ITP 30002-02 Operating System, Spring 2020

Homework 4

Tae Eun Kim 21400217, 21400217@handong.edu

1. Introduction

1.1 Goal

The ultimate goal of this homework was to implement an runtime deadlock monitoring program. To do so, I override the Pthread API with my own shared library, ddmon.so. This library works like a spy and sends information to the actual monitoring program, ddchck.c, via named pipe, fifo. More detailed technical approaches are to be discussed in section 2.

1.2 Requirements and Assumptions

The monitoring program, ddchck.c must maintain a lock graph that represents the locks held by the running threads. Whenever a cycle is detected, which means a deadlock, ddchck.c informs the user with an alert message. This message also contains thread ids and mutes addresses that are responsible for this deadlock.

The given assumptions are as follows. There are maximum 10 threads and mutexes used in the target program. Also, the fifo named ".ddtrace" is always given in the same directory where target program and the ddchck.c is located. It is assumed that ddchck.c is always restarted when the target program is restarted.

2. Approach

2.1 Overview

Here is a simple overview of the workflow. When the target program is executed, ddmon.so is loaded to the memory to be the first library to search for. Thus, every time the target program tries to call pthread_mutex_lock/unlock, it searches in the ddmon.so in prior to the actual Pthread API. The fake lock/unlock kindly calls for the real lock/unlock for the sake of the target program. However, the difference is that it sends the information to the monitoring program, ddchck.c. This communication is done through a named pipe, fifo named ".ddtrace".

Meanwhile, ddchck.c constructs a lock graph with the information sent by ddmon.so. It represents the locks held by the threads. Whenever ddchck.c discovers a cycle in the graph, it sends out an alert that deadlock is found.

2.2 Interpositioning

In order to get the information of the occurrences of some function calls, we need a spy. We place this spy between the target program and the actual library so it can catch the communication. Note that this term 'communication' is an analogy. Target program and the library don't talk to each other. Putting this spy is called in a more fancy way would be "inter-positioning". Thus, I also "inter-positioned" my shared library between the target program and the Pthread API. Inside ddmon.c, I made a fake lock and unlock functions which have the exact same name as the original ones. (I will refer to pthread_mutex_lock/unlock simply as lock/unlock) Then inside the functions, I made a function pointer and made them point to the real lock/unlock functions, so the target program may be able to achieve its goal.

```
int (*pthread_mutex_lockp)(pthread_mutex_t* m) ;
pthread_mutex_lockp = dlsym(RTLD_NEXT, "pthread_mutex_lock") ;
```

My shared library does not do less that the original library. It actually does some more. While providing the original functionality, ddmon.c also sends a message to ddchck.c. How it sends a message is discussed in the next section.

In order to make the target program use my library, I have to make its location the first place to look for undefined function calls. It can be done as below.

```
$ LD_PRELOAD="./ddmon.so" ./abba
```

2.3 Communication

The communication between ddmon.so and ddchck.c was made through the named pipe. A named pipe can be created with a "mkfifo" command. Its use is similar to the unnamed pipe. First, we need to make it. Then we need to open it and use the file descriptor to read or write on it. The important thing is, named pipe, in this case, fifo, is not writeable unless some process opened it as the read mode. So either the ddchck.c needs to always run first or the ddmon.so needs to open the fifo in O_RDWR mode.

There are 3 kinds of information that needs to be passed through the pipe. It can be answered with the following three questions. Was it a lock or a unlock? Who called this lock/unlock? What mutex is the subject of this lock/unlock? At the ddmon.so side, I constructed a single string that contains all the information. The first argument was an integer which is 0/1. 0 is for lock, 1 is for unlock. Then the string for lock would be "0 thread_id mutex_address". After it is sent, ddchck.c receives it and decomposes it into its needed format.

2.4 Lock Graph

The monitor, ddchck.c, maintains a lock graph that is updated by the information sent from ddmon.so. Every

time a thread calls for a lock, a node is added to the graph. If the thread was holding another mutex in prior to this one, an edge between two nodes is added to the graph. When a thread unlocks and releases a mutex, all the edges to the node that represent the mutex are removed. If no other thread is acquiring the mutex, node is also removed.

This is what happens when a lock information is received by the ddchck.c. 1) It firsts identifies if the mutex is a new one. If it is, a new index is assigned. Otherwise, the corresponding index is found. 2) Adjacency matrix of the graph is updated and adjacency list is created with it. 3) Figure out if a cycle exists. 4) Print alert message if there is a cycle.

I will elaborate on detailed steps. 1) I kept a list of mutexes to manage them efficiently. Whenever a lock information is given, ddchck.c checks if the mutex is new. Then if it is new, it adds the mutex to the mutex list, assigning an index. To use in the graph, mutex is represented as a node. This node has an index that corresponds to the mutex list, thread id of the thread that is currently holding the mutex. Thread id is updated only if the mutex was new or not occupied. 2) With the index of the mutex, the adjacency matrix is updated. This is the data structure that represents the edges in the graph. While adjacency matrix is only updated on every call, adjacency list is constructed from scratch with the adjacency matrix every time. 3) To find a cycle and identify the nodes that forms the cycle, I used the algorithm to find the strongly connected components. While running the depth first search with the adjacency list, I put every node I encounter into a stack. When a back edge is discovered, which means a cycle, I stop the search. Then I pop the stack until the node that the back edge was leading to is popped. These are the nodes that consist the cycle. 4) ddchck.c prints out all the mutexes and the thread ids that are responsible.

What happens at unlocking is simple. It identifies the index to see if it new. It the given mutex is new or it is not occupied, ddchck.c does not do anything because it there is nothing to do on this side. However, if the node is actually released, it updates the adjacency matrix and sets the node of that mutex to unoccupied by making the thread id 0.

3. Evaluation

Each component of the program, ddmon.so and ddchck.c worked well and produced expected results. I made 3 target programs to test this program.

3.1 Classic ABBA

First target program is a rather simple and classic one. There are 2 mutexes, 2 threads. Each thread is acquiring the mutexes in a opposite order. They sleep for a random amount of time between each locking and unlocking operation. This program may, or may not produce a deadlock depending on the amount of sleep they take. If a deadlock happens, it will be made by two threads and two mutexes. Let's see the results.

```
goodtaeeun@peace:~/OS_2020/hw4/implementation$ LD_PRELOAD="./ddmon.so" ./abba
thread 140170068887296 locked 0x6010e0
thread 140170056361728 locked 0x6010a0
thread 140170056361728 locked 0x6010e0
thread 140170068887296 locked 0x6010a0
```

```
goodtaeeun@peace:~/OS_2020/hw4/implementation$ ./ddchck
Deadlock detected!!
Mutexes involved:
0x6010a0
0x6010e0
Threads involved:
140170056361728
140170068887296
```

We can see that the threads took locks in a "abba" manner, producing a deadlock. Meanwhile, ddchck.c successfully detected the deadlock.

```
thread 140138506417920 locked 0x6010e0
thread 140138506417920 locked 0x6010a0
thread 140138493892352 locked 0x6010a0
thread 140138506417920 unlocked 0x6010a0
thread 140138506417920 unlocked 0x6010e0
thread 140138493892352 locked 0x6010e0
thread 140138493892352 unlocked 0x6010e0
thread 140138493892352 unlocked 0x6010a0
goodtaeeun@peace:~/OS_2020/hw4/implementation$
```

This would be a successful version, since the first thread managed to acquire both locks first.

3.2 Three threads

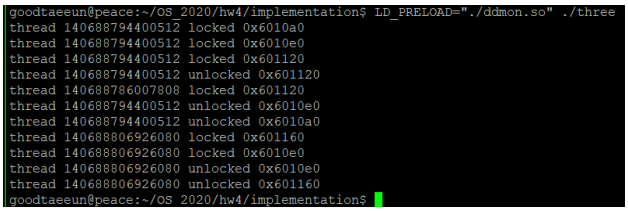
I named this target program because there are three threads. In this case, each thread acquires one mutex each. However, one thread acquires one more mutex, but is not related to the potential deadlock. I put it to see if my program excludes mutex in the graph that are not part of the cycle. Mechanism is similar to the abba example. If they are lucky, one of them might acquire both mutex before it is taken and there will be no deadlock. Otherwise, they will form a triangle of a deadlock. It is even scarier than the Bermuda triangle. Lets check the results.

```
goodtaeeun@peace:~/OS_2020/hw4/implementation$ LD_PRELOAD="./ddmon.so" ./three
thread 140431263627008 locked 0x601120
thread 140431272019712 locked 0x6010a0
thread 140431284545280 locked 0x601160
thread 140431272019712 locked 0x6010e0
thread 140431272019712 locked 0x601120
thread 140431284545280 locked 0x6010e0
thread 140431263627008 locked 0x601160
```

```
goodtaeeun@peace:~/OS_2020/hw4/implementation$ ./ddchc
Deadlock detected!!
Mutexes involved:
0x6010e0
0x601160
0x601120
Threads involved:
140431272019712
140431284545280
140431263627008
```

We can see that the mutexes that were called to lock was 4 in total. However, only 3 appears on the deadlock alert.

This means that ddchck.c distinguishes the nodes in and not in a cycle.



This is a lucky case when it is successful.

4. Discussion

Having the experience with dynamic linking was a new experience. I learned about it last year in computer architecture class. However, it seemed to me as something that happened under the surface that I cannot reach. I even could not clearly understand how it works. Now having the experience of creating and loading a shared library myself, I can understand much better how the dynamic linking works.

I had hard time with managing the lock graph. It was frustrating for me because it didn't seem like the main part of this homework. I finished the IPC with fifo and loading the shared library quite fast, but it took so much longer to implement the lock graph part. Additionally, at first, I could only figure out the two nodes that was the last link to the cycle. However, I figured out a way not only to discover the cycle but to identify the strongly connected components.

For suggestions, I came up with a idea that ddmon.c can stop the target program if the expected result is a deadlock. Then, it would not be a deadlock detector, but a deadlock preventer. The method is simple. It sends a message before it calls for the real lock. Then if ddchck.c tells that it will cause a deadlock, it aborts the call for locking. However, in this case, the target program must also have some resilient behavior to appreciate this functionality.

Another thing that I wanted to apply to my program was to automatically shut down or restart ddchck.c whenever the target terminates. There are some ideas. I can set a time limit to restart when there are no inputs for a certain amount of time. Or I can make ddmon.so to send a message when the target program terminates. Or I can check the list of processes at the ddchck.c side to monitor if the process of the target program is running or not. However I didn't have enough time to implement any of those brilliant ideas.

5. Conclusion

I built a program that monitors a deadlock situation. Pthread API was overridden by my own shared library, to send the locking and unlocking behaviors to the monitoring agent. The monitoring program maintains a graph that represents the locks held by different threads. Whenever encountering a cycle, it reports a deadlock alert, alongside with the thread ids and the mutex addresses involved in the deadlock.

The program proved its functionality with the target programs from simple abba examples to complex programs with more than 2 threads involved.

I had an experience of building my own shared library, and loading it to override existing libraries. It helped me understand how dynamic linking works. I thank professor Shin of offering me this precious chance.

Video Link: https://youtu.be/PNkRAszvE4I