Betriebssysteme

- Übung 1 -

Franz Pawlus
1422169

22. November 2025

1 Einleitung

System-Calls stellen, wie in der Vorlesung behandelt, den einzigen Mechanismus dar, durch den Anwendungsprogramme aus dem User Space Operationen im Kernel Space anfordern können. Dieser Übergang ist mit Overhead verbunden, da ein vollständiger Kontextwechsel vom User Mode zum Kernel Mode und zurück durchgeführt werden muss (siehe Beispielprogramm `cs.cpp`).

In dieser Übung standen uns vier verschiedene Operationen zur Auswahl, um diese Latenzen zu untersuchen. Ich habe mich schlussendlich für die File-I/O-System-Calls `open()`, `read()`, `write()` und `close()` (ORWC) entschieden und diese unter verschiedenen Messbedingungen getestet *(Caching vs. kein Caching, WSL2-Mount vs. nativen Linux-Filesystemen)*.

2 Methodik

2.1 Hard- und Softwareumgebung

Meine Messungen habe ich auf meinem privaten Heimrechner mit einer AMD Ryzen 7 5800X (8-Core, 16 Threads) unter Windows 11 durchgeführt. Als Testumgebung nutze ich das Windows Subsystem for Linux (WSL2) mit Ubuntu 24.04.1 LTS (Kernel 5.15.167.4). Der Code wurde mit g++ 13.3.0 kompiliert. Ähnlich zum Beispiel aus der Vorlesung habe ich die Stopwatch-Klasse aus dem Vorlesungs-Repository benutzt (nutzt intern `std::chrono::high_resolution_clock`)

2.2 Testprogramm-Implementierung

Mein implementiertes Testprogramm orientiert sich strukturell stark an dem Kontextwechsel-Beispiel der Vorlesung (`cs.cpp`). Ich habe jedoch einige Modifikationen vorgenommen, um die vier Operationen (Open, Read, Write, Close) isoliert voneinander messen zu können, u.a.:

1. **Fehlerbehandlung:** Nach jedem System-Call wird mittels assert geprüft, ob die Operation erfolgreich war (z.B. $fd \geq 0$). Dies verhindert, dass fehlgeschlagene Aufrufe die Statistik verfälschen.

2. **Warm-up Phase bei Caching Szenarien:** Die ersten 100 Iterationen werden nicht aufgezeichnet, um "Cold-StartEffekte (z.B. Page Faults beim ersten Programmzugriff) zu reduzieren.

Anbei ein Auszug aus meiner Implementierung (Cached-Variante), der meine Modifikationen verdeutlicht:

```
for (int i = 0; i < N + WARMUP; i++) {
    bool record = (i >= WARMUP); // Flag f r Warm-up

    // Messung OPEN
    sw.reset(); sw.start();
    int fd = open("test.dat", O_RDWR | O_CREAT | O_TRUNC, 0644);
    sw.stop();
    assert(fd >= 0); // Sicherstellung der Datenintegritaet
    double open_us = sw.elapsed_microseconds();

    // Messung WRITE
    sw.reset(); sw.start();
    ssize_t written = write(fd, buf, BUFFER_SIZE);
    sw.stop();
    assert(written == BUFFER_SIZE);
    double write_us = sw.elapsed_microseconds();

    // lseek() (wird nicht gemessen)
    lseek(fd, 0, SEEK_SET);

    // Messung READ & CLOSE (analog zu oben)...

    if (record) { /* Speichern in CSV */ }
}
```

Der `reset()`-Aufruf vor jedem `start()` habe ich bewusst gesetzt, da die Stopwatch-Klasse akkumulierte Zeiten speichert:

```
accumulated_time_ += Clock::now() - start_time_;
```

Außerdem wurde der System Call `lseek()` zwischen `write()` und `read()` nicht gemessen, da dies die reine Kernel-Verweildauer des `read()`-Syscalls verfälschen würde.

2.3 Testszenarien

Ich habe mich zusätzlich zur Aufgabenstellung entschieden, den Einfluss von Caching und Virtualisierung zu messen. Dabei entstanden vier verschiedenen Testszenarien:

1. **Linux Native (Cached):** 10.000 Messungen auf dem nativen ext4-Dateisystem der WSL-Instanz - inkl. "Warm-Up".

2. **Linux Native (No-Cache):** 100 Messungen unter *Cold-Start*-Bedingungen. Zudem werden in jedem Schleifendurchlauf die Caches (Page, Dentry, Inode) mit dem Befehl
(`echo 3 > /proc/sys/vm/drop_caches`) geleert und `sync()` schreibt zuvor ausstehende Änderungen sofort auf die Disk. *(Anmerkung: Mir ist bewusst, dass die nach `open()` folgenden Operationen wieder von Caching-Effekten profitieren)*

3. **WSL Mount (Cached):** Wiederholung der 10.000 Messungen aus Szenario 1. Nun aber auf dem gemounteten Windows-NTFS-Dateisystem (`/mnt/c/`). Ich will so den Overhead untersuchen, der durch die Protokoll-Übersetzung (`9P`) zwischen Linux-Kernel und Windows-Host entsteht.

4. **WSL Mount (No-Cache):** Analog zu Szenario 2. 100 Messungen mit Cache-Leerung auf dem NTFS-Mount.

2.4 Statistische Auswertung

Die Ergebnisse der vier verschiedenen Szenarien habe ich in .csv-Dateien gespeichert und anschließend mit `Polars`, `Numpy` und `Matplotlib` (Python-Bibliotheken) analysiert. Nachfolgend stelle ich meine Ergebnisse und Interpretationen dar.

3 Ergebnisse und Diskussion

3.1 Performance unter Linux Native

Zuerst habe ich die Basis-Performance auf dem nativen `ext4`-Dateisystem untersucht. Tabelle 1 zeigt die statistischen Kenngrößen für das Szenario mit aktivem Cache (N=10.000).

Tabelle 1: Statistische Kenngrößen (Linux Native, Cached)

Syscall	Mittelwert (μs)	Min (μs)	Max (μs)	StdAbw (μs)	95%-KI (μs)
OPEN	**124.88**	86.08	2485.19	41.50	[124.07, 125.70]
WRITE	**5.70**	4.73	42.67	1.25	[5.67, 5.72]
READ	**0.86**	0.69	11.35	0.25	[0.85, 0.86]
CLOSE	**12.76**	10.81	197.70	7.32	[12.62, 12.91]

Die Messungen zeigen, dass `read()` mit durchschnittlich **0.86** μs die mit Abstand schnellste Operation ist. Dies liegt vermutlich an der Effizienz des Linux Page Cache: Da die Daten unmittelbar zuvor geschrieben wurden, liegen sie noch im RAM (Page Hit) und müssen nicht von der Disk geholt werden. Die `write()`-Operation ist mit durchschnittlich **5.70** μs etwas langsamer, da der Kernel die Daten als "dirty"markieren, Zeitstempel

aktualisieren und Dateisystem-Buchhaltung betreiben muss. `close()` benötigt mit ca. **12.76** μs noch mehr Zeit, da hier finale Aufräumarbeiten anfallen. Der `open()`-Call dominiert die Gesamtlatenz mit ca. **125** μs deutlich, da der Kernel erst den Pfad zur Datei ermitteln und alle zugehörigen Metadaten prüfen und laden muss.

3.2 Das Open-Paradoxon (Cached vs. No-Cache)

Ein Ergebnis hat mich besonders überrascht: Beim Vergleich der Szenarien mit und ohne Cache (siehe Abbildung 1) scheint `open()` ohne Cache deutlich schneller zu sein.

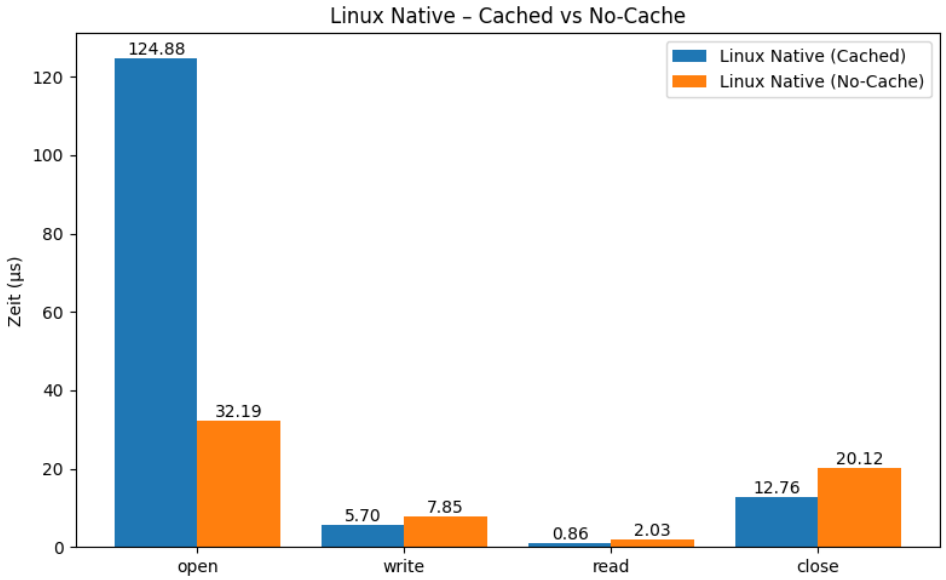

Abbildung 1: Vergleich Linux Native Cached vs. No-Cache

Auf den ersten Blick wirkt es unlogisch, dass `open()` ohne Cache schneller ist (32 μs vs. 125 μs). Ich führe diesen Effekt auf meine Testmethodik zurück:

1. **Im Cached-Szenario** läuft mein Programm in einer sehr engen Schleife. Das Dateisystem steht unter hoher Last, weil es ständig Journaling-Metadaten aktualisieren und interne Locks verwalten muss, wodurch Konkurrenz um Ressourcen entsteht.

2. **Im No-Cache-Szenario** sorgt der `drop_caches`-Aufruf für eine deutliche Pause vor jeder Iteration; wenn anschließend `open()` ausgeführt wird, trifft der Aufruf auf ein komplett unbelastetes Dateisystem (idle).

Ich vergleiche hier also weniger den eigentlichen Cache-Effekt, sondern vor allem den Unterschied zwischen einem ausgelasteten System und einem System im Leerlauf (idle). Bei `read()`, `write()` und `close()` tritt hingegen das erwartete Verhalten auf: Sie profitieren zwar von den Cache-Strukturen, die durch die vorherige `open()`-Operation wieder aufgebaut wurden, sind aber trotzdem langsamer, weil `drop_caches` und `sync()` das System vor jeder Iteration in einen `Idle`-Zustand zwingen, aus dem CPU und Dateisystem erst wieder hochfahren müssen.

3.3 Der Overhead von WSL2

Zusätzlich wollte ich wissen, wie viel Leistung die Virtualisierung kostet. Die Ergebnisse auf dem gemounteten Windows-Laufwerk (`/mnt/c/`) zeigen einen massiven Performance-Einbruch (Abbildung 2).

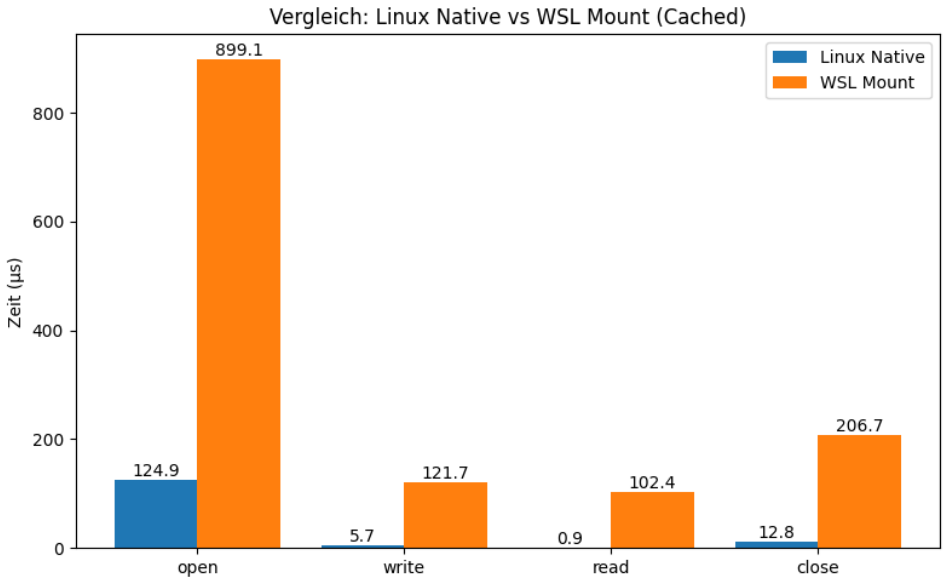

Abbildung 2: Vergleich Linux Native vs. WSL Mount (beide Cached)

Die starken Verzögerungen entstehen durch die Architektur von WSL2: Das Linux läuft in einer eigenen, leichtgewichtigen VM, deren internes `ext4`-Dateisystem direkt innerhalb der VM verarbeitet wird (vgl. Abbildung 3). Zugriffe auf `/mnt/c` gehören jedoch zum Windows-NTFS und müssen daher über die VM-Grenze hinweg an den Windows-Host delegiert werden. Diese Kommunikation erfolgt über das `9P`-Protokoll und verursacht für jeden einzelnen System-Call einen zusätzlichen Übertragungs- und Übersetzungsaufwand. Besonders kleine Operationen wie ein 1-KB-`read()` sind davon stark betroffen, weshalb sich die Latenz hier um den Faktor 119 erhöht.

Ein interessantes Detail zeigt sich beim Vergleich der WSL-Werte der Cached und No-Cached-Szenarien (siehe Abbildung 4): Im Gegensatz zum nativen Linux ist `open()` hier in der No-Cache-Variante tatsächlich langsamer (1177 μs) als in der Cached-Variante (899 μs). Das ist ein weiteres Indiz für meine zuvor beschriebene Interpretation: Während im nativen Linux die "No-Cache-Pause" die interne Ressourcen-Konkurrenz auflöst, zwingt der `drop_caches`-Befehl im WSL-Szenario den Linux-Kernel dazu, sämtliche Pfad-Informationen zu vergessen. Für das erneute `open()` muss Linux den gesamten Pfad erneut über das `9P`-Protokoll vom Windows-Host anfordern, was aufgrund der hohen Protokoll-Latenz die Zeitersparnis durch den Leerlaufzustand (Idle) zunichte macht.

[1] https://medium.com/@boutnaru/the-windows-concept-journey-wsl2-windows-subsystem-for-linux-version-2-5500fd847543

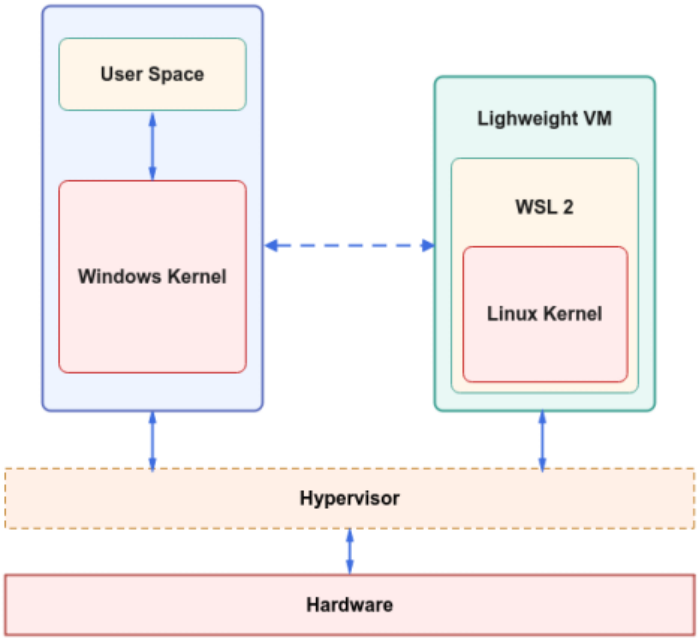

Abbildung 3: Schematische Darstellung des WSL-Subsystems[1]

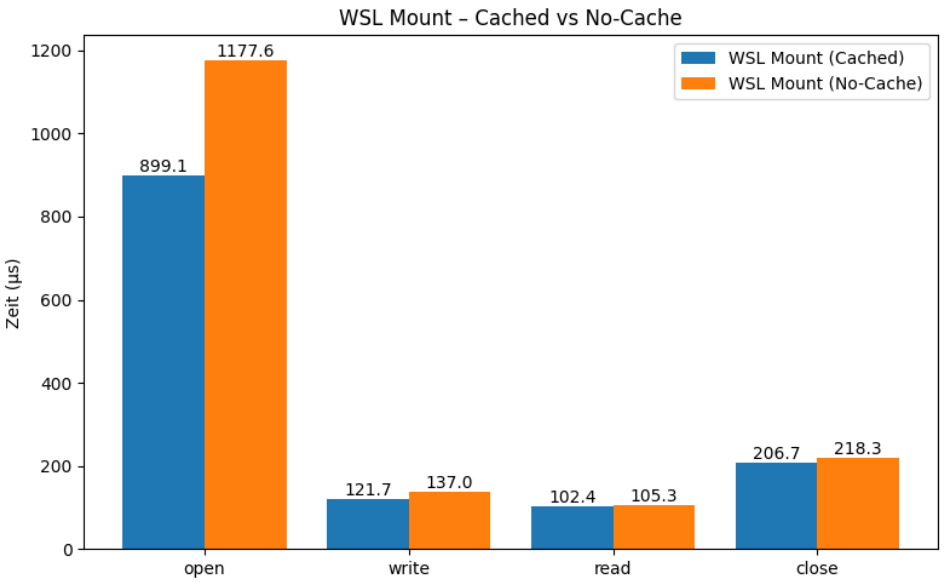

Abbildung 4: Vergleich WSL Mount Cached vs. WSL Mount No-Cache