Report for Deadlock Detect 2015

李文昊，张家睿，高宇
F1324004

2015.12.23

摘要

这篇报告整理了我们组三人实现的关于死锁检测的各个工作，其中包括对死锁概念的理解，死锁算法的设计，在用户态对动态库的劫持与最后的实现。

目录

1 What is Dead Lock **2**

2 Algorithm Design **2**

2.1 Modeling 2

2.2 Algorithm 3

3 Hook **3**

4 Implement **4**

4.1 labling 4

4.2 Main 4

4.3 DeadLock 5

5 Example **5**

5.1 ReLock 5

5.2 MultiLock 6

5.3 ProdAndCons 7

5.4 WaitAndSignal 7

6 Thought and Conclusion **8**

7 Task Assignment **9**

7.1 高宇 9

7.2 张家睿 9

7.3 李文昊 9

1 What is Dead Lock

在计算机系统中有很多独占性资源，在任一时刻它们只能被一个进程使用。最为一般人所熟知的有打印机，同时让两个进程使用将会使得打印出来的东西显得混乱不堪。在进程间通信时，对指定地区内存的修改『权限』也是一种独占性资源，如果允许多个进程进行同一个数据的修改，将会使得这个数据最后被修改成的值变得无法预知。若我们仅仅只是用单进程系统来执行一系列操作，这一切看上去并不会有什么问题，因为工作的线性会使得独占资源每时每刻只会被一个进程所使用。

而独占性资源的问题在如今的多道程序设计 (Multiprogramming) 下却会带来极大的麻烦。比如现在我们有进程 A，B，它们都需要使用扫描仪与打印机。如果 A 先申请到了扫描仪，而去请求打印机，而 B 先申请到了打印机而去请求扫描仪。由于扫描仪和打印机都是独占性资源，A，B 各自对打印机和扫描仪的请求都不能得到成功，而在请求没有成功之前它们又都不会放开已经申请到的资源。这样一来，A，B 两个进程就都被『卡死』在这里，一直在等待而毫无进展。这种状况我们称之为死锁 (Dead Lock)。

死锁一般都是和不可抢占资源 (Nonpreemptable Resource) 有关，一般来说，不可抢占资源不仅是独占性资源，而且这种资源不能在使用完之前释放，比如打印机就不能在打印某一份资料的时候突然交给另外一个进程使用，这样会使得当前打印的资料面目全非。为了更好地刻画对这一类资源的使用情况，我们把一个进程使用一个资源所需要的事件顺序刻画如下：

1. 请求资源
2. 使用资源
3. 释放资源

如果请求资源时资源不可用，那么请求的进程就会进入被迫等待的过程中。在这个前提下，死锁的定义如下：如果一个进程集合中的每个进程都在等待只能由该进程集合中的其他进程才能引发的事件，那么，该进程集合就是死锁的。

为了能够在用户态级别处理不可抢占资源，很多库中都提供了不同的方式来做到这一点，其中一个就是互斥信号量。我们能够通过互斥信号量给我们需要独占的资源上锁与解锁。

2 Algorithm Design

为了更好地进行以下算法的描述，我们可以简单的把一个互斥信号量看成一个资源，进程对资源的需求不再是直观的打印机或者内存中的某一个特定区域，而仅仅是互斥量的需求。我们在下面的叙述中的资源统一表述的是互斥量。

2.1 Modeling

根据死锁的定义，如果死锁，必然是存在这样一个链，进程 A 需要资源 b，而 b 被进程 C 所持有，同时进程 C 正在请求资源 d，d 被 E 持有，...，G 正在请求 h，而 h 被 A 所持有。正是有这样一个需求与持有构成的环，所以整个进程集合才会这样无限的等待下去。

如果考虑到进程和资源之间所有的所属关系，可能会使这个问题变得相对复杂，但是观察到死锁发生的环境，我们发现，我们只关心每一个进程当前所请求的资源，与每个资源现在的从属。

假如我们简单地把资源与进程抽象成一个个的点，并且对于每个 a 进程请求资源 b，或者资源 a 被进程 b 持有这样的关系，从 a 到 b 连一条有向边，那么死锁存在与否只和当前的图中是否存在一个环有关。

2.2 Algorithm

观察到，每一个点的出度最多只为 1，我们只需要简单的记忆每一个点出去的边。

环的出现必定是伴随着某个加边操作，并且这条边一定是出现在某一个环上，所以我们在每一次加边操作时进行环的判定，从这条边上的某个端点开始沿着自己的出边走，如果走到了一个点没有出边，那么说明当前的加边操作并没有导致环的出现；否则，必定是走到了自己本身这个点，那么这时候就需要报出死锁的信息了。以下是这个算法的简单代码。

```
int detect_cycle(int x) {
    int y = go[x], length = 0;
    while (y != -1 && y != x) {
        y = go[y];
        length++;
    }
    if (y == x) {
        return 1;
    } else {
        return 0;
    }
}
```

3 Hook

就算理解了死锁的基本概念和死锁检测的算法，但如果没法有效地得到资源与进程之间的持有与请求关系，那么一切都是空谈。好在 C 程序中对 lock、unlock 函数的调用都是通过动态库链接的方式实现的，这让我们有机会通过动态库劫持的方式进行信息的提取。

在 linux 操作系统的动态链接库中，存在一个特殊的环境变量 *LD_PRELOAD*，它可以影响程序的运行时链接，让你优先加载你自己所写的动态链接库。loader 在进行动态链接时会将有相同符号名字的函数或者变量覆盖成 *LD_PRELOAD* 中指定的 so 文件中的符号，这样一来我们就可以用我们自己 so 库中的函数替换原来库里面有的函数，从而达到 hook 的目的。

根据我们的需求，我们需要去 Hook 上锁和解锁的函数，由这些函数的使用情况我们可以获得当前资源和进程之间的从属与需求关系。为了不干扰原程序的正常运行，我们希望原函数也能在我们记录完需要的信息之后继续执行。好在我们能在原有的库中找到原函数，并且用函数指针记录下来，方便之后的重调。

4 Implement

有了以上的前备知识，我们就可以开始有关代码的实现了。

4.1 labling

在建图时，对锁与进程的标号是一个很大的问题。因为我们现在仅仅 Hook 了 lock 和 unlock 函数，我们能做的一切操作仅仅是在这两个函数中，那么有两个很大的问题是：第一点，我们该如何来表示一个资源，使得其可以和其他的资源或者进程进行区分。第二点，我们该如何知道当前所在的进程是什么，这个信息并没有体现在任何函数的输入参数中。

为了解决以上问题，我们三人查看了大量 *pthread* 的文档，终于明白了资源的类 (或者说是锁的类) 原本是用一个 *unsigned int* 来表示，所以我们也可以继续使用这个数字来给它唯一的标号。而对于进程，可以使用 *pthread_self()* 函数来直接查看它本身的类的 ID。

由于这些 lable 都是很大的数字，一个传统的做法是使用红黑树，每加入一条边，对于这条边的两个端点，我们和之前加入的点进行比较，如果之前加过，我们就直接得到了这个点的标号，否则我们赋予其一个没有使用过的最小的正整数标号。这样的做法的劣势在于复杂度稍大，每次需要 log 的复杂度。

我们采用的方法是使用了开散列表，对原始标号进行挂链哈希，记录原始编号和新的编号的一一映射。

为了避免进程的 ID 和锁的 *unsigned int* 相互冲突，我们对他们进行了不同方式的重编码。

4.2 Main

编号之后，就是我们的主要算法了，我们把事件分为以下几类

- Unlock
  这是最简单的情况，我们只需要删除这种情况对应的边即可。
- Lock and the key is free 这种情况也相对简单，我们只需要把相应的资源向相应的进程连一条边。注意，如果原来此进程向此资源有过需求等待，需要把此进程向此资源的连边删除。
- Lock and the key is belong to others 这种情况相对复杂，因为不仅需要把相应进程像资源连一条边，而且可能会出现死锁，这个时候，我们需要从这个进程所代表的点出发搜寻环的存在。
- condition wait 这种情况较为简单，我们只需要根据相应的步骤先删边，再调用原 wait 函数，最后加边，这里一定要注意不能直接调用 hook 后的 lock 和 unlock 函数。

算法的正确性依赖于这样一个事实，情况二不会产生死锁。因为如果资源处于 $free$ 状态，说明该资源没有入边，那么我们加上了该资源到某进程的这条边之后，自然也不可能会产生环。

4.3 DeadLock

完成了以上算法部分，还有一个很重要的地方需要考虑，那就是我们死锁检测的代码的死锁问题。

可以发现，由于算法的先天性质，我们找环，删边，加边都是一种独占性资源，这也意味着我们的代码不能同时被多个线程运行。因此，我们需要给自己的函数上锁。

因为我们本身就是 hook 的加锁函数，而我们现在又要给自己上锁，导致这里的实现十分 tricky，一旦实现不当就会死锁。我这里列举几点主要的需要注意的地方。

- 什么时候获得原来的 lock 与 unlock 函数

  这个问题粗看上去很显然，自然是在第一次调用 Hook 后的 lock 和 unlock 函数的时候获取原 lock 与 unlock 函数，但是要意识到一点，我们在 hook 后的 lock 函数内部要使用原 unlock 函数，这个导致我们不得不在第一次调用 Hook 后的 lock 函数时就要把原 lock 与 unlock 函数都获取到。

  那我们能不能简单的去掉 unlock 函数中，对原 lock 与 unlock 函数的获取呢？这样的行为是危险的。因为我们并不能保证外部的程序不会错误的在没有给任何锁 lock 之前 unlock，换句话来说，外部程序可能会更早的调用 unlock 函数。而在这里面，我们并没有去获取原 lock 和 unlock 函数，然后贸然给我们的函数上锁自然也是会出错的。

  所以我们只能每次调用我们 Hook 后的 lock 与 unlock 函数都去原来的库里面重新获取一遍原 lock 和 unlock 函数，而不是仅仅只获取一次。

- 运行我们的死锁检测与最后的调用原 lock 和 unlock 函数的顺序

  这个也一定要注意，我们必须要在解锁完之后才能进行原 lock 与 unlock 函数的调用，中间不能出现任何形式的跳出，不然就真的产生了死锁了。

  这样的一个限制将会给死锁检测带来一定的麻烦，因为一些信息的输出需要得到原 lock 与 unlock 的返回值，但这么一来，我们没法继续分析它们的返回值了，导致一部分工作并不好继续进行。

5 Example

在测试的过程中，我们总共使用了 3 种 example。

5.1 ReLock

第一个例子是最简单的对一个锁进行多次上锁操作。比如以下实例

```
int main() {
    pthread_mutex_lock(&first);
    pthread_mutex_lock(&first);
    pthread_mutex_unlock(&first);
    return 0;
}
```

对于这个例子，很显然，会导致一个死锁，我们的程序输出结果也正如预期：

```
liwenhao@ubuntu:~/Desktop/final$ ./make example/relock.c
hook.c: In function ‘get_mutex_id’:
hook.c:71: warning: cast from pointer to integer of different size
hook.c: In function ‘detector_wait’:
hook.c:109: warning: incompatible implicit declaration of built-in function ‘exit’
Deadlock Detected!
circle length = 2
Caused by thread 1 wait pthread mutex t 0
```

我们发现了一个长度为 2 的环，这显然是进程因为进程 1 等待锁 0，而锁 0 又已经被进程 1 所持有。

5.2 MultiLock

第二个例子，我们把经典的两个锁两个进程造成死锁扩展，变成 2 个进程，多个锁所造成的死锁，以下是程序的部分：

```
const int M = 20;
pthread_mutex_t level[20 + 1];

void *func_a() {
        int i, count = CNT;
        while (count--) {
        printf("a start %d\n", count);
        for (i = 0; i < M; i++) {
            printf("a want %d\n", i);
            pthread_mutex_lock(&level[i]);
        }
        puts("a");
        for (i = M - 1; i >= 0; i--) {
            pthread_mutex_unlock(&level[i]);
        }
        printf("a over %d\n", count);
    }
}

void *func_b() {
    int i, count = CNT;
    while (count--) {
        printf("b start %d\n", count);
        for (i = M - 1; i >= 0; i--) {
            printf("b want %d\n", i);
            pthread_mutex_lock(&level[i]);
        }
        puts("b");
        for (i = 0; i < M; i++) {
            pthread_mutex_unlock(&level[i]);
        }
```

```
        printf("b over %d\n", count);
    }
}
```

对于这种死锁，很显然，是因为 a, b 进程与 c, d 资源交叉需求与持有造成的死锁，死锁的环的长度应该为 4。以下是程序输出的结果。

```
liwenhao@ubuntu:~/Desktop/final$ cat report
Deadlock Detected!
circle length = 4
Caused by thread 1 wait pthread_mutex_t 4
```

5.3 ProdAndCons

这里测试的是最经典的消费者与生产者模型，由于代码很长，又特别经典，就不再贴上来了。由于模型过于经典，写法也是没有任何死锁的可能，经过我们程序的测试，的确没有产生死锁的结果。

```
liwenhao@ubuntu:~/Desktop/final$ ./make example/prod_cons.c
hook.c: In function ‘get_mutex_id’:
hook.c:71: warning: cast from pointer to integer of different size
hook.c: In function ‘detector_wait’:
hook.c:110: warning: incompatible implicit declaration of built-in function ‘exit’
No DeadLock detected!
```

5.4 WaitAndSignal

为了测试我们实现的 signal 与 wait 造成的死锁，我们制造了这样的一个例子：

```
pthread_cond_t cond;
pthread_mutex_t s,b;
pthread_t aa,bb;

void *fa(void *argv) {
        pthread_mutex_lock(&s);
        pthread_mutex_lock(&b);
        printf("Done\n");
        pthread_cond_wait(&cond, &s);
        printf("Start?\n");
        pthread_mutex_unlock(&s);
        pthread_mutex_unlock(&b);
        pthread_exit(0);
}
void *fb(void *argv) {
        sleep(1);
        pthread_mutex_lock(&s);
        printf("Locked\n");
        pthread_cond_signal(&cond);
        printf("Sent\n");
        pthread_mutex_lock(&b);
```

```
        pthread_mutex_unlock(&s);
        pthread_mutex_unlock(&b);
        pthread_exit(0);
}
int main() {
        pthread_mutex_init(&s, 0);
        pthread_mutex_init(&b, 0);
        pthread_cond_init(&cond, 0);
        pthread_create(&aa, 0, fa, 0);
        pthread_create(&bb, 0, fb, 0);
        pthread_join(aa,0);
        pthread_join(bb,0);
        pthread_mutex_destroy(&s);
        pthread_mutex_destroy(&b);
        pthread_cond_destroy(&cond);
        return 0;
}
```

这样一个程序的死锁埋的有点深，在于在进程 B signal 了之后，进程 A 其实一直在等待自己的那把锁，所以我们需要 hook signal 函数，在 signal 的同时去进行相应的加边操作。

```
liwenhao@ubuntu:~/Desktop/final$ ./make example/cond_wait.c
hook.c: In function ‘hash_clear’:
hook.c:55: warning: assignment makes pointer from integer without a cast
hook.c:55: warning: assignment makes integer from pointer without a cast
hook.c: In function ‘get_mutex_id’:
hook.c:78: warning: cast from pointer to integer of different size
hook.c: In function ‘get_cond_id’:
hook.c:82: warning: cast from pointer to integer of different size
hook.c: In function ‘detector_wait’:
hook.c:120: warning: incompatible implicit declaration of built-in function ‘exit’
Done
Locked
Sent
Deadlock Detected!
circle length = 4
Caused by thread 4 wait pthread mutex t 2
```

可以发现，我们的程序很好地检测到了这一类的死锁。

6 Thought and Conclusion

这次的死锁检测 Project 是一次很有意思的实验，对于我们三个没有研究过 linux 源码的人来说，起步着实是一件很困难的事。在这之中，我们进行了很多无用的尝试，最后才得到我们的成果，我想在这里列举一点我们遇到的困难。

1. 如何使用 Hook。
   一开始我们是决定在内核级别使用 Hook 来得到更多的信息，而由于在内核态遇到的太多的困难导致我们不得不选择在用户态做。

2. 哪些函数和死锁有关，线程库的源码。
   为了不在我们自己的算法层面出现死锁的情况，我们不得不经过了很多的考究，包括我们 Hook 的各个函数之间的影响，这里想要做到没有错误也是需要很多的考究。

3. 我们到底还可以做到什么。
   为了能尽可能的检测到多的死锁，我们也思考了很多死锁的形式，其中一个就是 signal 信号的丢失。还有内核级别的死锁检测。甚至，关于死锁预防和死锁解除我们也了解了一些资料，但是这都是一些理论上的知识，要真的实现的确是不太可能。

我认为我们所做的尝试虽然很多都失败了，但是还是很有启发意义，每一次失败都能使我们对 linux 更深切的领悟。十分感谢这次做 Project 的机会，也谢谢大家一起的合作。

7 Task Assignment

7.1 高宇

1. 算法设计（包括算法中可能涉及到的死锁）。

2. 程序正确性的检查。

3. 撰写 report。

7.2 张家睿

1. 内核态和用户态的 Hook 方法的研究。

2. 设计 Example

7.3 李文昊

1. 资料收集，对线程库的资料研究。

2. 代码实现，debug。