Betriebssysteme

Portfolioaufgabe 2

https://github.com/mmaywald05/OS2

Inhalt

Aufgabe und Experimentaufbau

Ziel der Aufgabe ist es, die minimalen Latenzen verschiedener IPC Mechanismen experimentell zu ermitteln (Konfidenz: 95%). Dieser Abschnitt beschreibt, welche Mechanismen ich geprüft habe und wie die Experimente aufgesetzt wurden. Folgende Mechanismen wurden getestet:

- Spinlocks in zwei Threads innerhalb eines Prozesses mit dediziertem Shared-Memory Bereich
- Semaphore in zwei Threads innerhalb eines Prozesses
- ZMQ `inproc` Protokoll für Intraprozesskommunikation
- ZMQ `ipc` Protokoll für Interprozesskommunikation
- Kommunikation zweier Docker container.

In den Experimenten werden $N = 1\,000\,000$ Iterationen des entsprechenden Mechanismus ausgeführt und mit Nanosekunden-Präzision abgespeichert. In jeder Iteration vollführen zwei Threads P1 und P2 abwechselnde minimale Kommunikation (z.B. P1 setzt einen Bit auf 1 und signalisiert P2, P2 liest den Bit und setzt ihn wieder auf 0, usw.).

Die Zeiterfassung bezieht sich auf eine komplette „Kommunikationsrunde“, startet also mit dem Beginn eines Austauschs (setzen des Bit bzw. senden einer Nachricht) und endet, wenn der nächste Austausch beginnt. In dieser Kommunikationsrunde findet der entsprechende Mechanismus zweimal statt, zusätzlich zu der Verarbeitung des minimalen Workloads. Außerhalb der Zeiterfassung werden die Ergebnisse abgespeichert. So resultiert für jedes Experiment eine Liste mit 1 mio. individuellen Zeiten für Kommunikationsrunden dieses

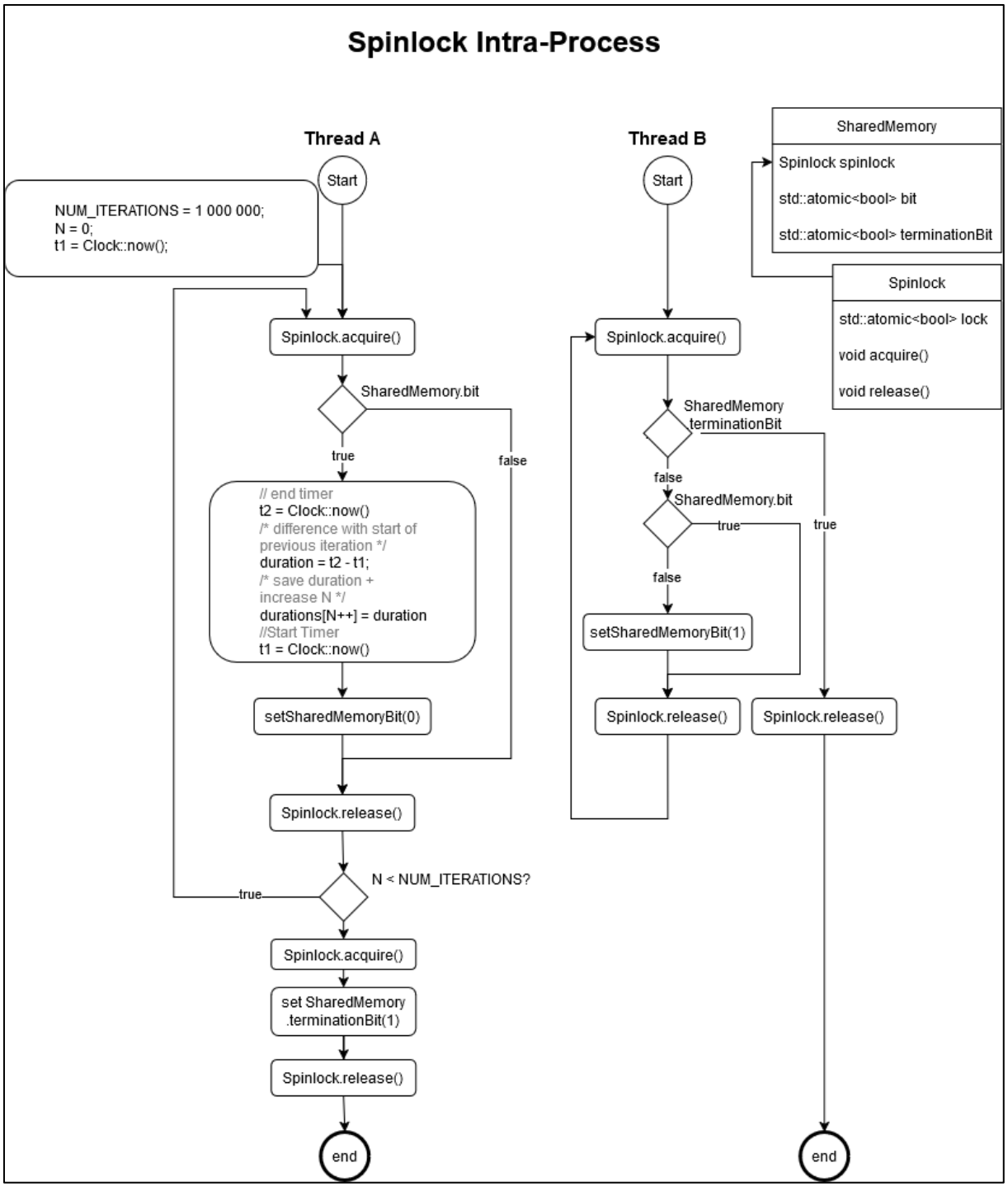
Mechanismus. Diese Werte werden dann graphisch dargestellt und das Konfidenzintervall für die minimale Latenz ermittelt.

Alle Experimente wurden in C++ programmiert und auf einem MacBook Air mit M2 Chip durchgeführt, mit minimalen Hintergrundprozessen. Die Implementierungen sind dabei teilweise speziell für MacOS (Unix) und Apple Silicon Prozessoren ausgelegt.

Programme

Die Programme folgen alle folgendem Grundlegenden Ablauf:

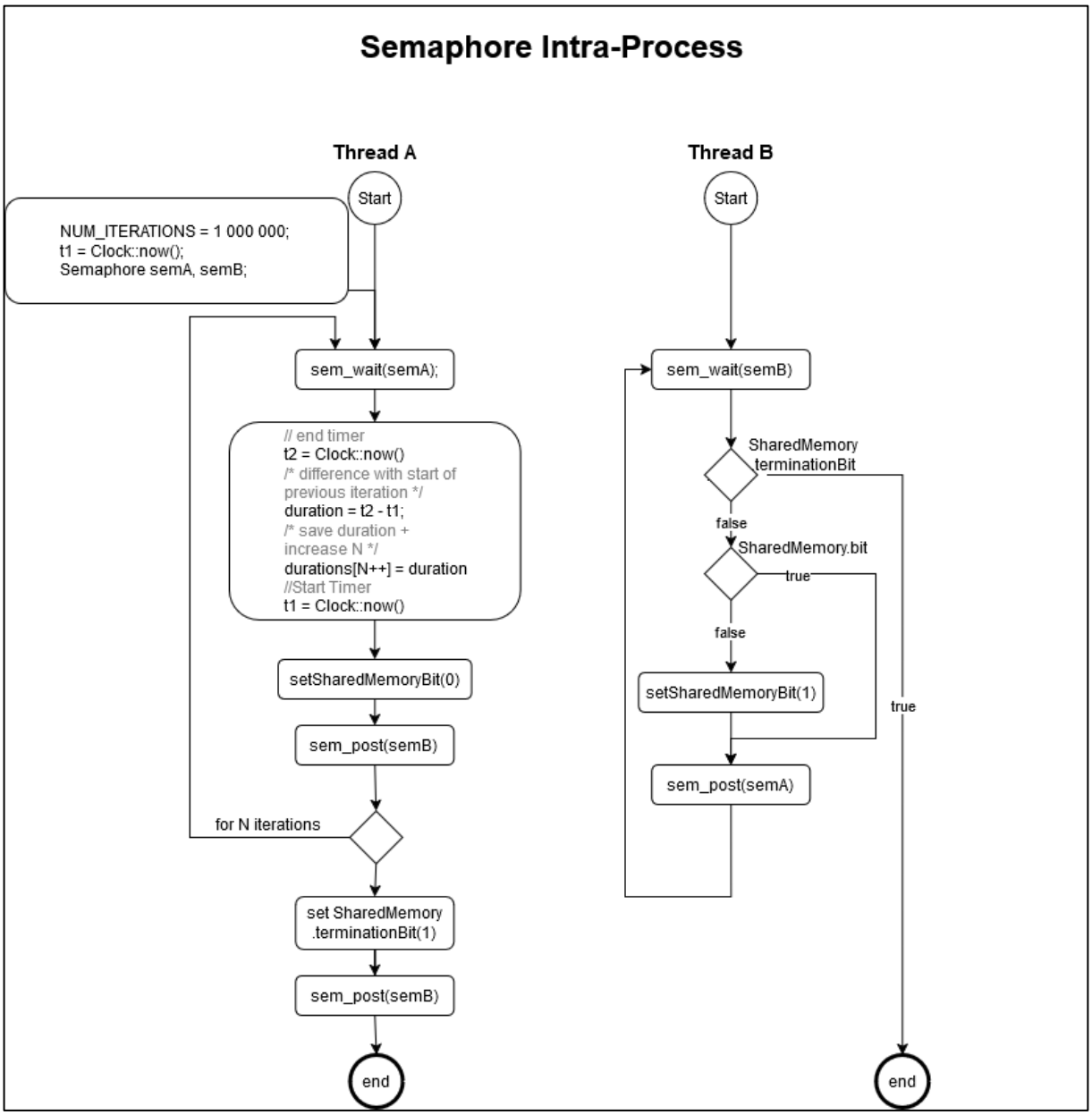
- Zwei Threads (P1, P2) oder Prozesse werden initialisiert. Die Startzeit t1 wird auf den aktuellen Moment gesetzt.
- P1 iteriert N mal. Zeiterfassung findet ausschließlich in P1 statt.
- P2 wartet dauerhaft auf eingehende Kommunikation.
- In jeder Iteration:
    - Endzeitpunkt t2 wird
    - Differenz zu Startzeitpunkt t1 der vorherigen Kommunikationsrunde wird berechnet
    - Ergebnis wird als Nanosekunden abgespeichert
    - Startzeitpunkt t1 wird aktualisiert
    - Neue Kommunikationsrunde beginnt. Signal an P2 und auf Antwort warten, dann wiederholen. Nach Ablauf aller Iterationen signalisiert P1 an P2 ein Kill Signal und beide Prozesse enden.
- Jedes Programm wird alle gemessenen Zeiten selbständig in eine Datei zur späteren Verwendung abspeichern. Bei Docker muss diese Datei noch manuell aus dem Container herauskopiert werden (siehe Repository ReadMe)



Spinlocks

Das Spinlock Programm folgt dem generellen Ablauf. Es werden zwei Threads P1 und P2 gestartet. Thread P1 setzt in jeder Iteration ein Bit im gemeinsamen Speicherbereich. P2 reagiert darauf, indem das Bit zurückgesetzt wird. Bevor eine neue Kommunikationsrunde beginnt, muss P1 das zurückgesetzte Bit registrieren. Erhält P1 das Lock bevor P2 den Bit zurückgesetzt hat, wird das Lock sofort wieder freigegeben und die Iteration übersprungen. Die Threads terminieren durch ein bestimmtes Kill-Signal, das von P1 nach allen Iterationen im gemeinsamen Speicher gesetzt wird. Der gemeinsame Speicher wird dabei explizit auf

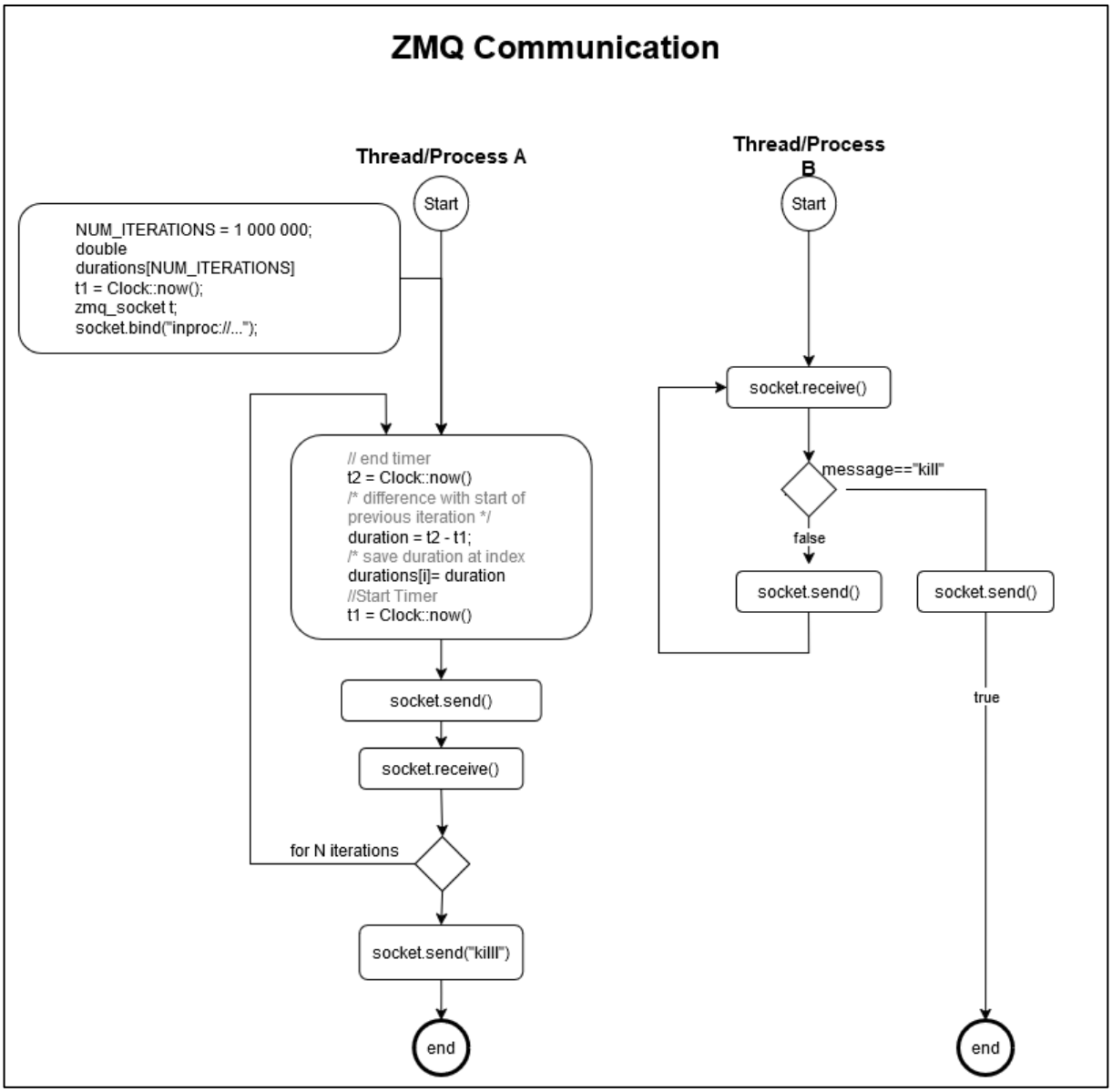
Hardware-Ebene allokiert (mit `shm_open` und `mmap`). Zu den Spinlocks wurde eine primitive und eine zusätzliche, effizientere Implementierung umgesetzt[1].



Sempahore

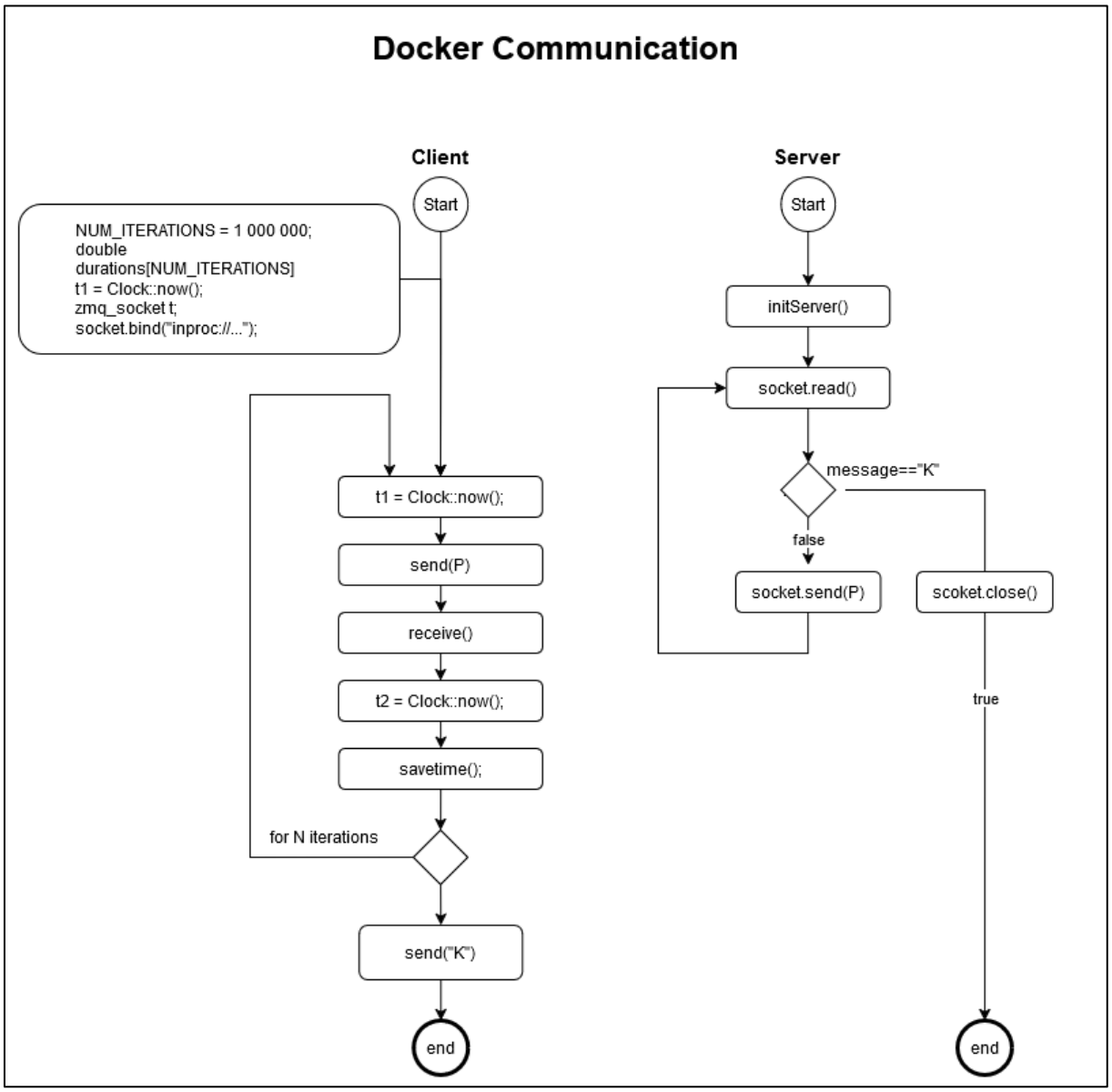
Das Programm zur Ermittlung der Latenzen von Semaphoren folgt wieder dem generellen Ablauf und ist dem zu Spinlocks sehr ähnlich. Wieder werden zwei Threads P1 und P2 erzeugt. In einer Iteration schreibt P1 ein Bit, signalisiert P2 und wartet. P2 liest das Bit, überschreibt es, signalisiert P1 und wartet. Wieder wird dedizierter gemeinsamer Speicher verwendet und die Terminierung erfolgt wieder über ein Kill-Signal.

[1] https://rigtorp.se/spinlock/



ZMQ

Sowohl für Kommunikation mittels ZMQ innerhalb eines Prozesses als auch zwischen zwei Prozessen gilt obiges Ablaufdiagramm. Die Zeitmessung erfolgt wieder Iterationen-Übergreifend, im Grunde wird jedoch die Dauer einer `socket.send()`- und `socket.receive()`-Operation gemessen. Der unterschied zwischen Intra- und Interprozesskommunikation mit ZMQ liegt in der Definition des Transportprotokolls des Sockets (`inproc` oder `ipc`) und dem Ausführen der beiden Routinen als Threads bzw. als eigenständige Prozesse (über `fork()`). Da ZMQ Nachrichtenaustausch implementiert wird der workload-Overhead minimiert, indem Nachrichten aus einzelnen `char` bestehen. Für die Kompilierung muss ZMQ in der richtigen Lokation vorhanden sein („/opt/homebrew/“). Auf meinem Rechner, und damit für die Ausführen des Shell-Skriptes zum Starten der ZMQ-Programme notwendig, wurde ZMQ mit Homebrew in den Standard Homebrew Zielordnern installiert.



Docker

Für die Experimente mit Docker werden ein Server-Prozess und ein Client Prozess in eigenen Containern gestartet. In einer Iteration sendet der Client eine Nachricht an den Server und wartet die Antwort ab. Der Server wartet einfach auf eingehende Nachrichten und beantwortet alles Eingehende direkt. Das erfasste Zeitfenster umfasst hier die Dauer für beide Aufrufe, also eine ganze Kommunikationsrunde.

Ausführung

Eine detaillierte Beschreibung der Ausführung ist in der ReadMe im Abgaberepository. Es müssen einige Pfade umgestellt werden und einiges verifiziert werden, speziell für die ZMQ/Docker Programme. Mit den beiliegenden Shellscripts sollte sich alles ausführen lassen.

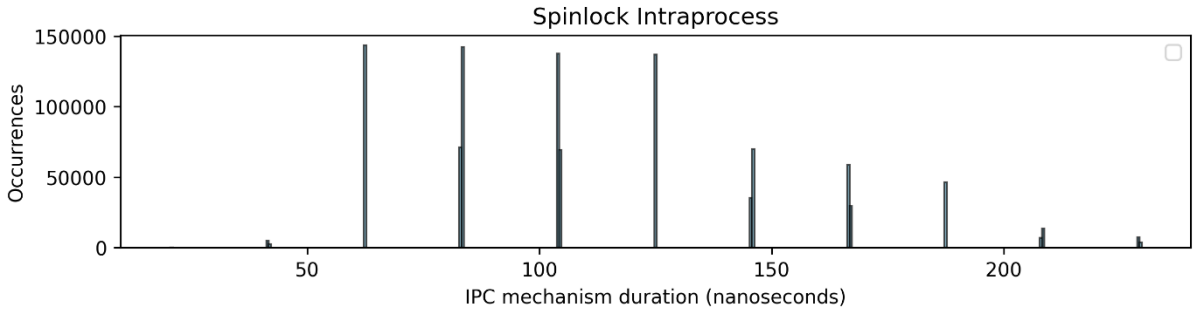
Ergebnisse

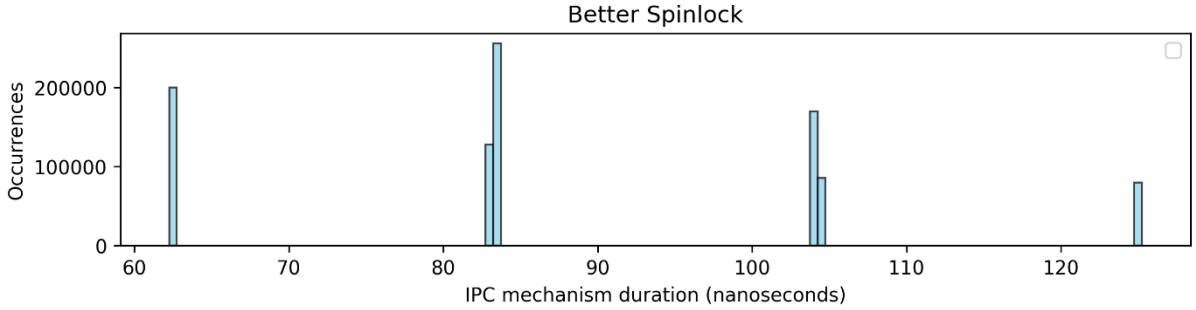
In jedem Experiment finden 10 000 Aufwärm-Iterationen statt, bevor Zeiten gespeichert werden. Die Ergebnisse wurden feingranular Abgespeichert und mit einem Python Skript statistisch aufbereitet. Im Skript wird jede Datei eingeladen und zunächst alle Werte halbiert (Approximation der One-Way Zeiten). Dann werden mit der 1.5-IQR-Methode Ausreißer entfernt, da diese (abhängig von der Methode) sehr stark ausfallen können. Die Daten werden als Histogramm dargestellt und das 95%-Konfidenzintervall berechnet[2].

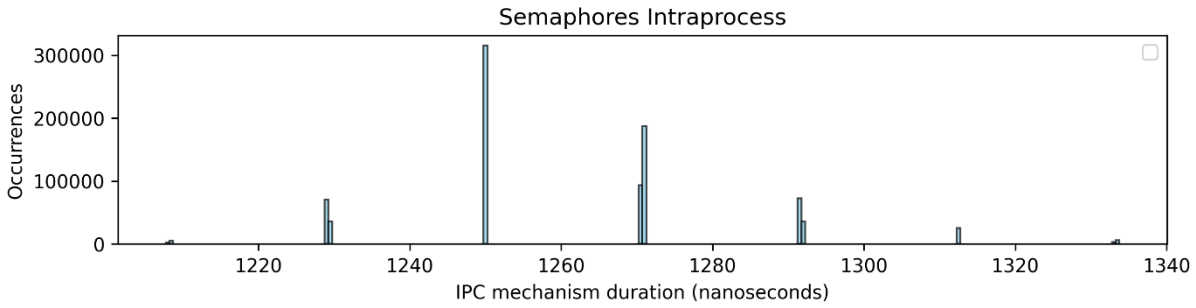
	Minimal Latenz (Konfidenzintervall) (Nanosekunden)		
IPC Mechanismus	Untere Grenze	Mittelwert	Obere Grenze
Spinlock	113.45	113.53	113.61
Besseres Spinlock	88.16	88.2	88.24
Semaphore	1261.93	1261.98	1262.03
ZMQ Intraproc	8866..19	8868.49	8870.80
ZMQ Interproc	28163.27	28168.28	28173.29
Docker Container	17779.21	17781.39	17783.56

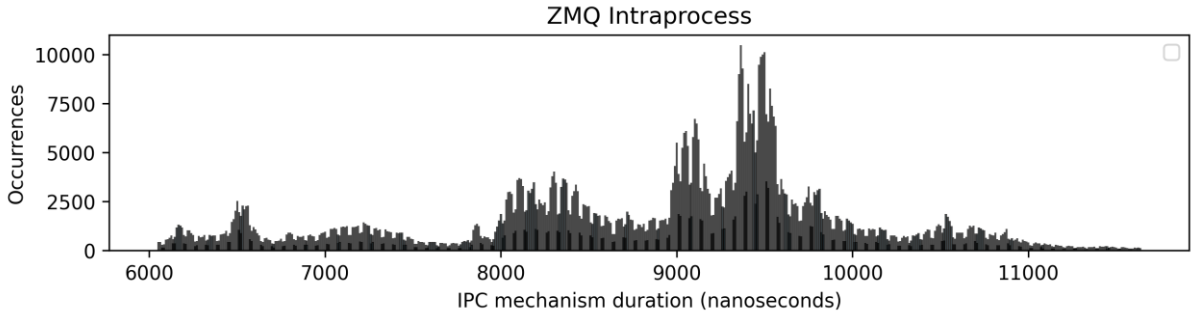
In den Verteilungen der Zeiten wird sichtbar, dass die Zeiten von Spinlocks und Semaphoren viel weniger Varianz zeigen. Es kommen weniger Werte viel häufiger vor. Bei den Mechanismen, die auf externe Ressourcen zurückgreifen (Docker und ZMQ), lässt sich viel besser eine statistische Verteilung beobachten, auch weil die Zeiten generell in einer größeren Spannweite auftreten. Die Ergebnisse sind alle von Ausreißern durchzogen, die zu zufälligen Zeiten auftreten und sich von den anderen Werten teils um Größenordnungen unterscheiden. Besonders interessant finde ich das verbesserte Ergebnis der optimierten Spinlock-implementierung. Die Zeiten der Semaphore sind außerdem viel höher als erwartet, da es ein so grundlegender Mechanismus der Threadkommunikation ist habe ich hier mit besserer Performance gerechnet.

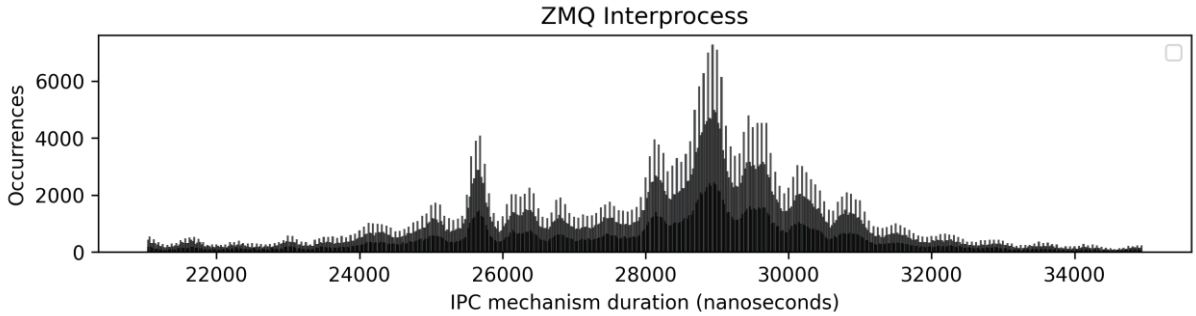
[2] https://stackoverflow.com/questions/15033511/compute-a-confidence-interval-from-sample-data

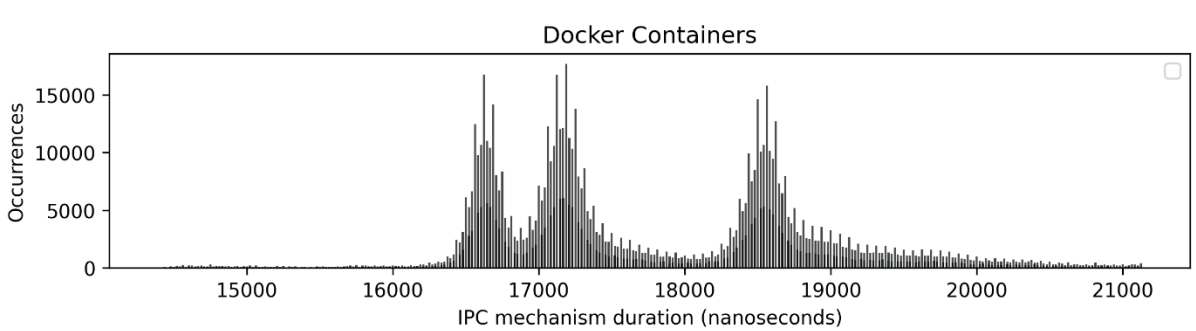

Spinlock Intraprocess
Occurrences
150000
100000
50000
0
50
100
150
200
IPC mechanism duration (nanoseconds)


Better Spinlock
Occurrences
200000
100000
0
60
70
80
90
100
110
120
IPC mechanism duration (nanoseconds)


Semaphores Intraprocess
Occurrences
300000
200000
100000
0
1220
1240
1260
1280
1300
1320
1340
IPC mechanism duration (nanoseconds)


ZMQ Intraprocess
Occurrences
10000
7500
5000
2500
0
6000
7000
8000
9000
10000
11000
IPC mechanism duration (nanoseconds)


ZMQ Interprocess
Occurrences
6000
4000
2000
0
22000
24000
26000
28000
30000
32000
34000
IPC mechanism duration (nanoseconds)


Docker Containers
Occurrences
15000
10000
5000
0
15000
16000
17000
18000
19000
20000
21000
IPC mechanism duration (nanoseconds)