Betriebssysteme
Übung 2

Latenzen verschiedener IPC-Mechanismen

Als Betriebssystem wurde für diese Aufgaben ein Ubuntu WSL auf Windows 11 verwendet. Um die minimale Latenz zu ermitteln wurden Aufgabenteile 1. und 2. mit zwei Threads im selben Prozess durchgeführt, da hierbei die Latenz geringer sein sollte, als bei einer Implementierung in verschiedenen Prozessen. Zusätzlich wurde C als Programmiersprache verwendet, da diese eine der performantesten Programmiersprachen ist.

1. Kommunikation über Spinlocks (Busy Waiting) zwischen zwei Threads innerhalb eines Prozesses

Bei dieser Umsetzung sind zunächst schlechte Ergebnisse vorgekommen, da die Messung der Zeit vor beziehungsweise nach der Schleife durchgeführt wurde. Dadurch waren die Zeiten gleich lange wie die Ergebnisse aus Aufgabenteil 2., bei der die gleiche Vorgehensweise für die Berechnung der Zeit gewählt wurde. Die Zeit lag hierbei bei einer Medianzeit von ~2,5 ms und einer Standardabweichung von 0,118ms für beide Programme.

Nachdem die Zeitmessung innerhalb der Schleife vorgenommen wurde, haben sich die Zeiten extrem verbessert, sodass die Angabe der Zeit auf ns gewechselt wurde, da ansonsten 0 ausgegeben wurde. Zusätzlich konnte durch die Speicherung der Zeit in einem lokalen Array des Threads statt einem Array für beide Threads die Zeit ebenfalls so gering wie möglich gehalten werden.

Die neuen Zeiten waren schließlich folgende:

```
sebi@DESKTOP-QSERSJA:~$ ./BS_A2_1
Thread 0 - Mittelwert der Zeit: 44.108300 Nanosekunden
Thread 0 - Standardabweichung der Zeit: 112.496171 Nanosekunden
Thread 0 - 95% Konfidenzintervall: [41.903375, 46.313225] Nanosekunden
Thread 1 - Mittelwert der Zeit: 43.020100 Nanosekunden
Thread 1 - Standardabweichung der Zeit: 5.145415 Nanosekunden
Thread 1 - 95% Konfidenzintervall: [42.919250, 43.120950] Nanosekunden
Final shared data: 20000
```

Wie zu sehen ist, sind die Zeiten extrem niedrig mit einem Mittelwert von 44 beziehungsweise 43 ns. Erstaunlich ist jedoch, dass die Werte zwischen den beiden Threads sehr unterschiedlich sind, was die Standardabweichung bzw. Das 95% Konfidenzintervall betrifft. In Thread 1 sind diese extrem nahe bei einander und es ergibt

sich nur eine sehr geringe durchschnittliche Abweichung. In Thread 0 jedoch wurde eine Standardabweichung von 112 ns erhalten, was fast das Dreifache des Mittelwerts ist, obwohl das Konfidenzintervall nicht sehr viel größer ist, als in Thread 0. Sowohl die untere als auch die obere Grenze des Konfidenzintervalls hat hier eine Abweichung von ~5% zum Mittelwert.

2. Kommunikation über Semaphore zwischen zwei Threads innerhalb eines Prozesses

Für Semaphore wurde wie in Aufgabenteil 1 ein C-Code erstellt, der zwei Threads innerhalb eines Prozesses erstellt. Im Vergleich dazu, waren die Zeiten jedoch um "einiges schlechter" als bei Spinlocks:

```
sebi@DESKTOP-QSERSJA:~$ ./BS_A2_2
Thread 1 - Mittelwert der Zeit: 76.889200 Nanosekunden
Thread 1 - Standardabweichung der Zeit: 524.866423 Nanosekunden
Thread 1 - 95% Konfidenzintervall: [66.601818, 87.176582] Nanosekunden
Thread 0 - Mittelwert der Zeit: 74.334600 Nanosekunden
Thread 0 - Standardabweichung der Zeit: 419.292598 Nanosekunden
Thread 0 - 95% Konfidenzintervall: [66.116465, 82.552735] Nanosekunden
Final shared data: 20000
```

Der Mittelwert lag hier bei 76 beziehungsweise 74 ns. Das schlechtere 95% Konfidenzintervall war hierbei zwischen 66 und 87 ns. Ebenfalls waren auch die Abweichungen vom Mittelwert mit 15% bzw. 12% auch weitaus größer als in Aufgabe 1, was zeigt, dass der Spread der Werte weitaus größer war, als zuvor. Dies ist ebenfalls an der viel höheren Standardabweichung erkennbar.

Für diesen Aufgabenteil wurde zwar ein langsamerer Wert als in Aufgabenteil 1 auch erwartet, jedoch sind die Werte vergleichsweise nahe an den anderen Werten dran.

3. Kommunikation über die universelle Message-Queue ZeroMQ zwischen zwei Threads innerhalb eines Prozesses UND zwischen zwei Threads in verschiedenen Prozessen.

Für diesen Aufgabenteil sollten beide Varianten programmiert werden. Für diese Aufgabe war zu erwarten, dass die Ergebnisse der Message-Queue ZeroMQ langsamer als Semaphore ist und ZeroMQ zwischen zwei Threads innerhalb eines Prozesses schneller ist als in verschiedenen Prozessen.

Die Ergebnisse spiegelten die Erwartungen hierbei wider:

```
Thread 1 - Mittelwert der Zeit: 27958.913000 Nanosekunden
Thread 1 - Standardabweichung der Zeit: 9689.115993 Nanosekunden
Thread 1 - 95% Konfidenzintervall: [27769.006327, 28148.819673] Nanosekunden
Thread 0 - Mittelwert der Zeit: 27951.165400 Nanosekunden
Thread 0 - Standardabweichung der Zeit: 9088.989497 Nanosekunden
Thread 0 - 95% Konfidenzintervall: [27773.021206, 28129.309594] Nanosekunden
```

Innnerhalb eines Prozesses lagen die Werte bei ~27955 ns und hatten ein 95% Konfidenzintervall von 27769 ns – 28148 ns. Verglichen mit dem Mittelwert sind hierbei die Konfidenzintervalle sehr schmall. Ebenfalls ist hier auch die Standardabweichung im Vergleich zum Mittelwert geringer, wenn auch absolut viel größer.

```
sebi@DESKTOP-QSERSJA:~$ ./BS_A2_3_receiver &
[2] 5819
sebi@DESKTOP-QSERSJA:~$ ./BS_A2_3_sender
Sender - Mittelwert der Zeit: 97816.867400 Nanosekunden
Sender - Standardabweichung der Zeit: 380806.793230 Nanosekunden
Sender - 95% Konfidenzintervall: [90353.054253, 105280.680547] Nanosekunden
[2]-  Done                    ./BS_A2_3_receiver
```

Die Werte mit zwei verschiedenen Prozessen, wobei ein Receiver und ein Sender erstellt wurden, waren wie zu erwarten schlechter als in einem Prozess. Interessanterweise waren die Zeiten mit 97816 ns mehr als dreimal höher als in einem Prozess. Ebenfalls ist die Standardabweichung wieder höher als der Mittelwert und der Spread des Konfidenzintervalls auch wieder höher. Sowohl absolut als auch prozentual.

4. Kommunikation zwischen zwei Anwendungsthreads in verschiedenen Docker-Containern.

Da es sich bei dieser Aufgabe um eine Kommunikation in verschiedenen Docker-Containern handelt, hat es sich angeboten, den Code aus Aufgabenteil 3 für verschiedene Prozesse zu nutzen. Da bei diesem jedoch zuvor auf den localhost verbunden wurde und dieser nun verschieden war, musste der Code leicht angepasst werden.

Beim Ausführen des Codes mit Docker-Containern wurden folgende Ergebnisse erhalten:

```
sebi@DESKTOP-QSERSJA:~$ docker run --network=mynetwork --name receiver-container -d receiver-image
2982d8b82a3668748a26f725346617d550b67f306f2065aff5773b7da07bd4ed
sebi@DESKTOP-QSERSJA:~$ docker run --network=mynetwork --name sender-container -d sender-image
c3bdd8e376be91a6c0ee7b8cf0a27e4be3c44732bfea3c710155115cfacef771
sebi@DESKTOP-QSERSJA:~$ docker logs sender-container
Sender - Mittelwert der Zeit: 186760.765900 Nanosekunden
Sender - Standardabweichung der Zeit: 9227733.941931 Nanosekunden
Sender - 95% Konfidenzintervall: [5897.180638, 367624.351162] Nanosekunden
```

Wie zu erwarten, waren die Werte schlechter als ohne Docker, jedoch ”nur” etwa doppelt so langsam wie zuvor. Jedoch kann man anhand des Konfidenzintervalls und der Standardabweichung erkennen, dass der Spread der Werte enorm hoch ist. Dies könnte bedeuten, dass der Overhead durch die Docker-Container nicht optimal ist wodurch Latenz und Variabilität der Latenz erklärt werden können.

Finale Auswertung der gewonnenen Ergebnisse

Wie zu erwarten, ist Spinlocks mit wenigen ns am Schnellsten gewesen. Jedoch sollte hierbei angemerkt werden, dass diese Umsetzung bei mehreren Threads aufgrund des Busy Waiting zu schlechteren Ergebnissen führen wird und viele Ressourcen benötigen kann.

Erstaunlicherweise ist das Ergebnis mit Semaphoren vergleichsweise wenig schlechter als Spinlocks. Dadurch könnte es sein, dass sich Semaphore für viele Threads als effizienter erweisen könnte als Spinlocks.

Anhand von Aufgabenteilen 3 ist zu erkennen, dass komplexere Kommunikations-mechanismen zu schlechteren Zeiten führen und auch verschiedene Prozesse zu schlechteren Zeiten führen, da hierdurch ebenfalls eine zusätzliche Komplexität hinzugefügt wird.

Aufgabenteil 4 verdeutlicht, dass nicht nur die Kommunikation zwischen Prozessen, sondern auch die Kommunikation zwischen Servern/Systemen eine zusätzliche Komplexität und dadurch auch eine zusätzliche Zeit benötigen.