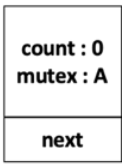
Dynamic Deadlock Detection&Prediction

21700583 JiHyun Lee
21400240 Hyeongjun Kim

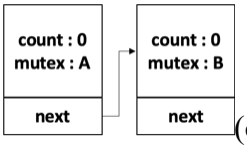
1. Online deadlock detector

A. Implement

In the deadlock detector, there are linked list structure. Because there is no limited number in mutex in the actual world, we use a linked list rather than an array. Each node has
`int count, pthread_mutex_t *mutex, struct Node *next.`

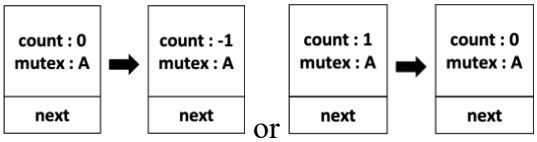


There can be three status of count
Count = 0 means one critical parts is holding that mutex.
Count = 1 means mutex in unlocked and free to use
Count < 0 means some critical parts is already using this mutex. And other critical is wating for this mutex.
In this program, there are functions to override pthread_mutex_lock and pthread_mutex_unlock.
When the target program calls pthread_mutex_lock, then the overridden function is executed.
When the mutex is first in use, the node is made and linked to the existing linked list. The new node's count value is initialized as 0. By searching the linked list, the deadlock detector can know whether this mutex is first in use or not.



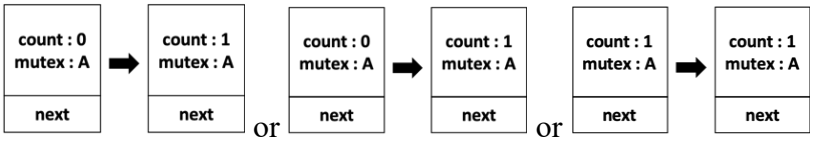
(case of new mutex B)

If the mutex already exists in the linked list, then decreasing count by one.



or

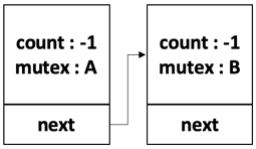
When the mutex is unlocked, then search for that mutex and increasing count by one.



or or

If there are no such mutex or, already it's count is 0, do nothing. Because mutex is like a binary semaphore mutex, the node's count variable cannot have a bigger value than 0.

For every change in the linked list, execute the detect function. Detection function checks whether all of the count value in the linked list is a negative number or not. If all count values are a negative number, it means all critical sections are waiting for the unlocked mutex.



(case of deadlock)

Therefore, it means a deadlock situation. Then deadlock detector print out deadlock message to console.

2. Offline deadlock predictor

A. Implement

Dmonitor

Dmonitor saves information of thread creation and mutex_lock. To fulfill these functions, Dmonitor overrides pthread_mutex_lock and pthread_create.

Monitor mutex

The structure for mutex information has
`long time; pthread_t thid; pthread_mutex_t *mutex;`
elements. `Thread Id` means current thread ID, `time` means current time(in Nanoseconds), and `mutex` means the address of mutex. Using that structure there are array
`struct Mnode monitor[threadNum][mutexNum];` which saves the information about lock mutex. Each row has the same thread ID.

Mutex A Thid 13 Time 0	Mutex B Thid 13 Time 1	Mutex NULL Thid 0 Time 0
Mutex B Thid 10 Time 2	Mutex C Thid 10 Time 3	Mutex NULL Thid 0 Time 0
Mutex C Thid 12 Time 4	Mutex A Thid 12 Time 5	Mutex NULL Thid 0 Time 0
Mutex NULL Thid 0 Time 0	Mutex NULL Thid 0 Time 0	Mutex NULL Thid 0 Time 0

(Example of `monitor` array)

When mutex_lock is called by target program, `addToMonitor()` works.
In this function, first, it finds a proper location for the mutex.
It took two steps;1) Finding proper row by threadID, 2) Finding the tail of that row and locate the mutex node. If there is no same threadID, then locating it to the new row.

Monitor thread creation

Also, Dmonitor uses overridden pthread_creation and gets information by this. This information is saved in struct thEdge thEdges[threadNum];
Each node in the thEdges array is thEdge which has
`long time; pthread_t src; pthread_t dest;` elements.

Src 13 Dest 10 Time 1.5	Src 10 Dest 12 Time 3.5	Src 12 Dest 13 Time 4.5

When target program calls pthread_creation, then overridden pthread_creation call void addTothEdges(pthread_t present_th, pthread_t new_th)
In addTothEdges, it saves present thread Id in src, new thread Id in dest, and present time(in Nano second) in time variable.

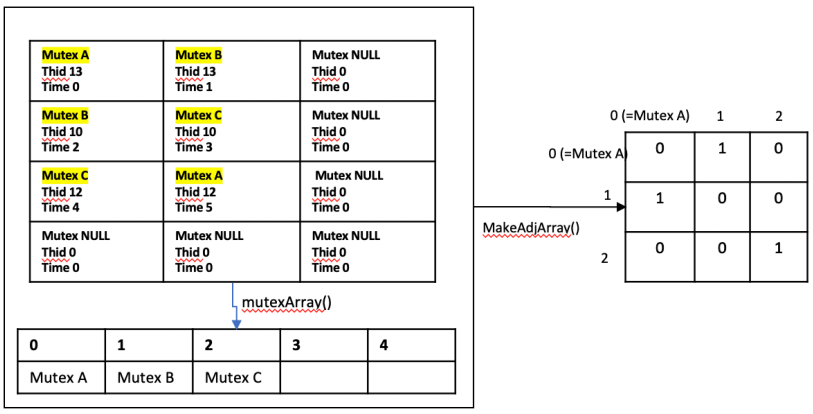
Call Backtrace and print out to the dmonitor.trace file
After doing that, call `Backtrace()` and get second stack form trace array(which point the target program's line). And print out mutex monitor information, thread monitor information and backtrace information to the dmonitor.trace file
After all these works finished, then each overridden functions write monitor's information and thEdges's information in dmonitor.trace file.

DPredictor

Read file

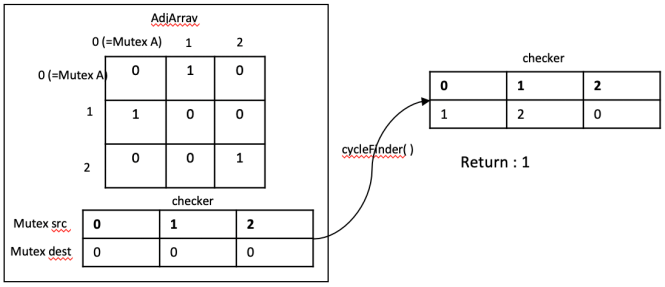
DPredictor analyzes data from Dmonitor. To do that, first read all data in dmonitor.trace and save it in struct Mnode monitor[threadNum][mutexNum]; and struct thEdge thEdges[threadNum]. They are the same as Dmonitor's array. To predict, DPredictor has to find a cycle. The steps of detecting the cycle are as follows.

Make AdjArray



Frist, read monitor array to find src to dest relation. In `MakeAdjArray()` makes AdjArray to detect cycle easily. Additionally in `MakeAdjArray()`, there are sub function `mutexArray()` makes array dinctionary to convert mutex as integer number. With the information of mutex dictionary(=`mArr[mutexNum]`) and monitor, `MakeAdjArray()` makes adjarray.

Find cycle



After that, make an array that has the size of the distinct number of mutexes.

By using `mutexArray()` Predictor can get the number of distinctive mutexes. In the case of example picture, its value is 3('A',' B' ,'C'). By this number, create a checker that is used for saving cycled mutex information.

After make checker, execute `cycleFinder()`. `cycleFinder()` use DFS algorithm, to detect cycled. `cycleFinder()` returns integer value which is one of the index of mutex in cycle. If there was no cycle, then `cycleFinder()` return -1;

Now return to main, if the return value of `cycleFinder()` is -1 then exit the DPredictor with the "safe" message .

Make edges array

If the return value is not -1, then it means cycle detected and cycle information is saved in a checked array. The return value means one of the indexes in the cycle. By this mutex index, `counterMutex()` can find the number of mutex in the cycle.

Time 0 Thid 13 Src A Dest B Guard NULL	Time 2 Thid 10 Src B Dest C Guard NULL	Time 4 Thid 12 Src C Dest A Guard NULL

(example of edges[])

After finishing all above, then make edge array "`edges[cycledMutex]`". Edge is a structure with `long time; pthread_t thid; pthread_mutex_t *src;pthread_mutex_t *dest;pthread_mutex_t *guard;` elements.

Using checker, cycledmutex number and index(which was the return value of cycleFinder. Index means one of mutex index participate in the cycle), `fillEdges()` fills the edge array.

Checking cycle's validity

After filling the edge array, now DPredict checks the validity of the cycle.

In check1, checking the thread ID `of Edges[]`. If more than two of mutex's thid are the same, it means the cycle is no danger.

In check 2, checking the guard mutex. In the Edge node, there are guard elements. Check2 checks if mutexes have the same guard or not. If two of them have the same guard, it means no danger.

In check3, checking the thread segmentation by time. If the mutex cycled part is executed earlier and thread creating executed later, it means safe. To do this, check3 checking mutex's made time and also checking thread made time. If thread made time is later than mutex's made time, then it means no danger.

Backtrace analysis and output

After checking all, if the cycle is judged as a danger, then read backtrace information in dmonitor.trace file and tokenize it. In order to use the addr2line() function, executable file name and address value are needed

to be parsed from the stack. In order to extract the two pieces of data information from the stack, we store the start and end addresses of required data into a structure called a token. By using that executable file name exists between the characters '/' and '(', and the address value exists between '[' and ']', we extracted the data we have needed.

The function that handles this data processing is called split_store_token(). Also, since the command for addr2line should be used as a console, we need to use system() function, which enables command for the console to be also executed in the source file. Therefore, we concatenated the above all data with strcat() function and use the function addr2line() using the system () function.

After that, DPredictor prints out the danger line of the target program and also prints out danger thread num using Edge array's thid.

B. Limitation

By using our model, the user can get a prediction of deadlock. However, there are some limitations to our program.

First, when making AdjArray, MakeAdjArray( ) reads the monitor[] one by one. For example, if the monitor array has A B B A value, then MakeAdjArray( ) think that it has A->B, B->B, B->A relation which was not my intent. My intent was A->B B->A. Because I didn't hook the unlock function, those malfunction happens. To fix that error, I give one condition telling that if src.mutex and dest.mutex are the same, just move next. By using this condition, I fixed the related problem but it has side effects. My predictor can not predict B -> B(self cycle) as a deadlock.

Secondly, DPredcitor can only report error messages one by one. In a real compiler, they give all errors and warning message but my DPredictor can only report one issue at once. Making multiple predictions was too complicated to make.

Finally I'm not sure whether this program will work properly in any kinds of deadlock or not. I only made predict program for abba deadlock and dinning problem deadlock and some false positives. Ther might be far more situation that I didn't covered.

C. Difficult point and how solve it

Linked list to array

At first, we want to use linked list rather than array. Because in the real world there is no limit of thread and mutex. However, it is very complicate job. So, we turned to array.

Index – mutex dictionary

Even we using array, there were many things to think to saving data efficiently. One way of simplifying job is using dictionary. By using index-mutex dictionary, making AdjArray and other process become easier.