1242. Web Crawler Multithreaded

Problem Explanation

Our task is to build a web crawler in any programming language, using the provided HtmlParser interface. We're supposed to build a multi-threaded web crawler that can crawl through all links under the same hostname as the `startUrl`. By multi-threaded, it means that we need to design a solution that can work on multiple threads simultaneously and fetch the pages, rather than fetching one by one.

We're given some constraints on what hostname can be, which is an important clue about the problem. In our input, we have multiple urls, and a query/url to start with.

Here is a step-by-step visual explanation of how to approach this problem:

Let's consider this simple example input:

urls = ["http://news.yahoo.com", "http://news.yahoo.com/news", "http://news.google.com"]

startUrl = "http://news.yahoo.com"

First, our crawler starts from startUrl which is "http://news.yahoo.com" and fetches all its urls using HtmlParser.getUrls(url). If there are more urls under the same host, it will push them in the queue for further crawling. So the output will be: "news.yahoo.com" "news.yahoo.com/news"

Then, the crawler, in parallel will consider next url in queue and fetches its urls. As there is no url, it will not add anything to the queue.

In the end, our final output will be ["news.yahoo.com", "news.yahoo.com/news"]

The problem can be solved by implementing a Breadth-First Search (BFS) algorithm running concurrently in different threads. At the start, spin up the threads, then use BFS on each thread. We ensure that we don't visit the same URL twice by keeping track of URLs in a visited set.

Let's look at the solutions for different languages.

C## Solution

```
csharp
public class Solution {
    public IList<string> Crawl(string startUrl, HtmlParser htmlParser) {
        HashSet<string> ans = new HashSet<string>();
        string PREFIX = startUrl.Split("/")[2];
        Queue<string> queue = new Queue<string>();
        queue.Enqueue(startUrl);
        ans.Add(startUrl);
        Parallel.ForEach(Enumerable.Range(0, Environment.ProcessorCount * 2), _ => {
            while (true) {
                string current = null;
                lock (queue) {
                    if (queue.Count == 0) return;
                    current = queue.Dequeue();
                }
                foreach (var nextUrl in htmlParser.GetUrls(current)) {
                    if (!nextUrl.Split("/")[2].Equals(PREFIX) || ans.Contains(nextUrl)) {
                        continue;
                    };
                    lock(ans) {
                        ans.Add(nextUrl);
                    }
                    lock(queue) {
                        queue.Enqueue(nextUrl);
                    }
                };
            }
        });
        return ans.ToList();
    }
}
```

Python Solution

```
python
class Solution:
    def crawl(self, startUrl: str, htmlParser: 'HtmlParser') -> List[str]:
        import threading
        visited = set()
        visited_lock = threading.Lock()
        crawl = [False]
        crawl_lock = threading.Condition()

        def worker(url, htmlParser):
            nonlocal crawl
            while True:
                my_html = None
                with visited_lock:
                    for html in htmlParser.getUrls(url):
                        hostname = html.split('/')[2]
                        if hostname == startUrl.split('/')[2] and html not in visited:
                            visited.add(html)
                            my_html = html
                            break
                if my_html != None:
                    worker(my_html, htmlParser)
                else:
                    with crawl_lock:
                        crawl[0] = crawl[0] - 1
                        if crawl[0] == 0:
                            crawl_lock.notify_all()
                        crawl_lock.wait()

        with crawl_lock:
            with visited_lock:
                visited.add(startUrl)
            crawl[0] = threading.active_count()
            threading.Thread(target=worker, args=[startUrl, htmlParser]).start()
            crawl_lock.wait()
        return list(visited)
```

Java Solution

```
java
class Solution {
    public List<String> crawl(String startUrl, HtmlParser htmlParser) {
        Set<String> result = ConcurrentHashMap.newKeySet();
        String hostname = getHostname(startUrl);

        ExecutorService executor = Executors.newFixedThreadPool(64);
        result.add(startUrl);
        crawl(result, startUrl, hostname, executor, htmlParser);
        executor.shutdown();

        try {
            executor.awaitTermination(10, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        return new ArrayList<>(result);
    }

    private String getHostname(String url) {
        int idx = url.indexOf('/', 7);
        return (idx != -1) ? url.substring(0, idx) : url;
    }

    private void crawl(Set<String> result, String start, String hostname, ExecutorService executor, HtmlParser htmlPa
        List<Future> futures = new ArrayList<>();
        for (String url : htmlParser.getUrls(start)) {
            if (url.startsWith(hostname) && result.add(url)) {
                futures.add(executor.submit(() -> crawl(result, url, hostname, executor,htmlParser)));
            }
        }
        for (Future f : futures) {
            try {
                f.get();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

JavaScript Solution

```
javascript
// Import `HtmlParser` interface

// Initialize list, host, visited, and queue
let list = htmlParser.getUrls(startUrl)
let host = startUrl.split("/")[2];
let visited = new Set([startUrl])
let queue = [startUrl]


// Crawler worker
const worker=(url)=>{
    // Check if URL has been visited
    if (visited.has(url)){
        return
    }
    // Check if URL is same host
    if(url.indexOf(`http://${host}`)!== 0){
        return
    }
    // Add to visited
    visited.add(url);
    let urls = htmlParser.getUrls(url)
    urls.forEach(worker)
}

// Iterate over each list item
for (let i=0; i<list.length;i++){
    worker(list[i])
}

// Return final result from visited set
return [...visited]
```

Please note that JavaScript does not support native multi-threading (although worker threads are available in Node.js). This is a single-threaded solution but it explains the basic idea.

C++ Solution

```
cpp
class Solution {
public:
    vector<string> crawl(string startUrl, HtmlParser htmlParser) {
        queue<string> q{{startUrl}};
        unordered_set<string> seen{{startUrl}};
        string hostname = getHostname(startUrl);

        vector<thread> threads;
        mutex mtx;
        condition_variable cv;

        auto worker = [&]() {
            while (true) {
                unique_lock<mutex> lock(mtx);
                cv.wait_for(lock, 30ms, [&]() { return q.size(); });

                if (q.empty())
                    return;

                auto url = q.front(); q.pop();
                auto urls = htmlParser.getUrls(url);

                lock.unlock();

                for (const auto& url : urls) {
                    if (url.find(hostname) != string::npos) {
                        lock_guard<mutex> lock(mtx);
                        if (seen.insert(url).second)
                            q.push(url);
                    }
                }

                lock.lock();
                cv.notify_all();
            }
        };

        for (int i = 0; i < thread::hardware_concurrency(); ++i)
            threads.emplace_back(worker);

        for (auto& t : threads) t.join();

        return {seen.begin(), seen.end()};
    }
private:
    string getHostname(const string& url) {
        return url.substr(0, url.find('/', 7));
    }
};
```

These solutions effectively use multithreading to provide a fast and parallelized web crawler. With these implementations, you can efficiently crawl millions of web pages under the same hostname without the need to wait for a single page to finish loading before moving onto the next one.This C++ solution uses a similar approach to the solutions above, however, it adds in condition variables and mutexes to allow for more efficient thread synchronization. The queue data structure is used to store the urls that need to be visited and once a thread is done with its task, it waits (cv.wait_for(...)) for more urls to be added to the queue. If the queue is not empty, it pops a url and continues its task. If all the urls have been visited, the thread will hang indefinitely (if no timeout was specified), therefore a timer (30ms) is added to automatically exit the thread when its inactive for a certain period.

This setup allows for a dynamic number of threads to be created, based on the hardware used (thread::hardware_concurrency()). This results in utilizing the maximum potential of the machine by using the maximum possible threads that the hardware supports. Therefore you will get different number of threads for different machines (high-end server vs home PC).

Every thread runs the same worker function, and to ensure that the shared queue and set (containing the URLs) are not corrupted by multiple threads accessing them simultaneously, mutex locks are used over the operations which modify them.