Die Aufgaben wurden im HTWG Container erledigt

9.2

. /lottery.py -j 2 -l 10:1,10:100

Der zweite Prozess der deutlich mehr Tickets hat wird in fast allen Fällen zuerst ausgeführt, er hat eine Wahrscheinlichkeit von mehr als 99% Chance ausgeführt zu werden mit 100 Tickets. Job 0 hat eine Wahrscheinlichkeit geringer als 1% vor Job 1 ausgeführt zu werden mit einem Ticket.

Job 0 hat 10-mal eine Chance von weniger als 1% vor Job 1 ausgeführt zu werden. Generell kann man sagen, dass wenn es eine solche unfaire Ticketverteilung gibt, Jobs höchstwahrscheinlich verhungern werden. Es besteht zwar eine kleine Chance, dass diese auch mal zwischendrin ausgeführt werden, diese ist aber gering und somit muss meistens auf den anderen Job gewartet werden. Bei unserer Versuchsdurchführung musste Job 0 solange warten, bis Job 1 komplett fertig war.

Die Flag

-j wird benutzt, um die Anzahl Jobs festzulegen.

-l wird benutzt, um die einzelnen Jobs zu spezifizieren: Joblänge: Ticketanzahl

9.3

./lottery.py -l 100:100,100:100 -s 1 -c

196 / 200 = 0.98

./lottery.py -l 100:100,100:100 -s 3 -c

199 / 200 = 0.995

./lottery.py -l 100:100,100:100 -s 10 -c

197 / 200 = 0.985

Wir haben den Versuch mit drei unterschiedlichen Seeds getestet, mit 1, 3 und 10.
Den schnelleren Job haben wir im Anschluss mit dem langsameren Job dividiert.
Das Ergebnis kann auch als Unfairness Metrik bezeichnet werden.
Dabei ist der Faktor 1 komplett fair. Die Ergebnisse sind aufgrund der gleichen Ticketverteilung von 100 sehr nahe an 1 und damit sehr fair.

Die Flag

-s wird benutzt, um den Seed zu setzen.

9.4

./lottery.py -q 1 -l 100:100,100:100 -c

192 / 200 = 0.96

./lottery.py -q 2 -l 100:100,100:100 -c

188 / 200 = 0.94

./lottery.py -q 5 -l 100:100,100:100 -c

160 / 200 = 0.8

./lottery.py -q 20 -l 100:100,100:100 -c

140 / 200 = 0.7

./lottery.py -q 100 -l 100:100,100:100 -c

100 / 200 = 0.5

Wir haben den Versuch mit 5 unterschiedlichen Quantumlängen getestet. 1, 2, 5, 20 und 100.
Desto höher die Quantumlänge, desto unfairer wird der Scheduler selbstverständlich. Da die Jobs pro Time Slice länger ausgeführt werden, gibt es weniger time Slices insgesamt.

Der Test bestätigt die Theorie, denn desto höher die Quantumlänge wird, desto niedriger wird die Unfairness Metrik. So sind wir bei einer Quantumlänge von 1 noch bei 0.96, also nahezu komplett fair. Bei einer Quantumlänge von 100 hingegen sind wir schon bei einer Unfairness Metrik von 0.5, was in dem Versuch das unfairste Ergebnis ist, dass es gibt.

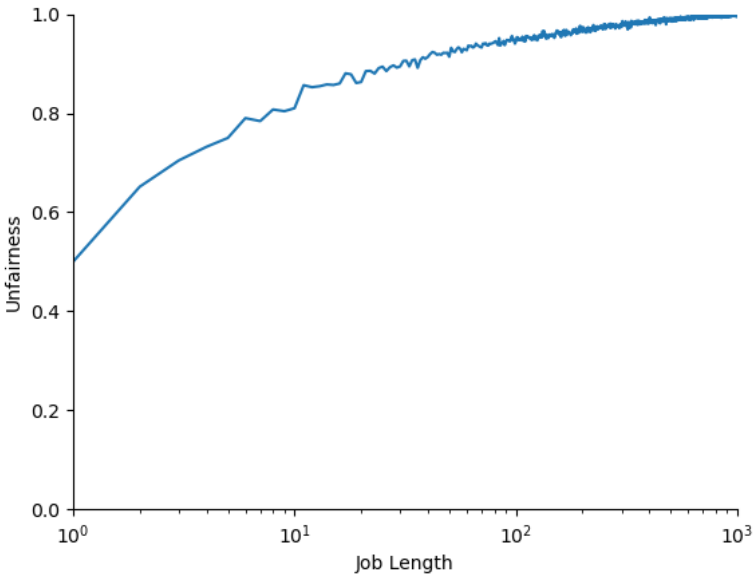
Die Flag

-q wird genutzt, um die Quantumlänge zu setzen.

9.5

Die Response time wäre noch sehr interessant, um diese mit der Unfairness Metrik zu vergleichen, da zum Beispiel bei einer hohen Quantumlänge nicht nur die Unfairness Metrik wächst, sondern vermutlich auch die Response time.
Auch die Turnaround time wäre wahrscheinlich auch noch interessant zu sehen. Ist vermutlich so dass bei variierender unfair Metrik auch die Turnaround time schwankt.

Lottery:



Wie wir in dem Schaubild oben sehen, wird der Graph mit zunehmendem Job länge fairer. Bei einem Job Länge von 1 mit einer Quantumlänge von 1 ist er natürlich am unfairsten mit 0.5 (1 / 2 = 0.5) und nähert sich daraufhin der 1.0 an, dem fairsten Wert der Unfairness Metrik.

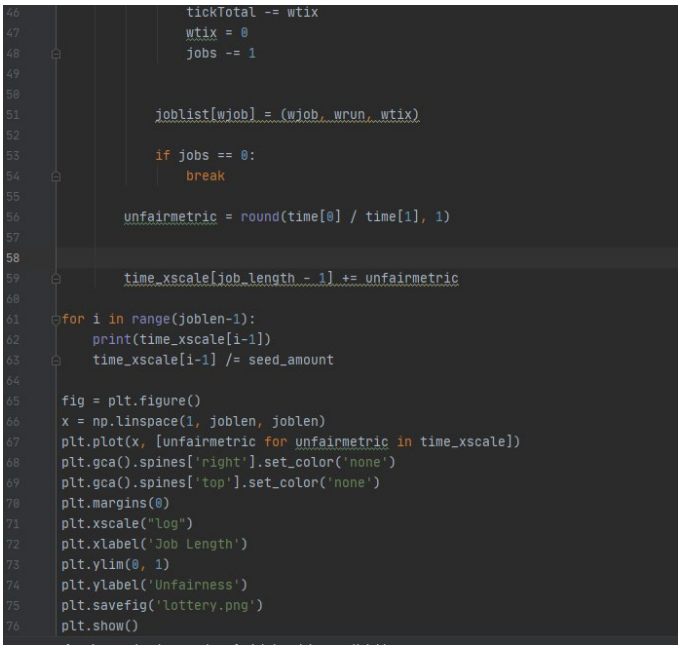
```
seed = 0
joblen = 1000
joblenone = joblen + 1
seed_amount = 200
time_xscale = [0] * joblen
for a in range(seed_amount):
    seed = a
    for job_length in range(1, joblenone):
        joblist = [[0, job_length, 100], [1, job_length, 100]]
        jobs = len(joblist)
        tickTotal = 100 * jobs
        runTotal = jobs * job_length
        quantum = 1
        clock = 0
        time = []

        for b in range(runTotal):
            r = int(random.random() * tickTotal)
            winner = int(r % tickTotal)

            current = 0
            wjob = 0
            wrun = 0
            wtix = 0
            for (job, runtime, tickets) in joblist:
                current += tickets
                if current > winner and runtime > 0:
                    (wjob, wrun, wtix) = (job, runtime, tickets)
                    break
```

Insgesamt haben wir 200 Seeds durchlaufen lassen und wie im Buch bis zu einem Job Länge von 1000 simuliert.

Dabei zählen wir in einer for-Schleife jeweils die Seeds hoch. In der nächsten for-Schleife wird für jede Job länge jeweils 200-mal (für jeden Seed) die Schleife durchgeführt. Im Anschluss ziehen wir wie in der Originaldatei lottery.py ein Gewinnerticket und vollführen den Ablauf wie in der Originaldatei.



```
                tickTotal -= wtix
                wtix = 0
                jobs -= 1


            joblist[wjob] = (wjob, wrun, wtix)

            if jobs == 0:
                break

        unfairmetric = round(time[0] / time[1], 1)


        time_xscale[job_length - 1] += unfairmetric

for i in range(joblen-1):
    print(time_xscale[i-1])
    time_xscale[i-1] /= seed_amount

fig = plt.figure()
x = np.linspace(1, joblen, joblen)
plt.plot(x, [unfairmetric for unfairmetric in time_xscale])
plt.gca().spines['right'].set_color('none')
plt.gca().spines['top'].set_color('none')
plt.margins(0)
plt.xscale("log")
plt.xlabel('Job Length')
plt.ylim(0, 1)
plt.ylabel('Unfairness')
plt.savefig('lottery.png')
plt.show()
```

Zum Schluss werden noch die Zeiten dividiert und für die jeweiligen Seeds addiert. Im Anschluss gehen wir erneut in eine for-Schleife, um den Durchschnitt pro Job Länge zu berechnen. Danach plotten wir die Ergebnisse.

Stride:

Ich interpretiere die Aufgabe so, dass ich beschreiben soll wie der Graph mit einem Stride Scheduler aussehen würde. Der Stride Scheduler würde ebenfalls mit dem Wert 0.5 anfangen (1 / 2 = 0.5).

Allerdings würde sich der Stride Scheduler um einiges schneller der 1.0 annähern, da dieser nicht über Zufallswerte geht, sondern darüber welcher Job den geringsten pass Wert hat. Somit in unserem Beispiel (100:100,100:100) wechselt er den Job nach jedem time slice und wird mit zunehmendem Job länge noch schneller fairer als der Lottery Scheduler.