Patrick Felschen, Matr. Nr. 932056

Julian Voß, Matr. Nr. 934505

Aufgabenblatt 6

Implementierung

Starten der Threads

```
//// Reader Thread
std::thread pro(producer, &queue, fileName);

//// Client Threads
for (int i = 0; i < threadCount; i++) {
    threads[i] = std::thread(consumer, &queue, i);
}
pro.join();
for (int i = 0; i < threadCount; i++) {
    threads[i].join();
}
```

Über die Thread Klasse lassen sich in C++ Threads anlegen. Diese bekommen als Parameter eine auszuführende Funktion (void*), des Weiteren können Parameter für die auszuführende Funktion angegeben werden.
Da die Anzahl der Webrequest-Threads dynamisch ist, werden die Threads in einer Schleife angelegt und in einem Array abgespeichert.
Zum Schluss werden die Threads über die *join*-Funktion synchronisiert.

Producer

```
void *producer(void *q, char *fileName) {
    Queue *queue = (Queue *) q;
    const std::regex pattern(URLPATTERN);
    std::ifstream file(fileName);
    std::string str;
    while (std::getline(file, str)) {
        char *cstr = new char[str.length() + 1];
        strcpy(cstr, str.c_str());
        if (!std::regex_match(cstr, pattern)) {
            continue;
        }
        std::unique_lock<std::mutex> lock(queue->mutex);
        while (queue->isFull()) {
            queue->notFull.wait(lock);
        }
        queue->addItem(cstr);
        queue->notEmpty.notify_all();
    }
    queue->setEnd(true);
    file.close();
    return nullptr;
}
```

Zur internen Verwaltung der Daten wird eine Queue verwendet (FIFO-Prinzip). Aus einer Steuerdatei (Parameter fileName) werden zeilenweise URLs ausgelesen und auf ein richtiges Format geprüft (regex). Ist es eine valide URL wird der Mutex der Queue gesperrt. Solange die Queue nicht voll ist, werden die URLs zur Queue hinzugefügt. Alle wartenden Threads werden über eine Condition Variable über den Zustand der Queue (nicht leer) benachrichtigt. Wenn die Queue voll ist, wird die Mutex-Sperre aufgehoben und solange gewartet, bis die Queue nicht mehr voll ist.

Consumer

```
void *consumer(void *q, int id) {
    Queue *queue = (Queue *) q;
    while (true) {
        std::unique_lock<std::mutex> lock(queue->mutex);
        while (queue->isEmpty()) {
            if (queue->isEnd()) {
                return nullptr;
            }
            queue->notEmpty.wait(lock);
        }
        char *url;
        queue->delItem(&url);
        lock.unlock();
        do_consume(url, id);
        queue->notFull.notify_one();
    }
}
```

Der Consumer startet in einer Schleife, die beendet wird, wenn die Steuerdatei vollständig ausgelesen ist und die Queue leer ist. Ist die Queue leer, die Steuerdatei aber noch nicht vollständig ausgelesen, wird gewartet, bis die Queue nicht mehr leer ist. Wenn keiner der Fälle eintritt, wird die älteste URL aus der Queue ausgelesen und entfernt. Da die Queue nicht mehr benutzt wird, kann die Sperre in diesem Thread aufgehoben werden. Die Website wird über ein GET-Request mit der URL

abgerufen und in einer .html Datei abgespeichert. Da ein Element entfernt wurde, wird der Producer-Thread über den Zustand (nicht voll) benachrichtigt.

Tests

Queuegröße: 2
Threads: 20
Beobachtung: Consumer-Threads hängen überwiegen in der „isEmpty“-Schleife

Queuegröße: 5
Threads: 1
Beobachtung: Producer-Thread hängt überwiegend nach erstmaligen Füllen der Queue in der „isFull“-Schleife.

Queuegröße: 10
Threads: 10
Beobachtung: Bei gleicher Queuegröße und Threadanzahl ist die Bearbeitung sehr schnell

Queuegröße: 50
Threads: 2
Beobachtung: Queue wird schnell gefüllt, Consumer-Threads können diese ungestört abarbeiten.

Laufzeiten

Queuegröße: 10

	Proxy Verzögerung [ms]			
#Threads	**0**	**300**	**500**	**1000**
1	9418	18234	24696	39646
2	4851	9090	12874	18912
5	2262	3999	5390	9232
10	1480	2421	3034	4336
15	1446	2036	2132	3042
20	1418	2023	2031	2944