ITP 30002-01 Operating System, Spring 2020

Homework 4

21600212 Nam Jinwoo | 21600212@handong.edu | Demo video < https://youtu.be/jOLdFVCnvas >

1. Introduction

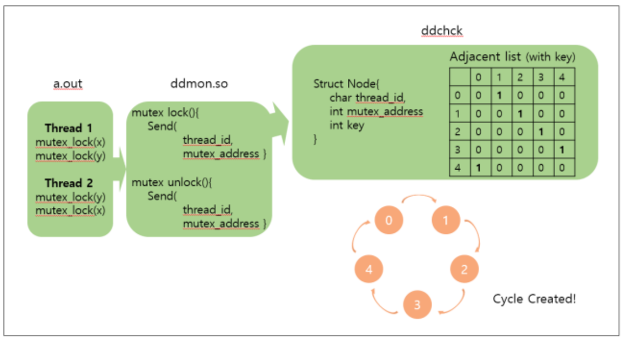
Multi-Thread programming has many advantages like shared recources. To modify shared resource, programmer needs to be careful. They need to block other threads from accessing same resource at the same time. That's why 'pthread_mutex_lock()' function is needed. However mutex lock also must be handled carefully. Otherwise, "Dead Lock" could happen.

In order to provide convenience to programmers to avoid deadlock, in this paper I propose "Dead Lock Detector (DLD)". It automatically detect whether deadlock ocurred or not and print out which threads and mutexes are locked in deadlock. Develop DLD requires me three essential techniques.

1. How to interpose runtime library.
2. How and what to be provided to detect deadlock.
3. How to detect deadlock.

2. Approach section descibe these techinques.

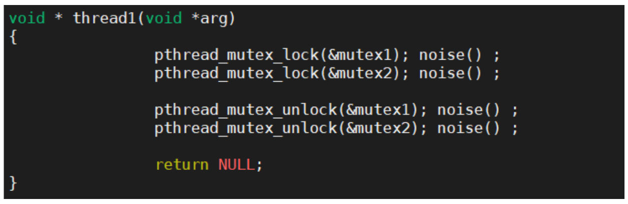
2. Approach



	0	1	2	3	4
0	0	1	0	0	0
1	0	0	1	0	0
2	0	0	0	1	0
3	0	0	0	0	1
4	1	0	0	0	0

< Figure 1. Overall workflow of the DLD >

Figure 1. explains the overall workflow of DLD(Dead Lock Detector). First, when program a.out execute mutex lock function, interpositioning occurs. 'ddmon.so' interrupt lock function to get information about which thread is trying to access, which mutex is involved. After ddmon gets information, it sends those information to 'ddchck'. Then, ddchck checks whether now program a.out has deadlock or not. When ddchck detect deadlock, I build adjacent list graph to find dependency cycle is made or not. I introduce details on the 2.2 How to detect deadlock.

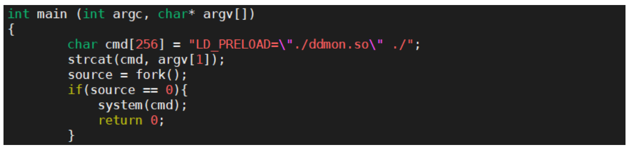


```
void * thread1(void *arg)
{
                pthread_mutex_lock(&mutex1); noise() ;
                pthread_mutex_lock(&mutex2); noise() ;

                pthread_mutex_unlock(&mutex1); noise() ;
                pthread_mutex_unlock(&mutex2); noise() ;

                return NULL;
}
```

< Figure 2. mutex lock example a.out>

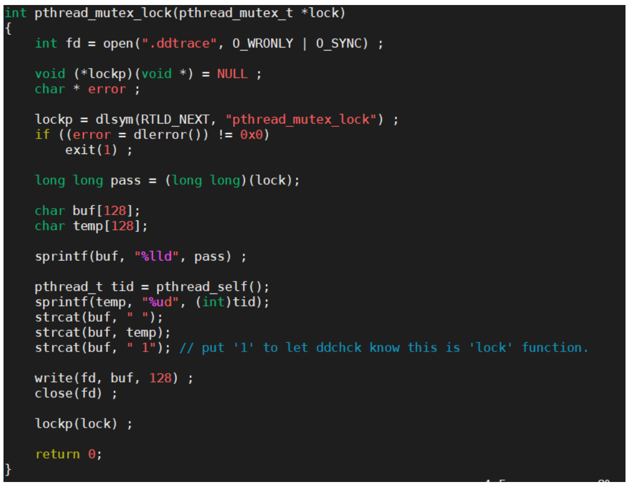


```
int main (int argc, char* argv[])
{
        char cmd[256] = "LD_PRELOAD=\"./ddmon.so\" ./";
        strcat(cmd, argv[1]);
        source = fork();
        if(source == 0){
            system(cmd);
            return 0;
        }
```

< Figure 3. ddchck execute a.out with ddmon.so >

2.1 How to interpose rutime library

Figure 3. shows how to interpose runtime library. In order to detect deadlock with a.out, you need to execute ddchck with argument a.out (e.g ./ddchck a.out). Then, ddchck runs a.out with ddmon.c which sends thread and mutex information.



```
int pthread_mutex_lock(pthread_mutex_t *lock)
{
    int fd = open(".ddtrace", O_WRONLY | O_SYNC) ;

    void (*lockp)(void *) = NULL ;
    char * error ;

    lockp = dlsym(RTLD_NEXT, "pthread_mutex_lock") ;
    if ((error = dlerror()) != 0x0)
        exit(1) ;

    long long pass = (long long)(lock);

    char buf[128];
    char temp[128];

    sprintf(buf, "%lld", pass) ;

    pthread_t tid = pthread_self();
    sprintf(temp, "%ud", (int)tid);
    strcat(buf, " ");
    strcat(buf, temp);
    strcat(buf, " 1"); // put '1' to let ddchck know this is 'lock' function.

    write(fd, buf, 128) ;
    close(fd) ;

    lockp(lock) ;

    return 0;
}
```

< Figure 4. ddmon.c mutex lock interposition >

When pthread_mutex_lock() is called on a.out program, it calls ddmon.so instead of original library. Then ddmon.so can collect mutex lock information and send it to ddchck. After all modified process, it calls original mutex lock library.

2.2 How and what to be provided to detect deadlock.

What to be provided:

thread id, mutex address, option(lock or unlock)

In order to detect deadlock, ddchck needs to check mutex dependecy cycle is made or not. So to make nodes and graph at the ddchck, not only mutex address but aslo thread_id is also essential. Because we need to distinguish which thread is trying to lock this mutex. According to thread id, ddchck action changes.

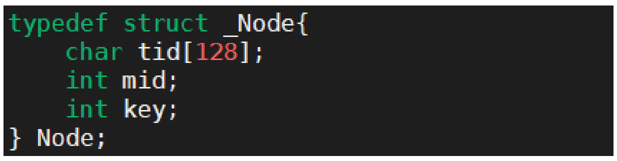
Since there are two functions which are lock() and unlock(), ddchck needed to distinguish them. To help distinguishing, ddmon provides which function is it by add option code at the end of the message. Option ‘1’ means lock() and option ‘2’ means unlock().

How to send information: FIFO pipe line

In order to send collected information from ddmon, I use FIFO pipe line. ddmon and ddchck are being run by different processes. To communicate with each other, those need a pipe line, and I used an API called FIFO.

I assume that FIFO file .ddtrace is already make before ddchck starts to run. “mkfifo .ddrace” command allows you to make FIFO pipe file. I wanted to send all information at once, that is why I use strcat to merge all information in one string.

2.3 How to detect deadlock



```
typedef struct _Node{
    char tid[128];
    int mid;
    int key;
} Node;
```

< Figure 5. struck of node with key >

In order to detect deadlock, ddchck first neet to make graph of nodes with provided information from ddmon. However thread id and mutex address seems not proper to make graph. So I make struct of node with ‘key’ value which is simple integer number that helps me to build graph easily.

And then I could find out that I should not create node when coming mutex is already exist in graph. Therefore, I divide ddchck’s behavior into four cases according to incoming information.

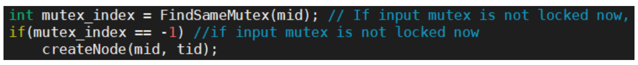
Case 1. Incoming mutex is new. (not in the graph)

Case 2. Incoming mutex is not new and it’s owner is incoming thread.

Case 3. Incoming mutex is not new and it’s owner is different with incoming thread.

Case 4. Incoming option is unlock() option.

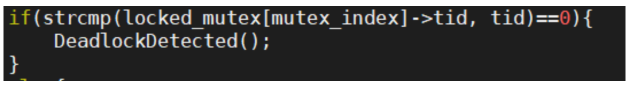
Case 1.



```
int mutex_index = FindSameMutex(mid); // If input mutex is not locked now,
if(mutex_index == -1) //if input mutex is not locked now
    createNode(mid, tid);
```

If incoming mutex is not in the graph, then ddchck the create new node and add it to graph. As I mentioned, node has thread id, mutex address and key value.

Case 2.



```
if(strcmp(locked_mutex[mutex_index]->tid, tid)==0){
    DeadlockDetected();
}
```

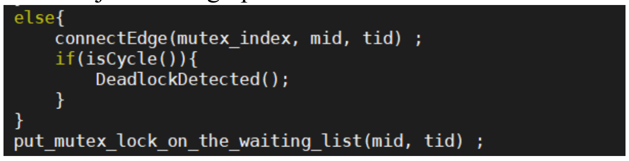
If incoming mutex is already exist, and incoming thread is it’s owner, then this means that self-deadlock occurs.

If thread try to lock mutex which is already locked, it should wait until that mutex is getting unlocked. However, what if owner of locked mutex is itself? No other threads are available to unlock that mutex, so it will stop forever. In other words, it gots deadlock.

Case 3.

If incoming mutex is not new and it’s owner is incoming thread. Then this means that incoming thread becomes dependent on the owner of the mutex. In other words, it can not do anything until mutex owner unlock that mutex.

At this time dependency between incoming thread’s mutexes and other thread’s mutex is made. Therefore, edge of the adjacent list graph need to be drawn.

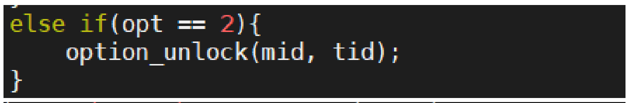


```
else{
    connectEdge(mutex_index, mid, tid) ;
    if(isCycle()){
        DeadlockDetected();
    }
}
put_mutex_lock_on_the_waiting_list(mid, tid) ;
```

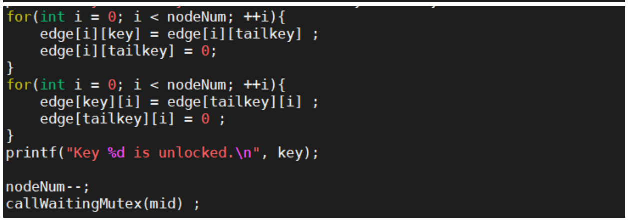
Since new edge is created, ddchck check whether there is cycle or not. If yes, it means that deadlock is detected. I will not explain detail with finding cycle in graph. I just use DFS search to find cycle. If there is no cycle, continue to monitoring.

And at this time, we should remember mutex lock operation is ‘waiting’ not ‘terminated’. So, createNode() function need to be called when that mutex is unlocked. So to make it possible, ddchck put provided information on the waiting list. This will be used in the unlock function.

Case 4.



```
else if(opt == 2){
    option_unlock(mid, tid);
}
```



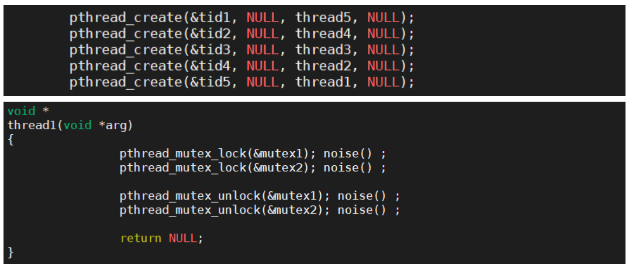
```
for(int i = 0; i < nodeNum; ++i){
    edge[i][key] = edge[i][tailkey] ;
    edge[i][tailkey] = 0;
}
for(int i = 0; i < nodeNum; ++i){
    edge[key][i] = edge[tailkey][i] ;
    edge[tailkey][i] = 0 ;
}
printf("Key %d is unlocked.\n", key);

nodeNum--;
callWaitingMutex(mid) ;
```

If option is unlock, then ddchck delete that node and remove all edges connected to that node. To facilitate maintenance of the Graph, fill an empty space with the node at the end.

After unlocking, it’s time to check waiting list if there is thread that want to lock mutex which is same with unlocked mutex.

3. Evaluation



```
        pthread_create(&tid1, NULL, thread5, NULL);
        pthread_create(&tid2, NULL, thread4, NULL);
        pthread_create(&tid3, NULL, thread3, NULL);
        pthread_create(&tid4, NULL, thread2, NULL);
        pthread_create(&tid5, NULL, thread1, NULL);
```

```
void *
thread1(void *arg)
{
                pthread_mutex_lock(&mutex1); noise() ;
                pthread_mutex_lock(&mutex2); noise() ;

                pthread_mutex_unlock(&mutex1); noise() ;
                pthread_mutex_unlock(&mutex2); noise() ;

                return NULL;
}
```

< *Figure 6. example program* >

I wrote example programs to demonstrate my Dead Lock Detector accurately detects deadlocks. It has 5 threads and each try to lock two mutexes.

Thread 1: lock(mutex1), lock(mutex2)

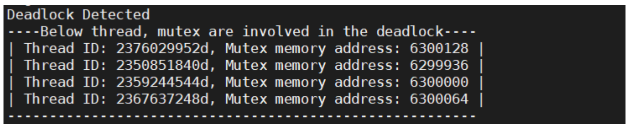
Thread 2: lock(mutex2), lock(mutex3)

Thread 3: lock(mutex3), lock(mutex4)

Thread 4: lock(mutex4), lock(mutex5)

Thread 5: lock(mutex5), lock(mutex2)

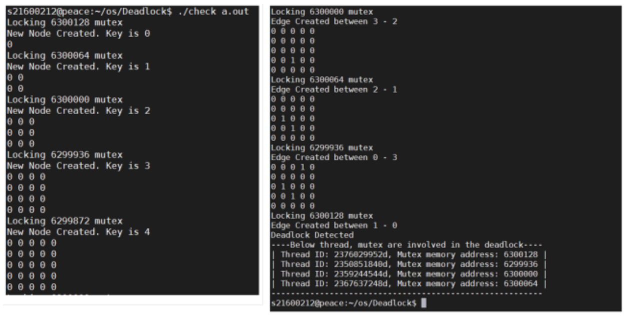
This example program occurs cycle [2-3-4-5-2].



```
Deadlock Detected
----Below thread, mutex are involved in the deadlock----
| Thread ID: 2376029952d, Mutex memory address: 6300128 |
| Thread ID: 2350851840d, Mutex memory address: 6299936 |
| Thread ID: 2359244544d, Mutex memory address: 6300000 |
| Thread ID: 2367637248d, Mutex memory address: 6300064 |
--------------------------------------------------------
```

< *Figure 7. Dead lock detected result* >

I can see my DLD detect dealock accurately. And also DLD shows us information of involved thread and mutex's address. In order to more precise evaluation, I print out node creation, edge creation, and adj graph.

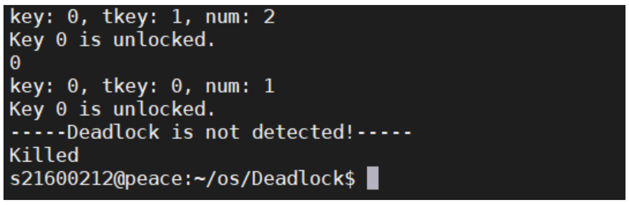


```
s21600212@peace:~/os/Deadlock$ ./check a.out
Locking 6300128 mutex
New Node Created. Key is 0
0
Locking 6300064 mutex
New Node Created. Key is 1
0 0
0 0
Locking 6300000 mutex
New Node Created. Key is 2
0 0 0
0 0 0
0 0 0
Locking 6299936 mutex
New Node Created. Key is 3
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
Locking 6299872 mutex
New Node Created. Key is 4
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

```
Locking 6300000 mutex
Edge Created between 3 - 2
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 1 0 0
0 0 0 0 0
Locking 6300064 mutex
Edge Created between 2 - 1
0 0 0 0 0
0 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 0 0
Locking 6299936 mutex
Edge Created between 0 - 3
0 0 0 1 0
0 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 0 0
Locking 6300128 mutex
Edge Created between 1 - 0
Deadlock Detected
----Below thread, mutex are involved in the deadlock----
| Thread ID: 2376029952d, Mutex memory address: 6300128 |
| Thread ID: 2350851840d, Mutex memory address: 6299936 |
| Thread ID: 2359244544d, Mutex memory address: 6300000 |
| Thread ID: 2367637248d, Mutex memory address: 6300064 |
--------------------------------------------------------
s21600212@peace:~/os/Deadlock$ █
```

< *Figure 8. more detailed process printing* >

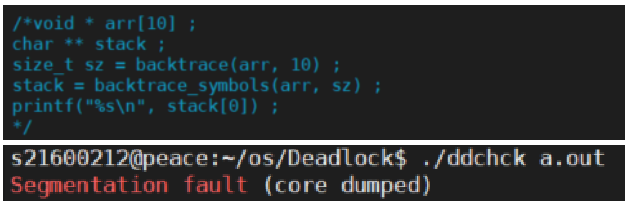
Otherwise, if I remove thread 5's lock(mutex2), DLD works properly print out "Deadlock is not detected".



```
key: 0, tkey: 1, num: 2
Key 0 is unlocked.
0
key: 0, tkey: 0, num: 1
Key 0 is unlocked.
-----Deadlock is not detected!-----
Killed
s21600212@peace:~/os/Deadlock$ █
```

4. Discussion

After build all requiredments, I've tried to build Exra Point problem which shows the source code line numbers where one or more mutexes involved in a deadlock are acquired. However I failed to complish it. I thought I fully understand professor's backtrace lecture, but I cound't find what's wrong with my code.



```
/*void * arr[10] ;
char ** stack ;
size_t sz = backtrace(arr, 10) ;
stack = backtrace_symbols(arr, sz) ;
printf("%s\n", stack[0]) ;
*/
```

```
s21600212@peace:~/os/Deadlock$ ./ddchck a.out
Segmentation fault (core dumped)
```

< *Figure 9. backtace failed* >

I just copy backtrace part of professor's code to mine, but it shows me Segmentation fault error.

I couldn't solve it this time, but I really want to study what the problem was later on. I think this technique will be a really powerful tool. Line tracing technique seems very useful for me to debug source codes.

5. Conclusion

In this paper, I propose Dead Lock Detector (DLD) which detect cycle deadlock in programs. To detect deadlock, I needed to interpose runtime mutex lock library. It was very good experience because this technique seems to be useful in many debugging situations. Not only detecting deadlock, but also detecting other in program errors, it could be useful. And I also use FIFO API which is very simple but strong to communicate between processes. To detect cycle, I had to build code ddchck in consideration of four situations. It was not that easy, but is was very good experience. I've never experienced such a complicated situation before. In the future, I will probably encounter more difficult and complicate problems, this experience gives me great help, both technically and empirically.