Performance Engineering of Software Systems
Massachusetts Institute of Technology — 6.172
Prof. Charles E. Leiserson and Prof. Julian Shun — Handout 16

Homework 10: Data Synchronization

Introduction

The focus of this problem set is the theoretical side of the material taught in class. This problem set focuses on data synchronization and comparing lock-based and lock-free FIFO queue implementations.

1 Data synchronization

Figures 1 and 2 present two implementations of a FIFO queue. Figure 1 is a lock-based implementation, and Figure 2 is 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 the node `node` to the pool. In the implementation of the lock-free queue, the compare-and-swap instruction `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;
```

For the questions below, assume that `CAS` can operate on the entire `pointer_t`, that the compiler cannot change the order of instructions, and that 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, and thus all writes are atomic.

Read both implementations carefully. Before you start answering the 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 initialization of both the lock-based and lock-free version of the deque is a dummy value and never dequeued. It is only used to denote an empty deque.

2 Check-off Questions

1. What are constraints on `enqueue` and `dequeue` in the FIFO queue? You do not need to look at the code yet.

2. What is the advantage of using two locks over one lock?

3. In the style of comments of the lock-based FIFO queue code, add comments to the lock-free code (on paper), explaining what each line does. The comments should be short and precise (not more than 10 words each). We have provided you a copy of the code in Figure 2.

4. Explain how a new node is inserted into the lock-free queue. How many successful `CAS`es are needed per node? What happens if the `CAS` in line 96 fails? How far can the tail lag behind? Is the program correct without line 96?

5. Carefully look at the code for the lock-free dequeue operation and answer the following questions:

   (a) Line 104 checks what was already assigned in line 101. Why do we need line line 104?

   (b) In line 111 the value of the node is read before the head is updated in line 112. Why is this important? What can happen if we change the order of the lines?

   (c) What happens if the `CAS` in line 112 is unsuccessful?

6. Which implementation do you expect to run faster — the lock-based or the lock-free? Explain your answer in terms of cost of the synchronization primitives, contention, synchronization overhead, etc.

7. 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 in your own words why your solution is correct (i.e. any execution sequence keeps the FIFO ordering).

8. 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 in your own words why your solution is correct (i.e. any execution sequence keeps the FIFO ordering) and why it is non-blocking.

9. Explain how `count` is used to handle the ABA problem discussed in recitation.

```
struct node_t {
  data_t value;
  node_t* next;
};
struct queue_t {
  node_t* head;
  node_t* tail;
  mutex_t h_lock;
  mutex_t t_lock;
};

void initialize(queue_t* q, data_t value) {
  node_t* node = new_node();  // Allocate a new node
  node->value = value;
  node->next = NULL;          // Make it the only node in the queue
  q->head = node;             // Both head and tail point to it
  q->tail = node;
  q->h_lock = FREE;           // Locks are initially free
  q->t_lock = FREE;
}

void enqueue(queue_t* q, data_t value) {
  node_t* node = new_node();  // Allocate a new node
  node->value = value;        // Copy enqueued value into node
  node->next = NULL;          // Set next pointer of node to NULL
  lock(&q->t_lock);           // Acquire t_lock to access tail
  q->tail->next = node;        // Append node at the end of queue
  q->tail = node;              // Swing tail to node
  unlock(&q->t_lock);         // Release t_lock
}

bool dequeue(queue_t* q, data_t* pvalue) {
  lock(&q->h_lock);           // Acquire h_lock to access head
  node_t* node = q->head;     // Read head
  new_head = node->next;      // Read next pointer
  if (new_head == NULL) {     // Is queue empty?
    unlock(&q->h_lock);       // Release h_lock before return
    return false;             // Queue was empty
  }

  *pvalue = new_head->value;  // Queue not empty. Read value
  q->head = new_head;         // Swing head to next node
  unlock(&q->h_lock);         // Release h_lock
  free_node(node);            // Free node
  return true;                // Dequeue succeeded
}
```

Figure 1: C-like pseudocode declaring, initializing, and adding for lock-based FIFO queue.

```
struct pointer_t {
  node_t* ptr;
  unsigned int count;
};
struct node_t {
  data_t value;
  pointer_t next;
};
struct queue_t {
  pointer_t head;
  pointer_t tail;
};

void initialize(queue_t* q, data_t value) {
  node_t* node = new_node();
  node->value = value;
  node->next.ptr = NULL;
  q->head.ptr = node;
  q->tail.ptr = node;
}

void enqueue(queue_t* q, data_t value) {
  node_t* node = new_node();
  node->value = value;
  node->next.ptr = NULL;
  pointer_t tail;
  while (true) {
    tail = q->tail;
    pointer_t next = tail.ptr->next;
    if (tail == q->tail) {
      if (next.ptr == NULL) {
        if (CAS(&tail.ptr->next, next, (struct pointer_t) {node, next.count + 1 })) {
          break;
        }
      } else {
        CAS(&q->tail, tail, (struct pointer_t) {next.ptr, tail.count + 1 });
      }
    }
  }
  CAS(&q->tail, tail, (struct pointer_t) { node, tail.count + 1 });
}
```

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

```c
bool dequeue(queue_t* q, data_t* pvalue) {
  pointer_t head;
  while (true) {
    head = q->head;
    pointer_t tail = q->tail;
    pointer_t next = head.ptr->next;
    if (head == q->head) {
      if (head.ptr == tail.ptr) {
        if (next.ptr == NULL) {
          return false;
        }
        CAS(&q->tail, tail, (struct pointer_t) { next.ptr, tail.count + 1});
      } else {
        *pvalue = next.ptr->value;
        if (CAS(&q->head, head, (struct pointer_t) { next.ptr, head.count + 1})) {
          break;
        }
      }
    }
  }
  free_node(head.ptr);
  return true;
}
```

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