ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Special Issue 7, October 2015

Detecting Data Race Error using Enhanced Lockset Algorithm

Suchetha Vijaykumar[1], Roopa Karani[2], Shrushti Avatagi[2]

Asst. Professor, Dept. of MCA, AIMIT, St. Aloysius College (Autonomous), Mangalore, Karnataka, India[1]

Students V Semester, Dept. of MCA, AIMIT, St. Aloysius College (Autonomous), Mangalore, Karnataka, India[2]

ABSTRACT: It is easy to make a mistake in synchronization that produces a data race, yet it can be extremely hard to locate this mistake during debugging. It describes a new tool, called Eraser, for dynamically detecting data races in lock-based multithreaded programs. Eraser uses binary rewriting techniques to monitor every shared-memory reference and verify that consistent locking behavior is observed. The algorithm checks if two accesses to a variable are ordered by a happens-before relation. We can handle interesting cases including object initialization, thread-locality, and dynamically changing locksets over time. In this algorithm we are proposing a new dynamic lockset algorithm that detects race conditions from execution traces of concurrent programs.

KEYWORDS: Eraser, Multithreaded programs, Lockset, Data race**.**

I. INTRODUCTION

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. A thread is a light-weight process. In particular, the threads of a process share the latter's instructions (its code) and its context (the values that its variables reference at any given moment).. Most commercial operating systems support threads, and popular applications like Microsoft Word and Netscape Navigator are multithreaded. Unfortunately, debugging amltithreaded program can be difficult.

A data race occurs in a multithreaded program when two threads access the same memory location without any intervening synchronization operations, and at least one of the accesses is a write. Data races almost always indicate a programming error and such errors are notoriously difficult to find and debug, due to the non-deterministic nature of multithreaded programming. Furthermore, since a data race typically results in corrupted shared data structures rather than an immediate crash, programs may continue to execute, leading to mysterious failures later in unrelated code. Automatically finding data races in a program thus is widely recognized as an important research problem.

II. LITERATURE SURVEY

Automated tools for race detection generally take either a static or a dynamic approach.An early attempt to avoid data races was the concept of a monitor. Monitor is nothing but what we call 'lock' in multi-threading. It is a way of controlling the area in which only desired number of threads (mostly one) can enter.

A Monitor is an object designed to be accessed from multiple threads. The member functions or methods of a monitor object will enforce mutual exclusion, so only one thread may be performing any action on the object at a given time. If one thread is currently executing a member function of the object then any other thread that tries to call a member function of that object will have to wait until the first has finished.Thus monitors provide a static, compile-timeguarantee that accesses to shared variables are serialized and therefore free from data races.The primary drawback of this approach is excessive false positives (reporting a potential data race when none exists), since the compile-time analysis is often unable to determine the precise set of possible thread interleaving's and thus must make a conservative estimate. Scaling is also difficult, since the entire program must be analyzed.

ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Special Issue 7, October 2015

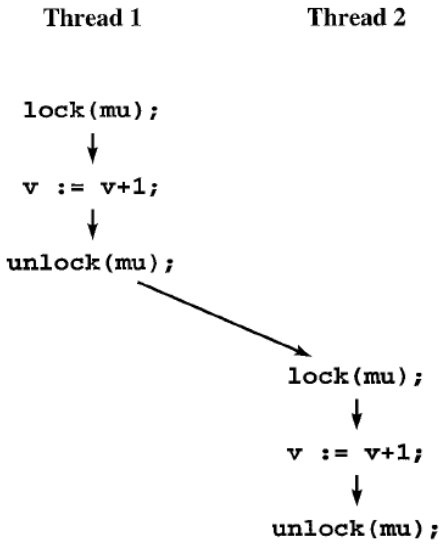
For dynamic race detection,Thehappens-before order is a partial order on all events of all threads ina concurrent execution. The happened-before relation (denoted:⟶) is a relation between the result of two events, such that if one event should happen before another event, the result must reflect that hb(r, w) means that r is executed before w AND w can see the result of r.

In a single thread, happens-before reflects the temporal order of event occurrence.Between threads, A happens before B if A is a lock access in one thread, and the next access to that lock (event B) is in a different thread and if the accesses obey the semantics of the lock (can't have two successive locks, or two successive unlocks, or a lock in one thread and an unlock in a different thread)

• Let event a be in thread 1 and event b be in thread 2.

– If a = unlock(mu) and b = lock(mu) then
a → b(a happens-before b)

• Data races between threads are possible if accesses to shared variables are not ordered by happens-before.



The arrows represent happens-before.The events represent an actual execution of the two threads.Instead of a logical clock, each thread might maintain a "most recent event" variable. In T1, the most recent event is unlock(mu); when T2 executes lock(mu) the system can establish the happens-before relation.The lock of mu by Thread 2 is ordered by happens-before with theunlock of mu by Thread 1 because a lock cannot be acquired before itsprevious owner has released it. Finally, the three statements executed byThread 2 are ordered by happens-before because they are executed sequentiallywithin that thread.

If two threads both access a shared variable, and the accesses are notordered by the happens-before relation, then in another execution of theprogram in which the slower thread ran faster and/or the faster thread ranslower, the two accesses could have happened simultaneously; that is, adata race could have occurred, whether or not it actually did occur. Allprevious dynamic race detection tools that we know of are based on thisobservation. These race detectors monitor every data reference and synchronizationoperation and check for conflicting accesses to shared variablesthat are unrelated by the happens-before relation for the particularexecution they are monitoring.Unfortunately, tools based on happens-before have two significant drawbacks.

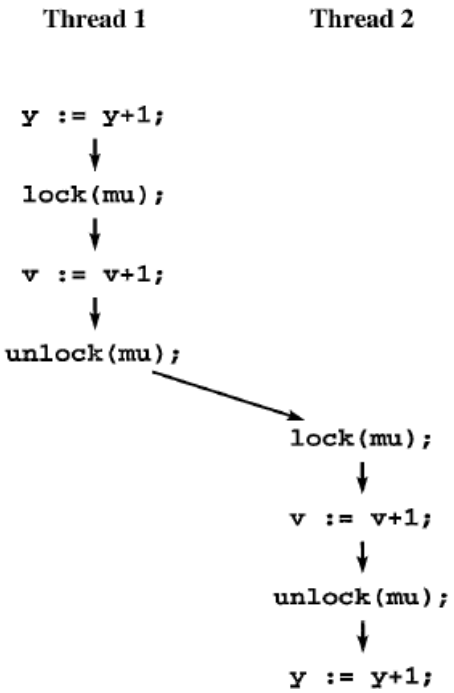
First, they are difficult to implement efficiently because they requireper-thread information about concurrent accesses to each shared-memorylocation. More importantly, the effectiveness of tools based on happensbeforeis highly dependent on the interleaving produced by the scheduler.

ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Special Issue 7, October 2015



If Thread 2 executes before Thread 1, happens-before no longer holds between the two accesses to y, so the possibility of a data race occurs and should be notified to the programmer

A tool based on happensbeforewould detect the error only if the scheduler produced an interleaving in which the fragment of code for Thread 2 occurred before the fragment of code for Thread 1. Thus, to be effective, a race detector based on happensbeforeneeds a large number of test cases to test many possible interleaving.

III. LOCKSET ALGORITHM

Lockset algorithm is a classic runtime race detection algorithm which is implemented in the eraser tool. The lockset algorithm ensures that simple locking discipline on the shared variables is maintained i.e. each shared variable should be protected by a lock that is held by the thread whenever the variable is accessed. The shared variables are protected by the same lock and the thread should hold the corresponding locks whenever they access the shared variable. The main aim of the algorithm is to make sure that this locking discipline is followed by monitoring the read and write access to the variables as the thread executes.

a. ALGORITHM

For each shared variable x, the algorithm maintains the set lockSet(x) of candidate locks. Candidate locks are all the possible locks that can be held on variable x to protect the variable. lockSet(x) is initialized when a variable is created, it is initialized to contain all locks. A lock l is in lockSet(x) if every thread that has accessed x was holding lock l at the time of access. The locks held by a thread is denoted by locksHeld(t), It contains set of locks that are currently held by the thread t. When a thread t accesses x, the lockset presumed to protect the data is updated by lockSet(x) :=lockSet(x) NlocksHeld(t) (The lockset is reduced to the intersection of the previous lockset and the set of locks held by the thread on the current access). This process, called lockset refinement, ensures that any lock that consistently protects x is contained in lockSet(x). lockSet(x) is reduced to the intersection of the previous lockSet(x) and the set of locks held by the thread locksHeld(t). This progressively removes locks from the lockSet(x). If the lockSet(x) ever becomes empty an error is reported because no lock protects access to the data. If some lock l consistently protects x, it will remain in lockSet(x).

Let locksHeld(t) be the set of locks held by thread t.

For each x, initialize lockSet(x) to the set of all locks.

ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Special Issue 7, October 2015

On each access to x by thread t,
setlockSet(x) := lockSet(x) Nlocks_held(t);
iflockSet(x) = { }, then issue a warning;

b. IMPLEMENTATION

Program	locksHeld(t)	lockSet(x)
INT X;	{}	{mu1,m2}
X:=1024;		
Lock(mu1);		
	{mu1}	{mu1}
X:=X+1;		
Unlock(mu1);		
	{}	
Lock(mu2);		
	{mu1}	{} - Issue warning
X:=X+1;		
Unlock(mu2);		
	{}	

c. MAJOR DRAWBACKS

1. The major drawback of lockset algorithm is that it produces too many false alarms.
2. It only works with multithreaded programs using lock-based synchronization primitives.
3. Lockset algorithm falsely reports races, even though they are prevented by the barriers as it is unaware of casual ordering of events. This drawback can be overcome by integrating the happens before algorithm and the lockset algorithm.
4. Another major drawback is, the candidate lockSet of the variable becomes empty. This happens because the lockSet(x) is modified whenever a thread accesses x. When a thread accesses x the lockSet(x) is refined, and the locks from lockSet(x) are progressively removed making it empty.

IV. THE PROPOSED ALGORITHM

The following proposed algorithm overcomes a drawback of lockset algorithm i.e. empty lockSet(x) after refinement. The algorithm basically works on the lockset algorithm. Few modifications have been made to the lockset algorithm to ensure that lockSet(x) does not become empty, and the variable x is protected by a lock which is present in lockSet(x). In this algorithm we use thread set threadSet(x), which is a set that contains information about threads that hold lock on

ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Special Issue 7, October 2015

variable x and we also use C(x), to store a duplicate copy of the initial candidate lockSet(x). The lockSet(x) and locksHeld(t) are initialized as they are initialized in lockset algorithm. The refinement on lockSet(x) is done the same way as in lockset algorithm.

a. ALGORITHM

In this algorithm whenever a thread t accesses x, information about the thread is stored in ThreadSet(x) by ThreadSet(x) := threadSet(x) U Thread(t) and the lockset presumed to protect the data is updated by lockSet(x) :=lockSet(x) NlocksHeld(t) (The lockset is reduced to the intersection of the previous lockset and the set of locks held by the thread on the current access). This progressively removes locks from the lockSet(x). if lockSet(x) is empty but threadSet(x) is not empty i.e. a thread is accessing the variable then it is an error, so to prevent this the lockset is updated by lockSet(x) :=C(x) NlocksHeld(t). Still if the lockset(s) is empty then report an error.

Let locksHeld(t) be the set of locks held by thread t.
Let threadSet(x) contain information about threads that currently access x.
For each x, initialize lockSet(x) to the set of all locks.
C(x)=lockSet(x)
On each access to x by thread t,
setlockSet(x) := lockSet(x) N locks_held(t);
setthreadSet(x) := threadSet(x) U Thread(t);
if (lockSet(x) = { } and threadSet(x) !={ } } then
setlockSet(x) := C(x) N locks_held(t);
if (lockSet(x) = { } and threadSet(x) !={ } } then
issue warning;

b. IMPLEMENTATION

Program (Thread t1)	**locksHeld(t)**	**lockSet(x)**	**threadSet(x)**
INT X;	{}	{mu1,m2}	
X:=1024;			
Lock(mu1);			
	{mu1}	{mu1}	{t1}
X:=X+1;			
Unlock(mu1);			
	{}		
Lock(mu2);			
	{mu1}	{}	{t1}
		{mu1}	
X:=X+1;			

ISSN(Online) : 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Special Issue 7, October 2015

Unlock(mu2);	{}		

c. BENEFITS OVER LOCKSET ALGORITHM

The proposed algorithm overcomes the drawback of the candidate lockset of a variable becoming empty. By ensure that the candidate lockset does not become empty we ensure that the variable is always protected by a lock when it is being accessed.

VII. CONCLUSION

This paper has described the lockset algorithm in details. It also gives the pros and cons lockset algorithm. Lockset algorithm is widely used and implemented to detect data race errors. But lockset algorithm too has certain drawbacks and one the drawbacks is overcome by the proposed algorithm described in this paper.

REFERENCES

1. DETLEFS, D. L., LEINO, R. M., NELSON, G., AND SAXE, J. B. 1997. Extended static checking.Tech. Rep. Res. Rep. 149, Systems Research Center, Digital Equipment Corp., Palo Alto,Calif.
2. DINNING, A. AND SCHONBERG, E. 1991. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging. ACM SIGPLAN Not. 26*, 12 (Dec.), 85–96.
3. HOARE, C. 1974. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct.), 549–557.
4. STEFAN SAVAGE University of Washington MICHAEL BURROWS, GREG NELSON, and PATRICK SOBALVARRO Digital Equipment Corporation and THOMAS ANDERSON University of California at Berkeley