Course Project Report for CS3211: A Verification-Oriented Concurrent Web Crawler

Group 8

Nabeel Zaheer Bin Mohamed Taufiq

Abstract

In this report, I design a concurrent web crawler to continuously crawl a set of URLs, first initialised by the seed URLs provided, which is then repopulated by the URLs found from the crawling process. The found or crawled URLs are then indexed, and its HTML files stored in a disk or local directory.

1. Introduction

This report would provide insights and reasoning behind the design and implementation of the web crawler and the measures taken to handle issues that arise due to concurrency as well as the choices made to ensure the efficiency of the crawler. A high-level concurrency design in the form of CSP modelling will be evaluated whereby the concurrent aspects of the crawler are modelled and using the PAT software, provides verification as to its susceptibility to deadlock and data race issues. Based on the findings of the PAT software, a manual attempt to “replay” a concurrent bug through the use of *Thread.sleep()* and an additional lock will be assessed. Finally, measures to automate the replication of the concurrent bug will be touched upon using the concept of Java instrumentation.

The technical implementation of each of the mentioned parts of the course project would then be addressed and justified based on the motivations mentioned above.

An evaluation on the crawling efficiency as well as the limitations of the implemented web crawler would then be covered to allow for future iterations of the project to consider and potentially make further improvements to the design and implementation.

Finally, the report will be concluded with an overall takeaway of the nature and concepts of the course project as well as future works and studies that could be undertaken to further enhance the current design and enrich the understanding on concurrent systems.

2. Approach

In this section, the design and methodology involved in the various parts of the course project will be highlighted accompanied by the reasonings and rationale behind them.

2.1 Concurrency Webpage Crawler:

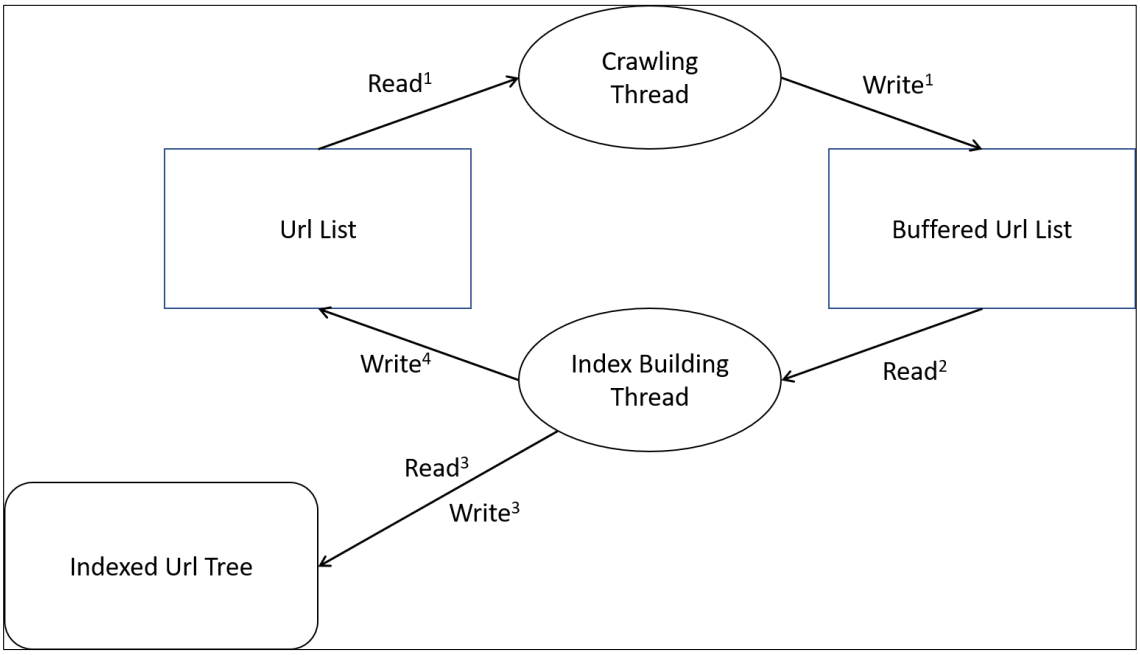


Figure 1: Overall Flow of Concurrent Web Crawler

As seen above in Figure 1, the various components of the web crawler and its main purpose (read and write) are shown. The *Url List*, *Buffered Url List (BUL)* and the *Indexed Url Tree (IUT)* serve as the main data structures incorporated in the web crawling process. The Crawling Thread (CT) simply "crawls" the provided URLs which involves parsing its HTML document to find embedded links or URLs of other web pages. The Index Building Thread (IBT) would then be involved in storing the found URLs, resulting from the crawling process to the *IUT*.

The *Url List* serve as a data structure to initially store the seed URLs provided and subsequently, the found or crawled URLs that have been stored into the *IUT*. Essentially, it will always contain URLs that have yet to be crawled. Mainly, The *Url List* has no limit when it comes to its capacity. However, a data race issue could arise if a CT attempts to retrieve a URL from an empty *URL List*. As such, a form of synchronization would have to be implemented amongst any access to the *URL List* either from the CT or IBT.

The *BUL* would serve as the temporary storage for the crawled URLs and has a fixed capacity of 1000. Only once the *BUL* is full, will it be cleared by the IBT. Naturally, the CT should not be able to add into the *BUL* when it is full. Similarly, since checking of the state of the *BUL* is required between CTs and IBT sharing a single *BUL*, their access to it would also have to be synchronized to ensure that the desired behaviour is observed.

The *IUT* not only serves as a storage for the HTMLs of the crawled URLs, it also serves as a form of index maintenance which would aid in checking if the HTML of a particular URL has been crawled and stored. As such, and index system consisting of 3 layers could be introduced increase the efficiency of the indexing and checking process. The first layer of indexing will be the prefix of the domain of the URL mapped to its corresponding prefix file name on the local directory. For instance, an URL with domain prefix of “a” would be mapped to a file named “a.txt”. This prefix file would then be the second layer of indexing which contains the domains beginning with that prefix and the domain file name it is being mapped to. As such, this means that a URL with domain, “abc.com” would be mapped to “abc.com.txt”. Finally, within the domain file, all the URLs belonging to the specified domain will be mapped to the path to its corresponding HTML file. This would reduce the number of checks required as compared to iterating through a list of all the found URLs. A potential data race issue would be duplicate URLs being stored by multiple IBTs and as such, synchronized access between all IBTs would have to be included.

The CT would crawl the URL received from the *URL List* and then transferring its output to the *BUL*. The crawling process will be done by a library call from an external library, jsoup.

The IBT would store the found URLs in the *IUT* once it has checked that they are not duplicates. The checking of duplicates is done in the IBT to reduce the number of threads sharing the same resources, thus reducing potential data race issues.

From an efficiency standpoint, there will be better time and energy conservation if the number of duplicate URLs being crawled is reduced. First the duplicates form the seed URL provided have to be removed. Secondly, all URLs sent to the *URL List* from the IBT have to be unique. Fortunately, the second condition is already satisfied given the proposed design of the IBT.

2.2 Concurrency Design Modelling:

To model the web crawler in the PAT software, a scaling down on the number of CTs, IBTs and *BULs* would have to be done. This is because the purpose of the modelling is to only verify its concurrency behaviours and not its efficiency.

It is clear from Section 2.1 that the only parts of the web crawler process which involves aspects of concurrency would be the critical regions whereby shared resources are being accessed. As such, all other events that does not contribute to these concurrent aspects will be abstracted.

Furthermore, certain aspects of the web crawler such as the IBT writing back to the *Url List* would not be necessary to model as it will only lead to an infinite loop and a non-terminating process. Thus, a fixed size of seed URLs could be provided and once they have been “crawled” and “stored”, the process will end. Additionally, the behaviour of the IBT to only clear the *BUL* when it is full need not be modelled as it represents more of a behavioural property of the IBT instead of contributing to learning on the concurrent behaviours of the system.

The assert capability of the PAT software can then be used to verify if the system meets certain requirements such as being deadlock free and not being susceptible to data race issues. A check for data race would be to check if the *IUT* contents at the end of the process contain duplicate entries which would simulate duplicate HTMLs being stored in the *IUT*.

2.3 Controlling the Java Scheduler:

To replay or replicate the concurrency bug of data race from the trace obtained through CSP modelling, the removal of any synchronization would have to be done first. Then, a manual insertion of a “Thread.sleep()” method or an additional lock could be done to prevent a single thread from completing its execution in a critical region. As such, this can cause a data race issue, primarily, causing duplicates in the *IUT* by adding these manual insertions in the critical region after the conditional statement to check for duplicate URLs.

2.4 Instrumentation for the Java Scheduler:

To automate the concurrency bug, java instrumentation would be used. It is essentially a Java API which allows for the bytecodes of methods in a program to be modified without making amendments to its source codes [1]. This modification can either be done in a static or dynamic manner. For each of these modifications, a Java agent would have to be used which is essentially a class that either uses a premain() or agentmain() method to initiate static or dynamic modification respectively.

As the name suggest, the premain() method will be executed prior to the main() method of the web crawler program and as such the modifications would have already been done before the program executes.

As for agentmain(), it will execute the modifications while the web crawler program is running and load the agent into the JVM using the Java Attach API.

As for replaying the concurrency bug, since the location at which the code to be inserted is already pre-determined, a static modification using Java agent with the premain() method would be used.

3. Implementation

This section shows address the technical implementations of the proposed design and methodology stated in Section 2.

3.1 Concurrency Webpage Crawler:

The *URL List* is implemented using a Hash Set of Strings as the choice of data structure. This is primarily due to the property of the said set to only allow unique values. As such, when the seed URLs are first obtained and stored into the *URL List*, the Hash Set would remove any duplicate URLs. This would then satisfy the first condition to ensuring efficient crawling by removing any duplicates from the seed URLs.

As for the *BUL*, it is implemented using a Blocking Queue of Url objects. The Blocking Queue data structure is chosen as a limit can be set and the property of the queue is such that elements cannot be retrieved if it is empty and cannot be added if it is full. As such, alongside these inherent checks, coupled with additional conditional statements to ensure that it is only cleared when it is full, it serves as a suitable candidate to fulfil the requirements of the *BUL*.

The *IUT* is implemented with a series of Hash Maps. This is done to ensure that only unique entries are allowed and allow for O(1) time complexity when it comes to checking if a value exists. These Hash Map are then stored as files in the local directory as XML files so as to allow the index files to be human readable.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_161" class="java.beans.XMLDecoder">
 <object class="java.util.HashMap">
  <void method="put">
   <string>https://abc.xyz/investor/</string>
   <string>/home/seed/Documents/cs3211/4Bul/6ct/html/https--abc.xyz-investor-</
string>
  </void>
  <void method="put">
   <string>https://abc.xyz/investor/other/google-code-of-conduct/</string>
   <string>/home/seed/Documents/cs3211/4Bul/6ct/html/https--abc.xyz-investor-
other-google-code-of-conduct-</string>
  </void>
 </object>
</java>
```

Figure 2: Domain index file example

Figure 2 shows an example of a domain index file which contains all the URLs belonging to a particular domain and its mapping to a local directory path to its respective HTML files. The methods involved to add new HTML entries onto local directories are included in an IndexedUrlTree class object which also initialises the structure of the index files immediately after the program starts.

The CT is implemented using a CrawlingThread class which contains the methods involved to retrieve URLs from the *Url List* as well as to crawl them and storing them into the *BUL*. The crawling process begins by first ensuring that the URL obtained is a valid

one, that is, it responds with a HTTP response that indicates that the site is reachable. This is done via a method in a URL class developed which is used to initialise URL objects but also contains methods relating to the properties of the URL. In this case, a method, getResponseCode() is used to return with the respective response code. As long as the site is able to send a proper HTTP response, including a 200 or 404 response, it will not be discarded. A second check is to check if it is crawlable and this is done by a second method, getCrawled() which returns an Elements object which contains all of the found URLs. The size of this object can be acquired and if it is non-zero, the found URLs will proceed to be stored into the *BUL*. However, if the size of the Element object is zero, the URL will not be discarded immediately. Instead, an additional manipulation to convert it to its root URL will be done and the same checks will be conducted on it. This is done so as to maximise the results obtained from crawling. The found URLs will then be used to initialise a URL object along with its linked or source URL as its properties.

The IBT proceeds by first doing a synchronized access to the full *BUL* and will copy all the URL objects into a temporary array. This is done to allow the CT to continue to access the *BUL* without waiting for the entire process of the IBT to be completed since the access has been synchronized. The IBT would then obtain the HTML String of each of the found URL in the URL object via method getHtml(). The same checks to indicate if the found URLs are crawlable will be conducted here as well. This will then proceed to index the found URL and store its HTML via method in the IndexedUrlTree class called addNewEntry(). This method will return an empty string if the found Url is a duplicate, a “ignored” string if the number of HTMLs stored has reached the maximum allowed number, “dead-url” if the found url is not crawlable and the local directory path to the HTML file if it was successfully stored. Based on these return values, only those that were considered to be duplicates and dead will be discarded and not be passed to the *Url List*.

3.2 Concurrency Design Modelling:

The modelling of the *URL List* will be done using a channel variable. This is done as a channel is suitable for the removing and adding elements in a queue-like manner. To implement this, a channel would have to be first initialised and this is not the usual use case of the channel. Conventionally, the system will begin with an empty channel which will get input and outputs elements while the system is running. However, since the seed URLs is a fixed set of “URLs” that is provided before the web crawler even executes, the channel would need to be populated first. To model the URLs, integers would be used instead for simplicity purposes. First an integer array will be declared an initialised with the seed URLs. This array would then be converted into a user-defined List variable.

```
// Initialise fixed size of seed urls
//var seedUrlArr = call(InitRandom,URL_SIZE,MAX_URL_VAL+1);
var seedUrlArr[URL_SIZE] = [1,2,1,7];
var<List> seedUrlList = new List(seedUrlArr);
channel urlList URL_SIZE;
var seedUrl;

UrlInit(i) = if(i<=URL_SIZE){get{seedUrl = seedUrlList.GetUrl(0)} -> urlList!seedUrl -> UrlInit(i+1)}else{Skip};
```

Figure 3: Initialise channel with seed URLs

Figure 3 simply depicts a code snippet from the CSP file where a process, UrlInit(i) is designed such that it continuously transfers all the contents of the List into the channel.

As for the *BUL*, it will also be modelled using a channel variable as its use is similar to that of the *Url List*. The *IUT*, however will be modelled using a user-defined List.

```
// Shared BUL
channel bul[NUM_BUL] BUL_SIZE;
var bulIndexArr = call(InitIndex, NUM_BUL, NUM_CT);
var bulIndexList = new List(bulIndexArr);
// CT variables
var url[NUM_CT*NUM_BUL];
var newUrl[NUM_CT*NUM_BUL];
// IBT variables
var<List> iut = new List();
```

Figure 4: Declaring of BUL and IUT data structure models

Figure 4 shows the declaration of the variables to simulate the *BUL* and the *IUT*. Additionally, other variable which aids in the maintaining the index of the multiple *BULs* and to simulate individual URLs are also declared.

```
CrawlingThread(i) = if(!call(empty,urlList)){urlList?toCrawl{url[i] = toCrawl} -> crawl.i{newUrl[i] = url[i]} -> bul[bulIndexList.Get(i)]!newUrl[i] -> CrawlingThread(i)}else{CrawlingThread(i)};
```

Figure 5: Crawling Thread process

Figure 5 shows the Crawling Thread process where it will first check if the channel containing the uncrawled “URLs” is empty. If it is not, it will retrieve the first “URL” integer and associate its value to a variable url[i]. This variable simulates an uncrawled URL. Then a crawl event ensues which initialises a variable newUrl[i] with the value of url[i]. This simulates the crawling process whereby newUrl[i] represents the found URL. This also means that this model assumes that a Url is crawled to return itself. This found URL would then be stored into the *BUL* channel.

```
36  IBTWrites(i) = if(!iut.Contains(i)){index{iut.Add(i)} -> urlListWriter -> Skip}else{Skip};IBTSystem;
37
38  IndexBuildingThread(i) = if(!call(cempty,bul[i])){bul[i]?crawled -> IBTWrites(crawled)}else{IndexBuildingThread(i)};
39
40  IBTSystem = |||x:{0..NUM_IBT-1}@IndexBuildingThread(x);
41  CTSystem = |||x:{0..(NUM_CT*NUM_BUL)-1}@CrawlingThread(x);
42  System = UrlInit(1);((CTSystem) ||| (IBTSystem));
43
44
45  #assert System deadlockfree;
46  #assert System reaches goal;
```

Figure 6: Index Building Thread and indexing process

The behaviour of the IBT process is then shown in Figure 6 whereby as long as the *BUL* channel is not empty, the IBT would retrieve the found URL integer and proceed to execute the process of writing it to the *IUT*. This indexing process is indicated by process IBTWrites(i) which takes in the value of the found URL integer and checks if the *IUT* List contains that particular integer. If not, the found URL integer can be added to the *IUT* and the event to signify adding the found URL back into the *URL List* is also included.

Figure 6 also shows the execution of the entire system which begins by first initialising the *URL List* and then executing the CT and IBT processes concurrently in an interleaving manner. Two assertions are also made to verify that the system if both deadlock free and reaches a goal to indicate that there are duplicates in the *IUT*. The goal is defined by an isDuplicate() method designed in the methods of the user-defined List variable which checks all the contents of the List and returns a Boolean “true” value if a duplicate is found.

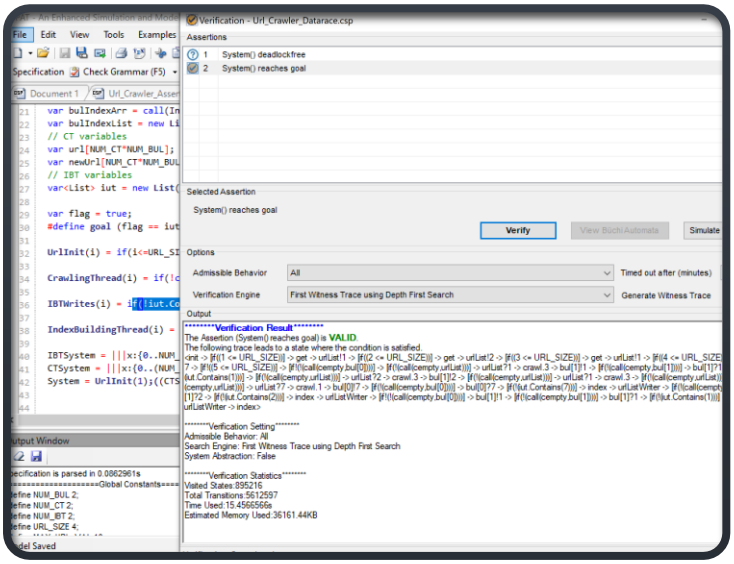


Figure 7: Assertion of duplicate existing is valid

Figure 7 shows that when verification of the assertion to check for duplicates or in other words for proof of data race is conducted, it returns a valid assertion. This means that the software can provide a trace of the system which results in the *IUT* List to contain duplicate values. Through close inspection of the trace, it can be determined that this is due to the critical region of reading and writing the *IUT* to not be synchronized amongst

the multiple IBT process in the system. As such to counter this, a form of synchronization which enables a single IBT process to complete reading and writing into the *IUT* List at any one time is introduced.

```
IBTWrites(i) = atomic{if(!iut.Contains(i)){index{iut.Add(i)} -> urlListWriter -> Skip}else{Skip}};IBTSystem;

IndexBuildingThread(i) = if(!call(cempty,bul[i])){bul[i]?crawled -> IBTWrites(crawled)}else{IndexBuildingThread(i)};
```

Figure 8: Synchronization added to IBTWrites process

Figure 8 shows that the sequence of reading the *IUTI* and writing to it has been categorised under the atomic event. This gives this set of sequence of events to be given a higher priority and thus, no interleaving would occur within the sequence.

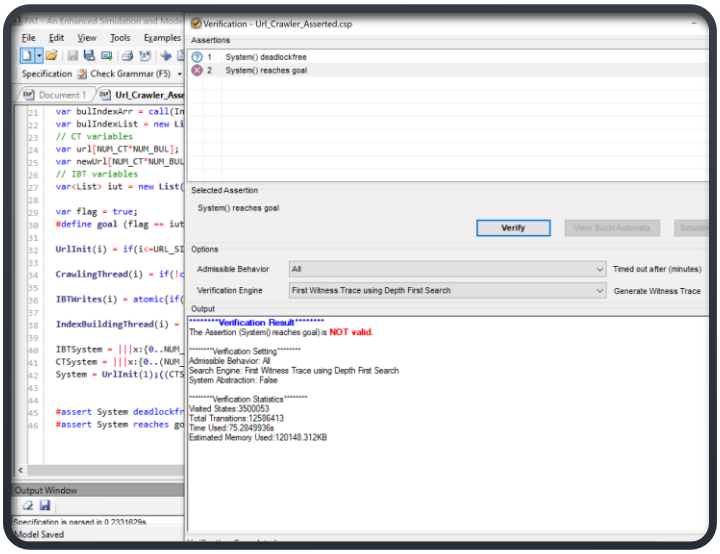
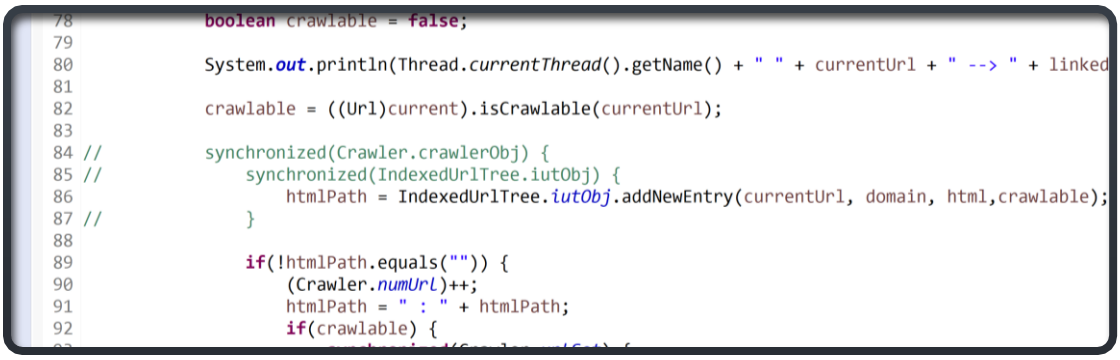


Figure 9: Assertion of duplicate existing not valid

Figure 9 shows that the result of verification of the data race assertion would not return to be not valid and this would then prove that the program could not find a trace which results in the *IUT* List to end up with duplicate URL integers and thus, indicate that the system is now free from the data race issue.

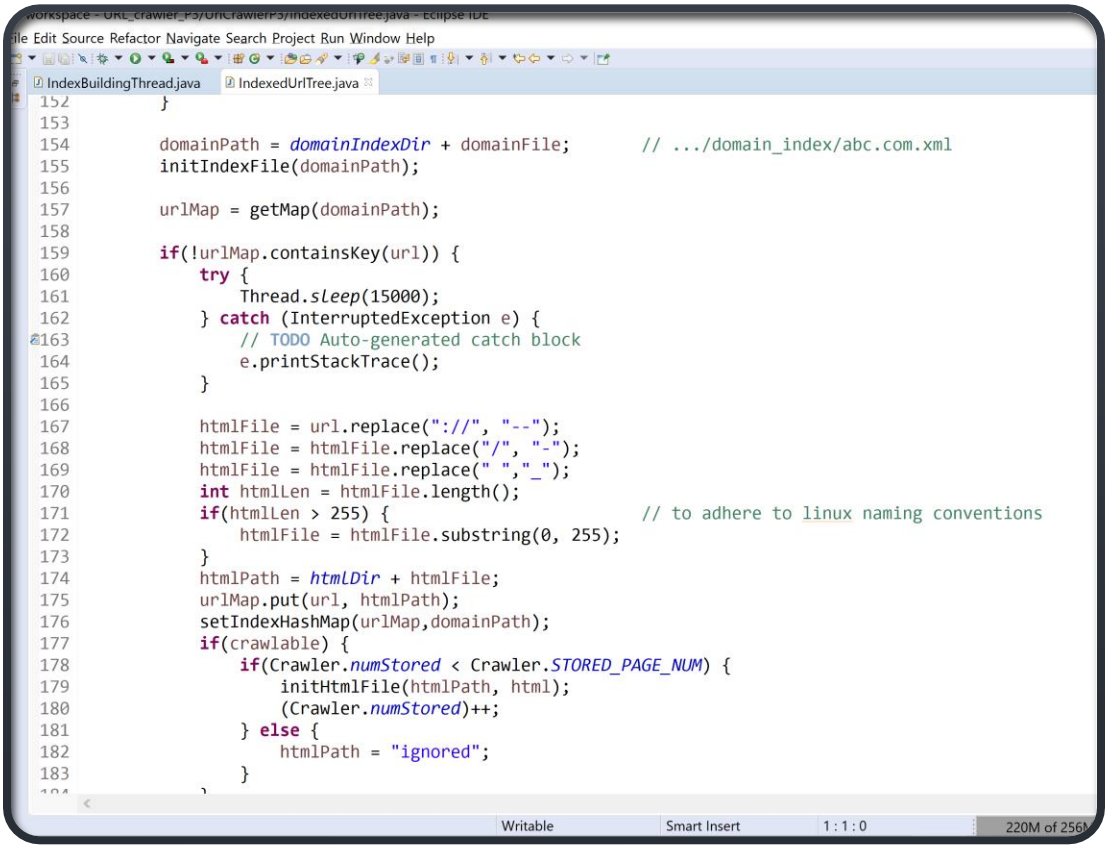
3.3 Controlling the Java Scheduler:

According to the trace obtained from the previous section, it can be posited that the data race occurred when there were no synchronizations and when multiple IBT are in the critical region at the same time. Therefore, to replay this in the Java program, a Thread.sleep() method can be used to force a current thread to remain in a critical region while another enters it as well. The critical region in this case, would be between reading from the *IUT* to check for duplicates and writing the corresponding HTML to local directory.



```
78              boolean crawlable = false;
79
80              System.out.println(Thread.currentThread().getName() + " " + currentUrl + " --> " + linked
81
82              crawlable = ((Url)current).isCrawlable(currentUrl);
83
84  //          synchronized(Crawler.crawlerObj) {
85  //              synchronized(IndexedUrlTree.iutObj) {
86                      htmlPath = IndexedUrlTree.iutObj.addNewEntry(currentUrl, domain, html,crawlable);
87  //              }
88
89                  if(!htmlPath.equals("")) {
90                      (Crawler.numUrl)++;
91                      htmlPath = " : " + htmlPath;
92                      if(crawlable) {
```

Figure 10: Removal of synchronization statements



```
File Edit Source Refactor Navigate Search Project Run Window Help
IndexBuildingThread.java   IndexedUrlTree.java
152         }
153
154         domainPath = domainIndexDir + domainFile;       // .../domain_index/abc.com.xml
155         initIndexFile(domainPath);
156
157         urlMap = getMap(domainPath);
158
159         if(!urlMap.containsKey(url)) {
160             try {
161                 Thread.sleep(15000);
162             } catch (InterruptedException e) {
163                 // TODO Auto-generated catch block
164                 e.printStackTrace();
165             }
166
167             htmlFile = url.replace("://", "--");
168             htmlFile = htmlFile.replace("/", "-");
169             htmlFile = htmlFile.replace(" ","_");
170             int htmlLen = htmlFile.length();
171             if(htmlLen > 255) {                         // to adhere to linux naming conventions
172                 htmlFile = htmlFile.substring(0, 255);
173             }
174             htmlPath = htmlDir + htmlFile;
175             urlMap.put(url, htmlPath);
176             setIndexHashMap(urlMap,domainPath);
177             if(crawlable) {
178                 if(Crawler.numStored < Crawler.STORED_PAGE_NUM) {
179                     initHtmlFile(htmlPath, html);
180                     (Crawler.numStored)++;
181                 } else {
182                     htmlPath = "ignored";
183                 }
Writable   Smart Insert   1 : 1 : 0   220M of 256M
```

Figure 11: Addition of Thread.sleep() in the critical region

Figure 10 shows the synchronizations statements being removed from the IBT class to access the *IUT*. This would mean that interleaving threads can access the *IUT* at the same time. Additionally, Figure 11 shows the addition of the Thread.sleep() method in the critical region as mentioned before. This would allow for an opportunity for two IBT trying to index the same found URL to not detect any duplicate and thus, cause the duplicates to be logged.

This duplicate could then be shown via the output form the res.txt file which shows all the found URLs. Ideally, it should be a unique list of found URLs and with the HTML path being unique as well. However, by importing the res.txt into an Excel and check for

duplicate values in the column of HTML paths, it can be determined if the data race bug was successfully replayed.

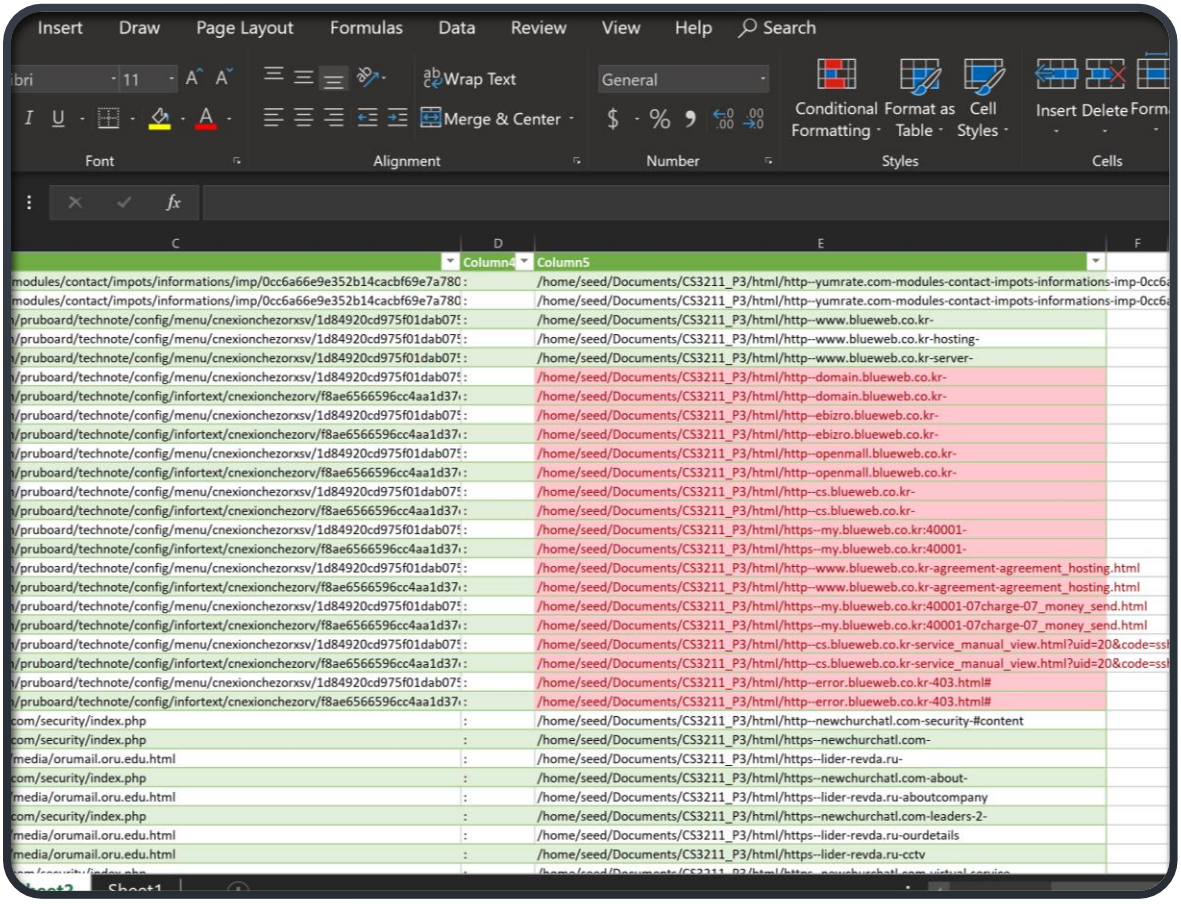


Figure 12: Duplicate values detected in HTML paths

Figure 12 shows that there exist multiple duplicate HTML paths recorded in the res.txt file. This is not usually possible as Linux systems would not allow two files in the same folder to have the same name. However, this is still shown in the res.txt file which means that duplicate entries were successfully “stored” and the addNewEntry() method returned duplicates HTML path since it was not able to detect for duplicates. This would then be printed out in res.txt, thus proving that the data race issue was replicated.

4. Evaluation

To evaluate the efficiency of the web crawler, multiple factors were put into consideration. Chiefly, the number of *BULs* and the number of CTs were examined and its effects on the average number of crawled URLs per hours is calculated. There were other factors that may affect the performance such as other programs running on the computer which could only be controlled to a certain extent.

To conduct this test, the number of URLs found was maintained in each execution of the program. Then, this number would be printed out every hour. The average rate would then be computed at the end of execution.

First, based on the recommendations of the project guideline, 3 *BULs* is used with differing number of CTs.

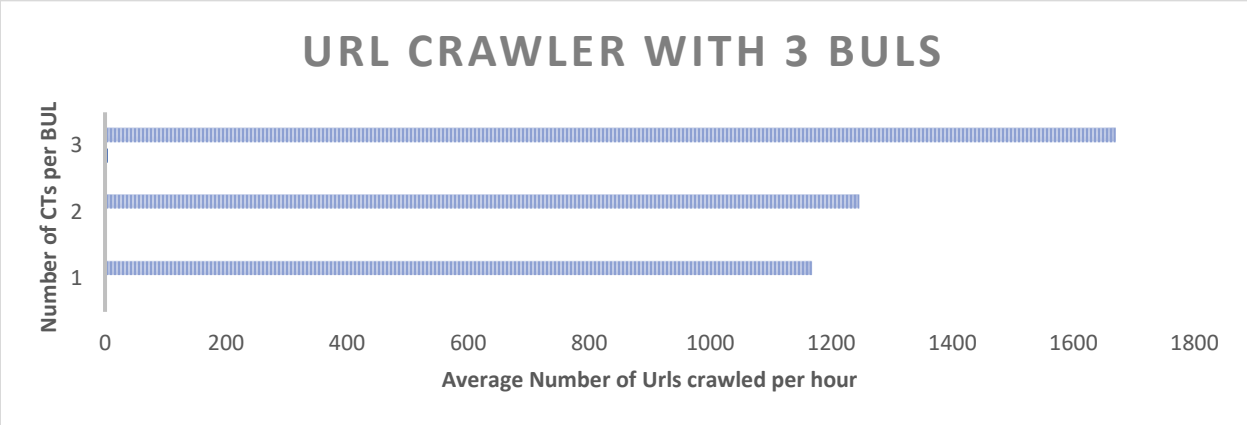


Figure 13: Plot of rate of number of found URLs against increasing number of CTs for 3 BULs

Figure 13 shows that the performance and thus efficiency of the web crawler increases with increasing number of CTs employed.

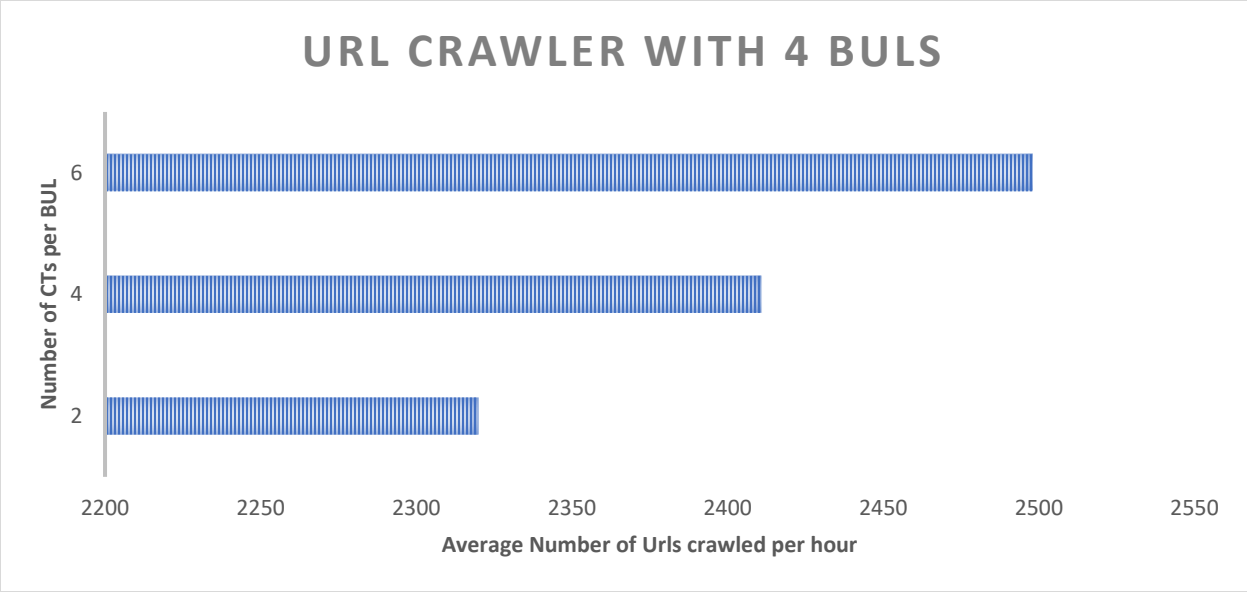


Figure 14: Plot of rate of number of found URLs against increasing number of CTs for 4 BULs

As see in Figure 14, increasing the number of *BULs* did result in an increase in the average number of URLs found per hour. It was also found that with this configuration, only using any more than 6 CTs per *BUL* would cause an insufficient memory error. Thus, the configuration of using 6 CTs per *BUL*, 4 *BULs*, 4 IBT, 1 *IUT* and 1 *Url List* would be used as it provided with the best performance. These values were obtained over the course of running the program for 8 hours. Since the efficiency of the program is not consistent throughout the entire duration, if it was to be executed for 24h, the values shown above may not give a good basis for extrapolation.

As for the concurrency bug replay, the beforementioned results can be obtained rather reliably by feeding seed URLs with the same domain. As such, this increases the chances of multiple IBTs potentially indexing the same URLs and thus, with the manual insertions of the Thread.sleep() method, would cause the data race bug to be observed yet again.

5. Discussion

One of the pros of the designed web crawler is the intensive checking on the validity of the various variables at the different portions of the program. This means that it is unlikely that the program would encounter an input that may cause it to crash or to encounter an error that is not handled. Thus, this allows the program to continue executing for as long as possible.

Furthermore, due to the implementation of the web crawler data structure, it also reduces the eliminates the number of duplicate URLs getting crawled.

However, a limitation of the program is the self-designed concept of indexing using Hash Maps. Although the concept is driven by the need for efficiency, the execution of the entire indexing and storing process could have been improved.

6. Conclusion

In conclusion, in possible future iterations of the project, a more extensive research into the indexing and file writing process could be done to further improve the efficiency of the program. Incorporations of existing external libraries could even be considered and have its efficiency assessed.

Overall, the course project has provided a well-rounded scope into the topic of concurrency and the factors that influence it or are affected by it. The web crawler program, for instance, has allowed one to understand the extent of the effect that the number of threads and the number of shared resources has on the efficiency of the program and that greater quantity may not necessarily be associated with an increased performance. There are other external factors that have to be put into consideration, and it is the balance between these factors that will determine if the web crawler program is indeed one that provides optimum performance.

Reference:

[1] Precub, A. (2019, December 31). Guide to Java Instrumentation. Retrieved from https://www.baeldung.com/java-instrumentation