CS766: Analysis of concurrent programs 2022

Lecture 5: Concurrent objects

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2022-02-18

Topic 5.1

Concurrent objects

Conurrent libraries

- We often do not write low-level concurrent code ourselves
- Concurrency is usually handled by concurrent libraries
- We only call the functions of the libraries; library code negotiates concurrent access.
- They provide interface to abstract concurrent objects such as : stacks, queues, sets, etc

Concurrent objects

Definition 5.1

A concurrent object is a data structure that can be modified by multiple threads and provides an interface with certain guarantees.

Example 5.1

A concurrent queue can store a collection of things. Threads can call the following functions.

- *enq(e) – adds element e in the collection*
- $e = deq()$ *– removes element e in the collection*

The elements enter/leave the collection in FIFO order.

Example: LockQueue

```
int head = 0, tail 0;                    // always increasing
Object items[CAP];                       // some size
Lock lock;                               // guarded by lock

Object deq() {
  l.lock();
  if (tail == head) {                     // Queue is empty
    l.unlock();
    throw Empty;
  }
  x = items[head % CAP];                  // pick from store
  head++;                                 // remove an element
  l.unlock();                             // release lock
  return x;
}
```

LockQueue: enq

```
void enq( Object x ) {
  l.lock();
  if (tail-head == CAP) {        // Queue is full
    l.unlock();
    throw Full;
  }
  items[tail % CAP] = x;         // place in store
  tail++;                        // insert element
  l.unlock();                    // release lock
}
```

Why Lockqueue is a queue?

- Due to lock, one thread accesses data at a time
- We can intuitively see it is a queue. We will see the formal definition.

Let us make things complicated. Let us drop Locks!!!

$$\text{Locks} \Rightarrow \text{waiting}$$

Example: LockFreeQueue

```
int head = 0, tail 0;                      // always increasing
Object items[CAP];                         // some size

Object deq() {
  if(tail == head) throw Empty;            // Queue is empty
  x = items[head % CAP];head++;            // remove an element
  return x;
}

void enq( Object x ) {
  if (tail-head == CAP) throw Full; // Queue is full
  items[tail % CAP] = x;tail++;            // insert element
}
```

Is the above a queue?

No exclusive access

LockFreeQueue: single enq and single deq

In general, LockFreeQueue may not be a queue.

Example 5.2

A bad interleaving between two concurrent enq.

```
enq1: items[0] = x1;
enq2: items[0] = x2;
enq1: tail = 1
enq2: tail = 1
```

Let us consider a scenario, where there is one thread for enqueue and one for dequeue.

Is LockFreeQueue a queue in the restrictive setting?

LockFreeQueue is a queue

Assume exception were not thrown.

The following proves invariants at various program locations.

$$\{0 \le \texttt{tail} - \texttt{head} \le \texttt{CAP}\}$$

$$
\begin{array}{lcl}
\begin{array}{l}
\mathit{Loop} : \cdots = \mathit{deq}() \\
\qquad \{0 \le \texttt{tail} - \texttt{head} \le \texttt{CAP}\} \\
\quad 0 : \texttt{if}(\texttt{head} == \texttt{tail}) \\
\qquad \{P : 0 < \texttt{tail} - \texttt{head} \le \texttt{CAP}\} \\
\quad 1 : \texttt{x} = \texttt{items}[\texttt{head}\%\texttt{CAP}]; \\
\qquad \{0 < \texttt{tail} - \texttt{head} \le \texttt{CAP}\} \\
\quad 2 : \texttt{head}++; \\
\qquad \{0 \le \texttt{tail} - \texttt{head} \le \texttt{CAP}\}
\end{array}
& \| &
\begin{array}{l}
\mathit{Loop} : \mathit{enq}(x) \\
\qquad \{0 \le \texttt{tail} - \texttt{head} \le \texttt{CAP}\} \\
\quad 0 : \texttt{if}(\texttt{tail} - \texttt{head} \ne \texttt{CAP}); \\
\qquad \{Q : 0 \le \texttt{tail} - \texttt{head} < \texttt{CAP}\} \\
\quad 1 : \texttt{items}[\texttt{tail}\%\texttt{CAP}] = \texttt{x}; \\
\qquad \{0 \le \texttt{tail} - \texttt{head} < \texttt{CAP}\} \\
\quad 2 : \texttt{tail}++; \\
\qquad \{0 \le \texttt{tail} - \texttt{head} \le \texttt{CAP}\}
\end{array}
\end{array}
$$

$$\{0 \le \texttt{tail} - \texttt{head} \le \texttt{CAP}\}$$

Exercise 5.1

Verify interference checks.

LockFreeQueue is a queue

If both the threads about to execute updates on `items`.

$$\begin{array}{c} \{P : 0 < \texttt{tail} - \texttt{head} \leq \texttt{CAP}\} \\ 1 : \texttt{x} = \texttt{items}[\texttt{head}\%\texttt{CAP}]; \end{array} \quad || \quad \begin{array}{c} \{Q : 0 \leq \texttt{tail} - \texttt{head} < \texttt{CAP}\} \\ 1 : \texttt{items}[\texttt{tail}\%\texttt{CAP}] = \texttt{x}; \end{array}$$

Both the invariants have to be true, i.e., $P \wedge Q = 0 < \texttt{tail} - \texttt{head} < \texttt{CAP}$

Therefore, $\texttt{tail}\%\texttt{CAP} \neq \texttt{head}\%\texttt{CAP}$

Therefore, they do not have race condition over elements of `items`.

Therefore, the *enq*() and *deq*() will always behave as if they run one after another.

How do we decide who ran first?

LockFreeQueue : who ran first?

Usually writes are commit points, where a call to the interface says to the world that it is done.

$$\begin{array}{lcl} \{0 < \texttt{tail} - \texttt{head} \leq \texttt{CAP}\} & & \{0 \leq \texttt{tail} - \texttt{head} < \texttt{CAP}\} \\ 2 : \texttt{head}++; & \| & 2 : \texttt{tail}++; \end{array}$$

The thread that executed their update statement first ran first.

Why these writes are the commit point?

We need to look at LockFreeQueue more precisely.

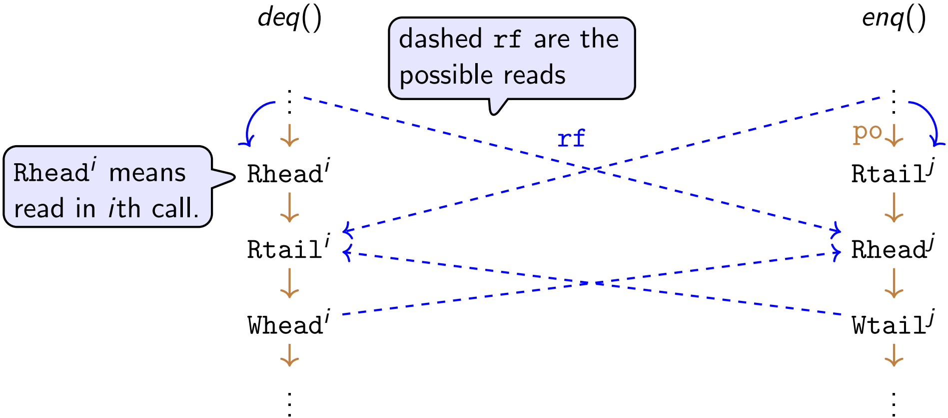
Machine accurate LockFreeQueue

```
int head = 0, tail = 0; Object items[CAP];

Object deq() {
  l_tail = tail;
  l_head = head;
  if(l_tail == l_head) throw Empty;
  x = items[l_head % CAP];
  head = l_head + 1;
  return x;
}

void enq( Object x ) {
  l_tail = tail;
  l_head = head;
  if (l_tail-l_head == CAP) throw Full;
  items[l_tail % CAP] = x;
  tail = l_tail + 1;
}
```

Data is first copied to locals then used.
At the end result is copied to globals

Read write pattern in LockFreeQueue

Two reads and a write in each call.



If $\texttt{Whead}^i$ and $\texttt{Wtail}^j$ are ordered according to $(\texttt{po} \cup \texttt{rf} \cup \texttt{ws} \cup \texttt{fr})^+$ then then we run them in same order in sequential version. Otherwise, we can choose any order.

Proof technique (vague overview)

- In an algorithm, identify data and control variables.
  - Control variables control the flow of execution and decide when to update data
  - Data simply stores the data
- Find invariants at all locations that show that there is no race over data variables.
- Show that in all executions we can linearize: identify points where each thread committed.
- Compare the linearized execution with a reference implementation.

Topic 5.2

Michel & scott queue

Example: Michel & scott queue

- Let us look at a real lock-free queue
- Data is stored at linked list
- `tail` is lazily updated
- Needs help of CAS instruction from hardware

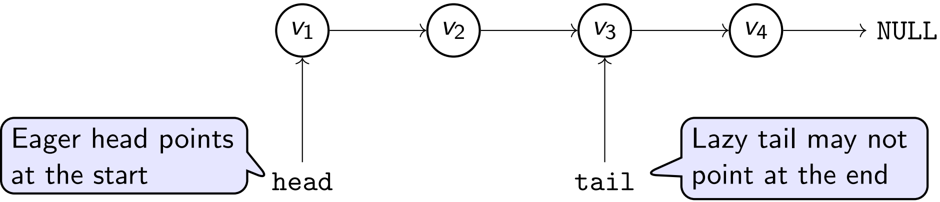
Compare-and-swap(CAS)

CAS instruction is atomic and has the following effect.

```
bool CAS( p, old_val, new_val) {
  if( *p == old_val ) {
    *p = new_val;
    return true;    // update and return true
  }
  return false;     // do nothing and return false
}
```

Michel & scott queue: initialization

```
struct Node = { Object data, Node* next; }

Node* head, tail;

initialize() {
  node = new Node()  // Allocate a free node
  node->next = NULL  // Make it the only node in the list
  head = tail = node // Both Head and Tail point to it
}
```



Michel & scott enqueue

```
enq(Object x){
  node = new node(); node->data = x; node->next = NULL;//Allocate

  while(1) {                                    //try until done
    l_tail = tail;l_next = l_tail->next        //read queue state

    if(l_next == NULL) {                        //really a tail?
      if( CAS(&(l_tail->next), l_next, node) ) {//try to insert
        CAS( &tail, l_tail, node);               //may return false
        return;
      }
    }else{ // tail is not at the end. ODD!!  :-(
      CAS( &tail, l_tail, l_next)               //correcting failure
    }
  }
}                                       // Enqueue done.
```

Michel & scott dequeue

```
Object dequeue(){
  while(1){                              // try until done
    l_head = head
    l_tail = tail
    l_next = l_head->next                // read queue state

    if(l_head != l_tail) {
       v = l_next->value // Read value before CAS (why?)
       if( CAS(&head, l_head, next) ) return v;//dequeue done
    }else{
      if( l_next == NULL) return NULL; // queue is empty
      CAS(&tail, l_tail, l_next) // Try to advance tail
    }
  }
}
```

Topic 5.3

UnboundedQueue

Example: UnboundedQueue

Here is an another concurrent implementation of queue

```
Vector q;

void* enq(int x) { // x > 0
  q.push_back(x)
}

int deq() {
  while( true ) {
    l = q.length()
    for( i = 0 ; i < l; i ++ ) {
      atomic{ x=q[i]; q[i]=0; }                // atomic
      if ( x != 0) return x;
    }
  }
}
```

Is the above a queue?

End of Lecture 5