Systemverifikation - Assertions

Danny Milosavljevic, Matnr. 0826039

April 3, 2015

```c
#include <stdio.h>
#include <stdint.h>
#include <assert.h>
#include <stdbool.h>

/* Intention: calculate y + 1 */
static uint32_t addOne(uint32_t y) {
  uint32_t x = (y + 1);
  return x;
}

/* Intention: calculate x - 1 */
static bool compareOneOff(uint32_t x) {
  uint32_t y = x;
  assert(x == y);
  y--;
  assert(x > y);
  return x > y;
}

/* Calculates congruence class of y in mod x.
 * @param x Modulus. Must be != 0.
 * @param y Original value. Must be != x. */
static uint32_t wraparound(uint32_t x, uint32_t y) {
  assert(x != y && x != 0);
  y = y % x;
  assert(y > 0 && x != 0);
  return y;
}

/* exchanges input values x and y. */
static void swap(uint32_t x, uint32_t y) {
  assert(x == y);
  x = x ^ y; // x2 = x ^ y
  y = x ^ y; // y2 = x2 ^ y
  x = x ^ y; // x3 = x2 ^ y2 = x2 ^ (x2 ^ y)
  assert(x == y); // x3 == y2
  // i.e. x2 ^ (x2 ^ y) == x2 ^ y
```

```
  // i.e. (x2 ^ x2) ^ y == x2 ^ y
  // i.e. y == x2 ^ y
  // i.e. y == x ^ (y ^ y)
  // i.e. y == x
  // i.e. will not fail.
}

int main() {
  // will not fail:
  swap(0U, 0U);
  assert(addOne(0U) == 1U);
  (void) compareOneOff(1U);
  (void) wraparound(5U, 8U);
  // will fail:
  assert(addOne((1ULL << 32) - 1) == 1ULL << 32);
  // The following values were chosen s.t.:
  // - the first assertion each does not fail
  // - but the second assertion each does fail
  (void) compareOneOff(0U);
  (void) wraparound(4U, 8U);
  return 0;
}
```

```
#include <assert.h>

// Using modulo arithmetic if not specified otherwise!

static void testLoop(unsigned i, unsigned j) {
  unsigned x = i;
  unsigned y = j;
  while (x != 0) { // same as x > 0
    // assert(y - 1 == i + j - (x + 1)); // assert(y == i
        + j - x);
    x--;
    // assert(y - 1 == i + j - x);
    y++;
    // x_n = i - n, n == i - x_n
    // y_n = j + n, y_n == j + (i - x_n)
    assert(y == i + j - x);
  }
  // x == 0
  // y == i + j
  assert((i != j) || (y == 2 * i));
  // Note: means i == j => y == i + j.
  // Note: Stronger: y == i + j, which is certain.
}
int main() {
  testLoop(10U, 20U);
  testLoop(0U, 0U);
```

```
  testLoop(0U, 20U);
  testLoop(20U, 0U);
  testLoop(20U, 10U);
  testLoop(20U, ~0U);
  testLoop(0U, ~0U);
  return 0;
}
```

```
#define _BSD_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <pthread.h>
#include <assert.h>
#include <errno.h>
#include <unistd.h>

static pthread_t tid[2];
static volatile bool thread1_wants_B = false;
static volatile bool thread2_wants_A = false;
static pthread_mutex_t A = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t B = PTHREAD_MUTEX_INITIALIZER;

#define lock(x) pthread_mutex_lock(&x)
#define unlock(x) pthread_mutex_unlock(&x)

/* Idea: In order to detect deadlocks, both threads
    signal their demands before doing anything. */

/* thread1:
  Before acquiring any lock, we state our demands.
  After only acquiring the first lock (which is safe), we
      check the demands of thread2.
  If the demands conflict, our assertion fails.
  thread2's demands are only checked while we are
     hoarding at least one of the locks, so:
    - thread2's demands cannot decrease while being
       checked.
    - thread2's demands can increase while being checked.
      If that happens, our assertion will fail faster.
    - thread2's demands can stay the same, in which case
       they will soon increase.
      But if we check before it increases, we end up
         acquiring lock B which will:
      - block thread2 at lock(B)
      - until we rescind lock B, then !thread1_wants_B.
      - for what thread2 does, see its comment.
*/
static void* thread1(void* arg) {
```

```
        (void) arg;
        while (true) {
                thread1_wants_B = true;
                lock(A);
                assert(!thread2_wants_A);
                lock(B);
                printf("thread 1 OK\n");
                fflush(stdout);
                usleep(10000);
                thread1_wants_B = false;
                unlock(B);
                unlock(A);
//              usleep(10000);
        }
        return NULL;
}
/* thread2: dual comment of thread1. exchange "thread1"
   and "thread2" and exchange "A" and "B".
   I will not copy&paste&replace here since that would
      hinder understanding and go out of sync. */
static void* thread2(void* arg) {
        (void) arg;
        while (true) {
                thread2_wants_A = true;
                lock(B);
                assert(!thread1_wants_B);
                lock(A);
                printf("thread 2 OK\n");
                fflush(stdout);
                usleep(10000);
                thread2_wants_A = false;
                unlock(A);
                unlock(B);
                usleep(10000);
        }
        return NULL;
}
int main() {
        int status = pthread_create(&tid[0], NULL, &
           thread1, NULL);
        if (status != 0) {
                perror("pthread_create");
                exit(1);
        }
        status = pthread_create(&tid[1], NULL, &thread2,
           NULL);
        if (status != 0) {
                perror("pthread_create");
                exit(1);
        }
```

```
        pthread_join(tid[1], NULL);
        pthread_join(tid[0], NULL);
        return 0;
}
```