Operating System PA3 REPORT

DEMO URL : https://youtu.be/mfHPWbRKB4g
21700398 Shin Yu Jin
21700646 Jeon Hye Won

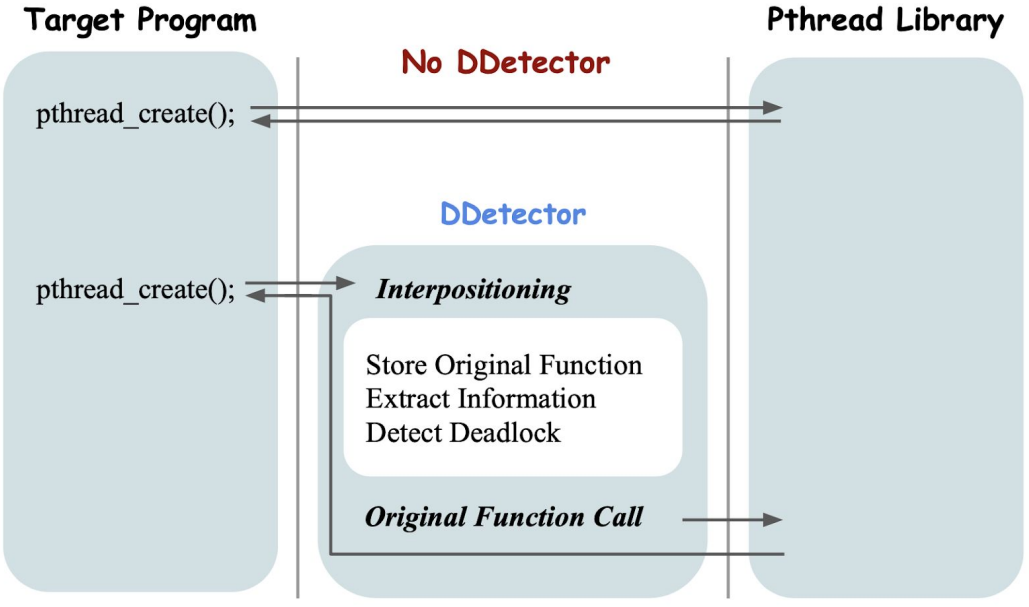
1. Intro

A. Problem Analysis

a. Deadlock Detection

'DDetector' simply refers to Deadlock Detector which notifies a user that deadlock has occurred when a target program is running, so DDetector has to be implemented by considering that the condition is online.

DDetector is one of the dynamic linking library (*.so: shared object file), and it works between a target program and pthread library. When a target program which uses multi-thread or lock is executed, its related functions such as pthread_create, pthread_join, pthread_mutex_lock, and pthread_mutex_unlock are also called. So before these real pthread functions are executed, DDetector interposes in a tiny gap and extracts the information needed to detect deadlock. And of course, DDetector should not affect the outcome or behavior of the target program.



For the deadlock detection, we need to check about graph because whether deadlock occurs or not is related to whether there is a cycle in graph or not. In detail, graph consists of some nodes and edges, and a node means mutex, so when lock function is called, a node which means this mutex is created in graph. In terms of an

edge, the direction of edge is important because it means the nested relation between mutexes. Through the process of lock and unlock, if there is a cycle between node ‘X’ and ‘Y’, it is a situation where ‘X’ and ‘Y’ are waiting for each other in order to hold the opponent’s lock. In short, this situation means that deadlock has occured.

b. Deadlock Prediction

`'DPredictor'` refers to Deadlock Predictor which shows a user the potential of deadlock. We can implement `DPredictor` by using the same principle of `DDetector`, but there are limits in this way. One is that at the moment `DDetector` catches the deadlock, a target program is already in deadlock, and it is not prediction, and because the `DDetector` creates and frees nodes and edges at that moment, the results may vary depending on the thread scheduling on each time the program is run. So we can predict deadlock without freeing nodes and edge, but it can make many false-positive predictions. For these reasons, we had to think of a different way.

`DPredictor` extracts the information in the similar way `DDetector` does, but `DPredictor` ifself does not have to be done in online. `DMonitor` observes the flow of lock and unlock and saves the extracted information in a desirable form, and DPredictor analyzes the potential of deadlock by using the given information, so each of them can be implemented dynamic linking library and normal C file in order. In addition, there is ‘Goodlock Algorithm’ for dealing with false-positive cases. To avoid these false-positive cycles, we have to remember three conditions. A cycle which created by one thread, a cycle with a gate lock, and a cycle which has a clear order in thread segment are false-positive cycles.

Lastly, we need to deliver the information from `DMonitor` to `DPredictor`. Whenever the data about lock and unlock is added newly while running a target program, `DMonitor` writes the information about edge at the file `'dmonitor.trace'`, and the sharing of information between `DMonitor` and `DPredictor` takes place through this file.

B. Solution Overview

a. Deadlock Detection

We declare a function of exactly the same name as the pthread function we want to hijack when the target program is executed. In addition, the `LD_PRELOAD` environment variable makes target program to find the function in the `DDetector` before the pthread library, and as a result, the function we defined is executed rather than the function of the pthread intended in the target program.

At this time, we save the current execution thread ID and address of the mutex variable to determine deadlock and call the original pthread function. Through this

process, we can obtain the information we want, and the target program can also be executed as intended.

b. Deadlock Prediction

The process of deadlock prediction is divided into two parts as mentioned before. One is to monitor, and the other is to predict. In `DMonitor`, we extract the information by using the same way like `DDetector`, but we need more information than `DDetector` to apply Goodlock Algorithm. This information is stored in linked list data structure or array, and the result of monitor is written in `'dmonitor.trace'` file. After this monitoring is complete, `DPredictor` reads this file and analyzes the potential of deadlock.

2. Approach - Solution Design

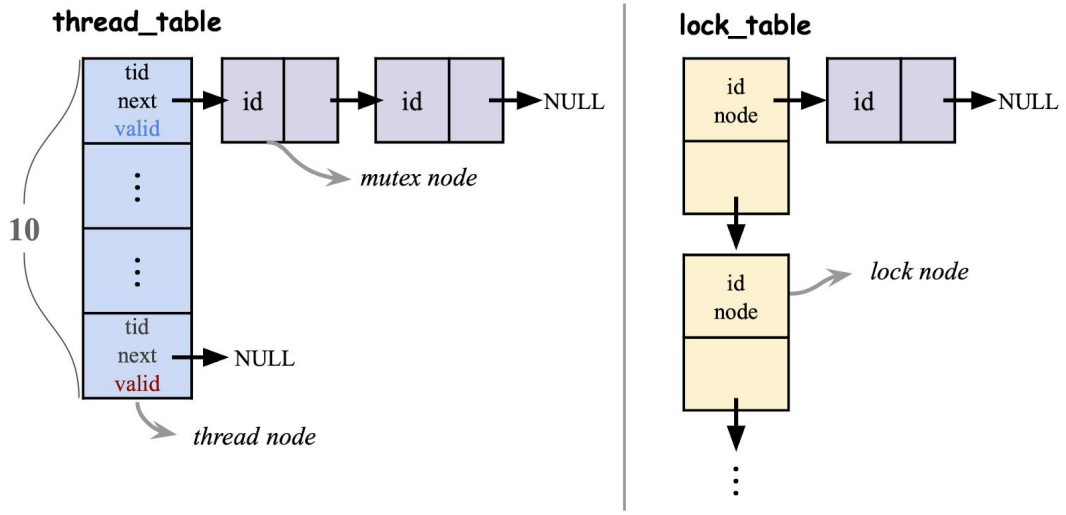
A. DDetector

a. Overall Data Structure

When we thought about the data structure for `DDetector`, we cannot help considering the given assumptions in pdf file. There are two assumptions:

- *A target program creates no more than 10 threads.*
- *A target program creates no more than 100 mutexes.*

To manage the memory more efficiently, we decided to use array for `'thread_table'` whose size is 10 and to use linked list for `'lock_table'` although we can also use array for `'lock_table'` because we thought that size 100 is so big enough, and a sparse matrix is not efficient. And in the case of thread, the threads are typically created when the target program runs and joined at the end of the program execution, but in the case of lock, it would be much more frequent to create and free from the graph. In such cases, the linked list might be more efficient than the array.



b. Timing to Change Data

In `DDetector`, when pthread lock functions is called, its new information about mutex is added in lock_table, and before the original pthread function is executed, `DDetector` checks whether there is a cycle in graph or not because if a target program is already in deadlock, `DDetector` cannot alert a user that deadlock has occurred. `thread_table` is updated at the time when it was guaranteed that deadlock did not happen after deadlock detection and execution of original function. This is because we can say that the thread has successfully held the lock only when the original lock function has been terminated.

B. DMonitor

a. Protocol Between DMonitor and DPredictor

As mentioned above, we had to store more information to apply `'Goodlock Algorithm'`, so we need function '`backtrace`'. However, we did not use it as it was because we thought that the return from backtrace is just address, and it is never guaranteed that memory will remain in that location when `DPredictor` uses this data. So we used structure for the data and copied the trace string to structure and saved the structure. Even if we need big memory space because the size of string is dynamic, saving whole structure was more consistent protocol.

b. Save Original Functions

Typically, the memory address of the original pthread function is stored within the hijacking function, and original function is called in a hijacking function. But in the dmonitor, several threads had to be accessed to `dmonitor.trace` to store the edge information. And because multiple access should be prevented from the a common resource, we called the original lock and unlock directly.

C. DPredictor

a. detect cycle

In `DPredictor`, not only did it detect cycle, but it also needed to have information on edge where cycle occurred because goodlock algorithm should check all pair of each edge. In this process, another structure was added to the `DDetector`'s cycle detection algorithm to add information on each edge. Through this information, we were able to obtain information about whether or not a cycle exists in the graph and the edges forming the cycle at the same time.

b. thread segment

In our view, the cases where dependency between codes exists are the case of new junction's appearance through the thread creation and through the function call. The latter of them can be solved through the condition that deadlock does not occur in the same thread, but the former case need check separately, so we came up with

thread segment graph which a new junction is added whenever a thread is created. We also thought that there is 'happens-before relation' between codes only in direct parents-child relation.

3. Manual

A. Makefile

The submitted zip file consists of simply DDetector.c, DMonitor.c, DPredictor.c, Makefile, and the some related example code files. The command '`make`' which is implemented through Makefile should be entered in the directory including all the files above. There is one more caution about `dmonitor.trace` file. If you execute this file repeatedly, the command '`make clean`' to remove this file is necessary.

4. Discussion

a. Difference in Two Dining Philosopher Version

We wondered what the difference was between the two dining philosopher examples that the professor gave us, the potential deadlock version and the deadlock free version. When we execute the deadlock free version with `DMonitor` and the `DPredictor`, we could check difference. The potential deadlock version does not consider the condition of other mutexes at the moment, and acquire the mutex on the left and the mutex on the right. However, the deadlock-free version waits until thread can hold both mutex, without holding or acquiring any other mutex. This condition makes to graph rarely generates edge, so graph can be free from cycle.

b. Influence on Target Program in Case of DMonitor

The program we create should have no side effect on target program's behavior, and have light-weighted. Because in multi-threading programming, one small thread scheduling can cause deadlock. However, in `DMonitor`, file I/O operations had to be performed each time an edge was created. And it had to affect the behavior of the target program.

In fact, when we run the program, we can see that the probability of deadlock is much higher when we run a program with dmonitor. We thought it was because we kept file writing the structure containing large amounts of information. To overcome this problem, there was a way for us to reduce the absolute size of the structure that writes the file, or to store the edge information to variable and write to the file at a time when the target program is finished. However, it was hard to change because the overall data structure of the `DMonitor` was already set, the `DMonitor` was a shared library, we could not confirm whether the target program is terminated or not.