ITP 30002-02 Operating System, Spring 2020
Homework 4
21800500 Goeun Lee, 21800500@handong.edu
https://www.youtube.com/watch?v=xADVntLEURA

1. Introduction :

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. Dead lock detector consists of probe part and checker part. Probe part override *pthread_mutex_lock* and *pthread mutex_unlock* and emits thread is, mutex address into named pipe when target program which we want to detect deadlock try to lock or unlock a mutex. Check part that receive emitted information within named pipe is an independent program which differ from probe part located between target program and pthread API. Checker part keeps drawing graph with tid and mutex address. While drawing graph, if a cycle is made alert that deadlock is detected. A cycle is not necessarily a deadlock, but this assignment suppose if there is any cycle, report it as deadlock.

I approached to this assignment with adjacency matrix. There are two main types of deadlocks that I understand, the first is the form in which you find yourself when you follow the mutex that has a dependency on each other. The second one is that when a process locks a mutex, the process try lock the mutex once again. To find deadlock, a matrix that shows relation between processes and mutexes and matrix that explains relations between mutexes and mutexes. The relation between process and mutex means that a process locks a mutex or a process want to lock a mutex which is locked already by another process so that the process cannot lock it yet. Let's suppose a situation that a mutex called 'a' is locked by a process called 'A', a mutex called 'b' is locked by a process called 'B'. If process 'B' try to lock mutex 'a', process 'B' has to wait in waiting queue until mutes 'a' is unlocked. Until process 'B' wake up due to unlocked mutex 'a', As mutex 'b' is tied up, mutex 'b' can be considered to have a dependency on mutex 'a'. I defined a relation between mutexes as this kind of dependency between mutexes.

2. Approach :

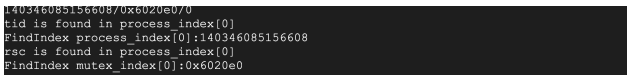
As I explained in the Introduction section, there are two cases in Deadlock. It is the case where several mutex constructs a cyclic structure and the case in which it constructs a cyclic structure on its own. I implemented the detection of these two cases using two matrices.

- indexing

Two 10x10 matrix are required. Each is a matrix that represents the relationship between the process – Mutex, Mutex – Mutex, and indexed each thread id and mutex address to make more convenient to implement the matrix. To save tid and mutex address, I declared int * mutex_index, unsigned long * process_index and dynamically allocated 10 spaces. Mutex is declared to be a structure called pthread_mutex_t, so I tried to make it a dynamic allocation to pthread_mutex_t** when creating an index array to store the address of the mutex, but it was found that it could not be declared pthread_mutex_t**. Thus, I used int since the address value was the same as 32bit. When I compile it, there is a warning message, but it was possible to use it.

```
140346085156608/0x6020e0/1
tid is new one : 0
FindIndex process_index[0]:140346085156608
rsc is new one : 0
FindIndex mutex_index[0]:0x6020e0
```

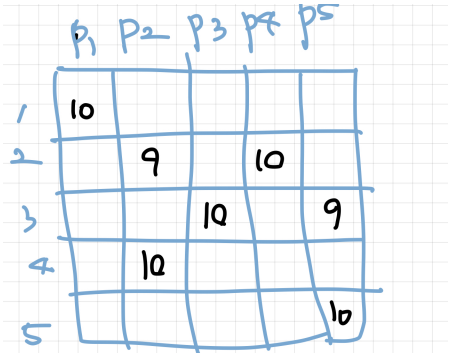
This is the result of running ddchck in debug mode. The first line of the output sentence means that the process of 140346085156608 tid attempted to lock the mutex of address 0x6020e0. It was the first tid to enter ddchck, so it was indexed to 0 and the address of mutex was also the first address to enter ddchck, so it was indexed to number 0. If the tid is already stored in process_index, use that index and grant a new index if it is not in process_index.



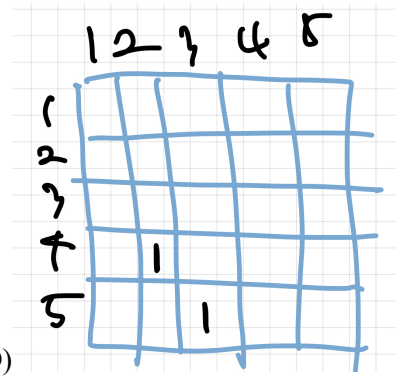
In the second picture, you can check the output that the tid and rsc were found, respectively, because the information is already stored in process_index and mutex_index when locked.

- lock

I'll call the two matrices P and D Matrix respectively. The former is the matrix that shows the relationship between the process and the mutex, and the latter is the matrix that represents the duration between the mutex. The two matrices are initialized to zero. When an attempt of lock is made through FIFO, the address of the tried tid and mutex is stored in a variable named long tid , int rsc and finds the index. Mark on the P[rsc index][tid index].
If P[rsc index][] does not have 10, mark 10. 10 in P[rsc index][] means that another tid is already locking the mutex. In this case, mark the lowest number of non-zero numbers minus 1. If P[rsc index] has one 10-person compartment, one 9-person compartment, and the rest is filled with 0, P[rsc index][tid index] becomes 8. Numbers greater than zero and less than ten indicate waiting priority. P[][tid index] has 10 but P[rsc index][tid index] is not 10, which means that the tid is locking one mutex while attempting to lock another mutex and is waiting in wait queue. In this case, the mutex, which has already been locked, has caused the process to wait in the waiting queue, to become dependent. In other words, D[P[][tid index] == 10][[P[][tid index] != 0 & & P[][tid index] != 10] should indicate the presence of defendency. Dependency is divided into 1 and 0.



Ex P)



Ex D)

P5 is already locking mutex 5, but is waiting at waiting queue while attempting locking mutex 3. As such, by marking 1 in D[5][3], we saved that mutex5 has a dependency in mutex3.

- unlock

When a thread attempt to unlock a mutex, P[rsc index][tid index] will become 0. If there are numbers lower than 10 but greater than 0 in P[rsc index][], Add 1 to them. Surely, D matrix has to change following the situation. Let us suppose x is an index of P[rsc index][] become 10 by adding 1. If there is in D[rsc index], set D[rsc][x] as 0.

- Detect deadlock

After setting things up so far, you can detect deadlock easily. In case of self deadlock, you can check within P matrix. When you set P matrix with lock input and there is 10 already, It is self-deadlock. If you attempt to set P matrix, you may know the mutex address and tid. In case of another case, you can check within D matrix. You have to use DFS to find the cycle. Attempting to visit a node that has already been visited while exploring with DFS means that a cycle has occurred. In D matrix, you can only know mutex index. Thus, find the tid number and mutex address with mutex_index and process_index.

- **Add ddmon to ddchck**

Create a child process using fork() and load ddmon.so and execute the target program. I programmed assuming
"gcc -shared -fPIC -o ddmon.so ddmon.c -ldl" has already been executed and ddmon.so file exists. I executed “LD_PRELOAD=./ddmon.so targetprogram” with system() function.

3. Evaluation :

- ddmon

As target program is executed, information such as tid, mutex address, lock/unlock is written in FIFO in real time. Since pthread_mutex_lock is overridden, naturally when the target program attempts to lock, pthread_mutex_lock within the ddmon is automatically called. Suppose the dlsym command returned the original pthread_mutex_lock to the function pointer called normal_lock. In the case of deadlocks, the information was written to the FIFO first and then called the original pthread_muthex_lock because it was caught in the deadlocks as soon as it was executed. Execute the original pthread_mutex_lock and receive the return value and return it to the target program from the ddmon.

- ddchck

As I executed "LD_PRELOAD=./ddmon.so targetprogram" in ddchck.c with system() function, you should enter name of target prgram with ./ddchck. Whenever the lock or unlock information comes in, ddchck draws a graph by updating the P and D matrices. Explore the D matrix with DFS, and if cycle is found, detect it with deadlock and output the deadlock and mutex address. As suggested in the task, if one thread locks one mutex, it is added to the P matrix, and if X is held and Y is acquired, it is added to the D Matrix. If X is released, P and D matrix were modified. If a cycle is detected by dependency alone, it is not included in the case of having a cycle by itself, so a separate condition was given.

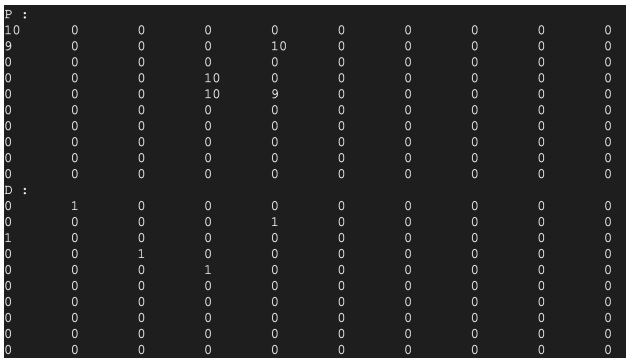
Like When a thread acquire lock or unlock, tid, mute address, and lock/unlock are included message which is delivered by FIFO, I intended to include line number with them and the line number is printed out when deadlock is detected, but I could not completed.

4. Discussion :

The subject that penetrated into one from Assignment 1 to 4 was to intercept and customize what the computer originally did. In the case of the program of this task, if a cyclic graph is drawn while running simultaneously with the target program, it was detected as deadlock, but I wonder if there is a way to detect deadlock a little earlier. In particular, I know that random numbers are generated using the current time, but it seems to be possible to detect deadlock by generating random numbers in the same order assuming the situation after 10 seconds. The algorithm that detects deadlock also seems to be able to detect the real deadlock by adding various conditions, but it seems that it would be more inefficient to do the calculation for every code for deadlock that happens very often in a typical program.

5. Conclusion

This program overrides pthread_mutex_lock and pthread_mutex_unlock of pthread API and emits tid,mutex address to external function in real time to detect deadlock. The external function created two matrices, one to know which processes have which mutexes, and the other to see if there is a dependency between mutex or not. If you run ddchck with debug mode, you can check in realtime the values of each matrix.



If you look at P[0][0], P[0][1] of this picture, process 0 is holding mutex 0 and waiting to acquire mutex 1. Since 1 is written in D[0][1], it means that mutex 0 has a dependency on mutex 1. Searching D matrix with DFS, you can check whether each situation is cyclic or not. Thus, with these two programs, you can detect deadlock in real time even if it's a cycle by itself.