**Problem Set 1**

Problem Set 1

# Parallelism and Data Synchronization

## Introduction

The focus of this project is the theoretical side of the material taught in class. However, feel free to test your solutions by writing *short* programs and to use profiling tools to test your answers. The first half of the problem set focuses on data synchronization and comparing lock-based and lock-free FIFO queue implementations. In the second part you are asked to design, implement and analyse short Cilk++ programs.

This project should be done *individually*. Please treat it like a take-home quiz. Do not discuss problems with classmates, and cite any external resources you use clearly (i.e. books, published papers, wikipedia ...).

Please submit a PDF file with your answers on the class stellar web site.

## 1 Data synchronization

Figure 1 and Figure 2 are both implementing a FIFO queue. Figure 1 is a lock-based implementation and Figure 2 is a lock free implementation. Note that 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)` returns the node to the pool. In the implementation of the lock-free queue, the Compare-And-Swap (CAS) instruction returns TRUE if the value stored in the memory location equals the old value and thus the memory location was successfully updated with the new value. Otherwise, it returns FALSE. The ABA problem is solved by splitting the pointer to a node into two parts - one is the pointer itself and the other one is a counter. Both parts fit into one machine word and are read and written together (Since we cannot use the whole word as a pointer to memory, we must have the pre-allocated pool of nodes, and the number of nodes we can have is limited by the number of bits that are used in the pointer part of the word).For the questions below, assume that the compiler cannot change the order of instructions, and that there is always enough free nodes in the pool to perform all enqueue operations.

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.

1. What is the advantage of using two locks over one lock?
2. In the style of comments of the lock-based FIFO queue code, add comments to the lock-free code, explaining what each line does. The comments should be short and precise (not more than 10 words each). A text file with the code is provided. Copy it to your document and add your comments at the end of each line.

3. Explain how a new node is inserted into the lock-free queue. How many CASes are needed per node? What happens if the CAS in E17 fails? How far can the tail lag behind?

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

   (a) Line D5 checks what was already assigned in line D2. Why do we need line D5 ?

   (b) In line D12 the value of the node is read before the Head is updated in line D13. Why is this important? What can happen if we change the order of the lines?

   (c) What happens if the CAS in line D13 is unsuccessful?

5. 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.

6. Show how to simplify the lock-based code if only one process may enqueue nodes to the queue. Write the pseudo code and comment it. Explain in your own words why your solution is correct (i.e. any execution sequence keeps the FIFO ordering).

7. Show how to simplify the lock-free code if only one process may dequeue nodes from the queue. Write the pseudo code 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.

```
structure node_t {value: data type, next: pointer to node_t}
structure queue_t {Head: pointer to node_t, Tail: pointer to node_t,
                 H_lock: lock type, T_lock: lock type}

initialize(Q: pointer to queue_t)
   node = new_node()          // Allocate a free node
   node->next = NULL          // Make it the only node in the linked list
   Q->Head = Q->Tail = node   // Both Head and Tail point to it
   Q->H_lock = Q->T_lock = FREE       // Locks are initially free

enqueue(Q: pointer to queue_t, value: data type)
   node = new_node()          // Allocate a new node from the free list
   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 in order to access Tail
      Q->Tail->next = node    // Link node at the end of the linked list
      Q->Tail = node          // Swing Tail to node
   unlock(&Q->T_lock)         // Release T_lock

dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
   lock(&Q->H_lock)           // Acquire H_lock in order to access Head
      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
      endif
      *pvalue = new_head->value   // Queue not empty.  Read value before release
      Q->Head = new_head          // Swing Head to next node
   unlock(&Q->H_lock)             // Release H_lock
   free(node)                     // Free node
   return} TRUE                   // Queue was not empty, dequeue succeeded
```

**Figure 1:** Lock based FIFO queue

## 2 Parallelism using Cilk++

1. Write a short Cilk++ program that uses a reducer of your own design to determine whether a string of parentheses over the set { ”(”, ”)” } is well formed. For example, “(())()” is well formed, but “(()))(()” is not. Your reduce function should run in O(1) time. Analyze the asymptotic work and span of your solution.

2. Answer problem 27-4 from CLRS third edition. The chapter on Multithreaded Algorithms is available on the class stellar site.

```
structure pointer_t {ptr: pointer to node_t, count: unsigned integer}
  structure node_t {value: data type, next: pointer_t}
  structure queue_t {Head: pointer_t, Tail: pointer_t}

  initialize(Q: pointer to queue_t)
     node = new_node()
     node->next.ptr = NULL
     Q->Head.ptr = Q->Tail.ptr = node

  enqueue(Q: pointer to queue_t, value: data type)
   E1:   node = new_node()
   E2:   node->value = value
   E3:   node->next.ptr = NULL
   E4:   loop
   E5:      tail = Q->Tail
   E6:      next = tail.ptr->next
   E7:      if tail == Q->Tail
   E8:         if next.ptr == NULL
   E9:            if CAS(&tail.ptr->next, next, <node, next.count+1>)
  E10:               break
  E11:            endif
  E12:         else
  E13:            CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
  E14:         endif
  E15:      endif
  E16:   endloop
  E17:   CAS(&Q->Tail, tail, <node, tail.count+1>)

  dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
   D1:   loop
   D2:      head = Q->Head
   D3:      tail = Q->Tail
   D4:      next = head.ptr->next
   D5:      if head == Q->Head
   D6:         if head.ptr == tail.ptr
   D7:            if next.ptr == NULL
   D8:               return FALSE
   D9:            endif
  D10:            CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
  D11:         else
  D12:            *pvalue = next.ptr->value
  D13:            if CAS(&Q->Head, head, <next.ptr, head.count+1>)
  D14:               break
  D15:            endif
  D16:         endif
  D17:      endif
  D18:   endloop
  D19:   free(head.ptr)
  D20:   return TRUE
```

**Figure 2:** Lock Free FIFO queue