CSCI-UA.0201

Computer Systems Organization

Concurrency – Correctness of Concurrent Objects

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies
Concurrent Computation

memory

object

object
Objectivism

• What is a concurrent object?
  – How do we **describe** one?
  – How do we **implement** one?
  – How do we **tell if it is correct**?
Objectivism

• What is a concurrent object?
  – How do we describe one?

  – How do we tell if it is correct?
FIFO Queue: Enqueue Method

enq(q, o)
FIFO Queue: Enqueue Method

$\text{enq}(q, o)$
FIFO Queue: Dequeue Method

deq(q) /
FIFO Queue: Dequeue Method

\texttt{deq(q) / \bigcirc}
Lock-Based Queue

CAPACITY = 8
Lock-Based Queue

Fields protected by single shared lock

CAPACITY = 8
A Lock-Based Queue

typedef struct {
    int head, tail;
    void* items[CAPACITY];
    phread_mutex_t lock;
} queue_t;

Fields protected by single shared lock
Lock-Based Queue

Initially head = tail
Lock-Based deq()
int deq(queue_t q, void **elem) {
    int res;
    pthread_mutex_lock(&q->lock);
    if (q->tail == q->head) res = 0;
    else {
        *elem = q->items[q->head % CAPACITY];
        q->head++;
        res = 1;
    }
    pthread_mutex_unlock(&q->lock);
    return res;
}
Acquire Lock

My turn ...

Waiting to enqueue…
int deq(queue_t q, void **elem) {
    int res;

    pthread_mutex_lock(&q->lock);
    if (q->tail == q->head) res = 0;
    else {
        *elem = q->items[q->head % CAPACITY];
        q->head++;
        res = 1;
    }
    pthread_mutex_unlock(&q->lock);
    return res;
}

Implementation: deq()
Check if Non-Empty

Waiting to enqueue…
Implementation: `deq()`

```c
int deq(queue_t q, void **elem) {
    int res;
pthread_mutex_lock(&q->lock);
    if (q->tail == q->head) res = 0;
    else {
        *elem = q->items[q->head % CAPACITY];
        q->head++;
        res = 1;
    }
pthread_mutex_unlock(&q->lock);
    return res;
}
```

If queue empty return "failure"
Modify the Queue

head

0

1

x

y

2

3

7

6

5

4

tail

Waiting to enqueue…
Modify the Queue

Waiting to enqueue...
Modify the Queue

Waiting to enqueue…
Implementation: `deq()`

```c
int deq(queue_t q, void **elem) {
    int res;
    pthread_mutex_lock(&q->lock);
    if (q->tail == q->head) res = 0;
    else {
        *elem = q->items[q->head % CAPACITY];
        q->head++;
        res = 1;
    }
    pthread_mutex_unlock(&q->lock);
    return res;
}
```

Queue not empty?
Remove item and update head
Release the Lock

head

0

1

tail

My turn!

0 1 2 3 4 5 6 7

x
Implementation: `deq()`

```c
int deq(queue_t q, void **elem) {
    int res;
    pthread_mutex_lock(&q->lock);
    if (q->tail == q->head) res = 0;
    else {
        *elem = q->items[q->head % CAPACITY];
        q->head++;
        res = 1;
    }
    pthread_mutex_unlock(&q->lock);
    return res;
}
```

Release lock no matter what!
Implementation: \texttt{deq( )}

```c
int deq(queue_t q, void **elem) {
  int res;
  pthread_mutex_lock(&q->lock);
  if (q->tail == q->head) res = 0;
  else {
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    res = 1;
  }
  pthread_mutex_unlock(&q->lock);
  return res;
}
```
Implementation: \texttt{deq()}

```c
int deq(queue_t q, void **elem) {
    int res;
    pthread_mutex_lock(&q->lock);
    if (q->tail == q->head) res = 0;
    else {
        *elem = q->items[q->head % CAPACITY];
        q->head++;
        res = 1;
    }
    pthread_mutex_unlock(&q->lock);
    return res;
}
```

Should be correct because modifications are mutually exclusive...
Now consider the following implementation

- The same thing without mutual exclusion
- For simplicity, only two threads
  - One thread enq only
  - The other deq only
Wait-free 2-Thread Queue

CAPACITY = 8
Wait-free 2-Thread Queue
Wait-free 2-Thread Queue

head

0 1

x y

7

2
tail

6

5 4

3

z
Wait-free 2-Thread Queue

elem* = items[head]
Wait-free 2-Thread Queue

elem* = items[head]

items[tail] = z
Wait-free 2-Thread Queue
Wait-free 2-Thread Queue
Wait-free 2-Thread Queue

```c
int deq(queue_t q, void **elem) {
    if (q->tail == q->head) return 0;
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    return 1;
}

int enq(queue_t q, void *x) {
    if (tail-head == CAPACITY) return 0;
    q->items[q->tail % CAPACITY] = x;
    q->tail++;
    return 1;
}
```

No lock needed !
int deq(queue_t q, void **elem) {
    if (q->tail == q->head) return 0;
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    return 1;
}

int enq(queue_t q, void *x) {
    if (tail-head == CAPACITY) return 0;
    q->items[q->tail % CAPACITY] = x;
    q->tail++;
    return 1;
}
Wait-free 2-Thread Queue

int deq(queue_t q, void **elem) {
    if (q->tail == q->head) return 0;
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    return 1;
}

int enq(queue_t q, void *x) {
    if (tail-head == CAPACITY) return 0;
    q->items[q->tail % CAPACITY] = x;
    q->tail++;
    return 1;
}

How do we define “correct” when modifications are not mutually exclusive?
What is a Concurrent Queue?

• Need a way to specify a concurrent queue object
• Need a way to prove that an algorithm implements the object's specification
• Let's talk about object specifications ...
Correctness and Progress

• In a concurrent setting, we need to specify both the safety and the liveness properties of an object
• Need a way to define
  – when an implementation is correct
  – the conditions under which it guarantees progress
Correctness and Progress

• In a concurrent setting, we need to specify both the safety and the liveness properties of an object

• Need a way to define
  – when an implementation is correct
  – the conditions under which it guarantees progress

Let's begin with correctness
Sequential Objects

• Each object has a *state*
  – Usually given by a set of *fields*
  – Queue example: items, head, tail

• Each object has a set of *methods*
  – Only way to manipulate state
  – Queue example: `enq` and `deq` methods
Sequential Specifications

• If (precondition)
  – the object is in such-and-such a state
  – before you call the method,

• Then (postcondition)
  – the object will be in some other state
  – and the method will return a particular value
Pre and Postconditions for Dequeue

• Precondition:
  – Queue is non-empty

• Postcondition:
  – Returns 1

• Postcondition:
  – Removes first item in queue
Pre and Postconditions for Dequeue

• Precondition:
  – Queue is empty

• Postcondition:
  – Returns 0

• Postcondition:
  – Queue state unchanged
Why Sequential Specifications Totally Rock

• Interactions among methods captured by side-effects on object state
  – State meaningful between method calls

• Documentation size linear in number of methods
  – Each method described in isolation

• Can add new methods
  – Without changing descriptions of old methods
What About Concurrent Specifications?

- Methods?
- Documentation?
- Adding new methods?
Methods Take Time
Methods Take Time

Invocation 12:00

enq(q, ...)

time
Methods Take Time

invocation 12:00

enq(q, o)

Method call

time
Methods Take Time

invocation 12:00

enq(q, \( \_ \))

Method call

time
Methods Take Time

Invocation: 12:00

Response: 12:01

enq(q, ...)
Sequential vs Concurrent

• Sequential
  – Methods take time? Who knew?

• Concurrent
  – Method call is not an event
  – Method call is an interval.
Concurrent Methods Take **Overlapping** Time
Concurrent Methods Take **Overlapping Time**
Concurrent Methods Take Overlapping Time
Concurrent Methods Take Overlapping Time
Sequential vs Concurrent

• Sequential:
  – Object needs meaningful state only *between* method calls

• Concurrent
  – Because method calls overlap, object might *never* be between method calls
Sequential vs Concurrent

• Sequential:
  – Each method described in isolation

• Concurrent
  – Must characterize all possible interactions with concurrent calls
    • What if two enqs overlap?
    • Two deqs? enq and deq? …
Sequential vs Concurrent

• Sequential:
  – Can add new methods without affecting older methods

• Concurrent:
  – Everything can potentially interact with everything else
Sequential vs Concurrent

• Sequential:
  – Can add new methods without affecting older methods

• Concurrent:
  – Everything can potentially interact with everything else
The Big Question

• What does it mean for a concurrent object to be correct?
  – What is a concurrent FIFO queue?
  – FIFO means strict temporal order
  – Concurrent means ambiguous temporal order
int deq(queue_t q, void **elem) {
    int res;
    pthread_mutex_lock(&q->lock);
    if (q->tail == q->head) res = 0;
    else {
        *elem = q->items[q->head % CAPACITY];
        q->head++;
        res = 1;
    }
    pthread_mutex_unlock(&q->lock);
    return res;
}
int deq(queue_t q, void **elem) {
    int res;
    pthread_mutex_lock(&q->lock);
    if (q->tail == q->head) res = 0;
    else {
        *elem = q->items[q->head % CAPACITY];
        q->head++;
        res = 1;
    }
    pthread_mutex_unlock(&q->lock);
    return res;
}
Intuitively

\[
deq(q)
\]

\[
enq(q)
\]

\[
time
\]
Intuitively

lock() deq(q)

enq(q)

lock() unlock()
Intuitively
Intuitively
Intuitively

lock()  \text{deq}(q)  unlock()

\text{enq}(q)  \text{lock()}  \text{enq}  \text{unlock}()  \text{deq}
Intuitively

Behavior is “Sequential”
Let intuitively:

Let's capture the idea of describing the concurrent via the sequential.

Behavior is “Sequential”
Linearizability

• Each method should
  – “take effect”
  – instantaneously
  – between invocation and response events
• Object is correct if this “sequential” behavior is correct
• Any such concurrent object is called
  – Linearizable
Is it really about the object?

• Each method should
  – “take effect”
  – instantaneously
  – between invocation and response events

• Sounds like a property of an execution...

• A linearizable object: one all of whose possible executions are linearizable
Example

time

[Diagram with three-colored circles and an arrow labeled 'time']
Example

\text{enq}(q,x)
Example

\[ \text{enq}(q, x) \quad \text{enq}(q, y) \]

\[ \square \quad \square \quad \square \]

\text{time}
Example
Example

enq(q,x)  enq(q,y)  deq(q,x)  deq(q,y)

time
Example
Example

enq(q,x)
enq(q,y)
deq(q,x)
deq(q,y)

Valid?
Example
Example

\[ \text{enq}(q, x) \]

\[ \text{time} \]
Example

enq(q,x)    deq(q,y)

time
Example

\[ \text{enq}(q,x) \quad \text{deq}(q,y) \quad \text{enq}(q,y) \]

\text{time}
Example

\[
\begin{align*}
\text{enq}(q, x) & \quad \text{enq}(q, y) \\
\text{deq}(q, y) & \quad \text{enq}(q, y) \\
\end{align*}
\]
Example

\[
\begin{align*}
&\text{enq}(q,x) \\
&\text{deq}(q,y) \\
&\text{enq}(q,x) \\
&\text{enq}(q,y)
\end{align*}
\]
Example

\[
\begin{align*}
\text{enq}(q, x) \\
\text{enq}(q, y) \\
\text{deq}(q, y) \\
\text{enq}(q, y)
\end{align*}
\]
Example

\[\begin{array}{c|c|c}
\text{Blue} & \text{Red} & \text{Blue} \\
\end{array}\]

time
Example

\[ \text{enq}(q, x) \]

\[ \text{time} \]
Example

\text{enq}(q,x)

\text{deq}(q,x)

\text{time}
Example

\[ \text{enq}(q, x) \]

\[ \text{deq}(q, x) \]
Example

\( \text{enq}(q, x) \)

\( \text{deq}(q, x) \)

linearizable
Example

\text{enq}(q,x)
Example

\[ \text{enq}(q,x) \]

\[ \text{enq}(q,y) \]

Time
Example

\[ enq(q,x) \]
\[ enq(q,y) \]
\[ deq(q,y) \]

---

**Example**
Example

enq(q, x)

enq(q, y)

deq(q, y)

deq(q, x)

time
Comme ci  Example

deq(q,y)  deq(q,x)

enq(q,y)  enq(q,x)
Comme ci  
Comme ça  

Example

\[ \text{enq}(q,x) \quad \text{enq}(q,y) \]
\[ \text{deq}(q,y) \quad \text{deq}(q,x) \]

\[ \text{time} \]
Comme ci
Comme ça

Example

multiple orders OK
linearizable

tenq(q,x)
deq(q,y)
tenq(q,y)
deq(q,x)

79
Talking About Executions

• Why executions?
  – Can't we specify the linearization point of each operation without describing an execution?

• Not Always
  – In some cases, linearization point depends on the execution
Linearizable Objects are Composable

- Modularity
- Can prove linearizability of objects in isolation
- Can compose independently-implemented objects
Reasoning About Linearizability: Locking

```c
int deq(queue_t q, void **elem) {
    int res;
    pthread_mutex_lock(&q->lock);
    if (q->tail == q->head) res = 0;
    else {
        *elem = q->items[q->head % CAPACITY];
        q->head++;
        res = 1;
    }
    pthread_mutex_unlock(&q->lock);
    return res;
}
```
Reasoning About Linearizability: Locking

```c
int deq(queue_t q, void **elem) {
    int res;
    pthread_mutex_lock(&q->lock);
    if (q->tail == q->head) res = 0;
    else {
        *elem = q->items[q->head % CAPACITY];
        q->head++;
        res = 1;
    }
    pthread_mutex_unlock(&q->lock);
    return res;
}
```

Linearization points are when locks are released
More Reasoning: Wait-free

```c
int deq(queue_t q, void **elem) {
    if (q->tail == q->head) return 0;
    *elem = q->items[q->head % CAPACITY];
    q->head++;
    return 1;
}

int enq(queue_t q, void *x) {
    if (tail-head == CAPACITY) return 0;
    q->items[q->tail % CAPACITY] = x;
    q->tail++;
    return 1;
}
```
More Reasoning: Wait-free

```c
int deq(queue_t q, void **elem) {
    if (q->tail == q->head)
        *elem = q->items[q->head % CAPACITY];
    q->head++;
    return 1;
}

int enq(queue_t q, void *x) {
    if (q->tail == q->head)
        q->items[q->tail % CAPACITY] = x;
    q->tail++;
    return 1;
}
```

Linearization order is order head and tail fields modified.
More Reasoning: Wait-free

int deq(queue_t q, void **elem) {
    if (q->tail == q->head)
        *elem = q->items[q->head % CAPACITY];
    q->head++;
    return 1;
}

int enq(queue_t q, void *x) {
    if (tail-head == CAPACITY) return 0;
    q->items[q->tail % CAPACITY] = x;
    q->tail++;
    return 1;
}

Linearization order is order head and tail fields modified

Remember that there is only one enqueuer and only one dequeuer
Strategy

• Identify one atomic step where method “happens”
  – Critical section
  – Machine instruction

• Doesn't always work
  – Might need to define several different steps for a given method
Linearizability: Summary

- Powerful specification tool for shared objects
- Allows us to capture the notion of objects being “atomic”
- Don't leave home without it
Progress

• We saw an implementation whose methods were lock-based (deadlock-free)
• We saw an implementation whose methods did not use locks (lock-free)
• How do they relate?
Progress Conditions

• *Deadlock-free*: *some* thread trying to acquire the lock eventually succeeds.

• *Starvation-free*: *every* thread trying to acquire the lock eventually succeeds.

• *Lock-free*: *some* thread calling a method eventually returns.

• *Wait-free*: *every* thread calling a method eventually returns.
## Progress Conditions

<table>
<thead>
<tr>
<th>Everyone makes progress</th>
<th>Non-Blocking</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wait-free</td>
<td>Starvation-free</td>
</tr>
<tr>
<td>Someone makes progress</td>
<td>Lock-free</td>
<td>Deadlock-free</td>
</tr>
</tbody>
</table>

- **Wait-free**
- **Starvation-free**
- **Lock-free**
- **Deadlock-free**