BERICHT ZUR LATENZEN VERSCHIEDENER IPC-MECHANISMEN

Narek Grigoryan

Matrikelnummer: 1547616

Kurs: Betriebsysteme– Übung 2

Inhalt

Aufgabe 1: Kommunikation über Spinlocks

Disclaimer: In der folgenden Untersuchung wurde eine **Eigenimplementierung eines Spinlocks** mittels atomic_flag verwendet, da **macOS keine native Unterstützung** für POSIX-Spinlocks (pthread_spinlock_t) bietet. Dieser benutzerdefinierte Spinlock entspricht den Anforderungen der Aufgabenstellung, da er Busy Waiting verwendet und für Threads und Prozesse gleichermaßen geeignet ist.

Einleitung

Im der Aufgabe 1 wurde die Latenzzeit von Spinlock-Operationen experimentell untersucht. Die Aufgabe bestand darin, zwei unterschiedliche Varianten der Synchronisation zu analysieren:

1. Kommunikation zwischen zwei Threads innerhalb eines Prozesses mit einem benutzerdefinierten Spinlock.
2. Kommunikation zwischen zwei Prozessen unter Verwendung eines Shared Memory-Bereichs mit einem im Speicherbereich platzierten Spinlock.

Ziel der Experimente war es, die minimale Latenzzeit jeder Variante zu ermitteln, zu vergleichen und die Ergebnisse zu interpretieren. Die Implementierung wurde in der Programmiersprache C realisiert, wobei atomic_flag für die Spinlock-Mechanik genutzt wurde.

Methodik

Spinlock-Operationen (Threads innerhalb eines Prozesses)

Ein Spinlock ist ein Synchronisationsmechanismus, der Busy Waiting verwendet, um sicherzustellen, dass nur ein Thread den kritischen Abschnitt betreten kann. Anders als ein Mutex (*meiner erster Versuchsimplementierung war mit Mutex*) blockiert ein Spinlock den Thread nicht, sondern wiederholt aktiv die Abfrage, bis der Lock verfügbar ist.

Implementierungsdetails:

- atomic_flag: Diese Funktionalität der Bibliothek stdatomic.h wurde verwendet, um den Spinlock umzusetzen. atomic_flag_test_and_set setzt den Lock und gibt gleichzeitig dessen vorherigen Status zurück.
- Es wurden zwei Threads erstellt, die denselben Spinlock verwenden. Innerhalb jedes Threads wurden die Spinlock-Operationen („Lock“ und „Unlock“) jeweils eine Million Mal in einer Schleife ausgeführt.

- Zur Messung der Latenz wurde die Funktion clock_gettime aus der Bibliothek time.h verwendet, die hochauflösende Zeitmessungen im Nanosekundenbereich ermöglicht.

Der Ablauf war wie folgt:

1. Initialisierung eines globalen Spinlocks mit atomic_flag.
2. Erstellung von zwei Threads, die parallel auf den Spinlock zugreifen.
3. Messung der Zeitdauer für eine definierte Anzahl von Spinlock-Operationen.
4. Berechnung der durchschnittlichen Latenzzeit pro Spinlock-Operation.

Spinlock mit Shared Memory

Shared Memory erlaubt es mehreren Prozessen, auf denselben Speicherbereich zuzugreifen. Für die Synchronisation dieser Zugriffe wurde ein Spinlock verwendet, der direkt im Shared Memory abgelegt wurde. Dieser Ansatz simuliert eine typische Interprozesskommunikation (IPC) und ermöglicht es, Prozesse in getrennten Adressräumen effizient zu synchronisieren.

Implementierungsdetails:

- fcntl.h und sys/mman.h: Diese Bibliotheken wurden genutzt, um einen Shared Memory-Bereich mit shm_openzu erstellen und ihn mittels mmap in den Adressraum der Prozesse zu mappen.
- atomic_flag: Der Spinlock wurde ähnlich wie bei den Threads implementiert, jedoch im Shared Memory gespeichert, sodass sowohl Eltern- als auch Kindprozess darauf zugreifen konnten.
- Zwei Prozesse (Eltern- und Kindprozess) wurden mithilfe von fork erstellt. Beide Prozesse synchronisierten sich über denselben Spinlock im Shared Memory.
- Jeder Prozess führte eine Million Spinlock-Operationen durch, und die Latenzzeit wurde analog zur ersten Variante gemessen.

Der Ablauf war wie folgt:

1. Erstellung eines Shared Memory-Bereichs mit shm_open und Konfiguration der Größe mittels ftruncate.
2. Zuordnung des Shared Memory-Bereichs zu den Adressräumen der Prozesse mit mmap.
3. Initialisierung des Spinlocks im Shared Memory.
4. Erstellung eines Kindprozesses mit fork, der parallel zum Elternprozess arbeitet.
5. Messung der Latenzzeit für Spinlock-Operationen in beiden Prozessen.

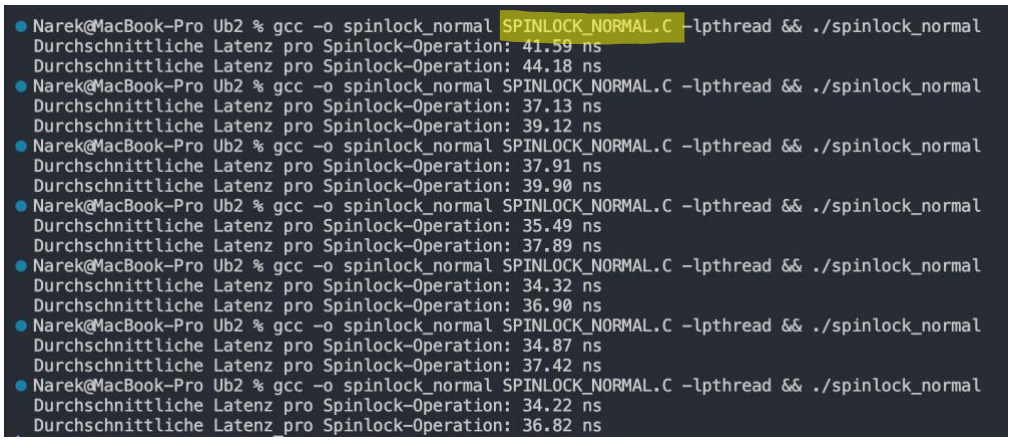
6. Nach Abschluss der Messungen wurde der Shared Memory-Bereich mit shm_unlink entfernt.

Um das 95%-Konfidenzintervall für die gemessenen Latenzzeiten zu bestimmen, wurden die durchschnittlichen Latenzen aus den 7 durchgeführten Experimenten pro Thread verwendet. Für jeden Thread wurden folgende Schritte durchgeführt:

1. **Mittelwert berechnen (x)**:
   - Der Mittelwert der Latenzzeiten wurde berechnet, um einen zentralen Wert für die Daten zu erhalten.
2. **Standardabweichung berechnen (s)**:
   - Die Standardabweichung wurde verwendet, um die Streuung der Messungen um den Mittelwert zu quantifizieren.
3. **Konfidenzintervall berechnen (95%)**:
   - Das Konfidenzintervall wurde mit der folgenden Formel berechnet: Konfidenzintervall=x ± z * (s / sqrt(n))
     - z=1.96 für eine Konfidenz von 95%.
     - n=7, da 7 unabhängige Experimente durchgeführt wurden.

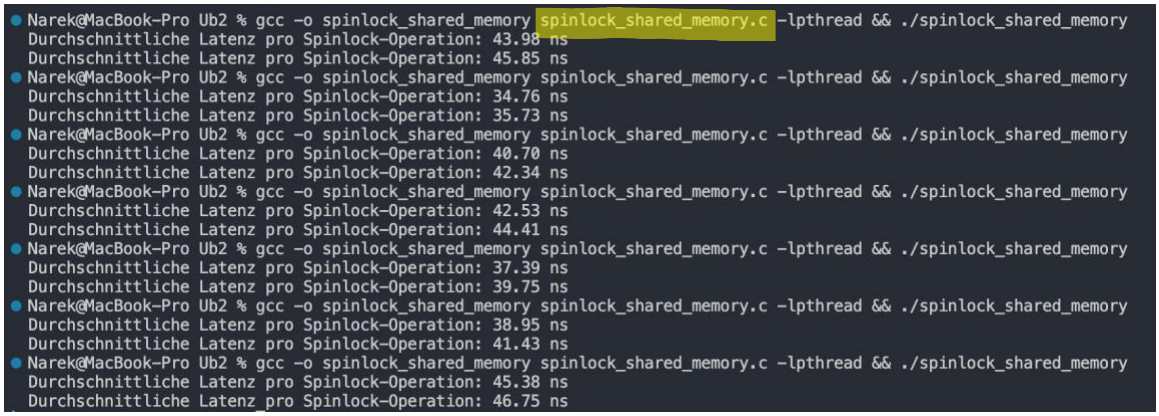
Ergebnisse

Die Ergebnisse der Latenzmessungen für beide Varianten (7 Abläufe jeweils) sind in der folgenden Bildern zu sehen:



Konfidenzintervall Thread 1 SPINLOCK NORMAL=[35.00ns,38.86ns]

Konfidenzintervall Thread 2 SPINLOCK NORMAL =[36.85ns,40.65ns]



Interpretation der Ergebnisse:

1. Threads im selben Prozess:
    - Die Latenzzeit ist minimal, da beide Threads im selben Adressraum arbeiten und der Overhead durch Speicherverwaltung entfällt.
    - Die Schwankungen sind gering, was die Effizienz des Spinlocks innerhalb eines Prozesses zeigt.
2. Prozesse mit Shared Memory:
    - Die Latenz ist leicht erhöht, da der Zugriff auf den Spinlock über den Shared Memory erfolgt. Der Overhead durch Kontextwechsel und Speicherverwaltung trägt zu den höheren Werten bei.
    - Die Ergebnisse sind dennoch konsistent und zeigen, dass Shared Memory eine effiziente Lösung für Interprozesskommunikation darstellt.

Aufgabe 2: Kommunikation über Semaphore

Einleitung

In dieser Aufgabe wurde die Effizienz von Semaphore-Mechanismen zur Synchronisation zwischen zwei Threads innerhalb eines Prozesses untersucht. Ziel war es, die minimale Latenzzeit einer Semaphore-Operation zu messen und zu analysieren. Semaphore bieten einen blockierenden Mechanismus, der Threads schlafen legt, wenn der Zugriff auf eine Ressource eingeschränkt ist, und eignen sich daher besonders für Synchronisation in zeitkritischen Anwendungen.

Methodik und Implementierung

1. Initialisierung des Semaphores:
    - Es wurde ein Semaphore (sem_open) verwendet, das innerhalb des Prozesses für beide Threads zugänglich ist.

    - Das Semaphore wurde mit einem Startwert von 1 initialisiert, um sicherzustellen, dass immer nur ein Thread gleichzeitig den kritischen Abschnitt betreten kann.
2. Thread-Operationen:
    - Zwei Threads wurden erstellt, die dieselbe Funktion ausführen.
    - Innerhalb jedes Threads wurden die folgenden Operationen in einer Schleife mit einer Million Iterationen durchgeführt:
        - sem_wait: Der Thread wartet, bis das Semaphore verfügbar ist, und verringert den Zähler.
        - sem_post: Der Thread gibt das Semaphore frei und erhöht den Zähler wieder.
    - Diese Operationen simulieren eine typische Synchronisation in einem Multithreading-Szenario.
3. Zeitmessung:
    - Zur Messung der Zeit wurde die Funktion clock_gettime verwendet, um hochauflösende Zeitstempel im Nanosekundenbereich zu erfassen.
    - Die durchschnittliche Latenzzeit pro Semaphore-Operation wurde aus der Gesamtzeit für alle Iterationen berechnet.
4. Ressourcenfreigabe:
    - Nach Abschluss der Thread-Operationen wurden die Semaphore-Ressourcen freigegeben:
        - sem_close: Schließt das Semaphore.
        - sem_unlink: Entfernt das Semaphore aus dem System.

Ergebnisse

Die Ergebnisse der Latenzmessungen für (7 Abläufe) sind in der folgenden Bild zu sehen:

```
Narek@MacBook-Pro 02_Semaphore % gcc -o semaphore_local semaphore_local.c -lpthread && ./semaphore_local
Durchschnittliche Latenz pro Semaphore-Operation: 3227.53 ns
Durchschnittliche Latenz pro Semaphore-Operation: 3227.57 ns
Narek@MacBook-Pro 02_Semaphore % gcc -o semaphore_local semaphore_local.c -lpthread && ./semaphore_local
Durchschnittliche Latenz pro Semaphore-Operation: 3194.37 ns
Durchschnittliche Latenz pro Semaphore-Operation: 3194.41 ns
Narek@MacBook-Pro 02_Semaphore % gcc -o semaphore_local semaphore_local.c -lpthread && ./semaphore_local
Durchschnittliche Latenz pro Semaphore-Operation: 3221.16 ns
Durchschnittliche Latenz pro Semaphore-Operation: 3221.14 ns
Narek@MacBook-Pro 02_Semaphore % gcc -o semaphore_local semaphore_local.c -lpthread && ./semaphore_local
Durchschnittliche Latenz pro Semaphore-Operation: 3210.22 ns
Durchschnittliche Latenz pro Semaphore-Operation: 3210.21 ns
Narek@MacBook-Pro 02_Semaphore % gcc -o semaphore_local semaphore_local.c -lpthread && ./semaphore_local
Durchschnittliche Latenz pro Semaphore-Operation: 3199.27 ns
Durchschnittliche Latenz pro Semaphore-Operation: 3199.28 ns
Narek@MacBook-Pro 02_Semaphore % gcc -o semaphore_local semaphore_local.c -lpthread && ./semaphore_local
Durchschnittliche Latenz pro Semaphore-Operation: 3191.43 ns
Durchschnittliche Latenz pro Semaphore-Operation: 3191.42 ns
Narek@MacBook-Pro 02_Semaphore % gcc -o semaphore_local semaphore_local.c -lpthread && ./semaphore_local
Durchschnittliche Latenz pro Semaphore-Operation: 3276.39 ns
Durchschnittliche Latenz pro Semaphore-Operation: 3277.66 ns
```

- Die durchschnittlichen Latenzzeiten pro Semaphore-Operation lagen im Bereich von 3191 ns bis 3227 ns.
- Die Konfidenzintervall wurde auch hier analog zu der Vorherigen Aufgabe berechnet:

    - Konfidenzintervall Thread 1 =[3195.69ns,3239.27ns]
    - Konfidenzintervall Thread 2 =[3195.76ns,3239.58ns]
- Die Ergebnisse sind konsistent mit minimalen Schwankungen.

Interpretation der Ergebnisse

1. Hohe Latenz durch blockierende Mechanismen:
    - Semaphore blockieren Threads, wenn der Zähler 0 erreicht. Dieser blockierende Mechanismus fügt Overhead hinzu, da das Betriebssystem den wartenden Thread schlafen legt und später wieder aufweckt.
    - Die Systemaufrufe sem_wait und sem_post tragen ebenfalls zur höheren Latenzzeit bei.
2. Vergleich mit Spinlocks:
    - Spinlocks (Latenz: 34–46 ns) sind für kurze kritische Abschnitte effizienter, da sie Busy Waiting verwenden und keine Systemaufrufe benötigen.
    - Semaphore eignen sich besser für Szenarien, in denen längere Wartezeiten auftreten oder ein blockierender Zugriff erforderlich ist, da sie zwar weniger effizient als Spinlocks für kurze kritische Abschnitte sind – aufgrund der Nutzung blockierender Mechanismen und Systemaufrufe – ihre Stärke jedoch in der Fähigkeit liegt, Threads zu blockieren, wodurch Ressourcen effizient genutzt werden können.

Aufgabe 3: Kommunikation über ZeroMQ

Einleitung

In dieser Aufgabe wurde die minimal erreichbare Latenz für Interprozesskommunikation (IPC) unter Verwendung des ZeroMQ-Frameworks gemessen. Zwei Szenarien wurden analysiert:

- Kommunikation zwischen zwei Threads innerhalb eines Prozesses (In-Process-Kommunikation).
- Kommunikation zwischen zwei Threads in verschiedenen Prozessen.

Ziel war es, die durchschnittliche Latenz zu bestimmen und ein 95%-Konfidenzintervall für die Ergebnisse zu berechnen. Dabei wurden jeweils 7 Durchläufe pro Szenario durchgeführt, um statistisch valide Werte zu erhalten.

Methodik und Implementierung

Szenario 1: In-Process-Kommunikation

Für dieses Szenario wurde ZeroMQ mit dem *inproc://* -Transport verwendet, der eine effiziente Kommunikation zwischen Threads im gleichen Prozess ermöglicht. Die Implementierung bestand aus:

- Server-Thread: Bindet einen PAIR-Socket an die Adresse *inproc://test*, empfängt Nachrichten vom Client und sendet sie zurück (Echo-Mechanismus).
- Client-Thread: Verbindet sich mit *inproc://test*, sendet Nachrichten an den Server und empfängt die Antworten. Hier wird die Zeit für jede Nachrichtensendung und -empfang gemessen.

Die Latenzmessung wurde mit *clock_gettime(CLOCK_MONOTONIC)* durchgeführt. Die durchschnittliche Latenz wurde über alle gesendeten Nachrichten hinweg berechnet.

Szenario 2: Kommunikation zwischen Prozessen

Für dieses Szenario wurde ZeroMQ mit dem *ipc://* -Transport verwendet, der UNIX-Domain-Sockets nutzt, um die Kommunikation zwischen Prozessen zu ermöglichen. Die Implementierung bestand aus:

- Server-Prozess: Bindet einen PAIR-Socket an die Adresse *ipc:///tmp/test*, empfängt Nachrichten vom Client-Prozess und sendet sie zurück.
- Client-Prozess: Verbindet sich mit *ipc:///tmp/test*, sendet Nachrichten und misst die Zeit für den Empfang der Antworten.

Der Hauptprozess wurde mithilfe von *fork()* in zwei Prozesse aufgeteilt: Einen Server-Prozess und einen Client-Prozess. Der Client-Prozess wartet kurz, bis der Server-Prozess bereit ist, um Synchronisationsprobleme zu vermeiden.

Ergebnisse

In beiden Szenarien wurden sieben Durchläufe durchgeführt. Die durchschnittlichen Latenzen sowie die berechneten 95%-Konfidenzintervalle sind wie folgt:

Szenario 1: In-Process-Kommunikation

Berechneter Mittelwert: 24.01 µs

Berechnetes 95%-Konfidenzintervall: [23.49 µs, 24.53 µs]

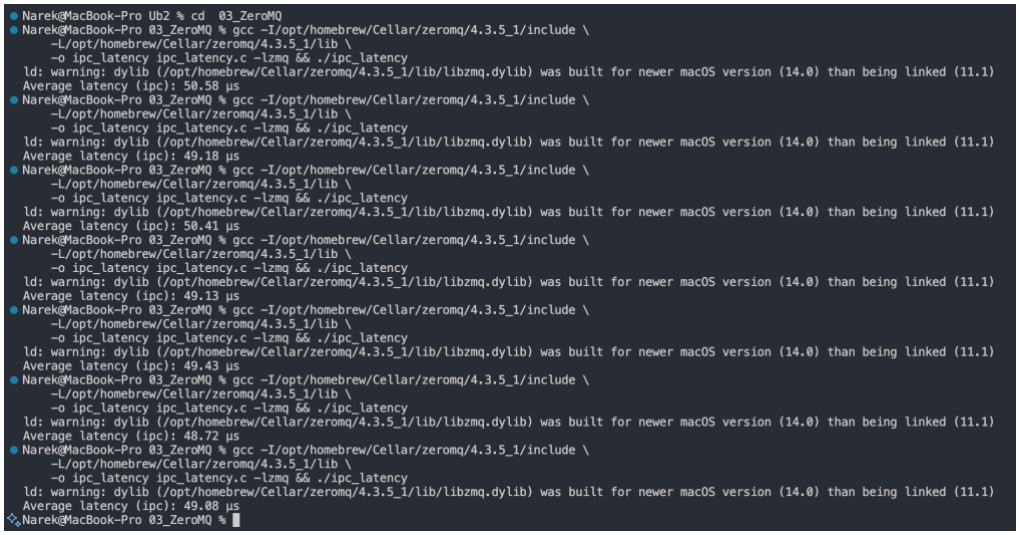
Szenario 2: Kommunikation zwischen Prozessen

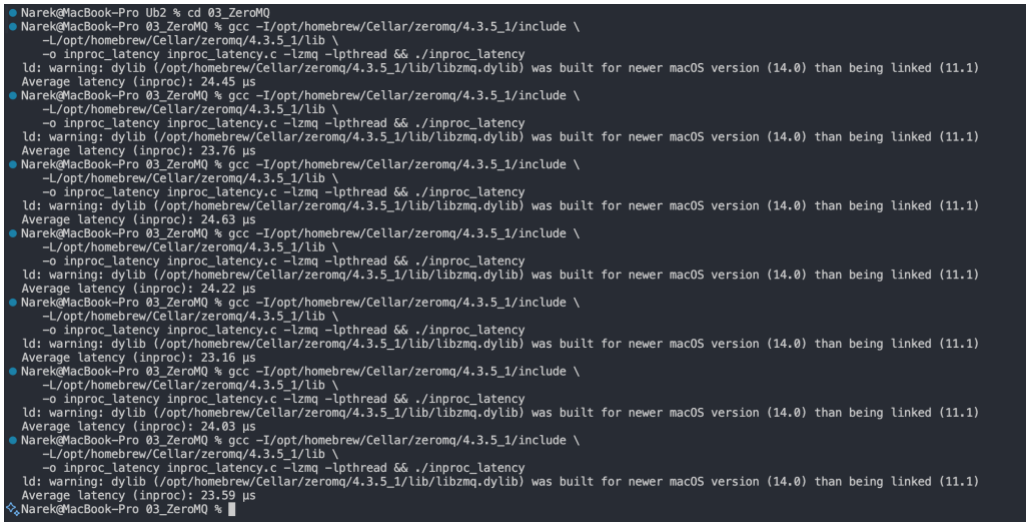
Berechneter Mittelwert: 49.50 µs

Berechnetes 95%-Konfidenzintervall: [48.84 µs, 50.16 µs]

Visuelle Darstellung

Die Ergebnisse der einzelnen Durchläufe können den folgenden Screenshots entnommen werden:





Schlussfolgerungen

- Die durchschnittliche Latenz von ca. 24 µs zeigt, dass ZeroMQ mit inproc:// eine sehr schnelle und effiziente Kommunikation zwischen Threads innerhalb eines Prozesses bietet.
- Die durchschnittliche Latenz bei der Kommunikation zwischen Prozessen ist etwa doppelt so hoch (ca. 50 µs). Dies ist auf den zusätzlichen Overhead durch UNIX-Domain-Sockets und den getrennten Adressraum zurückzuführen.
- Die Konfidenzintervalle in beiden Szenarien sind eng, was auf konsistente und stabile Ergebnisse hinweist.

ZeroMQ eignet sich hervorragend für schnelle In-Process-Kommunikation, zeigt aber den erwarteten Overhead bei Interprozess-Kommunikation. Für Szenarien mit höherem Overhead könnten alternative Mechanismen untersucht werden.

Aufgabe 4: Kommunikation zwischen Docker-Containern

Einleitung

In dieser Aufgabe wurde die Latenz der Kommunikation zwischen zwei Anwendungsthreads untersucht, die in verschiedenen Docker-Containern ausgeführt werden. Ziel war es, die minimale Latenz bei der Interprozesskommunikation (IPC) über das Netzwerk zwischen den Containern zu messen. Die Implementierung wurde mithilfe des bereits bekannten ZeroMQ-Frameworks realisiert. Wie in vorherigen Aufgaben wurde die durchschnittliche Latenz berechnet und durch 7 Durchläufe ein 95%-Konfidenzintervall bestimmt.

Methodik und Implementierung

Logik und Vorgehensweise

Die Kommunikation zwischen den Containern wurde über das ZeroMQ PAIR-Socket-Modell realisiert, das für eine direkte Punkt-zu-Punkt-Verbindung geeignet ist. Die Daten wurden über den *tcp://* -Transport zwischen den Containern übertragen, der Netzwerkkommunikation über das Docker-Bridge-Netzwerk ermöglicht.

- **Server-Container:** Der Server lauscht auf einem bestimmten Port (5555) und wartet darauf, Nachrichten vom Client zu empfangen. Er antwortet mit denselben Nachrichten (Echo-Mechanismus).
- **Client-Container:** Der Client sendet Nachrichten an den Server und misst die Zeit, bis die Antworten empfangen werden. Die durchschnittliche Latenz wird aus der Gesamtdauer der Nachrichtensendungen und -empfänge berechnet.

Implementierung

1. **Server-Implementierung:** Der Server bindet einen PAIR-Socket an eine TCP-Adresse und verarbeitet eine vorgegebene Anzahl von Nachrichten. Dabei empfängt er Nachrichten vom Client und sendet diese unmittelbar zurück.
2. **Client-Implementierung:** Der Client verbindet sich über die TCP-Adresse mit dem Server, sendet Nachrichten und misst die Zeit, bis die Antworten empfangen werden. Die Latenzmessung erfolgt über eine geeignete Zeitmessungsfunktion.
3. **Docker-Konfiguration:** Zwei Container wurden mit **Docker Compose** erstellt:
    - Der Server-Container lauscht auf einem spezifischen Port.

- Der Client-Container verbindet sich mit dem Server-Container über ein Docker-Bridge-Netzwerk. Dabei wurden dedizierte IP-Adressen für die Container vergeben, um die Netzwerkkommunikation zu erleichtern.

Ergebnisse

Die Latenzen wurden in 7 Durchläufen gemessen, und ein 95%-Konfidenzintervall wurde analog zu den vorherigen Aufgaben berechnet.

Durchschnittliche Latenzen aus den Durchläufen (in Mikrosekunden):

- 111.40 µs, 115.78 µs, 111.36 µs, 111.30 µs, 111.95 µs, 113.35 µs, 122.73 µs

Berechneter Mittelwert:

- 113.98 µs

Berechnetes 95%-Konfidenzintervall:

- [110.11 µs, 117.85 µs]

Schlussfolgerungen

1. **Effizienz der Netzwerkkommunikation:** Die durchschnittliche Latenz von ca. 114 µs zeigt, dass die Kommunikation zwischen Containern über das Netzwerk mehr Overhead verursacht als In-Process- oder Interprozess-Kommunikation.
2. **Stabilität:** Das Konfidenzintervall ist relativ eng, was auf konsistente Ergebnisse hinweist.
3. **Netzwerk-Overhead:** Der Anstieg der Latenz im Vergleich zu Interprozess-Kommunikation (ca. 50 µs) zeigt die zusätzlichen Kosten durch Netzwerk-Stack und Container-Virtualisierung.

Die Ergebnisse zeigen, dass ZeroMQ auch für die Kommunikation über Docker-Container effizient ist, jedoch mit einem erwarteten Overhead im Vergleich zu lokaleren IPC-Mechanismen.

Vergleich der Mechanismen und Fazit

Die Experimente analysierten die minimalen Latenzen verschiedener IPC-Mechanismen, darunter Spinlocks, Semaphore und ZeroMQ in verschiedenen Szenarien. Die Ergebnisse zeigten deutliche Unterschiede in der Effizienz und im Overhead der Mechanismen. Die folgende Tabelle fasst die Ergebnisse zusammen:

Mechanismus	Szenario	Durchschnittliche Latenz	Konfidenzintervall
Spinlock	Innerhalb eines Prozesses	37,64 ns	[36,05 ns, 39,23 ns]
Spinlock	Mit Shared Memory	41.4 ns	[39,3 ns, 43,6 ns]
Semaphore	Innerhalb eines Prozesses	3217,3 ns	[3200,9 ns,3233,7 ns]
ZeroMQ	In-Process	22,500 ns	[20,910 ns , 24,090 ns]
ZeroMQ	Interprozess-Kommunikation	49,420 ns	[49,030ns, 49,810ns]
ZeroMQ (Docker)	Zwischen Containern	113,980 ns	[110,110 ns, 117,850 ns]

Schlussfolgerungen

1. **Effizienz der Mechanismen:** Spinlocks bieten die geringste Latenz und sind optimal für kurze kritische Abschnitte. Semaphore und ZeroMQ zeigen höheren Overhead, bieten jedoch Flexibilität und Sicherheit.
2. **Netzwerkkommunikation:** Die Kommunikation zwischen Containern hat erwartungsgemäß die höchste Latenz. Dennoch bleibt sie mit ZeroMQ effizient und eignet sich für verteilte Anwendungen.

3. **Eigene Fazit:**
    - **Spinlocks:** Ideal für lokal begrenzte Szenarien mit höchsten Latenzanforderungen.
    - **Semaphore:** Besser geeignet bei komplexen Synchronisationsanforderungen.
    - **ZeroMQ:** Flexibel und skalierbar, besonders für verteilte Systeme und containerisierte Anwendungen.

Die Experimente zeigen, dass die Wahl des IPC-Mechanismus stark von den Anforderungen an Effizienz, Flexibilität und Kontext abhängt. Spinlocks eignen sich für Minimal-Latenz-Szenarien, während ZeroMQ eine ausgezeichnete Balance zwischen Leistung und Skalierbarkeit bietet.