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

Group 3:

Niklas Forsstroem- A0208754Y Bernardo Altamirano- A0214411Y

Rachit Bansal- A0214350W Bartosz Struzinski- A0209841A

Johan Mallo Bakken- A0214419J

Abstract

In this report, we designed a web crawling program that efficiently crawls as many URLs from a given URL list seed. The objective of our program was to retrieve as many URLs, along with their respective HTML content, as possible. This was to be done without duplicating then URLs. Our initial approach to the system configuration was to have 3 Buffered URL List (BUL), each associated with 2 Crawling Threads (CT) and 1 Index Building Thread (IBT) that writes the data from the BULs into a Indexed URL Tree (IUT). The detailed explanations of these threads and artefacts can be found in the preceding sections of the report.

The project was divided into 4 main parts:

- Building the Java concurrency program
- building a CSP# concurrency model that is verified in Process Analysis Toolkit (PAT) for the abstracted problem
- Manually performing replay of the Java program to illustrate concurrency properties
- Utilizing java agents to modify the binary files with the goal of enabling custom scheduling

1. Introduction

In order to create a program to concurrently crawl URLs from the web without duplicates, our program consisted of two types of threads and two artifacts. Their names and descriptions are the following:

Crawling Thread (CT): This thread is in charge of finding new URLs and verifies duplicates in the IUT, if it is a duplicate it is discarded, else it is written into the BUL. CTs will be finding URLs consistently and should write into their corresponding BUL until it is full. When it is full, CT waits idle until it is cleared by the IBT.

Index Building Thread (IBT): This thread is the intermediary between the BUL (Buffered URL List) and the IUT (Indexed URL Tree). Whenever a BUL reaches its capacity, IBT connected to this data structure is notified and empties it, populating the IUT at the same time. Otherwise (when the BUL is not fully populated), the IBT is idle, waiting to be notified by one of the CT connected to this particular BUL.

Buffered URL List (BUL): The BUL is an ArrayList that stores the URLs scraped by the Crawling Threads. BUL has a maximum capacity of 1000 objects. URLs are stored inside an object called URLTuple, which additionally stores the HTML content of a website connected to that URL. That HTML content is written to disk when an index building thread empties the buffer and populates the IUT structure.

Indexed URL Tree (IUT):

Our model consists of 3 Buffered URL List (BUL), each associated with 2 Crawling Threads (CT) and 1 Index Building Thread (IBT) that writes the data from the BULs into a Indexed URL Tree (IUT). The purpose of the indexed url tree is to store urls on disk with fast read/write access. We have thought of different implementations here. Our initial thought was to make a Trie, implementing with in-memory storage is trivial, but implementing a trie with on-disk storage might not be the best solution. We later thought of using the MapDB library, but concluded that the library really wasn’t meant for as frequent read/write access. The final implementation of the IUT is calculating the hashcode of each url that is getting stored, we have a file for each possible combination of the first three digits and store the urls hash codes in its correspondent file file. This provides fast reading and writing to the files.

P1:

For the first part, we used Java in order to create a concurrent web crawler which is a modified implementation of a well known producer-consumer concurrency design pattern. Crawling thread, which is the consumer, populates the buffer. When the buffer reaches its capacity, the producer waits until the IUT connected to that buffer empties it fully. There are two crawling threads (producers) connected to each buffer, together with one index building thread. To prevent data race, they synchronize using the intrinsic lock of the BUL object (which is just an ArrayList).

P2:

After making the design of our solution to the problem, half of the team worked on modeling the Java solution in a higher level concurrency design using PAT. By doing this, we could demonstrate the flaws that can arise if adequate synchronization is not performed. The PAT-model was designed to mimic the components of the web crawler, allowing us to see how it's various components can possibly interact within the system. Analyzing the traces of events that can occur in this model and verifying that it is deadlock free were the goals of this part of the project.

P3 & P4:

After having seen the potential pitfalls from inadequate locking in PAT we wanted to display these scenarios in java. This was first done by means of directly altering the source code in a way which explicitly incurred data races in our model. We managed to construct these in such a way that we could clearly monitor the occurrence of the data race. These findings were later carried over into java agent which has the effect of only altering the binary files of the program. The consequence was that we could adequately display the bug without making hazardous changes in the source code of our web-scraper. Lastly the agent was improved such as to provide complete orchestrating privileges of the index builders. This enabled us to perfectly mirror a particular execution trace from PAT, simply by providing a string of the desired trace as a parameter to the agent.

2. Approach

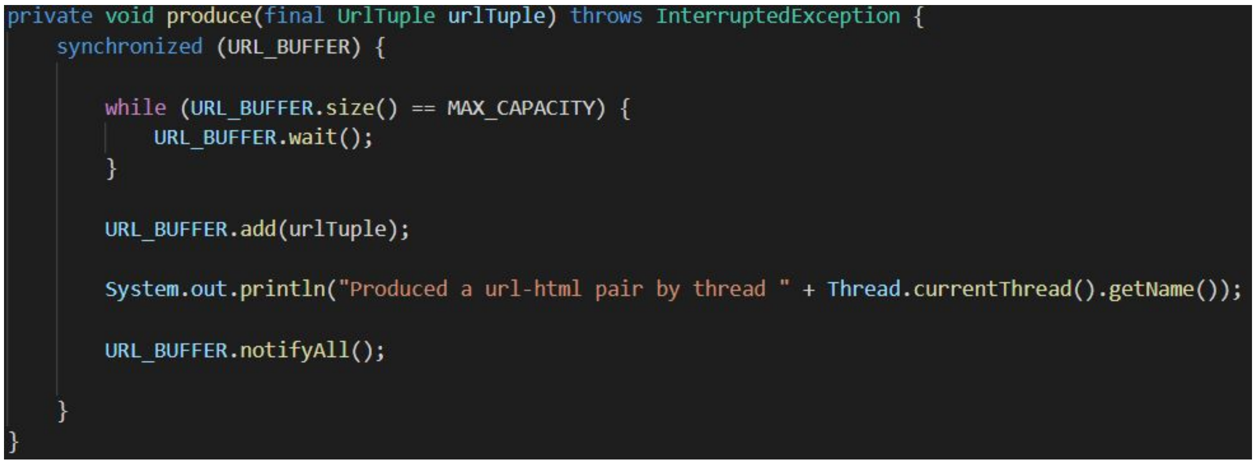
2.1 Concurrency Webpage Crawler:

As mentioned before, our model consists of 3 Buffered URL List (BUL), each associated with 2 Crawling Threads (CT) and 1 Index Building Thread (IBT) that writes the data from the BULs into a Indexed URL Tree (IUT). Therefore in total there are 6 CTs, each with its own task stack containing URLs which are to be visited. The URLs are popped off the top of the stack, and whenever a URL is obtained from a website's source code, it is pushed on top of the stack. Such design means that:

A. The URLs which have been originally placed at the top of the stack might never be visited.
B. When the stack runs out of URLs, the crawler becomes in a way dead, because it will have no tasks in its stack, and therefore will not do anything.

As a first step the seed file is divided (almost) equally between the 6 crawling threads. Since there are 2000 URLs, 5 threads will get 333 URLs and the 6th thread will get 334 URLs. Then the Crawling Threads (producers) are initialized with a task stack, a buffer, and buffer's capacity. Immediately after the index building thread (consumers) are initialized with the buffer (on which synchronization will occur). In our concurrency model we have used a modified producer-consumer concurrency design pattern. In the original problem the producer populates the buffer whenever it is not full, and the consumer empties the buffer whenever it is not empty and does it one element at a time. In our design, however, the consumer thread is woken up when the buffer reaches its full capacity and then empties the buffer, all elements at a time (in our case 1000).

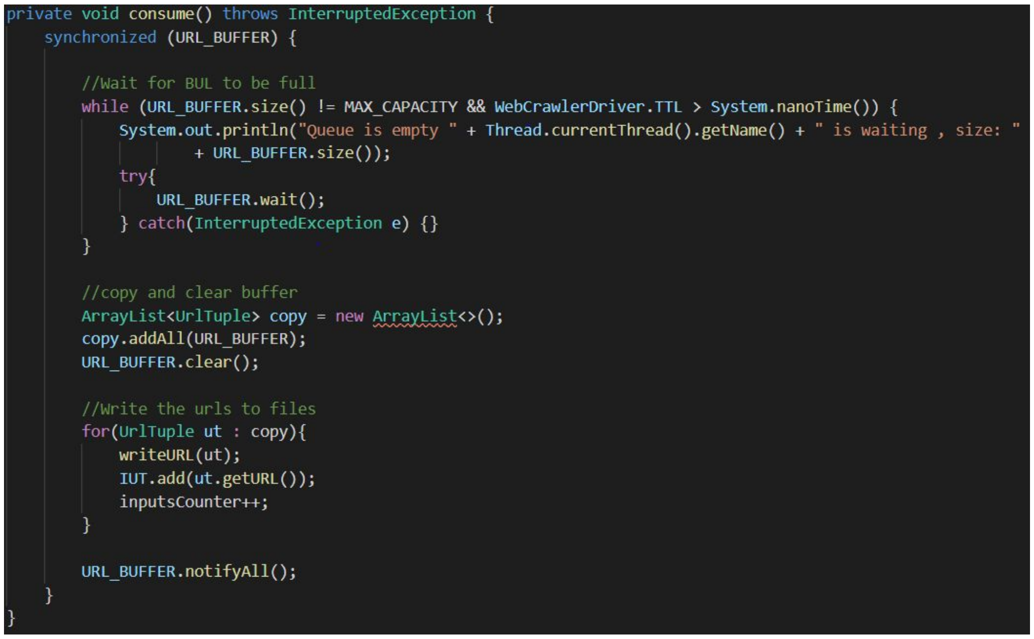
Let me now outline the way the web crawling threads and index building thread cooperate. Each buffer, which is an ArrayList object, is shared between three threads - two crawlers and one builder. We are using an intrinsic lock to provide synchronization on the buffer.



```
private void produce(final UrlTuple urlTuple) throws InterruptedException {
    synchronized (URL_BUFFER) {

        while (URL_BUFFER.size() == MAX_CAPACITY) {
            URL_BUFFER.wait();
        }

        URL_BUFFER.add(urlTuple);

        System.out.println("Produced a url-html pair by thread " + Thread.currentThread().getName());

        URL_BUFFER.notifyAll();

    }
}
```

Figure 1. Produce method of the crawling thread

As one can see, whenever the buffer capacity has been reached, the web crawler releases the lock it holds, allowing for some other thread to wake up. Only two other threads can wake up at this point - either another crawling thread, which will immediately block since the buffer is full, or a building thread, which will empty the buffer in Figure 2. When the crawling thread adds the URL to the buffer it prints its progress message to the console and notifies all thread blocked on the buffer that the lock has been released.



```java
private void consume() throws InterruptedException {
    synchronized (URL_BUFFER) {

        //Wait for BUL to be full
        while (URL_BUFFER.size() != MAX_CAPACITY && WebCrawlerDriver.TTL > System.nanoTime()) {
            System.out.println("Queue is empty " + Thread.currentThread().getName() + " is waiting , size: "
                    + URL_BUFFER.size());
            try{
                URL_BUFFER.wait();
            } catch(InterruptedException e) {}
        }

        //copy and clear buffer
        ArrayList<UrlTuple> copy = new ArrayList<>();
        copy.addAll(URL_BUFFER);
        URL_BUFFER.clear();

        //Write the urls to files
        for(UrlTuple ut : copy){
            writeURL(ut);
            IUT.add(ut.getURL());
            inputsCounter++;
        }

        URL_BUFFER.notifyAll();
    }
}
```

Figure 2. Consume method of the building thread

Building thread firsts checks whether or not the buffer is full, since it wants to empty the entire buffer at once. Then, for each URL object in the buffer, it writes the HTML content into the disk and the URL itself into the IUT. At the end of its task it notifies other threads blocked on the buffer and the entire producer-consumer cycle repeats. The try-catch surrounding the call on wait() is there to make sure interrupted exception is caught if the thread gets interrupted while waiting.

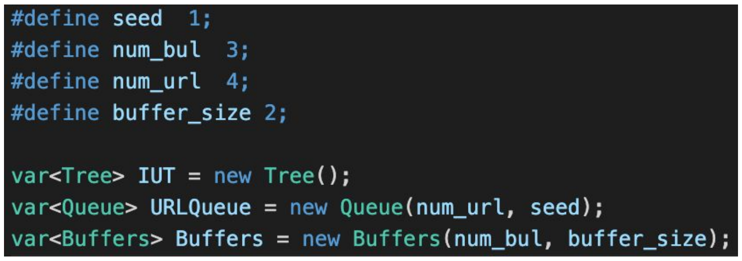
Our ideal thought was to terminate the threads by having logical conditions that made the threads finish their run method and then terminate. When there is a chance of threads waiting we also had to come up with a way to make these terminate. We have the same logic as with the other threads, but we make sure to interrupt them from the main method to make sure none of them are stuck waiting.

2.2 Concurrency Design Modelling:

We modelled the URL crawler using PAT to check the following the concurrency properties:
Assertion 1: The IUT should not have duplicated URL (no data race).
Assertion 2: The system can always keep working until all the URLs have been found (no deadlock).

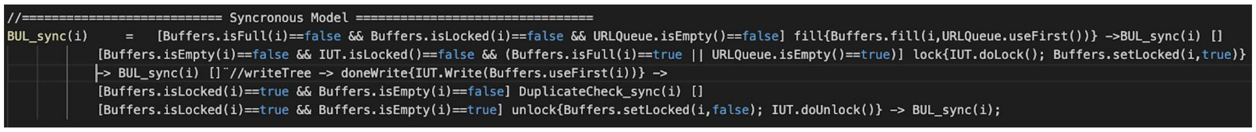
In order to be able to model our design in PAT, we had to use CSP with C# classes that allowed us to make our data structures. The classes are all included in our repository and the structures will be described in the next section, a screenshot of how we initialize these classes is shown in Figure 3.

Firstly, we used the *Queue* data structure to represent the URLs in the queue as numbers. So, occurrence of the same number will represent the occurrence of the same URL, we did this to simplify our model as much as possible in PAT so that we could focus on the assertions that we wanted to prove. The constructor of the Queue asks for the number of URLs you want to have in the queue and a random seed generator, so that the numbers are random, but we can still work with them again to perform tests.
Secondly, we defined a tree data structure named *Tree* to represent the IUT which is a queue which stores the distinct URLs which are put into the tree from the buffers. This constructor is empty, since the IUT is initialized empty and has some specific methods programmed so that it can work the way we want it to, as compared to a normal Queue. We added a lock method into this structure so that the Queue could be locked while a IBT is writing into it, preventing another one to read or write and get a duplicate.
Thirdly, we define a data structure to store the buffers called *Buffers* which is an *Array* whose elements are *Buffers;* the constructor is called with the number of BULs to be initialized and their size. The only enhancement made to the buffers, was also a lockable state so that the CTs are not able to write into them while they are full and in process of being emptied out.


```
#define seed  1;
#define num_bul  3;
#define num_url  4;
#define buffer_size 2;

var<Tree> IUT = new Tree();
var<Queue> URLQueue = new Queue(num_url, seed);
var<Buffers> Buffers = new Buffers(num_bul, buffer_size);
```
Figure 3. Data structures used

As for our PAT model, we designed 3 processes to represent all of the paths that the URL could take all the way from the URL list up to when it is stored in the IUT: CT(i), CheckDuplicates( ) and URLTree( ).

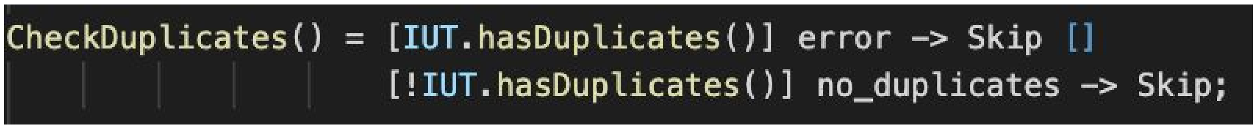
The first one, **BUL_sync(i)**, is the main process of the model. It has four general choices that branch into different events that might happen when the BUL is called. The *first scenari*o is when the BUL is not full and it is not locked, which means that it should be filled up with the next element of the URLQueue (as long as it is not empty), this will call again BUL_sync recursively until the condition changes. The *second scenario* is if the BUL is not empty, it is not locked and either it is full or the URLQueue is empty, which means that it should now be emptied into the IUT to either refill again or to terminate the program if there are no more URLs to retrieve. In this scenario, both the IUT and the BUL are locked and BUL_sync is called again which will fall into the next scenario on the next iteration. The *third scenario* is when the BUL is locked and it is not empty, which means that it should be emptied out and written into the tree until there are no more elements on the BUL. The *fourth and last scenario* is when the BUL is locked and it is empty, in which case it should unlock the BUL and the IUT again so that they can work as normal. The previously described process is shown in Figure 2.



```
//========================== Syncronous Model ===============================
BUL_sync(i)     =   [Buffers.isFull(i)==false && Buffers.isLocked(i)==false && URLQueue.isEmpty()==false] fill{Buffers.fill(i,URLQueue.useFirst())} ->BUL_sync(i) []
            [Buffers.isEmpty(i)==false && IUT.isLocked()==false && (Buffers.isFull(i)==true || URLQueue.isEmpty()==true)] lock{IUT.doLock(); Buffers.setLocked(i,true)}
            -> BUL_sync(i) []¨//writeTree -> doneWrite{IUT.Write(Buffers.useFirst(i))} ->
            [Buffers.isLocked(i)==true && Buffers.isEmpty(i)==false] DuplicateCheck_sync(i) []
            [Buffers.isLocked(i)==true && Buffers.isEmpty(i)==true] unlock{Buffers.setLocked(i,false); IUT.doUnlock()} -> BUL_sync(i);
```

Figure 2. CT(i) main process.

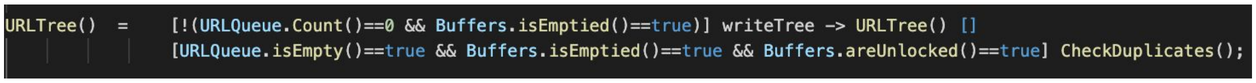
The second one, **CheckDuplicates( )**, is a simple process designed to verify if the model will ever get into our *error* state of duplicate. As you can see in Figure 3, this process calls the method we programmed into the IUT of *hasDuplicates(),* in case that it does, it goes into the error state and else it goes to a no_duplicates state and then skips. This process is called on the URLTree() process which is explained in the next paragraph.



```
CheckDuplicates() = [IUT.hasDuplicates()] error -> Skip []
                    [!IUT.hasDuplicates()] no_duplicates -> Skip;
```

Figure 3. CheckDuplicates() process for duplicate detection.

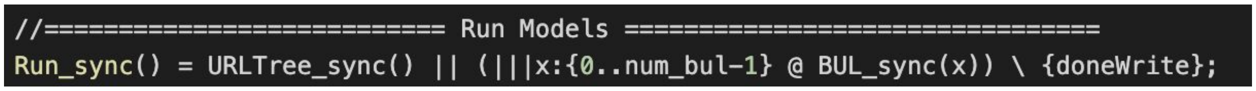
The final one, **URLTree( )**, is the process that runs synchronized with CT(i). This process has two general choices, as you may see in Figure 4. The first one, checks if there are still URLs to be written and that the BULs are not emptied, in that case, it writes into the tree whichever URL comes from the BUL to be emptied out. The other choice happens if there are no more URLs, if the BULs are unlocked and if the BULs are emptied, in that case it calls *CheckDuplicates( )* to verify that we didn't add any duplicates into our IUT.



```
URLTree()  =    [!(URLQueue.Count()==0 && Buffers.isEmptied()==true)] writeTree -> URLTree() []
                [URLQueue.isEmpty()==true && Buffers.isEmptied()==true && Buffers.areUnlocked()==true] CheckDuplicates();
```

Figure 4. URLTree() process to see if the IUT should be written or checked for duplicates.

To run our model, we called URLTree() in parallel and synchronized with as many BULs as we desire which run independently from each other. \{doneWrite} is called there too so that an unimportant state is muted out of the traces and we get the behaviour in our model we expect.

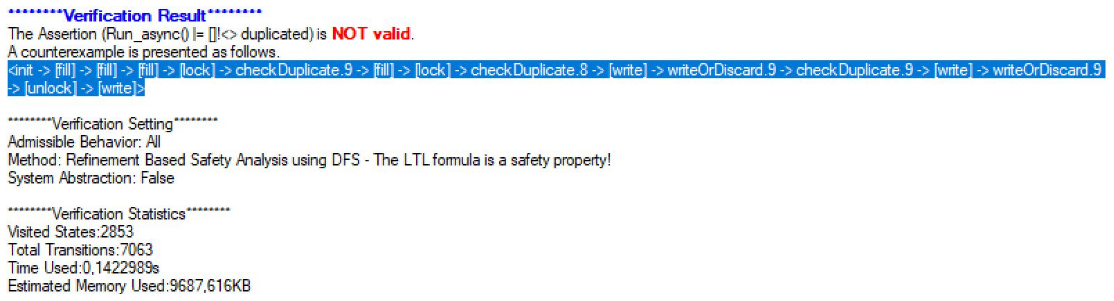


```
//========================== Run Models ===============================
Run_sync() = URLTree_sync() || (|||x:{0..num_bul-1} @ BUL_sync(x)) \ {doneWrite};
```

Figure 5. Run() main process to be executed.

2.4 Controlling the Java Scheduler by the use of agents:

In order to control the java scheduler based on a PAT trace we needed some way of transfering the trace information over to java. In the event that an assertion turns out to be false, PAT will automatically generate a trace string. This string is marked in the figure below. (Note that our newest version of the software uses threads 6,7,8 instead. The range/number of threads can easily be modified in the PAT model to support more than 3 threads if necessary.)



```
********Verification Result********
The Assertion (Run_async() |= []!<> duplicated) is NOT valid.
A counterexample is presented as follows.
<init -> [fill] -> [fill] -> [fill] -> [lock] -> checkDuplicate.9 -> [fill] -> [lock] -> checkDuplicate.8 -> [write] -> writeOrDiscard.9 -> checkDuplicate.9 -> [write] -> writeOrDiscard.9
-> [unlock] -> [write]>

********Verification Setting********
Admissible Behavior: All
Method: Refinement Based Safety Analysis using DFS - The LTL formula is a safety property!
System Abstraction: False

********Verification Statistics********
Visited States:2853
Total Transitions:7063
Time Used:0,1422989s
Estimated Memory Used:9687,616KB
```

This trace-string is provided to the java agent as a runtime parameter. The string can be copied over directly from PAT, without any alterations. This string is then parsed by the agent to extract the relevant information. The parsing and evaluating of the string results in an array of executions that are provided to the orchestrating method. This is a method defined in the agent, assigned with the task of controlling the execution flow of the program. An reference to this method is injected into the bytecode, by the agent, before the program is executed.

The orchestrating method relies on that the reading and writing operations can be locked and released. Calls to additional methods, defined in the agent, are also made to the reading and writing methods associated with the index building threads.

3. Implementation

3.1 Concurrent Java Web Crawler:

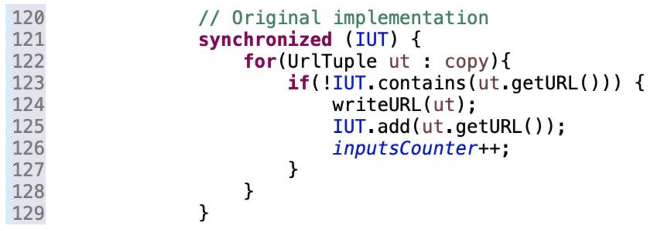
As mentioned above, we have implemented a modified instance of Consumer Producer concurrent design pattern. The only difference between our implementation and the original problem is that the consumer thread (building threads), rather than consuming one element from the shared buffer at the same time, empties it all elements at once. The crawling threads use a JAVA library JSOUP for establishing connection with the URLs to be visited and parsing their source code to retrieve the anchor tags. Stacks are used with each crawling thread to sequentially store the URLs to be crawled. Once the URL is visited and the source code retrieved, they are wrapped in an object called URLTuple which is then stored in the Buffered URL List of maximum capacity 1000, implemented using ArrayLists. When a consumer thread empties the buffer, HTML files are written to the disk and URLs are placed in Indexed URL Tree.

3.2 CSP# concurrency model with PAT:

In order to implement the web crawler problem in PAT, we had to be able to make an abstraction of the problem in order to make a successful model. To do so, we used random numbers instead of URLs, since the important part was to check for the insertion and the duplicates, but not of the nature of the URLs themselves. Also, we did not model the actual crawling procedure, since that was also irrelevant as for the concurrency properties to verify: no data-race and deadlock free design.

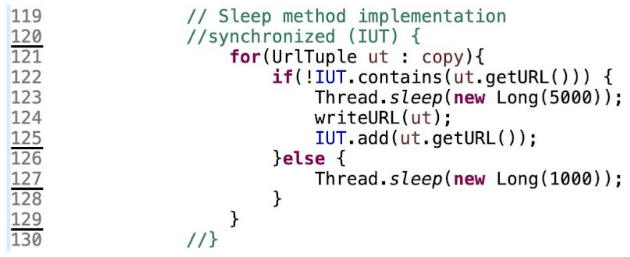
3.3 Incurring a data race by altering source code:

Two methods of manually changing the source code were used to control the java scheduler and display the concurrency bugs. One which utilized thread.sleep() - and one that used locking features. These methods alternated the behaviour of the index building threads. In particular, they altered the behaviour between reading from and writing to the IUT. The code segment associated with these operations can be seen below. Note the synchronized - keyword on line 121. The threads are locked wrt. The IUT, which is a shared resource. Therefore, only one thread is allowed to access the critical section at once. The IBT remains locked to the IUT until all elements are transferred. Moreover, if an element of the IBT is already present in the IUT that element won't be written.



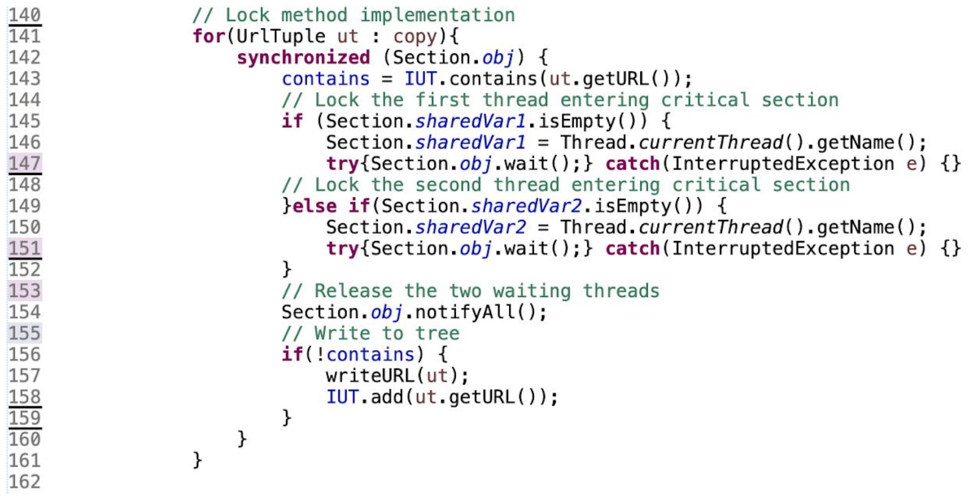
```
120            // Original implementation
121            synchronized (IUT) {
122                for(UrlTuple ut : copy){
123                    if(!IUT.contains(ut.getURL())) {
124                        writeURL(ut);
125                        IUT.add(ut.getURL());
126                        inputsCounter++;
127                    }
128                }
129            }
```

The the code for the tread.sleep() implementation can be seen in the following figure. First note that line 120 marks the beginning of the critical section. If the synchronized keyword is used the PAT model tells us there won't be any data races. Note however that line 120 is commented out, and therefore there is a possibility for a data race. The duplicate check is done on line 122 and the writing is done on lines 124 & 125. Note that a thread.sleep() operation is placed between the reading and the writing. This ensures that the other threads will have the time to also read the tree before the first thread gets the chance to write (provided line 120 remains commented out). In the scenario where two of these threads contain the same element and that element does not exist in the IUT beforehand, a data race will occur.



```
119            // Sleep method implementation
120            //synchronized (IUT) {
121                for(UrlTuple ut : copy){
122                    if(!IUT.contains(ut.getURL())) {
123                        Thread.sleep(new Long(5000));
124                        writeURL(ut);
125                        IUT.add(ut.getURL());
126                    }else {
127                        Thread.sleep(new Long(1000));
128                    }
129                }
130            //}
```

The critical section for the lock-implementation can be seen in the following figure. The check to see if the current value is already represented in the tree is done on line 143. The writing of the value is done on lines 157 & 158. Note the synchronized keyword on line 142, which has the effect of only letting one thread enter at a time. The String variables sharedVar1 & sharedVar2 are initially zero. Therefore, line 147 causes the first thread which enters the critical section to wait. Once it does so, the next thread is allowed to pass line 142. However, this thread is made to wait at line 151. This allows the third and final thread to enter the section. The third thread is allowed to pass all the way through to line 154, where it releases the other two threads. Note that at this time all threads have read from the IUT, yet none of them has written to it. Therefore, a data race will occur if the threads all carry the same value and that value was previously not represented in the IUT.



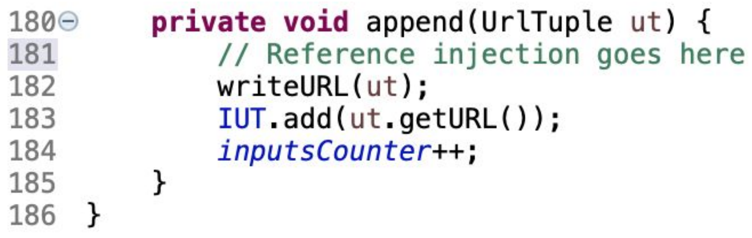
```
140            // Lock method implementation
141            for(UrlTuple ut : copy){
142                synchronized (Section.obj) {
143                    contains = IUT.contains(ut.getURL());
144                    // Lock the first thread entering critical section
145                    if (Section.sharedVar1.isEmpty()) {
146                        Section.sharedVar1 = Thread.currentThread().getName();
147                        try{Section.obj.wait();} catch(InterruptedException e) {}
148                    // Lock the second thread entering critical section
149                    }else if(Section.sharedVar2.isEmpty()) {
150                        Section.sharedVar2 = Thread.currentThread().getName();
151                        try{Section.obj.wait();} catch(InterruptedException e) {}
152                    }
153                    // Release the two waiting threads
154                    Section.obj.notifyAll();
155                    // Write to tree
156                    if(!contains) {
157                        writeURL(ut);
158                        IUT.add(ut.getURL());
159                    }
160                }
161            }
162
```

3.4 Incurring a data race by means of an agent:

The above implementations are sufficient from the perspective of displaying the data race. However, they have the undesirable effect of permanently altering the source code. A java agent was therefore implemented as a way of injecting these functionality alterations directly into the bytecode. This provides equivalent functionality to the solution in the preceding section, whilst maintaining the original functionality if the program is run without the agent. The agent was constituted of 3 files

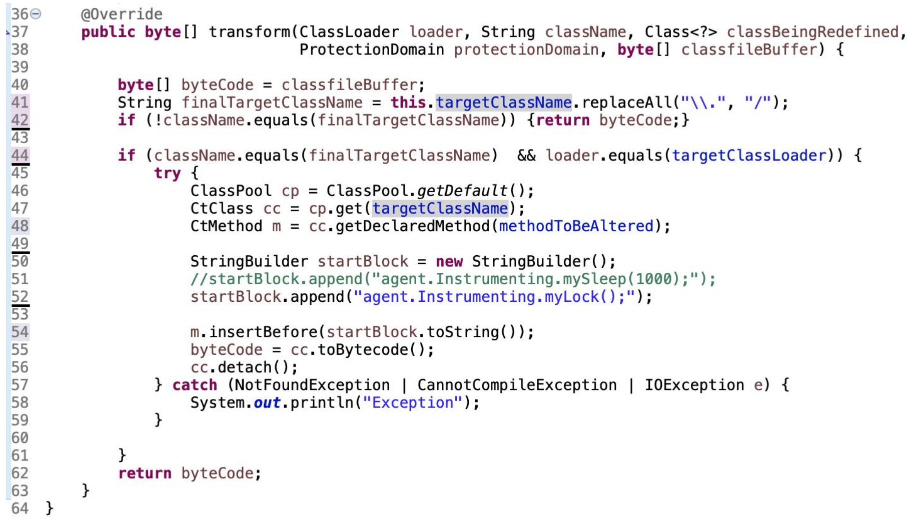
- **Instrumenting.java -** Contains methods that are to be injected
- **SleepAgent.java -** Contains the premain method and defines the methods to be altered
- **SleepTransformer.java -** Finds the methods to be altered and alternates their bytecode

In order to facilitate changes to the code, the write operation in the index building threads was encapsulated as a method according to the following figure. Note that the comment is included for illustration purposes only. It illustrates where the agent will inject a reference to an appropriate method in the Instrumenting class.



```java
    private void append(UrlTuple ut) {
        // Reference injection goes here
        writeURL(ut);
        IUT.add(ut.getURL());
        inputsCounter++;
    }
}
```

The Instrumenting class contains two methods, one for the sleep implementation and one for the lock implementation. The complete methods, which greatly resemble the ones in section 3.3, can be found in Appendix 7.1. The injection to these methods is performed by the SleepTransformer class, which implements the ClassFileTransformer interface. The code responsible for the injection can be seen in the following figure.

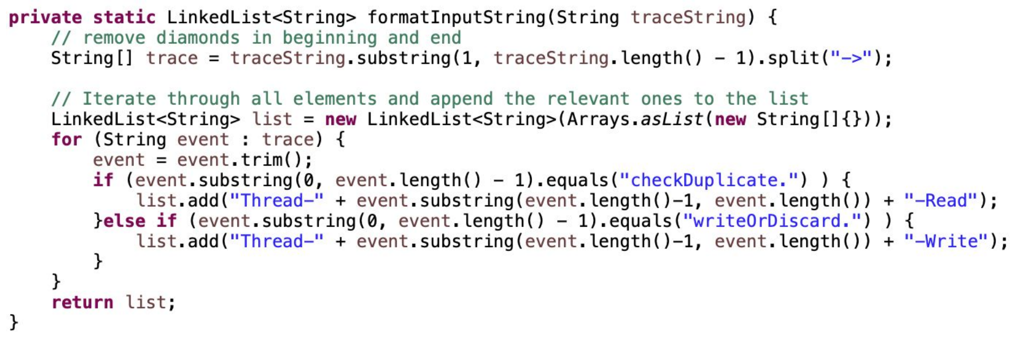


```java
36      @Override
37      public byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
38                              ProtectionDomain protectionDomain, byte[] classfileBuffer) {
39
40          byte[] byteCode = classfileBuffer;
41          String finalTargetClassName = this.targetClassName.replaceAll("\\.", "/");
42          if (!className.equals(finalTargetClassName)) {return byteCode;}
43
44          if (className.equals(finalTargetClassName)  && loader.equals(targetClassLoader)) {
45              try {
46                  ClassPool cp = ClassPool.getDefault();
47                  CtClass cc = cp.get(targetClassName);
48                  CtMethod m = cc.getDeclaredMethod(methodToBeAltered);
49
50                  StringBuilder startBlock = new StringBuilder();
51                  //startBlock.append("agent.Instrumenting.mySleep(1000);");
52                  startBlock.append("agent.Instrumenting.myLock();");
53
54                  m.insertBefore(startBlock.toString());
55                  byteCode = cc.toBytecode();
56                  cc.detach();
57              } catch (NotFoundException | CannotCompileException | IOException e) {
58                  System.out.println("Exception");
59              }
60
61          }
62          return byteCode;
63      }
64  }
```

The string *startBlock* is appended to the beginning of the append method. Changing the method to inject is as easy as changing which of the lines 51 & 52 is commented out. After the method has been altered, an array with the corresponding bytecode is returned. This is in turn used, instead of the bytecode associated with the original implementation, upon execution of the program.

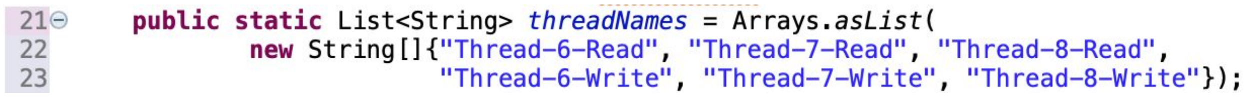
3.5 Controlling the Java Scheduler by altering source code:

The agent needed to be further improved to enable control of the scheduler based on a PAT trace. Firstly, the trace is supplied as a runtime parameter to the agent. It is parsed and converted into a useful List by the following code. The only parts of the trace that are being kept are the checkDuplicate and writeOrDiscard operations. These commands are then translated to the corresponding java-names before being inserted into the list.



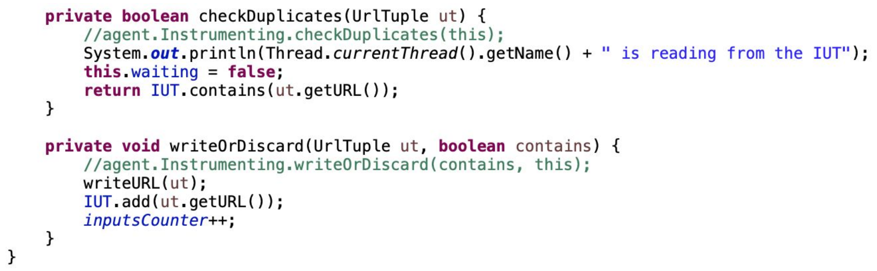
```
private static LinkedList<String> formatInputString(String traceString) {
    // remove diamonds in beginning and end
    String[] trace = traceString.substring(1, traceString.length() - 1).split("->");

    // Iterate through all elements and append the relevant ones to the list
    LinkedList<String> list = new LinkedList<String>(Arrays.asList(new String[]{}));
    for (String event : trace) {
        event = event.trim();
        if (event.substring(0, event.length() - 1).equals("checkDuplicate.") ) {
            list.add("Thread-" + event.substring(event.length()-1, event.length()) + "-Read");
        }else if (event.substring(0, event.length() - 1).equals("writeOrDiscard.") ) {
            list.add("Thread-" + event.substring(event.length()-1, event.length()) + "-Write");
        }
    }
    return list;
}
```

We chose to control read/write operations by locking the index building treads to elements of a shared array, which can be seen below. Each element in the array was associated with a particular operation of interest.



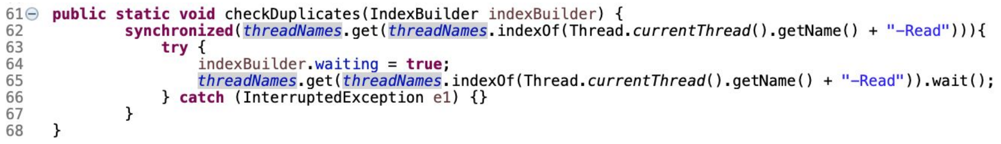
```
    public static List<String> threadNames = Arrays.asList(
            new String[]{"Thread-6-Read", "Thread-7-Read", "Thread-8-Read",
                         "Thread-6-Write", "Thread-7-Write", "Thread-8-Write"});
```

Functionality had to be added in order to control when the read/write operation, associated with a particular index building thread, would be performed. The read/write methods of the IBT class are shown below. The comments represent the injections that were made into the bytecode.



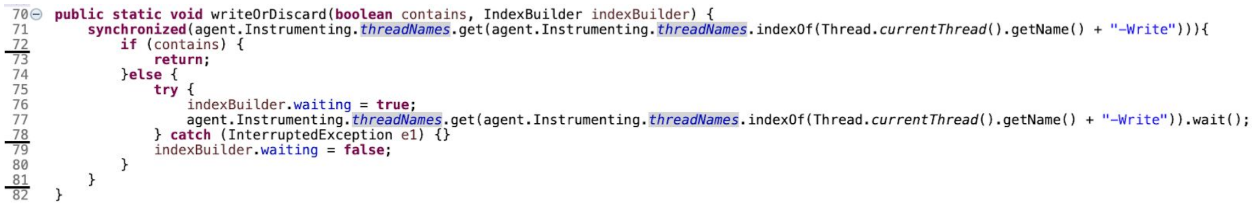
```
    private boolean checkDuplicates(UrlTuple ut) {
        //agent.Instrumenting.checkDuplicates(this);
        System.out.println(Thread.currentThread().getName() + " is reading from the IUT");
        this.waiting = false;
        return IUT.contains(ut.getURL());
    }

    private void writeOrDiscard(UrlTuple ut, boolean contains) {
        //agent.Instrumenting.writeOrDiscard(contains, this);
        writeURL(ut);
        IUT.add(ut.getURL());
        inputsCounter++;
    }
}
```

The implementation of the injected checkDuplicate method, written in the Instrumenting class, can be seen in the following figure. Note that line 64 causes the thread to wait on it's associated read-lock in the lock-list.



```
61  public static void checkDuplicates(IndexBuilder indexBuilder) {
62          synchronized(threadNames.get(threadNames.indexOf(Thread.currentThread().getName() + "-Read"))){
63              try {
64                  indexBuilder.waiting = true;
65                  threadNames.get(threadNames.indexOf(Thread.currentThread().getName() + "-Read")).wait();
66              } catch (InterruptedException e1) {}
67          }
68  }
```

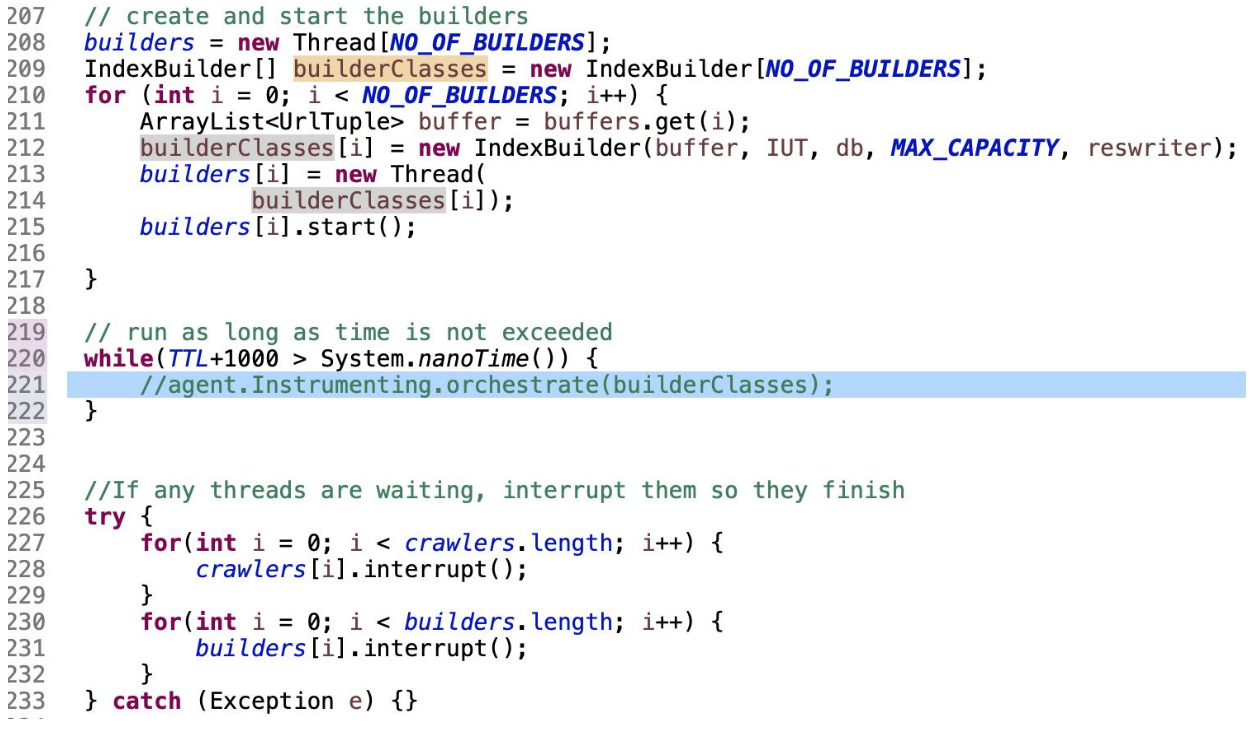
The implementation of the injected writeOrDiscard method, written in the Instrumenting class, can be seen in the following figure. Note that line 77 causes the thread to wait on it's associated write-lock in the lock-list.



```
70  public static void writeOrDiscard(boolean contains, IndexBuilder indexBuilder) {
71      synchronized(agent.Instrumenting.threadNames.get(agent.Instrumenting.threadNames.indexOf(Thread.currentThread().getName() + "-Write"))){
72          if (contains) {
73              return;
74          }else {
75              try {
76                  indexBuilder.waiting = true;
77                  agent.Instrumenting.threadNames.get(agent.Instrumenting.threadNames.indexOf(Thread.currentThread().getName() + "-Write")).wait();
78              } catch (InterruptedException e1) {}
79              indexBuilder.waiting = false;
80          }
81      }
82  }
```

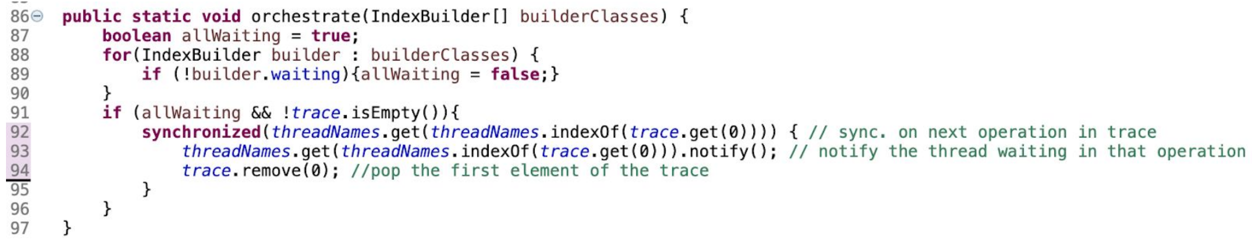
The effect of these locks will be that all the ITBs will go to a waiting state in the beginning of their read or write operations. By calling x.notify(), where x is an element of the lock-list, the associated thread will be allowed to resume it's execution. Once one read or write operation has been performed the thread will once again return to a waiting state.

To control which thread is executed when we make use of an orchestrating method. The call to this method is injected in the WebCrawlerDriver class. Specifically, it is injected in the loop that keeps track of the TTL of our system. The injection is demonstrated on line 221 below.



```
207   // create and start the builders
208   builders = new Thread[NO_OF_BUILDERS];
209   IndexBuilder[] builderClasses = new IndexBuilder[NO_OF_BUILDERS];
210   for (int i = 0; i < NO_OF_BUILDERS; i++) {
211       ArrayList<UrlTuple> buffer = buffers.get(i);
212       builderClasses[i] = new IndexBuilder(buffer, IUT, db, MAX_CAPACITY, reswriter);
213       builders[i] = new Thread(
214               builderClasses[i]);
215       builders[i].start();
216
217   }
218
219   // run as long as time is not exceeded
220   while(TTL+1000 > System.nanoTime()) {
221       //agent.Instrumenting.orchestrate(builderClasses);
222   }
223
224
225   //If any threads are waiting, interrupt them so they finish
226   try {
227       for(int i = 0; i < crawlers.length; i++) {
228           crawlers[i].interrupt();
229       }
230       for(int i = 0; i < builders.length; i++) {
231           builders[i].interrupt();
232       }
233   } catch (Exception e) {}
```

The actual implementation of the orchestrating method can be seen in the following figure. The allWaiting variable is a volatile variable which keeps track if all index building threads are in a waiting state. If we are in a situation where all threads are waiting we notify the thread which is waiting for the next operation of the trace. After this, wew pop that element off the trace.



```
public static void orchestrate(IndexBuilder[] builderClasses) {
    boolean allWaiting = true;
    for(IndexBuilder builder : builderClasses) {
        if (!builder.waiting){allWaiting = false;}
    }
    if (allWaiting && !trace.isEmpty()){
        synchronized(threadNames.get(threadNames.indexOf(trace.get(0)))) { // sync. on next operation in trace
            threadNames.get(threadNames.indexOf(trace.get(0))).notify(); // notify the thread waiting in that operation
            trace.remove(0); //pop the first element of the trace
        }
    }
}
```

4. Evaluation

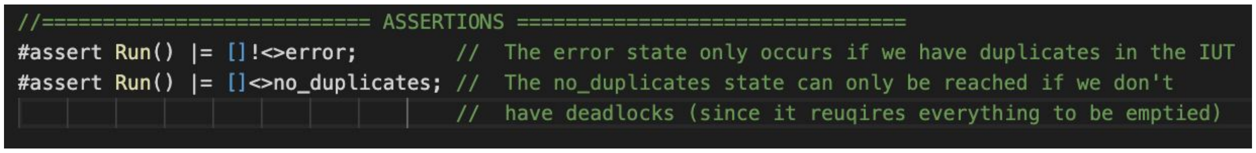
First of all, as has already been outlined in section 2.1, there are two issues with implementing a web crawling thread with a personal task stack of URLs to be crawled:

A. The URLs which have been originally placed at the top of the stack might never be visited.
B. When the stack runs out of URLs, the crawler becomes in a way dead, because it will have no tasks in its stack, and therefore will not do anything.

The first issue is not a problem since our web crawler is not required to visit all the URLs from the seed file and we only care in this project about the sheer value of visited web pages. The second one, however, is an issue that should be addressed, since a thread could become inactive in that case, however it still gains access to the critical section guarded by the BUL's intrinsic lock, which means it can . A possible improvement would be to populate the stack with some random (yet still valid) URL so that the thread again has something to work with. Another approach would be to save some part of the initial seed file for such emergency situations and fetch URLs from it whenever its stack becomes empty. Final solution would be to obtain URLs from another thread.

As far as testing is concerned, when run on the NCL, our web crawler managed to obtain a decent score of 70 000 visited websites per hour of testing.

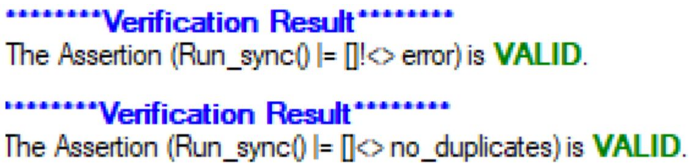
For our concurrency model in PAT, we wanted to prove our two main assertions.
Assertion 1: The IUT should not have duplicated URL (no data race).
Assertion 2: The system can always keep working until all the URLs have been found (no deadlock).
In order to verify this, we wrote in PAT the #assert statements shown in Figure 6.



```
//========================== ASSERTIONS ===============================
#assert Run() |= []!<>error;        //  The error state only occurs if we have duplicates in the IUT
#assert Run() |= []<>no_duplicates; //  The no_duplicates state can only be reached if we don't
                                    //  have deadlocks (since it reuqires everything to be emptied)
```

Figure 6. Assertions to be verified by PAT.

After iterating through different versions of our code and improving it, we finally got the desired VALID output for both of the assertions as seen in Figure 7, meaning that we have no data race and no deadlock.



```
********Verification Result********
The Assertion (Run_sync() |= []!<> error) is VALID.

********Verification Result********
The Assertion (Run_sync() |= []<> no_duplicates) is VALID.
```

Figure 7. Proof of assertions being valid for our model.

The scheduling of the concurrent java program can be conducted fully with the use of a PAT trace. We have not been able to spot any circumstances where the scheduler experiences difficulties.

5. Discussion

We believe that implementing a producer consumer design pattern, due to its simplicity and the fact that it is known to be an optimal solution to the problem involving threads reading and writing to the buffer, we have created a web crawler of optimal efficiency. Our web crawler maintains the concurrency invariant in which deadlock is impossible due to synchronization on the shared BULs objects using their intrinsic locks.

The big benefit of directly altering the source code, as we did in P3, is simplicity. It allows you to quickly verify a postulated behaviour related to the system, without introducing a major surrounding framework. There are quite a few issues related to directly altering the source code. One of them is the ability to quickly go back to the unmodified state. This can be solved with the use of version control systems, but it nevertheless introduces unwanted complexity. There are also difficulties associated with sharing the modifications. You could share a new version of the program. In the case of large software systems this is an unsatisfactory way of doing it. You might opt to only share the files containing the modifications, but there might be lots of small changes spread out across the software (an example is if you put logging statements all across your program).

These issues are all addressed with the use of java agents. The agents are lightweight - only containing the changes that will be injected into the software. It can therefore easily be distributed to stakeholders in the project. Thanks to the uniform nature of the JVM, the agent can be run by anyone having access to the original project. It’s also possible to quickly change from running the modified software to running the original software. The agent can even be dynamically linked, allowing the user to switch from the original code to the modified program within the same session. However, the agent comes at a price. Developing it means exposing oneself to far more complexity than simply modifying the source code. For small verification purposes, conducted in parallel with the development process, agents may bring to much complexity.

6. Conclusion

As for P2, it is important to recognize that some programs can be very complicated and might have a lot of possible bugs or errors that are important to keep in mind. Having tools like PAT are a great advantage since you can make an abstract representation of the model and test it to see if it ever reaches these undesired states, and if it does, find the trace of how it happens and then work in order to prevent it from happening.
Agents can effectively be utilized as a way of modifying existing software, without affecting the source code. This is particularly useful when you’re working on large-scale projects as it also gives a lightweight way of reliably producing the desired behaviour. Depending on one's needs,

java agents can be applied in both a static and dynamic way. Hence they can be injected into applications which are already running.

Reference:

[1] Process Analysis Toolkit User Manual, accessed on 10th of April 2020.

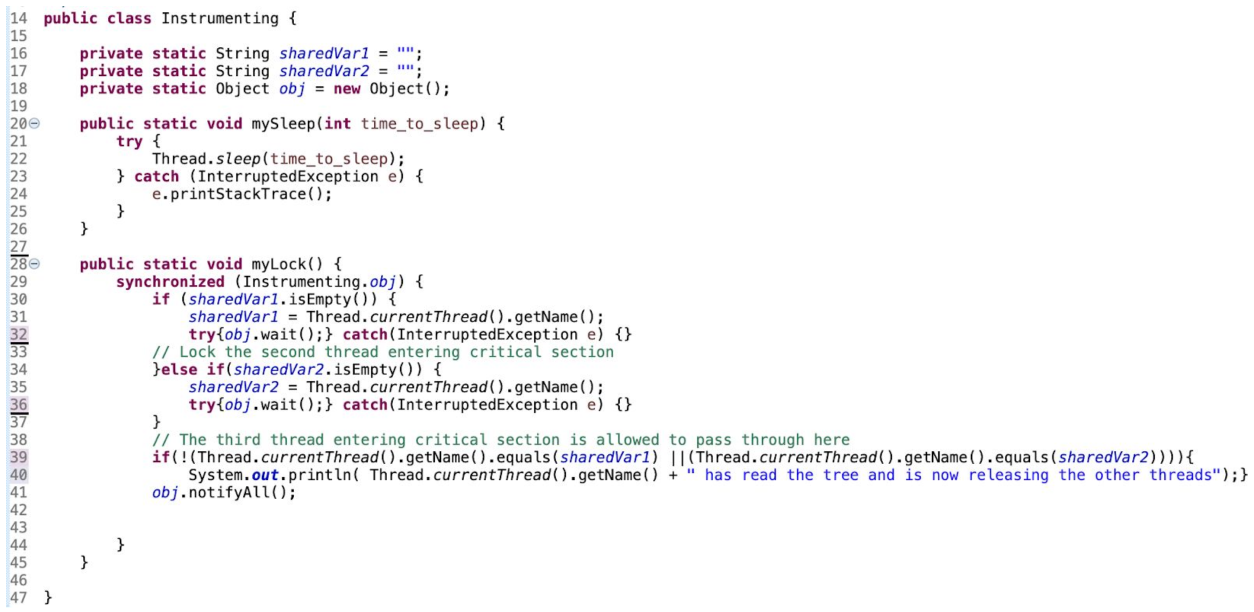
https://pat.comp.nus.edu.sg/wp-source/resources/OnlineHelp/htm/index.htm

[2]

7. Appendix

7.1 Incurring a data race by means of an agent:

The following figure shows the entire Instrumentation class. The two methods *mySleep* and *myLock* show great resemblance to the methods presented in section 3.3



```java
public class Instrumenting {

    private static String sharedVar1 = "";
    private static String sharedVar2 = "";
    private static Object obj = new Object();

    public static void mySleep(int time_to_sleep) {
        try {
            Thread.sleep(time_to_sleep);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void myLock() {
        synchronized (Instrumenting.obj) {
            if (sharedVar1.isEmpty()) {
                sharedVar1 = Thread.currentThread().getName();
                try{obj.wait();} catch(InterruptedException e) {}
            // Lock the second thread entering critical section
            }else if(sharedVar2.isEmpty()) {
                sharedVar2 = Thread.currentThread().getName();
                try{obj.wait();} catch(InterruptedException e) {}
            }
            // The third thread entering critical section is allowed to pass through here
            if(!(Thread.currentThread().getName().equals(sharedVar1) ||(Thread.currentThread().getName().equals(sharedVar2)))){
                System.out.println( Thread.currentThread().getName() + " has read the tree and is now releasing the other threads");}
            obj.notifyAll();


        }
    }

}
```