<https://leetcode.libaoj.in/web-crawler-multithreaded.html>

## Web Crawler Multithreaded

### Description

Given a url startUrl and an interface HtmlParser, implement **a Multi-threaded web crawler** to crawl all links that are under the **same hostname** as startUrl.

Return all urls obtained by your web crawler in **any** order.

Your crawler should:

* Start from the page: startUrl
* Call HtmlParser.getUrls(url) to get all urls from a webpage of given url.
* Do not crawl the same link twice.
* Explore only the links that are under the **same hostname** as startUrl.



As shown in the example url above, the hostname is example.org. For simplicity sake, you may assume all urls use **http protocol** without any **port** specified. For example, the urls http://leetcode.com/problems and http://leetcode.com/contest are under the same hostname, while urls http://example.org/test and http://example.com/abc are not under the same hostname.

The HtmlParser interface is defined as such:

interface HtmlParser {

// Return a list of all urls from a webpage of given url.

// This is a blocking call, that means it will do HTTP request and return when this request is finished.

public List<String> getUrls(String url);

}

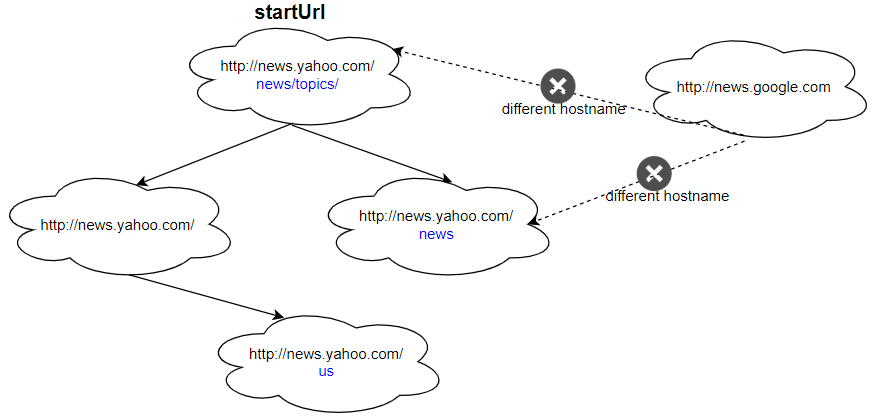
Note that getUrls(String url) simulates performing a HTTP request. You can treat it as a blocking function call which waits for a HTTP request to finish. It is guaranteed that getUrls(String url) will return the urls within **15ms.** Single-threaded solutions will exceed the time limit so, can your multi-threaded web crawler do better?

Below are two examples explaining the functionality of the problem, for custom testing purposes you'll have three variables urls, edges and startUrl. Notice that you will only have access to startUrl in your code, while urls and edges are not directly accessible to you in code.

**Follow up:**

1. Assume we have 10,000 nodes and 1 billion URLs to crawl. We will deploy the same software onto each node. The software can know about all the nodes. We have to minimize communication between machines and make sure each node does equal amount of work. How would your web crawler design change?
2. What if one node fails or does not work?
3. How do you know when the crawler is done?

**Example 1:**



**Input:**

urls = [

  "http://news.yahoo.com",

  "http://news.yahoo.com/news",

  "http://news.yahoo.com/news/topics/",

  "http://news.google.com",

  "http://news.yahoo.com/us"

]

edges = [[2,0],[2,1],[3,2],[3,1],[0,4]]

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

**Output:** [

  "http://news.yahoo.com",

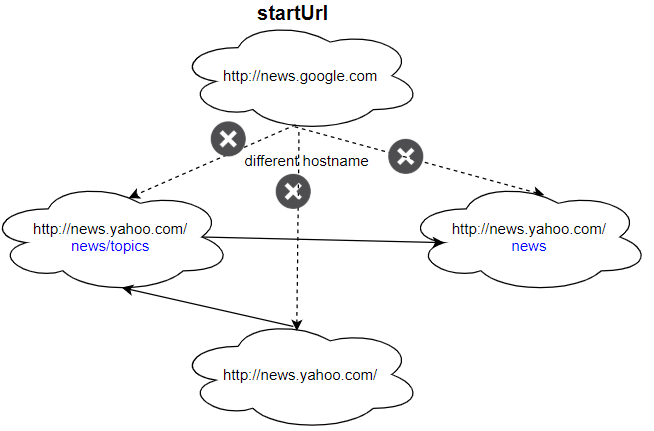
  "http://news.yahoo.com/news",

  "http://news.yahoo.com/news/topics/",

  "http://news.yahoo.com/us"

]

**Example 2:**

****

**Input:**

urls = [

  "http://news.yahoo.com",

  "http://news.yahoo.com/news",

  "http://news.yahoo.com/news/topics/",

  "http://news.google.com"

]

edges = [[0,2],[2,1],[3,2],[3,1],[3,0]]

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

**Output:** ["http://news.google.com"]

**Explanation:** The startUrl links to all other pages that do not share the same hostname.

**Constraints:**

* 1 <= urls.length <= 1000
* 1 <= urls[i].length <= 300
* startUrl is one of the urls.
* Hostname label must be from 1 to 63 characters long, including the dots, may contain only the ASCII letters from 'a' to 'z', digits from '0' to '9' and the hyphen-minus character ('-').
* The hostname may not start or end with the hyphen-minus character ('-').
* See:  <https://en.wikipedia.org/wiki/Hostname#Restrictions_on_valid_hostnames>
* You may assume there're no duplicates in url library.

<https://www.webarchitects.io/dropbox-interview-question-multi-threaded-web-crawler/>

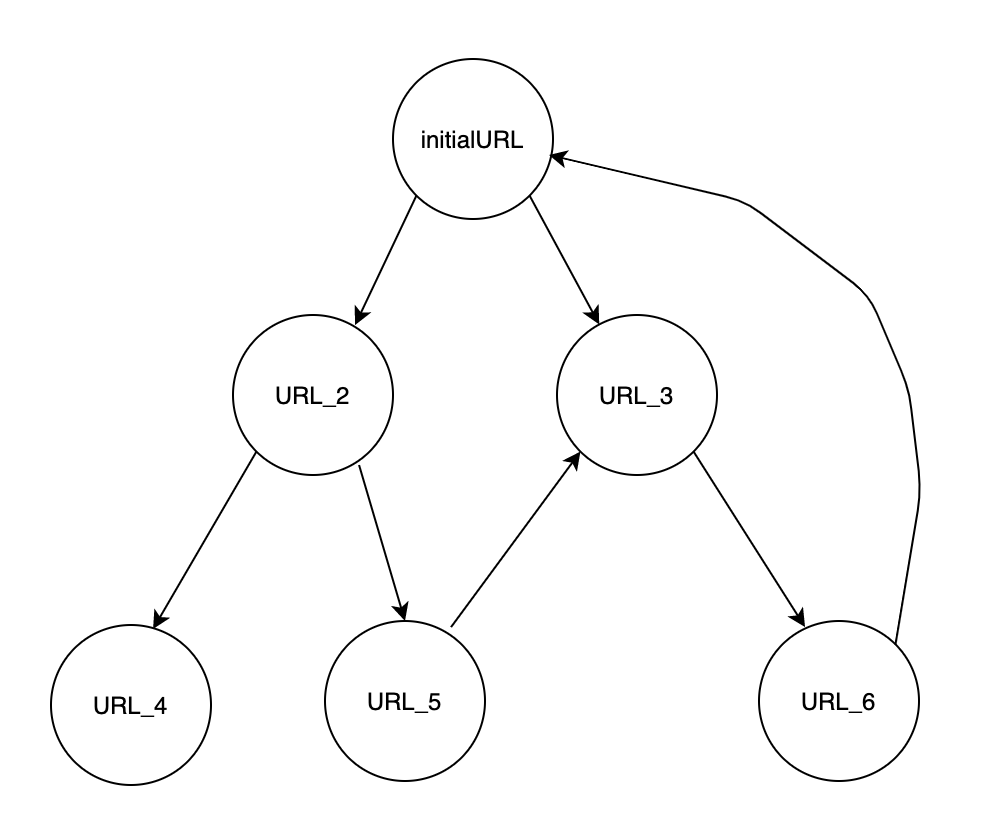
**DROPBOX INTERVIEW QUESTION: MULTI-THREADED WEB CRAWLER**

## Question

Given a getLinks(String initialURL) function, which takes an URL, scan the page and returns a list of URLs found on this page, write a function to find all the URLs that can be accessed by the initialURL.

## Solution

The first question is pretty straightforward. You can think of all the accessible URLs as nodes in a graph. The starting node is initialURL. The directed arrow from URL a to URL b means a contains b. Now the question is equivalent to traverse a directed graph.



You can use either Depth-first search or Breadth-first search. I’m going to use Breadth-first search here since in practice most web crawlers are implemented using BFS. The basic idea is to use a queue to store all the unscanned URLS found by calling getLinks(String URL) and use a set to store all the URLs that have been scanned. The code is shown below.

public Set<String> function findAllURLs(String inititalURL) {

Queue<String> queue = new LinkedList<>();

Set<String> set = new HashSet<>();

queue.addAll(getLinks(inititalURL));

while(!queue.isEmpty()) {

String url = queue.poll();

for(String childURL: getLinks(url)) {

if (!set.contains(childURL)) {

set.add(childURL);

queue.offer(childURL)

}

}

}

return set;

}

## Follow-up

The first part of this question is really a warmup and the follow-up question is the more difficult part, which involves multi-threading. Once you finish the first question, the interviewer will follow up by asking you what’s the most time-consuming part of your web crawling function and how you can improve the performance.

## Solution

The most time-consuming part is the getLinks() function, which needs to scan the page for URLs. So how can we make the page scanning multi-threaded while still keep the function thread-safe. The code is shown below.

public class WebCrawler {

Queue<String> queue = new LinkedList<>();

Set<String> visited = new HashSet<>();

public void crawl() {

while(true) {

String nextUrl;

synchronized(this) {

while(queue.isEmpty()) {

try {

wait();

} catch(InterruptedException e) {

e.printStackTrace();

}

}

nextUrl = queue.poll();

}

List<String> URLs = getLinks(nextUrl);

synchronized(this) {

for(String newUrl: urls) {

if(!visited.contains(newUrl)) {

queue.offer(newUrl);

visited.add(newUrl);

}

}

notifyAll();

}

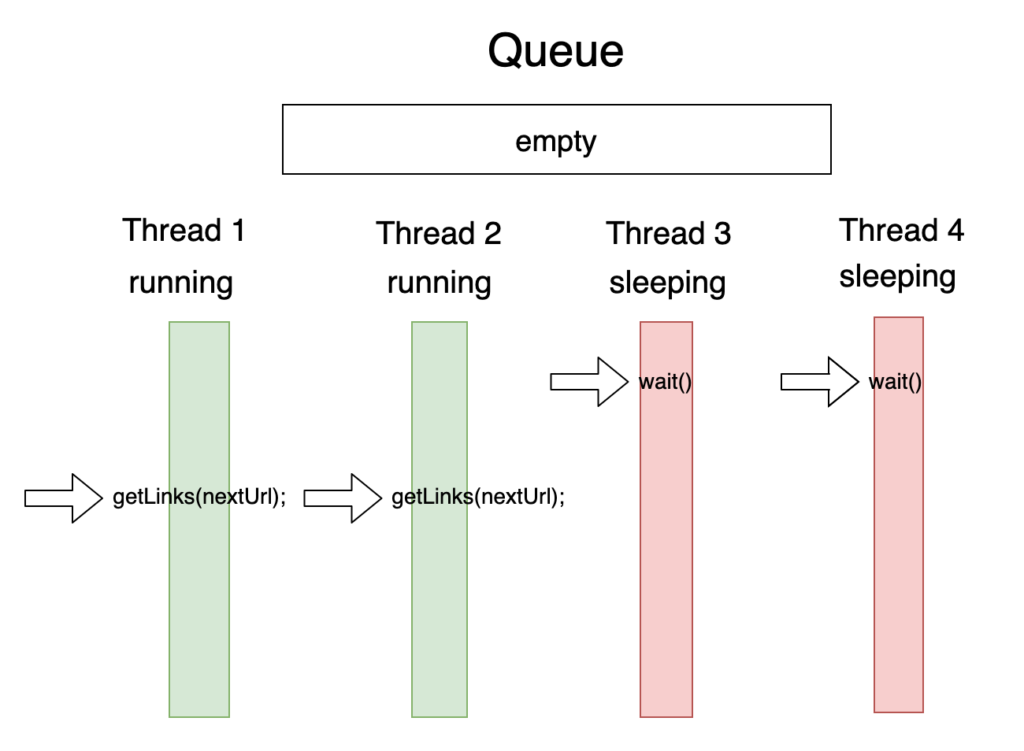
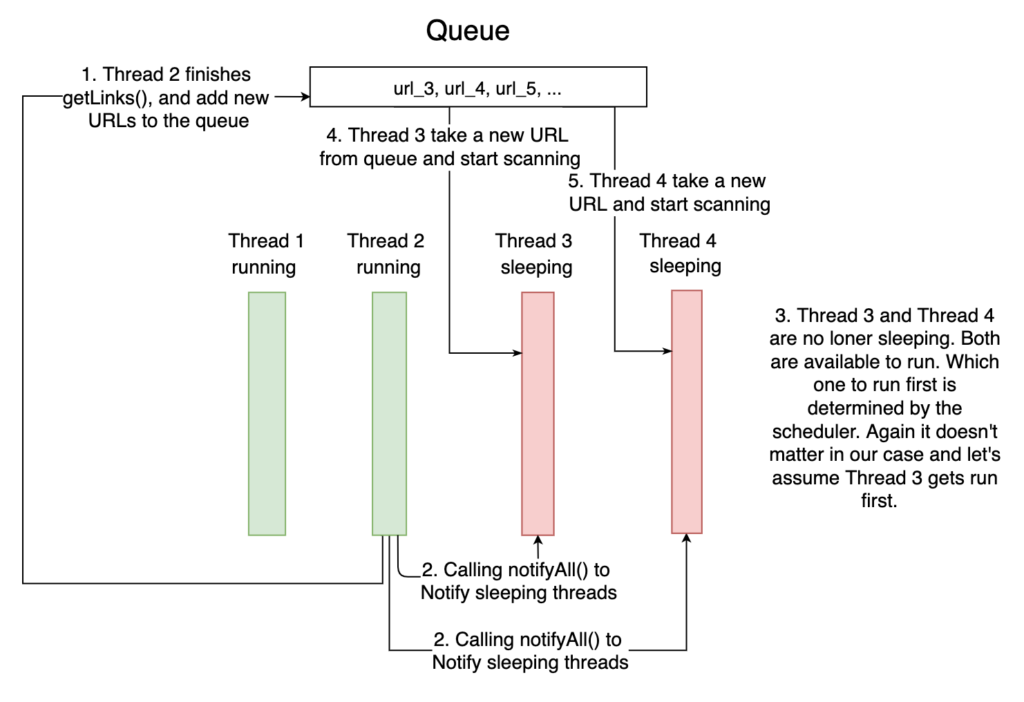
}

}

}

Before we dive into the code, let’s first go over some premise on Java thread synchronization mechanism, specifically synchronized block, wait(), and notifyAll(). A synchronized block is a way Java uses to synchronize different threads. When you wrap a piece of code inside a synchronized block, you are telling Java that only one thread can execute the code in the synchronized block at one time. To enter a synchronized block, a thread needs to first get a lock, also called the monitor object, which is the argument passed to the synchronized block. In our case, it is the **this** keyword, which is an object of the WebCrawler class. In our code, you can see there are two synchronized blocks, synchronized on the same monitor object. This means at any time, there can be only one thread executing in either one of the synchronized blocks. If there is already one thread executing in either one of the synchronized blocks, no other threads can execute in neither one of them. Why do we need to synchronize threads? Because we want to protect the shared resource and prevent race conditions. In our case, the shared resource is the queue and the set. Without the synchronized block, all kinds of race conditions can happen. One of the race conditions can be two threads check if the set already contains the same URL at the same time and both see the URL doesn’t exist in the set, they both proceed into the if block, and two duplicate URLs are added to the queue. The wait() method basically means the current thread gives up the lock and goes to sleep until some other thread enters the same monitor and calls notifyAll().  The notifyAll() method will wake up all the threads that called wait() on the same object.

Now let’s get into the code. You can think of this code as consisted of three parts, the first synchronized block, the calling of the getLinks() method and the second synchronized block. In the first synchronized block, we guarantee that the thread can only proceed when there are some URLs in the queue. In the second synchronized block, we are looping through the newly found URLs and checking if they are already scanned and only add those unscanned URLs to the queue. At the end of the synchronized block, we call notifyAll() to wake up all the sleeping threads to continue after wait(). Note that not all of these awakened threads are getting back to the execution in the first synchronized block at the same time. Remember only one thread in synchronized block at a time. Their order to execute is determined by the scheduler. In our case, this order doesn’t matter. The getLinks() method call isn’t in any synchronized block and it’s this method we want to utilize multi-threading. In the first graph below, I show the status of different threads at a certain point in time, and the second graph shows what happens when one thread finishes the getLinks() method and add new URLs to the queue.

The arrow shows which line of code the thread is currently executing or stopped at before going to sleep.The number shows the order of the events.

## Exit Condition

What happens when all of the URLs are found? Assume the number of URLs accessible by the initialURL is finite. Towards the end of the processing, the number of URLs in the queue will become less, and eventually, the queue will be empty. At that point, all the threads will go to sleep and will never be awakened again. Is there is a more graceful approach by the time all the URLs are found, all thread will exit nicely without hanging around forever? Yes! It’s easy to achieve that. We only need one more variable, which stores the number of threads that are currently scanning for URLs. The updated code is shown below and the added code is marked.

public class WebCrawler {

Queue<String> queue = new LinkedList<>();

Set<String> visited = new HashSet<>();

int workingThreads = 0; // new code

public void crawl() {

OUTER\_LOOP: while(true) {

String nextUrl;

synchronized(this) {

while(queue.isEmpty()) {

if(workingThreads == 0 && queue.isEmpty()) { // new code

break OUTER\_LOOP;

}

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

nextUrl = queue.poll();

workingThreads++; // new code

}

List<String> URLs = getLinks(nextUrl);

synchronized(this) {

for(String newUrl: urls) {

if(!visited.contains(newUrl)) {

queue.offer(newUrl);

visited.add(newUrl);

}

}

workingThreads--; // new code

notifyAll();

}

}

}

}

The thread will exit when there are no other threads scanning for URLs and the queue is empty. When the queue is empty and there no other threads scanning for URLs, we know that no more URLs will be added to the queue and the thread can exit safely.

## Recommended Reading

There are tons of articles talking about thread synchronization on the Internet. Here are just a few of them.

* <https://www.baeldung.com/java-wait-notify>
* <http://tutorials.jenkov.com/java-concurrency/synchronized.html>
* <https://howtodoinjava.com/java/multi-threading/wait-notify-and-notifyall-methods/>

If you want to get a systematic view of multi-threading and concurrency in Java, here is a great book to read.  
Java Concurrency in Practice by Brian Goetz.