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方线程，很具有实践性.

3 代码

单纯死锁的代码如下，下称死锁代码，

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/types.h>

#define nr_lock 100
pthread_mutex_t resource[nr_lock];
```

```
pthread_t task0, task1;



void* fun0(void* arg){
  pthread_mutex_lock(&resource[0]);
  sleep(1);
  pthread_mutex_lock(&resource[1]);
  pthread_mutex_unlock(&resource[0]);
  pthread_mutex_unlock(&resource[1]);
  pthread_exit(NULL);
}
void* fun1(void* arg){
  pthread_mutex_lock(&resource[1]);
  sleep(1);
  pthread_mutex_lock(&resource[0]);
  pthread_mutex_unlock(&resource[1]);
  pthread_mutex_unlock(&resource[0]);
  pthread_exit(NULL);
}

int main(){
  int i;
  for (i = 0; i < nr_lock; ++i)
  {
    pthread_mutex_init(&resource[i], NULL);
  }
  pthread_create(&task0, NULL, fun0, NULL);
  pthread_create(&task1, NULL, fun1, NULL);
  pthread_join(task0, NULL);
  pthread_join(task1, NULL);
}
```

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，

```
int selfid[1000];
int lselfid = 0;
int self2id(int self){
  int i = 0;
  for (i = 0; i < lselfid; ++i)
  {
    if(self == selfid[i])return i;
  }
  selfid[i] = self;
  lselfid ++;
  return i;
}
int lock2id(int lock){
  return self2id(lock);
}
#define getid() self2id(pthread_self())
#define getlockid(self) lock2id((unsigned int)self)


int map[1000];
int visited[1000];
```

```
void add_edge(int s, int t){
  map[s+1] = t+1;
}
void remove_edge(int s, int t){
  if(map[s+1] == t+1)
    map[s+1] = 0;
}
int is_there_a_loop(int s){
  memset(visited, 0, sizeof(visited));
  int cur = map[s+1];
  visited[cur] = 1;
  while(map[cur] != 0){
    cur = map[cur];
    if(visited[cur]){
      return 1;
    }
    visited[cur] = 1;
  }
  return 0;
}
void print_a_loop(int s){
  printf("Loop: ");
  memset(visited, 0, sizeof(visited));
  int cur = map[s+1];
  visited[cur] = 1;
  printf("%d", cur-1);
  while(map[cur] != 0){
    cur = map[cur];
    printf(" to %d", cur-1);
    if(visited[cur]){
      printf("\n");
      return;
    }
    visited[cur] = 1;
  }
}
void pthread_mutex_lock_extended(pthread_mutex_t *lock){
  int thread_id = getid();
  int lock_id = getlockid(lock);

  printf("Thread#%x: try to lock %x\n", thread_id, lock_id);
  add_edge(thread_id, lock_id);
  if(is_there_a_loop(thread_id))
  {
    printf("There is deadlock!\n");
    print_a_loop(thread_id);
    /*printf("Trying to recover\n");

    int i;
    for (i = 0; i < nr_lock; ++i)
    {
      pthread_mutex_unlock(&resource[i]);
    }*/

    exit(1);
  }else{
    pthread_mutex_lock(lock);
```

```
  }
  remove_edge(thread_id, lock_id);
  add_edge(lock_id, thread_id);
  printf("Thread#%x: %x locked\n", thread_id, lock_id);
}
void pthread_mutex_unlock_extended(pthread_mutex_t *lock){
  int thread_id = getid();
  int lock_id = getlockid(lock);

  printf("Thread#%x: try to unlock %x\n", thread_id, lock_id);

  remove_edge(lock_id, thread_id);
  pthread_mutex_unlock(lock);

  printf("Thread#%x: %x unlocked\n", thread_id, lock_id);
}

#define pthread_mutex_lock(res) pthread_mutex_lock_extended(res)
#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:

```
run:
   gcc -pthread -o dl deadlock.c
   ./dl
hidden:
   gcc -pthread -o dl hiddendeadlock.c
   ./dl
norm:
   gcc -pthread -o dl normdeadlock.c
   ./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

```
Thread#0: try to lock 1
Thread#0: 1 locked
Thread#2: try to lock 3
Thread#2: 3 locked
Thread#0: try to lock 3
Thread#2: try to lock 1
There is deadlock!
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

```
void* fun0(void* arg){
  pthread_mutex_lock(&resource[0]);
  pthread_mutex_lock(&resource[1]);
  pthread_mutex_unlock(&resource[0]);
  pthread_mutex_unlock(&resource[1]);
  pthread_exit(NULL);
}
void* fun1(void* arg){
  pthread_mutex_lock(&resource[1]);
  pthread_mutex_lock(&resource[0]);
  pthread_mutex_unlock(&resource[1]);
  pthread_mutex_unlock(&resource[0]);
  pthread_exit(NULL);
}
```

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

```
Thread#0: try to lock 1
Thread#0: 1 locked
Thread#0: try to lock 2
Thread#0: 2 locked
Thread#0: try to unlock 1
Thread#0: 1 unlocked
Thread#0: try to unlock 2
Thread#0: 2 unlocked
Thread#3: try to lock 2
Thread#3: 2 locked
Thread#3: try to lock 1
Thread#3: 1 locked
Thread#3: try to unlock 2
Thread#3: 2 unlocked
Thread#3: try to unlock 1
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

```
void* fun0(void* arg){
  pthread_mutex_lock(&resource[0]);
  pthread_mutex_lock(&resource[1]);
  pthread_mutex_unlock(&resource[1]);
  pthread_mutex_unlock(&resource[0]);
  pthread_exit(NULL);
}
void* fun1(void* arg){
  pthread_mutex_lock(&resource[0]);
  pthread_mutex_lock(&resource[1]);
  pthread_mutex_unlock(&resource[1]);
  pthread_mutex_unlock(&resource[0]);
  pthread_exit(NULL);
}
```

可知，这段代码是没有死锁风险的，程序输出

```
Thread#0: try to lock 1
Thread#0: 1 locked
Thread#0: try to lock 2
Thread#0: 2 locked
Thread#0: try to unlock 2
Thread#0: 2 unlocked
Thread#0: try to unlock 1
Thread#0: 1 unlocked
Thread#3: try to lock 1
Thread#3: 1 locked
Thread#3: try to lock 2
Thread#3: 2 locked
Thread#3: try to unlock 2
Thread#3: 2 unlocked
Thread#3: try to unlock 1
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和进程，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，

```
void* fun0(void* arg){
  pthread_mutex_lock(&resource[0]);
  pthread_mutex_lock(&resource[1]);
  pthread_mutex_unlock(&resource[0]);
  pthread_mutex_unlock(&resource[1]);
  pthread_exit(NULL);
}
void* fun1(void* arg){
  pthread_mutex_lock(&resource[1]);
  pthread_mutex_lock(&resource[0]);
  pthread_mutex_unlock(&resource[1]);
  pthread_mutex_unlock(&resource[0]);
  pthread_exit(NULL);
}
```

程序输出

```
Thread#0: try to lock 1
Thread#2: try to lock 3
Thread#0: 1 locked
Thread#0: try to lock 3
Thread#2: 3 locked
Thread#2: try to lock 1
There is deadlock!
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

```
void* fun0(void* arg){
  pthread_mutex_lock(&resource[0]);
  pthread_mutex_lock(&resource[1]);
  pthread_mutex_unlock(&resource[0]);
  pthread_mutex_unlock(&resource[1]);
  pthread_exit(NULL);
}
void* fun1(void* arg){
  pthread_mutex_lock(&resource[2]);
  pthread_mutex_lock(&resource[0]);
```

```
  pthread_mutex_unlock(&resource[2]);
  pthread_mutex_unlock(&resource[0]);
  pthread_exit(NULL);
}

void* fun2(void* arg){
  pthread_mutex_lock(&resource[1]);
  pthread_mutex_lock(&resource[0]);
  pthread_mutex_unlock(&resource[1]);
  pthread_mutex_unlock(&resource[0]);
  pthread_exit(NULL);
}
```

运行程序五次中，有两次输出

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

有一次输出

```
Thread#0: try to lock 1
Thread#2: try to lock 3
Thread#4: try to lock 5
Thread#0: 1 locked
Thread#0: try to lock 3
Thread#2: 3 locked
Thread#2: try to lock 1
There is deadlock!
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．