

# **CONCURRIT: 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*

# 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!

```
// check if any assertion fails
test_Context() {
    ...
    JSContext *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!



```
// 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

```
// 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

Wan-Teh Chang 2002-08-29 16:08:33 PDT

[Description](#) [\[reply\]](#) [\[-\]](#) [\[reply\]](#) [\[-\]](#)

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

There is a race condition when we use PR\_Interrupt to interrupt PR\_WaitCondVar.

Suppose thread A is calling PR\_WaitCondVar and thread B is interrupting thread A. The following event sequence is problematic.

**DO NOT READ!**

```
Thread A                                Thread B
=====                                =====
Test its interrupt flag
Set thred->waiting to cvar

                                           Set thread A's interrupt
                                           flag

                                           Call pthread_cond_broadcast
                                           on thread A's 'waiting'
                                           cvar

Call pthread_cond_wait

=====
```

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

[paul.barnetta@smx.co.nz](mailto:paul.barnetta@smx.co.nz) 2009-02-04 13:54:41 PST [Description](#) [\[reply\]](#) [\[-\]](#) [\[reply\]](#) [\[-\]](#)

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.

**DO NOT READ!**

The crashes consistently occurs inside js\_GC in the following code block:

```
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

[Igor Bukanov](#) 2009-03-09 17:47:12 PDT

[Comment 5](#) [\[reply\]](#) [\[-\]](#) [\[reply\]](#) [\[-\]](#)

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;
```

**DO NOT READ!**

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`.

**Fixed, known schedule for threads A and B**

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`.

**Unknown schedule for A and C**

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:

Thread A	Thread B
-----	-----
T0: Test its interrupt flag	T0: Sleep 1 second
T0: Set thred->waiting to cvar	
T0: Sleep 2 seconds	
	T1: Set thread A's interrupt flag
	T1: Call pthread_cond_broadcast on thread A's 'waiting' cvar
T2: Call pthread_cond_wait	

# 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 demonstrates the issue. It spawns many threads, each of which create and destroy contexts, exiting. I'd expect the number of contexts active to be bounded by [0..THREADS], possibly transitioning between 0 and showing a race condition in the code?

Reproducible: Always

Steps to Reproduce:  
Below is a simple application that exhibits the issue when run directly from the command line:

```
8<----
#include <stdlib.h>
#include <pthread.h>
#include "jsapi.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(0x1000000);
    if (rt == NULL)
        return 1;
    /* Uncommenting this to guarantee there's always at least
     * one context in the runtime prevents this problem. */
    // JSContext *cx = JS_NewContext(rt, 0x1000);
    int i;
    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;
}
8<----
```

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.

8<----

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

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

```
#define THREADS 100
```

```
static void * testfunc(void *ignored) {
```

```
    JSContext *cx = JS_NewContext(rt, 0x1000);
```

```
    if (cx) {
```

```
        JS_BeginRequest(cx);
```

```
        JS_DestroyContext(cx);
```

```
    }
```

```
    return NULL;
```

```
}
```

# Possible buggy schedule from bug report

**DO NOT READ!**

[Igor Bukanov](#) 2009-03-09 17:47:12 PDT

[Comment 5](#) [\[reply\]](#) [\[-\]](#) [\[reply\]](#) [\[-\]](#)

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 `js_DestroyContext` and thread C calls `js_NewContext`.

**Fixed, known schedule for threads A and B**

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`.

**Unknown schedule for A and C**

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](#).

# Concurrit: A DSL for writing concurrent tests

[Igor Bukanov](#) 2009-03-09 17:47:12 PDT [Comment 5](#) [\[reply\]](#) [\[-\]](#) [\[reply\]](#) [\[-\]](#)

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;
```

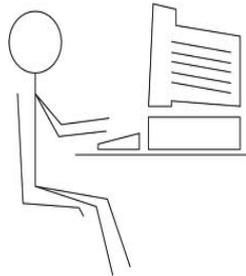
outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B calls js\_DestroyContext and thread C calls js\_NewContext.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

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](#).

**Insights/ideas about thread schedules**



```
Software Under Test

#define THREADS 100

static void * testfunc(void *ignored) {

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

    return NULL;
}
```

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

Test in Concurrit 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

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

## Test in Concurrit DSL

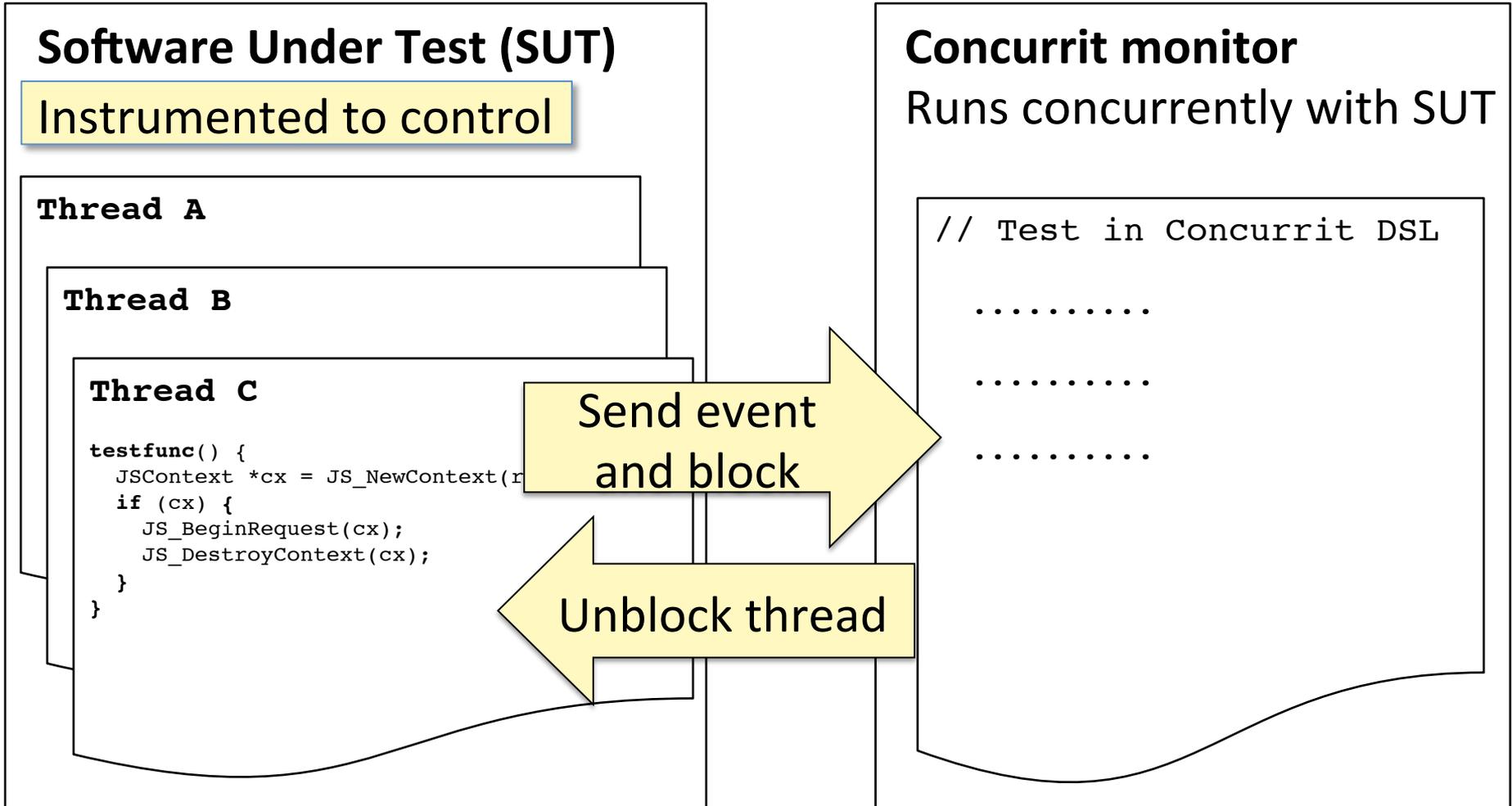
Runs concurrently with SUT

.....  
.....  
.....  
.....  
.....  
.....

**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)



**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

# Possible buggy schedule from bug report

[Igor Bukanov](#) 2009-03-09 17:47:12 PDT

[Comment 5](#) [[reply](#)] [[-](#)] [[reply](#)] [[-](#)]

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 `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)

```
// 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  
}
```

Loop until all 3 threads complete



# 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  
}
```

Run selected thread until it completes



○ Backtrack/choice point

Thread completes

TA

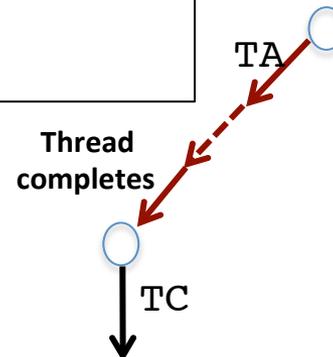
# First test: Run each thread sequentially until completion (No interleaving)

```
// 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



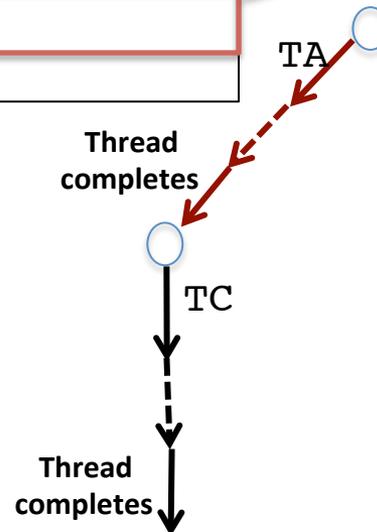
○ Backtrack/choice point



# 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  
}
```

Run selected thread until it completes



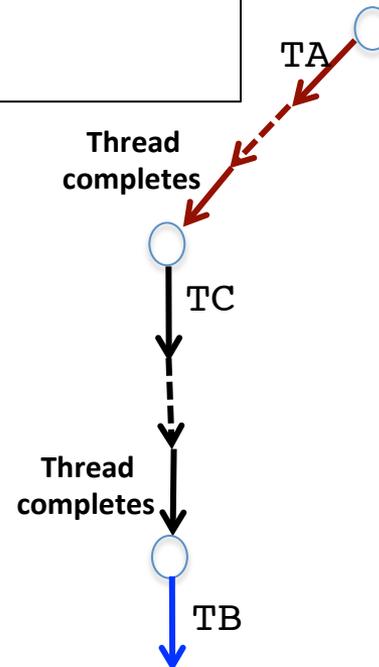
# First test: Run each thread sequentially until completion (No interleaving)

```
// 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



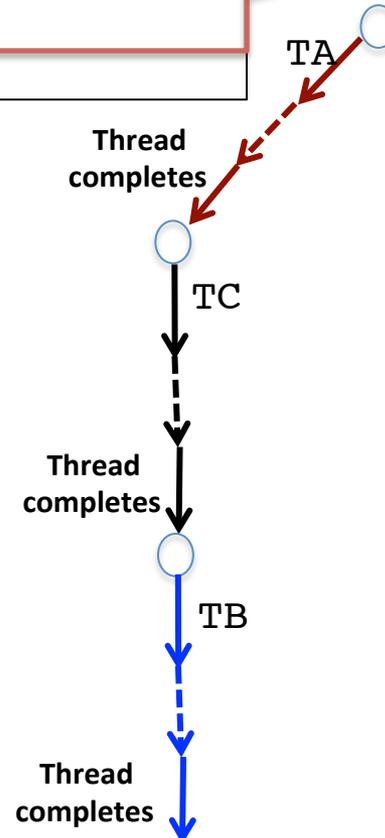
# First test: Run each thread sequentially until completion (No interleaving)

```
// 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  
}
```

Run selected thread until it completes



○ Backtrack/choice point



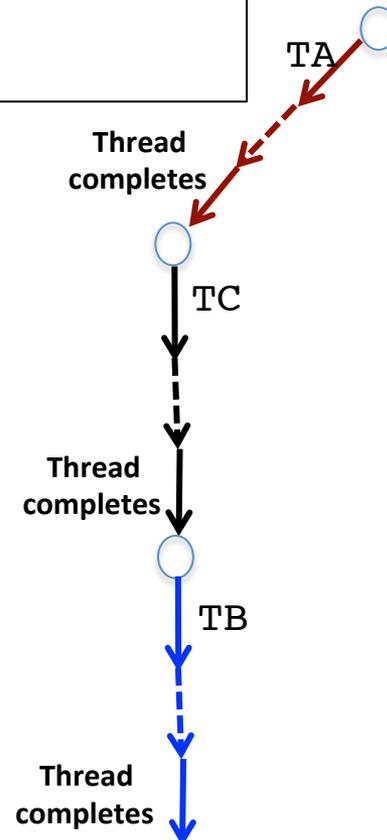
# First test: Run each thread sequentially until completion (No interleaving)

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

Pick a different thread  
when backtracked



○ Backtrack/choice point



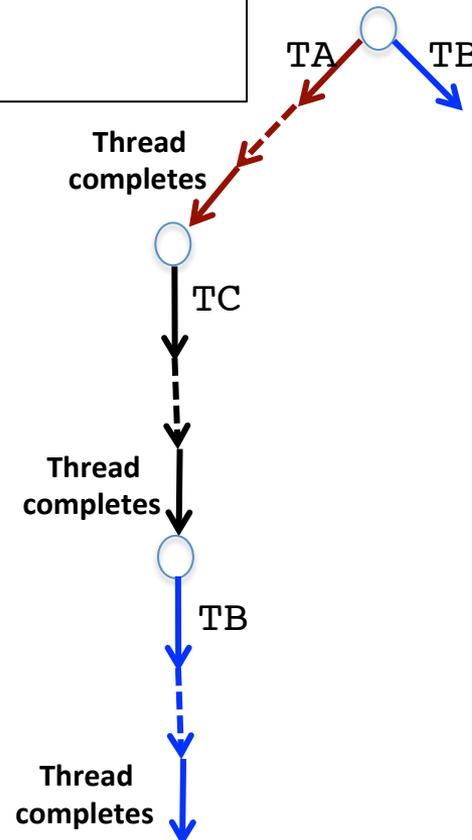
# First test: Run each thread sequentially until completion (No interleaving)

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

Pick a different thread  
when backtracked



○ Backtrack/choice point

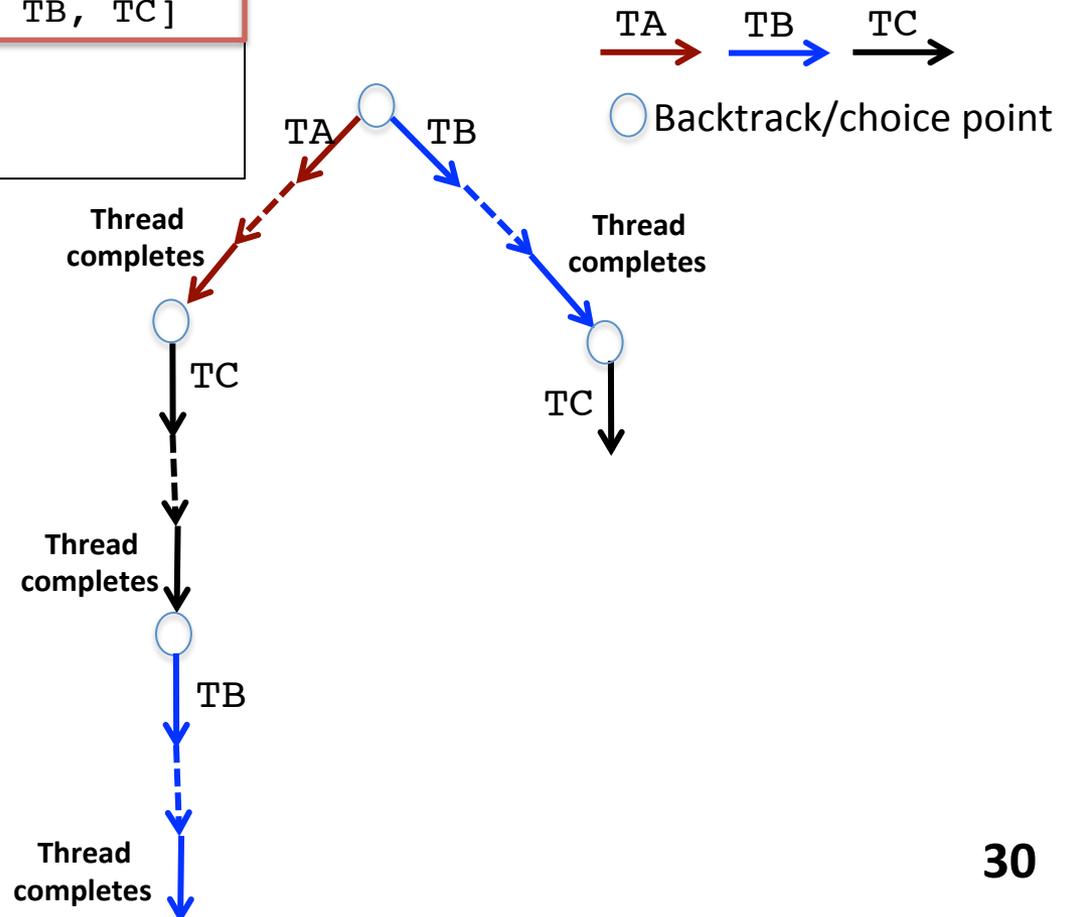




# First test: Run each thread sequentially until completion (No interleaving)

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

Pick a different thread  
when backtracked



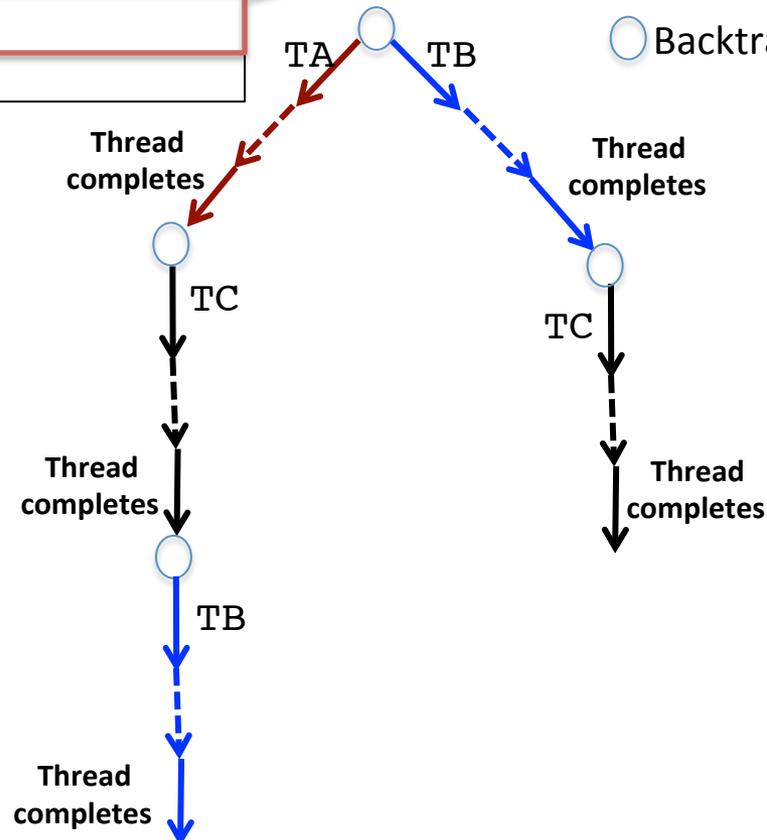
# First test: Run each thread sequentially until completion (No interleaving)

```
// 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  
}
```

Run selected thread until it completes



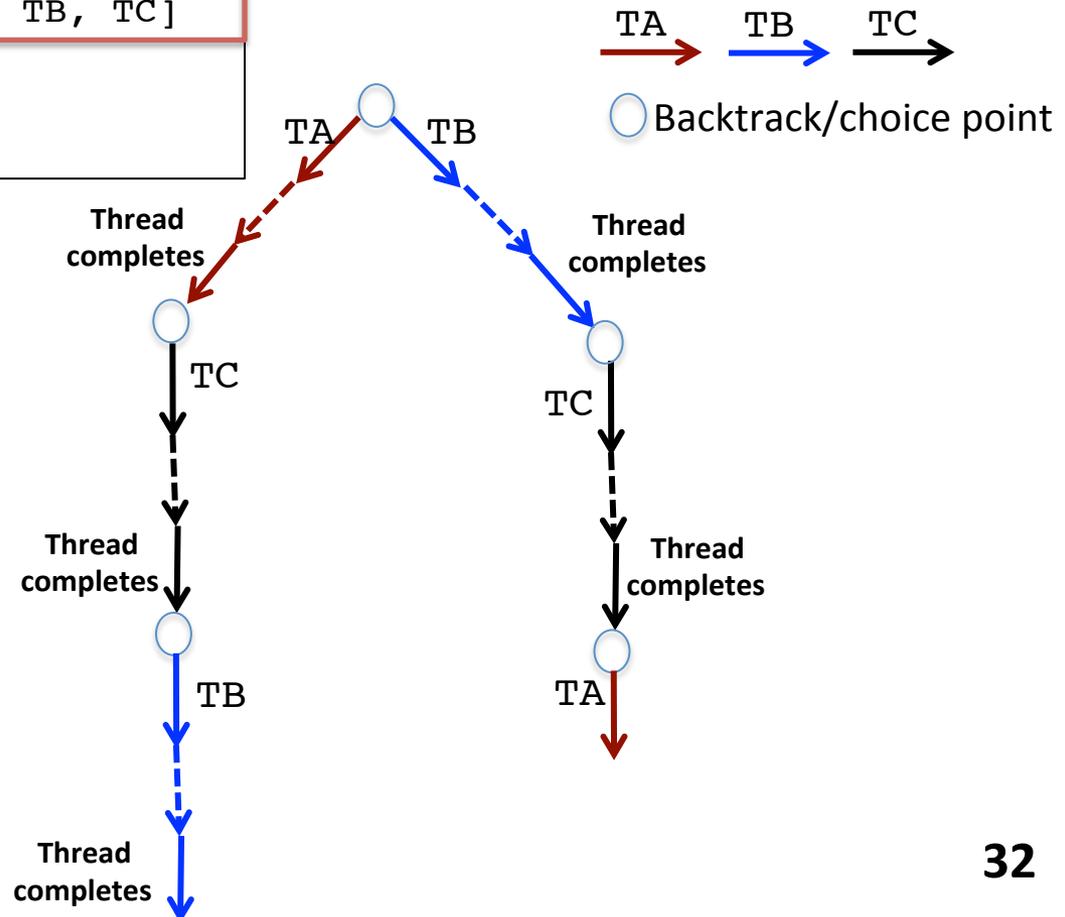
○ Backtrack/choice point



# 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:   BACKTRACK HERE WITH T IN [TA, TB, TC]
4:   RUN T UNTIL COMPLETES
}
```

Pick a different thread  
when backtracked



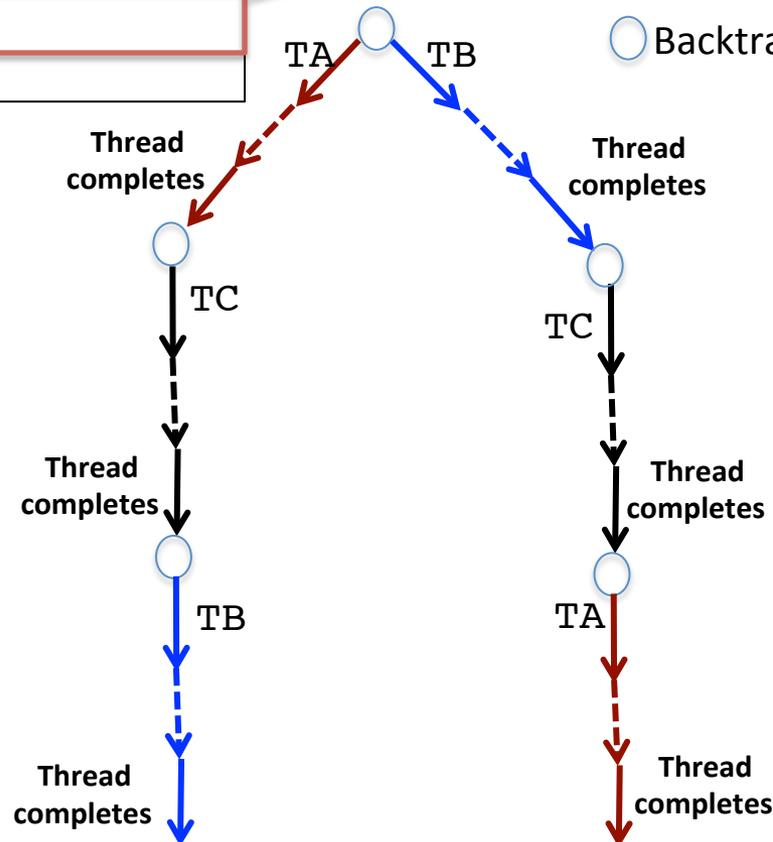
# First test: Run each thread sequentially until completion (No interleaving)

```
// 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  
}
```

Run selected thread until it completes



○ Backtrack/choice point





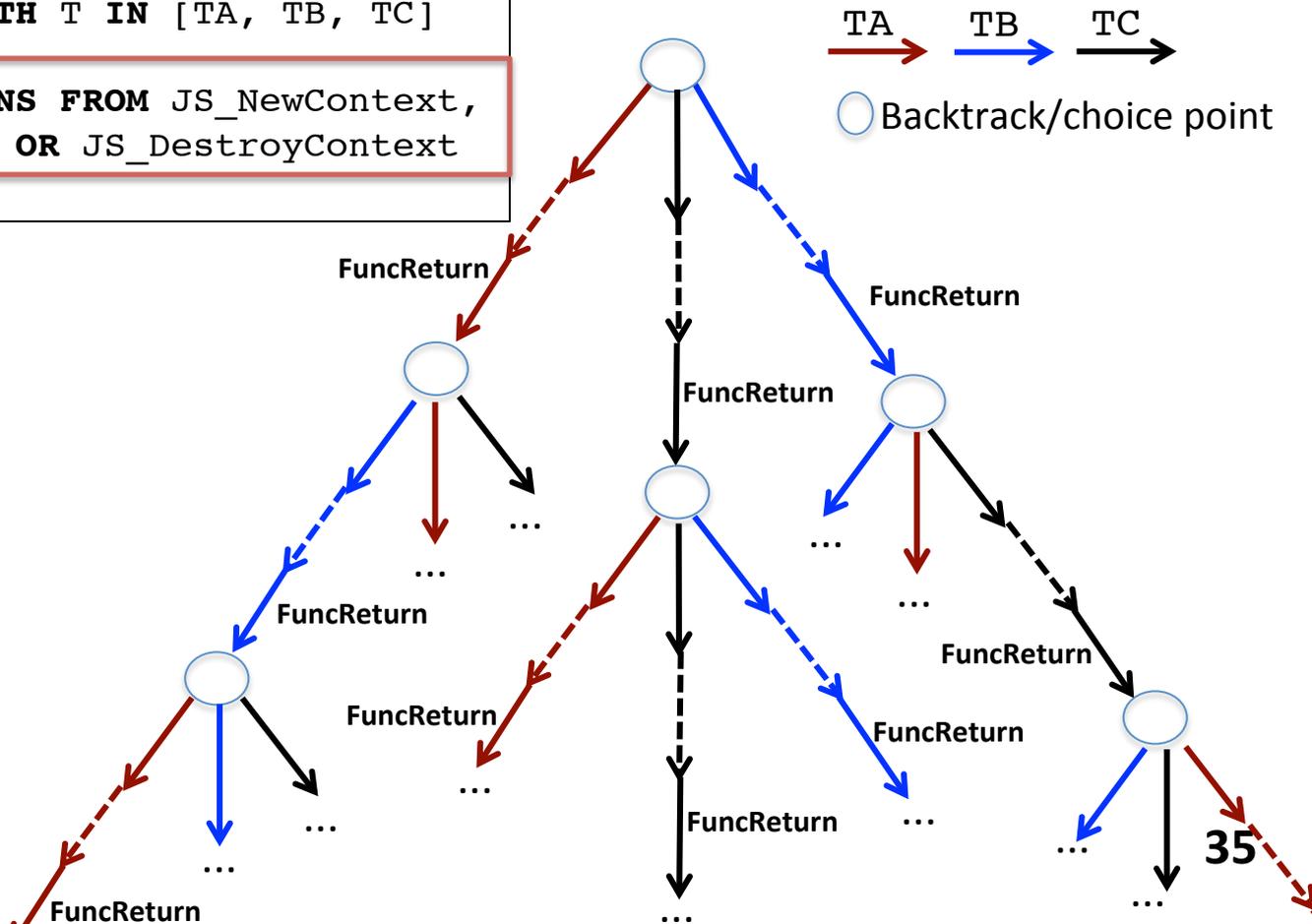
# Second test: Run each thread sequentially until it returns from function

```
// 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

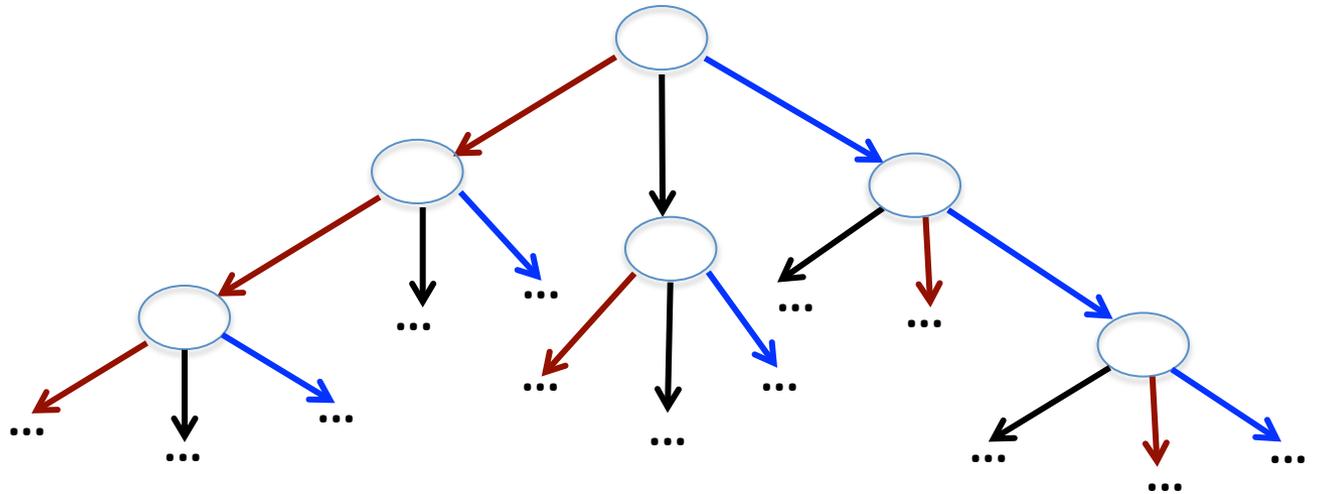
```
// Test in Concurrit DSL
1: TA, TB, TC = WAIT_FOR_DISTINCT_THREADS()
2: LOOP UNTIL TA, TB, TC COMPLETE {
3:   BACKTRACK HERE WITH T IN [TA, TB, TC]
4:   RUN T UNTIL NEXT EVENT
}
```

## Result:

> 100,000 schedules  
Assertion failure  
after a night!



○ Backtrack/choice point



# 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

# Possible buggy schedule from bug report

[Igor Bukanov](#) 2009-03-09 17:47:12 PDT

[Comment 5](#) [[reply](#)] [[-](#)] [[reply](#)] [[-](#)]

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 `js_DestroyContext` and thread C calls `js_NewContext`.

First thread A runs the GC and releases the lock. The last one so that the latter skips the GC.

Threads A, B

```
static void * testfunc(void *ignore)
```

Thread C

```
    JSContext *cx = JS_NewContext(rt, 0x1000);
```

```
    if (cx) {
```

```
        JS_BeginRequest(cx);
```

```
        JS_DestroyContext(cx);
```

```
    }
```

```
    return NULL;
```

```
}
```

At this stage the GC runs and releases the lock. The GC is DOWN.

Thread C runs

Thread C that runs

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](#).

# 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  
}
```

# Exploiting programmer's insights about bug

```
// Test in Concurr
```

```
1: TC = WAIT_FOR_T
```

```
2: TA = WAIT_FOR_D
```

```
3: TB = WAIT_FOR_D
```

```
4: LOOP UNTIL TA,
```

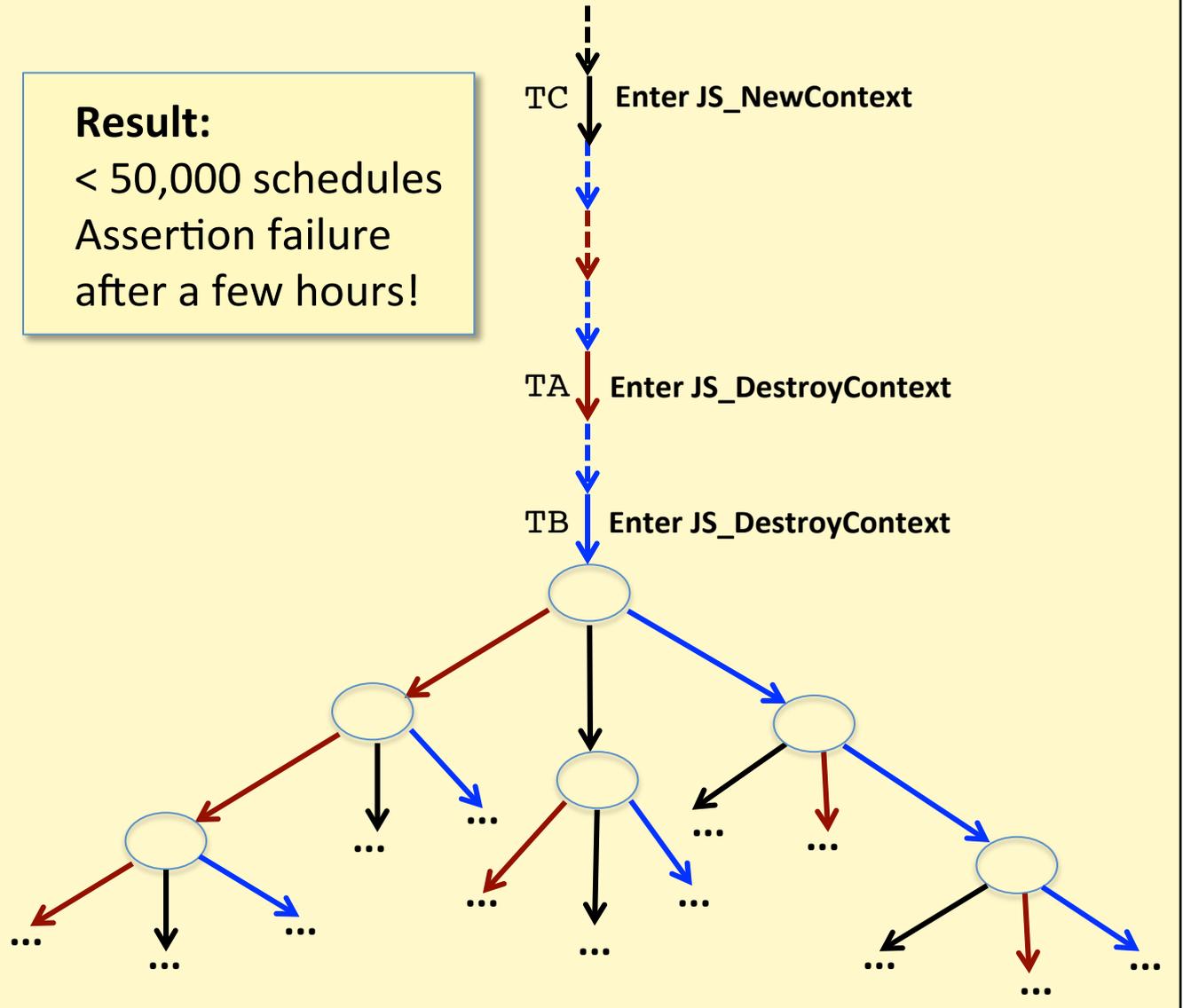
```
5:   BACKTRACK HE
```

```
6:   RUN T UNTIL
```

```
}
```

**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 `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](#).

# Exploiting programmer's insights about bug

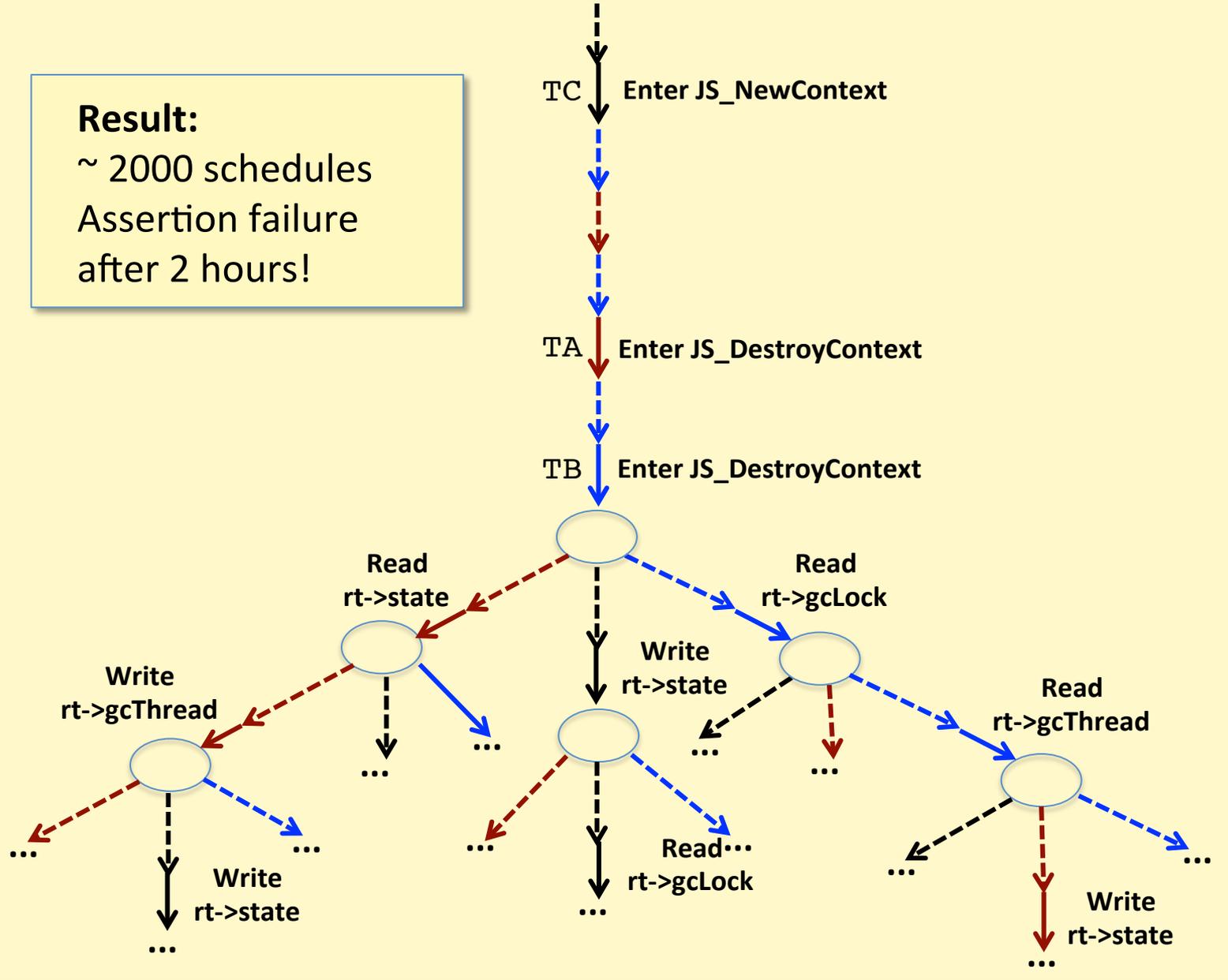
```
// Test in Concurrit DSL  
  
1: TC = WAIT_FOR_THREAD(ENTERS JS_NewContext)  
  
2: TA = WAIT_FOR_DISTINCT_THREAD(ENTERS JS_DestroyContext)  
  
3: TB = WAIT_FOR_DISTINCT_THREAD(ENTERS JS_DestroyContext)  
  
4: LOOP UNTIL TA, TB, TC COMPLETE {  
  
5:     BACKTRACK HERE WITH T IN [TA, TB, TC]  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

# Exploiting programmer's insights about bug

```
// Test in Co
1: TC = WAIT
2: TA = WAIT
3: TB = WAIT
4: LOOP UNTIL
5: BACKTRA
6: RUN T U
}
```

## Result:

~ 2000 schedules  
Assertion failure  
after 2 hours!



# Possible buggy schedule from bug report

[Igor Bukanov](#) 2009-03-09 17:47:12 PDT

[Comment 5](#) [[reply](#)] [[-](#)] [[reply](#)] [[-](#)]

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 calls `js_DestroyContext` and thread C calls

Setup

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 runtime list, discovers that it is the last one, calls `the-last-context-cleanup`, runs the GC and

Fixed, known schedule for threads A and B

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 that the GC lock is held by thread A. It calls `js_NewContext` that the newly allocated context has `rt->state` is DOWN, it releases the GC lock and starts its initialization procedure. That procedure involves allocating atoms and it will happen when the thread A finishes its GC. See to the first stack trace from the [comment 4](#).

Unknown schedule for A and C

# Final test

```
// 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

}
```

[Igor Bukanov](#) 2009-03-09 17:47:12 PDT

[Comment](#)

At least one problem that I can see from the code is that js  
if (rt->state != JSRTS\_UP && gckind != GC\_LAST\_CONTEXT)  
return;

outside the GC lock. Now  
calls js\_DestroyContext

Setup

First thread A removes its context from the runtime list. Th  
the last one so thread does not touch rt->state and eventual  
latter skips the above c

Before this moment the t  
runtime list, discovers  
the-last-context-cleanup

Fixed, known schedule  
for threads A and B

At this stage the thread A gets the GC lock, setup itself as  
runs the GC and releases the GC lock to proceed with the GC  
DOWN.

Now the thread C enters  
js\_NewContext that the n  
rt->state is DOWN, it re  
initialization procedure  
atoms and it will happen  
to the first stack trace from the [comment 4](#).

Unknown schedule  
for A and C

# Final test

[Igor Bukanov](#) 2009-03-09 17:47:12 PDT [Comment 5](#) [\[reply\]](#) [\[-\]](#) [\[reply\]](#) [\[-\]](#)

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;
```

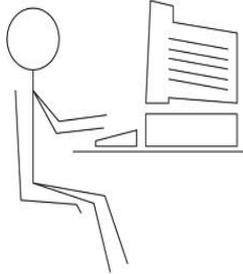
outside the GC lock. Now suppose there are 3 threads, A, B, C. Threads A and B calls `js_DestroyContext` and thread C calls `js_NewContext`.

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](#).



Software Under Test

.....  
.....

+

```
// 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
}
```

Triggers assertion failure  
in < 30 thread schedules  
(Add to regression test suit)

# Implementation/Evaluation

- **Implementation:** DSL embedded in C++
  - Prototype: <http://code.google.com/p/concurrit/>
- 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

```
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

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

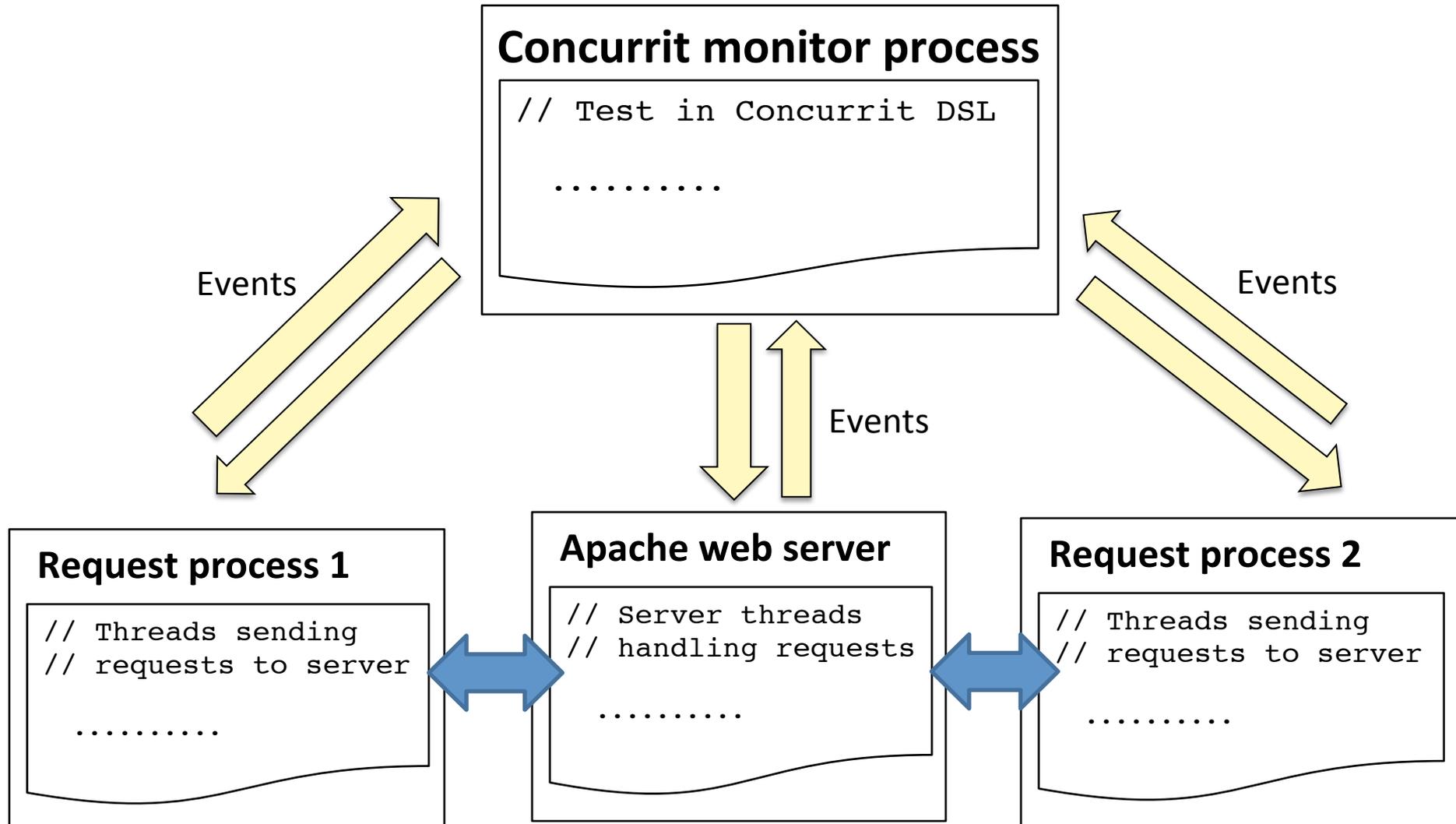
## Test in Concurrit DSL

Runs concurrently with SUT

.....  
.....  
.....  
.....  
.....  
.....

# Ongoing work: Integration testing

## Controlling multi-process/distributed applications



# 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.

