

Concurrency and Synchronization: Detection, Reasons, Tools and Applications

Govind Khandelwal, Shriram Sonwane, Sachin Ware

Department of Electronics and Telecommunication Engineering
Vishwakarma Institute of Information Technology, Kondhwa bk, Pune

Abstract- Concurrency and Synchronization in digital electronics where algorithms are used to comprehend all the calculations for work. Digital machines ranging from Embedded Systems, IOT, Computers, Smartphones, Servers and Networking systems. Synchronization has become a very crucial part of basic programs running in the background of any operating system, that is the “Kernel”. These algorithms are the basic part of the OS for its smooth working in multi-programming, load balancing, time synchronization, data I/O ops within and out of the system, parallel computing with GPUs, I/O ops with IOT and cloud systems, Network and data security, mathematical calculations, etc. Synchronization programs are used to prevent conditions such as data races, deadlock, network latency, data corruption, manipulation and many more. Conditions created by these bugs can be visible or invisible in the user space. This Research paper is a comprehensive analysis on Concurrency and Synchronization. Source code examples of such conditions are given below from the original source code of some of the common linux distros. Applications of solutions to some of these issues in programs and systems to help progress for development of the performance and results.

Index Terms- Detection, Reasons, Tools and Applications

I. INTRODUCTION

Concurrency [1][5][7] is where different parts of a same program run simultaneously or parallelly without affecting the outcome. They are either set in order or can be set to run at the same time as per the user inputs. These processes are either Synchronous or Asynchronous.

Synchronous processes [1][2] are easy to execute and compile and run smoothly, as they execute in with the calculations of time and memory management. These processes are constructed with strict order of instructions and have a smooth memory management with components such as garbage collector, stack allocation, explicit resource management, object recycling, memory pooling, reference counting and many more.

Asynchronous processes [1][2] are the processes which do not follow order in instruction set or time management. These processes can be a little bit difficult to execute and can cause issues sometimes.

The Motive behind the research in this paper have been from experiences of issues such as system hang, crash and data corruption. We have countered these issues from time to time which exist through the whole system in a very wide perspective. Sometimes these issues results can be small such

as a delay in program and otherwise big such as crash. Sometimes these issues come from developers program while compilation or execution of program, and otherwise from exploiters or hackers who intentionally run these kind of programs in system. Sometimes these issues can be on the foreground such as data and result manipulation and otherwise in the background such as exploitation of sensitive information(GI)[5] and DoS(Denial of Service) attack[5].

The Objective of this paper is to make a comprehensive understand and analyze issues related to concurrency and give a basic understanding of these issues in many different types of systems. Applications of solutions to these issues in their programs and systems to help progress for development of the performance and results.

II. LITERATURE SURVEY

The Context for this topic. The research papers, journals and developer communities from the web have described about these systems and the issues they face. We have comprehensively described about these with the necessary details. These issues can be found in several different types of applications such as computers, smartphones, embedded systems, GPUs, cloud systems, cryptography, data and network security and many more. Most common issues resulting in are data races, data manipulation, corruption, loss,

theft, security breaches, network latency, data network jamming, exploits such as DOS(denial of service) attack, sensitive information theft and many more.

Data race [1][2][5][7][8] conditions in Linux Kernels with multi-threading in simple programs such as a mathematical operation. When multiple threads try to access data on a same memory location it can cause a 'race' condition. This race condition can cause data corruption, misplaced, loss, deadlock (sometimes system getting hang), program crash; any either of them. Few of the test cases were taken from Bugzilla, which were analysed and replicated.

Embedded systems [2][5][7][8] designs can cause synchronization bugs very hard to trace, because model checkers and dynamic simulation use synchronous model whereas the actual behaviour is according to an asynchronous model. Some of the issues are replicated using a model checker which shows model checkers can be utilized for more than just model checking. Model checkers associate with hardware and software systems to specify the liveness and safety requirements. It also helps in checking that the finite state model of a system meets the given specifications.

Parallel Computing using GPUs [3][7] is used in systems with data of complex problems such as Graphics processing, Deep Neural networks and Big data. Radar systems in defence also use parallel computing to process large amounts of data. Here a complex problem is transferred into GPU and is broken into several basic simple problems consisting of basic operations and the result is outputted. CUDA (compute unified device architecture) is the proprietary platform and API developed by NVIDIA for computing on GPUs, similar to C++. Detection of Synchronization bugs on this platform can be made possible via Memory-access modelling. Memory-access models play a critical role to determine the performance, scalability, stability and efficiency of a multi processor system.

Cloud systems [5][4] of today's time tend to capture a lot of data from different components of the system such as machine to machine application. This data is suppose to be transferred to data base and server to process it on a large scale. Real world systems are used such as HDFS, Hadoop

MapReduce and Hbase to do analysis of root causes. Tools used to loop identifier and static identifier to make statistical analysis of the multiple bugs found. One such is: Due to multiple request working asynchronously can cause Network latency issues which are kind of synchronization issue. These can cause data misplacement, corruption, loss and manipulation and security threats to data such as theft.

Cryptography [5][7] is use to secure data files while transferring. Many time concurrency issues tend to manipulate

or misplace data from cryptography files. Trusted components can access the file with a particular key to access the file. When concurrency issues manipulate data, it makes the resource files vulnerable to data theft or task such as executing malicious code, disrupting services and gaining root privileges. Some of these can cause threat to security key as well. A well known example is the "DIRTY COW" vulnerability which allowed attackers to cause race conditions which allowed attackers to change files such as "etc/passwd". Techniques use to prevent these are auto updates, central server management, check and sync files and many more.

Bitmap operations [6] are data structures for matrices that are use to store images (color pixels and range). The data is an integer which states the state of the pixel in the image. Concurrency issues in this situation can be due to improper management of large data which leads to data corruption, atomicity, visibility problem, performance bottlenecks, etc. Solutions are mentioned below.

The above paragraphic information are about different systems and the descriptions about different concurrency issues that can be seen in the systems. We have mentioned the behavior of different types of issues in the system that are commonly found.

Some of the real life examples where concurrency and synchronization issues have caused havoc are discussed.

In 2024, the JP Morgan Chase's "Infinite money glitch"[1] is an example of synchronization issues in proprietary code that was written in COBOL. It let counterfeit payment cheques to withdraw money without approving the clearance of the cheque. This caused people to withdraw money they never deposited in their bank accounts. Later, the bank charged their accounts with the unpaid amount and the balance went in negative.

In 1996, 'Ariane 5 Rocket' of the 'European space agency(ESA)' had a calculation error due to synchronization issues in the "Inertial reference system" which led to a 64-bit floating point number was converted to a 16 bit integer. This issue led to the system to propel and take a 90 degree turn at the same time after 37 seconds of it's launch which led to explosion of the rocket.

In 2012, "Knight Capital Money Burn Speed Run", where Knight capital company used algorithms to perform trades in the stock market. The developers accidentally used a variable 'POWER_PEG', which was a variable of the outdated testing item and was used to manipulate the virtual market by buying high and selling low which had no impact on real market trade. however, using the same variable here caused synchronization issues in trading and when pushed in production, it started to make incorrect trades by buying high and selling low. Knight Capital announced, 4 million trades in

45 minutes were effected and the company lost 440 million dollars in a single day and wiped out 75% of the investor's equity.

III. METHODOLOGY

Approach for the Research of this Document

Concurrency bugs can be classified on many basis: deadlock and non-deadlock; data manipulator and non data; atomicity violation, order violation, network effector and many depending on its behaviours and execution orders.

Framework of the research is based on many references that were taken from across the web. Linux developers and users who have been working in the industry for quite a good time, their opinions on different problems were evaluated and different objectives for classifying the bugs and vulnerabilities are made. We have described them below as:

When multiple threads working in the same memory location. For instance, when in a C++ code we call for random variable generation for two or more variables. C++ language does not have garbage collector which makes it volatile to memory leak. This will intent the memory manager to make the code vulnerable to manipulation or loss.

In heavy programs such as games or graphics programs, random operations are used in abundance. Duplication of a variable in the code can be observed very frequently which makes it also volatile to crashes or system hang or lags.

Here many time race conditions are also generated where multiple threads try to modify a single variable simultaneously while the execution of the compiled ones are going on. This will result in undesired changes in results of the executable.

Manipulation of a reference variable by multiple threads concurrently from different files, without proper synchronization. In some applications such as a browser, dynamic data structures in kernel, same global variables are called by multiple threads. This causes heavy data traffic in the memory(both RAM and Cache) due to multiple request of data. Sometimes while execution as different threads have different program files, referring to same variable with different values will result in deadlock. Memory leaks also occur due to manipulation of variable values, as compiled threads in execution phase can not access the variable (or resource). Memory leaks are commonly caused due to pools of memory getting inaccessible, when pools of memory is not freed after the process is completed. The acquiring of variable with no proper sequence or synchronization will lead to memory stuck and leak.

Order Violation related in APIs are a kind of bug that is a lot exploited by hackers(attackers) which can help them initiate Denial of Service attack and double-deletion crash. When attackers run multiple threads on same files, such as a "timer file", which monitors the expiration of a running process. These can cause manipulation or misplacing of command data due to race conditions. Exploiting these further can cause more problems and multiple threads cause such a race condition that can lead to improper order of commands in the API of the component.

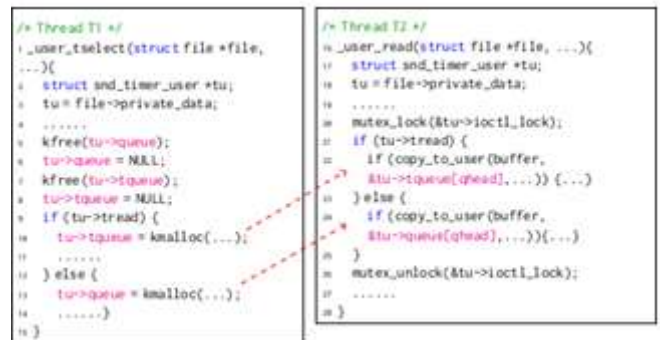


Figure 1: Order Violation in APIs

Order Violation in Memory Initialization are a kind of condition which can cause Gain Information(GI), which can disclose sensitive information to exploiters. Commands such as copy() and flag() with order violation are concurrent copies and are copied into buffer space in user space. Order violation also cause race to allocate new memory to undesired call functions, thus give memory to undesired programs. To trigger the program, attacker may use a two threaded program where the threads operate on same file pointer repeatedly to invoke usr_select() and usr_read(). Once the aforementioned interleaving occurs, uninitialized memory content will be leaked to the user space. These exploitation happens in background and is very difficult to detect.

Atomic operations are series of uninterrupted operations than are serialized from a single unit or component. These are series of operations that sometimes take a lot of memory and thus lead to leak. Leak can also cause sensitive data to get vulnerable. Atomic processes are not secured properly, so the data can be accessed by other threads and can be manipulated. This leads to undesired data being stored into the active variables and this may lead to crash sometimes. These can also cause race conditions, deadlocks and many malfunctions. Proper initialization of the memory, algorithm in the operation and use of garbage collector can help overcome this issue. Mechanisms such as locks to thread access, memory core access,etc. help in securing the data of these operations.

```

1 struct timerfd_ctx {
2     .....
3     struct list_head clist;
4     bool might_cancel; }

/* Thread T1 */
5 _remove(timerfd_ctx *ctx) {
6     if(ctx->might_cancel){
7         ctx->might_cancel = false;
8         spin_lock(&cancel_lock);
9         list_del_rcu(&ctx->clist);
10        spin_unlock(&cancel_lock);
11    }
}

/* Thread T2 */
12 _setup(timerfd_ctx *ctx, ...){
13    .....
14    if(!ctx->might_cancel){
15        ctx->might_cancel = true;
16        spin_lock(&cancel_lock);
17        list_add_rcu(&ctx->clist,
18                   &cancel_list);
19        spin_unlock(&cancel_lock);
20    } else if(ctx->might_cancel){
21        _remove(ctx);
22    }
}

```

Figure 2. "Timer" file Descriptor and Operations; Order violation in Memory Initialization

```

//src/main/java/org/apache/hadoop/mapreduce/lib/output/JobOutputCommitter.java
public void handle(JobEvent event)
{
    writeLock.lock();
    doTransition();
    writeLock.unlock();
}

//src/main/java/org/apache/hadoop/mapreduce/lib/output/JobOutputCommitter.java
public void commitJob(JobContext context)
{
    for(FileStatus stat: getMRCommittedTaskPaths(context)) {
        mergePaths(fs, stat, finalOutput);
    }
}

//getMRCommittedTaskPaths()
//Get a list of all paths where output from committed tasks are stored.
//Merge two paths together. Anything in 'from' will be moved into 'to'.
//If there are any nested subdirs when merging the files or directories in 'from' will
getPrivateStatic void mergePaths(FileSystem fs,
                                final FileStatus from, final Path to)
{
    for(FileStatus subFrom: fs.listFilesFrom(getPath())) {
        mergePaths(fs, subFrom, subTo);
    }
}
}

```

FIGURE 3. The complexity analysis of MR4813.

In the writeLock critical section, there are a 2-depth loop and a recursive loop. Sign "99K" means going through several calls and sign "→" means directly calling.

Experimental design, few algorithm blocks from program of kernel and application source codes made by developers and integrated by us have been described.

Algorithm 1: Critical section analysis algorithm Input: A method M and its control-flow graph G

Output: A set of critical sections C=C—C is a critical section in M

Set<Critical section> C Null; if M is a synchronized method then new C

```

C      BBG
C.add(C)
for i in InstructionsOf(M) do
if i is a Lock-Require instruction then new C
BBC      Null

```

```

for j in Lock-ReleaseInstructionsOf(i) do
BBC
BBC      (SuccBBs(BBi) P recBBs(BBj
))
end
C      BBC
C.add(C) end return C;

```

Analysis of the Pseudo Code

For effectively pinpointing areas of program that require mutual exclusion to prevent concurrency issues this mechanism is applied. It identifies critical sections in a program by analyzing its control-flow graph. It returns a set which contains all the critical sections that use synchronization tool to perform write functions to the shared resources. For each lock-required instruction, it initializes a new critical section. It identifies basic blocks and indexes for common predecessors for corresponding lock-release instructions. This is then added into the critical section set. This mechanism is similar to RCU(Read- Copy-Update) mechanism.

Algorithm 2: Loop field analysis algorithm Input: A method M and its control-flow graph G
Output: A set of loop field L=L—L is a loop in M

```

Set<Loop Info> L      Null; for i in InstructionsOf(M) do
if IndexOf(SuccInstruction(i)) < IndexOf
(i)      then new E
End      SuccInstruction(i)
Estart  i
if Eend equal FirstInst(any Lk in L) then
continue new L
BBL      (SuccBBs(Eend) P recBBs( Estart ))
L      BBL
L.add(L) end return L;

```

Analysis of the Program

For performance optimization, this mechanism is used which identifies all the loops within certain method of the synchronization variable and return all of them in a particular set. It identifies the steps by their index to initialize sequence and compares the loop function with other previous loops identified. If the same loop is already their then it skips and evaluates the next one, and if a unique loop in the memory is identified then it will create a space for it in the return set and adds the information into the set.

Tools and Technology Synchronization Mechanisms

These are mechanisms use to handle concurrency issues and related issues in the memory, background and in user and buffer space. These can be of many types based on the program's execution behaviour. These can be thread locks, file locks, memory locks, ownerships, compiler objectives and many more.

Garbage collector: these are mechanisms or algorithms that are used to free up memory where out of scope or expired data is placed. Java has an automatic garbage collector. C++ has a garbage collector which needs to be manually used by the developer. Rust has a different approach which is discussed further.

Garbage collector example code block in C++:

```
int buffer1, buffer2;  
buffer1 = (int*) malloc (100*sizeof(int)  
);  
buffer2 = (int*) realloc (buffer1, 500* sizeof(int));  
//perform operation or task free(buffer1); free(buffer2);  
return 0;
```

Analysis of the program:

In C++, garbage collectors are called manually by 'free' function. 'Malloc' is used to allocate new particular size of memory to a mutable variable and 'realloc' is used to reallocate the memory with a reference to the location of the memory to be reallocated. Another function 'calloc' is used to allocate memory without any size to variable and is free size. To free the memory, reference to the variable is called with 'free' function after the operation is performed and the variable goes out of scope.

Mutexes

When multi-threading is called in, then mutex can be used as a monitoring parameter to block more than one thread to execute a block of code. 'mutex var' is used to refer to the memory where the variable is present and is initialized to prevent race condition commonly.

Spinlocks

A kind of locking mechanism used where only one core can lock the use of the code block. Only one core can have the lock. Provides mutual exclusion.

Semaphores

These are kind of special mechanisms where only special kind of code blocks with synchronization primitives can access the shared-resource data. These shared resource data are flagged.

Scoped Locks

When mutex flagged variable, scoped locks provide lock to the data in the memory, and ensures it goes unlocked when data goes out of scope. Can be referred as it's working is similar to a garbage collector but here the memory is locked.

Std Locks

provide locks to different executables to avoid deadlock using deadlock avoidance algorithm. The objects are locked by unspecified random names for locks. These are used exclusively in programs with same code run simultaneously with other code blocks to execute different results.

Seq Locks

Seq locks allow multiple readers to access a resource concurrently while enabling exclusive access for writers.

Mechanism: A sequence number is maintained. Readers check this number before and after reading. If the sequence number changes during a read, the read is considered invalid, and it must be retried. The sequence number changes due to when writing operations are performed.

Read-Copy-Update (RCU)

RCU allows readers to access data without locking, enabling high concurrency.

Readers can access the Data Freely

Writers make a copy of the data, update the new copy, and then switch pointers atomically.

Old versions remain accessible until all readers finish using them, so that when an old copy goes out of scope it is cleared from the memory to free up space.

Two types of locks on a slightly different approach: Exclusive locks

Grants access to only one thread for writing. Other threads are blocked until the lock is released.

Shared Locks

Allows multiple threads to read concurrently, but no thread can write until all shared locks are released.

IV. RESULTS

Data we have implemented, tested and used are from different applications and programs. The programs are accessing files, text, operating mathematical functions, accessing data and are used for many requirements. We have added graphs and data for referencing different past and present evaluated changelogs experiences to eye on problems that have been encountered and are currently being countered. source codes of kernels of different distros are evaluated from the official releases with their changelogs also being evaluated to derive technical conclusion. different programming languages give different experiences in different applications.

V. DISCUSSION

We have collected, implemented, obtained results and interpreted the data to come on to a conclusion for this research. C++ is a high level writing and low-level interpreting language. This means that while writing it can look very human, while can work with very low level systems. Which is why it is a much preferred language by

developers for writing programs for Embedded systems, Data bases, back-end remote servers, game engines and GPUs (CUDA is extension of C and C++ by NVIDIA), Operating systems and many more where performance with memory consumption is critical.

The approach for memory management and garbage collection in C++ is very manual and can power and time consuming. Garbage collection in C++ is manually done with no bounding of data to its variable and can manipulated easily through vulnerabilities. Which is we have implemented an another option.

The Rust Programming Language

Rust is a memory safe compiled, high-level simplicity writing with low-level performance system. Which is why it is one of the most rising preferred language for writing programs of Operating systems, data bases, game engines, servers, embedded systems and many more. It is also used in web assembly which makes browsing more smooth working.

This language uses a concept of 'Ownership and

Borrowing' in memory management to manage memory safety to ensure no memory leak. It uses a concept where every default variable is immutable to be used in the stack memory. Mutable variables with unknown size at compile time are stored in Heap memory. Variable name assign to a data value is known as the owner and when the variable goes out of scope, the memory allocated to it is dropped automatically. When we want to assign the value to a new a variable, we can do it by reference which is called borrowing and ownership remains the same during the compile time.

Analysis and Illustration of program examples are evaluated below.

Code for Borrowing

```
{
let var_a = light::new( );} walk (&var_a);}
}
```

Analysis of the program:

This is a pseudo code example of Ownership and Borrowing in Rust. In the first block, memory is allocated with the variable 'var_a', known as the owner. After the variable goes out of scope, the value is dropped, and memory is freed. In the second block, the value of 'var_a' is allowed to be borrowed to a reference variable, which helps with memory safety and efficiency.

Application of code block of Ownership and Borrowing in Rust:

```
// This function takes ownership of a box and destroys it
fn eat_box_i32(boxed_i32: Box<i32>) { println!("Destroying
box that contains
{}", boxed_i32);
```

```
}
// This function borrows an i32
fn borrow_i32(borrowed_i32: &i32) { println!("This int is:
{} ", borrowed_i32
);
}
fn main() {
// Create a boxed i32 in the heap, and a i32 on the stack
// Remember: numbers can have arbitrary underscores added
for readability
// 5_i32 is the same as 5i32
let boxed_i32 = Box::new(5_i32); let stacked_i32 = 6_i32;
// Borrow the contents of the box.
Ownership is not taken,
// so the contents can be borrowed again
borrow_i32(&boxed_i32); borrow_i32(&stacked_i32);
{
// Take a reference to the data contained inside the box
let _ref_to_i32: &i32 = &boxed_i32;
// Error!
// C a n t destroy boxed_i32 while the inner value is
borrowed later in scope.
eat_box_i32(boxed_i32);
// FIXME Comment out this line
// Attempt to borrow _ref_to_i32 after inner value is
destroyed
borrow_i32(_ref_to_i32);
// _ref_to_i32 goes out of scope and is
no longer borrowed.
}
// boxed_i32 can now give up
ownership to eat_box_i32 and be
destroyed eat_box_i32(boxed_i32);
}
```

Analysis of the program:

This code is an example of application of Ownership and Borrowing in Rust. Variable is boxed integer to store i32 data type integer. "eat_box_i32(boxed_i32:Box<i32>)" takes the ownership of a boxed integer and destroys the box. "borrow_i32(borrowed_i32: &i32)" borrows a reference to the integer and prints its value without taking its ownership. "boxed_i32" is allocated to heap memory and "stacked_i32" is allocated to stack memory.

Program to implement multi-threading in Rust.

```
use std::thread;
use std::time::Duration;

fn main() {
thread::spawn(|| { for i in 1..10 {
println!("hi number {i} from the spawned thread!");
thread::sleep(Duration::from_millis(1));
}
});
```

```
for i in 1..5 {
println!("hi number {i} from the main thread!");
thread::sleep(Duration:: from_millis(1));
}
}
```

Output of the above code:

```
hi number 1 from the main thread! hi number 1 from the spawned thread!
hi number 2 from the main thread! hi number 2 from the spawned thread!
hi number 3 from the main thread! hi number 3 from the spawned thread!
hi number 4 from the main thread! hi number 4 from the spawned thread!
```

Analysis of the program:

Multi-threading in Rust is enabled with ensuring memory safety, programs without memory safety are not executed generally. In the above code, ‘thread’ and ‘duration’ libraries are called for thread handling and time synchronization.

A new thread is spawned with an execution of print for counts from 1 to 9 and sleeps for a millisecond after each print step. The main thread is executed in loop and made to sleep for a millisecond after each print step.

Here both threads are running concurrently, ensuring that no race condition is created using the sleep function to make use of time synchronization.

Mutex code example in C++:

C++ code example:

```
#include <thread> #include <mutex> using namespace std;

std::mutex m; int i = 0;

void makeACallFromPhoneBooth() { m.lock();
cout << i << " Hello Wife" << endl
; i++;
m.unlock();
}

int main() {
thread man1(makeACallFromPhoneBooth)
;
thread man2(makeACallFromPhoneBooth)
;

man1.join();
man2.join(); return 0; }
```

Mutex are used in C++ to monitor and block access to a block of code that is accessed concurrently by multiple threads. Here in this code thread and mutex standard libraries are called. Variable ‘m’ is initialized with mutex and m.lock() is called in void() to execute the function that is next in the code

to get executed. After the code is executed, m.unlock() releases the block from the code and is free to get access by the next thread.

When the next thread is executed, mutex will block access to the block of code for all other threads. Main() calls the threads for execution in sequence, the sequence will only be followed during compilation and not during the execution. Mutex will ensure the memory to follow synchronization mechanism.

“DIRTY COW” vulnerability in Linux

“Dirty Cow” vulnerability in Linux OS has been and is currently is been one of the most targeted vulnerability creating threats for systems such as a DoS attack, Sensitive information leak, memory leak and corruption, gain privilege, etc.

This vulnerability work with the application of data race condition in memory. Exploiters run such programs in memory which intentionally create and target time_sync() file which monitors termination of programs after it goes out of scope. These race conditions targeting this file will create a situation where it will make the processor skip the time file which creates a situation where monitored programs will not be terminated after it’s work gets out of scope.

This conditions will then target “etc/passwd” file to get executed and do not get terminate after it is out of scope. And this will get the exploiter to gain access to the password of the machine. This vulnerability was highly active in the devices that used the Linux kernel version 4.8.3 till 2018 and is still active at a comparatively low level.

```
/* Thread T1 */
1 open("/etc/passwd", O_RDONLY);
2 map=mmap(...,PROT_READ,MAP_PRIVATE);
3 position=strstr(map,"test:x:1001");
4 content= "test:x:0000";
5 pthread_create(...,T2,file_size);
6 f=open( "/proc/self/mem",O_RDWR);
7 while(1){
8 lseek(f,position,SEEK_SET);
9 write(f,content,...);
10 }
```

```
/* Thread T2 */
11 while (1) {
12 madvise(map,file_size,
13 MADV_DONTNEED);
14 }
```

Figure 4. DIRTY COW in Linux Kernel

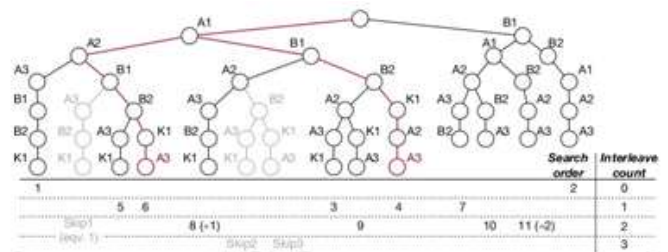


Figure 5. LIFS Process Tree LIFS is

Statistical Data

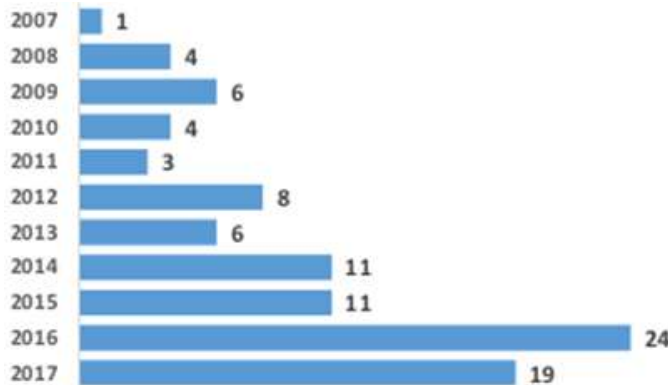


Figure 6. Number of Concurrency Vulnerabilities per year according to 'IEEE Xplore'

Vulnerability Type	Abbreviation	Num. Vulnerabilities	Percentage
Denial of Service	DoS	73	72.28%
Memory Corruption	MC	61	60.40%
Gain Privilege	GP	30	29.70%
Gain Information	GI	6	5.94%
Bypass Something	BS	5	4.95%
Execute Code	EC	2	1.98%

Figure 7. Distribution of Vulnerabilities Types

Vulnerability Type	Number of variables		Number of threads	
	one variable	> 1 variable	two threads	> 2 threads
Denial of Service	98.63%	1.37%	98.63%	1.37%
Memory Corruption	100.00%	0%	100.00%	0%
Gain Privilege	93.33%	6.67%	93.33%	6.67%
Gain Information	100.00%	0%	100.00%	0%
Bypass Something	100.00%	0%	100.00%	0%
Execute Code	100.00%	0%	100.00%	0%

Figure 8. Manifestations of Vulnerabilities for Exploitations

Number of Patches Needed	Number of Vulnerabilities	Percentage
1	19	18.81%
2	9	8.91%
3	16	15.84%
4	12	11.88%
5	8	7.92%
>5	37	36.63%

Figure 9. Number of Patches Needed to Fix a Vulnerability

Future Scope

We have Rust that does not have a garbage collector, but C++ have a garbage collector which only works when it is called in a program. Rust is a programming language that takes a very different approach for memory management. The scope for this kind of memory management can be very useful in building efficient Embedded systems, Servers, data bases, Operating systems and Web assembly. Rust has a good future scope where systems can be entirely be build using rust which can then avoid common problems like race conditions, data

manipulation, corruption and theft, Network latency, data network jamming and Enhanced Cybersecurity.

Developers have come on to conclusions that compile time of Rust is faster and sometimes the same or slower than C++. In the incremental lex benchmark, which modifies the largest src file, 'clang' was faster than 'rustc'. though many research results may show 'clang' or 'g++' to be faster than 'rustc', we have come across many results where Rust was faster than C++, with a lot of safety options for memory optimization. on a more complex case where we took test results of C++, Java, Python and Rust; results show that different languages are best for particular systems.

C++ is more optimized in embedded systems, Data servers, Operating systems, Game engines, GPUs and systems where memory optimization is a top priority and has simplicity in code. The only disadvantage in C++ is its manual Garbage collector. it is not a pure memory safe language and can be manipulated easily.

Java has a automatic garbage collector, but cannot be optimized for Embedded systems because of its 'JDK(Java development Kit)' and 'javac' compiler which can be too heavy for memory optimization where memory size is relatively low. Hence java is used generally in data applications where we have relatively abundance of memory. Python also has a automatic garbage collector, but cannot be used for memory optimization where memory size is relatively low. Hence python is used generally in data applications, language models and AI(Artificial Intelligence) where we have abundance of memory. Simplicity of Python also makes it a good choice for language models and AI where data I/O processing is on a large scale.

In 2021, 2022 and 2023; on "StackOverflow", Rust has been voted as the most loved programming language by developers who love it for its simplicity and memory safety. Its is a High-level Interpreting and Low-level Compiled programming language which has a very different approach for Memory optimization. It uses the concept of "Ownership and Borrowing", which makes it as the best choice for Embedded systems, data servers, Operating system kernels, GPUs, building high-performance web servers, APIs, backend services; Rust can be used to develop applications for various car components which has an embedded system, including Engine control units, Infotainment systems, and Advanced Driver Assistance systems (ADAS). C/C++ have traditionally been used in automaking and are still dominant. Though currently the application of language is less due to it's popularity but is increasing. Currently, Rust is utilized in the CSS engine of Firefox, enhancing both performance and safety. Mozilla's adoption of Rust demonstrates its capability to improve critical software components, reinforcing Rust's role in developing robust web technologies.

VI. CONCLUSION

We have made a research the one of the most common problems in the computer and digital electronic systems where multi-programming is major focus for development of the system. We have described the different systems that commonly encounter concurrency issues. We have described about detection, reasons, analysis and interpretation of these issues with the use of C++ and Rust programming languages. Comprehensively described the use of C++ and Rust and the potential Rust has over C++.

REFERENCES

1. Chinese Journal of Electronics, 2018; DOI:10.1049/cje.2018.03.015 — Linux Kernel Data Races in Recent 5 years.
2. SOSP '21, October 26–28, 2021, Virtual Event, Germany, ACM ISBN 978-1-4503-8709-5/21/10; doi.org/10.1145/3477132.3483549 — Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis.
3. IEEE/ICSE '20, May 23–29, 2020, Seoul, Republic of Korea; 2020 Association for Computing Machinery. ACM ISBN 978-1-4503-7121-6/20/05. ; DOI:10.1145/3377811.3380358 — Simulee: Detecting CUDA Synchronization Bugs via Memory-Access Modelling.
4. IEEE Access: DOI: 10.1109/ACCESS.2019.2923956; Understanding and Statically Detecting Synchronization Performance Bugs in Distributed Cloud Systems, National University of Defence Technology, Changsha, China, 410073.
5. arXiv:2212.05438v1 [cs.CR] 11 Dec 2022 — Understanding Concurrency Vulnerabilities in Linux Kernel.
6. Xuanhe Zhou, Ji Sun, Guoliang Li, Jianhua Feng. PVLDB, 13(9): 1416-1428, 2020; doi.org/10.14778/3397230.3397238 — Query Performance Prediction for Concurrent Queries using Graph Embedding.
7. Dae R. Jeong, Minkyu Jung, Yoochan Lee, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2023. In Eighteenth European Conference on Computer Systems (EuroSys '23), May 9–12, 2023, Rome, Italy. ACM, New York, NY, USA, 17 pages; doi.org/10.1145/3552326.3567486 — Diagnosing Kernel Concurrency Failures with AITIA.
8. A.Gupta, O. J. Pandey, M. Shukla, A. Dadhich, A. Ingle and P. Gawande, "Towards context-aware smart mechatronics networks: Integrating Swarm Intelligence and Ambient Intelligence," 2014 International Conference on Issues and Challenges in Intelligent Computing

Techniques (ICICT), Ghaziabad, India, 2014, pp. 64-69, doi: 10.1109/ICICT.2014.6781254.

- <https://distrowatch.com>
- <https://www.kali.org/docs/general-use/kali-linux-sources-list-repositories/>
- <https://bugs.kali.org/changelog-page.php>
- <https://sources.debian.org/>
- <https://www.debian.org/releases/stable/releasenotes>