CONCURRENCT: Testing Concurrent Programs with Programmable State-Space Exploration (A DSL for Writing Concurrent Tests)

Jacob Burnim, Tayfun Elmas*
George Necula, Koushik Sen

University of California, Berkeley

HotPar 2012
How to write an xUnit-like test for a concurrent program?

• Consider:
  – Mozilla SpiderMonkey JavaScript Engine
    • Used in Firefox browser
    • 121K lines of code
  – Want to test JS_NewContext, JS_DestroyContext
    • Contain 2K < lines of code
How to write an xUnit-like test for a *sequential* program?

- Fix inputs ➔ Deterministic test
  - If there is a bug, every run manifests it!

```c
// check if any assertion fails
test_Context() {
    ...
    JSCtx *cx = JS_NewContext(rt, 0x1000);
    if (cx) {
        ...
        JS_DestroyContext(cx);
    }
}
```
How to write an xUnit-like test for a concurrent program?

- Nondeterminism due to thread schedules
  - **Hard** to specify and control schedule!

```c
// check if any assertion fails
test_Context() {
    ...
    // create 10 threads to run testfunc
}

testfunc() {
    JSContext *cx = JS_NewContext(rt, 0x1000);
    if (cx) {
        ...
        JS_DestroyContext(cx);
    }
}
```
Approaches to testing concurrent programs

1. **Stress testing**: No control over thread schedules

   ➞ No guarantee about generated schedules

```c
// check if any assertion fails
test_Context() {
    Loop 1000 times {
        ... // create 100 threads to run testfunc
    }
}

testfunc() {
    JSContext *cx = JS_NewContext(rt, 0x1000);
    if (cx) {
        ...
        JS_DestroyContext(cx);
    }
}
```
Approaches to testing concurrent programs

1. **Stress testing:** No control over thread schedules
   - No guarantee about generated schedules

2. **Model checking:** Enumerate all possible schedules
   - Too many schedules
     - Not scalable for large programs!

**Missing:** Programmer has no direct control on thread schedule
- **Key** to effective and efficient testing
Programmers have often insights/ideas about which schedules to look at.

This bug affects the pthreads version of NSPR, which is used on most Unix platforms.

There is a race condition when we use PRInterrupt to interrupt PRWaitCondVar.

Suppose thread A is calling PRWaitCondVar and thread B is interrupting thread A. The following event sequence is problematic.

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test its interrupt flag</td>
<td>Set thread A's interrupt flag</td>
</tr>
<tr>
<td>Set thread-&gt;waiting to cvar</td>
<td>Call pthread_cond_broadcast on thread A's 'waiting' cvar</td>
</tr>
<tr>
<td>Call pthread_cond_wait</td>
<td></td>
</tr>
</tbody>
</table>

Thread A misses the broadcast and blocks in pthread_cond_wait forever.

This can be reproduced with the 'join' test program, at least on Red Hat Linux 6.2.
Programmers have often insights/ideas about which schedules to look at

I have a multi-threaded application that periodically crashes. I maintain a pool of JSContexts: as they're requested from the pool JS_SetContextThread and JS_BeginRequest are called; when they're returned JS_EndRequest and JS_ClearContextThread are called.

The crashes consistently occurs inside js_GC in the following code block:

```c
while ((acx = js_ContextIterator(rt, JS_FALSE, &iter)) != NULL) {
    if (!acx->thread || acx->thread == cx->thread)
        continue;
    memset(acx->thread->gcFreeLists, 0, sizeof acx->thread->gcFreeLists);
    GSN_CACHE_CLEAR(&acx->thread->gsnCache);
}
```

acx always appears to be valid but acx->thread == NULL when the application crashes (which may be in the memset or GSN_CACHE_CLEAR line). This shouldn't occur as these lines should be skipped if (!acx->thread).

What I suspect is happening is that one thread is calling JS_GC while a second is calling JS_EndRequest and JS_ClearContextThread (in returning a context to the pool). The call to JS_GC will block until JS_EndRequest finishes.. JS_GC then resumes.. but while JS_GC is running JS ClearContextThread also runs (no locking is done in this?), modifying the value of acx->thread as the code above runs. acx->thread becomes NULL just before it gets dereferenced and the application exits.
Programmers have often insights/ideas about which schedules to look at


At least one problem that I can see from the code is that js_GC does the check:

```c
if (rt->state != JSRTS_UP && gckind != GC_LASTCONTEXT) return;
```

outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B calls js_DestroyContext and thread C calls js_NewContext.

First thread A removes its context from the runtime list. That context is not the last one so thread does not touch rt->state and eventually calls js_GC. The latter skips the above check and tries to take the GC lock.

Before this moment the thread B takes the lock, removes its context from the runtime list, discovers that it is the last, sets rt->state to LANDING, runs the-last-context-cleanup, runs the GC and then sets rt->state to DOWN.

At this stage the thread A gets the GC lock, setup itself as the thread that runs the GC and releases the GC lock to proceed with the GC when rt->state is DOWN.

Now the thread C enters the picture. It discovers under the GC lock in js_NewContext that the newly allocated context is the first one. Since rt->state is DOWN, it releases the GC lock and starts the first context initialization procedure. That procedure includes the allocation of the initial atoms and it will happen when the thread A runs the GC. This may lead precisely to the first stack trace from the comment 4.
Inserting sleeps to enforce a schedule

Sleeps:
- **Lightweight and convenient tool** for programmer
- **BUT**: Ad hoc, not reliable for long, complex schedules.

→ **Need:** Formal and robust way to describe schedules!

With the patched NSPR library, run the 'join' test.
The events will happen at the following time instants:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0: Test its interrupt flag</td>
<td>T0: Sleep 1 second</td>
</tr>
<tr>
<td>T0: Set thread-&gt;waiting to cvar</td>
<td></td>
</tr>
<tr>
<td>T0: Sleep 2 seconds</td>
<td></td>
</tr>
<tr>
<td>T1: Set thread A's interrupt flag</td>
<td>T1: Call pthread_cond_broadcast on thread A's 'waiting' cvar</td>
</tr>
<tr>
<td>T2: Call pthread_cond_wait</td>
<td></td>
</tr>
</tbody>
</table>
Case study: A bug in SpiderMonkey (1.8rc1)

• In RADBench [Jalbert, Sen, HotPar’10]

DO NOT READ!
I have a multi-threaded application that periodically crashes, giving the following assertion error:

$ ./a.out
Assertion failure: rt->state == JSRTS_UP || rt->state == JSRTS_LAUNCHING, at jscntxt.cpp:465

I’ve attached a test program which does the following:

Steps to Reproduce:
1. Write a simple application that exhibits the issue when run directly from the command line:

```c
#include <stdlib.h>
#include <pthread.h>

static JSRuntime *rt;
#define THREADS 100

static void * testfunc(void *ignored) {
    JSContext *cx = JS_NewContext(rt, 0x1000);
    if (cx) {
        JS_BeginRequest(cx);
        JS_DestroyContext(cx);
    }
    return NULL;
}

int main(void) {
    rt = JS_NewRuntime(0x100000);
    if (rt == NULL) return 1;
    pthread_t thread[THREADS];
    for (i = 0; i < THREADS; i++) {
        pthread_create(&thread[i], NULL, testfunc, NULL);
    }
    for (i = 0; i < THREADS; i++) {
        pthread_join(thread[i], NULL);
    }
    return 0;
}
```

2. Run the application directly from the command line:

```bash
$ ./a.out
```

```
Assertion failure: rt->state == JSRTS_UP || rt->state == JSRTS_LAUNCHING, at jscntxt.cpp:465
```

It seems to be very sensitive to timings as I have trouble reproducing the issue in gdb. For me to trigger it there I just need create/destroy more contexts per thread, but YMMV.

Jason Orendorff

paul.barnetta@smx.co.nz

2009-03-11 14:14 PDT

Build Identifier: Current tip
Gecko/2009010509 Gentoo Firefox/3.0.5
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.5)
```
Possible buggy schedule from bug report

DO NOT READ!

Igor Bukanov 2009-03-09 17:47:12 PDT

At least one problem that I can see from the code is that js_GC does the check:

```c
if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT)
    return;
```

outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B calls js_DestroyContext and thread C calls js_NewContext.

First thread A removes its context from the runtime list. That context is not the last one so thread does not touch rt->state and eventually calls js_GC. The latter skips the above check and tries to to take the GC lock.

Before this moment the thread B takes the lock, removes its context from the runtime list, discovers that it is the last, sets rt->state to LANDING, runs the-last-context-cleanup, runs the GC and then sets rt->state to DOWN.

At this stage the thread A gets the GC lock, setup itself as the thread that runs the GC and releases the GC lock to proceed with the GC when rt->state is DOWN.

Now the thread C enters the picture. It discovers under the GC lock in js_NewContext that the newly allocated context is the first one. Since rt->state is DOWN, it releases the GC lock and starts the first context initialization procedure. That procedure includes the allocation of the initial atoms and it will happen when the thread A runs the GC. This may lead precisely to the first stack trace from the comment 4.

Fixed, known schedule for threads A and B

Unknown schedule for A and C
Concurrit: A DSL for writing concurrent tests

Software Under Test

```c
#define THREADS 100

static void * testfunc(void *ignored) {
    JSContext *cx = JS_NewContext(rt, 0x1000);
    if (cx) {
        JS_BeginRequest(cx);
        JS_DestroyContext(cx);
    }
    return NULL;
}
```

Insights/ideas about thread schedules

Systematically explore all-and-only thread schedules specified by DSL

Specify a set of schedules in **formal**, **concise**, and **convenient** way
Unit-testing programs with Concurrit
(What about integration tests?: Wait for conclusion)

Software Under Test (SUT)
Instrumented to control

Test in Concurrit DSL
Runs concurrently with SUT

Thread A

Thread B

Thread C
testfunc() {
  JSContext *cx = JS_NewContext(r
  if (cx) {
    JS_BeginRequest(cx);
    JS_DestroyContext(cx);
  }
}

Send event and block

Unblock thread

Kinds of events: Memory read/write, function enter/return, function call, end of thread, at particular source line, user-defined
Unit-testing programs with Concurrit
(What about integration tests?: Wait for conclusion)

Software Under Test (SUT)
Instrumented to control

Concurrit monitor
Runs concurrently with SUT

Thread A

Thread B

Thread C
testfunc() {
    JSContext *cx = JS_NewContext(rt, 0x1000);
    if (cx) {
        JS_BeginRequest(cx);
        JS_DestroyContext(cx);
    }
}

Send event and block

Unblock thread

Kinds of events: Memory read/write, function enter/return, function call, end of thread, at particular source line, user-defined
Outline

• Bug report for Mozilla SpiderMonkey
• Write tests in Concurrit DSL to generate buggy schedule

Simple schedules:
  • Few schedules BUT not manifesting bug

  All schedules:
  • Manifests bug BUT too many schedules

  Target buggy schedule in bug report
  • Few schedules AND manifests bug
At least one problem that I can see from the code is that js_GC does the check:

```c
if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT)
    return;
```

outside the GC lock. **Now suppose there are 3 threads, A, B, C.** Threads A and B calls js_DestroyContext and thread C calls js_NewContext.

First thread A removes its context from the runtime list. That context is not the last one so thread does not touch rt->state and eventually calls js_GC. The latter skips the above check and tries to to take the GC lock.

Before this moment the thread B takes the lock, removes its context from the runtime list, discovers that it is the last, sets rt->state to LANDING, runs the-last-context-cleanup, runs the GC and then sets rt->state to DOWN.

At this stage the thread A gets the GC lock, setup itself as the thread that runs the GC and releases the GC lock to proceed with the GC when rt->state is DOWN.

Now the thread C enters the picture. It discovers under the GC lock in js_NewContext that the newly allocated context is the first one. Since rt->state is DOWN, it releases the GC lock and starts the first context initialization procedure. That procedure includes the allocation of the initial atoms and it will happen when the thread A runs the GC. This may lead precisely to the first stack trace from the comment 4.
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: WITH T IN [TA, TB, TC]

4: RUN T UNTIL COMPLETES
First test: Run each thread sequentially until completion (No interleaving)

```plaintext
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:   WITH T IN [TA, TB, TC]

4:     RUN T UNTIL COMPLETES

}
```

Wait until 3 distinct threads sending events
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3:  WITH T IN [TA, TB, TC]

4:  RUN T UNTIL COMPLETES

}
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: WITH T IN [TA, TB, TC]

4: RUN T UNTIL COMPLETES
}

Pick one of the threads

Backtrack/choice point
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL
1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()
2: LOOP UNTIL TA, TB, TC COMPLETE {
3: WITH T IN [TA, TB, TC]
4: RUN T UNTIL COMPLETES
}
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: WITH T IN [TA, TB, TC]

4: RUN T UNTIL COMPLETES
}

Pick one of the threads

Backtrack/choice point

Thread completes

TA

TB

TC

TA

TC
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: WITH T IN [TA, TB, TC]

4: RUN T UNTIL COMPLETES

}
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {
    3: WITH T IN [TA, TB, TC]
    4: RUN T UNTIL COMPLETES
}

Pick one of the threads

Thread completes

Backtrack/choice point
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: WITH T IN [TA, TB, TC]

4: RUN T UNTIL COMPLETES

}
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: BACKTRACK HERE WITH T IN [TA, TB, TC]

4: RUN T UNTIL COMPLETES
}

Pick a different thread when backtracked
First test: Run each thread sequentially until completion (No interleaving)

```c
// Test in Concurrit DSL
1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()
2: LOOP UNTIL TA, TB, TC COMPLETE {
3: BACKTRACK HERE WITH T IN [TA, TB, TC]
4: RUN T UNTIL COMPLETES
}
```

Pick a different thread when backtracked
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL
1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()
2: LOOP UNTIL TA, TB, TC COMPLETE {
3: BACKTRACK HERE WITH T IN [TA, TB, TC]
4: RUN T UNTIL COMPLETES
}
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: BACKTRACK HERE WITH T IN [TA, TB, TC]

4: RUN T UNTIL COMPLETES
}

Pick a different thread when backtracked
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: BACKTRACK HERE WITH T IN [TA, TB, TC]

4: RUN T UNTIL COMPLETES

}
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: BACKTRACK HERE WITH T IN [TA, TB, TC]

4: RUN T UNTIL COMPLETES
}

Pick a different thread when backtracked
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: BACKTRACK HERE WITH T IN [TA, TB, TC]

4: RUN T UNTIL COMPLETES
}
First test: Run each thread sequentially until completion (No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {
   3: BACKTRACK HERE WITH T IN [TA, TB, TC]
   4: RUN T UNTIL COMPLETES
}

Result:
6 schedules
No assertion failure!
Second test: Run each thread sequentially until it returns from function

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: BACKTRACK HERE WITH T IN [TA, TB, TC]

4: RUN T UNTIL RETURNS FROM JS_NewContext, JS_BeginRequest, OR JS_DestroyContext

Result:
< 50 schedules
No assertion failure!
Outline

• Bug report for Mozilla SpiderMonkey
• Write tests in Concurrit DSL to generate buggy schedule
  – Simple schedules
    • Few schedules **BUT** not manifesting bug
  All schedules
    • Manifests bug **BUT** too many schedules
  – Target buggy schedule in bug report
    • Few schedules **AND** manifests bug
First test: Run each thread sequentially until completion
(No interleaving)

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: BACKTRACK HERE WITH T IN [TA, TB, TC]

4: RUN T UNTIL COMPLETES
}

Generate all thread schedules

// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: BACKTRACK HERE WITH T IN [TA, TB, TC]

4: RUN T UNTIL NEXT EVENT

}

Result:
> 100,000 schedules
Assertion failure after a night!
What is different from (traditional) model checking?

1. **Cannot control/instrument everything!**
   - Must tolerate uncontrolled non-determinism
   - Backtrack-and-replay-prefix may fail

2. **Localize the search**
   - To particular functions, operations, states, ...

**BUT:** Can express traditional model checking algorithms
   - If every operation can be controlled
   - Feasible for small programs, data structures, ...
Outline

- Bug report for Mozilla SpiderMonkey
- Write tests in Concurrit DSL to generate buggy schedule
  - Simple schedules
    - Few schedules **BUT** not manifesting bug
  - All schedules
    - Manifests bug **BUT** too many schedules

**Target buggy schedule in bug report**
- Few schedules **AND** manifests bug
Igor Bukanov 2009-03-09 17:47:12 PDT

At least one problem that I can see from the code is that js_GC does the check:

```c
if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT)
    return;
```

outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B calls js_DestroyContext and thread C calls js_NewContext.

First thread A runs the GC and releases the GC lock. Then the thread B runs the GC and after runs the GC still holds the lock and checks that thread C calling js_NewContext is the first thread to run. The latter skips the above check and tries to take the GC lock.

At this stage the thread A acquires the GC lock, sets itself as the thread that runs the GC and reenters the GC cleanup loop. That is the thread A runs the GC and releases the GC lock with rt->state is DOWN.

Now the thread C enters the picture. It discovers under the GC lock in js_NewContext that the newly allocated context is the first one. Since rt->state is DOWN, it releases the GC lock and starts the first context initialization procedure. That procedure includes the allocation of the initial atoms and it will happen when the thread A runs the GC. This may lead precisely to the first stack trace from the comment 4.
// Test in Concurrit DSL

1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()

2: LOOP UNTIL TA, TB, TC COMPLETE {

3: BACKTRACK HERE WITH T IN [TA, TB, TC]

4: RUN T UNTIL NEXT EVENT
}
Exploiting programmer’s insights about bug

// Test in ConcurrIt

1: TC = WAIT_FOR_THREAD(ENTERS JS_NewContext)
2: TA = WAIT_FOR_DISTINCT_THREAD(ENTERS JS_DestroyContext)
3: TB = WAIT_FOR_DISTINCT_THREAD(ENTERS JS_DestroyContext)
4: LOOP UNTIL TA, TB, TC COMPLETE {
5:    BACKTRACK HERE WITH T IN [TA, TB, TC]
6:    RUN T UNTIL NEXT EVENT
}

Result:
< 50,000 schedules
Assertion failure after a few hours!
What is different from (traditional) model checking?

1. Cannot control/instrument everything!
   • Must tolerate uncontrolled non-determinism
   • Backtrack-and-replay-prefix may fail

2. Localize the search
   • To particular functions, operations, states, ...

BUT: Can express traditional model checking algorithms
   • If every operation can be controlled
   • Feasible for small programs, data structures, ...
Possible buggy schedule from bug report

- Shared variables involved in the bug:
  - rt->state, rt->gcLock, rt->gcThread
- Reschedule threads when accessing them.

outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B calls \texttt{js\_DestroyContext} and thread C calls \texttt{js\_NewContext}.

First thread A removes its context from the runtime list. That context is not the last one so thread does not touch \texttt{rt->state} and eventually calls \texttt{js\_GC}. The latter skips the above check and tries to take the GC lock.

Before this moment the thread B takes the lock, removes its context from the runtime list, discovers that it is the last, sets \texttt{rt->state} to LANDING, runs the-last-context-cleanup, runs the GC and then sets \texttt{rt->state} to DOWN.

At this stage the thread A gets the GC lock, setup itself as the thread that runs the GC and releases the GC lock to proceed with the GC when \texttt{rt->state} is DOWN.

Now the thread C enters the picture. It discovers under the GC lock in \texttt{js\_NewContext} that the newly allocated context is the first one. Since \texttt{rt->state} is DOWN, it releases the GC lock and starts the first context initialization procedure. That procedure includes the allocation of the initial atoms and it will happen when the thread A runs the GC. This may lead precisely to the first stack trace from the \texttt{comment\_4}.
Exploiting programmer’s insights about bug

// Test in Concurrit DSL

1: TC = WAIT_FOR_THREAD(ENTERS JS_NewContext)

2: TA = WAIT_FOR_DISTINCT_THREAD(ENTERS JS_DestroyContext)

3: TB = WAIT_FOR_DISTINCT_THREAD(ENTERS JS_DestroyContext)

4: LOOP UNTIL TA, TB, TC COMPLETE {

5: BACKTRACK HERE WITH T IN [TA, TB, TC]

6: RUN T UNTIL NEXT EVENT

}
Exploiting programmer’s insights about bug

```c
// Test in Concurrit

1: TC = WAIT_FOR_THREAD(ENTERS JS_NewContext)

2: TA = WAIT_FOR_DISTINCT_THREAD(ENTERS JS_DestroyContext)

3: TB = WAIT_FOR_DISTINCT_THREAD(ENTERS JS_DestroyContext)

4: LOOP UNTIL TA, TB, TC COMPLETE {

5:    BACKTRACK HERE WITH T IN [TA, TB, TC]

6:    RUN T UNTIL Reads or Writes & rt->state, & rt->gcLock, & rt->gcThread!
}
```

Result:
~ 2000 schedules
Assertion failure after 2 hours!
Possible buggy schedule from bug report

Igor Bukanov 2009-03-09 17:47:12 PDT

At least one problem that I can see from the code is that js_GC does the check:

```c
if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT) return;
```

outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B calls js_DestroyContext and thread C calls js_NewContext.

First thread A removes its context from the runtime list. That context is not the last one so thread does not touch rt->state and eventually calls js_GC. The latter skips the above check and tries to take the GC lock.

Before this moment the thread B takes the lock, removes its context from the runtime list, discovers that it is the last, sets rt->state to LANDING, runs the last-context-cleanup, runs the GC and then sets rt->state to DOWN.

At this stage the thread A gets the GC lock, setup itself as the thread that runs the GC and releases the GC lock to proceed with the GC when rt->state is DOWN.

Now the thread C enters the picture. It discovers under the GC lock in js_NewContext that the newly allocated context is the first one. Since rt->state is DOWN, it releases the GC lock and starts the first context initialization procedure. That procedure includes the allocation of the initial atoms and it will happen when the thread A runs the GC. This may lead precisely to the first stack trace from the comment 4.

Unknown schedule for A and C

Setup

Fixed, known schedule for threads A and B

Igor Bukanov 2009-03-10 07:55:37 PDT

With the test program on 64-bit Linux I could not reproduce the bug from the comment 4 but I do see assert from the comment 0 after bumping the number of threads to 1000. The assert is indeed rare, about 2-3% of all runs and I could not reproduce it under GDB. On the other hand, good old printfs have shown what was going on. The problem comes from the following code in js_NewContext:

```c
JS_LOCK_GC(rt);
for (;;) {
  if (first && rt->contextList.next == &rt->contextList)
    break;
  if (rt->state == JSRTS_UP) {
    JS_ASSERT(!first);
    JS_WAIT_CONDVAR(rt->stateChange, JS_NO_TIMEOUT);
  }
  JS_APPEND_LINK(&cx->link, &rt->contextList);
  JS_WAIT_CONDVAR(rt->stateChange, JS_NO_TIMEOUT);
  ...
}
```
// Test in Concurrit DSL

TC = WAIT_FOR_THREAD(
    ENTERS JS_NewContext)

TA = WAIT_FOR_DISTINCT_THREAD(
    ENTERS JS_DestroyContext)

TB = WAIT_FOR_DISTINCT_THREAD(
    ENTERS JS_DestroyContext)

RUN TA UNTIL READS &rt->state IN js_GC

RUN TB UNTIL COMPLETES

RUN TA UNTIL WRITES &rt->gcThread IN js_GC

LOOP UNTIL TA, TC COMPLETE {
    BACKTRACK HERE WITH T IN [TA, TC]

    RUN T UNTIL READS OR WRITES MEMORY
}
At least one problem that I can see from the code is that js_GC does the check:
if (rt->state != JSRTS_UP && gckind != GC_LAST_CONTEXT)
return;
outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B
 calls ja_DestroyContext and thread C calls ja_NewContext.

First thread A removes its context from the runtime list. That context is not
the last one so thread does not touch rt->state and eventually calls js_GC. The
latter skips the above check and tries to to take the GC lock.

Before this moment thread B takes the lock, removes its context from the
runtime list, discovers that it is the last, sets rt->state to LANDING, runs
the-last-context-cleanup, runs the GC and then sets rt->state to DOWN.

At this stage the thread A gets the GC lock, setup itself as the thread that
runs the GC and releases the GC lock to proceed with the GC when rt->state is
DOWN.

Now the thread C enters the picture. It discovers under the GC lock in
ja_NewContext that the newly allocated context is the first one. Since
rt->state is DOWN, it releases the GC lock and starts the first context
initialization procedure. That procedure includes the allocation of the initial
atoms and it will happen when the thread A runs the GC. This may lead precisely
to the first stack trace from the comment 4.

Igor Bukanov 2009-03-10 07:55:37 PDT
With the test program on 64-bit Linux I could not reproduce the bug from the
comment 4 but I do see assert from the comment 0 after bumping the number of
threads to 1000. The assert is indeed rare, about 2-3% of all runs and I could
not reproduce it under GDB. On the other hand, good old printfs have shown what
was going on. The problem comes from the following code in js_NewContext:

```c
JS_LOCK_GC(rt);
for (;;) {
    first = (rt->contextList.next == &rt->contextList);
    if (rt->state == JSRTS_UP) {
        JS_ASSERT(!first);
        /* Ensure that it is safe to update rt->contextList below. */
        js_WaitForGC(rt);
        break;
    ...```

Run TA until reads \&rt->state in js_GC
Run TB until completes
Run TA until writes \&rt->gcThread in js_GC
Loop until TA, TC complete {
   Backtrack here with T in [TA, TC]
   Run T until reads or writes memory
}
Implementation/Evaluation

- **Implementation**: DSL embedded in C++
  - Wrote concise tests for (real/manually-inserted) bugs in well-known benchmarks
- Reproducing bugs
  - using < 20 lines of DSL code, after < 30 schedules
  - **Inspect**: bbuf, bzip2, pbzip2, pfscan
  - **PARSEC**: dedup, streamcluster
  - **RADBench**: SpiderMonkey 1/2, Mozilla NSPR 1/2/3
- **Ongoing**: Apache httpd, Chromium, Memcached
  - Can write various model checking algorithms (next slide)
Default search policies

```plaintext
EXPLORE_ALL_SCHEDULES(THREADS) {
    LOOP UNTIL ALL THREADS COMPLETE {
        BACKTRACK HERE WITH T IN THREADS
        RUN T UNTIL NEXT EVENT
    }
}

EXPLORE_TWO_CONTEXT_BOUNDED_SCHEDULES(THREADS) {
    BACKTRACK HERE WITH T1 IN THREADS
    BACKTRACK HERE LOOP NONDETERMINISTICALLY {
        RUN T1 UNTIL NEXT EVENT
    }

    BACKTRACK HERE WITH T2 IN [THREADS EXCEPT T1]
    BACKTRACK HERE LOOP NONDETERMINISTICALLY {
        RUN T2 UNTIL NEXT EVENT
    }
}

EXPLORE_THREADS_UNTIL_COMPLETION(THREADS)
}

EXPLORE_THREADS_UNTIL_COMPLETION(THREADS) {
    LOOP UNTIL ALL THREADS COMPLETE {
        BACKTRACK HERE WITH T IN THREADS
        RUN T UNTIL COMPLETION
    }
}
```
Positioning Concurrit: Usage scenarios

Insert sleeps:
Explore one schedule

Model checking:
Explore all schedules

Concurrit

Control user-defined events
- Portable, testing library
- Manual instrumentation
- Generate exact/perfect schedule

Control all operations
- Exhaustive testing tool
- Automated instrumentation
- Generate all schedules
Unit-testing programs with Concurrit

Software Under Test (SUT)

Instrumented to control

Test in Concurrit DSL
Runs concurrently with SUT

Thread A

Thread B

Thread C

testfunc() {
    JSContext *cx = JS_NewContext(r
    if (cx) {
        JS_BeginRequest(cx);
        JS_DestroyContext(cx);
    }
    }

Send event and block

Unblock thread
Ongoing work: Integration testing
Controlling multi-process/distributed applications

Concurrit monitor process

// Test in Concurrit DSL

...........

Events

Request process 1

// Threads sending
// requests to server

...........

Apache web server

// Server threads
// handling requests

...........

Events

Request process 2

// Threads sending
// requests to server

...........
Approaches to controlling thread schedules

Test run: A set of executions of the test driver.
Success: At least one execution in the run hits the bug.