ITP 30002-02 Operating System, Spring 2020

Homework 4

Jeon, Yeo Hun / 21500630 / 21500630@handong.edu

1. Introduction

In this assignment I am asked to implement a deadlock detector that makes alarm when a target program makes deadlock. It is expected that when this program runs with target program and there is a possible deadlock situation, the detector alerts user as stdout that the target program is in deadlock. And if it is possible, the detector also shows the line number of target program's source code where the deadlock occurs. This detecting system is divided into two part. one part is a dynamic link library called "ddmon" and the other one is called "ddchck", an individual checker program, receives thread and mutex information from ddmon and decides whether there is a deadlock or not.

The key approach of this deadlock detector system is a building lock graph based on the behavior of the target program. Since circular requesting of locks among multiple threads is condition of deadlock occurrence, the checker program, ddchck, decides the deadlock based on the existence of cycle in the lock graph.

To build the lock graph in runtime of the target program, the information of which thread holds which lock is needed. The dynamic link library, ddmon, comes out in this purpose. It has a wrapper function of lock and unlock of mutex, so it can extract the necessary information to build the graph before returns the lock result to the target program. After extracting the information, it writes the information into pipe, and checker program will read the pipe to build the lock graph.

Building lock graph is based on the passed thread identifier and mutex's address. The checker program has information which thread acquire which mutex. When the passed thread identifier from dynamic link library is does not hold any mutex, the checker creates new node and put it into linked list. If the given thread identifier is already in the list, the checker creates edge between the mutex it holds before and the newly given mutex by add the mutex into the list. When unlock behavior occurs in target program, the checker traverses the whole linked list and delete all nodes with the given mutex address. From this list, checker operates DFS traverse through the list and found out any possible cycle in the list.

2. Approach

The dynamic link library, ddmon, has a wrapper functions of "pthread_mutex_lock" and "pthread_mutex_unlock". By these functions, the original Pthread API functions are overridden. When the target program calls the functions, it hooks the call, and saves the original result of the function call by using "dlsym" function. Instead of returning the result, the wrapper function writes the thread identifier of thread who calls the function, and address of the mutex passed as the parameter from caller. The write destination is ".ddtrace", a FIFO pipe, which is shared between ddmon and ddchck. Therefore, by this wrapper function, the checker can get the necessary information to build lock graph whenever the target program calls pthread_mutex_lock and pthread_mutex_unlock.

By using above approach, it is needed to consider two issues. The first issue is that the form of writing into pipe from lock function and unlock function is exactly same. The checker, ddchck, has different behavior for lock and unlock. It needs to create new node or edge for lock behavior and delete node from the lock graph for unlock behavior. Therefore, when writing the thread and mutex information, ddmon adds operation identifier at the beginning part of the passing information. By this, the checker can distinguish which function is called in the target program.

The next issue is synchronization issue. Since there is multiple writing operation into pipe in different threads, there should be a synchronization control among the threads. However, by simply putting another mutex before and after the writing operation, the lock and unlock operation calls the wrapper function recursively. without any prevention, the wrapper function keeps writing the mutex address under the wrapper function into pipe. To avoid this issue, I set integer type value with static __thread keyword and increase the value by one whenever thread gets into the wrapper function and decrease the value by one when the thread returns. By using "static __thread" keyword, we can set the variable for each thread, not as shared variable. From now on, the writing action into pipe only occurs when the value is 1, which indicates the initial entrance to the wrapper function. In addition, the writing action between lock function and unlock function is also needed to be controlled. Therefore, each wrapper function

of pthread_mutex_lock and pthread_mutex_unlock has different “static __thread” variable called n_lock and n_unlock. For pthread_mutex_lock, it is only allowed to write into pipe when value of n_lock is 1 and value of n_unlock is 0. On the other hand, pthread_mutex_unlock is only allowed to write into pipe when value of n_lock is 0 and n_unlock is 1. With this mechanism, we can guarantee mutual exclusive between lock and unlock wrapper functions.

After getting thread and mutex information from the dynamic link library, the checker program needs to update the lock graph. Lock graph is implemented in array of linked list. Each node in the list contains structure with thread id, mutex address and pointer to the next. The head part of the list has the thread id and the first lock it acquires. If there is other mutex acquired by the same thread, the later mutex information is attached to the linked list of the array element pointing. Therefore, the array holds the unique thread id with mutex address it holds. Making the lock graph with given information from pipe is divided into 3 different scenarios.

The first case is that a thread acquires mutex lock without holding any other mutex lock. To check this, When the checker program gets information of mutex and thread with lock operation identifier from pipe, it looks up the array element and check if there is corresponding thread id with given one. If there is no thread id found in the array, it indicates that the thread does not hold any mutex at the point. In this case, the program creates new node with thread id, and mutex address and put it into the next of the last element of the array. This action is called “Add Node”

The second case is that a thread acquires mutex lock while having other lock(s). By same progress as described in the first case, the program determines whether the given thread already has the lock or not. If it is, it the newly read mutex address and thread id is made into struct and attached to the very last point of linked list connected to the array element. This action is called “Add Edge”

The last case occurs when the operation identifier read from pipe is unlock. At this situation, the program traverses the whole array and linked list of each array element and delete all nodes that have the given mutex address. Deletion in array is simply made by shifting the indexes and overwrite the target index. And deletion in linked list is conducted with disconnecting the link and make connection with node one before and one after the target node. This process can be considered as big overhead, but since the assumption of max number of thread and mutex is 10 for each, it would be fine by taking brute force strategy to delete the nodes. This action is defined as “Delete Node”.

The ultimate purpose of making lock graph is to find existence of cycle in the graph. Therefore, whenever the “Add Edge” operation conducted, the existence of cycle needed to be checked. The strategy to check cycle is finding back-edge by DFS. For each array element of the lock graph, the program conducts DFS and check if any node is visited again after its visiting is completely done. Since the lock graph is saved as a form of adjacency list of each thread id, it is very simple to have DFS. The program search for the child nodes of each node with lock id and keeps searching until it touches to the leaf node. When any node that already checked is tried to be visited again, it can be considered as a cycle. When the program detects the cycle, it needs to print out all the information of the threads and mutexes those are involved in the deadlock. Because of this, during the DFS traversing, the program records the path of the searching and print out all the path when the deadlock is detected.

With above fundamental detection, the program has additional functionality to show more detail information of deadlock. For this, the program uses backtrace() and addr2line operation to get line number where the deadlock occurs in target program. The backtrace() function collects given number log of function call and the address information. Since there is no possibility of deadlock occurrence in unlocking mutex, the backtrace() will be called for every locking mutex action. In this program, it is designed to collect last 10 function calls into string array. This array with function call information is passed to checker program via same FIFO pipe that used to deliver the thread and mutex information. This is because the decision of deadlock is made in checker program. Since the result of backtrace() contains all information of function calls includes those from linked library, the checker needs to have the target file name. When the checker starts, it requires target file name from user, and it compares the name with passed backtrace results. The results are collected as stack, which means the function called at the last save at the beginning index of the array. Therefore, the first string from the results that has the file name as substring will be saved as the possible faulty address when deadlock occurs.

In a point of user convenience, providing the faulty address is not efficient. Instead, the checker program converts the address into line number. This process is done by using “addr2line” command. To make this command in checker program, the program uses “popen” to open shell and make pipe with given command in string format. The address of

faulty function call is already saved from in the step of getting results of backtrace(), and the target program name is also saved as the initial argument of the checker program. Therefore, the program concatenates the information in "addr2line -e <target filename> <address>" and put it into popen function's parameter. As the result, the program can read the output of the command from file pointer used for the popen function and print it to stdout.

3. Evaluation

To evaluate the deadlock detection system, I set several questions.

1. Does the system successfully detect the deadlock?
2. Does the system does not make false alarm in ordinary execution of target program?
3. Does the system show the exact line number where invoke the deadlock?

To make answer to these questions, I prepared two test cases. First test case is a program using mutex with 'abba' pattern. In this program, two threads are created and one of the threads tries to acquire two mutexes and the other thread tries to acquire the same mutexes in opposite order. Therefore, deadlock will occur when the threads request lock held by each other. I tried the deadlock detection with this test file for several times. As the result, I could observe that the deadlock detection did not make an alarm when the target program successfully executed and terminated. When the target program made deadlock and abnormally running, the deadlock detector successfully detects the deadlock and prints out two threads and two mutexes as involved in the deadlock. In addition, it also shows the faulty line number. By manually checking the line, I found out that the detection points the next line of pthread_mutex_lock function.

The second test case is a program using 5 threads and with 5 mutexes. This program is very general case of deadlock named "dining philosopher's problem". In the program, the created 5 threads and they are all try to get a mutex from the next thread. when the all threads acquire the lock before any one of them release the lock, deadlock occurs. As the result, I could observe that the detector shows deadlock alert with five different threads and mutexes are involved. At this time, the pointed line number by the detection also the next line of actual faulty line.

4. Discussion

The most confusing part I experienced while implementing the system was the result of addr2line. Even though I turned off the compiler optimization option by giving -g O0 option, the detector consistently points the next line of the actual line where the deadlock is invoked. I am eager to know how and why this result came out.

Moreover, it would be the better program if there is a heuristic function in ddchck, the checker program. When user does not terminate the checker and keep running with multiple execution of the target program, the deadlock detection could be malfunctioned. This is because the variables in the checker are not re-initialized after the first execution of the target program. Therefore, it would be good to have function that detects the termination of the target program's execution and re-initialize all the variables and states of the checker. My idea is to clear all the variables and states when no thread or process holds lock or cycle in lock graph occurs. By this, the checker can be clear all the states and variables in both situation when the target program ordinarily terminated or blocked in deadlock.

5. Conclusion

In this assignment I implemented deadlock detection system using dynamic link library and individual checker program. The detection is made by reading thread and mutex information by catching mutex lock and unlock function by dynamic link library. The checker program gets it from the dynamic link library through FIFO pipe. Then the program makes lock graph based on the information and check whether cycle exists in the graph. Determining cycle is done by traversing the graph with DFS strategy. When the cycle is detected, the checker program prints the deadlock alert and information of threads and mutexes those are involved in the deadlock occurrence. In addition to detecting deadlock, the program also provides the information of line number of target program where deadlock is invoked. This function is implemented with backtrace to collect program name and address where function call made. The collected information is used to execute addr2line command with popen function to find line number from function address.

To evaluate the detecting system, I prepared two different test cases. Both of them are very general and common case of deadlock occurrence called 'abba pattern' and 'dining philosopher's problem'. As the result, I could observe the deadlock detection is successfully done, but getting faulty line with addr2line command consistently points one line next to the line where actually deadlock occurs.