Xi Liu, xl3504, hw5

1.1

The if condition might no longer be true because bob_balance is a shared variable.

thread 1: if(bob_balance > trans){
…}

thread 2: bob_balance = bob_balance – trans

If the first thread performs the conditional check but then is interrupted before the line of smutex_lock(&mtx), thread 2 could be interleaved in between, making the bob_balance < trans.

1.2

The fix is use smutex_lock(&mtx) before the condition if(bob_balance > trans).

2.1

For example, the deadlock happens when thread 1 is holding lock mtx[0] and waiting for another one (lock mtx[1]), but the thread 2 that holds lock mtx[1] is waiting for lock mtx[0] to be released.

```
thread 1: from = 0, to = 1
            transfer(0, 1, trans)
          smutex_lock(&mtx[0]);     //a.
          smutex_lock(&mtx[1]);     //b.

thread 2: from = 1, to = 0
            transfer(1, 0, trans)
          smutex_lock(&mtx[1]);     //c.
          smutex_lock(&mtx[0]);     //d.
```

a deadlock sequence is line acbd
thread 1 holds lock mtx[0],
lock mtx[0] is wanted by thread 2,
thread 2 holds lock mtx[1],
lock mtx[1] is wanted by thread 1

2.2

```
bool transfer(int from, int to, double trans) {
   if(from > to)
   {//high-to-low order
     smutex_lock(&mtx[from]);
     smutex_lock(&mtx[to]);
   }
   else
   {
     smutex_lock(&mtx[to]);
     smutex_lock(&mtx[from]);
   }

   bool result = false;
   if (balance[from] > trans) {
      balance[from] = balance[from] - trans;
      balance[to] = balance[to] + trans;
      result = true;
   }

   smutex_unlock(&mtx[to]);
   smutex_unlock(&mtx[from]);
   return result;
}
```

3.1 and 3.2

If highPriority() runs first, output can be A B C

If mediumPriority() runs first, output can be B A C

```
void mediumPriority() {
   ... // do something
   printf("B ");
}//lowPriority() can begin

void lowPriority() {
   smutex_lock(&res);
   ... // handle resource
   smutex_unlock(&res);
   ... // do something //highPriority() can begin
   printf("C ");
```

```
}
```

4.

```
void reader_release(struct sharedlock * lock)
{
	atomic_decrement(&(lock->value));
}

void writer_acquire(struct sharedlock * lock)
{/*when a lock has not called write_release(), its value is -1
  wait until the old contents of *addr become 0,
  at that case the lock is unlocked and is ready to
be acquired*/
	while(cmpxchg_val(&(lock->value), 0, -1) != 0){}
}

void writer_release(struct sharedlock * lock)
{/*the reverse operation of above, without the while part
If lock->value is -1, the lock has been acquired but not
yet to be released*/
	cmpxchg_val(&(lock->value), -1, 0);
}
```

5.1

The program creates 3 boxes, box 3 is inside box 2, and box 2 is inside box 1.
The expected output should be

```
insert box: placing id 12 inside id 37
insert box: placing id 19 inside id 12
id: 37
- id: 12
- - id: 19
```

5.2

```
insert box: placing id 12 inside id 37
insert box: placing id 19 inside id 12
id: 37
- id: -14092
```

5.3

The variable struct box inner is a local copy, the address of it probably would not point to the same object once the insert_box function returns.

5.4

```
#include <stdlib.h>
#include <stdio.h>

// A box. Each box has an ID and a pointer to the box that resides inside
// of it. If the box has nothing inside of it, inner_box should be equal
// to NULL.
struct box {
	int id;
	struct box *inner_box;
};

// Insert box: places the box "inner" inside of the box "outer".
// Since "outer" is being modified, we pass a pointer to "outer".
// Since "inner" is not being modified, we pass in "inner" directly.
void insert_box(struct box* outer, struct box * inner) {
	printf("insert box: placing id %d inside id %d\n", inner->id, outer->id);
	outer->inner_box = inner;
}

// Print box: prints a box and the box inside of it. This function
// is recursive and will end once a box is empty.
void print_box(struct box* first, int level) {
	int i;
	if (!first)
		return;

	for (i=0; i < level; ++i) {
		printf("- ");
	}
	printf("id: %d\n", first->id);
	print_box(first->inner_box, level+1);
}

int main() {
	// Create three boxes.
	struct box box1 = { .id = 37, .inner_box = NULL };
	struct box box2 = { .id = 12, .inner_box = NULL };
	struct box box3 = { .id = 19, .inner_box = NULL };

	// The box ordering should be box1 -> box2 -> box3
	insert_box(&box1, &box2);
	insert_box(&box2, &box3);
```

```
    // Print the boxes starting from the outside box.
    print_box(&box1, 0);

    return 0;
}
```