Aufgabe 4: Fahrradwerkstatt

Team-ID: 00811

Team: Lorian

Bearbeiter/-innen dieser Aufgabe:
Lorian Linnertz

15. November 2022

Inhaltsverzeichnis

Lösungsidee

Aufgabe 4) 1) Der Lösungs Ansatz hier für ist simpel: Man erstellt eine Klasse, welche alle Informationen der Aufträge enthält. Dann erstellt man eine Template Klasse, welche als Vorlage für alle anderen Klassen funktioniert und allgemeine Funktionen, wie das Updaten der Uhr, das überprüfen ob neue Einträge reingekommen sind oder auch ob der Arbeitstag zuende ist.

Zum genauen Ablauf, in der Klasse des jeweiligen Verfahrens wird entschieden welcher Auftrag als nächstes Abgearbeitet wir, die Dauer dieses Auftrags wird dann auf die Uhr addiert: Anschließend wird der Auftrag in die Liste der Erledigten Aufträge gestellt, während man ebenfalls überprüft, ob der Arbeitstag beendet ist. Falls dies der Fall sein sollte sowird die Uhr auf den nächsten Arbeitsbeginn vorgedreht und am Ende überprüft man noch ob ein neuer Auftrag reingekommen ist.

Das Komplexeste hierbei, ist das Regeln der Arbeitstage. Dazu verwendet man zwei Gleichungen:

Arbeitstagbeginn(Tag) = 9h*60 + 24h*60*Tag =540 + 1440*Tag

Arbeitstagende(Tag) = 17h*60 + 24h*60*Tag= 960 + 1440*Tag

Wenn man jetzt noch die Tage zählt erhält man aus den Gleichungen jeweils den Arbeitsbeginn des Tages und das Arbeitsende. Das einzige was man jetzt noch tun muss ist zu überprüfen ob man nach einem vollendeten Auftrag über dem Arbeitstagende ist. Sollte dies Zutreffen, so nimmt man die Differenz zwischen der tatsächlichen Uhr und dem Arbeitstagende: Anschließend berechnet man den nächsten Arbeitstagsbeginn und addiert die Differenz auf diesen nächsten Arbeitstagsbeginn.

Aufgabe 4) 2) Das neue Program reduziert zwar die maximale Wartezeit, jedoch erhöht sich damit auch die Durchschnittliche Wartezeit. Damit hat man zwar weniger Einzelfälle, welche insgesamt lange Warten müssen, dafür hat man jedoch viele Kunden welche sich durch die längeren Durchschnittswarte Zeiten aufregen.

Aufgabe 4) 3) Wenn wir uns ein vereinfachten Sachverhalt anschauen. Bei welchem Marc 24/7 arbeitet, können wir 4 Schlussfolgerungen machen.

1) Das Verfahren **First-in First-out** (Sein altes Verfahren): hat die geringste Maximale Wartezeit, da immer die Leute dran kommen, welche bereits am längsten Warten.

2) Das Verfahren **Shortest-First** (Sein neues Verfahren): hat die kleinst mögliche Durchschnittliche Wartezeit, da sie immer die kürzesten als erstes Abarbeitet. So ist die Mehrheit, der Auftrag schnell bearbeitet und da der Durchschnitt alle Aufträge gleich gewichtet bedeutet das die kleinst mögliche Durchschnittliche Wartezeit.

3)Das Verfahren **First-in Last-out** (Gegenteil seines alten Verfahrens): hat die längst mögliche Durchschnittliche Wartezeit, da immer die neuen Aufträge bearbeitet werden womit die ersten Aufträge warten müssen bis alle anderen Abgearbeitet sind.

4)Das Verfahren **Shortest-Last** (Sein neues Verfahren): hat die größt mögliche Durchschnittliche Wartezeit, da sie immer die kürzesten als letztes abarbeitet. Somit brauchen die Aufträge mit kurzer Bearbeitungszeit am längsten um bearbeitet zu werden und da der Durchschnitt alle Aufträge gleich gewichtet bedeutet das die größt mögliche Durchschnittliche Wartezeit.

Aus diesen Erkenntnissen lässt sich Schlussfolgern dass das Kundenfreundlichste Verfahren eine Kombination aus dem 1) und 2) Verfahren ist. Mein Verfahren verwendet somit abwechselnd das 1) und 2) Verfahren.

Umsetzung

Im ersten Schritt, erstellen wir die Aufträge in Form der Klasse Order. Die jeweiligen Verfahren erben die Klasse Template, in welcher sich der ganze relevante Code befindet.

Unterschiedlichen Algorithmen:

Beim **Alten Verfahren**, sortieren wir die Incoming_Orders nach Eintreffzeitpunkt und entnehmen immer das Element [0]

Beim **Neuen Verfahren,** sortieren wir die Incoming_Orders nach der Dauer und entnehmen immer das Element [0]

Bei **meinem Verfahren** wechseln wir bei jedem Durchgang zwischen dem alten und dem neuen Verfahren.

Algorithmus

Benötigte Variabeln

self.order_list => diese Liste enthält zu Beginn alle Aufträge

self.incoming_orders => diese Liste enthält alle Aufträge, welche reingekommen sind

self.finished_orders => diese Liste enthält alle beendeten Aufträge

self.clock => in dieser Variablen wird die Zeit gespeichert

self.day_counter => zählt die Tage

self.working_day => eine Liste welche in Form von Tupel (Arbeitsbeginn, Arbeitsende), alle Arbeitstage enthällt

Algorithmus

1) Initalisierung der Uhr

A) Berechnet den 1.Arbeitstag, sortiert die Liste nach den Eintreffzeitpunkt

B) Sucht mit der adjust_clock Funktion den ersten Auftrag.

→ Der nächst mögliche Auftrag ist immer das Element [0] aus der Order_list, da wir diese vorhin nach Eintreffzeitpunkt sortiert haben.

I) Falls der erste Auftrag vor dem aktuellen Arbeitstag ein

a) So wird die Uhr nach vorne gedreht

b) Der Auftrag wird zur Liste incoming_orders hinzugefügt

II) Falls der erste Auftrag während der Arbeitszeit eintrifft, so

a) Die Uhr wird auf das Eintreffen des Auftrages nach vorne gedreht

b) Der Auftrag wird zur Liste incoming_orders hinzugefügt

III) Falls der erste Auftrag nach dem Arbeitsende eintrifft

a) So berechnet man den nächsten Tag und fängt wieder bei Schritt I) an.

2) Man wähle den nächsten Auftrag entsprechen des Verfahrens aus incoming_orders,

A) Man speichert in der Klasse des Auftrags den Auftragsbeginn und addiert anschließend die Dauer des Auftrags auf die clock

B) Man überprüft ob der Auftrag zu lange war und das Arbeitsende überzogen hat.

I) Sollte dies zutreffen, so wird der nächste Tag

II) Nun wir auf die aktuallisiert Uhr die überzogen Zeit drauf addiert

C) Man überprüft nun ob neue Aufträge reingekommen sind.

D) Sollte die Liste incoming_orders leer sein, so sucht man mit der Funktion adjust_clock nach neuen Aufträgen. Die Funktion wird in Schritt 1)B) erklärt.

E) Überprüft ob sowohl die Liste order_list und incoming_orders leer ist.

I) Sollte dies zutreffen, so ist der Algorithmus beendet

II) Sollte dies nicht zutreffen, so beginnt man wieder bei Schritt 2)

3) Berechnet sowohl die durchschnittliche als auch die Maximale Wartezeit und gibt diese aus

Beispiele

----------fahrradwerkstatt0.txt----------

Altes Verfahren: Durchschnittliche Wartezeit: 2750 Maximale Wartezeit: 12982
Neues Verfahren: Durchschnittliche Wartezeit: 2073 Maximale Wartezeit: 22850
Mein Verfahren: Durchschnittliche Wartezeit: 2551 Maximale Wartezeit: 13188

----------fahrradwerkstatt1.txt----------

Altes Verfahren: Durchschnittliche Wartezeit: 90898 Maximale Wartezeit: 184654
Neues Verfahren: Durchschnittliche Wartezeit: 16777 Maximale Wartezeit: 415423
Mein Verfahren: Durchschnittliche Wartezeit: 62436 Maximale Wartezeit: 214695

----------fahrradwerkstatt2.txt----------

Altes Verfahren: Durchschnittliche Wartezeit: 2904 Maximale Wartezeit: 16895
Neues Verfahren: Durchschnittliche Wartezeit: 1814 Maximale Wartezeit: 34547
Mein Verfahren: Durchschnittliche Wartezeit: 2579 Maximale Wartezeit: 22550

----------fahrradwerkstatt3.txt----------

Altes Verfahren: Durchschnittliche Wartezeit: 2259 Maximale Wartezeit: 15386
Neues Verfahren: Durchschnittliche Wartezeit: 1903 Maximale Wartezeit: 37642
Mein Verfahren: Durchschnittliche Wartezeit: 2057 Maximale Wartezeit: 17260

----------fahrradwerkstatt4.txt----------

Altes Verfahren: Durchschnittliche Wartezeit: 3044 Maximale Wartezeit: 17041
Neues Verfahren: Durchschnittliche Wartezeit: 2229 Maximale Wartezeit: 27605
Mein Verfahren: Durchschnittliche Wartezeit: 2487 Maximale Wartezeit: 18087

Quellcode

```
class Order(): #Diese Klasse soll die Eingenschaften der Aufträge speichern
    def __init__(self,input_time,duration):
        self.input_time = input_time
        self.duration = duration
        self.order_start = None

    def waiting_time(self): #Diese Funktion soll die Wartezeit zurückgeben
        waiting_time = self.order_start - self.input_time #Die Wartezeit ist Definiert als die differenz zwischen der Eingabezeit und dem Auftragsbeginn
        return waiting_time

    def prt_information(self):
        print(f"Eingangszeitpunkt: {self.input_time}\t Dauer: {self.duration}\t Arbeitsbeginn: {self.order_start}, Wartezeit: {Order.wartezeit(self)}")
        pass

#---------------------------------------------------

class Template:
    def __init__(self,order_list):
        self.order_list = copy.deepcopy(order_list)
        self.incoming_orders = []
        self.finished_orders = []
        self.day_counter = 0 #Diese Variable soll die Arbeitstage zählen
        self.working_day = [] #in working_day sollen die Uhrzeiten der Arbeitstage in Form von (day_start,day_end) hinzugefügt werden
        Template.init_clock(self)

    def next_day(self):
        day_start = 540 + 1440*self.day_counter #Berechnet den Anfang des nächsten Tages
        day_end = 960 + 1440*self.day_counter #und das Ende
        self.working_day.append((day_start,day_end)) #Fügt den neuen Arbeitstag in die Liste
        self.day_counter += 1 #Erhöht den Tageszähler um 1

    def init_clock(self):
        self.order_list.sort(key=sort_input_time) #Sortiert die Liste, so dass die Aufträge welche die kleinste input_time haben vorne stehen
        first_incoming_order = self.order_list.pop(0) #Entnimmt den ersten Auftrag aus der Liste
        Template.next_day(self)
```

Template.adjust_clock(self)

Template.check_incoming_orders(self) #Im letzten Schritt überprüft man (in Fall 1) ob während man auf den Arbeitsbeginn wartet keine neuen Aufträge reingekommen sind.(Fall 2): ob zum first_incoming_order nicht noch parallel eine zweite eingegangen ist

def adjust_clock(self):

next_possible_order = self.order_list.pop(0)

while True: #Die while_Schlife dient dazu, im Falle dass der nächste Auftrag erst nach dem Arbeitstag eingeht, dieser Prozess so lange wiederholt wird bis der passende Arbeitstag gefunden wurde PS: dient zum sonderfall falls mehrere Tage keine neuen Aufträge rein kommen

if next_possible_order.input_time < self.working_day[-1][0]: # "self.working_day[-1][0]" Dieser Ausdruck schaut auf den Arbeitstag beginn des neusten Tages

#Die if-Schleife überprüft also, ob der Auftrag vor Arbeitsbeginn reingekommen ist.

self.incoming_orders.append(next_possible_order) #Falls dies zutriff, so wird der nächste Auftrag in die Wartelist gesetzt

self.clock = self.working_day[-1][0] #Und die Uhr wird auf den Arbeitstagbeginn gesetzt

break

elif next_possible_order.input_time >= self.working_day[-1][0] and next_possible_order.input_time < self.working_day[-1][1]: #"self.working_day[-1][1]" Dieser Ausdruck schaut auf das Arbeitstagsende des neusten Tages

#Diese if-Schleife überprüft ob der nächste Auftrag während einem Arbeitstag reingekommen ist.

self.incoming_orders.append(next_possible_order) #Falls dies zutriff, so wird der nächste Auftrag in die Wartelist gesetzt

self.clock = next_possible_order.input_time #Ebenfalls wird die Uhr auf die input_time des Auftrages nachvorne gedreht.

break

Template.next_day(self)

def check_incoming_orders(self):

while True:

if len(self.order_list) != 0 and self.order_list[0].input_time <= self.clock: #Überprüft als erstes ob noch ein Element in order_list ist und als zweites ob das Element[0] eine Input_time hat welche kleiner ist als die aktuelle Zeit.

new_order = self.order_list.pop(0) #Falls dies zutrifft so wird Element[0] der order_list entnommen

self.incoming_orders.append(new_order) #und zur incoming_orders list hinzugefügt

else: #Falls die obrige bedingung nicht zutrifft so wird die Schleife beednet

break

if len(self.incoming_orders) == 0: #Als letztes wird überprüft ob in incoming_orders kein Auftrag ist

Template.adjust_clock(self) #Falls dies Zutrifft, so wird die Uhr so angepasst, dass es wieder ein Auftrag gibt, denn es muss immer mindestens 1 Auftrag zur verfügung stehen

def update_clock(self,order):#Diese Funktion soll die Uhr updaten und gegebenden falls einen neuen Tag einleiten

self.clock += order.duration

if self.clock > self.working_day[-1][1]: #Falls der Auftrag über die Arbeitszeit hinaus geht, so müssen die überzogenen Minuten dem nächsten Arbeitstag zu gerechenet werden

```
        delta_duration = self.clock - self.working_day[-1][1] #Nimm die differenz aus der Zeit an welchem der Auftrag beendet wäre und dem Arbeitsende
        Template.next_day(self)
        self.clock = self.working_day[-1][0] + delta_duration #setzt die Uhr auf den Arbeitsbeginn des nächsten Tages und addiert noch die Überzogenen Minuten hinzu, so dass der Auftrag vollendet ist ohne überstunden



class FirstIn_FirstOut(Template):#Diese Klasse simuliert das Verfahren, bei welchem die Aufträge der Reihenfolge nach abgearbeitet werden
    def __init__(self,order_list):
        super().__init__(order_list) #Erbt alles von der Template Klasse
        FirstIn_FirstOut.create_workplan(self) #und startet den Algorithmus


    def create_workplan(self):
        while True:
            FirstIn_FirstOut.check_incoming_orders(self) #Überpfüft on keine neuen Aufträge eingekommen sind
            self.incoming_orders.sort(key=sort_input_time) #Sortiert die Aufträge nach ihrer Dauer
            target_order = self.incoming_orders.pop(0) #Entnimmt den Auftrag mit der geringsten input_time
            target_order.order_start = self.clock #setzt fest wann der Auftrag begonnen wurde
            self.finished_orders.append(target_order)
            Template.update_clock(self,target_order)
            if len(self.order_list) == 0 and len(self.incoming_orders) == 0: #Falls sowohl order_list als auch incoming_orders leer ist, so ist der Algorithmus fertig
                break


class Shortest_First(Template):
    def __init__(self,order_list):
        super().__init__(order_list)
        Shortest_First.create_workplan(self)


    def create_workplan(self):
        while True:
            Shortest_First.check_incoming_orders(self) #Überpfüft on keine neuen Aufträge eingekommen sind
            self.incoming_orders.sort(key=sort_duration) #Sortiert die Aufträge nach ihrer Dauer
            #print(self.clock-list(map(sort_input_time,self.incoming_orders))[0])


            target_order = self.incoming_orders.pop(0) #Entnimmt den Auftrag mit der geringsten input_time
            self.finished_orders.append(target_order)
            target_order.order_start = self.clock
            Template.update_clock(self,target_order)
```

```
        if len(self.order_list) == 0 and len(self.incoming_orders) == 0:
            break
class Compromis(Template): #Dies ist mein Lösungsansatz. Es ist ein Kompromis aus den beiden Vorherigen
    def __init__(self,order_list):
        super().__init__(order_list) #Erbt alles von der Template Klasse
        Compromis.create_workplan(self) #und startet den Algorithmus


    def create_workplan(self):
        counter = 0
        while True:
            Compromis.check_incoming_orders(self) #Überpfüft on keine neuen Aufträge eingekommen sind
            sort_keys = {0:sort_input_time,1:sort_duration}
            self.incoming_orders.sort(key=sort_keys[counter%2]) #Sortiert die Aufträge nach ihrer Dauer
            target_order = self.incoming_orders.pop(0) #Entnimmt den Auftrag mit der geringsten input_time
            target_order.order_start = self.clock #setzt fest wann der Auftrag begonnen wurde
            self.finished_orders.append(target_order)
            Template.update_clock(self,target_order)
            if len(self.order_list) == 0 and len(self.incoming_orders) == 0: #Falls sowohl order_list als auch incoming_orders leer ist, so
ist der Algorithmus fertig
                break
            counter += 1
```