Source Code of Test Cases for Iterative Deepening

Abstract

In our paper we show the effectiveness of combining dynamic data-race detection [7, 9] with stateless model checking [4–6]. Our approach involves adding new state spaces to explore each time a new data-race candidate is found. This document presents the source code of the 9 test cases we used, as supplemental material to our main paper.

1. "P2" thread library tests

The following tests were used to test the user-space POSIX-like thread libraries implemented by students in CMU's undergrad OS class, 15-410. They use the thread library API defined in [3] and the kernel system-call API defined in [2].

1. `broadcast_test`: Figure 1
2. `mutex_test`: Figure 2
3. `sem_test`: Figure 3
4. `signal_test`: Figures 4 and 5
5. `rwlock_test`: Figure 6
6. `join_test`: Figure 7

2. "Pintos" kernel tests

The following tests were used to test the first two Pintos kernel projects, "threads" and "userprog", implemented by students in Berkeley's and the University of Chicago's undergrad OS classes, CS162 and CMSC 23000 respectively [8]. They use the internal kernel API provided by the basecode, available online at [1].

1. `sched_test`: Figure 8
2. `alarm_test`: Figure 9
3. `wait_test`: Figures 10 and 11

```
#include <syscall.h>
#include <stdlib.h>
#include <thread.h>
#include <mutex.h>
#include <cond.h>
#include "410_tests.h"
#include <report.h>
#include <test.h>

#define STACK_SIZE 4096
#define ERR REPORT_FAILOUT_ON_ERR

mutex_t lock;
cond_t cvar;
static int ready = 0;

void *waiter(void *dummy)
{
        mutex_lock(&lock);
        while (ready == 0) {
                cond_wait(&cvar, &lock);
        }
        mutex_unlock(&lock);
        return NULL;
}

int main(void)
{
        report_start(START_CMPLT);

        ERR(thr_init(STACK_SIZE));
        ERR(mutex_init(&lock));
        ERR(cond_init(&cvar));

        ERR(thr_create(waiter, NULL));

        mutex_lock(&lock);
        ready = 1;
        cond_broadcast(&cvar);
        mutex_unlock(&lock);

        return 0;
}
```

Figure 1. `broadcast_test`, a test of `cond_signal`/`cond_broadcast` interaction.

```
#include <syscall.h>
#include <stdlib.h>
#include <thread.h>
#include <mutex.h>
#include <cond.h>
#include <assert.h>
#include "410_tests.h"
#include <report.h>
#include <test.h>

#define STACK_SIZE 4096
#define ERR REPORT_FAILOUT_ON_ERR

mutex_t lock;
static int num_in_section = 0;

void critical_section()
{
        mutex_lock(&lock);
        num_in_section++;
        yield(-1);
        assert(num_in_section == 1);
        num_in_section--;
        mutex_unlock(&lock);
}

void *contender(void *dummy)
{
        ERR(swexn(NULL, NULL, NULL, NULL));
        critical_section();
        // Do it again, in case lock is correct but unlock corrupts something.
        critical_section();
        vanish();
        return NULL;
}

int main(void)
{
        ERR(thr_init(STACK_SIZE));
        ERR(mutex_init(&lock));
        ERR(thr_create(contender, NULL));
        ERR(swexn(NULL, NULL, NULL, NULL));
        critical_section();
        vanish(); // bypass thr_exit
        return 0;
}
```

Figure 2. `mutex_test`, a test of `mutex_lock`/`mutex_unlock` correctness. Used with the special mutex-testing mode of data-race detection, described in the main paper (Section 3.1).

```
#include <syscall.h>
#include <stdlib.h>
#include <thread.h>
#include <mutex.h>
#include <cond.h>
#include <sem.h>
#include <assert.h>
#include "410_tests.h"
#include <report.h>
#include <test.h>

#define STACK_SIZE 4096
#define ERR REPORT_FAILOUT_ON_ERR

mutex_t little_lock; /* used to avoid spurious data race reports */
sem_t lock;
int num_in_section = 0;

// note: the mutex accesses in this function are ignored as PPs
// by use of without_function; they are here to prevent
// num_in_section acesses from being identified as data races.
void __attribute__((noinline)) critical_section() {
        mutex_lock(&little_lock);
        num_in_section++;
        mutex_unlock(&little_lock);
        yield(-1);
        mutex_lock(&little_lock);
        assert(num_in_section == 1);
        num_in_section--;
        mutex_unlock(&little_lock);
}
void *consumer(void *dummy) {
        ERR(swexn(NULL, NULL, NULL, NULL));
        sem_wait(&lock);
        critical_section();
        sem_signal(&lock);
        vanish();
        return NULL;
}
void producer() {
        ERR(mutex_init(&little_lock));
        ERR(sem_init(&lock, 0));
        ERR(thr_create(consumer, NULL));
        ERR(thr_create(consumer, NULL));
        sem_signal(&lock);
}
int main(void) {
        ERR(thr_init(STACK_SIZE));
        ERR(swexn(NULL, NULL, NULL, NULL));
        producer();
        vanish(); // bypass thr_exit
        return 0;
}
```

Figure 3. `sem_test`, a test of `sem_wait`/`sem_signal` interaction.

```
#include <syscall.h>
#include <stdlib.h>
#include <thread.h>
#include <mutex.h>
#include <cond.h>
#include "410_tests.h"
#include <report.h>
#include <test.h>

#define STACK_SIZE 4096
#define ERR REPORT_FAILOUT_ON_ERR

mutex_t lock1, lock2;
cond_t cvar1, cvar2;
int slept1 = 0; /* Whether thread1 has gotten to sleep on cvar1 */
int signaled1 = 0; /* Set right before main thread signals cvar1 */
int slept2 = 0; /* Whether thread1 has gotten to sleep on cvar2 */
int signaled2 = 0; /* Set right before main thread signals cvar2 */

void *thread1(void *dummy) {
        /* go to sleep on cvar1 */
        mutex_lock(&lock1);
        slept1 = 1;
        cond_wait(&cvar1, &lock1);
        if (!signaled1) {
                report_end(END_FAIL);
        }
        mutex_unlock(&lock1);

        /* go to sleep on cvar1 */
        mutex_lock(&lock2);
        slept2 = 1;
        cond_wait(&cvar2, &lock2);
        if (!signaled2) {
                report_end(END_FAIL);
        }
        mutex_unlock(&lock2);
        return NULL;
}
```

Figure 4. `signal_test`, a test for a deep bug in `cond_signal` (Part 1; see part 2 in Figure 5).

```
int main(void) {
        ERR(thr_init(STACK_SIZE));
        ERR(mutex_init(&lock1));
        ERR(mutex_init(&lock2));
        ERR(cond_init(&cvar1));
        ERR(cond_init(&cvar2));
        ERR(thr_create(thread1, NULL));

        /* Wait for thread1 to get to sleep on cvar1. */
        mutex_lock(&lock1);
        while (!slept1) {
                mutex_unlock(&lock1);
                thr_yield(-1);
                mutex_lock(&lock1);
        }
        /* Indicate that we are about to signal */
        signaled1 = 1;
        mutex_unlock(&lock1);

        /* Signal. Note that we know for sure that thread1 is asleep on
         * cvar1 (assuming correct cond vars and mutexes...) */
        cond_signal(&cvar1);

        /* Now do it all again for the second set of things. */
        /* Wait for thread1 to get to sleep on cvar2. */
        mutex_lock(&lock2);
        while (!slept2) {
                mutex_unlock(&lock2);
                thr_yield(-1);
                mutex_lock(&lock2);
        }
        /* Indicate that we are about to signal */
        signaled2 = 1;
        mutex_unlock(&lock2);

        /* Signal. Note that we know for sure that thread1 is asleep on
         * cvar1 (assuming correct cond vars and mutexes...) */
        cond_signal(&cvar2);

        return 0;
}
```

Figure 5. `signal_test`, a test for a deep bug in `cond_signal` (Part 2; see part 1 in Figure 4).

```
#include <thread.h>
#include <mutex.h>
#include <cond.h>
#include <rwlock.h>
#include <syscall.h>
#include <stdlib.h>
#include <stdio.h>
#include "410_tests.h"
#include <test.h>
#define STACK_SIZE 4096

int read_count = 0;
mutex_t read_count_lock;
rwlock_t lock;
int pass = -1;

void g() {
        mutex_lock(&read_count_lock);
        read_count++;
        if (read_count == 2) {
                pass = 0;
        }
        mutex_unlock(&read_count_lock);
}
void *f(void *arg) {
        rwlock_lock(&lock, RWLOCK_READ);
        g();
        return((void *)0);
}
int main() {
        int tid1;
        thr_init(STACK_SIZE);

        REPORT_ON_ERR(rwlock_init(&lock));
        REPORT_ON_ERR(mutex_init(&read_count_lock));
        rwlock_lock(&lock, RWLOCK_WRITE);

        if ((tid1 = thr_create(f, NULL)) < 0) {
                REPORT_END_FAIL;
                return -1;
        }
        rwlock_downgrade(&lock);
        g();
        if (thr_join(tid1, NULL) < 0) {
                REPORT_END_FAIL;
                return -1;
        }
        if (pass != 0)
                REPORT_END_FAIL;
        return pass;
}
```

Figure 6. `rwlock_test`, a test of all `rwlock` functions.

```
#include <stdio.h>
#include <thread.h>
#include <syscall.h> /* for PAGE_SIZE */

void *waiter(void *p) {
  int status;
  thr_join((int)p, (void **)&status);
  thr_exit((void *) 0);
}

int main() {
  thr_init(16 * PAGE_SIZE);
  (void) thr_create(waiter, (void *) thr_getid());
  thr_exit((void *)'!');
}
```

Figure 7. `join_test`, a test of `thr_create`, `thr_exit`, and `thr_join`.

```
#include <stdio.h>
#include "tests/threads/tests.h"
#include "threads/init.h"
#include "threads/malloc.h"
#include "threads/synch.h"
#include "threads/thread.h"
#include "devices/timer.h"

static struct semaphore sema;

static void priority_sema_thread (void *aux UNUSED)
{
  sema_down (&sema);
}

void test_priority_sema (void)
{
  int i;

  sema_init (&sema, 0);
  thread_set_priority (PRI_DEFAULT);
  for (i = 0; i < 2; i++)
    {
      int priority = PRI_DEFAULT;
      char name[16];
      snprintf (name, sizeof name, "priority %d", priority);
      thread_create (name, priority, priority_sema_thread, NULL);
    }

  for (i = 0; i < 2; i++)
    {
      sema_up (&sema);
    }
}
```

Figure 8. `sched_test`, a test of `thread_create`, `sema_up`, and `sema_down`.

```
#include <stdio.h>
#include "tests/threads/tests.h"
#include "threads/init.h"
#include "threads/synch.h"
#include "threads/thread.h"
#include "devices/timer.h"

struct semaphore done_sema;

// semaphore PPs here are ignored using without_function
static __attribute__((noinline)) void child_done() {
        sema_up(&done_sema);
}
// semaphore PPs here are ignored using without_function
static __attribute__((noinline)) void parent_done() {
        sema_down(&done_sema);
        sema_down(&done_sema);
}

static void sleeper (void *hax) {
        timer_sleep ((int)hax);
        child_done();
        thread_yield();
}
void test_alarm_simultaneous() {
        sema_init(&done_sema, 0);
        thread_create ("Patrice Godefroid", PRI_DEFAULT, sleeper, (void *)3);
        thread_create ("Cormac Flanagan", PRI_DEFAULT, sleeper, (void *)4);
        thread_yield();
        timer_sleep (2);
        parent_done();
        thread_yield();
}
```

Figure 9. `alarm_test`, a test of `timer_sleep`.

```
#include <syscall.h>
#include "tests/lib.h"
#include "tests/main.h"

void test_main (void)
{
  msg ("wait(exec()) = %d", wait (exec ("child-simple")));
}
```

Figure 10. `wait_test`, a test of `exec` and of `wait`/`exit` interaction.

```
#include <stdio.h>
#include "tests/lib.h"

int main (void)
{
  msg ("run");
  return 81;
}
```

Figure 11. `child_simple`, a helper program used by `wait_test` (Figure 10).

References

[1] R. Chen. CS 162 skeleton code for group projects. https://github.com/berkeley-cs162/group0, 2015.

[2] D. Eckhardt. Pebbles kernel specification. http://www.cs.cmu.edu/~410-f15/p2/kspec.pdf, 2015.

[3] D. Eckhardt. Project 2: User level thread library. http://www.cs.cmu.edu/~410-f15/p2/thr_lib.pdf, 2015.

[4] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040315. URL http://doi.acm.org/10.1145/1040305.1040315.

[5] P. Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 476–479, London, UK, UK, 1997. Springer-Verlag. ISBN 3-540-63166-6. URL http://dl.acm.org/citation.cfm?id=647766.733607.

[6] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855760.

[7] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM. ISBN 1-58113-588-2. doi: 10.1145/781498.781528. URL http://doi.acm.org/10.1145/781498.781528.

[8] B. Pfaff, A. Romano, and G. Back. The Pintos instructional operating system kernel. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 453–457, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-183-5. doi: 10.1145/1508865.1509023. URL http://doi.acm.org/10.1145/1508865.1509023.

[9] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4): 391–411, Nov. 1997. ISSN 0734-2071. doi: 10.1145/265924.265927. URL http://doi.acm.org/10.1145/265924.265927.