Aufgabe 4: Fahrradwerkstatt

Team-ID: 00100

Team-Name: Team

Bearbeiter/-innen dieser Aufgabe:
Arne de Borman

21. November 2022

Inhaltsverzeichnis

1 Lösungsidee

Um die Aufgabe lösen zu können, brauche ich eine Datenstruktur, die angekommene Aufträge enthält und in einer bestimmten Reihenfolge zurückgibt. Dafür eignet sich die Binärer Heap Datenstruktur (englisch: Binary Heap).

Teilaufgabe 1

Der Heap soll nun entweder nach Größe der Aufträge oder nach Eingangszeitpunkten sortieren. Dafür wird für jeden Auftrag die "Unwichtigkeit" berechnet (ab jetzt aufgrund mangels eines besseren Wortes "Latenz" genannt). Bei dem ersten Verfahren entspricht die Latenz also dem Eingangszeitpunkt, bei dem zweiten der Größe des Auftrags.

Teilaufgabe 2

Beim zweiten Verfahren müssen größere Aufträge teilweise sehr lange warten, da kleine Aufträge immer vorher bearbeitet werden.

Teilaufgabe 3

Hier könnte man neben der durchschnittlichen und maximalen Bearbeitungsdauer auch den Median als Kennzahl benutzen. Den Median der Bearbeitungsdauern erhält man, indem man alle Dauern sortiert und dann die Mittlere nimmt. Er gibt eine Auskunft darüber, wie lange eine typische Wartezeit ist.

Ein besseres Verfahren wäre eine Kombination der beiden Ersten. Dies kann man erreichen, indem man die Latenz sowohl mithilfe der Größe des Auftrags, als auch mit dem Eingangszeitpunkt errechnet.

$$latency = size + start_t * f$$

Der Eingangszeitpunkt wird mit einem Faktor f multipliziert, sodass man kontrollieren kann, ob das Verfahren eher dem Ersten oder dem Zweiten ähnelt.

2 Umsetzung

Die Lösungsidee wird in rust implementiert. Der Binäre Heap ist schon in rusts Standard Bibliothek in std::collections::BinaryHeap implementiert. Um verschiedene Verfahren implementieren gibt es eine allgemeine Funktion mit dem Namen handle_tasks. Sie nimmt die Aufträge und eine Funktion als Eingabe.

```
fn handle_tasks(tasks: Vec<[i32; 2]>, latency_fn: fn(i32, i32) -> i32) {
//...
}
```

Die Funktion wird benutzt um die Latenz der Aufträge zu bestimmen. Innerhalb der Funktion gibt eine Variable die Zeit in Minuten an. Sie wird immer wieder erhöht. Wenn ein Task an der Reihe wird er im Heap mithilfe der Task Klasse gespeichert. Die Task Klasse enthält die Größe des Auftrags, wann er in der Werkstatt ankam, wie lange schon an ihm gearbeitet wurde (der Fortschritt) und die Latenz.

```
struct Task {
    size: i32,
    time_worked_on: i32,
    start_t: i32,
    latency: i32,
}
```

Nachdem ein Auftrag bearbeitet wurde, wird die Bearbeitungsdauer in einem Vec gespeichert (so ähnlich wie ein Array).

3 Beispiele

Wir rufen das rust-Programm mit den verschiedenen BWINF-Eingabedateien auf. Diese Dateien liegen im selben Ordner wie das Programm. Das Programm wird mithilfe des "cargo run"Befehls ausgeführt. Das Terminal muss sich währenddessen im selben Verzeichnis wie die Cargo.toml Datei befinden. Hier wird vorausgesetzt, dass rust installiert ist. Alternativ kann man auch das vorkompilierte Programm für das Betriebssystem ausführen (z.B. fahrradwerkstatt-windows.exe). Das Programm gibt dann die Kennzahlen für die verschiedenen Verfahren aus.

```
$ cargo run fahrradwerkstatt0.txt
Estes Verfahren (Aufträge in Reihenfolge):
durchschnittliche Wartezeit: 10d 15h 3min, max: 29d 9h 1min, median: 8d 5h 52min
Zweites Verfahren (kleinste Aufträge zuerst):
durchschnittliche Wartezeit: 6d 20h 50min, max: 74d 1h 57min, median: 3d 19h 23min
Eingenes Verfahren:
durchschnittliche Wartezeit: 7d 5h 51min, max: 45d 6h 59min, median: 4d 8h 22min

$ cargo run fahrradwerkstatt1.txt
Estes Verfahren (Aufträge in Reihenfolge):
durchschnittliche Wartezeit: 17d 0h 6min, max: 36d 0h 39min, median: 14d 20h 41min
Zweites Verfahren (kleinste Aufträge zuerst):
durchschnittliche Wartezeit: 3d 10h 15min, max: 233d 4h 46min, median: 0d 9h 0min
Eingenes Verfahren:
durchschnittliche Wartezeit: 14d 7h 23min, max: 44d 2h 25min, median: 12d 6h 40min

$ cargo run fahrradwerkstatt2.txt
Estes Verfahren (Aufträge in Reihenfolge):
durchschnittliche Wartezeit: 8d 3h 57min, max: 26d 4h 21min, median: 7d 1h 28min
Zweites Verfahren (kleinste Aufträge zuerst):
durchschnittliche Wartezeit: 4d 13h 38min, max: 40d 7h 55min, median: 3d 1h 52min
Eingenes Verfahren:
durchschnittliche Wartezeit: 4d 14h 26min, max: 40d 7h 55min, median: 3d 3h 44min

$ cargo run fahrradwerkstatt3.txt
```

```
Estes Verfahren (Aufträge in Reihenfolge):
durchschnittliche Wartezeit: 4d 18h 7min, max: 18d 1h 44min, median: 4d 3h 18min
Zweites Verfahren (kleinste Aufträge zuerst):
durchschnittliche Wartezeit: 3d 10h 54min, max: 58d 1h 12min, median: 2d 3h 19min
Eingenes Verfahren:
durchschnittliche Wartezeit: 3d 16h 24min, max: 18d 2h 37min, median: 2d 4h 41min

$ cargo run fahrradwerkstatt4.txt
Estes Verfahren (Aufträge in Reihenfolge):
durchschnittliche Wartezeit: 22d 20h 49min, max: 62d 7h 18min, median: 19d 0h 7min
Zweites Verfahren (kleinste Aufträge zuerst):
durchschnittliche Wartezeit: 13d 19h 12min, max: 175d 0h 19min, median: 5d 7h 7min
Eingenes Verfahren:
durchschnittliche Wartezeit: 16d 18h 50min, max: 65d 4h 37min, median: 10d 19h 20min
```

Man kann feststellen, dass bei dem ersten Verfahren die maximale und beim zweiten die durchschnittliche Wartezeit am kleinsten ist. Beim zweiten Verfahren ist die Wartezeit teilweise sehr lang, zum Beispiel beträgt die maximale Wartezeit beim zweiten Beispiel (fahrradwerkstatt1.txt) über 233 Tage. Beim dritten Verfahren treten so extreme Bearbeitungsdauern bei den Beispieldaten nicht auf, dafür ist die durchschnittliche Bearbeitungsdauer je nach Beispiel etwas höher als beim zweiten Verfahren.

Ich finde, dass das dritte Verfahren am besten geeignet ist, da hier keine Kunden durch außerordentlich lange Wartezeiten verärgert werden, die Aufträge aber trotzdem durchschnittlich schneller bearbeitet werden als beim Ersten.

4 Quelltext

```rust
use std::{
    cmp::{Ordering, Reverse}, //Wird benutzt, um die Reihenfolge der Aufträge im Heap zu bestimmen
    collections::BinaryHeap,  //der Binäre Heap
    env,
    fs,
};

//Klasse, repräsentiert einen Auftrag
struct Task {
    size: i32,
    time_worked_on: i32,
    start_t: i32,
    latency: i32,
}

//Bestimmt die Reihenfolge der Aufträge im Heap
impl PartialOrd for Task {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        //Es handlet sich bei dem Binären Heap normalerweise um einen Max-Heap, das heißt größere Elemente
        //kommen zuerst. Reverse kehrt die Reihenfolge um, sodass Elemente mit kleinerer Latenz zuerst kommen
        Reverse(self.latency).partial_cmp(&Reverse(other.latency))
    }
}

const MINUTES_PER_DAY: i32 = 60 * 24;
const NEUN_UHR: i32 = 9 * 60;
const NEUNZEHN_UHR: i32 = 19 * 60;

fn handle_tasks_in_order(tasks: Vec<[i32; 2]>) {
    println!("Estes Verfahren (Aufträge in Reihenfolge): ");
    handle_tasks(tasks, |start_t, _| start_t)
}
fn handle_tasks_min_first(tasks: Vec<[i32; 2]>) {
    println!("Zweites Verfahren (kleinste Aufträge zuerst): ");
    handle_tasks(tasks, |_, size| size)
}
```

```rust
const F: f64 = 0.1; //Kontrolliert wie sehr das Verfahren den anderen beiden ähnelt.
fn handle_tasks_balanced(tasks: Vec<[i32; 2]>) {
    println!("Eingenes Verfahren: ");
    handle_tasks(tasks, |start_t, size| size + (start_t as f64 * F) as i32)
}

//Bearbeitet Aufträge in einer bestimmten Reihenfolge
fn handle_tasks(tasks: Vec<[i32; 2]>, latency_fn: fn(i32, i32) -> i32) {
    let mut heap: BinaryHeap<Task> = BinaryHeap::new(); //Speichert eingegangene Aufträge
    let mut durations = Vec::with_capacity(tasks.len()); //Die Wartezeiten

    //Die noch nicht eingegangenen Aufträge. Durch ".peekable()" kann man schauen,
    //wann der nächste Auftrag eingeht, ohne den Iterator zu konsumieren
    let mut tasks = tasks.into_iter().peekable();
    let mut curr_t = 0;
    let mut curr_task: Option<Task> = None;
    loop {
        //immer wieder schauen, wann der nächste Auftrag ankommt
        while let Some([arrival_t, _]) = tasks.peek() {
            //falls der nächste Auftrag noch nicht angekommen ist, wird die Schleife unterbrochen
            if arrival_t > &curr_t {
                break;
            }
            // Den nächsten Auftrag "aufbrauchen" und die Daten erhalten
            let [arrival_t, duration] = tasks.next().unwrap();
            // Den Auftrag zu angekommenen Aufträgen zum Heap hinzufügen
            heap.push(Task::new(
                duration,
                0,
                arrival_t,
                latency_fn(arrival_t, duration), //die Latenz mithilfe der Funktion berechnen
            ));
        }
        if let Some(task) = &mut curr_task {
            //task ist momentan in bearbeitung
            let day_time = curr_t % MINUTES_PER_DAY; //Die Tageszeit
            if (NEUN_UHR..NEUNZEHN_UHR).contains(&day_time) {
                //Der aktuelle Auftrag wird weiterbearbeitet
                task.time_worked_on += 1;
                if task.time_worked_on >= task.size {
                    //Der Auftrag ist fertig bearbeitet
                    let waiting_t = curr_t - task.start_t;
                    durations.push(waiting_t);
                    curr_task = None;
                }
            }
        } else if let Some(task) = heap.pop() {
            //ein neuer Task wird aus dem Heap geholt und jetzt bearbeitet
            curr_task = Some(task);
        } else if tasks.peek().is_none() {
            //keine Aufträge mehr über => fertig
            break;
        }
        //Zeit um eine Minute erhöhem
        curr_t += 1;
    }
    //Die Kennzahlen berechnen
    let total_duration: i32 = durations.iter().sum();
    durations.sort();
    let median = durations[durations.len() / 2];
    let avg_duration = total_duration as f64 / durations.len() as f64;
    let max_duration = durations.iter().max().unwrap();
    //Die Kennzahlen ausgeben
```

```
    println!(
        "durchschnittliche Auftragsdauer: {}, max: {}, median: {}",
        t_to_str(avg_duration as i32),
        t_to_str(max_duration.clone()),
        t_to_str(median)
    );
}
```