Assignment 1 report

Jesper Wingren
Jw223rn@student.lnu.se
1dt909

Innehållsförteckning

1. Dequeue

1.1 Setup

The dequeue is setup so that the head and tail will always be fixed values but when handling the values it in a way ignores these so for example addFront doesn't make it the head it adds it to the front but inside the head and tail. The find method is used to find the back of the dequeue making it possible to add and pop from the back.

1.2 Locking

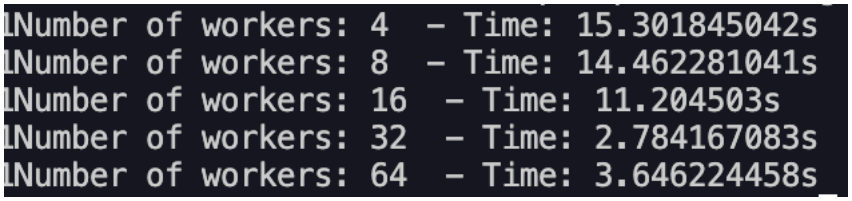
My dequeue implementation uses fine-grained locking by each node having its own mutex lock. In the code it only locks the node or nodes it Is currently handling for example the addFront it only locks the head node and not the whole code making it possible to concurrently handle other methods. This fine-grain locking also reduces the chances of methods concurring each other and avoiding deadlocks. By using a find method for the back method, the program only locks the two nodes its currently watching meaning in a longer dequeue it can use find concurrently because it's at different spots in the queue, hence not locking each other out.

2. Password-cracker

2.1 Setup

We begin by creating two channels, one for passwords to check and one for the found password. We set up a wait group and set the number of workers. The generate password method then creates that set number of workers to start creating passwords. The method then starts each worker with its part of the charset it is supposed to work on. Then using the helper method each worker uses its own part of the charset to create password while sending it to the channel called passwords. We then create a new wait group for the hashWorkers as I call them, these workers check through the password channel and hash the passwords to check if they match the targethash. I then create a go func that waits until the hashWorkers has found a person and then I check through the found channel and print the password that matches the targetHash.

2.2 Results



My result is the above times. Because my cracker works by splitting the charset to each worker it will not be any faster than the length of my charset. That is why 64 workers doesn't decrease the time but instead it creates a form of overhead making it slower. It scales very good with the amount of workers meaning it gets significantly faster when increasing to 32 workers from 4.

3. Counting semaphore

3.1 Setup

My counting semaphore is setup using 4 properties in a struct, it has a mutex lock a condition a count and a maxcount. The maxcount is the amount of goroutines allowed to work at the same time while count keeps that count. I initialize the semaphore by setting maxcount to N(a given number) and count to 0. The condition is set as a pointer to the mutex locks. We then setup number of workers and call them using goroutines while printing out which current workerid is in the critical section(allowed to do something). The acquire method works by first locking the semaphore and if the current count(number of workers in critical region) exceeds the max count it calls the wait method waiting until its signaled that its ok to enter the critical region and when that happens count is decremented. The release method first locks the semaphore then signals the condition to wake up and the increments counts.

3.2 Results

```
Goroutine 4 in critical section
Goroutine 32 in critical section
Goroutine 10 in critical section
Goroutine 33 in critical section
Goroutine 40 in critical section
Goroutine 41 in critical section
Goroutine 34 in critical section
Goroutine 42 in critical section
Goroutine 35 in critical section
Goroutine 43 in critical section
Goroutine 44 in critical section
Goroutine 36 in critical section
Goroutine 45 in critical section
Goroutine 37 in critical section
Goroutine 46 in critical section
Goroutine 38 in critical section
Goroutine 18 in critical section
Goroutine 16 in critical section
Goroutine 17 in critical section
Goroutine 59 in critical section
Goroutine 39 in critical section
Goroutine 20 in critical section
Goroutine 56 in critical section
Goroutine 19 in critical section
Goroutine 23 in critical section
Goroutine 57 in critical section
Goroutine 61 in critical section
Goroutine 58 in critical section
Goroutine 60 in critical section
Goroutine 0 in critical section
Goroutine 62 in critical section
Goroutine 15 in critical section
```

Here is the last part of the output where you can see which goroutine currently in the critical regions.