CONCURRENT/DISTRIBUTED DATA TYPES
Correctness Criteria, Verification
(Sequential) Data Types

Abstractions to simplify the manipulation of high-quantity data: objects (instances) + operations

- **Queue** = \texttt{enqueue}(value) + \texttt{dequeue}() => value
- Stack, Set, Key-value map, ...

Specifications of data types:

- API documentation

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Methods</strong></td>
</tr>
<tr>
<td><strong>Modifier and Type</strong></td>
</tr>
<tr>
<td>boolean</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>boolean</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
(Sequential) Data Types

Abstractions to simplify the manipulation of high-quantity data: objects (instances) + operations

- Queue = enqueue(value) + dequeue() => value
- Stack, Set, Key-value map, ...

Specifications of data types:

- API documentation
- Pre/Post conditions in Hoare logic (Abstract Data Types):
  - { Seq } enqueue(value) { Seq :: value }
  - { value :: Seq } dequeue() => value { Seq }
Concurrent Data Types

Basic blocks of software that needs to process data in parallel

**Concurrent Data Types:** operations can be invoked in parallel from different threads or sites in a network

Support high-frequency parallel accesses to high-quantity data

Deployed over a shared-memory or a network

Formal specification and verification?
Shared-State in Parallel Applications

- Parallelizing applications for efficiency

**Sequential**

```java
// reading data for future processing
q = new Queue();
while(...){
    X = readFile();
    q.enqueue(X);
}
```
Shared-State in Parallel Applications

- Parallelizing applications for efficiency

**Sequential**

```java
// reading data for future processing
q = new Queue();
while(...){
    X = readFile();
    q.enqueue(X);
}
```

**Parallel**

```java
q = new Queue();
while(...){
    X = readFile1();
    q.enqueue(X);
}
while(...){
    X = readFile2();
    q.enqueue(X);
}
```

- multi-threading
- distributed over a network
Shared-State in Parallel Applications

- Parallelizing applications for efficiency

### Sequential

```java
// reading data for future processing
q = new Queue();
while(...) {
    X = readFile();
    q.enqueue(X);
}
```

### Parallel

```java
q = new Queue();
l = new Lock();
while(...) {
    X = readFile1();
    l.lock();
    q.enqueue(X);
    l.unlock();
}
while(...) {
    X = readFile2();
    l.lock();
    q.enqueue(X);
    l.unlock();
}
```

- multi-threading
- distributed over a network
Parallelizing applications for efficiency

### Sequential

```java
q = new Queue();
while(...){
    X = readFile();
    q.enqueue(X);
}
```

### Parallel

```java
q = new Queue();
l = new Lock();
while(...){
    X = readFile1();
    l.lock();
    q.enqueue(X);
    l.unlock();
}
while(...){
    X = readFile2();
    l.lock();
    q.enqueue(X);
    l.unlock();
}
```

How to implement concurrent objects?
How is correctness defined? Verification?
Concurrent Objects

Multi-threaded programming

e.g. Java Development Kit SE

dozens of objects, including queues, maps, sets, lists, locks, atomic integers, …
Lock-Free Implementations

blocking reference implementation

enq: 1
(deq: 1, deq: 2)

(mutual exclusion)

enq: 2

(blocked)

efficient nonblocking implementation

enq: 1
(deq: 2, deq: 1)

atomic instructions, e.g., compare-and-swap
Replicated Objects (NoSQL)

To support failures, the state of the object is **replicated**

For availability, replicas may store different versions: **weak consistency**

**CAP** theorem: No replicated object is strongly **C**onsistent, highly-**A**vailable, and **P**artition-tolerant

**Instances**: key-value stores (Amazon Dynamo, Cosmos DB), CRDTs
Replicated objects

Distributed systems

Conflicting concurrent updates: how are they observed on different replicas?

Adversarial environments: crashes, network partitions
Pessimistic Replication

Using consensus algorithms to agree on an order between conflicting concurrent updates

- **Strong Consistency**: Yes
- **Availability**: No

CAP theorem [Gilbert et al.’02]: strong consistency + availability + partition tolerance is impossible
Optimistic Replication

Each update is applied on the local replica and propagated asynchronously to other replicas.

- strong consistency (X)
- availability (✓)

Replicas may store different versions of data: weak consistency
Verification Ingredients

- Specifying a Library: $\phi$
- Implementing a Library: $\bot$
- Verifying a Library implementation: $\bot \models \phi$
Specifying Sequential Objects
Object Specification

- How can we specify an object? (Library)
  - Objects API
  - Use cases
  - Pre and Post Conditions?
- What are the behaviors of a client using the library?
💡 for any client making library calls record the inputs and outputs of each call
What is a client?

- What is a client of the Library?
  - Program that issues calls to a library instance

```
// do something
q.enqueue(v)
// do something
x = q.dequeue()
// ...
```

How do we specify a Data Structure (DS) generically?

- Histories of calls and returns
- Constraint possible return values
Well Encapsulated Objects

- *Global* object state:
  - Possibly *local* thread state
  - A set of *operations* or *methods*
    - Input and output types
    - Methods are the only way to operate on the state
Sequential Object Specifications

- Library $L = \langle \Sigma, m_1, m_2, m_3 \rangle$

- Client C: Issues calls to the library methods
  - (Sequential) Most General Client [SMGC]

- We will talk about histories of calls with values
  - $\epsilon$ denotes the empty sequence,
  - $o$ denotes an operation (eg. $\langle \text{pop}(), v \rangle$), and
  - $\delta$ denotes a sequence of operations

```
SMGC(L):
    while true do
        $m_i = \text{choseMethodFrom}(L)$;
        $\text{args} = \text{choseInputsFor}(m)$;
        $m_i(\text{args})$;
    od
```
specifying a register

- Inductive histories of a register:
  1. $\epsilon$ is a register history (RH)
  2. $<\text{read}(), 0>*$ is a register history
  3. if $\delta$ is a RH, then so is $\delta \cdot <\text{write}(v), _>$
  4. if $\delta \cdot <\text{write}(v), _>$ is a RH, then so it is $\delta \cdot <\text{read}(), v>*$
Examples
Inductive histories of a Stack:

1. $\epsilon$ is a Stack History (SH)
2. if $\delta \cdot \langle \text{pop}(), v \rangle$ is a SH, then so is $\langle \text{push}(w), _ \rangle \cdot \delta$
3. if $\delta$ is a SH, and $|\{\langle \text{pop}(), v \rangle : \delta | v \neq \bot \}| = |\langle \text{push}(v), _ \rangle : \delta|$, then so it is $\delta \cdot \langle \text{pop}(), \bot \rangle^*$
4. same conditions as 3, and $\langle \text{pop}(), \bot \rangle$ does not occur in $\delta$ then, $\langle \text{push}(w), \bot \rangle \cdot \delta \cdot \langle \text{pop}(), w \rangle$ is a SH
5. if $\delta_0 \cdot \langle \text{pop}(), \bot \rangle$ is a SH, and $\delta_1$ is a SH, then $\delta_0 \cdot \delta_1$ is SH
Examples
Implementations

▶ A set implementation based on sorted linked lists

public class Entry {
    public Object value;
    public Entry next;
}

public class Set {
    Entry first;
    public boolean add(Object x) {...}
    public boolean remove(Object x) {...}
    public boolean contains(Object x) {...}
}
Implementations

- A set implementation based on sorted linked lists

Adding an Entry

Removing an Entry
Specifying Concurrent Objects
What about Concurrency?

while true do
    $m_1 = \text{choseMethodFrom}(L);$;
    $\text{args} = \text{choseInputsFor}(m);$;
    $m_1(\text{args});$
od

$s.\text{push}(v)$ return

while true do
    $m_1 = \text{choseMethodFrom}(L);$;
    $\text{args} = \text{choseInputsFor}(m);$;
    $m_1(\text{args});$
od

$s.\text{pop}()$ return $v$

Should this be legal?

Concurrent Consistency Criteria
Concurrent Clients

- Most General Client (seq)

- Most General Client (concurrent n threads)

- Concurrent Library Verification w.r.t. $\text{CMGC}_n(L)$ for any $n$
Concurrent Consistency Criteria

- Quiescence Consistency
- Sequential Consistency
- Linearizability

We will work with Registers to exemplify the definitions
Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order

```
r.write(1);
r.write(2);
r.read();
r.read();
```
Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order

- Method calls separated by a period of quiescence should appear to take effect in their real time order

\[ r\text{.write}(1) \quad \text{ret} \quad r\text{.read}() \quad \text{ret} 2 \]
\[ r\text{.write}(2) \quad \text{ret} \quad r\text{.read}() \quad \text{ret} 0 \]

✗

\[ <r\text{.write}(1),>_ \]
\[ <r\text{.read}(),2> \]

✗

\[ <r\text{.read}(),0> \]
\[ <r\text{.write}(2),>_ \]
Quiescent Consistency

- Method calls should appear to happen one-at-a-time, sequential order
- Method calls separated by a period of quiescence should appear to take effect in their real time order

\[
\text{r.write}(1) \quad \text{ret} \quad \text{r.write}(2) \quad \text{ret} \\
\text{r.read()} \quad \text{ret} \ 2 \quad \text{r.read()} \quad \text{ret} \ 0
\]
Sequential Consistency

- *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Computer Programs* [Lamport’79]
  - Each process issues operations in the order specified by its program.
  - Operations from all processors issued to a single object are serviced from a single FIFO queue. Issuing an operation consists in entering a request on this queue.
Sequential Consistency

r.write(1);  r.read();
r.write(2);  r.read();

✗
Sequential Consistency

```cpp
r.write(1); r.read();
r.write(2); r.read();
```

```
<r.write(1),_>  <r.write(2),_>
<r.read(),0>    <r.read(),1>
<r.read(),0>    <r.write(1),_>  <r.read(),1>  <r.write(2),_>
```
Sequential Consistency

- Quiescent Consistency +
- Method calls should appear to take effect in Program Order

```plaintext
r.write(1);
```

```plaintext
r.write(2);
```

```plaintext
r.read();
```

```plaintext
r.read();
```

- Each history $\delta$ induces a per-thread total order of operations
  - $o_1 \prec_\delta o_2$ iff $o_1$ and $o_2$ are on the same thread, and $o_1$ occurs before $o_2$ in $\delta$
- A history $\delta$ is Sequentially Consistent if there exists an equivalent *Sequential* history $\delta'$ (i.e. same operations), and
  - $o_1 \prec_\delta o_2$ implies $o_1 \prec_{\delta'} o_2$
Sequential Consistency

- x.write(1)  ret
- x.read()    ret
- x.write(2)  ret

- x.write(1)  ret
- x.read()    ret
- x.write(2)  ret

✔

✗
Linearizability

- Same conditions as Sequential Consistency +
- Each method call should appear to take effect instantaneously at some moment between its invocation (call) and response (return)
- That is: we can pretend that the execution of each method is uninterrupted by other calls to the object
- De-facto standard for Concurrent Object Correctness (e.g., java.util.concurrent)

Linearizability: A Correctness Condition for Concurrent Objects
[Herlihy and Wing ‘90]
Linearizability

- Each history $\delta$ induces a partial order on operations such that
  - $o_1 \sqsubseteq_\delta o_2$ iff $\text{ret } o_1$ occurs before $\text{call } o_2$ in $\delta$
- A history $\delta$ is Linearizable if there exists an equivalent Sequential history $\delta'$ (i.e. same operations), and
  - $o_1 \sqsubseteq_\delta o_2$ implies $o_1 \sqsubseteq_\delta' o_2$
- Ignoring uncompleted operations
- Strictly stronger than Sequential Consistency
Each operation takes place atomically within its call/return.
Linearizability

- Each operation takes place atomically within its call/return

\[ q.\text{enq}(v) \]

\[ q.\text{deq}, w \]

\[ q.\text{enq}(w) \]

Not Linearizable
Linearizability

- Each operation takes place atomically within its call/return
Each operation takes place atomically within its call/return
Linearizability vs. Sequential Consistency

\[ \text{x.write}(1) \quad \text{ret} \quad \quad \text{x.read}(1) \quad \text{ret} \]

\[ \quad \text{x.write}(2) \quad \text{ret} \]

Not linearizable to begin with!
Observational Refinement

- Linearizability => observational refinement

Reference implementation

```c
class AtomicStack {
    cell* top;
    Lock l;

    void push (int v) {
        l.lock();
        top->next = malloc(sizeof *x);
        top = top->next;
        top->data = v;
        l.unlock();
    }

    int pop () {
        ...
    }
}
```

Efficient implementation

```c
class TreiberStack {
    cell* top;

    void push (int v) {
        cell* t;
        cell* x = malloc(sizeof *x);
        x->data = v;
        do {
            t = top;
            x->next = top;
        } while (!CAS(&top,t,x));

        int pop () {
            ...
        }
    }
}
```

For every Client, Client x Impl included in Client x Spec
Linearizability: Compositionallity

- **Theorem**: A history $\mathcal{h}$ is linearizable if and only if for each object $o$ in $\mathcal{h}$, $\mathcal{h}_o$ is linearizable.

  **Proof**: Simple induction on the number of operations appearing in $\mathcal{h}$.

- **Corollary**: It is enough to show that each Library is linearizable to know that the system is
Some Object Implementations
Set Implementations
**Hand-over-Hand Set**

- A set implementation based on sorted linked lists

```java
public class Entry {
    public Object value;
    public Entry next;
}

public class Set {
    Entry first;
    public boolean add(Object x) {...}
    public boolean remove(Object x) {...}
    public boolean contains(Object x) {...}
}
```
Hand-over-Hand Set

Adding an entry

Removing an entry

- Coarse-grain vs Fine-grain locking
- Efficient solution: one lock per list node
- What nodes to lock during an insertion/removal? The predecessor/successor?
Hand-over-Hand Set

adding entry c:
Hand-over-Hand Set

adding entry c:
Hand-over-Hand Set

removing entry b:
Hand-over-Hand Set

removing entry b:
Hand-over-Hand Set

- Can we acquire locks only when reaching the modification place?
- adding entry c: advance until reaching (b,d) and then lock
Hand-over-Hand Set
“Basic” Objects
int Lock = 0;
TID owner = null;

void lock(){
    bool l;
    do {
        while(Lock == 1);
        l = cas(Lock, 0, 1);
    } until (l);
    owner = getTID();
    return;
}

void unlock(){
    owner = null;
    Lock = 0;
    return;
}
class IntPtr {
    int val;
}
IntPtr COU;

void inc(int v){
    int n;
    while(true) {
        n = COU->val;
        if (cas(COU->val, n, n+v))
            break;
    }
    return;
}

void dec(int v) {
    int n;
    while(true) {
        n = COU->val;
        if (cas(COU->val, n, n-v))
            break;
    }
    return;
}

int read() {
    return COU->val;
}
Stack Implementations
Treiber Stack

class Node {
    Node tl;
    int val;
}

class NodePtr {
    Node val;
    int val;
} TOP;

void push(int e) {
    Node y, n;
    y = new();
    y->val = e;
    while(true) {
        n = TOP->val;
        y->tl = n;
        if (cas(TOP->val, n, y))
            break;
    }
}

int pop() {
    Node y, z;
    while(true) {
        y = TOP->val;
        if (y==0) return EMPTY;
        z = y->tl;
        if (cas(TOP->val, y, z))
            break;
    }
    return y->val;
}

Systems Programming: Coping with Parallelism
[Treiber’86]
Treiber Stack (ABA bug)

pushed: 1, 2, 3  
popped: 1, 3, EMPTY  

PROBLEM  
not admitted by atomic stack
### HSY Elimination Stack

Extremely simplified version: 1 collision

```java
class Node {
    Node tl;
    int val;
}

class NodePtr {
    Node val;
} TOP;

class TidPtr {
    int clash;
}

// void push(int e) {
Node y, n;
TID hisId;
y = new();
y->val = e;

while (true) {
    n = TOP->val;
y->tl = n;
    if (cas(TOP->val, n, y))
        return;
    //elimination scheme
    TidPtr t = new TidPtr();
t->val = e;
    if (cas(clash, null, t)){
        wait(DELAY);
        //not eliminated
        if (cas(clash, t, null))
            continue;
        else break; //eliminated
    }
}
}

// int pop() {
Node y,z;
int t;
TID hisId;
while (true) {
    y = TOP->val;
    if (y == 0)
        return EMPTY;
    z = y->tl;
    t = y->val;
    if (cas(TOP->val, y, z)
        return t;
    //elimination scheme
    pusher = clash;
    while (pusher!=null){
        if (cas(clash,pusher,null))
            wait(DELAY);
        //not eliminated
        if (cas(clash, pusher, null))
            continue;
        else break; //eliminated
    }
    return pusher->val;
}
```

[Hendler et al.'04]
Queue Implementations
Two Locks Queue

```cpp
class Node {
    int val;
    Node tl;
}

class Queue {
    Node head;
    Node tail;
    thread_id hlock;
    thread_id tlock;
} Q;

void enqueue(int v) {
    Node n, t;
    n = new();
    n->val = v;
    n->tl = NULL;
    lock (&Q->tlock);
    temp = Q->tail;
    temp->tl = node;
    Q->tail = node;
    unlock (&Q->tlock);
}

int dequeue() {
    Node n, new_h;
    int v;
    lock (&Q->hlock);
    n = Q->head;
    new_h = n->tl;
    if (new_h == NULL) {
        unlock (&Q->hlock);
        return EMPTY;
    } else {
        value = new_h->val;
        Q->head = new_h;
        unlock (&Q->tlock);
        //dispose(n);
        return v;
    }
}

Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms
[Michael, Scott’96]
Michael and Scott Queue

class Node {
    int val;
    Node nd, nxt, tl;
    Node tl;
}
class Queue {
    Node head;
    Node tail;
} Q;

int dequeue() {
    Node nxt, hd, tl;
    int pval;
    while(true) {
        hd = Q->head;
        tl = Q->tail;
        nxt = hd->tl;
        if (Q->head != hd) continue;
        if (hd == tl) {
            if (nxt == NULL)
                return EMPTY;
            cas(Q->tail, tl, nxt);
        } else {
            pval = next->val;
            if (cas(Q->head, hd, nxt))
                return pval;
        }
    }
}

void enqueue(int v) {
    Node nd, nxt, tl;
    int b1;
    nd = new();
    nd->val = v;
    nd->tl = NULL;
    while(true) {
        tl = Q->tail;
        nxt = tl->tl
        if (Q->tail == tl) b1 = 1;
        else b1 = 0;
        if (b1!=0)
            if (nxt == 0)
                if (cas(tl->tl,nxt,nd))
                    break;
            else cas(Q->tail,tl, nxt);
        } cas(Q->tail, tl, nd);
    }

[Michael,Scott'96]
class Node {
    int val; // -1 NAN
    Node tl;
    thread_id alloc;
}

class Queue {
    Node head;
    Node tail;
} Q;

void enqueue(int value) {
    Node nd, tl;
    nd = new();
    nd->alloc = TID;
    nd->val = -1;
    nd->tl = NULL;
    atomic {
        tl = Q->tail;
        tl->tl = nd;
        Q->tail = nd;
    } // end of slot reservation;
    nd->val = value;//value written;
}

int dequeue() {
    Node curr, tail;
    int pval;
    while (true) {
        curr = Q->head;
        tail = Q->tail;
        while (curr != tail) {
            atomic { //atomic swap
                pval = curr->val;
                curr->val = -1;
                if (pval != -1)
                    return pval;
                curr = curr->tl;
            }
        }
    }
}

Linearizability: A Correctness Condition for Concurrent Objects
[Herlihy and Wing '90]