EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 13: October 11

Lecturer: Vijay Garg *Scribe: Harsh Yadav*

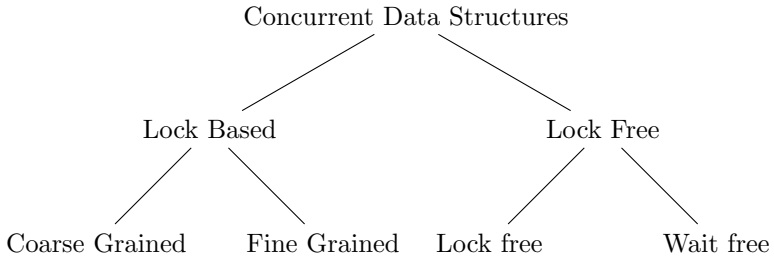
Agenda

The lecture focused on Concurrent Data Structures. Chapter-5 (Wait Free Synchronization) will be continued on next class meeting. Major concepts discussed -:

- Classifications of Concurrent Data Structures
- Concurrent Stacks
- Concurrent Queues
- Concurrent Linked Lists

13.1 Introduction

A **Concurrent Data Structure** is a particular way of storing and organizing data for access by multiple computing threads (or processes) on a computer. Concurrent Data Structures are classified as following -:



The main difference between lock based and lock free implementation lies in the technique to enable mutual exclusion. Lock based primarily works on the concept of *locking, mutually exclusive operation and then unlocking*. However, In lock-free, there is no explicit locking. The mutually exclusive operations happen using *CAS - Compare and Set operation*. The process keeps on trying in a while loop until *CAS* succeeds. *CAS* acts as the linearization point in lock-free scheme. (*Refer to AtomicInteger.Java for an example of lock-free using CAS on course github page*).

Coarse Grained - It refers to locking at entry and unlocking at exit while using a data-structure.It is less efficient in terms of exploiting concurrency. It is relatively easier to implement and should be used when Data-structure is not in the critical path (Data Structure's performance is not of much significance).

Fine Grained - It refers to using multiple locks and implement a more finer locking scheme into various operations instead of explicitly locking at entry and unlocking at exit. (Difficult to implement)

Code 1: Lock Free Implementation

```
int regularLockFreeOperation(){
    ...
    ...
    while(true){
        if(CAS(....)){
            return;
        }else{
            Thread.yield();
        }
    }
}
```

13.2 Concurrent Stacks

The main operations associated with stacks are -:

- **Push**
- **Pop**

The concurrent stack implementation is pretty straightforward.

Lock Based - Lock based stack can be implemented by simple usage of monitors (*Synchronized and Reentrant locks*) for *push* and *pop* operations.

Lock free - Lock free stack implementation can be achieved by standard *CAS* operation in a while loop as discussed in the previous section. (*Refer to LockFreeStack.java on course github page*)

NOTE - One thing to note while doing lock-free implementations using above discussed technique is to make while() loop as tight as possible. (ex - Refrain from allocating memory inside the while() loop)

Speedup trick for Lock-free Stack (Cancellation of Operations) - If there are concurrent push and pop operations happening on stack as shown in Fig 13.1, Stack can be made faster. As shown in the figure, only one of the *push* operation will succeed (either *push(70) or push(30)*).Let's say push(70) succeeded. As, there is a concurrent pop() operation happening, *push(30)* can be cancelled by *pop()* operation. In implementation, an Exchanger has to be implemented to keep record of all the operations and cancel them if a scenario like this exists.

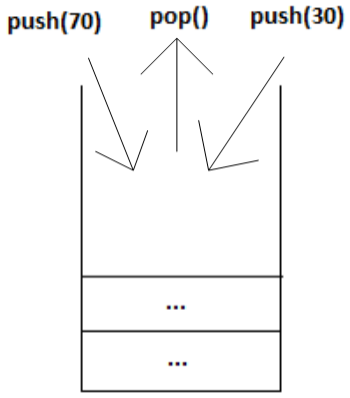
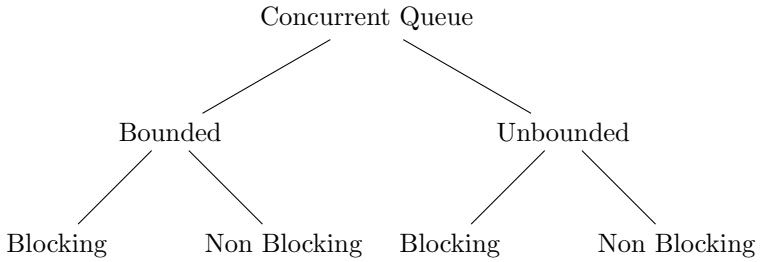


Figure 13.1: Concurrent Stack

13.3 Concurrent Queue

The main operations associated with stacks are -:

- **Enqueue**
- **Dequeue**



Each of the leaf nodes can be further classified into lock based and lock free.

13.3.1 Lock Based

Lock based implementation is usually achieved by using standard monitors. One tweak to fasten lock based implementations is to use multiple locks instead of using single lock. The reason why multiple locks are utilized here and not for Stacks is that operations are happening in queue at both ends *(Enqueue - Tail, Dequeue - Head)* as opposed to Stack where both *(Push and Pop)* operations were happening at the top of the stack. (*Refer to UnboundedQueue.Java and unbounded-total-lock-based-queue.txt on course github page*)

NOTE - Queue implementation can be made more faster by using the concept of *Sentinel* node. As we know, In both *Dequeue & Enqueue* operations, we have to check first that the head is not null. The *if*

statement gets executed in multiple instructions at processor level. These checkings can be avoided if Queue is designed to have at-least one node, even when it is empty (*Sentinel node*). One more thing that can be kept in mind for *Dequeue* operation is to shift the sentinel node(head) to next node instead of pointing the old head to the next. The concep of *Sentinel* node is more beneficial for Lock free implementation.

13.3.2 No lock No CAS queue

This is a special version of queue (bounded) which can be implemented without any Locks as well as without any CAS operations. The important thing to note here is this type of implementation can be done only for *Single Consumer Single Producer* scenario (*Refer to SingleQueue.Java on course github page*). Here *put* is reading the value of *head* and *get* is reading the value of *tail*. The consistency conditions on reading *head & tail* will give either the current value or the previous value depending on whether any concurrent operation is happening to update these registers or not. In both the cases, the legality of both *put() & get()* operations is maintained. It is impossible to make a Lock free and CAS free queue for multiple consumers or multiple producers scenario.

Code 2: No Lock No CAS Queue

```
public void put(Object x) {
  while (tail - head == items.length) {}; //busywait
  items[tail % items.length] = x;
  tail++;
}
public Object get() {
  while (tail - head == 0) {}; // busywait
  Object x = items[head % items.length];
  head++;
  return x;
}
```

13.3.3 Lock Free (Michael and Scott's Algorithm)

The usual Lock free implementation uses CAS operation. One problem that may arise in lock-free queue using CAS operations is **ABA problem**.

13.3.3.1 ABA problem

ABA problem occurs during synchronization, when a location is read twice, has the same value for both reads, and "value is the same" is used to indicate "nothing has changed". However, another thread can execute between the two reads and change the value, do other work, then change the value back, thus fooling the first thread into thinking "nothing has changed" even though the second thread did work that violates that assumption. Suppose a thread is in the process of swinging the *tail* pointer from *A* to *P* (while enquing P) using a CAS operation. Before the CAS happens, another thread T1 comes and swings the pointer to B (B enqueued). Now, CAS fails in compare operation. Now memory location A is free. Suppose another thread comes and swings the Tail pointer to A (in a new enqueue operation). Now the previous CAS operation which failed in compare will succeed and behave as if nothing has changed which is not the case as there is a separate enqueue(B) which happened. Note that this condition is very rare as it is very less probability for a new enqueue operation to exactly use the same memory location (A) just after it got freed.

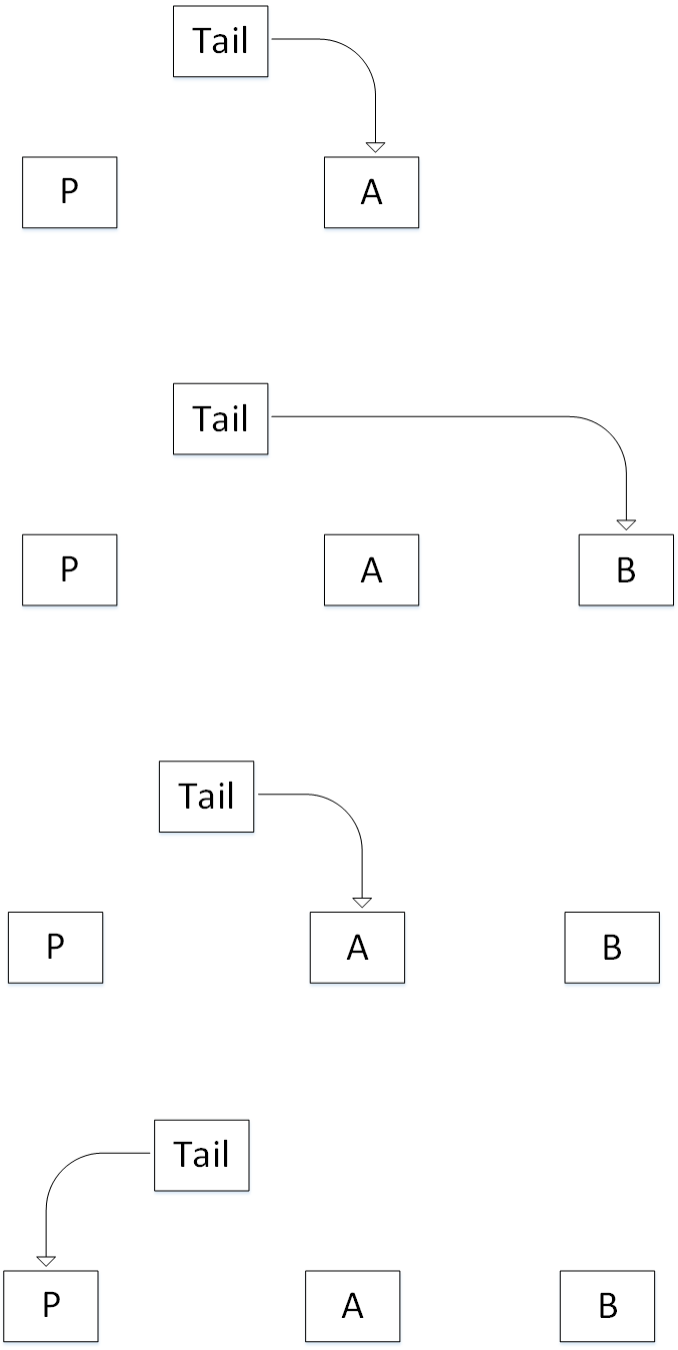


Figure 13.2: ABA problem

Solution - The solution to this problem was given by *Michael and Scott.*The solution is based on attaching a number to the pointer and increment it every time the Tail is assigned to the pointer. In this way, CAS (which reads a complete word) will be able to identify that A is a different version while doing the CAS operation second time in the above discussed scenario.

13.3.3.2 Helping

- Another feature given by *Michael and Scott's* algorithm is *helping*. As, In the lock-free scenario, the state of a process is visible, another process can take on the work of a process (who haven't finished yet) and finish it to speedup the execution. In the M&S imlementation, there are two CAS operations in the main loop *(in both enqueue and dequeue)*. Lets take an instance of enqueue operation, the first CAS is trying to add a new node at the end of the linked list (Linked List implementation of queue). If first CAS is successful, enqueue is completed and process exits from the loop. If CAS is not successful, then the process tries to help some other process (2nd CAS operation in loop) which is in the middle of enqueue to swing tail to the next node (2nd CAS operation - idempotent). After that, the process starts again for its own enqueue.

13.4 Concurrent Linked List

The main operations associated with stacks are -:

- **Add**
- **Remove**
- **Contains** - read only operation (should occur faster, avoid using locks in it)

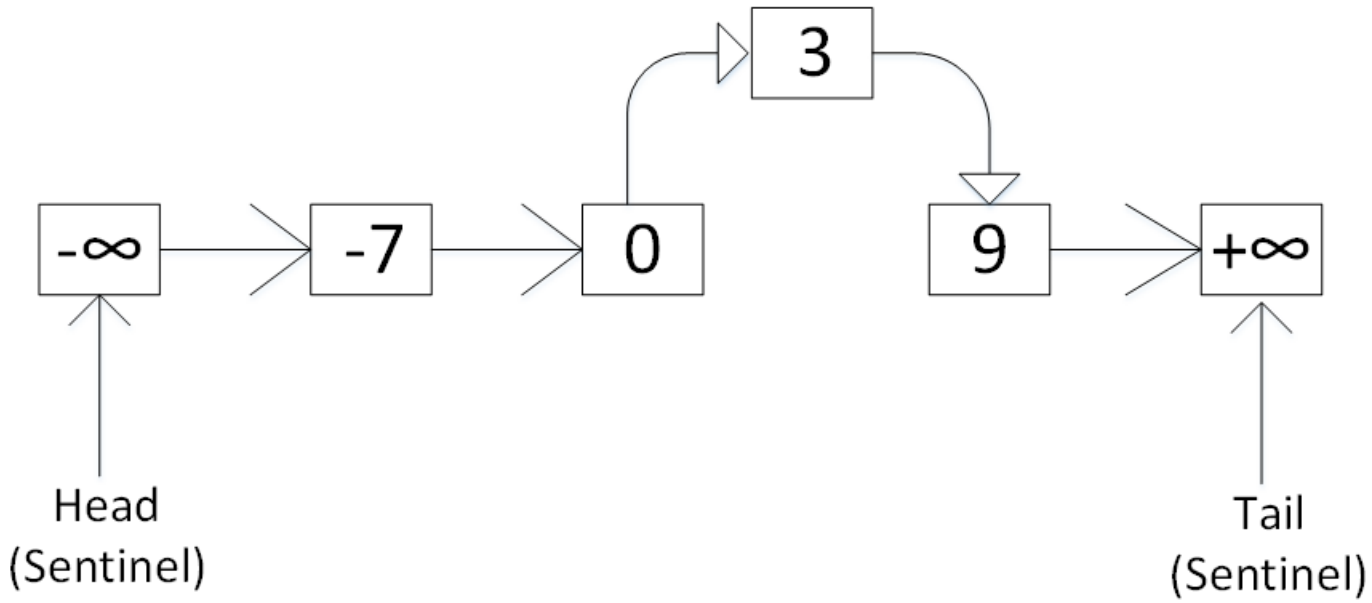


Figure 13.3: Linked List

We need locks for implementing linked lists because any operation (Add and Remove) involves two nodes. If we are doing lock-free implementation, there may be instances when there is an Add operation going on and someone removes previous node. For example, there is an add operation going on for *Node 3* and someone removes *Node 0*, at this point *Node 3* is lost. To prevent this kind of scenario, we need locks. One more

thing about Locked implementation is, it's not sufficient to just lock one element due to possibility of above mentioned scenario. So, we need locks on two elements.

Coarse grained linked list is implemented using usual monitor locks.

Fine grained - As discussed above, fine grained locking requires atleast two locks on successive elements for consistent operations.

Following techniques are used for efficient implementation of Linked Lists -:

- ***Hand Over Hand Locking*** - The idea is pretty simple. Instead of having a single lock for the entire list, you instead add a lock per node of the list. When traversing the list, the code first grabs the next nodes lock and then releases the current nodes lock (which inspires the name hand-over-hand).
- ***Lazy Deletion*** - A delete bit is placed on each node which is set to 1 before attempting the deletion of node. This enables other operations to know that this node is going to be deleted. The updating of this bit is the linearization point. Logical deletion is happening here before Physical deletion.
- ***Validate*** - Before an operation, the delete bit on nodes of importance for that operation should be checked.

NOTE - An important thing to remember in Linked List insert() operation is to do op(2) before op(1). The reason being, if op(1) happens before op(2), it can lead to a misconception of a broken list to a concurrent contains() operation.

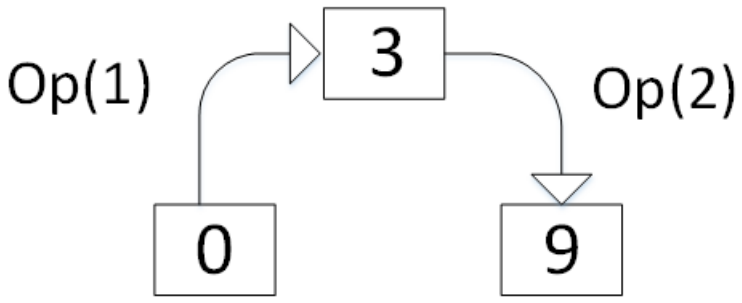


Figure 13.4: Linked List

References

[1] Vijay K Garg, Introduction to Multicore Computing

[2] Mark Moir and Nir Shavit, Concurrent Data Structures