Betriebssysteme
Übung 2

Tillmann Faust

12.1.2025

1 Vorüberlegungen

1.1 Verteilungsmodell

Ziel der Übung ist es Daten, insbesondere das Minimum, über die Latenz verschiedener Methoden der Ressourcenteilung zwischen Threads, bzw. Prozessen, zu ermitteln. Die zu erhebenden Daten haben dabei eine kleine Menge an erwarteten Eigenschaften, die bei der Wahl des Modells zu berücksichtigen sind: Latenzen können nicht geringer als 0 sein. Weiterhin ist zu erwarten, dass die Werte innerhalb eines Versuchs ähnlich und niedrig ausfallen werden, mit wenigen eventuellen Ausreißern nach rechts. Die relative Verteilung der Werte zwischen verschiedenen Versuchsarten kann sich jedoch unterscheiden. Ein einfaches Wahrscheinlichkeitsmodell, welches diese Faktoren berücksichtigt ist die Log-Normal Verteilung, welche den natürlichen Logarithmus der erhobenen Daten betrachtet und basierend auf diesem eine Normalverteilung bildet und auswertet.

1.2 Versuchsaufbau

Um aussagekräftige Daten erheben zu können, wurde für jeden der Versuchsfälle das Experiment 100 mal Wiederholt, die Latenzen gemessen und gespeichert. Nach Ablauf der Versuchsreihe wurden die gesammelten Daten in eine CSV-Datei geschrieben um später ausgewertet werden zu können. Der Aufbau der Versuche bestand im grundlegenden immer aus zwei miteinander kommunizierenden Threads oder Prozessen, folgend worker1 und worker 2 genannt, welche von einem Hauptprogramm instantiiert werden. Anschließend wurde wiederholt der folgende Prozess durchlaufen:

1. worker1 misst Startzeit in ns
2. worker1 übermittelt die Startzeit an worker2
3. worker2 empfängt Startzeit
4. worker2 misst Endzeit in ns
5. worker 2 berechnet die Differenz
6. Die ermittelte Latenz wird abgespeichert

Alle Programme wurden in Python 3.11 geschrieben und auf einem Raspberry Pi 2 Model B mit 1GB RAM und einem BCM2835 Quad Core Prozessor @ 900MHz ausgeführt.

2 Versuchsdurchläufe

2.1 Spinlocks und Semaphore

In Aufgabe 1 und 2 wurde die Kommunikationslatenz zweier Threads im selben Prozess gemessen. Sowohl spinlock_latency.py, als auch semaphore_latency.py arbeiten, bis auf die Verwendung von einem Spinlock, bzw. Semaphor gleich.

2.1.1 Programm

Die worker interagieren über eine 3-Elementige Liste 'data' miteinander:

- Letzter gemessener Startwert
- Boolean zum Interaktionsmanagement
- Letzte ermittelte Latenz

Beide Threads 'streiten' sich während des gesamten Programmablaufs um die geteilten Ressourcen. Das bedeutet, dass beide worker teilweise auch zum Lesen der Werte den anderen Thread ausschließen müssen. Die Latenzen lassen sich vermutlich weiter reduzieren, wenn beide worker jederzeit frei die Werte in 'data' lesen können. Der Ablauf ist:

1. worker1 misst den Startwert
2. worker1 schreibt den wert in data[0]
3. worker1 schreibt True in data[1]
4. worker1 wartet in einer Endlosschleife, bis data[1] == False gilt
5. worker2 Sieht, dass data[1] == True gilt
6. worker2 Misst die Zeit erneut
7. worker2 schreibt die Differenz zwischen der neuen Messzeit und data[0] in data[2]
8. worker2 setzt data[1] auf False
9. worker1 speichert data[2] ab

2.1.2 Auswertung, Spinlocks

Relevante Werte des Verteilungsmodells:

- Minimum: 4996,776 μs
- Durchschnitt: 5099,993 μs
- Konfidenzintervall
  - Untere Schranke: 5062,707 μs
  - Obere Schranke: 5137,481 μs

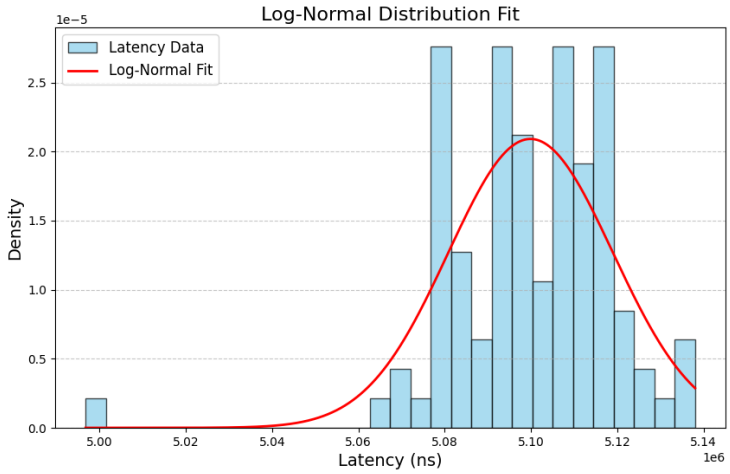

Abbildung 1: Log-Normal-Verteilung von spinlock_latenceis.py

2.1.3 Auswertung, Semaphore

Relevante Werte des Verteilungsmodells:

- Min: 5022,140 μs
- Durchschnitt: 5101,449 μs
- Konfidenzintervall
  - Untere Schranke: 5053,291 μs
  - Obere Schranke: 5149,947 μs

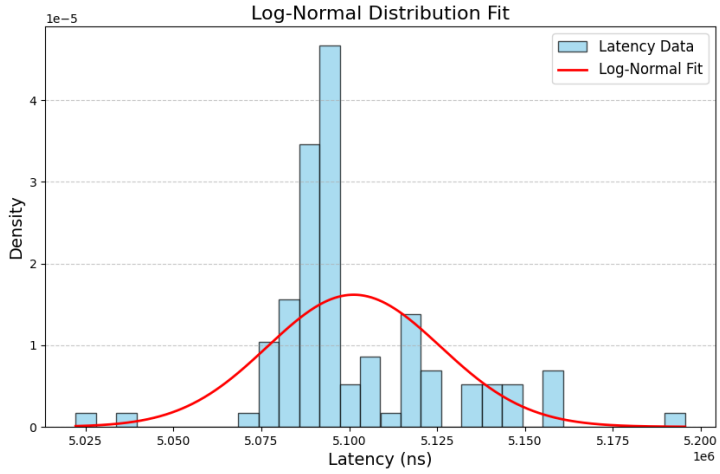

Abbildung 2: Log-Normal-Verteilung von semaphore_latenceis.py

2.2 ZeroMQ

2.2.1 Programm

Die Programme für Aufgabe 3 unterscheiden sich im Aufbau, abgesehen von der Verwendung von Threads bzw Prozessen nicht wesentlich voneinander. Der Programmablauf ist:

1. worker1 wird als Publisher instantiiert
2. worker2 wird als Subscriber instantiiert
3. worker1 misst Startzeit in ns
4. worker1 published die Startzeit
5. worker2 misst die Zeit erneut, wenn worker1 etwas published
6. worker2 berechnet die Differenz zwischen Start- und Endzeit
7. worker2 speichert die berechnete Latenz ab

2.2.2 Auswertung, Threads

Relevante Werte des Verteilungsmodells:

- Min: 392,656 μs
- Durchschnitt: 489,004 μs
- Konfidenzintervall
  - Untere Schranke: 361,471 μs
  - Obere Schranke: 647,093 μs

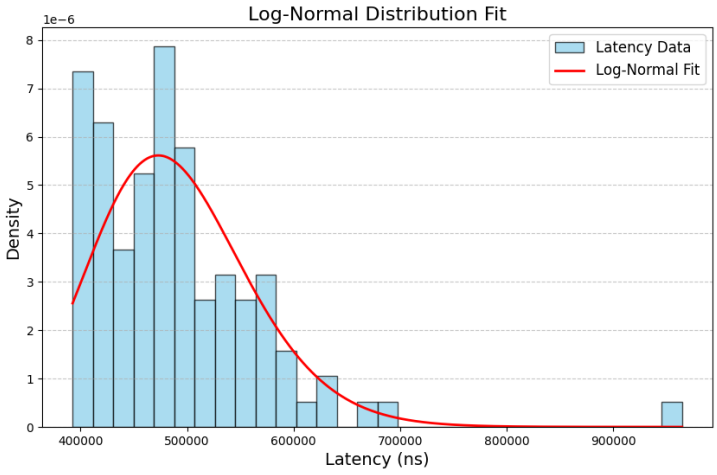

Abbildung 3: Log-Normal-Verteilung von zeromq_thread_latenceis.py

2.2.3 Auswertung, Prozesse

Relevante Werte des Verteilungsmodells:

- Min: 539,792 μs
- Durchschnitt: 675,263 μs
- Konfidenzintervall
  - Untere Schranke: 514,197 μs
  - Obere Schranke: 870,901 μs

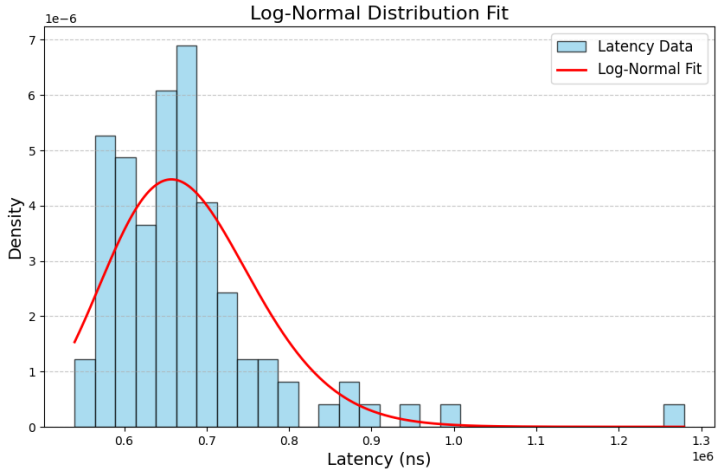

Abbildung 4: Log-Normal-Verteilung von zeromq_process_latenceis.py

2.3 Docker

Zur Umsetzung dieser Aufgabe wurde Docker-Compose verwendet.

2.3.1 Programm

Um die Kommunikation der Prozesse in unterschiedlichen Containern zu ermöglichen wurde TCP verwendet. Es werden zwei Dockerfiles definiert, welche wiederum ein Python image aufbauen, welche einen parent_worker und einen child_worker darstellen. Der Ablauf ist dann:

1. parent misst Startzeit in ns
2. parent sendet die Zeit an child
3. child liest die gesendete Zeit
4. child misst aktuelle Zeit
5. child berechnet die Latenz und speichert diese ab

2.3.2 Auswertung

Relevante Werte des Verteilungsmodells:

- Min: 353,387 μs
- Durchschnitt: 551,904 μs
- Konfidenzintervall
  - Untere Schranke: 379,737 μs
  - Obere Schranke: 775,915 μs

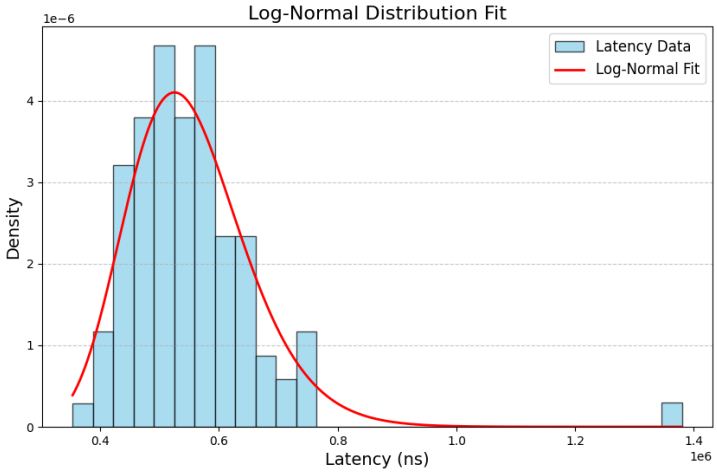

Abbildung 5: Log-Normal-Verteilung von docker_latenceis.py

3 Ergebnisse

Anhand der Minimalwerte lässt sich beobachten, dass die Latenz unter Verwendung von Spinlocks, bzw. Semaphoren signifikant höher ist, als bei anderen Methoden, inklusive der Kommunikation zwischen Docker-Containern. Zeitgleich waren diese Latenzen allerdings auch die Stabilsten, da sich die untere und obere Schranke der Latenzen im Konfidenzintervall um keine 100 μs unterscheiden. Dennoch war das Minimum bedeutend langsamer, als der Worst-Case bei jeder anderen Methode. Zusätzlich anzumerken ist, dass das gemessene Minimum eine Art von Anomalie darstellt, da bei jeder Messung immer der erste Messwert auch der geringste war, bevor sich die Latenzen nach oben stabilisieren konnten. Bei der Kommunikation mit ZeroMQ war die Thread-Anwendung erwarteterweise, und im Minimum um ca. 150 μs, schneller, als die Multiprocessing-Variante. Die Docker-Variante war, für mich genauso überraschend wie vermutet, die im Minimum schnellste Variante. Im Durchschnitt wurde Docker nur von ZeroMQ mit Threading outperformed. Zeitgleich hatte die Docker-Lösung die größte Varianz innerhalb des Konfidenzintervalls. Insgesamt lässt sich also sagen, dass, solange nur die minimale Latenz betrachtet wird, Docker mit TCP die beste

Prozesskommunikation bietet. Wenn die Möglichkeit besteht die Kommunikation zwischen Threads laufen zu lassen ist ZeroMQ eine gute Alternative, bei Interprozesskommunikation ist Docker jedoch trotzdem besser.

4 Abgabedateien

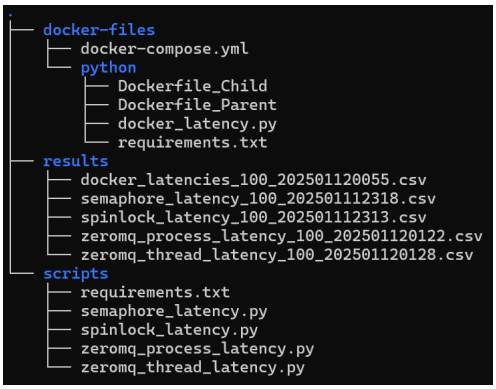

Abbildung 6: Ordnerstruktur der Abgabedateien

- 'docker-files': Alle Abgabedateien der 4. Aufgabe (excl. der csv-Dateien)
- 'results': csv-Dateien mit den Versuchsergebnissen, die in diesem Dokument verwendet wurden
- 'scripts': Die Abgabedateien der Aufgaben 1-3 (excl. der csv-Dateien)