Tools in high performance computing Exercise Set 2

Darina Öö

29.01.2024

1 Problem 1

1.1 Source Code

The program was designed to incrementally access array elements beyond its declared size until it caused a segmentation fault.

1.2 main.cpp

```
#include <iostream>
#include <exception>

int main() {
    int a[10];  // Array with 10 integers
    int i = 0;

    try {
        for (i = 10; ; ++i) {
            std::cout << "Accessing␣index␣" << i << ":␣";
            a[i] = i;  // Undefined behavior
            std::cout << a[i] << std::endl;
        }
    } catch (const std::exception& e) {
        std::cerr << "Exception␣caught␣at␣index
␣␣␣␣␣␣␣␣" << i << ":␣" << e.what() << std::endl;
    } catch (...) {
        std::cerr << "Unknown␣exception␣caught␣at␣index
␣␣␣␣␣␣␣␣" << i << std::endl;
    }

    return 0;
}
```

1.3 Compilation and Execution

The program was compiled with the GNU C++ Compiler using the following command:

```
g++ -o array_test array_test.cpp
```

It was then executed multiple times to determine the behavior when accessing out-of-bounds indices.

1.4 Results and Observations

The program's output varied with each run, but typically it accessed elements beyond the array's declared bounds without immediate crashing. However, when it reached index 1975, it resulted in a segmentation fault. This unpredictable behavior is due to the nature of undefined behavior in C++ when accessing out-of-bounds array elements.

1.5 Conclusion

Accessing an array beyond its bounds can lead to a segmentation fault, but the exact point at which this occurs is unpredictable due to the undefined nature of the behavior. This experiment reinforces the necessity of bounds checking when working with arrays in C++.

2 Problem 2

2.1 Methodology

The program was compiled with debugging symbols enabled and run within GDB. Upon reaching a segmentation fault, the contents of the array and the value of the last accessed index were printed using GDB's `print` command.

```
# Compile the program with debug symbols
g++ -g -o array_test array_test.cpp

# Start gdb with the compiled program
gdb ./array_test

# Inside gdb, run the program
(gdb) run

# If the program crashes, gdb will stop and you can inspect variables
(gdb) print a
(gdb) print i
(gdb) quit
```

2.2 Results

During the debugging session, the following values were observed:

```
(gdb) print a
$1 = {-1431626504, 43690, -134225856, 65535, -4096,
65535, -138644544, 65535, -3720, 65535}
(gdb) print i
$2 = 1036
```

2.3 Analysis

The array `a` contains what appear to be random or uninitialized values, indicating that the program was accessing memory beyond the allocated bounds of the array. The index `i` reached 1036 before the program crashed, demonstrating that the program was able to read and write to memory locations far beyond the end of the allocated array space without immediate detection by the operating system.

2.4 Conclusion

The experiment confirmed that accessing out-of-bounds array indices leads to undefined behavior. The program was able to continue until it attempted to access a restricted memory location, which resulted in a segmentation fault. The value of the index at the crash indicates that such errors can go undetected for a significant range of indices, leading to unpredictable program behavior and potential security vulnerabilities.

3 Problem 3

3.1 Source Code

```
#include <iostream>
#include <cmath>
#include <ctime>

template<typename T>
void compute_sum_and_time(long n) {
    T sum = 0;
    clock_t t1, t2;
    double cputime;

    t1 = clock();
    for (long k = 0; k <= n; ++k) {
        sum += exp(sin((T)k / 1000000));
    }
```

```
    t2 = clock();

    cputime = (double)(t2 - t1) / CLOCKS_PER_SEC;
    std::cout << "Sum using " << sizeof(T)*8 << "-bit floating point numbers:
    " << sum << std::endl;
    std::cout << "CPU time in seconds: " << cputime << std::endl;
}

int main() {
    long n = 100000000; // Example large value for n
    compute_sum_and_time<float>(n);    // 32-bit
    compute_sum_and_time<double>(n);   // 64-bit
    // For 128-bit, we assume a platform-specific 128-bit floating point type,
    like __float128
    // compute_sum_and_time<__float128>(n); // 128-bit (if available)

    return 0;
}
```

3.2 Compilation Instructions

To compile the program, use the following command:

```
g++ -std=c++11 -o sum_computation sum_computation.cpp
```

Execute the compiled program with the following command:

```
./sum_computation
```

3.3 Results and Discussion

The program calculates the sum and CPU time for 32-bit, 64-bit, and, if available, 128-bit floating-point numbers. The results from the calculation: Sum using 32-bit floating point numbers: 6.71089e+07 CPU time in seconds: 1.70312 Sum using 64-bit floating point numbers: 1.26865e+08 CPU time in seconds: 1.58722

The results showed that as the precision increased, the CPU time also increased. This is expected as higher precision requires more computational resources.

4 Problem 4

4.1 Compilation

The program was compiled with the flags `-O0 -pg -g` to disable optimizations and enable profiling with `gprof`.The program was compiled with the following command:

```
gcc -O0 -pg -g -o simpleprofiling simpleprofiling.c -lm
```

This command includes linking against the math library to resolve references to mathematical functions.

4.2 Execution and Profiling

The program was executed, and the profiling data was collected. The command used for profiling was:

```
gprof simpleprofiling gmon.out > analysis.txt
```

4.3 Profiling Results

The profiling results indicated the following CPU time usage:

- `main` function: 79.31%
- `_init` function: 20.69%

However, the profiler did not provide separate timings for the two inner loops.

The output from the program was:

```
2.50253e+10 2.4135e+07
```

These numbers represent the computed sums from the two inner loops.

4.4 Analysis and Discussion

Without detailed profiling for each loop, it is challenging to comment definitively on the time distribution between the two. The large percentage of time reported in `main` suggests that both loops are contained within it, but their individual contributions to the runtime cannot be discerned from the provided data.

For further analysis, I decided to restructure the code to isolate each loop into a separate function for more granular profiling.

4.5 Code Modification

The program simpleprofiling.c was modified (simpleprofiling2.c)to extract each inner loop into its own function. Below is the modified code snippet:

```
#include <math.h>
#include <stdio.h>
#include <time.h>

void loop1(long n) {
    double sum1 = 0.0;
    for (long i = 0; i < n; ++i) {
        sum1 += sin(i) * exp(i);
```

```
    }
    printf("Sum1: %e\n", sum1);
}

void loop2(long n) {
    double sum2 = 0.0;
    for (long i = 0; i < n; ++i) {
        sum2 += cos(i) * exp(i);
    }
    printf("Sum2: %e\n", sum2);
}

int main() {
    const long n = 1000000; // Adjust n as needed
    loop1(n);
    loop2(n);
    return 0;
}
```

4.6 Compilation and Profiling

The program was compiled with the following command to enable profiling:

```
gcc -O0 -pg -g -o simpleprofiling simpleprofiling2.c -lm
```

After running the program, the profiling was performed using:

```
gprof simpleprofiling gmon.out > analysis2.txt
```

4.7 Profiling Results

The profiling output indicated the following:

- `loop1` accounted for 70.11% of the total running time.
- `loop2` accounted for 9.20% of the total running time.

4.8 Analysis and Discussion

The granular profiling results highlighted a significant difference in execution time between the two loops. `loop1` requires substantially more time to compute than `loop2`. This suggests that the operations within `loop1` are more computationally intensive or less optimizable.

4.9 Conclusions

The code restructuring enabled precise profiling of the individual loops. The analysis showed that `loop1` is the primary consumer of CPU time and could be targeted for optimization to improve the program's overall performance.