

# OS Code HW 03

151220030 → 高子腾 : GZT@outlook.com

2017 年 5 月 19 日

## 第一部分 问题描述

实现一个并发多线程程序，并使其在运行过程中发生死锁，实现相应的检测算法报告死锁状态。

## 第二部分 解决思路

### 1 实现死锁

这里考虑互斥锁（pthread\_mutex）的情形，由于每个互斥锁的资源都只有一个，那么产生环是造成死锁的充分必要条件，所以构造环即可。考虑 A,B 两个线程，0,1 两个互斥锁，那么保证当 A 线程占用完 0 锁，B 线程占用完 1 锁时，A 线程请求 1 锁，B 线程请求 0 锁，就可形成一环，造成死锁。

### 2 检测死锁

利用宏替换，可将 pthread\_mutex\_lock 和 pthread\_mutex\_unlock 替换为我们的自定义函数，再在里面进行真正的锁和释放操作，并且每次锁时检测是否有环，如果有环输出环的每个节点，并报告死锁且退出程序。这种方法不依赖于时钟信号和第三线程，很具有实践性。

### 3 代码

单纯死锁的代码如下，下称死锁代码，

---

```
1 #include <string.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 #include <pthread.h>
6 #include <time.h>
7 #include <sys/types.h>
8
9 #define nr_lock 100
10 pthread_mutex_t resource[nr_lock];
```

```

11 pthread_t task0, task1;
12
13
14
15 void* fun0(void* arg){
16     pthread_mutex_lock(&resource[0]);
17     sleep(1);
18     pthread_mutex_lock(&resource[1]);
19     pthread_mutex_unlock(&resource[0]);
20     pthread_mutex_unlock(&resource[1]);
21     pthread_exit(NULL);
22 }
23 void* fun1(void* arg){
24     pthread_mutex_lock(&resource[1]);
25     sleep(1);
26     pthread_mutex_lock(&resource[0]);
27     pthread_mutex_unlock(&resource[1]);
28     pthread_mutex_unlock(&resource[0]);
29     pthread_exit(NULL);
30 }
31
32 int main(){
33     int i;
34     for (i = 0; i < nr_lock; ++i)
35     {
36         pthread_mutex_init(&resource[i], NULL);
37     }
38     pthread_create(&task0, NULL, fun0, NULL);
39     pthread_create(&task1, NULL, fun1, NULL);
40     pthread_join(task0, NULL);
41     pthread_join(task1, NULL);
42 }

```

---

可以在上述代码 13 行处加上如下代码，实现 pthread\_mutex\_lock 和 pthread\_mutex\_unlock 替换，并且检测死锁。下称替换代码，

---

```

1 int selfid[1000];
2 int lselfid = 0;
3 int self2id(int self){
4     int i = 0;
5     for (i = 0; i < lselfid; ++i)
6     {
7         if(self == selfid[i])return i;
8     }
9     selfid[i] = self;
10    lselfid ++;
11    return i;
12 }
13 int lock2id(int lock){
14     return self2id(lock);
15 }
16 #define getid() self2id(pthread_self())
17 #define getlockid(self) lock2id((unsigned int)self)
18
19
20 int map[1000];
21 int visited[1000];

```

```
22 void add_edge(int s, int t){
23     map[s+1] = t+1;
24 }
25 void remove_edge(int s, int t){
26     if(map[s+1] == t+1)
27         map[s+1] = 0;
28 }
29 int is_there_a_loop(int s){
30     memset(visited, 0, sizeof(visited));
31     int cur = map[s+1];
32     visited[cur] = 1;
33     while(map[cur] != 0){
34         cur = map[cur];
35         if(visited[cur]){
36             return 1;
37         }
38         visited[cur] = 1;
39     }
40     return 0;
41 }
42 void print_a_loop(int s){
43     printf("Loop: ");
44     memset(visited, 0, sizeof(visited));
45     int cur = map[s+1];
46     visited[cur] = 1;
47     printf("%d", cur-1);
48     while(map[cur] != 0){
49         cur = map[cur];
50         printf(" to %d", cur-1);
51         if(visited[cur]){
52             printf("\n");
53             return;
54         }
55         visited[cur] = 1;
56     }
57 }
58 void pthread_mutex_lock_extended(pthread_mutex_t *lock){
59     int thread_id = getpid();
60     int lock_id = getlockid(lock);
61
62     printf("Thread#%x: try to lock %x\n", thread_id, lock_id);
63     add_edge(thread_id, lock_id);
64     if(is_there_a_loop(thread_id))
65     {
66         printf("There is deadlock!\n");
67         print_a_loop(thread_id);
68         /*printf("Trying to recover\n");
69
70         int i;
71         for (i = 0; i < nr_lock; ++i)
72         {
73             pthread_mutex_unlock(&resource[i]);
74         }*/
75
76         exit(1);
77     }else{
78         pthread_mutex_lock(lock);
```

```

79     }
80     remove_edge(thread_id, lock_id);
81     add_edge(lock_id, thread_id);
82     printf("Thread#%x: %x locked\n", thread_id, lock_id);
83 }
84 void pthread_mutex_unlock_extended(pthread_mutex_t *lock){
85     int thread_id = getpid();
86     int lock_id = getlockid(lock);
87
88     printf("Thread#%x: try to unlock %x\n", thread_id, lock_id);
89
90     remove_edge(lock_id, thread_id);
91     pthread_mutex_unlock(lock);
92
93     printf("Thread#%x: %x unlocked\n", thread_id, lock_id);
94 }
95
96 #define pthread_mutex_lock(res) pthread_mutex_lock_extended(res)
97 #define pthread_mutex_unlock(res) pthread_mutex_unlock_extended(res)

```

---

两代码的汇总见 `deadlock.c` .

## 第三部分 实验设置与结果分析

### 4 实验平台

虚拟机 Debian Jessie 32bit (1 核), gcc version 4.9.2  
Makefile:

```

1 run:
2     gcc -pthread -o dl deadlock.c
3     ./dl
4 hidden:
5     gcc -pthread -o dl hiddendeadlock.c
6     ./dl
7 norm:
8     gcc -pthread -o dl normdeadlock.c
9     ./dl

```

---

### 5 实现细节

1. 为了确保死锁的发生, 即保证出现 A 线程占用 0 锁, B 线程占用 1 锁的情形, 在代码的 17 行和 24 行分别做了两次 `sleep` 操作,
2. 由于可获得的资源和线程的 `id` 本质上都是一指针, 为了检测环的方便, 所以在替换代码的 1 到 17 行实现了线程和资源一起的重新编号 `id`, 即从 0 开始,
3. 在替换代码的添加边, 删除边, 检测环中, `map` 的内容是出边连接的节点的编号加上 1, 而 0 代表着没有出边,

4. 在替换代码的 68 到 74 行有试图恢复死锁的代码，释放所有的锁，但是这一部分代码有可能造成临界区竞争，故注释。换以直接结束进程。

## 6 实验结果

当死锁代码加上替换代码运行时，输出

---

```
1 Thread#0: try to lock 1
2 Thread#0: 1 locked
3 Thread#2: try to lock 3
4 Thread#2: 3 locked
5 Thread#0: try to lock 3
6 Thread#2: try to lock 1
7 There is deadlock!
8 Loop: 1 to 0 to 3 to 2 to 1
```

---

由书上知识可知，死锁检测和汇报结果是正确的。

当死锁代码的两线程更改为

---

```
1 void* fun0(void* arg){
2     pthread_mutex_lock(&resource[0]);
3     pthread_mutex_lock(&resource[1]);
4     pthread_mutex_unlock(&resource[0]);
5     pthread_mutex_unlock(&resource[1]);
6     pthread_exit(NULL);
7 }
8 void* fun1(void* arg){
9     pthread_mutex_lock(&resource[1]);
10    pthread_mutex_lock(&resource[0]);
11    pthread_mutex_unlock(&resource[1]);
12    pthread_mutex_unlock(&resource[0]);
13    pthread_exit(NULL);
14 }
```

---

可知上述代码虽存在死锁风险，但很大程度上依赖于线程调度，通常不会发生，程序输出

---

```
1 Thread#0: try to lock 1
2 Thread#0: 1 locked
3 Thread#0: try to lock 2
4 Thread#0: 2 locked
5 Thread#0: try to unlock 1
6 Thread#0: 1 unlocked
7 Thread#0: try to unlock 2
8 Thread#0: 2 unlocked
9 Thread#3: try to lock 2
10 Thread#3: 2 locked
11 Thread#3: try to lock 1
12 Thread#3: 1 locked
13 Thread#3: try to unlock 2
14 Thread#3: 2 unlocked
15 Thread#3: try to unlock 1
16 Thread#3: 1 unlocked
```

---

可知，系统等到线程 A 完成之后才开始执行线程 B。此结果侧面说明了此算法不能检测潜在的

死锁，只有在死锁发生时才能检测。不过这种方式又能让我们知道进程与锁的时序关系，可以让我们发现潜在的死锁。

当死锁代码更改为

---

```

1 void* fun0(void* arg){
2     pthread_mutex_lock(&resource[0]);
3     pthread_mutex_lock(&resource[1]);
4     pthread_mutex_unlock(&resource[1]);
5     pthread_mutex_unlock(&resource[0]);
6     pthread_exit(NULL);
7 }
8 void* fun1(void* arg){
9     pthread_mutex_lock(&resource[0]);
10    pthread_mutex_lock(&resource[1]);
11    pthread_mutex_unlock(&resource[1]);
12    pthread_mutex_unlock(&resource[0]);
13    pthread_exit(NULL);
14 }

```

---

可知，这段代码是没有死锁风险的，程序输出

---

```

1 Thread#0: try to lock 1
2 Thread#0: 1 locked
3 Thread#0: try to lock 2
4 Thread#0: 2 locked
5 Thread#0: try to unlock 2
6 Thread#0: 2 unlocked
7 Thread#0: try to unlock 1
8 Thread#0: 1 unlocked
9 Thread#3: try to lock 1
10 Thread#3: 1 locked
11 Thread#3: try to lock 2
12 Thread#3: 2 locked
13 Thread#3: try to unlock 2
14 Thread#3: 2 unlocked
15 Thread#3: try to unlock 1
16 Thread#3: 1 unlocked

```

---

输出结果符合我们的预期。

## 第四部分 实验总结

当死锁发生时，我们可以通过多种方法检测到死锁的发生，其中包括关于时间的分析等。但关于时间的分析强烈依赖于第三方线程，且只能定时分析出死锁是否发生，并不能汇报死锁的具体情况。而在我的实验中，通过用宏替换 `pthread_mutex_lock` 和 `pthread_mutex_unlock`，可以实现对任意关于 `mutex` 的程序的死锁情况，并汇报死锁具体关联的资源 and 进程，这种方法还能知道锁与进程的具体时序关系，总的而言是非常实用的。

但这种方法也有缺陷，第一，为了简单起见，我的方法实现只关于 `mutex` 的死锁检测（环检测），而关于普通资源的死锁检测由于时间问题没有实现；第二，这种动态方法只能检测出死锁已经发生时的死锁，而不能像静态分析一样能检测出所有的死锁风险；第三，当已经检测出死锁时不能得以恢复，只能以结束进程为结果。

## 第五部分 改进

系统分给一个线程的时间片很大，在一般情况下潜在的死锁并不会发生，这就导致了用动态方法检测死锁的固有的局限性，这种方法也不例外。但是在修改的 `pthread_mutex_lock_extended` 里真正的 `pthread_mutex_lock` 后加上 `sleep(1)` 函数，相当于每个时间片相对于线程而言降的非常低之后，这种方法也能检测潜在的死锁，详见 `hiddendeadlock.c`。

这种修改后的方法更加实用，但也存在缺陷：第一，一次运行只能汇报一个死锁；第二，由于 `sleep` 函数的存在，检测死锁的效率变得低得多；第三，此方法只对双线程程序潜在的死锁有效。

不过对于第二个问题，可以用 `pthread_yield` 代替解决。

对于如下死锁代码，

---

```

1 void* fun0(void* arg){
2     pthread_mutex_lock(&resource[0]);
3     pthread_mutex_lock(&resource[1]);
4     pthread_mutex_unlock(&resource[0]);
5     pthread_mutex_unlock(&resource[1]);
6     pthread_exit(NULL);
7 }
8 void* fun1(void* arg){
9     pthread_mutex_lock(&resource[1]);
10    pthread_mutex_lock(&resource[0]);
11    pthread_mutex_unlock(&resource[1]);
12    pthread_mutex_unlock(&resource[0]);
13    pthread_exit(NULL);
14 }
```

---

### 程序输出

---

```

1 Thread#0: try to lock 1
2 Thread#2: try to lock 3
3 Thread#0: 1 locked
4 Thread#0: try to lock 3
5 Thread#2: 3 locked
6 Thread#2: try to lock 1
7 There is deadlock!
8 Loop: 1 to 0 to 3 to 2 to 1
```

---

可以推断这种方法对双线程潜在的死锁是有效的。

为了使对多线程程序的潜质死锁也适用，可以使 `sleep` 函数换成 `usleep` 一个随机的值，比如 `usleep(rand()%1000)`，但这种方法不能保证每次都汇报潜在的死锁，但还是极有实用价值的，对于

---

```

1 void* fun0(void* arg){
2     pthread_mutex_lock(&resource[0]);
3     pthread_mutex_lock(&resource[1]);
4     pthread_mutex_unlock(&resource[0]);
5     pthread_mutex_unlock(&resource[1]);
6     pthread_exit(NULL);
7 }
8 void* fun1(void* arg){
9     pthread_mutex_lock(&resource[2]);
10    pthread_mutex_lock(&resource[0]);
```

---

```

11 pthread_mutex_unlock(&resource[2]);
12 pthread_mutex_unlock(&resource[0]);
13 pthread_exit(NULL);
14 }
15
16 void* fun2(void* arg){
17 pthread_mutex_lock(&resource[1]);
18 pthread_mutex_lock(&resource[0]);
19 pthread_mutex_unlock(&resource[1]);
20 pthread_mutex_unlock(&resource[0]);
21 pthread_exit(NULL);
22 }

```

---

运行程序五次中，有两次输出

```

1 Thread#0: try to lock 1
2 Thread#2: try to lock 3
3 Thread#4: try to lock 5
4 Thread#0: 1 locked
5 Thread#0: try to lock 5
6 Thread#2: 3 locked
7 Thread#2: try to lock 5
8 Thread#4: 5 locked
9 Thread#4: try to lock 1
10 There is deadlock!
11 Loop: 1 to 0 to 5 to 4 to 1

```

---

有一次输出

```

1 Thread#0: try to lock 1
2 Thread#2: try to lock 3
3 Thread#4: try to lock 5
4 Thread#0: 1 locked
5 Thread#0: try to lock 3
6 Thread#2: 3 locked
7 Thread#2: try to lock 1
8 There is deadlock!
9 Loop: 1 to 0 to 3 to 2 to 1

```

---

有两次顺利结束。可以看出，对于随机化的检测程序而言，虽不能保证每次都检测出死锁，但还是以较高概率检测出死锁，并且可以检测出多种死锁方式。