Betriebsysteme - Praktikum 4

Aufgabe 1
Simulieren Sie in C/C++ mithilfe von Threads, Mutex und Semaphoren unter Linux das folgende Problem (Burger-Imbiss).

Siehe den Quellcode.

Aufgabe 1-1
Eine beim Programmaufruf über die Kommandozeile festzulegende Anzahl Mitarbeiter m verkauft Burger in einem Imbiss.

Siehe Bild 1.

```
int main(int argc, char *argv[]) {
    if (argc != 6) {
        cerr << "Not all arguments provided, check your input! \n";
        return 1;
    }

    numberOfWorkers = std::stoi(argv[1]);
    ingredientContainerCapacity = std::stoi(argv[2]);
    frequency = std::stoi(argv[3]);
    numberOfCustomers = std::stoi(argv[4]);
    preparationTime = std::chrono::microseconds(std::stoll(argv[5]));
```

Bild 1, Implementierung der Eingabe von Imbiss-Argumenten

Aufgabe 1-2
Ein Burger besteht aus den folgenden Zutaten: Fleisch, Salat, Gurke, Brötchen. Der Einfachheit halber wird jeweils 1 Einheit jeder Zutat benötigt. Die Zutaten befinden sich in eigenen Behältern mit Fassungsvermögen n. Die Behälter werden mit fester Rate von r Einheiten pro Sekunde von einem separaten Thread aufgefüllt (nur falls sie nicht mehr voll sind). Zu Beginn soll jeder Behälter voll sein.

Zutaten sind durch integer Werten simuliert. Maximal Wert jedes Wertes ist durch integer Variable „ingredientContainerCapacity“ simuliert (Siehe Bilder 1 und 2).

```
amountOfPatties = ingredientContainerCapacity;
piecesOfSalad = ingredientContainerCapacity;
slicesOfCucumber = ingredientContainerCapacity;
amountOfBuns = ingredientContainerCapacity;
```

Bild 2, Festlegen der Zutatenmenge

Auffüllen von Zutaten wird mithilfe von Thread/Funktion „fillingMachine“/„fillIngredients“ (Siehe Bild 3) durchgeführt. Um gewünschte Frequenz von Auffüllen zu implementieren, wird diese am Anfang der Funktion „fillIngredients“ in die Zeit in Mikrosekunden umgewandelt (Siehe Bild 4). Diese Zeit wird benutzt um Pausen zwischen einzelnes Auffüllen zu simulieren (Siehe Bild 5).

```
std::thread fillingMachine (fillIngredients);
```

Bild 3, Thread für Simulation des Auffüllens von Zutaten

```
void fillIngredients() {
    std::chrono::microseconds repetition = std::chrono::microseconds(1000000 / frequency);
```

Bild 4, Frequenz Umwandlung

```
std::this_thread::sleep_for(repetition);
```

Bild 5, Thread Schläft, um Pausen zwischen Auffüllen zu simulieren

Aufgabe 1-3

Eine Menge von k Kunden reiht sich in eine Warteschlange ein und bestellt beim nächsten freien Mitarbeiter eine zufällig gewählte Menge Burgern. Die Menge soll von jedem Kunden einzeln zwischen 1 und 10 zufällig gewählt werden.

```
vector <std::thread> customers;
for (int j = 1; j <= numberOfCustomers; j++)
    customers.push_back(std::thread (placeOrder, j));
```

Bild 6, Erstellen von Customer Threads

```
void placeOrder(int customerId) {
    sem_t* customerSemaphore = new sem_t;
    sem_init(customerSemaphore, 0, 0);

    sem_wait(&orderCapacity);

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dist(1, 10);

    int orderQuantity = dist(gen);

    ordersAccess.lock();
    orders.push(std::make_pair(std::make_pair(customerId, customerSemaphore),orderQuantity));
    ordersAccess.unlock();

    sem_post(&orderPlaced);

    notificationsMutex.lock();
    cout << "Customer " << customerId << " has placed an order of " << orderQuantity << " burgers \n";
    notificationsMutex.unlock();

    sem_wait(customerSemaphore);

    notificationsMutex.lock();
    cout << "Customer " << customerId << " has taken his order and left \n";
    notificationsMutex.unlock();

}
```

Bild 7, Funktion „placeOrder“

Customers-Threads werden in „main“ erstellt und werden die Funktion „placeOrder“ nutzen (Siehe Bild 6) um sich in Warteschlange zu stellen und Burgers zu bestellen. Semaphore „orderCapacity“ wird benutzt, um festzustellen, dass beim Anlegen einer Bestellung, mindestens ein Mitarbeiter frei ist (Siehe Bild 7). Queue „orders“ wird benutzt um Warteschlange zu simulieren (Siehe Bilder 7 und 8).

```
queue <std::pair<std::pair<int,sem_t*>,int>> orders; // customer id, semaphore, order quantity
```

Bild 8, Bestellungen (FIFO) Container

Aufgabe 1-4-1

Nach einer Bestellung nimmt ein Mitarbeiter aus jedem Zutatenbehälter nacheinander die benötigte Menge an Zutaten und stellt damit in Zeit t die bestellte Menge an Burgern fertig (um die dafür nötige Zeit t zu simulieren, können Sie den Systemaufruf usleep verwenden) und übergibt sie dem Kunden, der auch erst daraufhin die Warteschlange verlässt. Falls einer der Behälter leer ist, wartet der Mitarbeiter, bis dieser aufgefüllt wurde.

```
for (int k = 0; k < ordQuantity; k++) {

    pattiesAccess.lock();
    if (amountOfPatties == 0) {
        pattiesAccess.unlock();
        sem_wait(&pattiesAvailiability);
        pattiesAccess.lock();
        amountOfPatties--;
        pattiesAccess.unlock();
    } else {
        amountOfPatties--;
        sem_wait(&pattiesAvailiability);
        pattiesAccess.unlock();
    }
```

Bild 9, Prinzip Zugriffs auf Zutaten aus Sicht Mitarbeiters (Teil der Funktion „processOrder“)

Erstmal wird Mitarbeiter-Thread mithilfe von Mutex, den Zugriff auf Zutaten „schließen“ bzw. sichern (Siehe Bild 9). Falls Zutatenbehälter leer ist, wird Zugriff frei gemacht und geeignet Semaphor verwendet um auf Signal von Auffüllen-Thread zu warten, dass Zutaten wieder verfügbar sind. Außerdem, sichert Semaphor die Vermeidung von „Bussy-Waiting“. Falls Zutatenbehälter nicht leer sind, wird entsprechende Variable zusammen mit Semaphoren dekrementiert und Zugriff auf Zutaten wieder „aufgeschlossen“. Nachdem Mitarbeiter alle Zutaten die für ein Burger nötig sind entsprechend dekrementiert, wird Thread schlafen (Siehe Bild 10) und danach nächste Burger erstellen, bis alle Burger in der Bestellung nicht fertig sind.

```
std::this_thread::sleep_for(preparationTime);
```

Bild 10, Simulation der Vorbereitung Burgers

Aufgabe 1-4-2

Implementieren Sie das Problem Imbiss mithilfe von Threads, so dass die o.a. Parameter m, n, r, k und t über die Kommandozeile eingegeben werden können (also in der Art: "./imbiss <m> <n> <r> <k> <t>"). Dabei soll jeder Mitarbeiter sowie jeder Kunde jeweils von einem eigenen Thread simuliert werden. Zu Beginn der Simulation sind die Mitarbeiter ebenso wie der Thread zum Befüllen der Zutatenbehälter schon da, die Kunden werden nach und nach erstellt.

Siehe Bilder 1 und 2 und 11.

```
vector <std::thread> workers;
for(int i = 1; i <= numberOfWorkers; i++)
    workers.push_back(std::thread (processOrder, i));

vector <std::thread> customers;
for (int j = 1; j <= numberOfCustomers; j++)
    customers.push_back(std::thread (placeOrder, j));

std::thread fillingMachine (fillIngredients);

for (int k = 0; k < workers.size(); k++)
    workers.at(k).join();

for (int t = 0; t < customers.size(); t++)
    customers.at(t).join();

fillingMachine.join();
```

Bild 11, Threads erstellt in „main“

Aufgabe 1-4-3

Nutzen Sie Mutex und/oder Semaphor geeignet, um Behälter thread-safe zu befüllen bzw. Zutaten daraus zu entnehmen. Ebenso muss die Datenstruktur für die Warteschlange (FIFO) thread-safe implementiert werden. Achten Sie darauf, dass die Kunden so lange warten, bis die bestellte Anzahl Burger fertig ist und sie auch von genau dem Mitarbeiter die bestellten Burger entgegennehmen, der die Bestellung aufgenommen hat (Hinweis: dafür darf kein busy-waiting verwendet werden, stattdessen nutzen Sie Semaphore geeignet!). Nach Abarbeitung der Bestellung sollen sich die Kundenthreads beenden.

Thread-Safe Zugriff auf „shared-ressources“ wurde mithilfe von Mutex implementiert, siehe Bild 9 und Beschreibung der Lösung von Aufgabe 1-4-1, oder Bild 7 und Zugriff auf Queue „orders“, da wird es mithilfe von Mutex gesichert, dass nur ein Thread in dem gegebenen Moment, Zugriff auf Queue hat.

Datenstruktur für Warteschlange (FIFO) wird mithilfe von Queue und Logik in den Funktionen „placeOrder“ und „processOrder“ implementiert (Siehe Bilder 7, 12 und 13).

```
void processOrder(int workerId) {
    while (true) {
        sem_wait(&orderPlaced);
        ordersAccess.lock();

        if (orders.empty()) {
            ordersAccess.unlock();
            sem_post(&orderPlaced);
            break;
        }

        int custId = orders.front().first.first;
        int ordQuantity = orders.front().second;
        sem_t* custSemaphore = orders.front().first.second;
```

Bild 12, FIFO Logik-Implementierung („processOrder“), Teil 1

```
sem_post(custSemaphore);
numberOfServedCustomers++;
orders.pop();
amountOfBurgersMade = amountOfBurgersMade + ordQuantity;
ordersAccess.unlock();
sem_post(&orderCapacity);
if (numberOfServedCustomers == numberOfCustomers) {
    sem_post(&orderPlaced);
    break;
```

Bild 13, FIFO Logik-Implementierung („processOrder"), Teil 2

Zuerst stellt Customer die Bestellung in Queue Orders rein und übergibt Signal über Semaphore „orderPlaced", dass eine Bestellung reingelegt ist (Siehe Bild 7). Wenn eine Bestellung reingelegt ist, wird diese über „orderPlaced" Semaphore signalisiert, und ein Worker thread, wird auf Queue Zugriffen (Falls Zugriff nicht geschlossen ist, in dem Fall wird Thread warten müssen) (Siehe Bild 12). Wenn Worker-Thread Zugriff auf Queue bekommt, übernimmt er die erste Bestellung (älteste) und liest Daten daraus. Nachdem Worker-Thread die Bestellung fertig vorbereitet hat, löscht er diese Bestellung aus der Queue (Siehe Bild 13).

Damit Kunden warten, nur so lang, bis ihre Bestellung bereit ist, und damit es dabei keine „Bussy-Waiting" stattfindet, wird Semaphore „customerSemaphore" benutzt (Siehe Bild 7). Queue orders, enthält Bestellungen, jede Bestellung trägt drei Informationen: „customerId", „customerSemaphore" und „orderQuantity". Wenn ein Worker-Thread, eine Bestellung vorbereitet, er merkt sich Customer-Semaphore daraus, und wenn Bestellung fertig ist, signalisiert Worker-Thread dem entsprechenden Customer-Thread über seinen Semaphoren, dass seine Bestellung fertig ist, und dass er gehen kann (Siehe Bilder 7, 12 und 13).

Mechanismus, der es sichert, dass der Worker-Thread, der die Bestellung aufgenommen hat, der Worker-Thread ist, der die Bestellung vorbereitet, ist mithilfe von „ordersAccess" Mutex implementiert. Wenn ein Worker-Thread eine Bestellung vorbereitet, ist Zugriff allen anderen Threads auf Bestellungen-Queue sowie der Vorbereitung nicht erlaubt (Siehe Bilder 12 und 13).

Aufgabe 1-4-4

Messen Sie den Durchsatz an Burgern (Einheit Burger/Sekunde) und dokumentieren Sie dies. Setzen Sie dabei die Parameter n=20, r=100, k=50 fest und für m jeweils 1, 2, 5, 10, 20 sowie für t jeweils 1000, 10000, 100000 Mikrosekunden (Hinweis: usleep arbeitet mit Mikrosekunden). Hinweis: Dies kann man z.B. gut mit einem Skript erledigen, das ist jedoch nicht erforderlich.

Messen wurde mithilfe Skripts „imbissScript.sh" durchgeführt. Skript befindet sich in File „bs_p4" der Quellcode.

		Mitarbeiteranzahl				
		1	2	5	10	20
Vorbereitungszeit (Mikrosekunden / Ein Burger)	1000	118,5	107	137,5	103,3	144,5
	10000	95	134,5	130,5	102,7	129
	100000	10,21	10	10,11	10	9,93

Tabelle 1, Burger/Sekunde - Verhältnis im Zusammenhang mit Anzahl der Mitarbeiter und Vorbereitungszeit in Mikrosekunden/Ein Burger

```
dmaks@Ubuntu-22:~/HDA/BS/p4/bs_p4$ ./imbissScript.sh
Number of workers: 1   || Preparation time: 1000ms
237 Burger / 2 Seconds || Ratio of: 118.5

Number of workers: 1   || Preparation time: 10000ms
285 Burger / 3 Seconds || Ratio of: 95

Number of workers: 1   || Preparation time: 100000ms
296 Burger / 29 Seconds || Ratio of: 10.2069

Number of workers: 2   || Preparation time: 1000ms
321 Burger / 3 Seconds || Ratio of: 107

Number of workers: 2   || Preparation time: 10000ms
269 Burger / 2 Seconds || Ratio of: 134.5

Number of workers: 2   || Preparation time: 100000ms
270 Burger / 27 Seconds || Ratio of: 10

Number of workers: 5   || Preparation time: 1000ms
275 Burger / 2 Seconds || Ratio of: 137.5

Number of workers: 5   || Preparation time: 10000ms
261 Burger / 2 Seconds || Ratio of: 130.5

Number of workers: 5   || Preparation time: 100000ms
283 Burger / 28 Seconds || Ratio of: 10.1071

Number of workers: 10   || Preparation time: 1000ms
310 Burger / 3 Seconds || Ratio of: 103.333

Number of workers: 10   || Preparation time: 10000ms
308 Burger / 3 Seconds || Ratio of: 102.667

Number of workers: 10   || Preparation time: 100000ms
270 Burger / 27 Seconds || Ratio of: 10

Number of workers: 20   || Preparation time: 1000ms
289 Burger / 2 Seconds || Ratio of: 144.5

Number of workers: 20   || Preparation time: 10000ms
258 Burger / 2 Seconds || Ratio of: 129

Number of workers: 20   || Preparation time: 100000ms
288 Burger / 29 Seconds || Ratio of: 9.93103
```

Screenshot 1, Ausgabe Programms beim Ausführen Skripts