Lab Evaluation – 02

- CB.EN.U4CYS22033

Q1.

Code:

#include <stdio.h>

int main() {

char operator;

double num1, num2;

printf("Enter an operator (+, -, \*, /, %): ");

scanf("%c", &operator);

printf("Enter two operands: ");

scanf("%lf %lf", &num1, &num2);

switch (operator) {

case '+':

printf("%.2lf + %.2lf = %.2lf\n", num1, num2, num1 + num2);

break;

case '-':

printf("%.2lf - %.2lf = %.2lf\n", num1, num2, num1 - num2);

break;

case '\*':

printf("%.2lf \* %.2lf = %.2lf\n", num1, num2, num1 \* num2);

break;

case '/':

printf("%.2lf / %.2lf = %.2lf\n", num1, num2, num1 / num2);

case '%':

if ((int)num2 != 0){

printf("%d %d = %d\n", (int)num1, (int)num2, (int)num1 % (int)num2);

}

default:

printf("Invalid operator.\n");

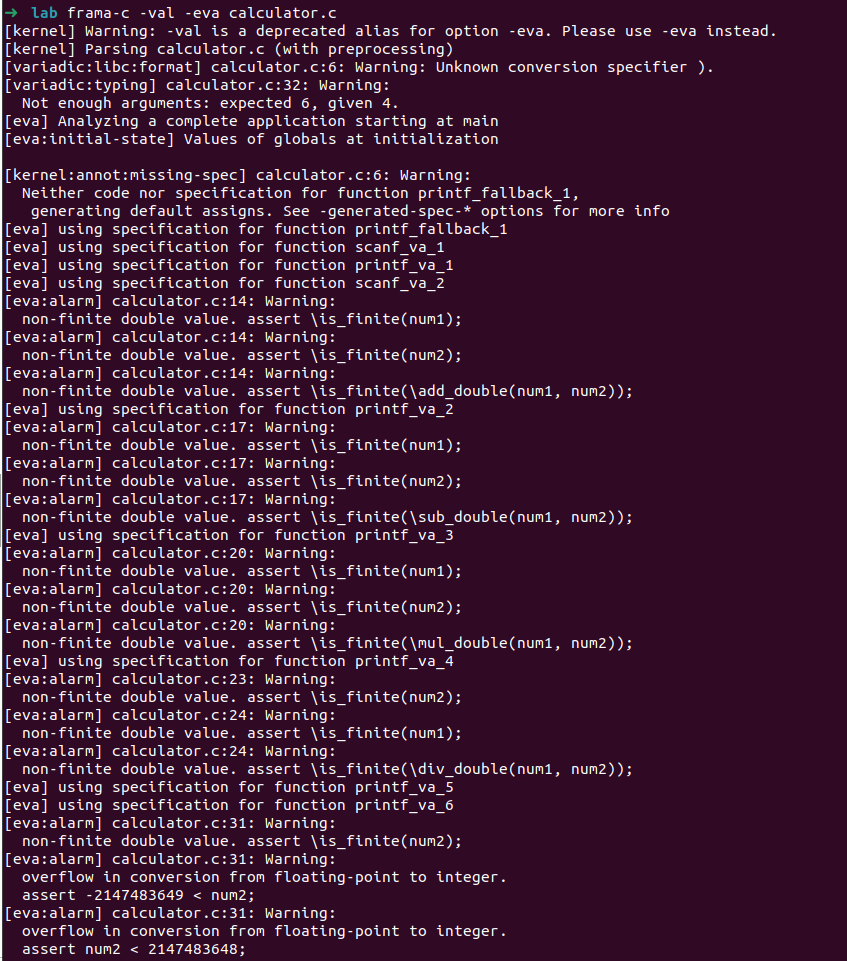
}

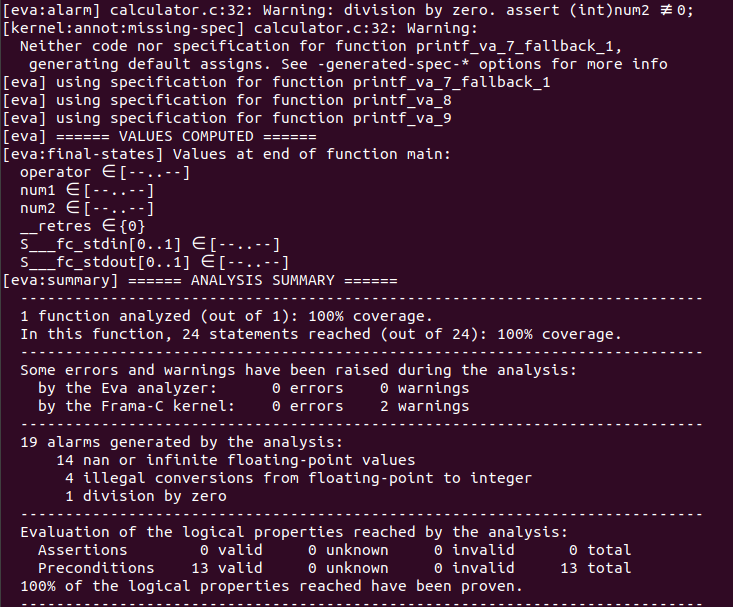
return 0;

}

a. Testing with frama-c tool

Screenshot:





Explanation:

Alarms Geneated by farma-c

Here i have taken frama-c tool because it is the tool used to analyze vulnerabilities during arithmetic

operations like addition, subraction, multiplication and division. It analysis the code and predicits the correct vulnerability present in the operatins which might leads to overflows, underflows, division by zeros etc.

1. 14 nan or infinite floating point values.

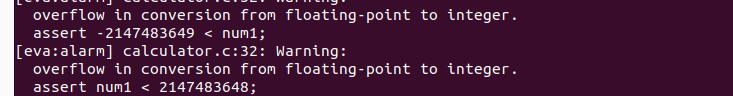
Frama-C raised alarms about non-finite double values in lines where arithmetic operations are performed, such as addition, subtraction, multiplication, and division. Specifically, it’s asserting that the operands (num1, num2) should be finite before performing these operations



2. 4 illegal conversions form floating point to intergers

`

In addition to the overflow warnings, Frama-C raised illegal conversion warnings related to the double to int cast. Since your operands are of type double, casting them to int might lead to unexpected behavior if the values are too large or non-finite.



3. 1 division by zero



Frama-C raised an alarm about potential division by zero when using the modulus operator (%). Since num2 is being cast to an int, Frama-C checks whether (int)num2 != 0. It detected that num2 could be zero, which would lead to a runtime error.

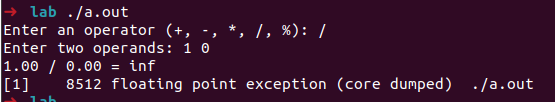
Outputs:

Example 01

1. operator = /

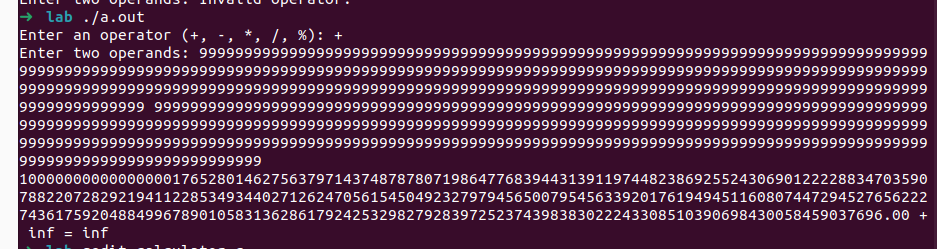
num1=1

num2=0

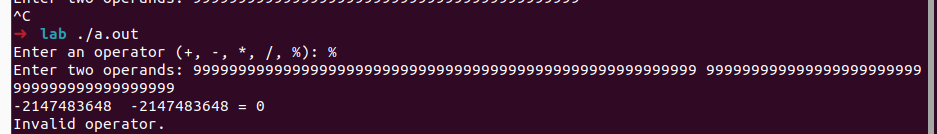


2.

Here if we supply a verify large number then the size of num1 and num2 ending with inf as output



3. Here in % i do typecasting double to int during which it causes to overflow .



b. 1a code with secure coding practice

Code:

#include <stdio.h>

#include <math.h>

#include <limits.h>

int main() {

char operator;

double num1, num2;

printf("Enter an operator (+, -, \*, /, %%): ");

scanf("%c", &operator);

printf("Enter two operands: ");

if (scanf("%lf %lf", &num1, &num2) != 2 || !isfinite(num1) || !isfinite(num2)) {

printf("Invalid input. Please enter valid numbers.\n");

return 1;

}

switch (operator) {

case '+':

printf("%.2lf + %.2lf = %.2lf\n", num1, num2, num1 + num2);

break;

case '-':

printf("%.2lf - %.2lf = %.2lf\n", num1, num2, num1 - num2);

break;

case '\*':

printf("%.2lf \* %.2lf = %.2lf\n", num1, num2, num1 \* num2);

break;

case '/':

// Check for division by zero

if (num2 != 0) {

printf("%.2lf / %.2lf = %.2lf\n", num1, num2, num1 / num2);

} else {

printf("Error: Division by zero.\n");

}

break;

case '%':

// Ensure both values are within the integer range

if ((int)num2 != 0 && num1 >= INT\_MIN && num1 <= INT\_MAX && num2 >= INT\_MIN && num2 <= INT\_MAX) {

printf("%d %d = %d\n", (int)num1, (int)num2, (int)num1 % (int)num2);

} else if ((int)num2 == 0) {

printf("Error: Division by zero in modulus operation.\n");

} else {

printf("Error: Integer overflow in modulus operation.\n");

}

break;

default:

printf("Invalid operator.\n");

break;

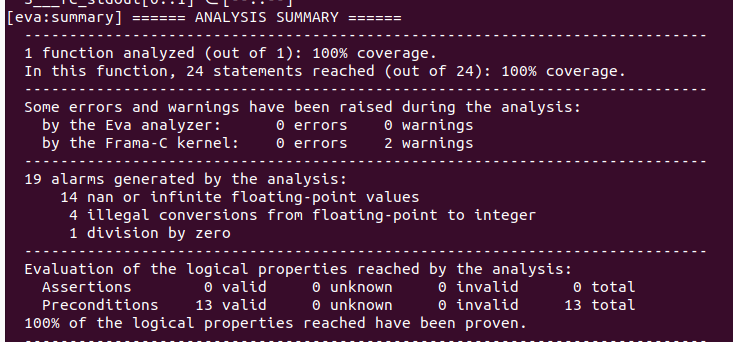
}

return 0;

}

Testing with frama-c tool

Screenshot:



Explanation:

Still frama c detects the 3 vulnerabilites present in before code but here I have improved the code

to handle all the 3 vulnerabilities

What improvement I have done:

1. To metigate infinite floating-point values

Used isfinite function from math library to verify that the given input is the finite range.

2. To metigate illegal conversions from double to int

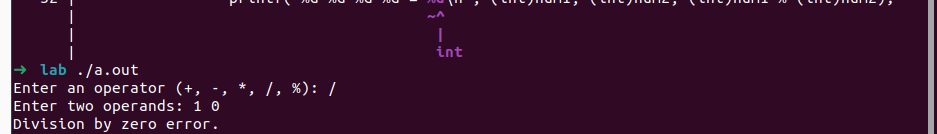
if have verified that the number num1 and num2 must be in the range upper and lower limit of interger datatype if it is not in that limit then no operations would be performed

3. Division by zero

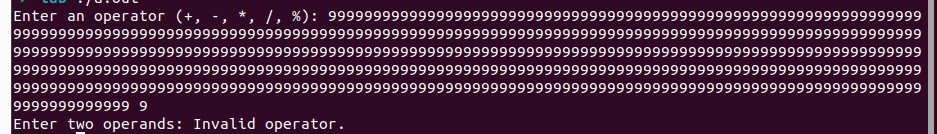
Coded an check condition on num2 to verify it is not a zero

Outputs:

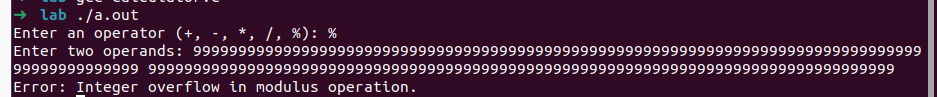
1. Division by zero vulnerability handled



2. Infinite input num vulnerability handled



3. Type casting vulnerability handled.



Q2.

a. Code to perform one thread for each operations without any synchronization

Since there is no synchronization, multiple threads accessing the result variable simultaneously could cause race conditions.

Code:

#include <stdio.h>

#include <pthread.h>

double num1, num2, result; // Global result variable

void\* addition(void\* arg) {

result = num1 + num2;

printf("Addition: %.2lf + %.2lf = %.2lf\n", num1, num2, result);

return NULL;

}

void\* subtraction(void\* arg) {

result = num1 - num2;

printf("Subtraction: %.2lf - %.2lf = %.2lf\n", num1, num2, result);

return NULL;

}

void\* multiplication(void\* arg) {

result = num1 \* num2;

printf("Multiplication: %.2lf \* %.2lf = %.2lf\n", num1, num2, result);

return NULL;

}

void\* division(void\* arg) {

if (num2 != 0) {

result = num1 / num2;

printf("Division: %.2lf / %.2lf = %.2lf\n", num1, num2, result);

} else {

printf("Division by zero error.\n");

}

return NULL;

}

int main() {

pthread\_t threads[4];

printf("Enter two numbers: ");

scanf("%lf %lf", &num1, &num2);

// Create all threads at once

pthread\_create(&threads[0], NULL, addition, NULL);

pthread\_create(&threads[1], NULL, subtraction, NULL);

pthread\_create(&threads[2], NULL, multiplication, NULL);

pthread\_create(&threads[3], NULL, division, NULL);

// Wait for all threads to finish

for (int i = 0; i < 4; i++) {

pthread\_join(threads[i], NULL);

}

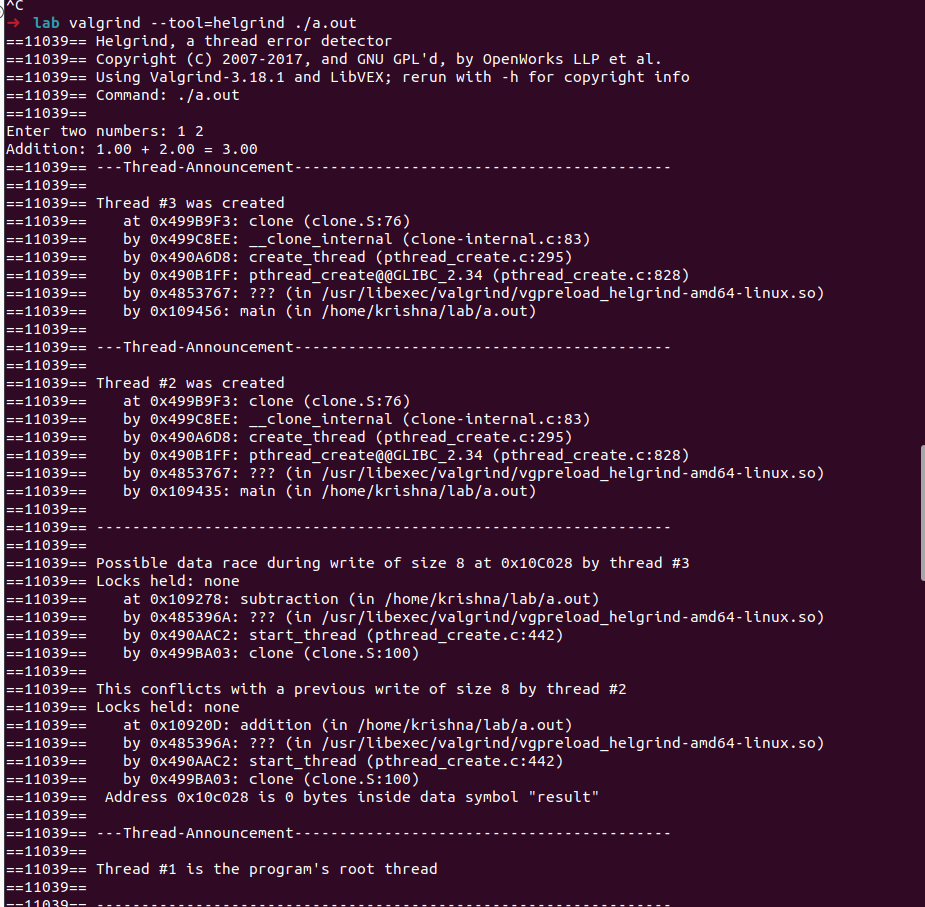
return 0;

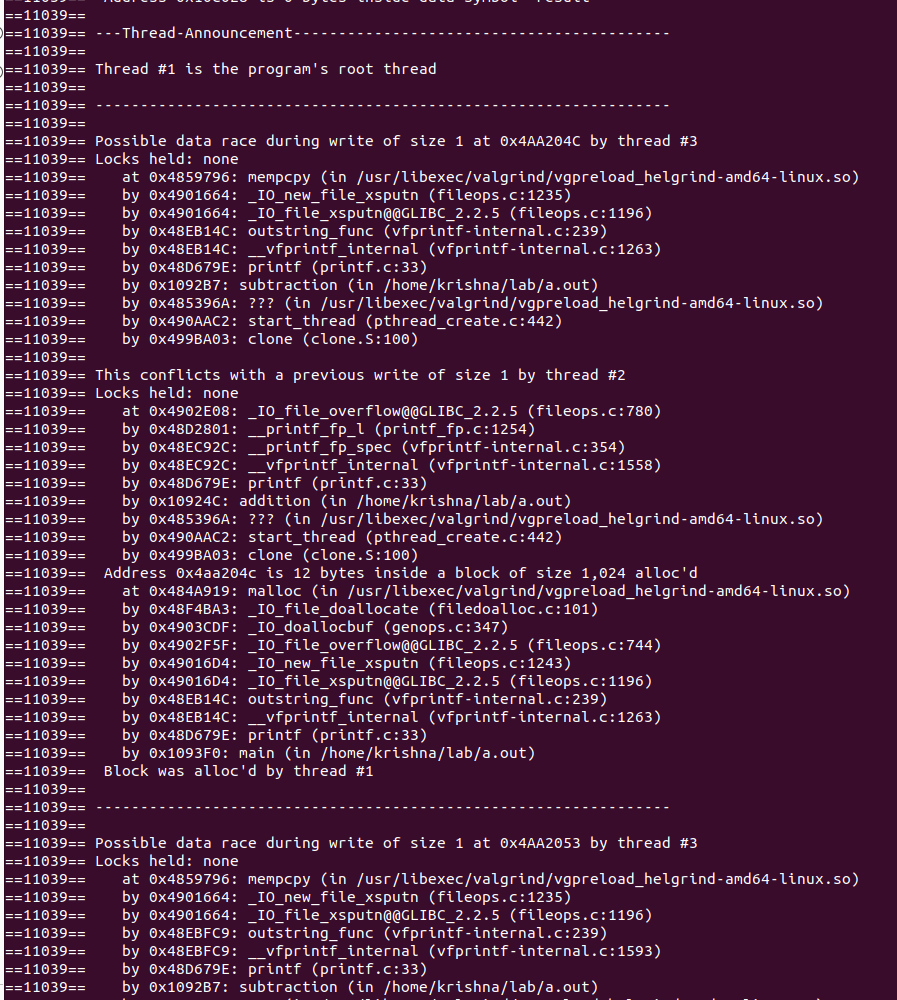
}

b. Test whith helgrind

Helgrind is the tool which is present inside the valgrind which is used to analyze the code for race condition vulnerability which occurs when multiple threads are trying to access the same variable at the same time.

command- valgrind --tool=helgrind ./a.out

f

c

c

Explanation :

1.Data Race on result: Multiple threads are modifying the result variable concurrently. For example:

* The subtraction thread writes to result while the addition thread is also writing to result.
* The multiplication thread writes to result while the subtraction thread is also writing to result, and so on for the other threads.

2. Print Operations: There are data races when different threads are trying to print their results simultaneously. Multiple threads access the standard output (stdout) at the same time, which also causes data races.

C. Here I have used muterlocks to prevent from race condtion

Code:

#include <stdio.h>

#include <pthread.h>

// Global variables for numbers

double num1, num2, result;

// Mutex for thread-safe access to result

pthread\_mutex\_t mutex;

// Function for addition

void\* addition(void\* arg) {

pthread\_mutex\_lock(&mutex); // Lock the mutex

result = num1 + num2;

printf("Addition: %.2lf + %.2lf = %.2lf\n", num1, num2, result);

pthread\_mutex\_unlock(&mutex); // Unlock the mutex

return NULL;

}

// Function for subtraction

void\* subtraction(void\* arg) {

pthread\_mutex\_lock(&mutex); // Lock the mutex

result = num1 - num2;

printf("Subtraction: %.2lf - %.2lf = %.2lf\n", num1, num2, result);

pthread\_mutex\_unlock(&mutex); // Unlock the mutex

return NULL;

}

// Function for multiplication

void\* multiplication(void\* arg) {

pthread\_mutex\_lock(&mutex); // Lock the mutex

result = num1 \* num2;

printf("Multiplication: %.2lf \* %.2lf = %.2lf\n", num1, num2, result);

pthread\_mutex\_unlock(&mutex); // Unlock the mutex

return NULL;

}

// Function for division

void\* division(void\* arg) {

pthread\_mutex\_lock(&mutex); // Lock the mutex

if (num2 != 0) {

result = num1 / num2;

printf("Division: %.2lf / %.2lf = %.2lf\n", num1, num2, result);

} else {

printf("Division by zero error.\n");

}

pthread\_mutex\_unlock(&mutex); // Unlock the mutex

return NULL;

}

int main() {

pthread\_t threads[4];

// Initialize the mutex

pthread\_mutex\_init(&mutex, NULL);

// Input two numbers

printf("Enter two numbers: ");

scanf("%lf %lf", &num1, &num2);

// Create threads for each operation

pthread\_create(&threads[0], NULL, addition, NULL);

pthread\_create(&threads[1], NULL, subtraction, NULL);

pthread\_create(&threads[2], NULL, multiplication, NULL);

pthread\_create(&threads[3], NULL, division, NULL);

// Wait for all threads to finish

for (int i = 0; i < 4; i++) {

pthread\_join(threads[i], NULL);

}

// Destroy the mutex

pthread\_mutex\_destroy(&mutex);

return 0;

}

Explanation:

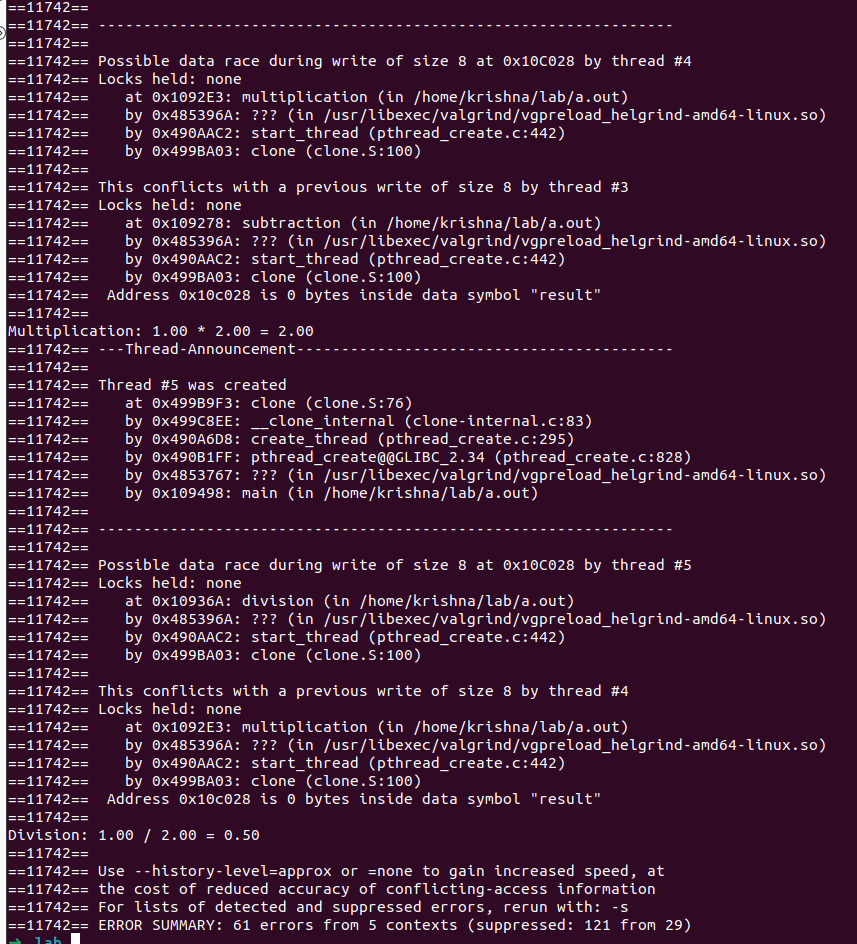
When a thread first starts executing the mutex for the thread will locked all other threads need to wait till the multex is released and the next thread is allocated based on the shudling implemented in the corresponding OS.

Output:

Without any errors executing



Testing with Helgrind



Helgrind still detects race condtion vulnerability, bujt each thread is been successully executed becase of mutex lock