IJCSE International Journal of Computer Sciences and Engineering Open Access
Survey Paper Vol.-7, Issue-6, June 2019 E-ISSN: 2347-2693

A Study on Race Condition & Dynamic Data Race Detection Techniques

Mithilesh Kumar Dubey[1], Devesh Lowe[2*], Bhavna Galhotra[3]

[1]Lovely Professional University, Punjab, India
[2,3]Jagan Institute of Management Studies, Rohini, New Delhi, India

Corresponding Author:devesh.lowe@jimsindia.org Tel.: 9810571097

DOI: https://doi.org/10.26438/ijcse/v7i6.4146 | Available online at: www.ijcseonline.org

Accepted: 14/Jun/2019, Published: 30/Jun/2019

***Abstract*:** Multithreaded programming has always presented a problem of race conditions which is one of the most common programming errors. If not handled properly, can lead to bugs with the potential to crash a system. A lot of work has been done in the past for detection of data races with a view to minimise the losses. Datarace can be detected at compile time (static race detection) and at runtime (dynamic race detection). This paper presents a study to understand the concept of parallel programming, race condition, semaphore, synchronization. We have also put in a detailed view on various techniques developed so far for dynamic data race detection.

Keywords: Parallel Processing, Race Condition, Semaphore, LockSet, Happens Before, Hybrid, Dynamic Data Race Detection

I. INTRODUCTION

With the increasing volume of processes and transactions on computing machines, there was an ever need to increase power and speed of our devices. The transition from sequential programming to parallel processing was slow but effective. This lead to growth of more operating systems and programming languages which support threads. Threads are lightweight and can be executed concurrently but leave a huge drawback that can be sometimes difficult to debug. It is evident that programming errors are frequent in large concurrent systems. Errors like deadlocks, starvation and race conditions have always been an area of concern for programmers and researchers working with multi-threaded programs. Simple errors in synchronized code may lead to race conditions which may turn out to be nightmare for programmers. We have included a discussion on parallel processing and Race condition in section 2 of this paper. There have been significant work done in area of race detection and synchronization. Race detection algorithms can be categorized in two broad areas as Static and Dynamic. Static approaches analyse the program source, while dynamic approaches analyse a trace or abstract-state representation generated by executing a program [1]. Data races can result in segmentation fault and deformation of data, therefore it is important to trace these data races by using any of the detection strategies. Some researchers classify race detection techniques in three categories as static, on-the-fly and post-mortem [2]. As per some researchers' view, algorithms that processes the programs event in parallel to execution are termed as on-the-fly detectors. Whereas writing event details in text files for a later review is called post-mortem strategy. Generally detecting race conditions statically is more demanding and requires overhead. There are different views presented for static race detection like of Flanagan [3], Boyapati et al. [4], Bacon et al. [5] etc. Static tools yield high coverage of shared resources by tracking all the possible situations of data races that might occur. Main benefit of static approaches is that they are highly efficient due to very limited runtime overhead. But they can be highly inefficient due to precision factor. Static approaches face a problem of high level of false positives and inefficiency due to scanning whole program. Precision of results is the area where most of Dynamic Data Race Detection algorithms score over static algorithms. Dynamic data race detection algorithms generate very few to no false positives. But they also face the problem of extra overhead and limited coverage of code. Most techniques are based on lockset based algorithms or happens before relations. We have included a discussion on crux of most of the Dynamic Detection techniques in section 3 of this paper. Our goal is to identify and present the work done by researchers in this field and prepare a detailed account of comparison on algorithms based on Happens before, Lockset or both.

II. PARALLEL PROGRAMMING & RACE CONDITION

In multi programming Operating Systems many process stay in main memory. OS is like resource manager which allocates shared resources to processes. Shared resources means that it is not personal resource to any process, all processes need to access it in a systematic manner. For e.g. printer is a sharable resource but it has to be used in non-sharable fashion i.e. mutually exclusive manner. We need a

policy or scheme so as to decide about how to use a shared resource. Let's consider an example case: In a program, a variable 'a' is assigned a value 10. Two processes p1 and p2 are using this shared variable 'a'. Let code of process as follows:

```
void p() {
read a;
a=a+1;
store a;
}
```

Figure 1: sample Code 1

Scenario 1: Let p1 and p2 both execute in a synchronized fashion i.e. first p1 executes and makes value of a as 11. Then p2 executes and makes value of a as 12. So final values of shared resource a is 12 which is universal now.

Scenario 2: Let's take another situation in which same two processes are executing but simultaneously i.e. process p1 starts, reads value of 'a' as 10, then context switch happens and control is transferred to p2 which reads value of 'a' as 10, makes it 11 and stores it back then exits. Now control is again transferred to p1 which starts from the point where it left i.e. it has value of 'a' as 10, it increments it makes it 11 and stores it back. Final value of shared variable 'a' is 11.
In first situation final value was 12 and in second situation final value was 11. This kind of problem is called as 'race condition'.

A. Race Condition

Race Condition commonly means that in case of shared resources, output changes if we change the order of execution of simultaneous processes. Or as defined by Savage et. al. [6] it is a situation which occurs when a common shared variable is accessed by two concurrent threads when at least one access is write, and no specific mechanism is used to synchronize access of shared space. When a process executes, there are different areas where it access its private and shared resources. Area where a process executes shared resources is called **Critical Section**. We observe that if two processes are accessing shared resource in systematic manner then they will not create a race condition. Problem (race condition) arises when they both access shared resource at the same time.

A Sample scenario on Race Condition: Let's observe the processes below, and try to find the total number of different results with value of B that can possibly come if both following codes (sample code 2) execute under race condition when B is a shared variable with initial value 2.

P1()	P2()
{	{
C=B-1;……1	D=2*B;…….3
B=2*C; ……2	B=D-1;……..4
}	}

Figure 2 Sample code 2

Above example will lead to 6 different permutations of statement execution with 3 different types of output. Following are 6 different cases of statement execution {1234, 3412, 1342, 3124, 1324, 3142} with output values of B as 3,4,2,3,2,2 respectively i.e. 3 different solutions just by changing the execution sequence of statements. In the above example, since both the processes are under execution in concurrent manner and both have write permission without mutual exclusion, this develops into a race condition giving three different types of outcomes.

B. Critical Section

It is that part of the code where process access shared resources. If a process tries to access critical section then it should not make our system inconsistent. There are a number of solutions possible for critical section problem like semaphore, Peterson's solution etc. Every solution is judged on the basis of three criteria namely Mutual Exclusion, Progress and Bounded Wait. First two are mandatory criteria which every solution to critical section problem must satisfy. Third criteria of Bonded Wait is though not mandatory but is desirable.

- Mutual Exclusion: (Mandatory Criteria) critical section must be accessed by one process at a time. i.e. in a mutual exclusive manner.
- Progress: (Mandatory Criteria) this clause specifies that only those processes which need to access the critical section, must be the processes in the race to access it. This eliminates the processes which do not want to access critical section. Effect of this clause is that the simplest method of 'Round Robin' is automatically eliminated since it will enforce all the processes to enter critical section according to their turn. If we do not enforce this clause then program logic will work but it will not be an efficient program since we are involving those processes in the race which do not need to. Similarly all the processes which are interested in accessing critical section should be given a fair chance.
- Bounded Wait: (Not mandatory but advisable) sometimes when processes are waiting in queue according to mutual exclusion, then they may end-up in a very long wait or starvation. So we can enforce this bounded wait option which makes sure that every process has access to resources after a particular time frame. But this criteria is not easy to maintain so it is advisable not mandatory.
-

C. Solutions to critical section problem

a) Two process solution with mutual exclusion: - Though in reality we need to perform *n* process solution, but let us understand two process solution first. In the following example code 3, we have two processes. P1 and P2 both running for infinite times.

P1	P2

While(1)
{
While(! flag=0);
Critical section;
Flag=0;
}
	While(1)
	{
	While(! flag=1);
	Critical section;
	Flag=1;
	}

Figure 3 Sample Code

We have used a global variable flag which varies between 0 and 1, and the process to enter critical section is decided by value of flag. P1 commences but faces a loop which goes infinitely if flag is 0. This means P1 will move to critical section only when it exits this loop which requires flag to be 0. Before exiting P1 sets value of flag to 1. P2 requires flag value to be 1 only then it will enter critical section. Here we also see that one process can context switch another process even in critical section but it will be of no use since other process was pre-empted so it will not change value of flag. Certainly it will not allow other process to enter critical section. Hence we can say that here both the processes are mutually exclusive which means it satisfies the first condition of mutual exclusion. But this solution does not satisfy the 'progress' criterion since we are following a strict alternation approach like round robin. What if p2 does not need to enter critical section, we are still forcing P2 to enter critical section. If p1 wants to enter critical section again, it has to wait for P2 to first enter CS, exit successfully change the value of flag and allow P1 to enter CS. So this solution is fulfilling the criteria of mutual exclusion but not of progression.

b) Two process solution with mutual exclusion and Progress: In the following example we can see that both the processes P1 and P2 are alternating but at the same time maintaining their own flags which indicate not only which process is inside critical section, but also their intent to enter the critical section. To implement this, we maintain an array of flags as flag [0] and flag [1] for both processes respectively.

P1	P2

While(1)
{
flag[0]=T
While(flag[1]);
Critical section;
Flag[0]=F;
}
	While(1)
	{
	Flag[1]=T
	While(Flag[0]);
	Critical section;
	Flag[1]=F;
	}

Figure 4: Sample code 4

Interesting thing is that both the processes are mutually exclusive and also they are following the system of progress i.e. only those processes are competing which are interested in entering critical section. Moreover if a process which has just exited the CS and wants to enter again, provided that other processes are not indicating to enter CS, it can surely enter the CS repeatedly. If they set their own flag value to true, they can enter critical section next time.

Problem: Given a situation where P1 is executing and immediately after statement (1), context switch happens and now P2 wants to enter critical section. Since statement (1) has already executed, so value of flag [0] is true which will not allow P2 to enter critical section. Now even though no process is in critical section, still both processes cannot enter it since flag [0] and flag [1] are both set to true. Hence clause of 'progress' fails and system moves into a **deadlock**.

c) Peterson's solution: Peterson's solution is implementation of observation of the previous two methods. We will now integrate both techniques of array of flags for processes to indicate their willingness to enter into CS, and a turn variable to control entry of processes in CS.

P1	P2

While(1)
{
flag[0]=T
turn=1;
While(turn==1 &&
flag[1]==T);
Critical section;
Flag[0]=F;
}
	While(1)
	{
	Flag[1]=T
	Turn=0;
	While(turn==0 &&
	Flag[0]==True);
	Critical section;
	Flag[1]=F;
	}

Figure 5: Sample code 5

In this case (ref Figure 5) whenever a process enter CS it sets turn to a value suitable to next process and checks a suitable condition build as a combination of willingness of other process to enter CS and their turn to enter CS. This will not only ensure mutual exclusion but also the rule of progression. Both the mandatory conditions ensure that if one process is interested then it can go for multiple entries. Also note that if both processes are interested, still they will not be entering into a deadlock even if context switch happens immediately after a process enters the procedure. Also to mention that if another process shows interest in entering the CS when first process is already in CS, it is allowed to enter immediately after the exit of first process. This way the third criteria of Bounded Wait is also satisfied by Peterson's algorithm. There is a limitation of Peterson's solution that it will only work efficiently for two processes but not for n processes.

d) Semaphores: Semaphores are simple integer variables which can give *n process* solution to multi-thread application trying to access shared resources. Apart from solving problems like critical section, semaphore can be used to solve problems like deciding the order of execution among processes or even resource management. A semaphore (let semaphore be *s initialized to 1*) is an integer variable which apart from initializing, is only accessed through two standard atomic operations wait(*s*) and signal(*s*). Wait(s) is a simple atomic operation which reduces the value of s by 1 whereas signal(s) increments the value of s by 1. Where signal works in simple manner as Signal(s) {s++ ;} but wait creates a condition like wait(s) {while(s<=0); s-- ;}. Now we can have n processes with embedded code for wait and signal. Any process p1 when wants to enter critical section, has to call wait(s) and when exiting CS has to call signal(s).

```
Process (P) {
  Wait(s)
     Critical Section
   Signal(s)
}
```

This ensures that value of s will be =0 whenever a process is inside CS, making CS unavailable to other processes. But once a process P1 moves out of CS it calls signal which again increments value of s to 1 hence clearing way for any other process *Pn*. This way semaphore satisfies conditions of mutual exclusion and progress.

After discussing all these techniques, it is also to be noted that detection of race condition is more important and fruitful than handling a race condition. Many scholars and researchers have contributed to this in form of developing several techniques of static and dynamic race detections. In following section we discuss all major work contributions in this area.

III. DYNAMIC DATA RACE DETECTION

Detecting a bug in a multi-threaded program can be difficult. A well-structured multi-threaded program, if starts producing bugs, may take weeks to decode. A lot of work has been done and published in the area of race detection. Race Detection algorithms are broadly classified in two categories i.e. Static Race Detectors and Dynamic Race Detectors. Dynamic race detection is on-the-fly approach where race condition is detected while execution of the program. A major advantage of dynamic data race detection is precision of work i.e. minimum probability of false positives. It also has a limitation that it doesn't cover all aspects of the program as compared to static race detectors. It can only detect data races while are reported in one instance of execution. So there is always a possibility of leaving out races undetected.

A pioneering work in field of early detection of race condition was done by Hoare [7] when they introduced the concept of 'monitor'. It is a group of shared variables, and a set of procedures that are required to access these variables. Both of these shared variables and procedures are packed together with a single lock which is automatically acquired and released at entry and exit of procedure. Monitors provide a cover to shared variables thereby making them unavailable and inaccessible to outside code. In other words monitors provide a compile time guarantee that accesses to shared variables are serialized and therefore free from data races. Monitors are generally considered effective when race conditions are to be detected at compile-time and all shared resources are declared in advance. But when it comes to dynamic nature of shared resources, more algorithms were required.

Most of the early work done in the field of Dynamic Race Detection is done based on Lamport's *Happens before Relations* [8]. Lamport gave an algorithm for solving the synchronization problems. However he concentrated majorly on spatially separated computers, but the work is appreciated for all multi-threaded applications evenly. His algorithm presents a concept of happens before relation in case of multiple processes based on time clocks for partial ordering of events. Premise of this algorithm is that if two processes happen to access a shared variable, then access is to be granted to the one which occurred first i.e. it happened before the other one. If events occur within a single a single process, then they are ordered according to their order of creation. In case of multi-threaded application, events are ordered according to the synchronization properties of the shared resource. Lamport also indicated that if two processes are accessing same resource without happens before ordering then there is a strong possibility of a data race. Savage et. al. [6] pointed out that though most dynamic race detectors are based on happens before algorithm, but they have recorded limitations. Firstly these algorithms are difficult to implement as they need to record a huge amount of thread specific information w.r.t. concurrent access to every shared location. Secondly it is very much dependent on scheduler of host system which generated interleaving event information.

Savage et. al. [6] presented a dynamic data race detection testing tool called 'Eraser' which claimed detection of more data races than 'happens before'. Eraser was aimed specifically at the lock-based synchronization. It followed a locking discipline, a programming policy, which ensured detection of more data races. Eraser uses a lockset refinement principle where it maintains separate set *C (v)* for each shared variable *v* containing all the possible locks of existing threads w.r.t. that shared variable *v*. As soon as any thread tries to access or acquire lock on any variable, it updates its set with the intersection of C (v) with the locks

held by the current thread. By this it ensures that any lock that protects v is contained in C (v). Eraser was tested on operating system kernels, Alta-Vista and Vesta Cache server and showed an increase in number of detected data races. But this method found out to be over conservative and also produced false alarms.

Muhlenfeld and Wotawa [9] suggested reducing number of false alarms of Eraser by user of polymorphic object destruction. Their work with polymorphic destruction of C++ objects reduced false alarms by 65%. Ali et al [10] suggested adding of happens before concept to Eraser algorithm. This resulted in reduction of number of false positives. Working with happens before is much slower than lockset based algorithm but when used in combination they can produce effective results. Bannerjee et.al. [11] Suggested more improvements over existing algorithms to detect more data races using limited access history.

Another key work was done by Choi et. al. [12] in their paper published in 2002 where they focussed on improving the performance of existing data race detection algorithms. Authors observed that most of the existing on-the-fly race detection algorithms either compromise on precision of algorithms to score on reducing extra overheads, or have to give in extra overhead just to be accurate. Problem of large number of false positives was observed, which was an extra burden. Choi et. al. worked on increasing the precision of datarace detection algorithms but without compromising on performance. Authors presented a concept of 'weaker than' relationship which they used to identify probable memory accesses which were redundant.

Robert O'Callahan and Jong-Deok Choi et al, [12] presented a model in 2003 which combined the advantages of both styles of dynamic data race detection i.e. lockset based and happens-before-based techniques. In the paper they called it Hybrid Dynamic Race Detector, which reduced overheads as compared to previous detectors particularly on large web servers. The algorithm was not purely based on happens-before, and presented a two stage approach to identify error prone program points and then focussing on those points of instrumentation. Metzger et. al. [13] in 2015 presented a unique approach where they suggested to integrate a race detector with debugger. Since it would substantially increase user work flow, authors proposed a method to reduce the overhead by allowing the user to define the scope of analysis. They extended the role of happens before algorithm and gave user the option to choose the parts of the program to check for data races. This kind of narrow analysis presented an efficient approach where false positives can be reduced.

Benjamin Wester et al. [14] suggested an improvement in speed of race detection by spreading the work over multiple cores. They used uniparallelism, to execute epoch in parallel to provide scalability, but executing all threads from a single epoch on a single core hence eliminated locking overhead. They implemented this uniparallelism strategy on two different algorithms of happen before style and lockset based detector and achieved results which were 4X faster as compared to original algorithms. Bodden et al. [1] presented a detection algorithm called *Racer*, an extension of *Eraser modelled on memory model of Java*, which was based on a language extension of AspectJ, an aspect oriented programming language based on java. They introduced three new point cuts as a language extension. It still faced a challenge of false positives, which they improved and re-implemented in 2010. Serebryany et al. [2] presented their algorithm called *Thread Sanitizer* for dynamic data race detection. *Thread Sanitizer* is a hybrid algorithm which used benefits of both happen-before and lockset. They proposed an API called dynamic annotations which as an extension of debugger, informs user about any synchronization errors.

K Leung et al [16] proposed a different perspective of race prevention called View Oriented Data Race Prevention (VODAP) to prevent data races in view oriented parallel programing model. It is a programming model cantered around view as a bundle of shared resources and data access. Authors used a memory protection mechanism available in UNIX with use of system calls such as mprotect(). When a view is acquired from mprotect(), only then any process is allowed to access that shared resource. Authors used this utility to trace any segmentation fault arising from any such asynchronous thread situation and handle it properly using like flashing error message to user, rather than crashing the system. Hence solving the problem of data race.

Baris kasikci et al, [17] in 2013, presented their unique algorithm as Race Mob which had low overhead and good accuracy. Proposed algorithm detects potential races statistically and dynamically traces whether they are true positives. Implementation of Race Mob was done on ten different systems to detect data races.

Cormac Flanagan et al, [18] in 2009, proposed an efficient and precise algorithm called Fast-track. Authors proposed to replace heavyweight vector clock with lightweight alternatives to support constant space and constant time operations. Fast-track was comparable to *Eraser* in terms of speed and showed speed in order of magnitude.

IV. CONCLUSION

In modern working environment it is hard to imagine an application which is not working on distributed system or multiple threads. Multithreaded or concurrent programs often show race conditions emerging as performance

hindrances. Though considerable amount of work has been done in area of dynamic data race detection, but problem of data race or race condition is one of the most common problem which still persists. In this paper we presented a study on various scholarly work done in the area of dynamic data race detection, and we acknowledge the quality of work done. But we also have to accept that this problem has still not faced a permanent solution.

REFERENCES

[1] K. H. Eric Baudden, "Aspect Oriented Race Detection in Java," IEEE Transactions on Software Engineering, 2010.

[2] T. I. Konstantin Serebryany, "ThreadSanitizer - Data Race Detection in Practice," Communications of ACM, 2009.

[3] S. N. F. Cormac Flanagan, "Type Based Race Detection for Java," ACM, 2000.

[4] M. R. C Boyapati, "A prameterized type system for race free java program," ACM, 2001.

[5] R. E. S. A. T. David F Bacon, "Guava: A Dialect of Java without datarace," ACM, 2000.

[6] M. B. G. N. P. S. t. A. Stefan Savage, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," ACM Transactions on Computer Systems, vol. 15, no. 4, pp. 391-411, 1997.

[7] C. A. R. Hoare, "Monitors: An Operating Systems Structuring Concept," Communications of ACM, vol. 17, no. 10, 1974.

[8] L. Lamport, "Time, Clocks, and the ordering of Events in a Distributed System," Communications of ACM, vol. 21, no. 7, 1978.

[9] F. W. Arndt Muhlenfeld, "Runtime race detection for multi-threaded C++ server applications," ACM Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering, 2007.

[10] K. B. V. P. W. T. Ali Janessari, "Helgrind+: An efficient dynamic race detector," 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, , 2009.

[11] Z. M. B. B. P. P. Utpal Banerjee, "A Theory of Data Race Detection," Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging, 2006.

[12] K. L. A. L. R. O. V. S. M. S. Jong-Deok Choi, "Efficient and Precise Datarace Detection for Multithreaded Object Oriented Programs," ACM, 2002.

[13] X. T. W. T. Markus Metzger, "User Guided Dynamic Data Race Detection," International journal of Parallel Programming, 2015.

[14] D. D. P. M. C. J. F. S. N. Benjamin Wester, "Parallelizing Data Race Detection," ACM, 2013.

[15] T. I. Konstantin Serebryany, "ThreadSanitizer: data race detection in practice," ACM, 2009.

[16] Z. Q. P. W. K Leung, "Data Race: tame the beast," Springer J Supercomput, 2010.

[17] C. Z. G. C. Baris Kasikei, "RaceMob: Crowdsouced Data Race Detection," ACM, 2013.

[18] S. N. F. Cormac Flanagan, "FastTrack: Efficint and Precise Dynamic Race Detection," ACM, 2009.

[19] K. H. Eric Boden, "Racer: Effective Race Detection using AspectJ," ACM, 2008.

Authors Profile

Mithilesh Kumar Dubey currently working as Associate Professor in Lovely Professional University, Jalandhar Punjab. Mithilesh does research in Computer Communications (Networks)

Devesh Lowe is currently working as Assistant Professor with JIMS, Sector-5 Rohini, New Delhi for past 5 years. Before joining at this profile, he had a work experience of 10 years in IT Industry and Academia. After starting his career in software development and Marketing for 2 year, he switched to academia and was associated with many colleges and universities under different profiles. He is a keen researcher and has published various papers in field of E-Learning, Computer Networks and cloud computing. His current areas of research include Sentiment Analysis and Dynamic data race detection.

Ms. Bhavna Galhotra is associated with Jagan Institute of Management Studies as an Assistant Professor since 8 years. She is Pursing Ph.D. and completed MTECH (IT) from USICT, GGSIPU. She has also done MCA from GGSIPU and is a graduate from Delhi University. She has also completed DOEACC 'O' and 'A' level Diploma's from Delhi University. Overall she has an experience of more than 10 years. Her research areas are Network Security commerce and Cloud Computing. She has published papers in International Journals on Computer Networks, Network Security, and Cloud Pay as per use system.

