ITP 30002-02 Operating System, 2023 Spring

# Homework 2

21900050 Kwon Eunhyeok, eunhyeoq@handong.ac.kr
21900215 Kim Hyeongi, 21900215@handong.ac.kr

## 1. Introduction

Delta debugging is a software testing technique that originated in the field of software engineering. It was developed to help software developers identify and isolate bugs in software programs more quickly and efficiently.

Delta debugging is tested with a variety of inputs or configurations, and when a bug is detected, it systematically fixes the program by removing other parts of the code to determine the changes that cause the bug. Finally, delta debugging allows you to find the smallest element that causes the bug.

Delta debugging is, above all, an automated technology that does not require human intervention, and is very powerful in that it can be used in any programming language.

In this HW2, we programmed a CIMIN (Crashing Input Minimizer) using the delta debugging algorithm to find out the cause of bugs in each program that deal with various types of files such as JSON, XML, and png. This requires the process of minimizing the initial crashing input. Therefore, I tried to use the recursion by dividing the input data into the head-tail, and mid parts int attempt to minimize the input.

## 2. Approach

### 2.1. Overview

cimin is an implementation of a variant of delta debugging algorithm that “gradually reduces the crashing input while preserving crashing result”. To test the target program with different substrings of the crashing input, it utilizes fork() and execv() to create new process and run the target program. Pipe() is also used to create unnamed pipe between the cimin and the child process running target program. Two pipes are utilized so that one can be used to send input from cimin to the standard input of the target program, and the other one used in cimin to read standard error from the target program.

### 2.2. Program Design

The program was subdivided into five big functions to decrease the complexity of writing the source code: (1) minimize(), the entrance of the minimizing algorithm where signal handlers are registered and reduce is called for the first time. (2) reduce(), an implementation of a delta debugging algorithm which is a recursive function that is recursively called when a smaller crashing input is found, (3) head_tail_mid_common(), function that tests if given crash message appears in the output generated by the target program using reduced input and returning integer value 0, 1, or 2 meaning no crash, crash, user interrupt respectively, (4) parent_proc(), function that is executed in the cimin program, used to actually hand over input and wait for the child’s output and read it. (5) child_proc(), function executed in child process to set up pipe connections to standard input and error of the process appropriately before executing the target program.

### 2.3. System Funtion

Two pipes were used in this program. One for writing input to the target program one for reading error from the target program. Since pipes are created at every iteration of the reducing process, it was declared as a global variable.
Signals were used handle situations when user interrupts the program or when the target program falls into infinite loop. To handle the case efficiently, the program was given three states, declared as integer variables in global scope: timeover, child_running, and interrupt_exit. These states are set as integer value zero at the beginning and updated to one in the handler function invoked by SIGALRM, SIGCHLD, SIGINT signals respectively.

Since it is not possible to know when or where in the running code, a signal will be raised, and to keep the program from exiting without producing any result, the desired action for each of the functions were not executed immediately but rather just modified its state value. However, sending SIGKILL signal to the child was done in the signal handler because if the target program fell in infinite loop, cimin would wait indefinitely for the error output to read from the target program.

Furthermore, when interrupt signal was raised by the user, signal handler was also programmed to kill the running child, since it is a sign to exit all currently working job and provide the most reduce crashing input found until then. Other mechanisms of the signals are examined before analyzing the error string produced by the target program, and actions were taken appropriately.

To make use of cimin not limited to string input, but available to all kind of input data in form of byte stream, for example png file, the program does not use string functions provided in string.h and null character as marking end of the crashing input when processing reduce() function. Instead it uses specific size and loop to copy input data of specificied size.

## 3. Evaluation

If you implement the cimin file according to Figure 1 algorithm, you can perform delta debugging on the crashes that occur in each program dealing with various types through cimin.

Criteria for Successful Accomplished Task

- Is the reduced file more minimized than the initial input file after debugging.
- Does Crash occur when running the reduced input generated after debugging.
- How long is the execution time of debugging.

### 3.1. jsmn

jsmn has a crash “heap-buffer overflow” error.
You can perform debugging by entering the following command line.

```
$ ls
cimin    jsmn
$ ./cimin -i jsmn/testcases/crash.json -m "AddressSanitizer: heap-buffer-overflow" -o reduced_jsmn jsmn/jsondump
…
$ ls
cimin    jsmn     reduced_jsmn
$ cat reduced_jsmn
{y}
$ jsmn/jsondump < reduced_jsmn
==7459==ERROR:AddressSanitizer:heap-buffer-overflow on …
…
```

### 3.2. libxml2

libxml2 has a crash “SEGV on unknown address” error.
You can perform debugging by entering the following command line.

```
$ ls
cimin   libxml2
$ ./cimin -i libxml2/testcases/crash.xml -m "SEGV on unknown address" -o reduced_libxml2 libxml2/xmllint --recover --postvalid -
…
$ ls
cimin   libxml2   reduced_libxml2
$ cat reduced_libxml2
<!DOCTYPE[<!ELEMENT
:(�,()><:
$ libxml2/xmllint --recover --postvalid - < reduced_libxml2
…
==37570==ERROR: AddressSanitizer: SEGV on unknown address …
…
```

### 3.3. balance

balance has a crash that falls into an infinite loop.
You can perform debugging by entering the following command line.

```
$ ls
cimin     balance
$ ./cimin -i balance/testcases/fail -m "balance" -o reduced_balance balance/balance
…
$ ls
cimin     libxml2     reduced_balance
$ cat reduced_balance
][]*[][
$ balance/balance < reduced_balance
(infinite loop…)
>(ctrl+c)
```

### 3.4. libpng

libpng has a crash that reads an uninitialized variable.
You can perform debugging by entering the following command line.

```
$ ls
cimin     libpng
$ ./cimin -i libpng/crash.png -m "MemorySanitizer: use-of-uninitialized-value" -o reduced_libpng libpng/libpng/test_pngfix
… (very long long time spent…)
$ ls
cimin     libxml2     reduced_libpng
$ xxd reduced_libpng
00000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452 .PNG........IHDR
…
000000c0: 5d00 0000 0970 4859 73 ]....pHYs
$ libpng/libpng/test_pngfix < reduced_libpng
…
==8557==WARNING:MemorySanitizer:use-of-uninitialized-value
…
```

Based on the 'Criteria for Successful Accomplished Task', you can analyze the results of four test cases.

First, if you run 'cimin' in all of the given cases and run the generated 'reduced' in the execution program, you can see that all of them output the error messages you want to extract through debugging. This shows that 'cimin' only debugged the elements that cause the crash without any other problems.

Second, in all four cases, the generated 'reduced' is more minimized than the initial 'crash_input'. This means that after debugging with 'cimin', a lot of excess data that is not related to crash has been deleted, and only the essential elements that are really related to crash remain.

Finally, it is necessary to check the amount of time it takes to perform debugging. Debugging all four test cases with 'cimin' shows that relatively small jsmn completes debugging very quickly. However, the relatively large libpng takes about an hour, so there is a lack in terms of debugging speed.

## 4. Discussion

Initial implementation of the cimin program directly followed the pseudo code provided in the assignment directions. However, after running libpng example, it turned out to be not very successful at minimizing the crashing input. The size of the crashing png input file was 2405 bytes and until almost 200 bytes the only thing cimin did was cutting the last byte value one by one. Which took half of the minimizing time and the other half of the time was iterating through all combinations of the smallest input found (201 bytes) to check if it can be reduced any further. The time wasted in those repetitive process was too much and could be reduced.

The reducing algorithm implemented in this homework begins by checking head and tail part, which is the first and last byte of the given crashing input, then checks the mid part which is the entire crashing input without the last byte. In the case of libpng test case, this made unnecessary checks of the first byte and last byte of the input before finding out that the mid part of the substring produced crash for almost 2000 times.

The first attempt to reduce the process time was done by changing the order of checking mid substrings and head and tail substrings. Since the by checking mid substrings of the crashing input first, cimin called reduce() function after checking that the mid substring produce crash and the time that was used to check the head and tail substrings could be eliminated for the first 2000 iterations. However, this improved time was too small and had no significant improvement on the overall time. This is because reducing the crashing input by one byte was too slow progress and significantly more time was spent on brute force checking the last 200 bytes of reduced crashing input.

Not implemented in this homework, but one possible solution to this problem could be using larger chunks of data to reduce the crashing input. Then in the case of libpng, it will produce reduce crashing input larger than the minimum. However, the time that take cimin to produce output will be decreased in proportion to the size of the chunk. This will be able to both reduce the iteration to approach significantly reduced crashing input and the time takes to test all combination of the reduced one.

## 5. Conclusion

In conclusion the cimin program was partially successful in minimizing the crashing input. It was able to produce significantly reduced crashing input for all the test cases and the produced results crashed the program. However, it took too long to minimize the crashing input for libpng program. Although we found acceptable solution for all test cases, it was a little disappointing that the last testcase took too much time (about 1 hour) to generate output file.

Throughout this homework, we could learn more about how to use signals and system functions and understand the concept in more detail. Overall, doing this homework was meaningful and beneficial in understanding the concept of OS.