 Context-switch and timing measurement

For kernel threads, the implementation records initial and final context-switch counters from the current task. It computes voluntary, involuntary, and total context switches as differences between the final and initial snapshots.
Runtime and CPU-time measurements are also recorded. The kernel module reports per-thread runtime, CPU time, CPU percentage, throughput, and latency summaries.
The pthread baseline uses `clock_gettime(CLOCK_THREAD_CPUTIME_ID)` and `getrusage(RUSAGE_THREAD)` to collect thread-level CPU-time statistics. It reads `/proc/[tid]/status` to obtain user-space context-switch counts.

3.7 /proc interface

The module creates:
`/proc/kthread_perf/stats`
Reading this file prints the kernel-thread statistics and, if available, the user-thread comparison data. Writing a JSON-style result from the user-space test updates the stored ULT metrics. This creates a simple communication channel between the user-space benchmark and the kernel module.

3.8 User-space pthread baseline

The user-space program creates pthreads with the same logical workload. Each thread is bound to a selected CPU, performs atomic increments, optionally calls `sched_yield()`, and reports throughput, context switches, CPU time, and correctness of the final counter.
The baseline is essential because the project requires comparison between kernel module threads and user-space pthreads. It also helps separate general workload behavior from kernel-specific behavior.

3.9 Coroutine library

The project also implements a coroutine library in `coroutine_lib/`. The implementation is based on `ucontext` and provides scheduler and coroutine operations such as:

- `coroutine_scheduler_init()`
- `coroutine_create()`
- `coroutine_resume()`
- `coroutine_yield()`
- `coroutine_run()`
- `coroutine_scheduler_destroy()`

The coroutine benchmark creates multiple coroutine workers, each with a fixed number of work iterations. In normal mode, workers increment their counters without yielding. In yield mode, each worker calls `coroutine_yield()` after a configured interval. The benchmark records throughput, user time, system time, process-level context switches, and logical coroutine switches.
This model is not a replacement for the required KLT-vs-ULT comparison. It is an advanced option that helps explain the performance difference between kernel-managed scheduling, pthread scheduling, and cooperative user-space scheduling.

3.10 Automation and analysis scripts

The project includes scripts that run normal and yield tests, collect kernel and user-space outputs, parse results, and generate summary tables and figures. The generated result files include:

- `analysis_summary (2).md`
- `analysis_summary.md`
- `perf_report_comparison_normal.txt`
- `perf_report_comparison_yield.txt`
- `run_comparison_metrics.png`
- `yield_interval_sweep.png`
- `core_mode_matrix.png`
- `workload_comparison.png`
- `thread_count_comparison.png`
- `coroutine_mode_comparison.png`

---

4 Experimental Design

4.1 Requirements mapping

The project requirements ask for a kernel module that creates kernel threads, binds them to CPUs, measures context switches, and compares kernel threads with user-space pthreads. The implementation additionally includes coroutine comparison as an advanced option. The mapping is as follows:

Table 6: Project requirements mapping

Requirement	Implementation
Create kernel threads	`kthread_create()` and `wake_up_process()`
Bind threads to CPUs	`kthread_bind()` and configurable `bind_cpu_count`
Perform measurable work	Shared atomic counter increment workload
Measure CSW	`task->nvcsw` and `task->nivcsw`
Export results	`/proc/kthread_perf/stats`
Compare KLT vs ULT	pthread benchmark and result parser
High-frequency yield scenario	`schedule_timeout()` in KLT, `sched_yield()` in ULT
Clean cleanup	`kthread_stop()`, completion wait, proc removal
Advanced comparison	`ucontext` coroutine library and coroutine benchmark

4.2 Scenario definitions

Two main scenarios are tested.
Scenario A: normal atomic workload. Four threads perform atomic increments until each thread completes 1,000,000 operations. The intended measurement is steady compute throughput with minimal voluntary yielding.
Scenario B: high-frequency yielding. Four threads perform the same atomic workload but yield periodically. In the kernel module, yielding is implemented with `schedule_timeout()`. In user space, it is implemented with `sched_yield()`. The intended measurement is the effect of voluntary context switching on throughput and CPU usage.
Additional experiments vary the yield interval, CPU-binding count, work target, thread count, and coroutine task count. The full comparison covers three models: KLT, ULT, and Coroutine.

4.3 Metrics

The main metrics are:

- Throughput in operations per second.
- Total work completed.
- User time and system time.
- Voluntary and involuntary context switches.
- CPU efficiency.
- Per-thread runtime and CPU time.
- Logical coroutine switches for the coroutine benchmark.

These metrics match the project requirements and allow a direct comparison between KLT and ULT behavior.

4.4 Fairness controls

The experiments use the same thread count or task count where possible, similar counter-increment workloads, fixed work targets, and CPU-affinity settings. Repeated runs are averaged in the analysis summary to reduce noise. The project also keeps normal and yield modes separate because they stress different scheduling paths.

4.5 Procedure

The typical procedure is:

1. Build the kernel module, user-space pthread program, and coroutine benchmark.
2. Insert the kernel module with selected parameters.
3. Read `/proc/kthread_perf/stats` to collect KLT results.
4. Run the pthread benchmark with matching settings.
5. Write the ULT JSON result back to `/proc/kthread_perf/stats`.
6. Read the combined comparison report.
7. Run the coroutine benchmark under normal and yield modes.
8. Repeat for normal, yield, binding, workload-size, and thread-count experiments.
9. Run the analysis script to generate summary tables and charts.

---

5 Experimental Results

5.1 Scenario A: normal atomic workload

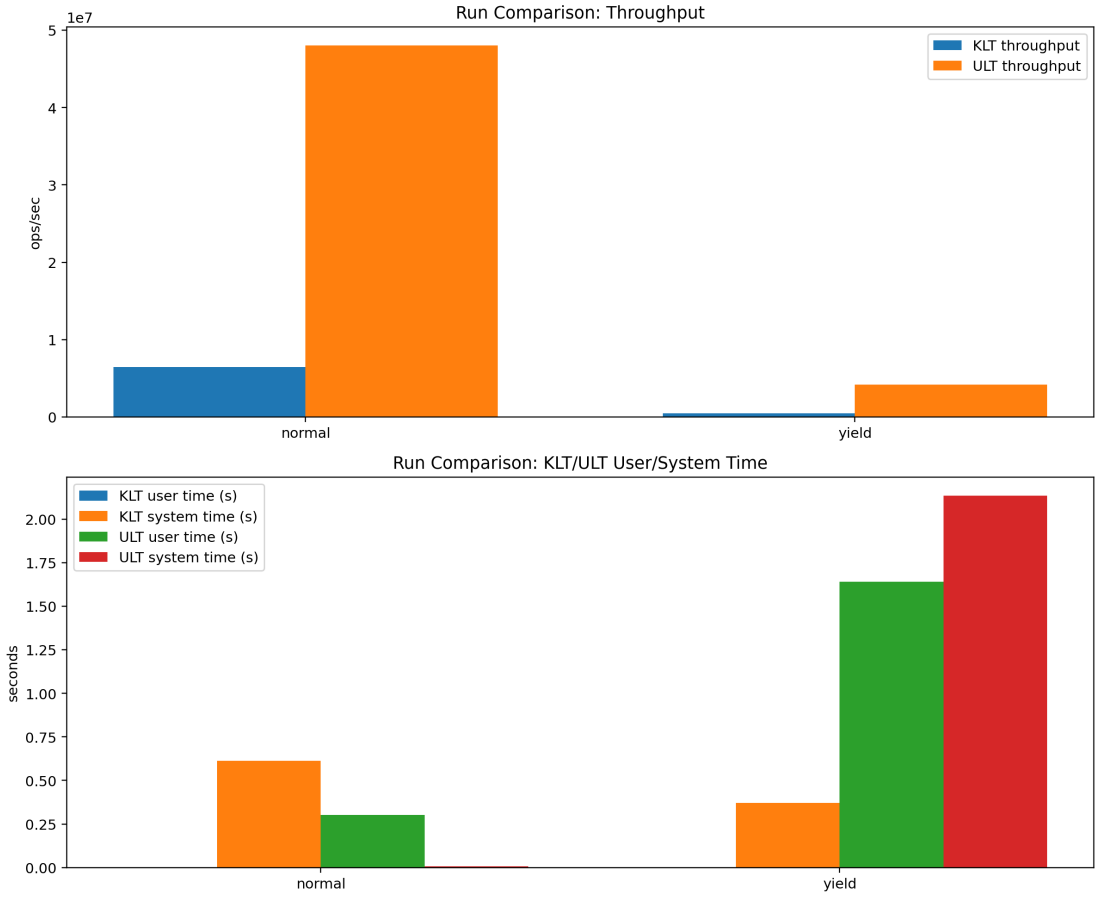

Figure 1: Run comparison between KLT and ULT

In the normal 4-thread experiment, each thread performed 1,000,000 atomic operations. The kernel module completed 4,000,000 total operations.

Table 7: Scenario A normal workload results

Metric	KLT	ULT
Throughput	6,478,375 ops/s	48,044,031 ops/s
KLT/ULT ratio	13.5%	100%
KLT system time	0.614 s	N/A
KLT voluntary switches	0	N/A
KLT involuntary switches	0	N/A
ULT user/system time	N/A	0.301896 s user, 0.005962 s system

The KLT implementation achieved the correct final counter value and produced nearly full CPU utilization. However, the pthread baseline achieved substantially higher throughput for this specific user-space atomic workload.
This result should be interpreted as the performance of an observable kernel-thread benchmark rather than the theoretical maximum speed of kernel threads. The kernel module records statistics, exposes procfs state, and performs accounting inside the measured path. The pthread baseline runs a tighter user-space loop, which explains why it achieves higher throughput for this synthetic atomic workload.

5.2 Scenario B: high-frequency yielding

In the yield scenario, each kernel thread also completed 1,000,000 operations, but each thread yielded periodically.

Table 8: Scenario B yield workload results

Metric	KLT	ULT
Throughput	497,802 ops/s	4,207,257 ops/s
KLT/ULT ratio	11.8%	100%
KLT system time	0.372 s	N/A
KLT voluntary switches	3,996	N/A
KLT involuntary switches	0	N/A
ULT user/system time	N/A	1.641086 s user, 2.134476 s system

The throughput of both implementations dropped sharply when yielding was introduced. The KLT result shows 3,996 voluntary context switches, matching the design expectation of approximately 999 yields per thread across four threads.
The yield result is useful because it deliberately stresses scheduler interaction. Instead of only comparing arithmetic speed, this scenario shows how often giving up the CPU can dominate total runtime. The drop in throughput confirms that scheduling behavior is a major part of the measured cost.

5.3 Yield-interval sweep

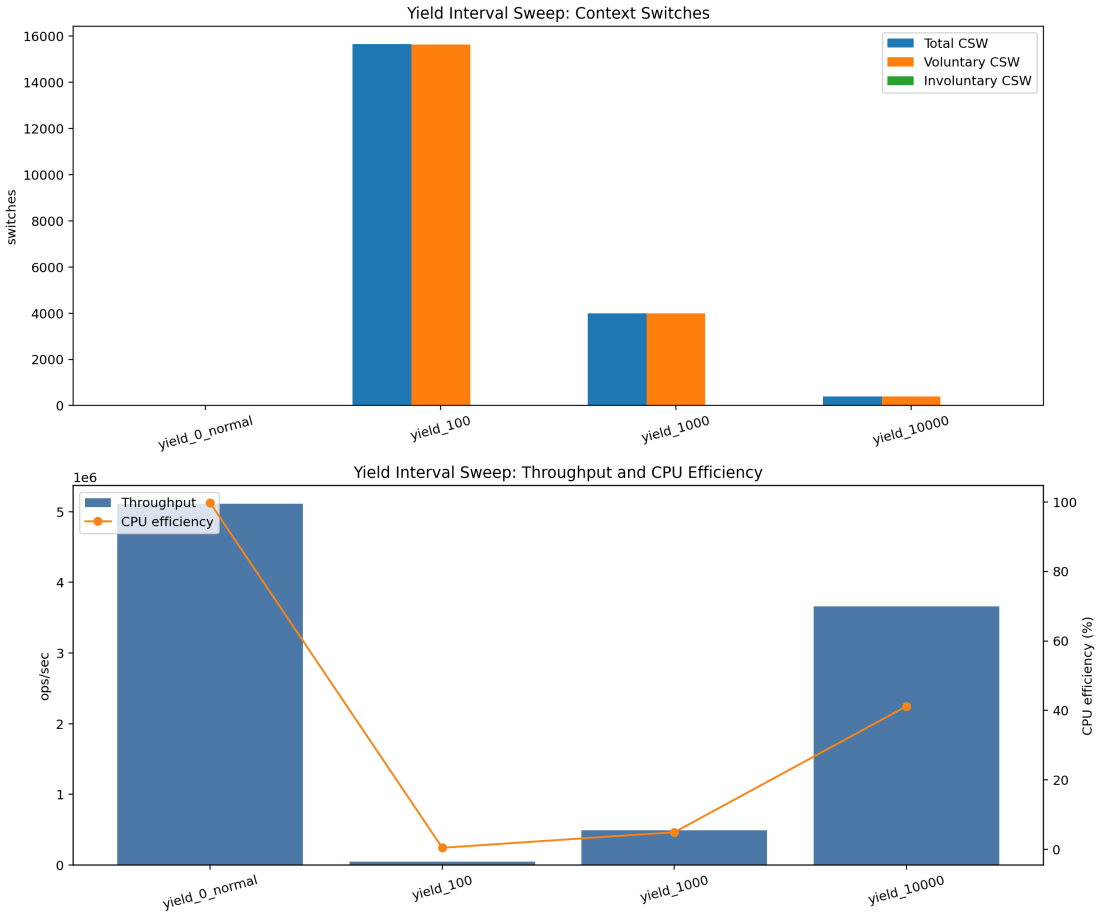

Figure 2: Yield interval sweep

The yield-interval sweep shows how yield frequency affects throughput and CPU efficiency.

Table 9: Yield-interval sweep results

Label	Yield interval	Throughput	CPU efficiency	Voluntary switches
Normal	0	5,112,064.67 ops/s	99.70%	0
Yield 100	100	49,042.67 ops/s	0.50%	15,646.67
Yield 1000	1000	489,778.00 ops/s	4.97%	3,996.00
Yield 10000	10000	3,662,569.00 ops/s	41.20%	396.00

The result is consistent: more frequent yielding causes more voluntary context switches, lower CPU efficiency, and lower throughput.
The sweep also validates the control variable in the experiment. When the yield interval changes from 100 to 10,000, voluntary switches fall sharply and throughput recovers. This makes the slowdown explainable: it is tied to the configured yield frequency rather than to random measurement noise.

5.4 CPU-binding comparison

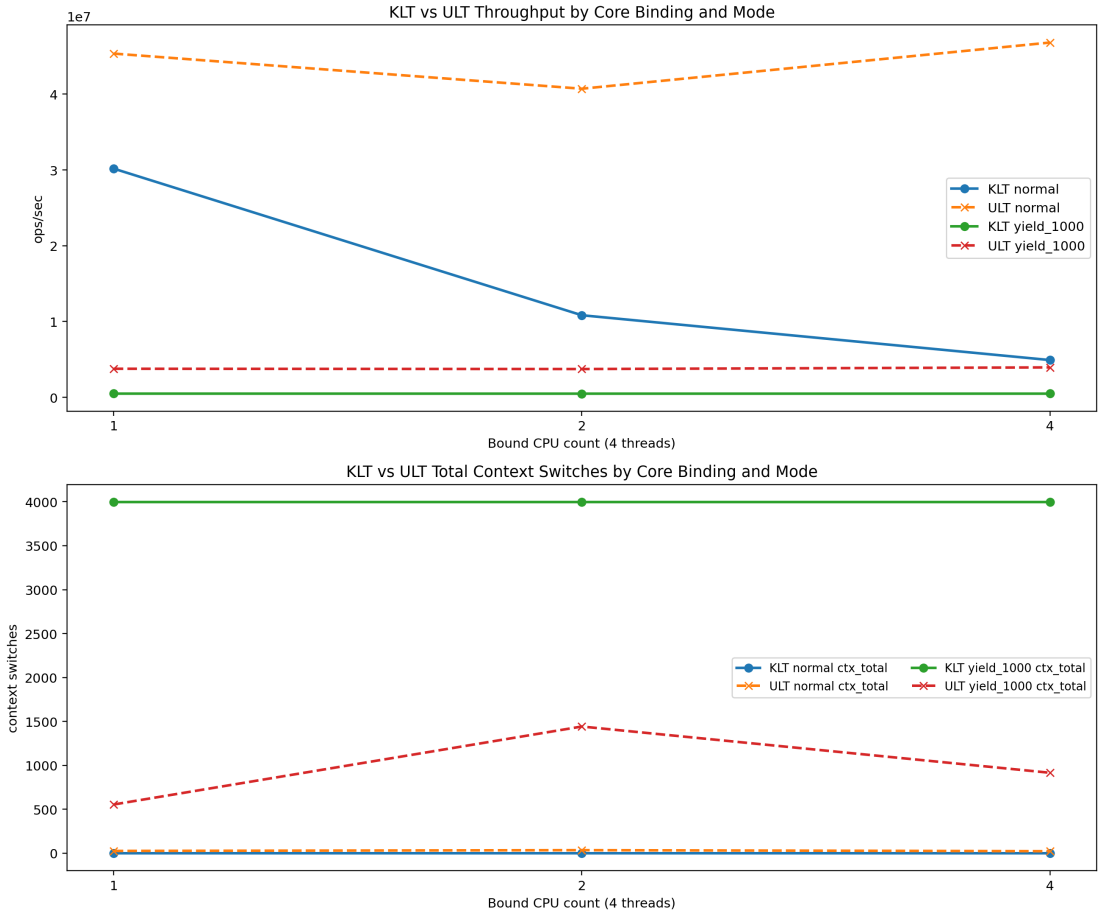

Figure 3: CPU binding impact on KLT and ULT throughput

With four threads and normal mode, binding to fewer CPUs changes the KLT/ULT relationship:

Table 10: CPU-binding comparison

Bound CPUs	Mode	KLT throughput	ULT throughput
1	Normal	30,172,515.67 ops/s	45,348,659.67 ops/s
2	Normal	10,832,564.00 ops/s	40,712,400.67 ops/s
4	Normal	4,919,058.67 ops/s	46,804,430.67 ops/s

Bound CPUs	Mode	KLT throughput	ULT throughput
1	Yield 1000	491,728.67 ops/s	3,773,221.00 ops/s
2	Yield 1000	487,039.33 ops/s	3,736,243.33 ops/s
4	Yield 1000	488,320.67 ops/s	3,950,989.00 ops/s

The ULT throughput remains relatively stable across binding counts in normal mode, while the KLT result is more sensitive to the binding configuration. This suggests that kernel-side measurement and shared atomic contention are affected strongly by scheduling and CPU placement.
The CPU-binding result should not be read as a simple "more CPUs is always better" rule. With a shared atomic counter, spreading work across CPUs can increase cache-coherence traffic and contention. Binding decisions therefore affect both scheduler placement and memory-system behavior.

5.5 Workload scaling comparison

With four threads bound to one CPU, increasing the work target produces stable trends.

Table 11: Workload scaling comparison

Work target	Mode	KLT throughput	ULT throughput
1,000,000	Normal	29,186,876.00 ops/s	46,656,497.33 ops/s
5,000,000	Normal	29,442,138.67 ops/s	44,161,246.67 ops/s
10,000,000	Normal	29,921,437.33 ops/s	44,465,376.33 ops/s
1,000,000	Yield 1000	486,426.67 ops/s	4,135,691.33 ops/s
5,000,000	Yield 1000	488,761.33 ops/s	3,859,280.67 ops/s
10,000,000	Yield 1000	487,733.67 ops/s	3,904,580.00 ops/s

The normal-mode throughput is stable as the workload size grows, which indicates that the benchmark is measuring steady-state behavior rather than startup overhead. Yield-mode throughput is also stable, because the yield interval dominates the execution rate.
This stability increases confidence in the benchmark. If throughput changed dramatically with larger work targets, startup or cleanup overhead might be dominating the measurement. Instead, the results suggest that the main loop behavior is the primary measured cost.

5.6 Thread-count comparison

With one bound CPU in normal mode, the thread-count experiment shows:

Table 12: Thread-count comparison

Thread count	Work iterations	KLT throughput	ULT throughput
4	1,000,000	29,578,459.00 ops/s	44,152,508.00 ops/s
16	1,000,000	30,242,576.67 ops/s	44,543,114.33 ops/s
32	1,000,000	29,754,987.00 ops/s	44,507,082.33 ops/s

Both implementations remain stable as thread count grows under one-CPU binding. The result suggests that the workload is dominated by atomic operation throughput and scheduler behavior rather than by thread creation cost after the benchmark begins.
Because the benchmark uses one bound CPU in this scenario, adding more threads mainly changes scheduling pressure rather than adding true CPU capacity. The stable throughput therefore shows that the workload is bottlenecked by the shared counter and the single CPU execution path.

5.7 Coroutine comparison

The implementation includes a coroutine benchmark using 32 coroutine tasks, one bound CPU, and 1,000,000 work iterations per task.

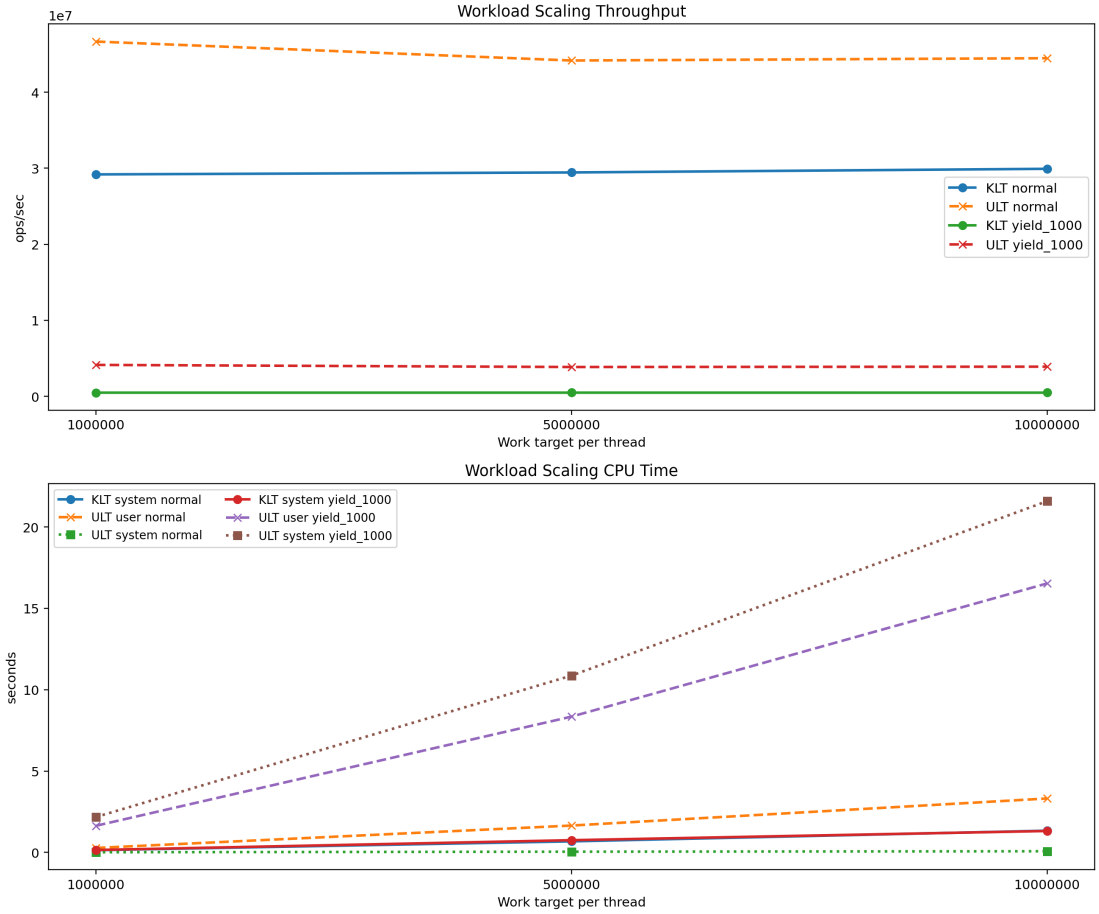

Figure 4: Workload scaling comparison

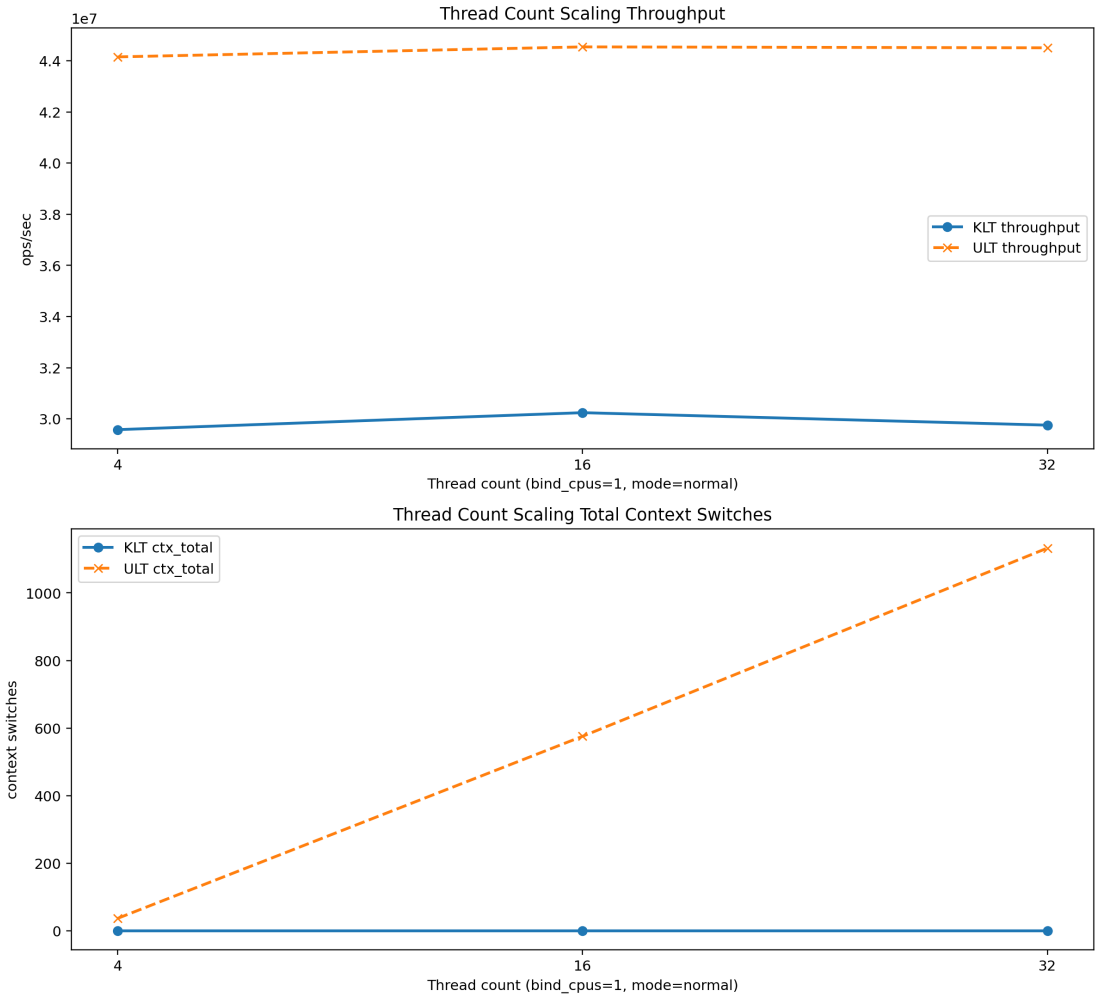

Figure 5: Thread-count scalability comparison

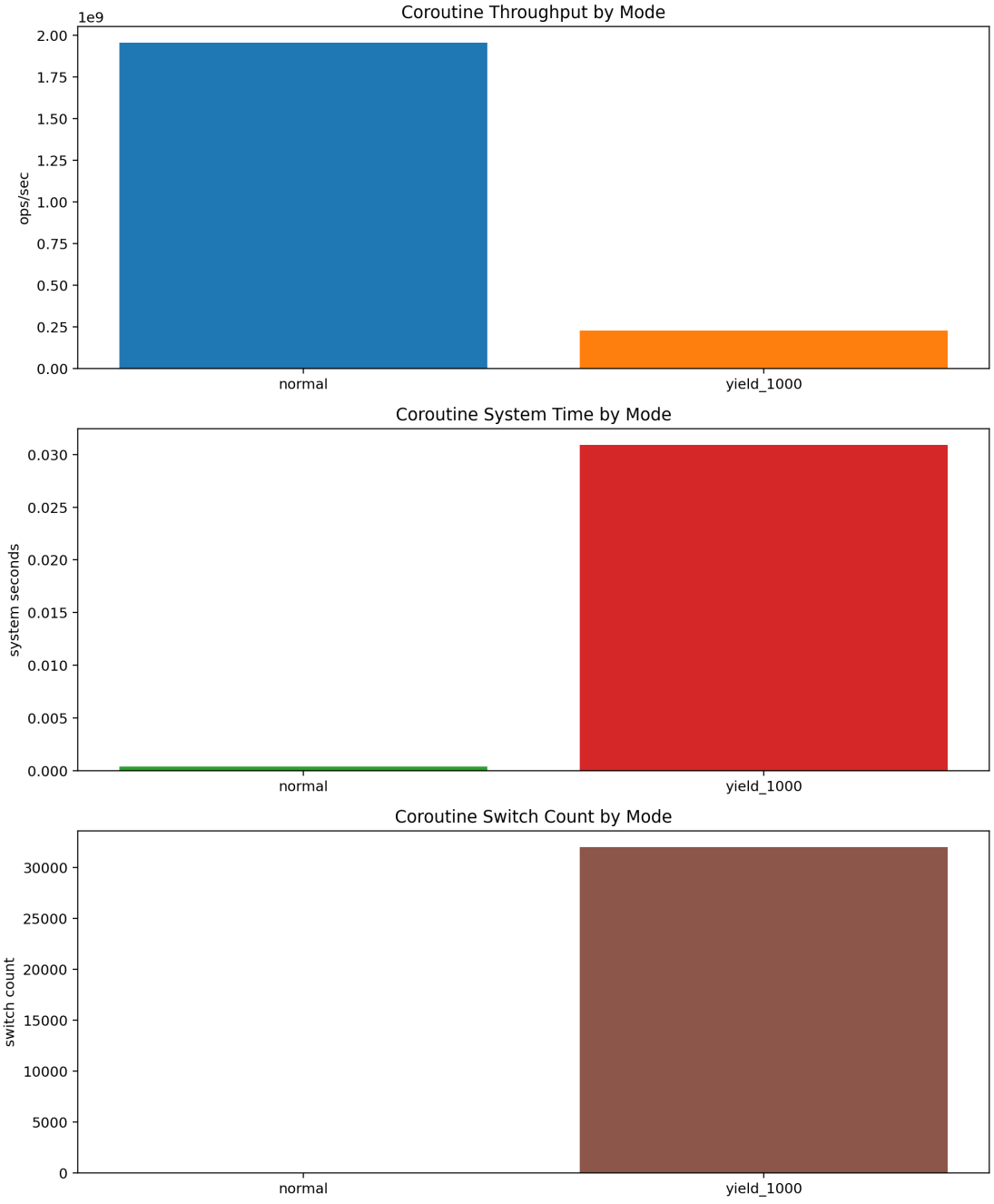

Figure 6: Coroutine throughput by mode

Table 13: Coroutine comparison

Scenario	Yield interval	Coroutine throughput	User time	System time	OS context switches	Coroutine switches
Normal	0	1,956,268,380.36 ops/s	0.015351 s	0.000375 s	8.00	0.00
Yield 1000	1000	227,482,320.41 ops/s	0.109427 s	0.030918 s	4.00	32,000.00

The coroutine result is much faster than both KLT and pthread in this specific counter-only benchmark. This is expected because coroutine switching is cooperative and remains in user space. The `coroutine_switches` value records logical coroutine scheduling events, not kernel context switches.
The coroutine numbers should be interpreted carefully. They show the advantage of cooperative user-space switching for a lightweight non-blocking workload, but they do not imply that coroutines replace kernel-scheduled threads in all cases. Coroutines depend on the program yielding voluntarily and do not provide automatic preemptive parallelism.

5.8 Strict aligned comparison

The strict aligned comparison uses one bound CPU, 32 tasks or threads, 1,000,000 work iterations, and three repeated runs.

Table 14: Strict aligned KLT vs ULT vs Coroutine comparison

Scenario	KLT throughput	ULT throughput	Coroutine throughput	Coroutine vs ULT	Coroutine vs KLT
Normal	29,415,583.00 ops/s	44,957,906.33 ops/s	2,061,391,118.76 ops/s	45.852x	70.078x
Yield 1000	499,228.67 ops/s	2,561,689.33 ops/s	229,176,571.04 ops/s	89.463x	459.061x

This table is the clearest summary of the comparison. KLT provides kernel-level control and observability, pthreads provide efficient kernel-scheduled user threads, and coroutines provide the highest throughput for cooperative user-space switching in this benchmark.
Overall, Section 5 shows a consistent trade-off rather than a single winner. KLT is valuable for kernel integration and observability, ULT is efficient for user-space threaded computation, and coroutines are extremely fast when the workload fits cooperative scheduling. The appropriate model depends on whether the goal is kernel visibility, parallel user execution, or low-cost logical task switching.

---

6 In-Depth Analysis

6.1 Overall assessment

The project successfully implements the required kernel-thread measurement system and compares it against a user-space pthread baseline. It also adds a coroutine benchmark. The results are internally consistent: normal mode has high CPU efficiency and few context switches, while yield mode has many voluntary or logical switches and much lower throughput.
The most important finding is that kernel-level execution does not automatically imply higher throughput. For this workload, pthreads are faster than the instrumented kernel module in both normal and yield scenarios, while coroutines are fastest when the benchmark is aligned to cooperative user-space scheduling. The value of KLT lies in kernel integration, direct scheduler visibility, CPU-affinity control, and kernel-space execution, not in a universal speed advantage.

6.2 Why ULT is faster in the normal scenario

In the normal atomic workload, the pthread baseline reaches about 48.0 million operations per second, while the kernel module reports about 6.48 million operations per second in the direct 4-thread comparison. This is not a contradiction of the theory that kernel threads can run in parallel. Instead, it reflects the specific implementation and measurement path.
The kernel module performs additional accounting inside kernel space: per-thread statistics, timing, latency sampling, proc-visible state, and atomic updates to shared kernel counters. These operations add overhead to each work iteration. The user-space pthread benchmark performs a tight atomic loop and avoids much of the kernel-side measurement overhead during the main workload.
Therefore, the normal-mode result should be interpreted as the performance of this instrumented kernel workload, not as the theoretical maximum performance of all kernel threads.

6.3 Why throughput drops sharply in the yield scenario

The yield scenario intentionally introduces scheduler interaction. In KLT mode, the thread enters `TASK_INTERRUPTIBLE` and calls `schedule_timeout()`. In ULT mode, the pthread calls `sched_yield()`. Both mechanisms reduce useful work per unit time.
The yield-interval sweep confirms this. When the yield interval is 100, throughput drops to about 49,043 operations per second and voluntary context switches rise above 15,600. When the interval increases to 10,000, throughput recovers to about 3.66 million operations per second and voluntary switches drop to about 396. This is a direct demonstration that yield frequency is a controlling variable.

6.4 Why KLT reports zero user time

Kernel threads execute in kernel mode and do not run user-space code. Therefore, their measured user time should be zero. The report correctly classifies KLT execution as system time. In contrast, the pthread baseline performs most normal-mode work in user mode, so the user-space result reports a high user-time percentage.

6.5 Why coroutines are fastest in the aligned benchmark

The coroutine benchmark reaches about 2.06 billion operations per second in normal mode and about 229 million operations per second in yield mode. These numbers are much higher because the benchmark avoids kernel scheduling for each logical task switch. The coroutine scheduler runs in user space and switches between coroutine contexts cooperatively.
This result should not be interpreted as “coroutines are always better than threads.” Coroutines require cooperative yielding and do not automatically provide parallel execution across multiple CPU cores. Their advantage is strongest when the workload is lightweight, non-blocking, and can be scheduled in user space. This is exactly the kind of workload used in the coroutine comparison.

6.6 CPU affinity and oversubscription effects

The CPU-binding experiments show that thread placement matters. Binding all four kernel threads to one CPU can produce different throughput from spreading them across four CPUs. Such differences

can come from shared atomic contention, scheduler placement, cache effects, and the details of how the measurement window is computed.
The experiment is still valuable because it makes CPU affinity visible and configurable. Without affinity control, repeated runs might silently migrate across CPUs and produce results that are harder to explain.

6.7 Context-switch interpretation

In normal mode, the kernel module reports zero context switches for the main 4-thread experiment. This is expected because each thread runs a short CPU-bound workload and reaches its target without voluntarily sleeping.
In yield mode, the module reports 3,996 voluntary context switches and zero involuntary switches. This is also expected because the yielding is deliberately introduced by the thread logic. The absence of involuntary switches indicates that the tested workload is not mainly being preempted by external pressure; rather, it is giving up the CPU voluntarily. For the coroutine benchmark, the key value is `coroutine_switches`, because logical coroutine yields are mostly invisible to the kernel as OS context switches.

6.8 Lessons from the implementation

Several implementation lessons stand out:

1. `kthread_create()` plus `wake_up_process()` gives more experimental control than `kthread_run()` because affinity can be configured before the thread starts.
2. Kernel threads must be designed to stop cooperatively with `kthread_should_stop()`.
3. Procfs is a practical interface for exposing kernel measurements to user space.
4. Measurement code can significantly affect performance, especially when executed inside a tight loop.
5. Coroutines can reduce scheduling overhead dramatically when cooperative scheduling is acceptable.
6. A valid systems experiment must interpret results according to the actual measurement target rather than assuming a simple KLT-vs-ULT winner.

6.9 Measurement validity

The experiment uses repeated runs, fixed work targets, controlled CPU binding, and separate normal/yield scenarios. These choices make the results more reliable than a single ad hoc run. The final counter values also act as correctness checks: if each worker reaches its target, the measured throughput is tied to completed work rather than partial execution.
There are still measurement limits. The KLT module includes instrumentation overhead, and the workload is a synthetic counter benchmark. Therefore, the results should be interpreted as evidence about this implementation and workload, not as a universal ranking of all thread systems. The strongest conclusion is qualitative: kernel visibility, pthread efficiency, and coroutine switching overhead represent different trade-offs.

6.10 System-level implications

For operating-system design, the results illustrate why modern systems provide multiple concurrency abstractions. Kernel threads are appropriate when execution must be visible to and controlled by the kernel, when CPU affinity matters, or when kernel-space work must be performed. Pthreads are appropriate for general-purpose parallel user programs. Coroutines are appropriate when the program can cooperate with a user-space scheduler and wants very low switching overhead.
The project therefore supports the central operating-system lesson: concurrency performance cannot be judged only by whether a mechanism is “kernel-level” or “user-level.” The correct choice depends on scheduling requirements, blocking behavior, synchronization cost, measurement overhead, and whether the workload needs true parallelism or lightweight cooperative switching.

---

7 Discussion

7.1 Main value of the project

The main value of this project is that it turns a conceptual discussion of thread models into a working measurement system. The implementation demonstrates kernel-thread creation, safe lifecycle control, CPU affinity, context-switch statistics, and KLT-vs-ULT comparison under repeatable workloads.
The project also shows that systems performance depends on workload and measurement design. Kernel-level threads provide direct scheduler integration and kernel-mode execution, but they do not automatically outperform pthreads. In this experiment, the pthread baseline is faster for the atomic workload, while the kernel module provides richer visibility into kernel-level behavior. The coroutine benchmark further shows that user-space cooperative scheduling can be extremely efficient when the workload does not need preemptive kernel scheduling.

7.2 Relationship to project requirements

The implementation satisfies the basic requirements:

- It creates kernel threads using kernel APIs.
- It binds threads to selected CPUs.
- It performs measurable work with a shared atomic counter.
- It records voluntary and involuntary context switches.
- It exports results through `/proc/kthread_perf/stats`.
- It compares KLT with a user-space pthread program.
- It includes a high-frequency yield scenario.
- It performs clean module cleanup and removes procfs entries.
- It implements an advanced comparison with a coroutine library.

It also extends the basic requirements with configurable yield intervals, CPU-binding experiments, workload scaling experiments, thread-count experiments, coroutine benchmarking, result parsing, and visualization.

8 Limitations and Future Work

8.1 Limitations

First, the workload is intentionally simple. Atomic increments make the experiment easy to compare, but they do not represent all real kernel-thread use cases.
Second, the kernel module includes measurement and reporting overhead. The measured KLT throughput is therefore the throughput of an instrumented kernel-thread workload, not a pure lower-bound scheduler benchmark.
Third, the experiments are performed on one Linux environment. Results may vary on machines with different CPU counts, kernel versions, scheduler settings, and hardware topology.
Fourth, the coroutine benchmark is intentionally cooperative and user-space only. It measures coroutine scheduling overhead well, but it does not provide the same preemptive parallel execution model as kernel threads or pthreads.

8.2 Future work

Future extensions could include:

1. Testing more realistic workloads, such as producer-consumer queues or kernel event processing.
2. Separating measurement overhead from workload overhead with lighter instrumentation.
3. Comparing different synchronization mechanisms, such as atomic operations, spinlocks, mutexes, and per-CPU counters.
4. Measuring scheduler latency and wake-up latency more directly.
5. Running the same experiments on machines with more CPU cores.
6. Adding debugfs or sysfs controls for changing parameters without reloading the module.
7. Extending the coroutine scheduler with I/O-style workloads to better demonstrate its intended use case.

---

9 Conclusion

This project implements and evaluates a kernel-level thread performance measurement system . The kernel module creates kernel threads, binds them to CPUs, measures context switches and timing information, and exports the results through procfs. A pthread-based user-space program provides the required baseline, and a `ucontext` coroutine library provides an advanced comparison model.

The experiments show that normal atomic workloads achieve high CPU efficiency and minimal context switching, while high-frequency yielding causes many voluntary context switches and significantly lower throughput. The pthread baseline outperforms the instrumented kernel-thread implementation in the tested workloads, which shows that kernel-level execution is not automatically faster. The coroutine benchmark achieves the highest throughput in the aligned comparison because it avoids most kernel scheduling overhead. The correct conclusion is therefore nuanced: KLTs provide kernel integration, direct scheduler participation, and kernel-space observability; pthreads provide efficient kernel-scheduled user threads; and coroutines provide very low-overhead cooperative scheduling when the workload fits that model.

Overall, the project meets the stated requirements and provides a systematic comparison of KLT, ULT, and Coroutine behavior.

---

10 References

1. Linux kernel source documentation and APIs: `kthread_create()`, `kthread_bind()`, `wake_up_process()`, `kthread_stop()`, `task_struct`, and procfs interfaces.
2. Linux manual pages: `clone(2)`, `pthread_create(3)`, `sched_yield(2)`, `proc(5)`.
3. Ingo Molnar, "Completely Fair Scheduler Design and Implementation", Linux Kernel Documentation, 2007.
4. Bradford Nichols and Dick Buttlar, *Pthreads Programming*.
5. Melvin E. Conway, "Design of a Separable Transition-Diagram Compiler", Communications of the ACM, 1963.
6. Robert Love, *Linux Kernel Development*, Pearson Education, 2010.