Portfolio Prüfung

2. Januar 2025

Inhaltsverzeichnis

1 Übung und Aufbau

Das hier bearbeitet Übungsblatt dient dazu, unterschiedliche IPC-Mechanismen zu evaluieren und deren minimale Latenzzeiten experimentell zu bestimmen. Zu den getesteten Mechanismen zählen Spinlocks, Semaphore, Message-Queues und die Kommunikation zwischen Containern.
Die verschiedenen Experimente wurden auf einem Linux (Mint 21.3) Betriebssystem ausgeführt und die zugrunde liegende Hardware bestand aus folgenden Spezifikationen:

Attribut	**Details**
Prozessor	Intel Core i5-8250U
Architektur	Kaby Lake R
Kerne	4
Threads	8
Basis-Taktfrequenz	1.6 GHz
L3-Cache	6 MB
Fertigungstechnologie	14nm
Speicherunterstützung	DDR4-2400, LPDDR3-2133
PCIe-Unterstützung	PCIe 3.0
Sockel	BGA 1356

Tabelle 1: Spezifikationen des Intel Core i5-8250U

1.1 Aufbau der Experimente & Ergebnisse

- **Sprache** für die Ausführung aller Experimente wurde Python verwendet; zum Visualisieren die Matplotlib.
- **Durchläufe:** Die verschiedenen Experiment werden jeweils 50-mal durchgeführt. In jedem Durchlauf werden Latenzen gemessen und in einem Array gespeichert.
- **Statistik:** Die Ergebnisse werden analysiert und folgende Metriken berechnet:
  - Durchschnittliche Latenz
  - Standardabweichung
  - 95%-Konfidenzintervall

2 Spinlocks und Busy Waiting (Aufgabe01_Shared_Memory.py)

Spinlocks basieren auf aktivem Warten: Ein Thread überprüft kontinuierlich (in einer Schleife), ob ein bestimmter Zustand erreicht ist, um fortfahren zu können. Dies ermöglicht eine schnelle Synchronisation, da kein Kontextwechsel erfolgt, kann aber ineffizient sein, da Ressourcen während des Wartens blockiert bleiben.
Shared Memory ist ein Mechanismus, bei dem mehrere Threads oder Prozesse auf denselben Speicherbereich zugreifen können. Dies erlaubt eine direkte und schnelle Datenübertragung, ohne dass Zwischenspeicher wie Pipes oder Message Queues verwendet werden müssen. In Kombination ermöglichen diese Mechanismen eine einfache, aber performante Kommunikation, deren Latenz hier experimentell ermittelt wird.

2.1 Code-Architektur

- **Prozesse und Synchronisation:** Zwei Prozesse, `thread_a` und `thread_b`, kommunizieren über gemeinsame Speicherbereiche und Ereignisse.
- **Synchronisationsmechanismus:** Ein Spinlock regelt den Zugriff auf den geteilten Speicherbereich `shared_mem`.
- **Durchführung:** In jedem Durchlauf wird die Zeit gemessen, die zwischen dem Schreiben in `shared_mem` durch `thread_a` und der Verarbeitung durch `thread_b` vergeht.

2.1.1 Verwendete IPC-Mechanismen

Shared Memory: Der Zugriff erfolgt über folgende Datenstrukturen:

- `multiprocessing.Value`: Ein thread-sicherer Wrapper für eine skalare Variable, die von mehreren Prozessen genutzt wird.
- `shared_mem` speichert den zu übertragenden Wert.

Events: Zur Synchronisation werden `multiprocessing.Event`-Objekte verwendet:

- `start_event`: Signalisiert den Start des Experiments.
- `end_event`: Signalisiert das Ende der Verarbeitung.

Spinlock: Der Spinlock wird folgendermaßen implementiert:

- `lock.value` wird von `thread_a` auf `1` gesetzt, um anzuzeigen, dass ein neuer Wert verfügbar ist.
- `thread_b` überprüft kontinuierlich den Status des Locks und verarbeitet den Wert, sobald dieser verfügbar ist.

3 Semaphores (Aufgabe02_Semaphores.py)

Semaphoren werden verwendet, um den Zugriff auf gemeinsam genutzte Ressourcen zu steuern. Sie arbeiten mit Zählern, die signalisieren, wie viele Ressourcen verfügbar sind: Ein Prozess kann die Semaphore "dekrementieren"(P-Operation) und eine Ressource beanspruchen, oder ïnkrementieren"(V-Operation), um sie freizugeben.

3.1 Code-Architektur

Die grundlegende Code-Architektur der zweiten Aufgabe ist größtenteils eine Weiterentwicklung der ersten. Es werden die gleichen Mechanismen verwendet, allerdings kommen anstelle von Spinlocks Semaphoren zum Einsatz.

- **Prozesse und Synchronisation:** Zwei Prozesse, `thread_a` und `thread_b`, kommunizieren über gemeinsame Speicherbereiche und Semaphore.
- **Synchronisationsmechanismus:** Eine Semaphore wird verwendet, um den Zugriff auf den gemeinsam genutzten Speicherbereich `shared_mem` zu synchronisieren.
- **Durchführung:** In jedem Durchlauf wird die Zeit gemessen, die zwischen dem Schreiben in `shared_mem` durch `thread_a` und der Verarbeitung durch `thread_b` vergeht.

3.1.1 Verwendete IPC-Mechanismen

Shared-Memory und Events äquivalent zur ersten Aufgabe.

Semaphore: Zur Synchronisation wird eine Semaphore genutzt:

- `semaphore.acquire()`: Blockiert den ausführenden Prozess, bis die Semaphore freigegeben wird.
- `semaphore.release()`: Gibt die Semaphore frei, sodass ein wartender Prozess fortfahren kann.

4 Message-Queues / ZeroMQ (Aufgabe03_ZeroMQ.py)

ZeroMQ (ZMQ) ist eine skalierbare Bibliothek für asynchrone Nachrichtenübermittlung, die verschiedene Kommunikationsmuster wie Publish-Subscribe (PUB-SUB), Request-Reply oder Push-Pull unterstützt. In diesem Experiment wird die Latenz von Nachrichtenübermittlungen unter zwei Bedingungen untersucht:

1. Innerhalb eines Prozesses mittels Threads.
2. Zwischen separaten Prozessen über TCP.

4.1 Code-Architektur

- **Publisher-Subscriber-Muster:** Ein Publisher sendet Nachrichten an alle verbundenen Subscriber. Dieses Muster ist ideal für Broadcast-ähnliche Szenarien.
- **Threads und Prozesse:**
  - Im ersten Experiment nutzen Publisher und Subscriber Threads innerhalb eines Prozesses. Hierbei teilen sich die Threads denselben Speicherbereich und den ZeroMQ-Kontext, was zu geringeren Latenzen führt.
  - Im zweiten Experiment kommunizieren Publisher und Subscriber als separate Prozesse. Die Kommunikation erfolgt über das TCP-Protokoll, was zu einem höheren Overhead führt, da jeder Prozess seinen eigenen ZeroMQ-Kontext besitzt.
- **Messung der Latenz:** Für jede gesendete Nachricht wird die Zeit zwischen dem Absenden durch den Publisher und dem Empfang durch den Subscriber gemessen. Diese Latenz wird in beiden Szenarien (Threads und Prozesse) untersucht und verglichen.

4.1.1 Innerhalb eines Prozesses

Bei der Kommunikation innerhalb eines Prozesses teilen sich die Threads denselben Speicherraum und denselben ZeroMQ-Kontext. Dies reduziert den Overhead und sollte zu geringeren Latenzen führen. Die Publisher- und Subscriber-Sockets werden mit der Methode `bind()` bzw. `connect()` auf denselben Port gebunden und kommunizieren direkt über den ZeroMQ-Kontext.

4.1.2 Zwischen separaten Prozessen

Bei der Kommunikation zwischen Prozessen erfolgt der Nachrichtenaustausch über das TCP-Protokoll. Dies führt zu einem höheren Overhead, da jeder Prozess einen eigenen ZeroMQ-Kontext besitzt und die Kommunikation über das Betriebssystem abgewickelt wird. In diesem Fall bindet der Publisher-Socket den Port und der Subscriber-Socket verbindet sich über `connect()`. Hierbei wird der Nachrichtenversand über das Netzwerk abgewickelt, was zu einer höheren Latenz führt.

4.2 Verwendete ZeroMQ Sockets

- **PUB-Socket (Publisher):** Der Publisher sendet Nachrichten an alle Subscriber, die sich mit dem gleichen Port verbunden haben. Dies erfolgt durch `socket.send_string()` und `socket.bind()`. - **SUB-Socket (Subscriber):** Der Subscriber empfängt Nachrichten vom Publisher, nachdem er mit dem entsprechenden Port verbunden wurde, und verwendet `socket.connect()` und `socket.setsockopt_string(zmq.SUBSCRIBE, )` für die Verbindung und das Abonnieren von Nachrichten.

5 Docker-Container (Ausgabe04_Docker)

Um die Kommunikation zwischen den Containern zu ermöglichen, wird Docker als Containerisierungstechnologie verwendet. Als Struktur für das Experiment wird ein Server-Container und ein Client-Container implementiert. Diese Container kommunizieren über ein Docker-Netzwerk miteinander. Im Folgenden wird der gesamte Aufbau beschrieben, einschließlich der notwendigen Befehle, wie man die Container baut, startet und stoppt.

5.1 Projektaufbau und Container-Setup

Für dieses Projekt sind zwei Python-Skripte erforderlich:

- `server.py` - Der Server empfängt Nachrichten vom Client und berechnet die Latenz.
- `client.py` - Der Client sendet Nachrichten an den Server. Die Nachrichten beinhalten den Absendezeitpunkt der Nachricht.
- `Intervalle` - Der Client sendet insgesamt 50 Nachrichten an den Server.

Beide Skripte laufen in separaten Docker-Containern.

5.2 Docker-Netzwerk erstellen

Damit die beiden Container miteinander kommunizieren können, müssen sie sich im selben Docker-Netzwerk befinden. Dieses Netzwerk ermöglicht die Kommunikation zwischen den Containern über deren Container-Namen.

- Erstelle ein Docker-Netzwerk:

```
docker network create my_network
```

5.3 Docker-Container bauen

Um die Docker-Container zu erstellen, müssen die jeweiligen `Dockerfiles` in den entsprechenden Verzeichnissen vorhanden sein. Die Container werden dann mit den folgenden Befehlen gebaut. Die requirements.txt beinhaltet hierbei lediglich die matplotlib. Diese wird zwar im client nicht verwendet, aber der Vollständigkeit halber gibt es auch in diesem Unterverzeichnis eine Requirements-Datei.

- Baue das Server-Image:

```
docker build -t server-container ./server
```

- Baue das Client-Image:

```
docker build -t client-container ./client
```

5.4 Docker-Container starten

Nachdem die Docker-Images gebaut wurden, müssen die Container gestartet werden.

- Starte den Server-Container:

```
docker run -it --name server-instance --network my_network server-container
```

- Starte den Client-Container:

```
docker run -it --name client-instance --network my_network client-container
```

5.5 Server und Client Kommunikation

Im Client-Skript wird der Hostname des Server-Containers als Zieladresse für die Verbindung angegeben. Der Server lauscht auf allen IP-Adressen (0.0.0.0) und wartet auf eingehende Verbindungen. Der Client verbindet sich mit dem Server über den Namen des Containers, z.B. `server-instance`, da beide Container im selben Netzwerk laufen.

- Der Client verbindet sich mit dem Server:

```
client_socket.connect(('server-instance', 12345))
```

- Der Server akzeptiert Verbindungen:

```
server_socket.bind(('0.0.0.0', 12345))
```

6 Ergebnisse

6.1 Kommunikation über Spinlocks (Busy Waiting)

Die Ergebnisse der Latenzmessung sind wie folgt:

- **Durchschnittliche Latenz**: 0.001085 Sekunden
- **Standardabweichung**: 0.000130 Sekunden
- **95%-Konfidenzintervall**: ±0.000036 Sekunden

Die **durchschnittliche Latenz** von 0.001085 Sekunden zeigt, dass die Kommunikation mit Spinlocks sehr schnell ist. Dies liegt an der geringen Komplexität des Mechanismus, bei dem die Threads direkt über den Lock synchronisieren. Die **Standardabweichung** von 0.000130 Sekunden ist gering, was auf eine stabile und konstante Performance des Systems während der Messungen hinweist. Das **95%-Konfidenzintervall** von ±0.000036 Sekunden bestätigt die Präzision und Stabilität der Messwerte.

6.2 Aufgabe 2: Kommunikation über Semaphore

Die Messwerte für die Latenz bei Verwendung von Semaphore lauten:

- **Durchschnittliche Latenz**: 0.001075 Sekunden
- **Standardabweichung**: 0.000100 Sekunden
- **95%-Konfidenzintervall**: ±0.000028 Sekunden

Im Vergleich zum Spinlock-Mechanismus ist die **durchschnittliche Latenz** von 0.001075 Sekunden nahezu identisch mit der des Spinlocks. Dies deutet darauf hin, dass Semaphore für diese Art der Kommunikation ebenfalls eine sehr geringe Latenz bieten. Die **Standardabweichung** von 0.000100 Sekunden ist ebenfalls gering, was auf eine stabile Performance hinweist. Das **95%-Konfidenzintervall** von ±0.000028 Sekunden bestätigt, dass auch die Semaphore-basierten Messungen sehr präzise sind.

6.3 Aufgabe 3: Kommunikation über ZeroMQ

In der dritten Aufgabe wurde die ZeroMQ Message-Queue zur Kommunikation zwischen zwei Threads untersucht, sowohl in einem einzigen Prozess als auch zwischen mehreren Prozessen.
Die Ergebnisse für die Kommunikation über ZeroMQ innerhalb eines Prozesses lauten:

- **Durchschnittliche Latenz (ZeroMQ, innerhalb eines Prozesses)**: 0.100326 Sekunden
- **Standardabweichung**: 0.000267 Sekunden
- **95%-Konfidenzintervall**: ±0.000074 Sekunden

Für die Kommunikation zwischen Prozessen mit ZeroMQ wurden die folgenden Werte ermittelt:

- **Durchschnittliche Latenz (ZeroMQ, zwischen Prozessen)**: 0.099400 Sekunden
- **Standardabweichung**: 0.000307 Sekunden
- **95%-Konfidenzintervall**: ±0.000085 Sekunden

Die **durchschnittliche Latenz** für ZeroMQ liegt bei etwa 0.100 Sekunden, was deutlich höher ist als bei den anderen getesteten IPC-Mechanismen. Diese höhere Latenz ist aufgrund der zusätzlichen Komplexität von ZeroMQ und seiner flexiblen Architektur zu erwarten. Die Kommunikation zwischen Prozessen (0.099400 Sekunden) ist dabei nur minimal schneller als die Kommunikation innerhalb eines Prozesses (0.100326 Sekunden), was darauf hinweist, dass die Latenzsteigerung durch die Verwendung von ZeroMQ im Vergleich zu einfacheren Mechanismen wie Spinlocks und Semaphore relativ gering ausfällt.
Die **Standardabweichung** für ZeroMQ liegt im Bereich von 0.000267 bis 0.000307 Sekunden, was immer noch akzeptabel ist, jedoch etwas größer als die der anderen Mechanismen. Das **95%-Konfidenzintervall** von ±0.000074 Sekunden für die Kommunikation innerhalb eines Prozesses und ±0.000085 Sekunden für die Kommunikation zwischen Prozessen zeigt ebenfalls eine stabile und präzise Messung.

6.4 Aufgabe 4: Kommunikation zwischen Docker-Containern

In der vierten Aufgabe wurde die Kommunikation zwischen zwei Threads in verschiedenen Docker-Containern untersucht. Die Messung der Latenz dieser Kommunikation erfolgte in einer isolierten Umgebung, in der zwei Docker-Container miteinander über IPC kommunizierten.
Die Ergebnisse der Latenzmessungen für die Kommunikation zwischen den Docker-Containern lauten:

- **Durchschnittliche Latenz**: 0.000324 Sekunden
- **Standardabweichung**: 0.000072 Sekunden
- **95%-Konfidenzintervall**: ±0.000020 Sekunden

Die **durchschnittliche Latenz** von 0.000324 Sekunden zeigt, dass die Kommunikation zwischen den Docker-Containern ebenfalls sehr schnell ist, wenn auch etwas langsamer als die Kommunikation innerhalb eines Prozesses oder zwischen Threads im selben Prozess. Dies liegt an der zusätzlichen Komplexität und den Overhead-Kosten durch die Containerisierung. Die **Standardabweichung** von 0.000072 Sekunden ist gering und deutet auf eine stabile und zuverlässige Performance hin. Das **95%-Konfidenzintervall** von ±0.000020 Sekunden bestätigt die Präzision der Messergebnisse.

6.5 Mögliche Ursachen für unerwartet schnelle Docker-Kommunikation

Obwohl die Kommunikation zwischen Docker-Containern in den Tests als schneller gemessen wurde als zwischen Threads im selben Prozess, ist dies in der Regel nicht zu erwarten. Normalerweise sollte die Kommunikation innerhalb eines Prozesses aufgrund des geringeren Overheads schneller sein. Da wir Messfehler und generelle Inkonsistenzen ausschließen können, bleiben folgende mögliche Ursachen für diese unerwarteten Ergebnisse:

- **Unterschiede im System-Overhead**: Die Kommunikation zwischen Threads innerhalb eines Prozesses könnte durch zusätzliche Systemoverhead-Kosten beeinflusst worden sein, etwa durch Suboptimalitäten im Thread Scheduling. Auf der anderen Seite könnte Docker für diese Art der Kommunikation optimiert sein und möglicherweise über einen schnelleren Netzwerkstack verfügen.
- **Unterschiedliche Netzwerkmodi in Docker**: Wenn Docker in einem speziellen Netzwerkmodus betrieben wurde, wie z. B. im ‘host‘-Netzwerkmodus, der den Containern direkten Zugriff auf das Host-Netzwerk ermöglicht, könnte dies die Latenz verringern und zu den unerwartet niedrigen Werten geführt haben.

6.6 Zusammenfassung und Vergleich der Ergebnisse

Die Ergebnisse zeigen deutliche Unterschiede in der Latenz der verschiedenen IPC-Mechanismen:

- **Spinlocks** und **Semaphore** bieten mit einer durchschnittlichen Latenz von etwa 0.001 Sekunden eine sehr schnelle Kommunikation, wobei die Semaphore mit 0.001075 Sekunden nur marginal langsamer als der Spinlock mit 0.001085 Sekunden sind.
- **ZeroMQ** bietet zwar eine höhere Latenz von rund 0.1 Sekunden, liefert jedoch eine skalierbare Lösung für komplexere Kommunikationsszenarien, bei denen Nachrichten zwischen Prozessen oder sogar über Netzwerke ausgetauscht werden müssen.
- **Docker-Container** bieten mit einer durchschnittlichen Latenz von 0.000324 Sekunden eine sehr schnelle, aber durch die Containerisierung leicht verzögerte Kommunikation.

Insgesamt zeigt sich, dass die einfacheren IPC-Mechanismen wie Spinlocks und Semaphore für schnelle, direkte Kommunikation innerhalb eines Prozesses oder zwischen wenigen Threads gut geeignet sind, während ZeroMQ besser für komplexere, skalierbare Anwendungen geeignet ist, bei denen höhere Latenzen in Kauf genommen werden. Docker-Container bieten eine weitere Möglichkeit für die Kommunikation zwischen Prozessen in einer isolierten Umgebung, die allerdings einen minimalen Overhead verursacht.
Die präzisen Ergebnisse und das geringe Konfidenzintervall für alle getesteten Mechanismen bestätigen die Zuverlässigkeit der Messungen und ermöglichen eine fundierte

Entscheidung bei der Auswahl des geeigneten IPC-Mechanismus für konkrete Anwendungsfälle.

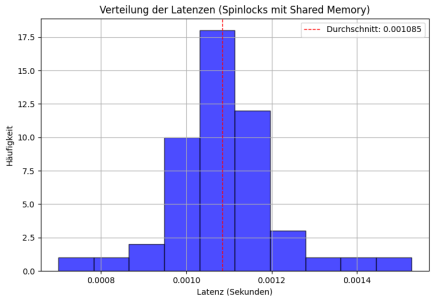

Abbildung 1: Spinlocks

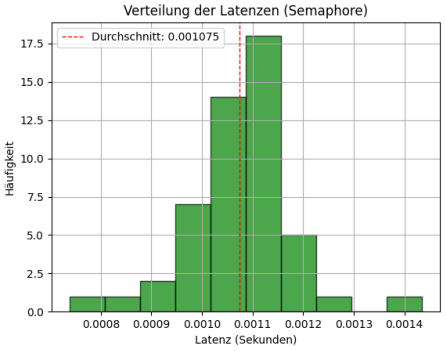

Abbildung 2: Semaphores

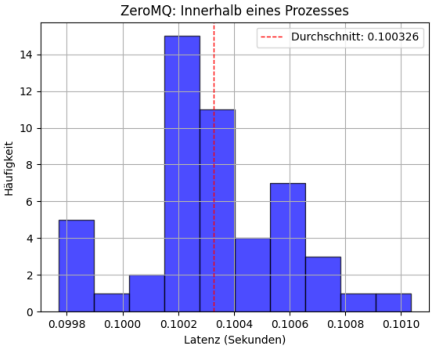

Abbildung 3: ZeroMQ Single Process

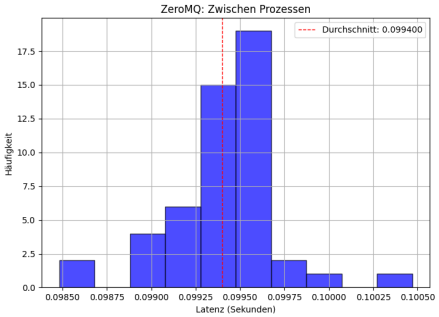

Abbildung 4: ZeroMQ Multi Process

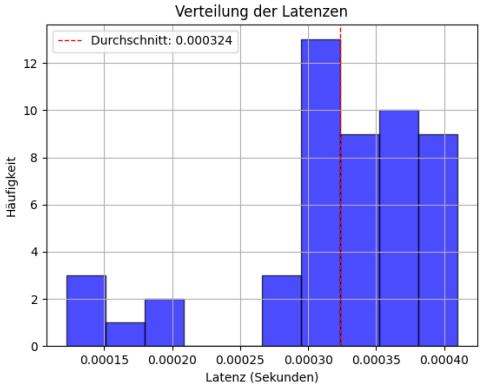

Abbildung 5: Docker