Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2021	**Praxisaufgabe 1** zu – Systemprogrammierung – Prof. Dr. Odej Kao
Abgabetermin:	**13.06.2021 23:59 Uhr**

Aufgabe 1: theChat

Das *theChat* in Berlin Mitte erfreut sich großer Beliebtheit. Das Restaurant ist liebevoll eingerichtet und das Essen schmeckt hervorragend. Jedoch mangelt es dem Personal an organisatorischem Talent und Kunden müssen nicht selten unverhältnismäßig lange auf ihre Speisen warten. Deshalb werden Sie nun vom Inhaber dazu beauftragt, ein Programm zu entwickeln, dass die Zuteilung von Tischen und die zeitliche Planung von Gerichten übernimmt.

Ein Großteil der Aufgabe ist bereits vorgegeben. Laden Sie sich zunächst die Vorgabe von der ISIS-Seite herunter und machen Sie sich mit dem Code und der Code-Struktur vertraut. Im Ordner `src/` finden Sie die C-Dateien, die zur Bearbeitung der Aufgabe notwendig sind. In `include/` befinden sich die zugehörigen Header Dateien, `obj_src/` enthält bereits die kompilierten Objekt Dateien der Musterlösung. Außerdem enthalten ist ein Makefile welches Sie zum Kompilieren, Testen und zur Erzeugung der Abgabe verwenden sollen. Weitere Informationen finden Sie unten in den Hinweisen.

Mit dem Befehl `make musterloesung` wird die Musterlösung kompiliert und ausführbar gemacht.

Restaurantablauf: Eine Kundengruppe betritt das Restaurant und wird gesetzt, sobald ein Tisch frei ist. Warten mehrere Gruppen auf einen Tisch, stellen diese sich an. Nachdem eine Gruppe sich hingesetzt hat, braucht diese Zeit, um sich zu entscheiden. Wenn sie sich entschieden hat, soll bestellt werden. Wir gehen davon aus, dass eine Kundengruppe, die bestellen will, auch sofort bestellen kann.

Die Bestellung wird an die Küche übergeben, alle Bestellungen werden nach HRRN sortiert. Schnelle Gerichte werden schnell zubereitet, sodass möglichst viele Tische an einem Abend bedient werden. Dennoch sollen sehr aufwändige Gerichte nicht zu lange auf sich warten lassen. Sobald ein Koch zur Verfügung steht, bearbeitet dieser das Gericht. Dank seiner langjährigen Berufserfahrung, schafft er es immer alle Gerichte aus der Bestellung einer Gruppe in der Zeit des langsamsten Gerichts fertigzustellen.

Wenn ein Gericht fertig ist, wird es sofort serviert. Die Gäste benötigen nun eine bestimmte Zeit um zu essen.

Anschließend wird gezahlt und das Restaurant unmittelbar verlassen.

Es ergeben sich folgende relevante Zeitstempel:
- `NOT_ARRIVED` - `ARRIVED` - `SEATED` - `DECIDED` - `ORDERED` - `COOKING` - `SERVED` - `EATEN` - `PAYED`

Hinterlegt werden diese in dem `int[] stamps` und markieren jeweils den Zeitpunkt, an dem eine Kundengruppe in den dazugehörigen Zustand gewechselt ist.

a) **Text auslesen (2+2P)**

Informationen zum Verlauf des Abends werden dem Programm durch eine Eingabedatei übergeben. Ihre erste Aufgabe ist es, zwei Funktionen zu implementieren, damit diese Informationen in schon vorgegebene Datenstrukturen gespeichert werden können.

Zwei Beispieleingabedateien und eine `README`, die genau beschreibt, wie man diese interpretieren soll, finden Sie im Ordner `inputfiles/`.

`parseLine():` Implementieren Sie die `parseLine()` Funktion in der Datei `src/parseLine.c`. Die Funktion soll einzelne Zeilen der Eingabedatei in eine Instanz der `CustomerParty` Datenstruktur abspeichern. Dafür bekommt sie als Paramenter einen Pointer auf eine Zeile der Eingabedatei, einen Pointer auf eine `CustomerParty` und das Menü des Abends, was ein Array aus `meal` structs ist (siehe `src/main.c`). Gehen Sie davon aus, dass alle Parameter Pointer sind, die auf gültigen Speicher zeigen.

Lesen sie für diese Aufgabe am Besten die `README` zu den Eingabedateien und schauen sie sich den `CustomerParty` struct in `include/customerParty.h` an.

`readData():` Implementieren Sie nun die Funktion, die `parseLine()` aufruft. Diese finden Sie in der Datei `src/readData.c`. Als Parameter bekommt `readData()` den Pfad zur Eingabedatei, einen Pointer auf eine Liste und das Menü des Abends. Die Funktion soll nun für jede Zeile eine CustomerParty erstellen, diese durch einen Aufruf der `parseLine()` Funktion initialisieren und die `CustomerParty` in die Liste einfügen. Für letzteres könnt ihr die Funktion `listInsert()` aus der Vorgabe benutzen. Schließlich soll `readData()` als Rückgabewert die Anzahl an ausgelesenen Zeilen, bzw. Kundengruppen, zurückgeben.

Hinweise: Am Ende des `readData()` Aufrufs sollten alle member der `CustomerParty` structs initialisiert sein. Kundeninformationen, die sich nicht aus der Eingabedatei ergeben, initialisieren Sie bitte mit `0`. Ob Sie dies in `parseLine()` oder `readData()` tun, ist Ihnen überlassen.

Reservieren sie dynamisch Speicher in den `readData()` und `parseLine()` Funktionen so, dass der in der `list.c` auch wieder korrekt freigegeben wird. Dabei sei ausdrücklich darauf Hingewiesen dass nicht die `list.c` verändert werden soll! Diese ist nicht Teil der Abgabe!

b) **Kunden bedienen (3P)** In dieser Aufgabe soll nun die Funktionalität des Servicepersonals implementiert werden. Kunden mit der `stage == ARRIVED`, `DECIDED` oder `EATEN` benötigen aktive Betreuung. Die `updateTick()` Funktion ruft dafür die Funktion `serveCustomer()` auf. Sie finden diese in ihrem Grundgerüst in der

`serve.c` Datei. Innerhalb der Funktion wird bereits zwischen den verschiedenen stages unterschieden. Ihre Aufgabe ist es für die stage `ARRIVED` und `EATEN` die fehlende Funktionalität zu implementieren.

`stage == ARRIVED`: Alle Kunden die *theChat* betreten sind initial auf der Suche nach einem Tisch. Dieser muss ihnen zugewiesen werden. Tun Sie dies, indem Sie die Tische des Restaurants nach einem Tisch durchsuchen, welcher noch nicht auf eine `CustomerParty` verweist. Die Tischnummern sollen dabei aufsteigend vergeben werden. Hinterlegen sie außerdem die Tischnummer in dem dazugehörigen Feld des übergebenen `*customer`. Falls ein freier Tisch existiert geben sie `0` zurück, sonst `-1`.

`stage == DECIDED`: Dieser Fall muss nicht weiter bearbeitet werden. Da die Bestellung schon mit dem Einlesen der Datei bekannt ist, ist hier keine weitere Funktionalität nötig, außer die Rückgabe von `0`.

`stage == EATEN`: Gäste, die ihr Essen verspeist haben möchten gerne zahlen. Es liegt nun an Ihnen den Preis zu ermitteln. Der Grundpreis ergibt sich als Summe über alle Speisen, die diese Kundengruppe bestellt hat. Zuzüglich dazu hat *theChat* eine einheitliche Trinkgeld-Policy, welche sich nach der Wartezeit richtet. Da nicht jede Wartezeit vermieden werden kann, existiert eine gewisse Kulanz. Das Trinkgeld beträgt standardmäßig 20% vom vorher berechneten Preis. Für jeden Tick vermeidbare Wartezeit über 5 Ticks werden 0,5% abgezogen. Die Wartezeit W berechnet sich dabei folgendermaßen:

Sei $d_{h,a}$ die Differenz zwischen der Ankunftszeit und dem Hinsetzen der Gäste, $d_{b,w}$ die Differenz zwischen dem Bestellzeitpunkt und der Wahl des Essens (d.h. der Zeitpunkt, an dem die Gäste entschieden haben, was sie bestellen werden) und $d_{z,b}$ die Differenz zwischen der Zubereitungszeit und dem Bestellzeitpunkt.

$$W = d_{h,a} + d_{b,w} + d_{z,b}$$

Die Formel für den Trinkgeldsatz p lautet demnach:

$$p = 20 - \big(0,5 \cdot (W - 5)\big)$$

Der Trinkgeldsatz sollte jedoch niemals unter 0 fallen bzw. über 20 steigen. Für W größer 45 gibt es also einfach kein Trinkgeld mehr. Zur besseren Übersicht für die Monatsabrechnung hinterlegen sie den Preis und das Trinkgeld in € getrennt, im jeweiligen Feld des `CustomerParty` struct und addieren es auch im Rahmen dieser Funktion auf die bisherigen Umsätze und Trinkgelder des Restaurants auf. Anschließend kann der Tisch geräumt werden indem in `res->tables[]` der dazugehörige Eintrag auf `NULL` gesetzt wird. Abschließend muss noch 0 an die aufrufende Funktion zurückgegeben werden.

Achten sie bei der Trinkgeldberechnung darauf, das `int` keine Nachkommastellen unterstützt.

Hinweis: Zur Berechnung der Wartezeit bieten sich die entsprechenden Zeitstempel der `CustomerParty` an.

c) **Scheduling (3P)**

Mit zunehmender Beliebtheit steigen die Anforderungen an die Küche ihres Restaurants. Dafür gilt es nun bestellte Mahlzeiten für die Zubereitung zu schedulen. Sie finden dafür zwei Funktionen in `kitchen.c`.

`getPrepTime()`: Diese Funktion bekommt ein `meal` Array in der Größe `orderSize` übergeben. Dies entspricht der Bestellung einer Kundengruppe. Ihre Köche bringen einige Erfahrung mit und benötigen für die Spesenzubereitung einer ganzen Gruppe nur so lange, wie das längste Gericht der Bestellung benötigt. Ermitteln Sie die Zubereitungszeit und geben sie den Wert zurück.

`cooking_queue_next_HRRN()`: Alle Bestellungen die noch nicht zuebreitet werden, sollen nach Highest Response Ratio Next geplant werden. Innerhalb dieser Funktion müssen alle Kundengruppen durchlaufen und für alle Gruppen, welche bereits bestellt haben, aber deren Bestellung noch nicht zubereitet wird, die Response Ratio berechnet werden. Die Response Ratio wird aus dem Verhältnis zwischen (Wartezeit seit Bestellung + Zubereitungszeit) und der Zubereitungszeit berechnet. Die Funktion gibt einen Pointer auf die Kundengruppe mit der höchsten Response Ratio zurück, oder `NULL`, falls gerade keine unbearbeiteten Bestellungen existieren. Bei Gleichstand sollte die Kundengruppe gewählt werden, welche das Lokal zuerst betreten hat.

Hinweise:

- **Make** Um ein Projekt in C zu kompilieren verwendet man C-Compiler wie GCC. Diese werden klassisch über die Konsole ausgeführt (z.B. `gcc foo.c bar.c -o foobar` um aus den beiden Programmtexten `foo.c` und `bar.c` die ausführbare Datei `foobar` zu erzeugen). Gerade bei größeren Projekten mit vielen Programmtexten ist es umständlich solche langen Befehle immer wieder erneut eingeben zu müssen. Das Programm *make* schafft hier Abhilfe.

  Make nutzt die im Makefile der Vorgabe bereits enthaltenen targets um das Projekt entsprechend zu kompilieren. `make theChat` kompiliert dabei das ganze Projekt (`*.c` in `src`), `make musterloesung` die Musterlösung (`*.o` in `obj_lsg`).

- **Ausführen** Das Programm erwartet den Pfad zur Eingabedatei als Parameter. Im Ordner `inputFiles/` sind schon zwei Beispieleingabedateien vorgegeben. Zum Testen ihrer Implementierung können Sie gerne auch weitere Eingabedateien erstellen.

- **Testen** Mit den targets `test_readData`, `test_parseLine`, `test_kitchen`, `test_serve` des Makefiles können die Teilaufgaben unabhängig voneinander getestet werden. Dabei wird die jeweilige `.c` Datei zusammen mit der Musterlösung kompiliert. **Achtung:** diese Tests ersetzen keinen vollständigen Funktionstest!

- **Abgabe** Die Abgabe erfolgt allein mit Hilfe des Befehls `make submission`. Dieser verschnürt die Code-Dateien `serve.c`, `kitchen.c`, `readData.c`, und `parseLine.c` zu einem ZIP-Archiv, welches bei ISIS hochzuladen ist. Es kann beliebig oft abgegeben werden, bewertet wird nur die letzte Abgabe. Die Abgabe muss mit dem Makefile der Vorgabe kompilierbar sein. Abzug gibt es für Compilerwarnings und Speicherfehler. Außerdem führen wir Plagiatsprüfungen durch!

- **Referenzsystem** Da der Zugang zu den Universitätsservern über ssh bzw. vpn immer noch eingeschränkt ist, verwenden wir als Referenzsystem Ubuntu 20.04. Abgaben, die unter Windows bzw. Mac funktionieren, tun dies u.U. nicht auf Linux Systemen!