ISSN: 1118-5872

FOS MULTI-DISCIPLINARY JOURNAL

(Alvan Ikoku Federal University of Education)

A Systematic Analysis of Concurrency Issues in Java Applications

[1]Emelike Chinaenye D, [2]Emelike Ogechi V, [1]Okorie Simon C. (PhD) & [1]Osigwelem Kenneth U.
[1,3]Computer Science Department, Alvan Ikoku Federal University of Education, Owerri
[2]PACMedical Solutions Limited, Owerri

Abstract
Computers involve hardware, software made of principles and protocols that make it function appropriately. In industries and the academia, computer these days involves multicore technology which is making a radical change in the way software is being manufactured. It has been discovered by researchers (Pinto, Torres, Fernandes, Castor & Barros, 2015) that current state of development of concurrent programming in system is not very well known. Race conditions and deadlocks are the most common and serious concurrency problems in Java. They can lead to unforeseen data corruption, program crashes, and quiet failures in high-performance systems, shared resource access can result in non-deterministic behaviours, including missing updates when atomic operations fail, if it is not properly synchronized. The research presented evidence on how concurrent programming involving java programming language have motivated software developers to use more efficient technique of concurrent programming. This study used Systematic Literature Review (SLR) to expose some of the issues in concurrent programming which java programming language as a widely used object oriented programming language on such issues like deadlock, non-atomic operation, blocking, race conditions and starvation with sample code lines and proffered solution.

Keyword: Object-oriented Programming, concurrency Programming, Java Programming Deadlock, Race Condition

Introduction

Concurrent programming

Concurrency programming is a programming activity that involves technique used in dealing with lots of activities at once. It is the property of a system where several computations can be going on simultaneously and at the same time interacting with one another. This execution can be on multiple cores on the same chips tactically with time shared thread on the same processor or separate processors. Java has a number of ways for efficiently managing concurrency which is through multithreading (Ashok, 2025).

Concurrent programming is an important technique and at the same time challenging. It is Important in that it provides a way for effective use of distributed and parallel system and structure systems that perform many simultaneous task. While challenging in the aspect that the interaction between concurrent executing process can be unpredictable giving rise to many difficulties in the software development process (Jan & Anders, 2007) and (Melo, Carver, Souza, & Souza (2018)).

In order word concurrent programming is the execution of two or more autonomous interacting programs over the same time frame. Their processing can be interwoven or simultaneous and they are used in several small and large systems, Ashok (2025) present.

Concurrency in Java enhances performance and resource usage by enabling many threads to run concurrently. On the other hand, mishandling shared data might result in mistakes and

erratic behavior. It allows operations to be executed in parallel for improved performance as well as increases the efficiency and responsiveness of applications. It has to handle shared resources carefully to prevent problems. Concurrent programming puts demands on software debugging and testing, as concurrent software understanding of concurrency bugs, their frequency and fixing-times would be helpful. Similarly, to design effective tools and techniques for testing and debugging concurrent software, understanding the differences between non-concurrency and concurrency bugs in real-word software would be useful (Asadollah, Sundmark, Eldh & Hansson, 2017).may exhibit problems not present in sequential software, e.g., deadlocks and race conditions. In aiming to increase efficiency and effectiveness of debugging and bug-fixing for concurrent software, a deep

Review of Related Literature

Concurrency Programming and Object Oriented Programming (OOP)

Concurrency refers to the execution of multiple instruction sequences at the same time. It occurs in an operating system when multiple process threads are executing concurrently. These threads can interact with one another via shared memory or message passing. Concurrency results in resource sharing, which causes issues like deadlocks and resource scarcity. This has become an issue considering that today's technology uses multi-core processors and parallel processing, which allows multiple processes and threads to be executed simultaneously, accessing the same memory space, the same declared variable in code, or even read or write to the same file. Due to the unpredictable nature of process execution times, it is often impossible to determine the order in which processes will complete, and this non-determinism is a primary source of concurrency as it is found in object oriented programming - related challenges (Liberman, Gwendolyn, Katelyn, & Laura (2011)).

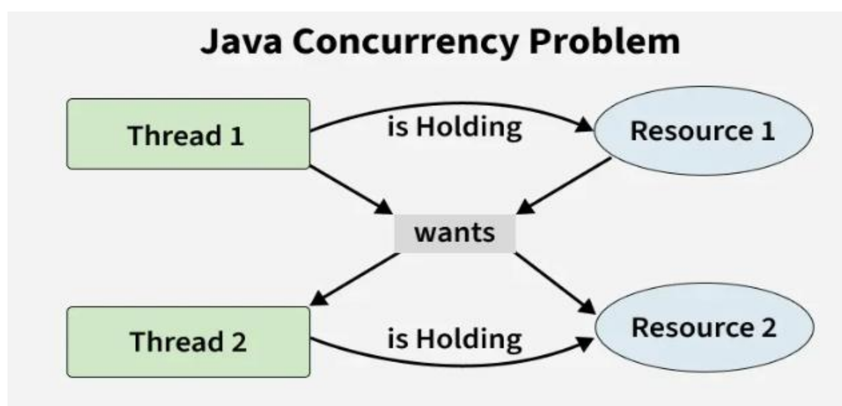

Figure 1: Concurrency Condition in Java

Object Oriented Programming (OOP)

Object Oriented Programming (OOP) is a core area in computer science. To learn OOP can be a major challenge in the field of computer science and computer education. On the other hand Java is a programming language and an Object Oriented Programming Language that has a widely used in teaching and learning OOP. It has also been observed that Java is somehow difficult to learn by students due to the concepts and principles like “encapsulation, inheritance and polymorphism. An example of OOP implementation used here is Java. There are many kinds

of OOP languages but Java is used here because of the earlier mentioned flexibility of Java programming language (Harel, Marron & Weiss, 2010).

Examples of the usage of concurrent programming

A typical computer at home or in the office can have several windows, were the user can run a text editor (notepad), another running browser and yet another video or music player and the clock can be running at the background all these execution can go on at the same time. Another instance is when a user uses MS Word and media player at the same time. Concurrency therefore is providing a way to structure solution to solve a problem that may not parallelized.

Issues of Concurrency

Various issues of concurrency are as follows:

1. Non-atomic

Several processes may happen issues. A non-atomic operation depends on other processes to run completely. Operations that are non-atomic are interruptible by multiple processes and can cause problems.

```
1 Thread 1::
2 if (thd->proc_info) {
3 fputs(thd->proc_info, ...);
4 }
5
6 Thread 2::
7 thd->proc_info = NULL;
```

In the issue above, we have two threads accessing the field proc_info based in the thd structure. In this case, if the value is a non-NULL, the value will be printed. The second thread is set to NULL. Hence if the first check is carried out the output will lead to deadlock before the output is carried out (fputs) as it attempts to run the second thread in between hence setting the pointer to NULL and on resuming the first thread, it can crash due to the NULL pointer referenced by the fputs.

Solution to Atomicity/Non-Atomic Bug

To solve the above concurrent issue, the need to add locks around the shared variable references being in mind that either of the access the proc_info field etc. considere the code below to solve the bug from non-Atomic bugs (Emmi & Enea, 2017):

```
1 pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3 Thread 1::
4 pthread_mutex_lock(&proc_info_lock);
```

```
5 if (thd->proc_info) {
6 fputs(thd->proc_info, ...);
7 }
8 pthread_mutex_unlock(&proc_info_lock);
9
10 Thread 2::
11 pthread_mutex_lock(&proc_info_lock);
12 thd->proc_info = NULL;
13 pthread_mutex_unlock(&proc_info_lock);
```

2. Race conditions –

A race condition occurs if the outcome depends on which of several processes gets to a point first.

In Java, a race problem occurs when the output of a program is determined by the timing or sequencing of other uncontrollable events.

Considering a marginally more difficult example; in the code below,

```
class Counter {
   int count = 0;

   void increment() {
      count++; // Not atomic: multiple threads may interfere
   }
}

public class GFG {
   public static void main(String[] args) throws InterruptedException {
      Counter c = new Counter();

      // Thread 1 increments count 1000 times
      Thread t1 = new Thread(() ->{
         for (int i = 0; i < 1000; i++) c.increment();
      });

      // Thread 2 increments count 1000 times
      Thread t2 = new Thread(() ->{
         for (int i = 0; i < 1000; i++) c.increment();
      });

      // Start thread 1
      t1.start();

      // Start thread 2
      t2.start();

      // Wait for thread 1 to finish
      t1.join();
```

```
        // Wait for thread 2 to finish
        t2.join();

        // May be < 2000 due to race condition
        System.out.println("Final Count: " + c.count);
    }
}
```

Output

Final count: 2000
Explanation:
Both threads increment the shared count variable concurrency.
Since count++ isn't atomic, intermediate updates can be lost, and result in incorrect final value.

The above code carryout simple insertion, but called by multiple thread at the same time lead to race condition. This race condition can be solve by surrounding the code with lock acquire and release;

```
1 void insert(int value) {
2 node_t *n = malloc(sizeof(node_t));
3 assert(n != NULL);
4 n->value = value;
5 pthread_mutex_lock(listlock); // begin critical section
6 n->next = head;
7 head = n;
8 pthread_mutex_unlock(listlock); // end critical section
9 }
```

OR using the method below:

```
1 void insert(int value) {
2 node_t *n = malloc(sizeof(node_t));
3 assert(n != NULL);
4 n->value = value;
5 do {
6 n->next = head;
7 } while (CompareAndSwap(&head, n->next, n) == 0);
8 }
```

3. Deadlock

In concurrent computing, dreadlock occurs when one group member waits for another member, including itself, to send a message and release a lock. It occurs when two processes are blocked and hence neither can proceed to execute, causing the system to freeze.

Deadlock in java is a programming situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when a thread (thread 1) is holding a lock (L1) and waiting for another lock (L2) and thread 2 is currently holding L2 and waiting L1 for usage, as shown in the code below:

```
Thread 1: Thread 2:
pthread_mutex_lock(L1); pthread_mutex_lock(L2);
pthread_mutex_lock(L2); pthread_mutex_lock(L1);
```

From the above code deadlock may occur when thread 1 seizes L1 and a switch occurs to thread 2. Thread 2 seizes L2 and tries to also secure L1. A deadlock will occur with each thread not being able to run.

DEADLOCK can arise if the following four conditions occur instantaneously (a very necessary condition):

Mutual exclusion ; two or more resources are non- shareable in which case only one process can use it at a time.

Hold and waits; when a process is holding at least one resource and waiting for resources

No pre-emption; A resources cannot be taken from a process unless the process releases the resources

Circular wait; when a set of processes are waiting for each other in circular form.

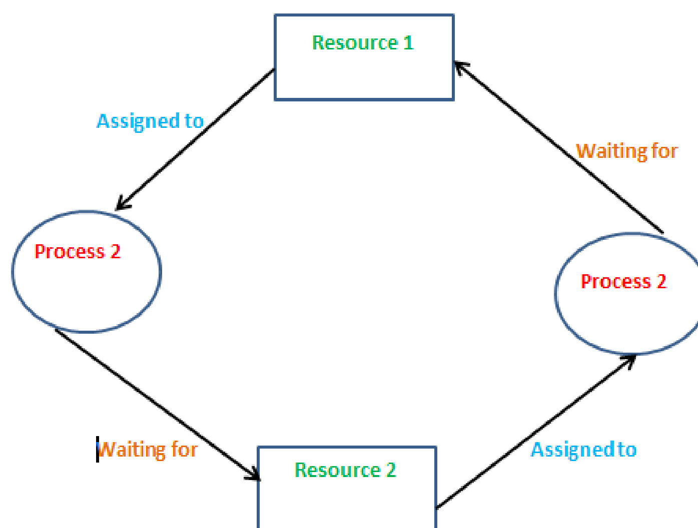

Figure 2: Pictorial representation of deadlock (self draw)

Solution to Deadlock Bug

1. Deadlock prevention or avoidance; The idea is to not let the system into a deadlock state .One can zoom into each category individually; prevention is done by negating one of the above mentioned necessary conditions for deadlock. Avoidance is kind of futuristic in nature. By using strategy of avoidance, we have to make an assumption. We need to insure that all information about resources which process will need are known to us prior to execution of the process (Cai & Cao .2016) .We use bankers algorithms (which is turn a gift from Diikstra)in order to avoid deadlock.
2. Deadlock detection and recovery; Let deadlock occurs , then do preemption to handle it once occurred.
3. IGNORE THE PROBLEM ALTOGETHER; If deadlock is very rare , then let it happen and reboot system . This is the approach that both Windows and UNIX take.

The hold-and-wait deadlock can be avoided by obtaining all locks at the same time atomically. This could be solved or prevented with the following:

```
1 pthread_mutex_lock(prevention); // begin acquisition
2 pthread_mutex_lock(L1);
3 pthread_mutex_lock(L2);
4 ...
5 pthread_mutex_unlock(prevention); // end
```

4. Blocking

A blocked process is waiting for some event, like the availability of a resource or completing an I/O operation. Processes may block waiting for resources, and a process may be blocked for a long time waiting for terminal input. If the process is needed to update some data periodically, it will be very undesirable. In Java, when we say that a thread blocks, we mean that the method (operation) that the thread calls, **put or take**, is blocking the thread from proceeding to execute the next line of code until some condition is met — the queue is **not full** or queue is **not empty**. See the code below:

```
package com.journaldev.concurrency;
import java.util.concurrent.BlockingQueue;
public class Consumer implements Runnable{
private BlockingQueue<Message> queue;
    public Consumer(BlockingQueue<Message> q){
      this.queue=q;
   }
   @Override
   public void run() {
      try{
         Message msg;
         //consuming messages until exit message is received
         while((msg = queue.take()).getMsg() !="exit"){
         Thread.sleep(10);
```

```
          System.out.println("Consumed "+msg.getMsg());
          }
        }catch(InterruptedException e) {
          e.printStackTrace();
        }
    }
}
```

5. Starvation

A problem in concurrent computing is where a process is continuously denied the resources it needs to complete its work. It could be caused by errors in scheduling or mutual exclusion algorithm. The code below is an example of starvation in Java.

```
class Starvation extends Thread {
 static int count = 1;
 public void run() {
  System.out.println(count + " Thread execution starts");
  System.out.println("Thread execution completes");
  count++;
 }
 public static void main(String[] args) throws InterruptedException {
  System.out.println("Parent thread execution starts");
  /* Priority of each thread given. */
      /* Thread 1 with priority 7. */
  Starvation thread1 = new Starvation();
  thread1.setPriority(7);
      /* Thread 2 with priority 6. */
  Starvation thread2 = new Starvation();
  thread2.setPriority(6);
      /* Thread 3 with priority 5. */
  Starvation thread3 = new Starvation();
  thread3.setPriority(5);
      /* Thread 4 with priority 4. */
  Starvation thread4 = new Starvation();
  thread4.setPriority(4);
      /* Thread 5 with priority 3. */
  Starvation thread5 = new Starvation();
  thread5.setPriority(3);

  thread1.run();
  thread2.run();
  thread3.run();
  thread4.run();
  /* Here thread 5 have to wait because of the
```

```
    other threads */
  thread5.run();

  System.out.println("Parent thread execution completes");
 }
}
```

Output of the above Code Will be:

Parent thread execution starts
1 Thread execution starts
Thread execution completes
2 Thread execution starts
Thread execution completes
3 Thread execution starts
Thread execution completes
4 Thread execution starts
Thread execution completes
5 Thread execution starts
Thread execution completes
Parent thread execution completes.

Summary

Issues of Concurrency programming include the following few:

Non-atomic: Operations that are non-atomic but interruptible by multiple processes can cause problems.

Race conditions: A race condition occurs of the outcome depends on which of several processes gets to a point first.

Blocking: Processes can block waiting for resources. A process could be blocked for long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable.

Starvation: It occurs when a process does not obtain service to progress.

Deadlock: It occurs when two processes are blocked and hence neither can proceed to execute.

Many techniques have been proposed to address concurrency bugs, including automated atomicity-violation repair, general automated concurrency bug fixing, the use of dynamic analysis, and maximum satisfiability approaches aimed at reducing the challenges associated with multithreaded programming ((Jin, Song, Zhang, Lu & Liblit (2011); Jin, Zhang, Deng Liblit & Lu (2012); Joshi, Naik, Sen & Gay, (2010); Weeratunge, D., Zhang, X., Jaganathan, S. However, existing studies indicate that many techniques for addressing concurrency bugs rely on the insertion of gate locks, either dynamically or statically, to enforce serialized thread execution during deadlock scenarios. While such approaches can mitigate immediate synchronization conflicts, prior research in concurrent systems has shown that excessive lock-based serialization may degrade performance and, in some cases, introduce secondary deadlocks due to increased

lock contention and poor lock ordering strategies. This limitation has been widely acknowledged in empirical studies on multithreaded Java applications.

In contrast, the work of Cai, Y. and Cao, L. (2016) proposes a novel approach, *DFixer*, which addresses deadlocks without introducing new ones by design. DFixer operates by selecting a single thread involved in a deadlock and enabling it to pre-acquire a lock *w* alongside another lock *h*. Typically, the deadlock arises when a thread holds lock *h* while waiting for lock *w*. By restructuring this sequence through pre-acquisition, the approach eliminates the hold-and-wait condition, which is a fundamental requirement for deadlock occurrence. Furthermore, the selected thread is carefully analyzed to ensure that no intermediate synchronization operations exist between the two lock acquisitions, thereby preserving execution correctness. Once applied, DFixer ensures that the resolved deadlock does not recur, removing the need for recovery-based techniques commonly used in traditional methods.

This approach aligns with a growing body of research that advocates for **prevention-based strategies** rather than **detection-and-recovery mechanisms** in concurrency control. Compared to conventional techniques such as dynamic analysis and automated bug-fixing frameworks, which often focus on identifying and patching issues post-occurrence, DFixer provides a more proactive and structurally sound solution (Cai & Cao, 2016).

Conclusion

In industries and the academia, computer these days involves multicore technology which is making a radical change in the way software is being manufactured. It has been discovered by researchers (Pinto, Torres, Fernandes, Castor & Barros, 2015) that current state of development of concurrent programming in system is not very well known.. they presented evidence on how concurrent programming involving java programming language have motivated software developers to use more efficient technique of concurrent programming. Studies has exposed some of the issues in concurrent programming which java programming language as a widely used object oriented programming language on such issues like deadlock, non-atomic operation, blocking, race conditions and starvation with sample code lines.

The research has looked at the types of issues in concurrent programming involving object oriented programming. First, we considered non-deadlock issues which are easier to control. They include atomic, race conditions, starvation and blocking. In which a sequence of instructions that should have been executed together was not, and order violations, in which the needed order between two threads was not enforced. Also briefly discussed deadlock: why it occurs, and what can be done about it. The problem is as old as concurrency itself, and many articles have been written about it. The best solution in practice is to be careful, develop a lock acquisition order, and so prevent deadlock from occurring. Wait-free approaches also have promise, as some wait-free data structures are now finding their way into commonly-used libraries and critical systems, including Linux. However, their lack of generality and the complexity to develop a new wait-free data structure will likely limit the overall utility of this approach. Perhaps the best solution is to develop new concurrent programming models: in systems such as MapReduce (from Google), programmers can describe certain types of parallel computations without any locks whatsoever. Locks are problematic by their very nature; perhaps

we should seek to avoid using them unless we truly must.

Recommendation:

Based on the reviewed literature, it is recommended that future research and practical implementations prioritize prevention-oriented techniques such as DFixer, particularly in complex multithreaded Java applications. However, further empirical validation across large-scale, real-world systems is necessary to evaluate its scalability, performance overhead, and integration with existing concurrency debugging tools. Additionally, combining DFixer with complementary approaches such as dynamic analysis could provide a hybrid solution that balances early prevention with runtime detection capabilities.

References

Abbaspour Asadollah, S., Sundmark, D., Eldh, S. & Hansson, H, (2017). Concurrency bugs in open source software: a case study. *J Internet Serv Appl* **8**, 4 (2017). https://doi.org/10.1186/s13174-017-0055-2

Ashok Lama (2025). "Multithreading in Java: Techniques for Concurrent Programming". INTERNATIONAL JOURNAL OF RESEARCH IN ENGINEERING & SCIENCE (IJRES) {ISSN- (PRINT) 2572-4274 (ONLINE) 2572-4304}, vol. 9, no. 2, 2025, pp. 81-85. DOI: https://dx.doi.org/10.5281/zenodo.15581906

Cai, Y & Cao, L.(2016). "Fixing deadlocks via lock pre-acquisitions," in *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*, New York, NY, USA: ACM, 2016, pp. 1109–1120. doi: 10.1145/2884781.2884819

Emmi, M., & Enea, C. (2017). Exposing Non-Atomic Methods of Concurrent Objects. *ArXiv, abs/1706.09305*.

Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor & Roberto S.M.Barros (2015). A large-scale study on the usage of Java's concurrent programming constructs. *Journal of Systems and Software* 2015 vol 106, pp 59 - 81

Harel, D., Marron, A., & Weiss, G. (2010, June). Programming coordinated behavior in java. In European Conference on Object-Oriented Programming (pp. 250-274). Springer Berlin Heidelberg. https://www.researchgate.net/publication/345087455 Object-Oriented Programming in Computer Science Chapter · January 2019 DOI: 10.4018/978-1-5225-7598-6.ch106 (retrieved 8/8/2022) https://www.researchgate.net/publication/228717393

Jan Lonnberg Jan & Berglund Anders (2007). Students' Understandings of Concurrent Programming, *Researchgate.net* (retrieved 08/08/2022)

Jin G., Song L. H, Zhang, W., Lu, S. & Liblit, B. (2011). Automated atomicity-violation fixing. In Proc. PLDI, 389–400, 2011.

Jin, G., Zhang, W., Deng, D., Liblit, B. & Lu, S. (2012). Automated concurrency-bug fixing. In Proc. OSDI, 221 - 236, 2012.

Liberman Benjamin. , Gwendolyn Seidman, Katelyn Y.A. McKenna, & Laura E. Buffardi (2011) Employee job attitudes and organizational characteristics as predictors of cyberloafing Computers in Human Behavior 27 (2011) 2192–2199

Melo, S. M., Carver, J. C., Souza, P. S., & Souza, S. R. (2018). Empirical research on concurrent software testing: A systematic mapping study. *Information and Software Technology*, *105*, 226–251. https://doi.org/10.1016/j.infsof.2018.08.017

Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh & Hans Hansson (2017). Concurrency bugs in open source software: a case study *Journal of Internet Services and Applications* **volume 8**, Article number: 4 (2017) (retrieved 8/8/2022)

Weeratunge, D., Zhang, X., Jaganathan, S. (2011). Accentuating the positive: Atomicity inference and enforcement using correct executions. In Proc. OOPSLA, 19–34, 2011

Zheng, L., Liao, X., Wu, S., Fan, X. and Jin, H.(2015). Understanding and identifying latent data races cross-thread interleaving. Frontiers of Computer Science (FCS), 9(4), 524–539, 2015.