Technisches Vorgehen

Meine Umsetzung der Aufgabe besteht aus zwei zentralen Komponenten. Die eigentliche Ausführung der Zeitmessungen wurde in C++ implementiert. Die Skripte habe ich anschließend über die Open-Source-Bibliothek Pybind[1] in Python eingebettet, um dort die Datenauswertung, die grafische Darstellung sowie das automatische Kompilieren der C++-Module zu realisieren. Ergänzend stellt der Python-Code einen Konsolendialog bereit, über den sämtliche Funktionalitäten bequem aufrufbar sind.

Da die Zeitmessung der verschiedenen Funktionsbereiche stets nach demselben Ablauf erfolgt, habe ich mich für ein Entwurfsmuster entschieden, das dem *Strategy Pattern* entspricht.[2] Alle C++-Module exponieren eine einheitliche `run`-Methode, in der die eigentlichen Messungen mithilfe einer leicht angepassten Variante der bereitgestellten `Stopwatch`-Klasse umgesetzt werden. Die Methoden liefern an Python eine Liste von Instanzen der Hilfsklasse zurück, wobei in Python ausschließlich die Konvertierungsfunktionen für die Zeiteinheiten Sekunden, Millisekunden, Mikrosekunden und Nanosekunden exponiert werden. Dadurch kann die `LatencyTest`-Klasse in Python sämtliche Skripte einheitlich verarbeiten, sodass weitere Tests hinzugefügt werden können, ohne dass Anpassungen im Python-Code erforderlich wären.

Das die ausgegebene Liste einen hohen Speicherbedarf verursacht ist mir bewusst. Die Aufgabenstellung fordert jedoch die Berechnung des Medians. Während inkrementelle Verfahren statistische Größen wie den Mittelwert exakt bestimmen können,[3] lässt sich der Median ohne Speicherung der vollständigen Stichprobe nur approximieren. Da somit ohnehin eine recht große Historie aller Messwerte vorgehalten werden muss, habe ich mich für die speicherintensivere, aber in diesem Anwendungsfall gut vertretbare Lösung entschieden. Zusätzlich stellen die Bibliotheken Numpy[4] und Scipy[5] weitere Funktionen zur Verfügung, welche die Auswertung der Datensätze weiter vereinfachen.

Aktives vs. blockierendes Warten

Hypothese

Aktives Warten (Busy Waiting) bedeutet, dass ein Thread in einer Schleife kontinuierlich prüft, ob ein Lock verfügbar ist. Dabei verbraucht der prüfende Thread permanent CPU-Zeit, was besonders bei scheduler-bedingten Verzögerungen zu hohen und stark schwankenden Latenzen führen kann.[6] *Blockierendes Warten* über Semaphore suspendiert den wartenden Thread dahingegen und vermerkt ihn in der zugehörigen Warteschlange des

[1] *pybind11: Seamless interoperability between C++ and Python*. Version v3.0.1. Besucht: 17.11.2025. 2025. URL: `https://github.com/pybind/pybind11`.

[2] Refactoring.Guru. *Strategy*. `https://refactoring.guru/design-patterns/strategy`. 2025.

[3] *Incremental Averaging*. `https://blog.demofox.org/2016/08/23/incremental-averaging/`.

[4] *NumPy: The fundamental package for scientific computing with Python*. Version 2.3.5. Besucht: 17.11.2025. 2025. URL: `https://github.com/numpy/numpy`.

[5] *SciPy: Open-source software for mathematics, science, and engineering*. Version v1.16.3. Besucht: 17.11.2025. 2025. URL: `https://github.com/scipy/scipy`.

[6] *Busy waiting*. `https://en.wikipedia.org/wiki/Busy_waiting`.

Semaphors.[7] Der Scheduler betrachtet diesen Thread dadurch als nicht ausführbar, bis das Semaphor ein Signal erhält. Somit werden keine CPU-Ressourcen verbraucht, und es entstehen typischerweise sehr geringe Wake-Up-Latenzen. Aus diesen Begebenheiten ergibt sich folgende Hypothese:

1. Aktives Warten erzeugt hohe, schedulerabhängige Latenzen mit großer Varianz.
2. Blockierendes Warten erzeugt stabile und bemerkbar niedrigere Latenzen.

Implementierung

Beide Messverfahren verwenden eine ähnliche Vorgehensweise: Jede Iteration initialisiert zwei Threads: einen wartenden und einen freigebenden. Beide Threads teilen sich eine gemeinsame `Stopwatch`-Instanz, wobei der Hauptthread die Messung unmittelbar vor der Freigabe des jeweiligen Synchronisationsobjekts startet und der Wartethread sie stoppt, sobald er die Freigabe erkennt. Dadurch wird ausschließlich die Latenz zwischen Freigabe und tatsächlichem Weiterlaufen des wartenden Threads gemessen. Die Synchronisationsprimitive (Spinlock bzw. Semaphore) werden für jeden Messdurchlauf neu angelegt und nach Abschluss der Messung wieder freigegeben.

Beim aktiven Warten wird zu Beginn jeder Iteration ein neuer Lock erzeugt und sofort in den gesperrten Zustand versetzt. Ein Wartethread prüft anschließend in einer Spin-Schleife, ob der Lock freigegeben wurde. Der Hauptthread startet die Zeitmessung unmittelbar vor der Freigabe. Sobald der Wartethread die Freigabe beobachtet und die Schleife verlassen hat, stoppt er die Uhr.

Beim blockierenden Warten wird jeweils eine neue Semaphore mit Startwert 0 initialisiert. Der Wartethread ruft sofort `sem_wait()` auf und blockiert im Kernel. Der Hauptthread startet die Messung direkt vor `sem_post()`, das den Wartethread weckt. Nach dem Aufwachen beendet der Wartethread die Messung.

Ergebnisse & Schlussfolgerung

Kennzahl	**Aktives Warten**	**Blockierendes Warten**
Mittelwert	372,77 ms	5,06 μs
Median	373,95 ms	3,69 μs
Minimum	0,421 μs	1,003 μs
Maximum	752,108 ms	390,859 μs
Std.-Abw.	213,68 ms	10,41 μs
95%-CI	[368,58; 376,96] ms	[4,85; 5,26] μs

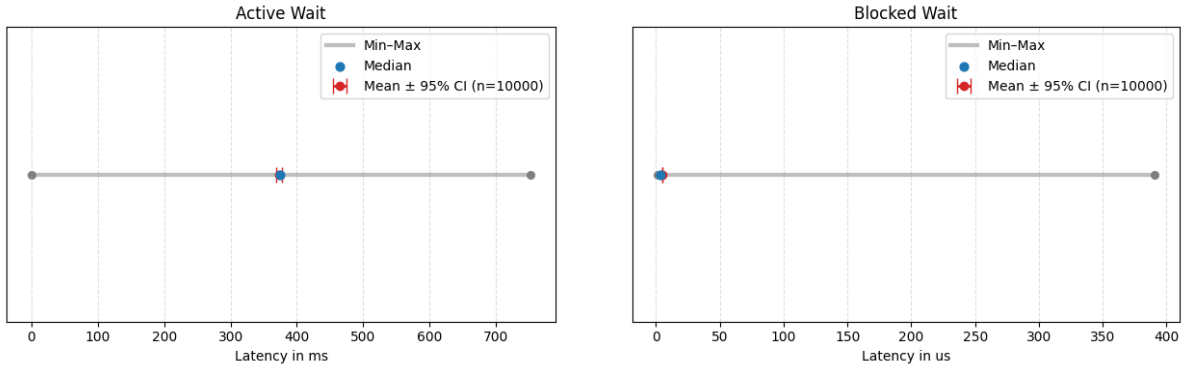
Tabelle 1: Vergleich der beiden Warteverfahren (jeweils 10 000 Messungen).

[7] Ahmed Ali-Eldin. *Linux Kernel Synchronization and Timers*. `https://lass.cs.umass.edu/~shenoy/courses/spring20/lectures/Lec20.pdf`. 2020.

Beim aktiven Warten liegt der Mittelwert mit 372,77 ms im Millisekundenbereich und damit signifikant höher als beim blockierenden Verfahren. Der Median (373 ms) ist nahezu identisch. Die extremen Werte – von wenigen Nanosekunden bis zu über 750 ms – und die sehr hohe Standardabweichung (213,68 ms) zeigen eine massive Streuung. Auch das 95%-Konfidenzintervall von 368–377 ms belegt, dass die Messwerte stark schwanken und dadurch signifikant vom Scheduling abhängig sind.

Beim blockierenden Warten befinden sich Mittelwert (5,06 μs) und Median (3,69 μs) im niedrigen Mikrosekundenbereich. Minimum (1,003 μs) und Maximum (390,859 μs) zeigen zwar einzelne Ausreißer, die Standardabweichung von 10 μs ist aber erheblich kleiner als beim aktiven Warten. Auch das enge 95%-Konfidenzintervall von 4,85–5,26 μs bestätigt die hohe Stabilität im Vergleich zur alternativen Methodik.

Zusammenfassend entsprechen die Messwerte der Hypothese: Aktives Warten führt zu hohen und stark schwankenden Latenzen, während das blockierende Warten kurze, konsistente Reaktionszeiten liefert. Für die Praxis bedeutet dies, dass blockierende Synchronisationsmechanismen gegenüber aktivem Warten klar zu bevorzugen sind. Entwickler sollten Spinlocks nur dann einsetzen, wenn schedulerbedingte Verzögerungen ausgeschlossen werden können. In allen anderen Fällen führen Semaphore oder vergleichbare Primitive zu vorhersehbarem Verhalten, geringerer CPU-Last und besserer Skalierbarkeit unter Last.



(a) Aktives Warten (b) Blockierendes Warten

Abbildung 1: Vergleich der Latenzverteilungen beider Warteverfahren.

Andere Messungen

ORWC und Pipelines

ORWC steht für *Open–Read–Write–Close* und bezeichnet die Abfolge grundlegender Dateisystemoperationen: Eine Datei wird geöffnet, ein kleiner Inhalt gelesen, etwas geschrieben und anschließend wieder geschlossen. Der Ablauf bildet typische I/O-Zugriffe ab und umfasst damit sowohl Kernel- als auch Dateisysteminteraktionen. Eine *Pipeline* ist ein leichtgewichtiges Kommunikationsmittel zwischen zwei Prozessen oder Threads.[8] Sie be-

[8] Tabrez Ahmed. *Inter-Process Communication Using Pipes and Signals in C++ on Unix based machine*. `https://tabreztalks.medium.com/inter-process-communication-using-pipes-and-signals-in-c-on-unix-based-machine-6f3c862ecfa5`. 2024.

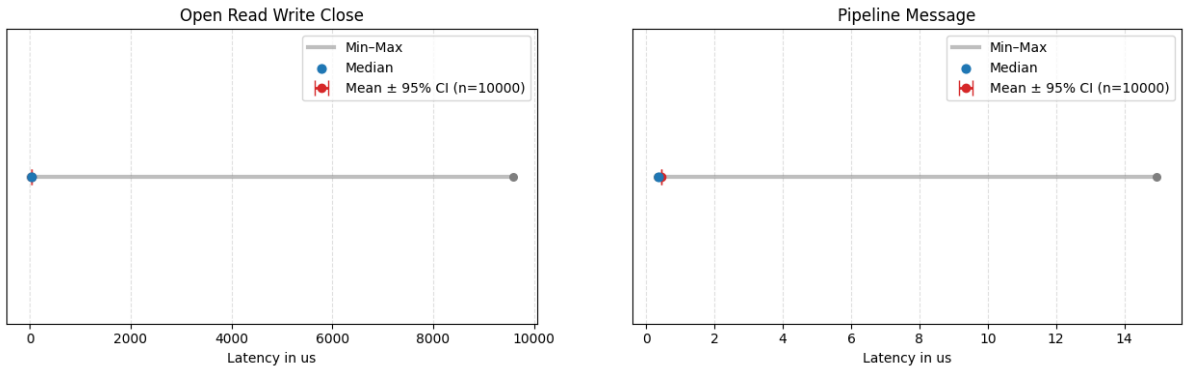
steht aus einem verbundenen Lese- und Schreibende und ermöglicht das Übergeben kleiner Datenmengen direkt im Speicher, ohne dass das Dateisystem beteiligt ist. Dadurch eignet sie sich besonders für schnelle und einfache Interprozesskommunikation.

Kennzahl	**ORWC**	**Pipeline**
Mittelwert	33,69 μs	0,437 μs
Median	35,20 μs	0,35 μs
Minimum	16,96 μs	0,326 μs
Maximum	9581,46 μs	14,945 μs
Std.-Abw.	96,33 μs	0,258 μs
95%-CI	[31,80; 35,58] μs	[0,432; 0,442] μs

Tabelle 2: Vergleich von ORWC und Pipelines (jeweils 10 000 Messungen).

Die ORWC-Operation zeigt Latenzen im zweistelligen Mikrosekundenbereich und weist eine deutlich sichtbare Streuung auf. Besonders die vereinzelten Ausreißer im Millisekundenbereich verdeutlichen, dass Dateisysteminteraktionen stark von Kernel- und I/O-Zuständen abhängig sind und daher nur eingeschränkt deterministisches Verhalten bieten. Für Anwendungen, die viele kleine Dateien öffnen, lesen oder schreiben müssen, kann sich dieses Verhalten spürbar auf die Gesamtlaufzeit auswirken.

Die Pipeline-Kommunikation arbeitet dagegen durchgängig im Submikrosekundenbereich und zeigt nahezu keinerlei Varianz. Die enge Verteilung deutet weiter auf eine sehr stabile Ausführung hin, da die Operation vollständig im Speicher stattfindet und keine Interaktion mit dem Dateisystem erfordert. Für latenzkritische IPC-Szenarien ist dieser Mechanismus daher deutlich besser geeignet und ermöglicht eine verlässliche Planung von Kommunikationsmustern zwischen Prozessen oder Threads.



(a) ORWC (b) Pipeline

Abbildung 2: Vergleich der Latenzverteilungen beider Verfahren.