Performance Engineering of Software Systems November 3, 2021
Massachusetts Institute of Technology 6.172
Saman Amarasinghe, Charles E. Leiserson, and Jonathan Ragan-Kelley Handout 14

Homework 9: Synchronization and Nondeterminism

Due: 11:59 P.M. (ET) on Tuesday, November 9

Last Updated: November 3, 2021

In this homework, you will investigate concurrent data structures and study how to address the race conditions using locks and other synchronization methods. You will also explore sources of nondeterminism in computer programs, including sources that arise from the system environment.

Contents

1 Getting started

Getting the code

You can get this assignment's code using Git:

```
$ git clone git@github.mit.edu:6172-fall21/homework9_<username>.git homework9
```

Submitting your solutions

Please answer all the recitation Checkoff Items and ask your TA for a checkoff. Then, answer the writeup questions in this handout and submit them on Gradescope by the deadline stated at the top of this handout.

Don't forget to submit your code by pushing to your Git repository as well!

2 Hashtables

In this section, we investigate the nondeterministic behaviors one might observe in a concurrent hashtable.

2.1 Locking the hashtable

Take a look at `hashtable-mt.c`, which contains an implementation of an open address hashtable with linear probing. This technique is a lot more cache-friendly compared to hashtables with linked lists.

The code in its initial form has issues with races. If you make and then run `./hashtable-mt`, you may sometimes see errors. You can see errors more reliably using `$ make hashtable-mt100`, which runs the `hashtable-mt` binary 100 times.

We will add synchronization to fix the errors. One possibility is to use a single mutex lock for the entire hashtable, as in `hashtable_insert_locked()`. Such a lock, however, would be very highly contended in this application.

A good technique for reducing lock contention on a shared data structure is to use a fine-grained locking paradigm, where different locks are used to synchronize different parts of the data structure. For example, instead of a single "tablelock" that applies to the entire hashtable, we might employ an "entrylock" approach and assign a distinct lock for each entry in the hashtable's array.

Although fine-grained locking can reduce lock contention, the entrylock approach incurs overheads in memory consumption and caching. To reduce these overheads, we shall explore a "stripelock" approach, which employs k locks: in order to access the i-th entry, we must first acquire the $(i \ \% \ k)$-th lock. This stripelock approach is finer-grained than a tablelock, to avoid contention, but coarser-grained than entrylocks, to reduce memory and caching overheads.

Modify `hashtable_fill()` to use `hashtable_insert_fair()`, which is a rather basic implementation of a stripelock.

> **Checkoff Item 1:** What problem is introduced by the fairness solution in `hashtable_insert_fair()`? Explain the behavior you see.

2.2 Lock-free concurrent hashtable

Let us now try to avoid using a lock altogether. Modern processors support compare-and-swap (CAS) instructions, where `CAS(addr, old_val, new_val)` is an atomic instruction that has the following effect:

```
if (*addr == old_val) {
  *addr = new_val;
  return true;
}
return false;
```

On modern Intel processors, the assembly instruction `CMPXCHG16B` implements the compare-and-swap operation for 16 bytes, and similar instructions exist for 4 bytes and 8 bytes. Early 64-bit processors, however, didn't support the 16-byte instruction. In this vein, we will use the `CMPXCHG8B` in our implementation to only work with 64-bit words. This means that we cannot atomically write a whole `entry_t`.

Checkoff Item 2: Use `InterlockedCompareExchange64()` (defined in `common.h`) to implement your changes in `hashtable_insert_lockless()`. Don't forget to modify `hashtable_fill()` to use your new code. Run `$ make hashtable-mt100` to ensure that it works. Describe your implementation.

3 FIFO queues

In this section, we examine the concurrent first-in first-out (FIFO) queue. A FIFO queue data structure supports `enqueue()` and `dequeue()` operations that allow users to add and remove elements, respectively, from the queue. The FIFO queue ensures that an element i added to the queue is removed from the queue before any element j that was added after i. Here, we focus on the theoretical side of the material shown in lecture; there is no implementation involved.

Figures 1–3 (at the end of this handout) present C-like pseudocodes of two implementations of a FIFO queue. Figure 1 shows a lock-based implementation, and Figures 2–3 show a lock-free implementation. In both queue implementations, a pool of nodes is allocated in advance. A call to `new_node()` grabs a free node from the pool of nodes, and `free_node(node)` returns `node` to the pool. For the questions below, assume that `CAS()` (described in Section 2.2) can operate on the entire `pointer_t`, the compiler cannot change the order of instructions, and there are always enough free nodes in the pool to perform all enqueue operations. Assume also that the nodes in the queue do not cross cache lines, thus all writes are atomic.

Read both implementations carefully. Before you start answering the following write-up questions, you may find it helpful to draw diagrams of an empty queue and a queue with a few nodes. Using these diagrams, try to understand how nodes are inserted and deleted from the queue in both implementations.

Note that the first node added in the initialization of both the lock-based and lock-free version of the queue is a dummy value and never dequeued. It is only used to denote an empty queue.

Checkoff Item 3: Add comments to the lock-free FIFO queue code (Figures 2 and 3) to explain what each line does, following the style of comments in the lock-based code (Figure 1).

Write-up 1: What is the advantage of using two locks over one lock?

Write-up 2: Carefully look at the code for the lock-free `enqueue()` operation and answer the following questions:

1. How many successful `CAS()` instructions are needed per node?
2. What happens if the `CAS()` in line 86 fails?
3. How far behind can the tail lag?
4. Is the program correct without line 86?

Write-up 3: Carefully look at the code for the lock-free dequeue operation and answer the following questions:

1. Line 94 checks what was already assigned in line 91. Why do we need line line 94?
2. In line 101, the value of the node is read before the head is updated in line 102. Why is this important? What can happen if we change the order of these two lines?
3. What happens if the `CAS()` instruction in line 102 is unsuccessful?

Write-up 4: Which implementation do you expect will run faster — the lock-based or the lock-free one? Explain your answer in terms of cost of the synchronization primitives, contention, synchronization overhead, etc.

Write-up 5: *(Optional)* Show how to simplify the lock-based code if only one thread may enqueue nodes to the queue. Write the pseudocode and comment it. Explain why your solution is correct (i.e., any execution sequence satisfies the FIFO ordering).

Write-up 6: *(Optional)* Show how to simplify the lock-free code if only one thread may dequeue nodes from the queue. Write the pseudocode and comment it. Explain why your solution is correct (i.e., any execution sequence satisfies the FIFO ordering) and why it is non-blocking.

Write-up 7: Explain how `count` is used in the lock-free code to handle the ABA problem discussed in lecture or recitation.

4 Optional: Environmental sources of nondeterminism

The most entertaining bugs to debug are the ones that depend on the system environment in which the program is run. Why would a bug happen only on `awsrun` but not on your development machine? Why would code that works great for you occasionally crash for your partner and always crash for your TA? Let's dig deeper into this using the simple program excerpt from `undef.c` shown in Figure 4.

The `Makefile` contains a number of useful targets for this exercise. Whenever you use a target, you may find it useful to examine the `Makefile` in detail and understand why certain targets have the result they do. Compare the results of running this program a few times using the following command:

```
$ make undef-compare
```

What parts of the output change from run to run?

Let us now see how to fix these nondeterministic outputs line by line.

Whenever your program is using undefined state, it will produce nondeterministic results. Undefined variables (or bits within one) are one such source of nondeterminism. Fix the code to define the variable and rerun `$ make undef-compare` to make sure that it worked.

Next, we will look at nondeterminism in addresses. The higher order bits of a pointer are random due to ***Address Space Layout Randomization*** (***ASLR***),[1] which is an important security feature now in all modern operating systems. But by randomizing memory addresses, ASLR can affect the behavior (and performance) of a program. To test how it impacts your address-dependent nondeterministic program, you can run your program without ASLR with the `setarch` command, as follows:

```
$ setarch x86_64 -R ./undef
```

Run `$ make undef-noaslr` to run the program a few times without ASLR. You should discover that certain memory addresses are now the same from run to run. How did disabling ASLR affect the program's execution?

4.1 Nondeterministic hashtables

Take a look at `hashtable-serial.c`. Try to run `./hashtable-serial` a few times. It may work for you most of the time. How about `$ make hashtable-serial100`, which runs this binary 100 times?

Write-up 8: *(Optional)* What do you need to do to make this program deterministic? Without altering `hashtable-serial.c`, modify the Makefile target for `hashtable-serial-good` so that the program is deterministic and all runs succeed. You may find it useful to examine `hashtable-serial.c` to see what system arguments it takes.

Before we fix our bugs, we want them to be reproducible.

Write-up 9: *(Optional)* Modify the Makefile target for `hashtable-serial-bad` so that the program is deterministic and always fails.

[1] https://en.wikipedia.org/wiki/Address_space_layout_randomization

Use any technique (GDB, `printf()` statement, eyeballing, etc) to find and fix the bug in `hashtable_insert()`.

Write-up 10: *(Optional)* What was the bug? What is your fix? Rerun `$ make hashtable-serial100` to ensure that your fix always works.

4.2 Reference: Replay debugging

To fix serial code, you can use the Process Record facility in GDB.[2] This facility allows the user to record the execution of a program and replay that execution, and to debug a program in reverse, that is, to step backwards through the program's execution. Here is an example of how you can use these facilities.

```
(gdb) break main
(gdb) run
(gdb) record
(gdb) continue
(gdb) reverse-next
(gdb) reverse-next
(gdb) reverse-continue
```

You can also execute these commands more succinctly using their aliases, as follows:

```
(gdb) b main
(gdb) r
(gdb) rec
(gdb) c
(gdb) rn
(gdb) rn
(gdb) rc
```

You may find it useful to watch some variables when debugging. You can instruct GDB to watch a variable as follows:

```
(gdb) watch some_variable_name
```

You don't necessarily have to use these tools — if you have plenty of time, you can always figure out any bug by staring at the code long enough (static analysis by eyeballing). However, a much easier approach is to run your code with assertions, `printf()` statements, or replay debugging techniques like in `gdb`.

[2] http://www.sourceware.org/gdb/wiki/ProcessRecord/Tutorial

```
01 struct node_t {
02   data_t value;
03   node_t* next;
04 };
05 struct queue_t {
06   node_t* head;
07   node_t* tail;
08   mutex_t h_lock;
09   mutex_t t_lock;
10 };
11
12 void initialize(queue_t* q, data_t value) {
13   node_t* node = new_node();  // Allocate a new node
14   node->value = value;
15   node->next = NULL;          // Make it the only node in the queue
16   q->head = node;             // Both head and tail point to it
17   q->tail = node;
18   q->h_lock = FREE;           // Locks are initially free
19   q->t_lock = FREE;
20 }
21
22 void enqueue(queue_t* q, data_t value) {
23   node_t* node = new_node();  // Allocate a new node
24   node->value = value;        // Copy enqueued value into node
25   node->next = NULL;          // Set next pointer of node to NULL
26   lock(&q->t_lock);           // Acquire t_lock to access tail
27   q->tail->next = node;       // Append node at the end of queue
28   q->tail = node;             // Swing tail to node
29   unlock(&q->t_lock);         // Release t_lock
30 }
31
32 bool dequeue(queue_t* q, data_t* pvalue) {
33   lock(&q->h_lock);           // Acquire h_lock to access head
34   node_t* node = q->head;     // Read head
35   new_head = node->next;      // Read next pointer
36   if (new_head == NULL) {     // Is queue empty?
37     unlock(&q->h_lock);       // Release h_lock before return
38     return false;             // Queue was empty
39   }
40
41   *pvalue = new_head->value;  // Queue not empty. Read value
42   q->head = new_head;         // Swing head to next node
43   unlock(&q->h_lock);         // Release h_lock
44   free_node(node);            // Free node
45   return true;                // Dequeue succeeded
46 }
```

Figure 1: C-like pseudocode for declaring, initializing, enqueueing, and dequeueing with a lock-based FIFO queue.

```
47 struct pointer_t {
48   node_t* ptr;
49   unsigned int count;
50 };
51 struct node_t {
52   data_t value;
53   pointer_t next;
54 };
55 struct queue_t {
56   pointer_t head;
57   pointer_t tail;
58 };
59
60 void initialize(queue_t* q, data_t value) {
61   node_t* node = new_node();
62   node->value = value;
63   node->next.ptr = NULL;
64   q->head.ptr = node;
65   q->tail.ptr = node;
66 }
67
68 void enqueue(queue_t* q, data_t value) {
69   node_t* node = new_node();
70   node->value = value;
71   node->next.ptr = NULL;
72   pointer_t tail;
73   while (true) {
74     tail = q->tail;
75     pointer_t next = tail.ptr->next;
76     if (tail == q->tail) {
77       if (next.ptr == NULL) {
78         if (CAS(&tail.ptr->next, next, (struct pointer_t) {node, next.count + 1 })) {
79           break;
80         }
81       } else {
82         CAS(&q->tail, tail, (struct pointer_t) {next.ptr, tail.count + 1 });
83       }
84     }
85   }
86   CAS(&q->tail, tail, (struct pointer_t) { node, tail.count + 1 });
87 }
```

Figure 2: C-like pseudocode for declaring, initializing, and enqueueing with a lock-free FIFO queue.

```
88 bool dequeue(queue_t* q, data_t* pvalue) {
89   pointer_t head;
90   while (true) {
91     head = q->head;
92     pointer_t tail = q->tail;
93     pointer_t next = head.ptr->next;
94     if (head == q->head) {
95       if (head.ptr == tail.ptr) {
96         if (next.ptr == NULL) {
97           return false;
98         }
99         CAS(&q->tail, tail, (struct pointer_t) { next.ptr, tail.count + 1});
100      } else {
101        *pvalue = next.ptr->value;
102        if (CAS(&q->head, head, (struct pointer_t) { next.ptr, head.count + 1})) {
103          break;
104        }
105      }
106    }
107  }
108  free_node(head.ptr);
109  return true;
110 }
```

Figure 3: C-like pseudocode for dequeueing with a lock-free FIFO queue.

```
111 int main() {
112   int i;
113   printf("value of i=%d\n", i);
114   printf("address &i=%p\n", &i);
115   printf("hash of i=%ld\n", ((uintptr_t)&i) & 127);
116   return 0;
117 }
```

Figure 4: An excerpt from `undef.c`.