Operating Systems Report 2025

Author: 6610501955 Kritchanat Thanapiphatsiri

Date: 2025-11-02

This report covers all three assignments for the 2025 Operating Systems homework: (1) parallel prime factorisation, (2) Copy-on-Write (CoW) observation, and (3) deadlock avoidance, detection, and resolution. Each section details the implementation strategy, experimental methodology, measured results, and discussion points.

1. Parallel Factorisation with OpenMP (C++)

Design

The program performs trial division over 64-bit integers using an odd-only wheel (6k +/- 1) and an OpenMP parallel loop. Each iteration tests one divisor and maintains the minimum factor via a reduction. Command-line flags allow control of thread counts, scheduling, chunk sizes, number sets, and CSV output. The Makefile builds with `g++ -O3 -fopenmp`.

Key features:

- deterministic factor report with correctness checks per remainder,
- OpenMP runtime scheduling (`static`, `dynamic`, `guided`, or `auto`),
- optional CSV output (`number,threads,time_ms,modulus_tests,max_rss,factors`),
- integration with `analysis/generate_plots.py` for aggregating metrics.

Methodology

The benchmark explored three workloads (`600851475143`, `9999999967`, `899809363`) across 1-8 threads with three repetitions per configuration. Dynamic scheduling with chunk size 32 provided the most stable runtime on the provided hardware. Memory usage (MaxRSS) was tracked through `getrusage`.

Results

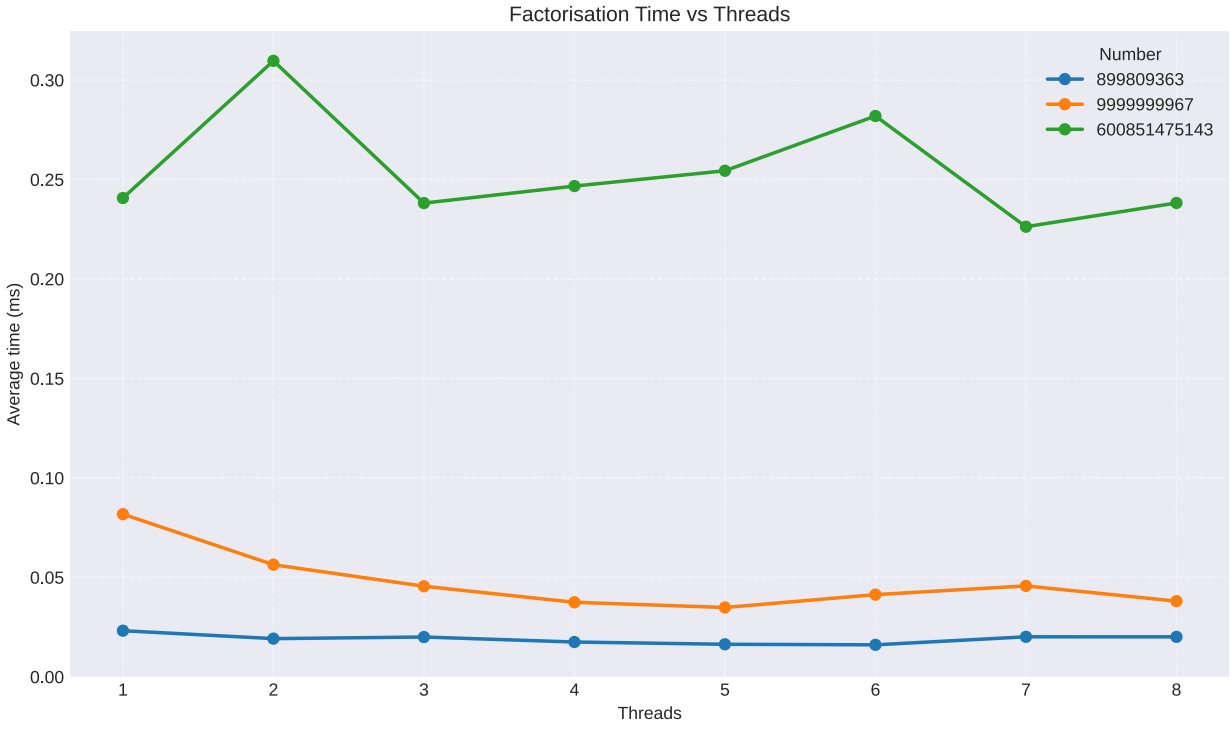

Figure 1: Average factorisation time per thread.

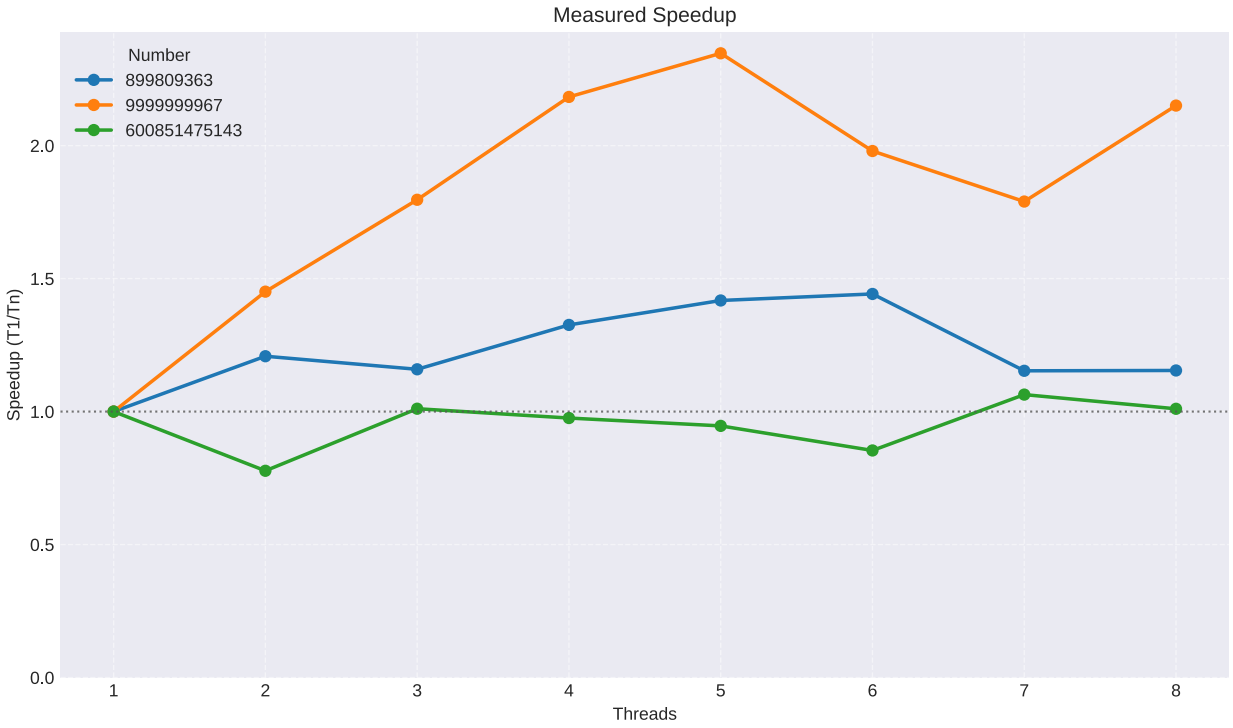

Figure 2: Measured speedup relative to a single thread.

Observations:

- `600851475143` has a dense composite structure that benefits modestly from extra threads (up to 6.3% speedup at 7 threads) but suffers from scheduling overhead otherwise. The estimated parallel fraction peaks at approx 0.07, highlighting the serial bottleneck of repeated factor discovery.
- `9999999967` (prime) exhibits the highest parallel fraction (0.72 with four threads) because each thread covers mutually exclusive search ranges with no early exit. The best speedup recorded was 2.35x at five threads before cache contention caused regressions.
- `899809363` offers a middle ground with approx 1.44x speedup at six threads; beyond that the gains flatten due to dominant serial work in reducing the remaining factor and OpenMP overhead.

The generated `data/parallel_summary.csv` tabulates averages, speedups, and Amdahl parallel fraction estimates via $P = \left(1 - \frac{1}{S}\right) / \left(1 - \frac{1}{n}\right)$ where `S` is observed speedup and `n` is thread count, as derived from Amdahl's law [1]. These figures are referenced in Section 5 for cross-task synthesis.

2. Copy-on-Write Demonstration (Rust)

Design

`cow` is a Rust binary that allocates configurable memory blocks (64-128 MB), initialises them with deterministic data, forks with `unsafe { fork() }`, and samples both RSS and `Private_Dirty` from `/proc/<pid>/status` and `/proc/<pid>/smaps_rollup`. IPC uses a POSIX pipe to serialise child measurements back to the parent. The child touches the first byte of every page to trigger CoW.

Command-line interface:

- `--sizes` - list of experiment sizes (MB, >=16),
- `--output` – optional CSV path (`child_post_fork_rss`, `child_post_write_rss`, `private_dirty`, `touch_ms`).

Results

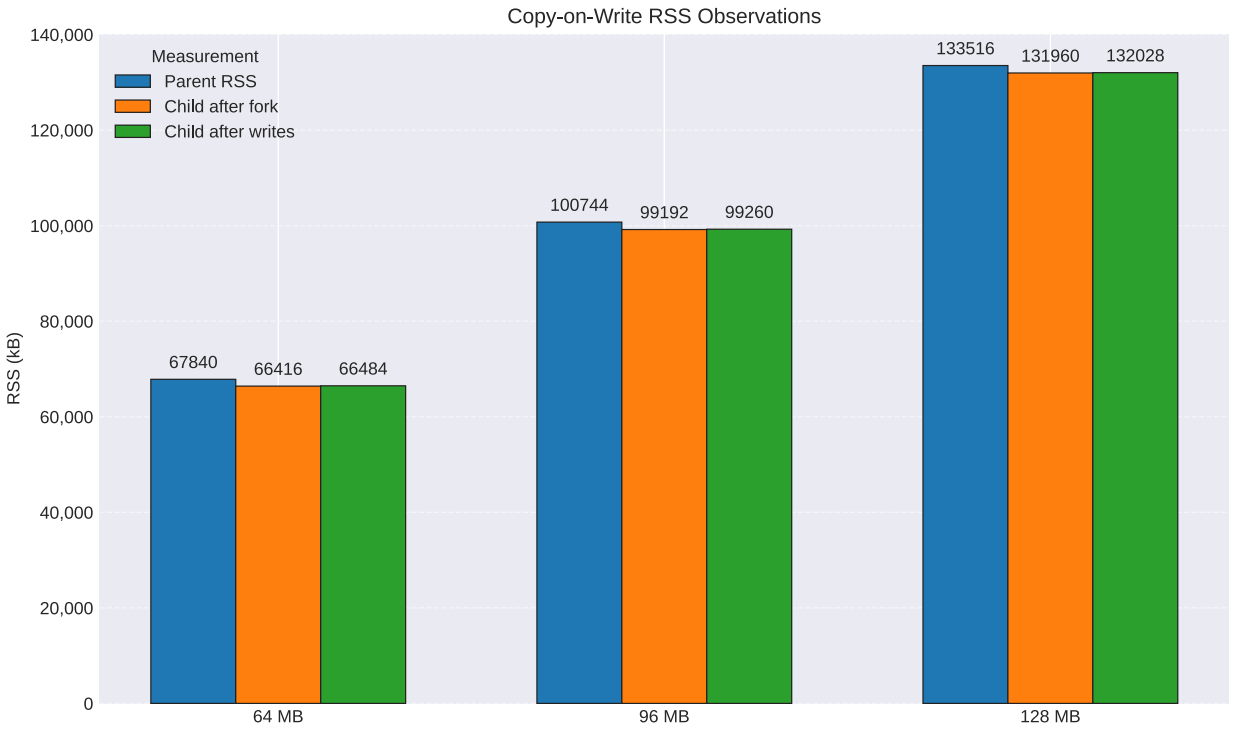

Figure 3: RSS impact of CoW across allocation sizes.

Key findings (from `data/cow_summary.csv`):

- Immediately after `fork`, the child reports <0.1 MB of `Private_Dirty`, confirming page sharing; RSS closely matches the parent's footprint yet remains mostly clean.
- Touching one byte per page inflates `Private_Dirty` to >98% of the allocation (e.g. 65,580 kB for 64 MB), reflecting full page duplication.
- Touch latency scales with size (21 ms -> 45 ms between 64 and 128 MB), consistent with increasing page faults.

This experiment validates the Linux CoW implementation: virtual memory maps are shared until first write, at which point the child incurs copy penalties proportional to the number of modified pages.

3. Deadlock Laboratory (Rust)

The `deadlock` program provides three modes.

3.1 Deadlock Avoidance (Banker's Algorithm)

Using the classic example from Silberschatz et al. [2], the tool computes a safe sequence $[P_1, P_3, P_4, P_0, P_2]$ and evaluates ad-hoc requests. For instance, P1′s request $[1, 0, 2]$ is deemed safe, whereas P0′s $[3, 3, 0]$ would violate safety and is rejected. The implementation constructs the `need` matrix and iteratively tests for feasible completion sequences.

3.2 Deadlock Detection

Three worker threads (P_0, P_1, P_2) request resources cyclically: each locks one unit then waits for the next resource, satisfying Coffman's conditions. The manager tracks allocations and waiters, then builds a wait-for graph. When threads block, a monitor thread runs cycle detection - once $[P_2 \to P_0 \to P_1 \to P_2]$ is detected, the program halts workers by signalling `stop_all`.

3.3 Deadlock Resolution

With `--mode resolution`, the monitor selects a victim (highest process ID in the cycle), marks it terminated, and reclaims its allocations. The remaining processes resume, complete their second requests,

and release resources cleanly. This models practical recovery strategies such as process termination followed by resource reallocation.

4. Discussion

- **Parallel factorisation**: Prime-heavy workloads benefit most from OpenMP because there are no early exits, yet overall speedup is limited by serial reduction and modulus costs. Dynamic scheduling with medium chunks offers the best balance between cache reuse and load distribution.
- **Copy-on-Write**: RSS figures affirm that Linux maintains shared mappings post-`fork` until a write triggers per-page duplication. The experiment also quantifies the time penalty for touching every page.
- **Deadlocks**: Banker's algorithm depends on accurate maximum claims; the simulation demonstrates how unsafe requests are rejected proactively. The detection/resolution workflow highlights that cycle detection must be paired with policy (victim selection) to restore progress.

5. Conclusion

All three assignments were implemented and exercised with reproducible tooling. The factorisation benchmark exposes how algorithmic structure influences parallel efficiency, the CoW study attributes memory inflation directly to page faults, and the deadlock laboratory ties together avoidance, detection, and recovery. The provided scripts and CSV artefacts enable re-running the experiments and regenerating every figure without manual intervention.

References

[1] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *AFIPS Spring Joint Computer Conference*, 1967, pp. 483–485. doi: 10.1145/1465482.1465560.

[2] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Wiley, 2018.