Jakob Schouren

Betriebssysteme, Übungsblatt 1, Winter 2025

Aufgabe 1: ORWC – Open Read Write Close

Die Zielsetzung dieser Aufgabe bestand darin, eine repräsentative Stichprobenanzahl an Latenzmessungen für das Lesen du Schreiben von Dateien (möglichst betriebssystem-nah) zu ermitteln und darauf aufbauend eine statistische Auswertung mit bewährten Größen durchzuführen und die Ergebnisse auch in graphischer Form wiederzugeben.

Voraussetzungen:

Da die Aufgabe hauptsächlich auch einem Laptop mit Windows 11 Home Version 24H2 bearbeitet wurde, ist auch die Wahl des Betriebssystems auf Windows gefallen. Als Entwicklungsumgebung wurde VS Code mit einer entsprechenden Extension für C/C++ verwendet. Es wurden insgesamt drei Beispielanwendungen mit dem Ziel der Latenzmessung für ORWC angelegt, wobei zwei die Windows-API verwenden (sowohl in C als auch C++) und das andere die Standard-C++-Library fstream. Für die C++-Programme wurde als Compiler g++ und für die Zeitmessung die bereits vorhandene Stopwatch-Klasse in stopwatch.cpp verwendet, für das C-Programm wurde analog hierzu ein zweckmäßiger Stopwatch-Strukturtyp in Cstopwatch.c implementiert, der ebenfalls die Windows-API verwendet. Um das C-Programm zu compilen, kommt gcc zum Einsatz. Pro Beispielanwendung wurden jeweils 100000 Messwerte erhoben, die in eine gleich formatierte csv-Datei geschrieben wurden. Die anschließende Auswertung der Messwerte erfolgt über ein Jupyter-Notebook bzw. in einem Python-File, wobei alle verwendeten Module in einer requirements.txt festgehalten wurden. Insbesondere bei den C/C++-Programmen wurde als Debugging-Hilfe sowie für weitere Empfehlungen zur Umsetzung GitHub Co-Pilot verwendet.

Ergebnisse:

Bezüglich der ermittelten Latenzen zeigt sich über alle Beispielanwendungen hinweg ein ziemlich eindeutiges Bild. Bei den Programmen, welche die Windows-API verwendet haben, dauert die Close-Operation am längsten (ca. 116-134 µs) und weist eine knapp höhere Latenz auf als Open (ca. 107-113 µs). Bei der Nutzung von fstream dagegen benötigt Open (ca. 141 µs) etwas mehr Zeit im Vergleich zu Close (ca. 112 µs). In allen drei Varianten landet Write jedoch latenztechnisch auf dem dritten Platz (ca. 68-76 µs) und Read ist mit großem Abstand die schnellste Operation (ca. 7-9 µs). Die Min-Werte sind für alle drei Beispielanwendungen fast identisch und für die Max-Werte lässt sich kein richtiges Muster erkennen, es handelt sich vermutlich einfach nur um starke Ausreißer. Die restlichen Werte decken sich mit den Beobachtungen für die Mittelwerte.

Programme:

latencyWinAPI.c: Diese Anwendung verwendet für die Implementierung der ORWC-Operationen die Windows-API, genauer werden die Funktionen CreateFileA(), WriteFile(), ReadFile() und CloseHandle() eingesetzt. Zunächst wird eine Temp-Datei mit

Lese- und Schreibrechten erstellt und wieder geschlossen. Dieses Vorgehen verringert die Latenz der nachfolgenden Open-Operationen, da nicht pro Iteration eine Datei komplett neu angelegt werden muss. Danach wird in den Iterationen die bereits erstellte Datei wieder geöffnet, wobei ein Handle auf die geöffnete Datei zeigt. Dieser Handle wird zusammen mit der Stopwatch an die Funktionen für die Read-, Write- und Close-Messungen übergeben, wobei er letzlich über CloseHandle wieder geschlossen wird. Die ermittelten Latenzen werden dann zusammen mit der jeweiligen Run-ID durch Kommata separiert in eine csv-Datei geschrieben. Nach dem alle Iterationen durchlaufen wurden, wird die Temp-Datei gelöscht und die Outfile-csv geschlossen. Für die Read- und Write-Operationen werden nur Werte berücksichtigt, wo alle vier Bytes von „Test“ in die Datei geschrieben bzw. gelesen wurden, ansonsten wird -1 zurückgegeben. Die verwendete Stopwatch ist der bereits in C++ implementierten nachempfunden, verwendet aber einen Strukturtypen, da es in C keine Klassen gibt. Für die Zeitmessung kommen QueryPerformanceFrequency und QueryPerformanceCounter zum Einsatz, wobei QueryPerformanceFrequency auf dem Timer-Takt des vorhandenen High Resolution Timers beruht. QueryPerformanceCounter gibt dagegen die aktuelle Anzahl an hochgezählten Ticks aus. Die Zeit wird also gemessen, indem die gezählten Ticks zu einem Zeitpunkt mit einem anderen Zeitpunkt verglichen werden und mithilfe der Takt-Frequenz ins Verhältnis gesetzt werden, wodurch als Resultat eine Sekundenanzahl entsteht.

latencyWinAPI.cpp: Diese Anwendung ist nahezu identisch zu latencyWinAPI.c, verwendet aber für die Zeitmessung nicht die Windows-API, sondern die Stopwatch-Klasse aus stopwatch.cpp, welche von der chrono-Library Gebrauch macht.

latencyFstream.cpp: Wie latencyWinAPI.cpp auch verwendet diese Anwendung die bereits vorhandene Stopwatch-Klasse, nutzt aber abweichend zur Windows-API die fstream-Library für den Input- und Output-Stream in die Temp-Datei, wobei die Funktionen open(), read(), write() und close() verwendet werden. Das generelle Vorgehen bleibt aber genau wie bei den beiden Varianten zuvor gleich.

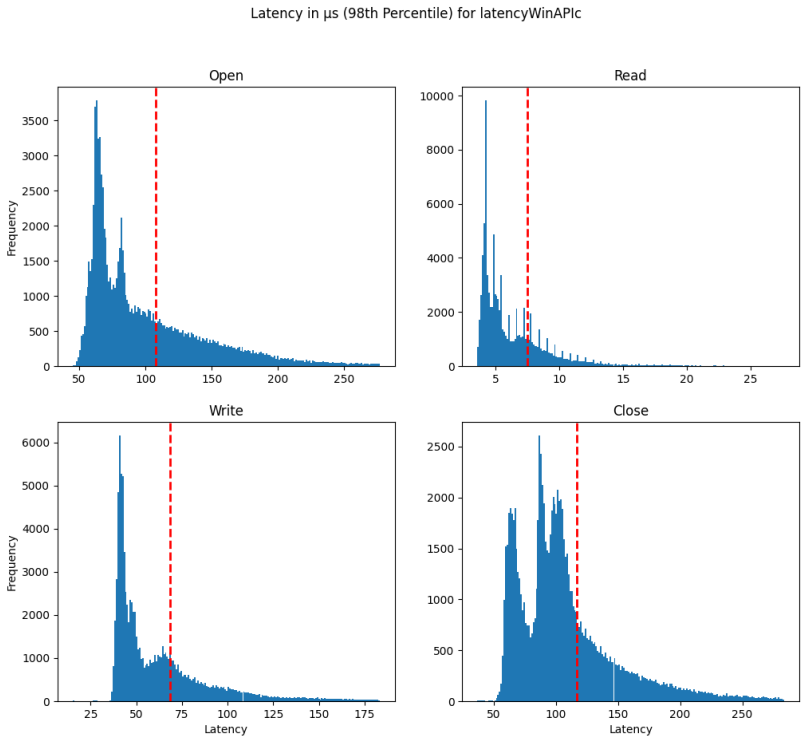
latency.py: Diese Anwendung nutzt pandas sowie scipy für die statistische Auswertung der in Results vorhandenen csv-Dateien. Berechnet wird hier Folgendes: Maximum, Minimum, Durchschnitt, Median, Standardabweichung, 95% Perzentil, 99% Perzentil, 95% Konfidenzintervall und 99% Konfidenzintervall. Im Anschluss wurde für die vier Operationen mit Matplotlib jeweils ein Subplot erstellt, welches die Frequenz der einzelnen Latenzwerte darstellt, wobei die Ausreißer entfernt wurden, da diese die Graphik verzerren würden. Zusätzlich wurde der Mittelwert als rote Linie eingefügt. Des Weiteren wurden errechneten statistischen Größen als Tabelle visualisiert. Alle Plots werden jeweils für alle durchlaufenen csv-Dateien als .png gespeichert.

Visualisierung:

Aus Platzgründen sind nachfolgend nur Graphiken für latencyWinAPIc eingefügt. Die restlichen Visualisierungen können entweder über GitHub bezogen oder über das Python-Skript selbst erzeugt werden.

Latency Statistics in µs (latencyWinAPIc)

	Max	Min	Mean	Median	Std Dev	Lower 95%	Upper 95%	Lower 99%	Upper 99%	LBound 95% Confi	UBound 95% Confi	LBound 99% Confi	UBound 99% Confi
Open	28394.8	45.2	107.43	84.0	151.23	57.6	210.6	52.5	341.9	106.49	108.37	106.2	108.66
Write	45334.9	15.3	68.43	52.1	170.28	39.4	136.6	37.8	232.8	67.37	69.48	67.04	69.82
Read	2346.9	3.6	7.53	5.4	17.06	3.9	13.3	3.8	41.5	7.42	7.63	7.39	7.67
Close	47079.0	37.0	116.69	99.8	223.99	61.7	213.0	58.2	358.7	115.31	118.08	114.87	118.52



Interpretation:

Die ermittelten Messwerte geben einen guten Einblick in die Latenzen von ORWC-Operationen, müssen aber immer Kontext der genannten Voraussetzungen sowie der spezifischen Umsetzung in den Beispielanwendungen betrachtet werden. Je nachdem welche Flags man setzt oder wie man Arbeitsschritte verlagert, können starke Abweichungen in den Latenzen entstehen. Beispielsweise macht es einen riesigen Unterschied, ob man bei jeder Iteration die Temp-Datei neu erstellt oder eine bereits vorhandene öffnet. Ein weiteres Beispiel ist das Verwenden von ios::trunc bei der Open-Operation in latencyFstream.cpp, weil dadurch der vorhandene Inhalt während des Öffnens gelöscht und nicht einfach überschrieben wird während der Write-Operation.

Auch das explizite Flushen des Dateiinhalts sorgt für zusätzlichen Overhead, der die Latenzen in Richtung der Write-Operation verschieben kann. Es lassen sich also grundsätzliche Aussagen über die Operationen treffen wie z.B. dass ein Write länger dauern sollte als ein Read, aber die genaue Verteilung der Latenzen hängt letztendlich von der Umsetzung und auch vom verwendeten Betriebssystem ab.