Aufgabe 4

Lösungsidee

Wie auch bei Aufgabe 2 ist der Lösungsweg hier ziemlich klar vorgegeben. Marcs Werkstatt soll hier mithilfe sogenannter "Scheduling-Algorithmen", wie sie zum Beispiel im Kernel von Computern zum Einsatz kommen, optimiert werden. Es werden zwei dieser Algorithmen vorgeschlagen: "first-in-first-out" und "shortest-job-next". Ihre Namen sagen bereits aus, wie sie funktionieren: während der erste die Aufträge in chronologischer Reihenfolge bearbeitet, priorisiert der andere kurze Prozesse. Das ist praktisch, da es die durchschnittliche Wartezeit gegenüber dem ersten Verfahren deutlich senkt, ist allerdings auch nur umsetzbar, wenn die benötigte Zeit im Voraus bekannt ist. Dafür steigt allerdings auch die maximale Wartezeit, da der längste Prozess wahrscheinlich erst gegen Ende angefangen wird, sofern Marc nicht weniger Aufträge bekommt als er in der gleichen Zeit abarbeiten kann.

Umsetzung

Als Programmiersprache habe ich Python gewählt. In meinem Programm wird zuerst für jeden Auftrag ein Objekt angelegt, das die gegebenen Attribute festhält und auf Anfrage die Endzeit der Bearbeitung und die Wartezeit berechnet, sobald die Startzeit bekannt ist. Das ist sie natürlich nicht von Beginn an, weshalb die Aufträge dann nach ihrem Eingangszeitpunkt sortiert werden, da das Programm nicht mehr wissen soll als Marc. Dieser kann leider nicht in die Zukunft schauen und muss deshalb mit den ihm bekannten Informationen auskommen. Je nach Verfahren beginnt Marc dann entweder den ältesten oder den kürzesten vorliegenden Auftrag. Ist dieser beendet, wird die Auswahl wiederholt, bis alle Aufträge des Beispiels beendet wurden. Dann werden die durchschnittliche und die maximale Wartezeit berechnet, wodurch die Priorisierung der beiden Verfahren verdeutlicht wird, da bei FIFO immer die maximale Wartezeit niedrig gehalten wird, wohingegen bei SJN die durchschnittliche Wartezeit besser dasteht, als bei FIFO.

zu 2)

Es werden auch hier nicht alle zufrieden sein, da ohnehin nie alle Kunden zufrieden sind. Abgesehen davon müssen hier die Kunden mit den längsten (und damit teuersten) Aufträgen am längsten warten. Da Marcs Preise nicht bekannt sind und damit auch nicht, ob er sich je Stunde oder pauschal bezahlen lässt, ist hier keine eindeutige Bewertung für Marcs Seite möglich. Allerdings ist es denkbar, dass diejenigen, die die längsten Aufträge abgeben auch am meisten Geld bezahlen. Diese Kunden mit schlechtem Service (-> langer Wartezeit) zu enttäuschen, würde sicherlich nicht gut bei ebendiesen Kunden ankommen. Andererseits kann ihm das jedoch egal sein, wenn er ohnehin voll ausgelastet ist und immer die gleiche Bezahlung pro Arbeitsstunde bekommt. Dann würde er nämlich bei kleinen Aufträgen genau das gleiche an einem Arbeitstag verdienen wie bei großen und er bräuchte sich nicht dafür zu interessieren, was für einen Auftrag er gerade annimmt. Ist das jedoch nicht gegeben und Marco hat aus Profitgründen ein Interesse daran, teurere (-> längere) Aufträge anzunehmen, sollte er sein erstes Verfahren beibehalten, oder das Gegenteil des SJN-Verfahrens machen: immer den längsten und profitabelsten Auftrag anzunehmen, der vorliegt.

zu 3)

Wäre ich Marco, würde ich Profit priorisieren. Wie schon in 2 ausgeführt gibt es dabei für verschiedene Ausgangssituationen verschiedene optimale Methoden. Allerdings könnte er auch eine "Prioritätsgebühr" einführen, sodass Kunden, die es eilig mit ihren Aufträgen haben, einen Aufpreis zahlen. Dieser ist reiner Profit für Marc, abgesehen davon dass die anderen Aufträge nach hinten verschoben und dadurch eventuell Kunden verärgert werden. Oder Marc könnte "Überstundenzuschlag" verlangen, wenn das von Kunden gewünschte Zeitfenster sonst nicht eingehalten werden könnte. In jedem Fall wäre für mich der wichtigste Faktor bei der Auswahl der Aufträge, wie profitabel sie sind. Vielleicht ist auch eine Inspektion zum Beispiel weniger rentabel als ein Reifenwechsel und sollte daher bei der Auswahl bevorzugt werden. Da diese Zusatzinformationen nicht in den Beispielen abgebildet werden, habe ich sie nicht in das Programm eingeführt.

Beispiele

```
$ python aufgabe4.py fahrradwerkstatt0.txt
FIFO
time_finished: 404 Tage, 13 Stunden und 42 Minuten
avg_waiting_time: 22 Tage, 17 Stunden und 53 Minuten
max_waiting_time: 47 Tage, 18 Stunden und 11 Minuten

SJN
time_finished: 404 Tage, 13 Stunden und 42 Minuten
avg_waiting_time: 11 Tage, 19 Stunden und 1 Minuten
max_waiting_time: 131 Tage, 1 Stunden und 34 Minuten
```

```
$ python aufgabe4.py fahrradwerkstatt1.txt
FIFO
time_finished: 425 Tage, 11 Stunden und 3 Minuten
avg_waiting_time: 44 Tage, 2 Stunden und 55 Minuten
max_waiting_time: 89 Tage, 2 Stunden und 41 Minuten

SJN
time_finished: 425 Tage, 11 Stunden und 3 Minuten
avg_waiting_time: 8 Tage, 6 Stunden und 3 Minuten
max_waiting_time: 301 Tage, 2 Stunden und 3 Minuten
```

```
$ python aufgabe4.py fahrradwerkstatt2.txt
FIFO
time_finished: 383 Tage, 15 Stunden und 35 Minuten
avg_waiting_time: 35 Tage, 13 Stunden und 14 Minuten
max_waiting_time: 77 Tage, 1 Stunden und 33 Minuten

SJN
time_finished: 383 Tage, 15 Stunden und 35 Minuten
avg_waiting_time: 10 Tage, 6 Stunden und 53 Minuten
max_waiting_time: 227 Tage, 3 Stunden und 27 Minuten
```

```
$ python aufgabe4.py fahrradwerkstatt3.txt
FIFO
time_finished: 384 Tage, 15 Stunden und 8 Minuten
avg_waiting_time: 20 Tage, 20 Stunden und 28 Minuten
max_waiting_time: 42 Tage, 5 Stunden und 41 Minuten

SJN
time_finished: 384 Tage, 15 Stunden und 8 Minuten
avg_waiting_time: 11 Tage, 23 Stunden und 22 Minuten
max_waiting_time: 265 Tage, 6 Stunden und 56 Minuten
```

```
$ python aufgabe4.py fahrradwerkstatt4.txt
FIFO
time_finished: 408 Tage, 13 Stunden und 59 Minuten
avg_waiting_time: 51 Tage, 16 Stunden und 27 Minuten
max_waiting_time: 116 Tage, 0 Stunden und 19 Minuten

SJN
time_finished: 408 Tage, 13 Stunden und 59 Minuten
avg_waiting_time: 29 Tage, 7 Stunden und 20 Minuten
max_waiting_time: 252 Tage, 4 Stunden und 35 Minuten
```

Wie eindeutig zu erkennen ist, ist die maximale Wartezeit beim ersten Verfahren immer deutlich niedriger. Beim zweiten müsste ein Kunde meist über 200 Tage (!) warten, bis der Auftrag abgeschlossen ist. Am extremsten zeigt sich der Unterschied in der zweiten Beispieldatei: 89 Tage maximale Wartezeit gegenüber 300, aber 44 Tage Durchschnitt gegenüber 8.

Quellcode

```
from typing import TypedDict

# Minuten eines Arbeitstags
WORKDAY = 480
# Minuten eines Kalendertags
CALENDARDAY = 1440


def minutes_to_days(minuten: int) -> str:
    tage = minuten // 1440
    stunden = (minuten // 60) % 24
    rest = minuten % 60
    return "{} Tage, {} Stunden und {} Minuten".format(tage, stunden,
rest)
```

```python
class Job:
    time_received: int
    duration: int
    time_started: int | None

    def __init__(self, time_received: int, duration: int) -> None:
        # Initialisieren
        self.time_received = time_received
        self.duration = duration
        self.time_started = None

    @property
    def time_finished(self) -> int:
        assert type(self.time_started) == int
        full_days = (self.duration // WORKDAY) * CALENDARDAY
        remainder = self.duration % WORKDAY
        if remainder >
Workshop.remaining_working_minutes(self.time_started):
            remainder += 960
        return self.time_started + full_days + remainder

    @property
    def waiting_time(self) -> int:
        assert type(self.time_started) == int
        return self.time_finished - self.time_received


class Workshop:
    jobs: list[Job]
    current_time: int

    def __init__(self, path: str) -> None:
        with open(path, "r") as file:
            lines = file.readlines()
        self.jobs = [Job(*map(int, args.strip().split(" "))) for args in
lines if args != "\n"]
        self.current_time = 0

    class Result(TypedDict):
        time_finished: int
        avg_waiting_time: int
        max_waiting_time: int

    def fifo(self) -> Result:
        # zurücksetzen der Simulationsumgebung
        self.reset_environment()
        # sortieren der Aufträge nach chronologischer Reihenfolge
        sorted_jobs = sorted(self.jobs, key=lambda x: x.time_received)
```

```
        for job in sorted_jobs:
            received = job.time_received
            if received >= self.current_time:
                # Liegt der Eingang in der Zukunft, beginnt die Arbeit
bei Eingang des Auftrags
                job.time_started = received
            else:
                # Sonst wird sofort begeonnen
                job.time_started = self.current_time
            self.current_time = job.time_finished
        return {"time_finished": self.current_time,
                "avg_waiting_time": sum(job.waiting_time for job in
self.jobs) // len(self.jobs),
                "max_waiting_time": max(self.jobs, key=lambda x:
x.waiting_time).waiting_time
                }

    def sjn(self) -> Result:
        # zurücksetzen der Simulationsumgebung
        self.reset_environment()
        # sortieren der Aufträge nach chronologischer Reihenfolge
        sorted_jobs = sorted(self.jobs, key=lambda x: x.time_received)
        while sorted_jobs:
            # liegt aktuell nur ein Auftrag vor, wird dieser bearbeitet
            if self.current_time <= sorted_jobs[0].time_received <
sorted_jobs[1].time_received:
                job = sorted_jobs.pop(0)
                self.current_time = job.time_received
            # sonst wird der kürzeste vorliegende Auftrag ausgewählt
            else:
                if self.current_time < sorted_jobs[0].time_received:
                    self.current_time = sorted_jobs[0].time_received
                job = min(filter(lambda x: x.time_received <=
self.current_time, sorted_jobs),
                                key=lambda x: x.duration)
                sorted_jobs.remove(job)
            job.time_started = self.current_time
            self.current_time = job.time_finished
        return {"time_finished": self.current_time,
                "avg_waiting_time": sum(job.waiting_time for job in
self.jobs) // len(self.jobs),
                "max_waiting_time": max(self.jobs, key=lambda x:
x.waiting_time).waiting_time
                }

    @staticmethod
    def is_working_hours(time: int) -> bool:
        return 540 <= (time % CALENDARDAY) < 1020
```

```
    @staticmethod
    def remaining_working_minutes(time: int) -> int:
        return 1020 - time % CALENDARDAY

    def reset_environment(self) -> None:
        self.current_time = 0
        for job in self.jobs:
            job.time_started = None
```