Non-Blocking Synchronization and Object-Oriented Operating System Design

C. Scott Ananian

cananian@csail.mit.edu

Computer Science and Artifical Intelligence Laboratory

Massachusetts Institute of Technology

Our Goal

The design of a

Object-Oriented

Non-Blocking

Operating System

Our Goal

The design of a *language to support*

Object-Oriented

Non-Blocking

Operating Systems

Why Object-Oriented?

Clear interfaces and strong encapsulation provide for:

- Safety
  - Software protection mechanisms.
- Ease
  - Clean composition semantics.
  - Uniform synchronization.
- Performance
  - Specialized implementations.
  - Natural grouping/locality.

Why Non-Blocking?

- Increased robustness
  - No deadlocks, no bookkeeping.
- Better decoupling
  - Better code structure; protection from asynchronous events.
- Increased parallelism
  - No idle processors, no convoys.
- Low overhead
  - No semaphore queue maintenance.
- Progress guarantees
  - Real-time properties; no priority inversion.

Outline

- Survey prior non-blocking O-O Operating Systems:
  - Synthesis [Massalin and Pu 1991]
  - Cache Kernel [Greenwald and Cheriton 1996]
- A more general approach:
  - Language support for synchronization
  - Functional Arrays
  - Single-object protocol
  - Multiple-object protocol
  - Lock-free functional arrays
- Assessment and conclusions

The Synthesis Kernel

- Synthesis is a lock-free OS implemented by Massalin and Pu.
- Explored use of run-time specialization for efficiency.
- Object encapsulation required to enable specialization; objects called quajects.
- Implemented in 680x0 assembly; macro support for quajects.
- Extremely high performance.

The Synthesis Kernel, cont.

Types of quajects:

- Threads
- Memory segments
- Symbol tables
- Data channels (I/O, pipes, filters, . . .)

Three quajects for synchronization:

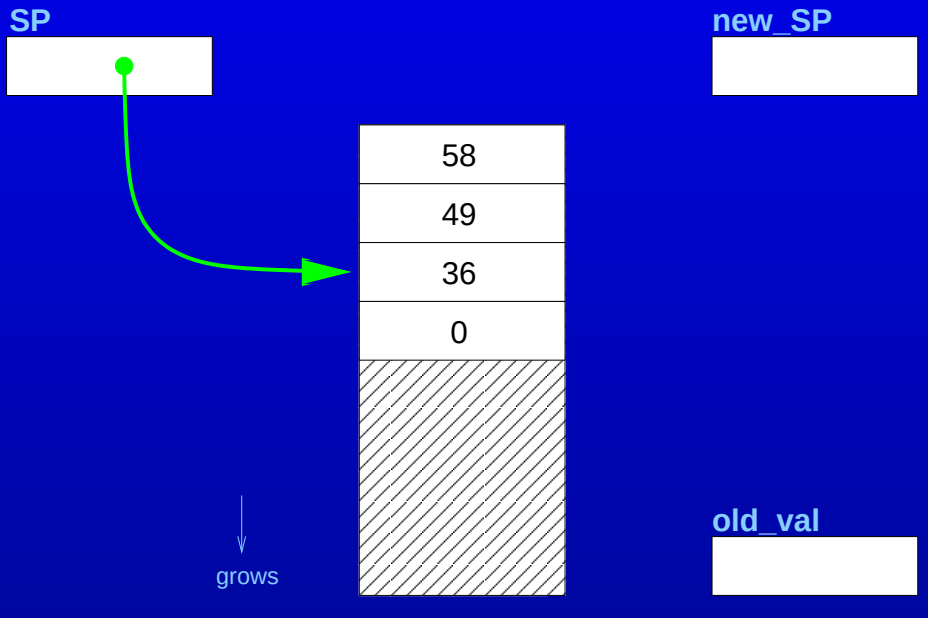
- LIFO stack
- FIFO queue (4 variants)
- Linked list

Synthesis Synchronization Errors

Three synchronization errors found in published lock-free algorithms for Synthesis.

- One ABA problem in LIFO stack.
- One likely race in MP-SC FIFO queue.
- One interesting corner case in quaject callback handling.

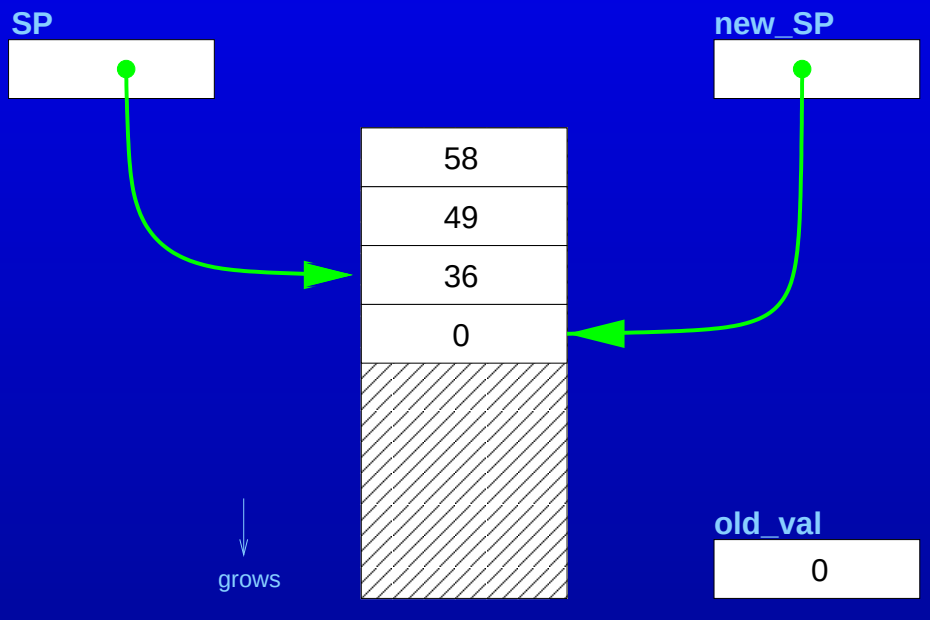
LIFO stack: Push




```
Push(elem)
{
retry:
    old_SP = SP;
    new_SP = old_SP - 1;
    old_val = *new_SP;
    if(DCAS(old_SP, old_val,
            new_SP, elem,
            &SP,    new_SP) == FAIL)
        goto retry;
}
```

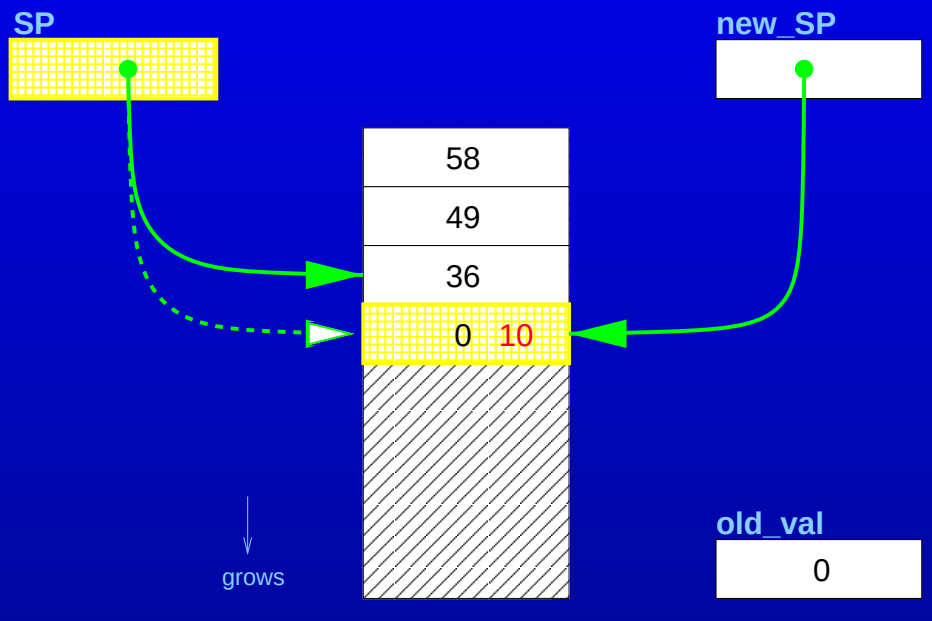
LIFO stack: Push

```
Push(elem)
{
retry:
     old_SP = SP;
     new_SP = old_SP - 1;
     old_val = *new_SP;
     if(DCAS(old_SP, old_val,
             new_SP, elem,
             &SP,    new_SP) == FAIL)
         goto retry;
}
```

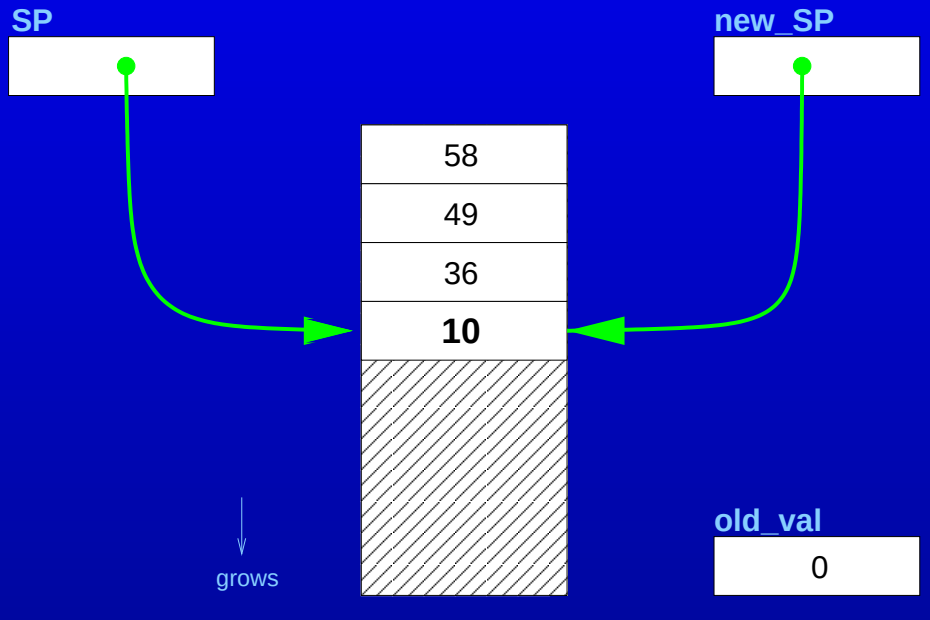


LIFO stack: Push

```
Push(elem)
{
retry:
     old_SP = SP;
     new_SP = old_SP - 1;
     old_val = *new_SP;
     if(DCAS(old_SP, old_val,
             new_SP, elem,
             &SP,    new_SP) == FAIL)
          goto retry;
}
```



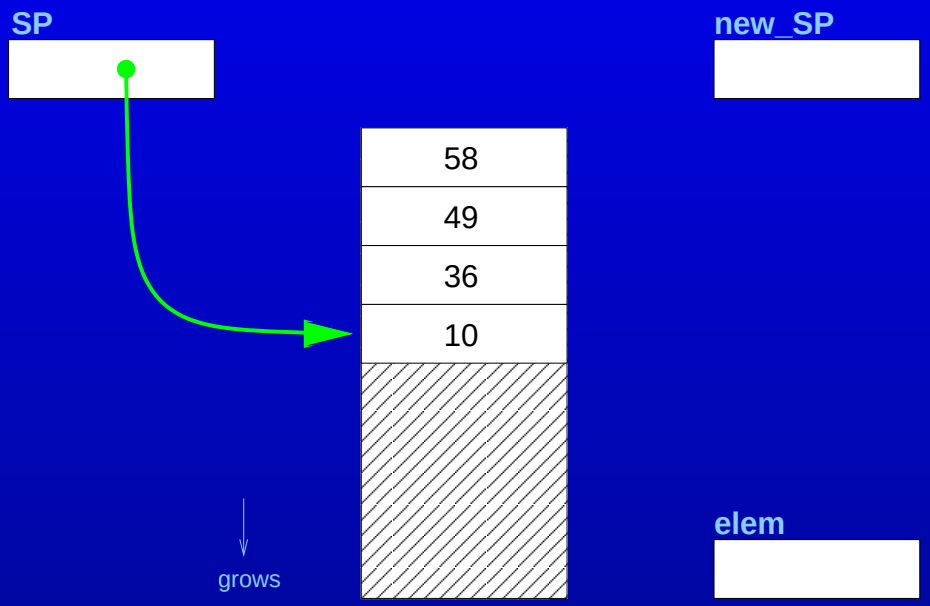
LIFO stack: Push




```
Push(elem)
{
retry:
     old_SP = SP;
     new_SP = old_SP - 1;
     old_val = *new_SP;
     if(DCAS(old_SP, old_val,
             new_SP, elem,
             &SP,    new_SP) == FAIL)
         goto retry;
}
```

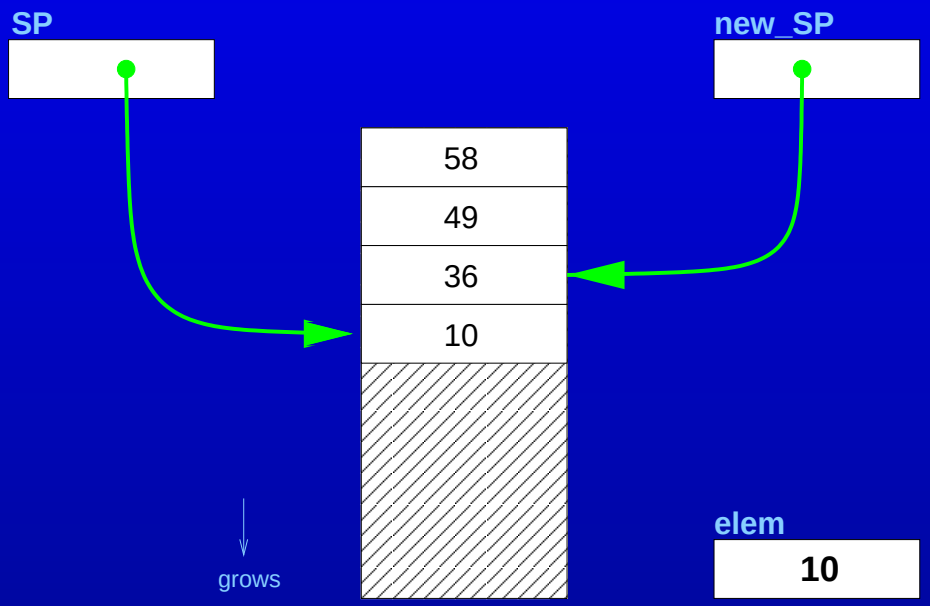
LIFO stack: Pop

```
Pop()
{
retry:
    old_SP = SP;
    new_SP = old_SP + 1;
    elem = *old_SP;
    if(CAS(old_SP, new_SP, &SP) == FAIL)
        goto retry;
    return elem;
}
```

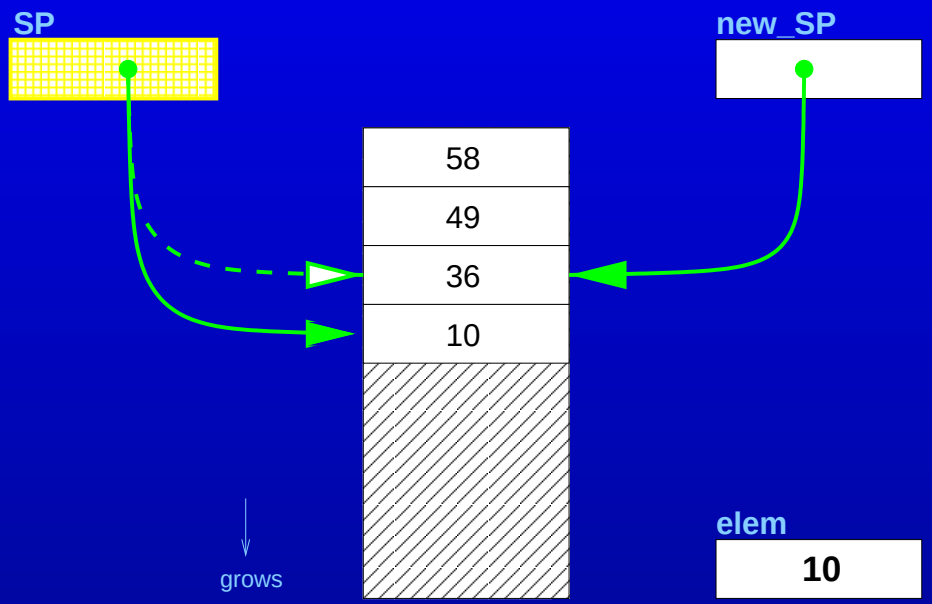


LIFO stack: Pop

```
Pop()
{
retry:
    old_SP = SP;
    new_SP = old_SP + 1;
    elem = *old_SP;
    if(CAS(old_SP, new_SP, &SP) == FAIL)
        goto retry;
    return elem;
}
```



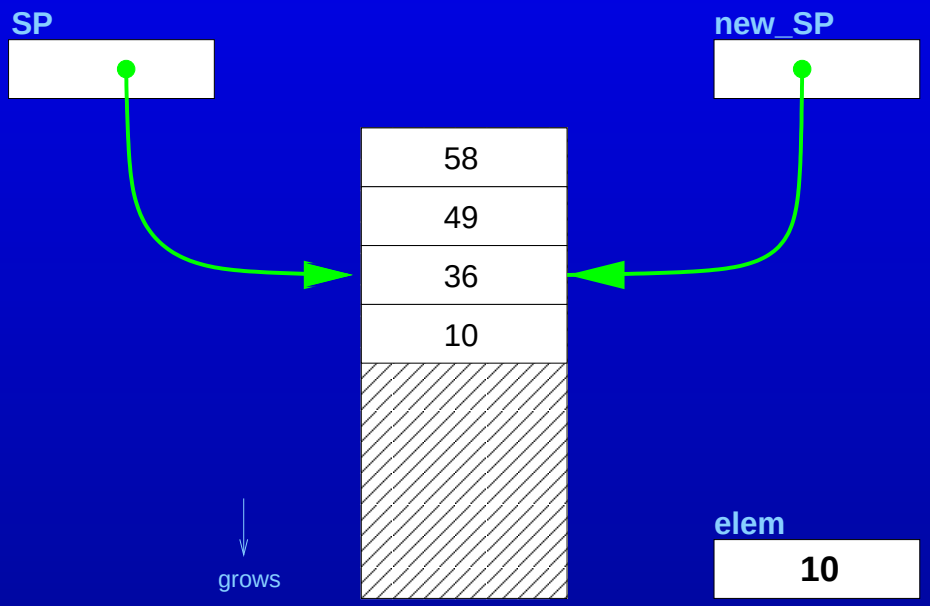
LIFO stack: Pop



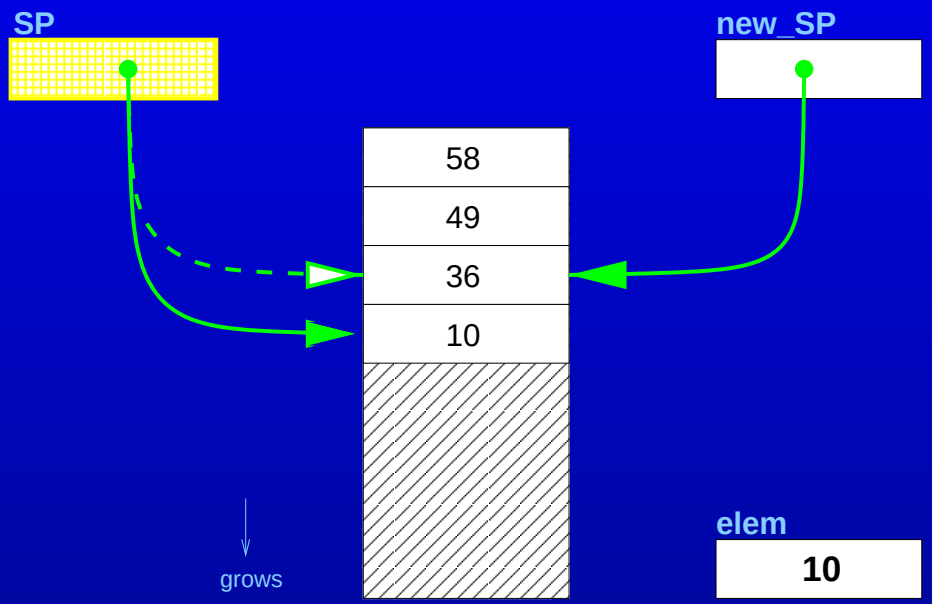

```
Pop()
{
retry:
    old_SP = SP;
    new_SP = old_SP + 1;
    elem = *old_SP;
    if(CAS(old_SP, new_SP, &SP) == FAIL)
        goto retry;
    return elem;
}
```

LIFO stack: Pop

```
Pop()
{
retry:
    old_SP = SP;
    new_SP = old_SP + 1;
    elem = *old_SP;
    if(CAS(old_SP, new_SP, &SP) == FAIL)
        goto retry;
    return elem;
}
```

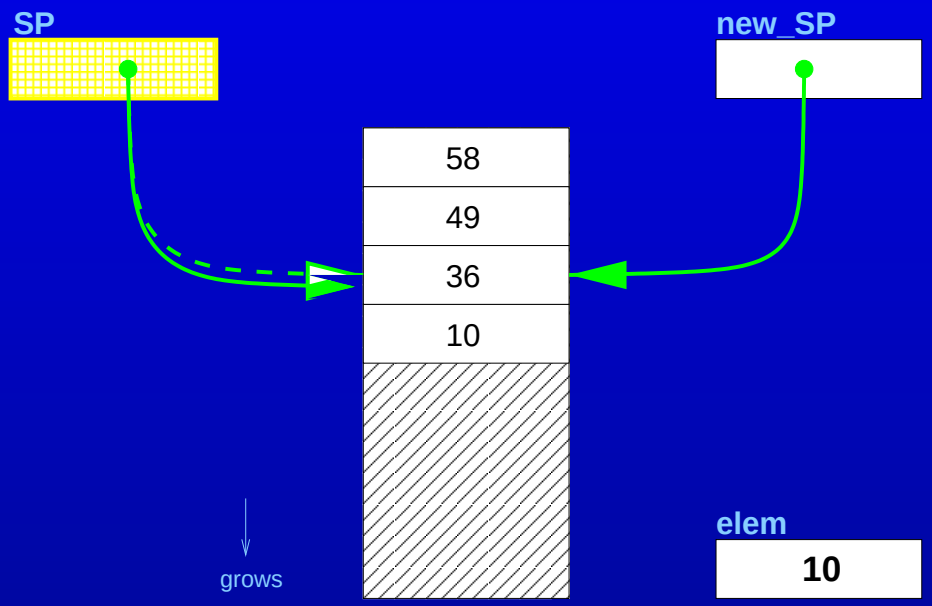


LIFO stack: Pop




```
Pop()
{
retry:
    old_SP = SP;
    new_SP = old_SP + 1;
    elem = *old_SP;
    if(CAS(old_SP, new_SP, &SP) == FAIL)
        goto retry;
    return elem;
}
```

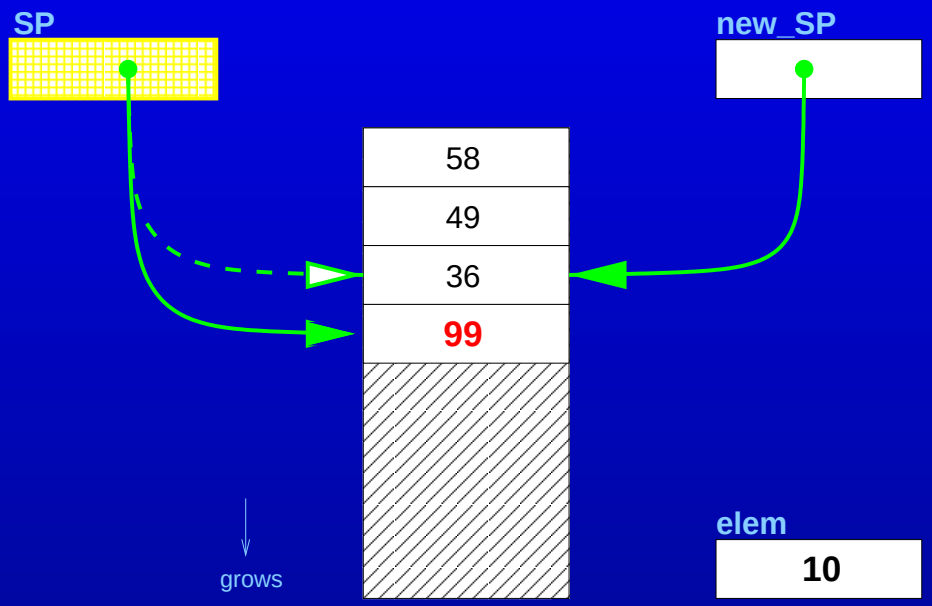
LIFO stack: Pop




```
Pop()
{
retry:
    old_SP = SP;
    new_SP = old_SP + 1;
    elem = *old_SP;
    if(CAS(old_SP, new_SP, &SP) == FAIL)
        goto retry;
    return elem;
}
```

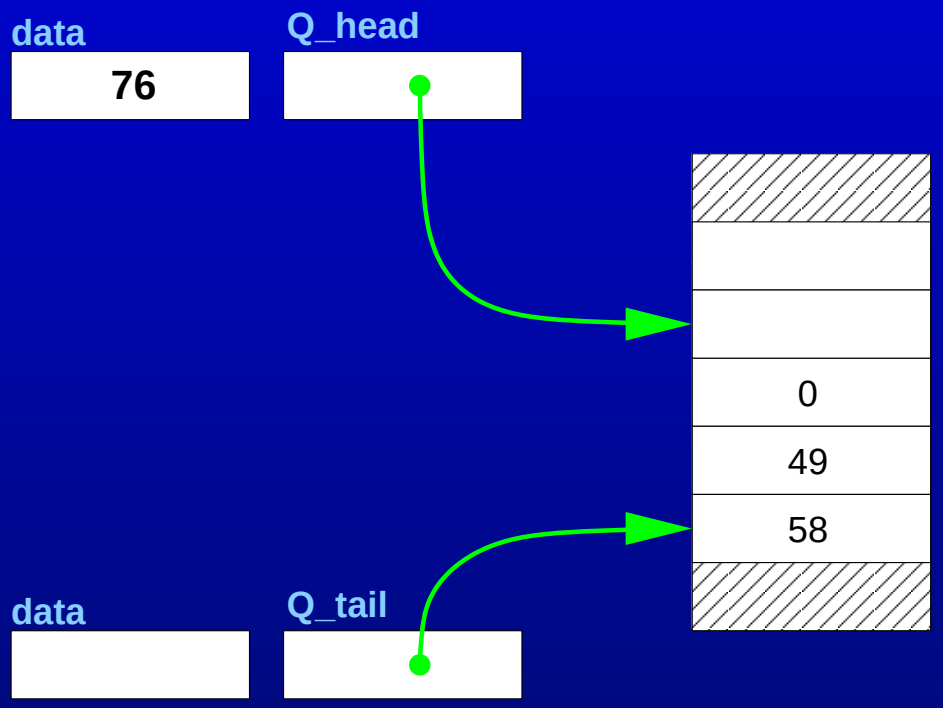
LIFO stack: Pop

```
Pop()
{
retry:
    old_SP = SP;
    new_SP = old_SP + 1;
    elem = *old_SP;
    if(CAS(old_SP, new_SP, &SP) == FAIL)
        goto retry;
    return elem;
}
```



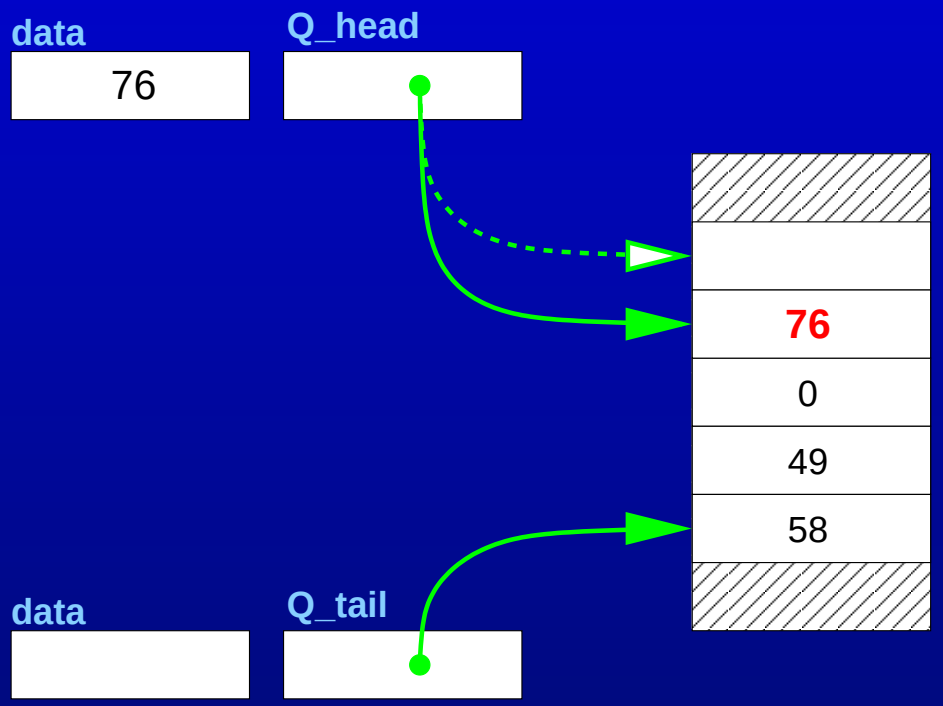
SP-SC FIFO queue

```
next(x) { return (x+1) % Q_size; }

Q_put(data) {
    h = Q_head;
    if (next(h) == Q_tail)
        wait;
    Q_buf[h] = data;
    Q_head = next(h);
}
Q_get() {
    t = Q_tail;
    if (t == Q_head)
        wait;
    data = Q_buf[t];
    Q_tail = next(t);
    return data;
}
```



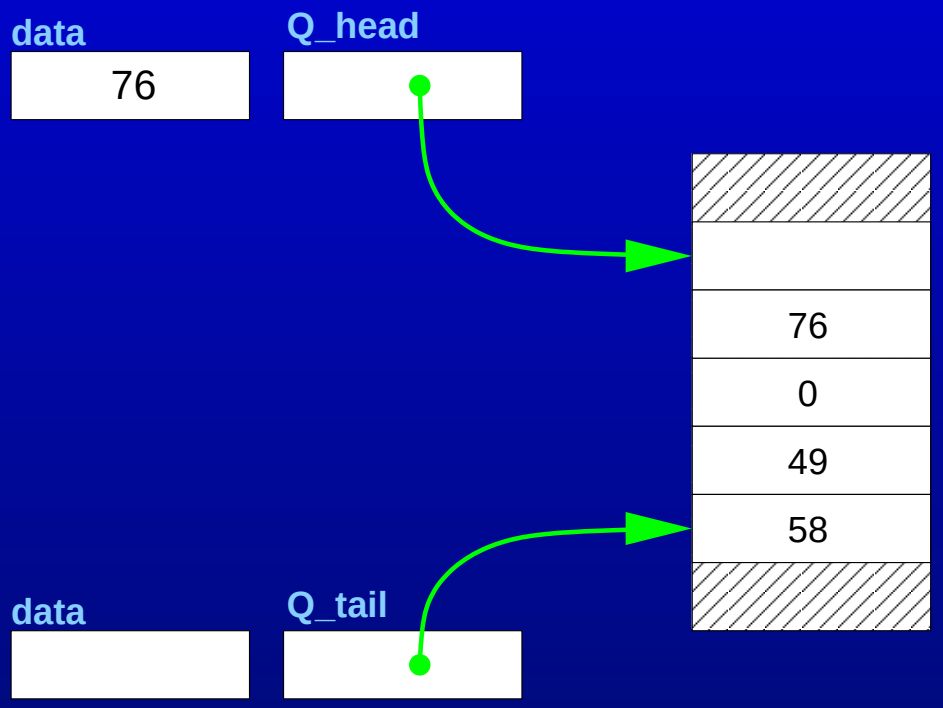
SP-SC FIFO queue

```
next(x) { return (x+1) % Q_size; }

Q_put(data) {
    h = Q_head;
    if (next(h) == Q_tail)
        wait;
    Q_buf[h] = data;
    Q_head = next(h);
}
Q_get() {
    t = Q_tail;
    if (t == Q_head)
        wait;
    data = Q_buf[t];
    Q_tail = next(t);
    return data;
}
```



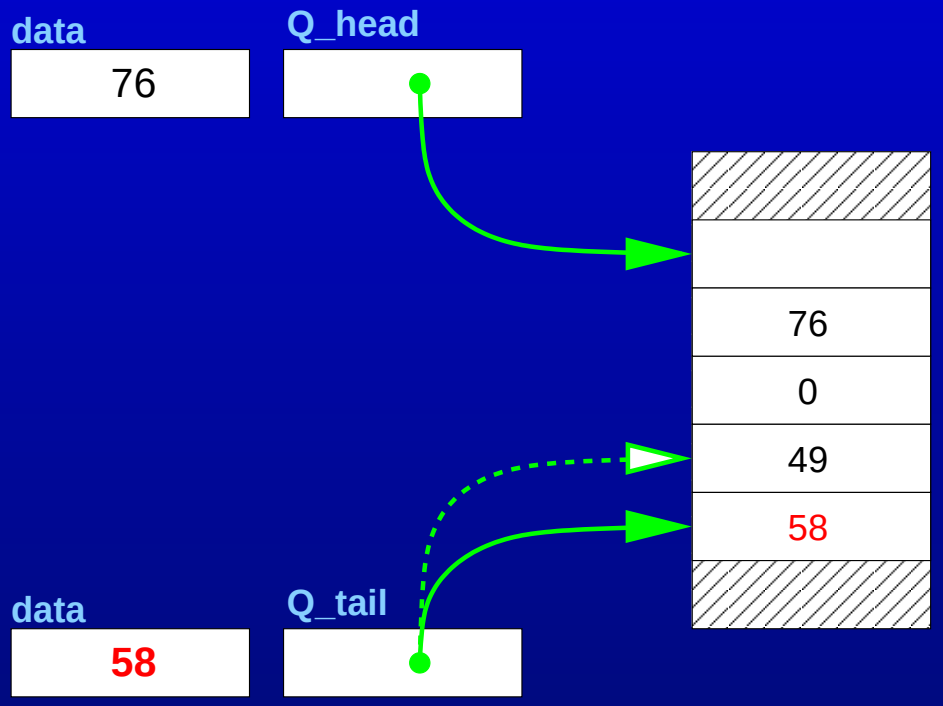
SP-SC FIFO queue

```
next(x) { return (x+1) % Q_size; }

Q_put(data) {
    h = Q_head;
    if (next(h) == Q_tail)
        wait;
    Q_buf[h] = data;
    Q_head = next(h);
}
Q_get() {
    t = Q_tail;
    if (t == Q_head)
        wait;
    data = Q_buf[t];
    Q_tail = next(t);
    return data;
}
```



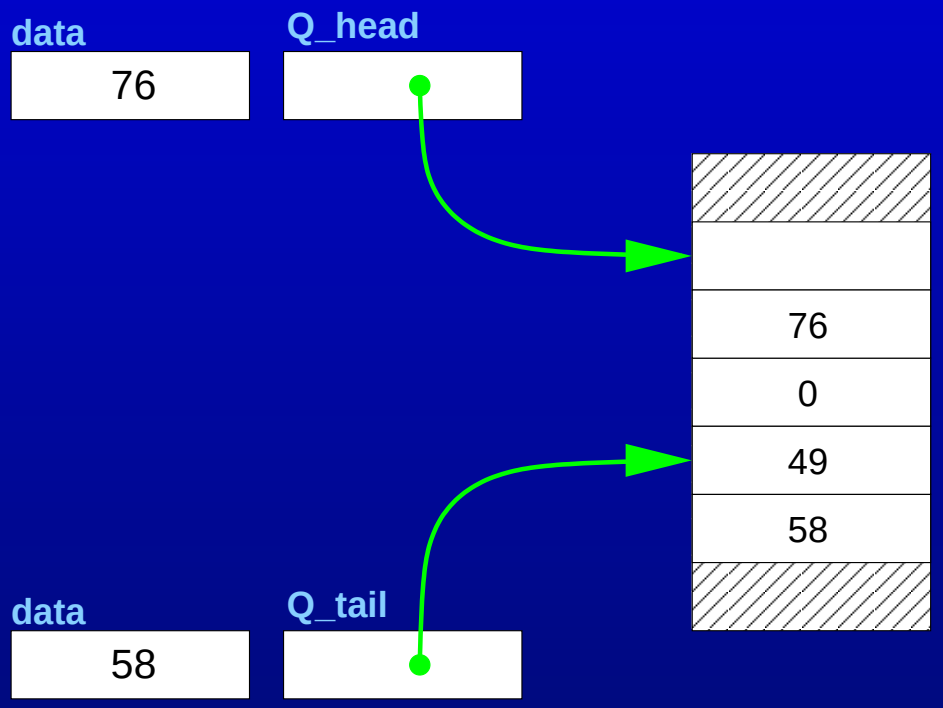
SP-SC FIFO queue

```
next(x) { return (x+1) % Q_size; }

Q_put(data) {
    h = Q_head;
    if (next(h) == Q_tail)
        wait;
    Q_buf[h] = data;
    Q_head = next(h);
}
Q_get() {
    t = Q_tail;
    if (t == Q_head)
        wait;
    data = Q_buf[t];
    Q_tail = next(t);
    return data;
}
```



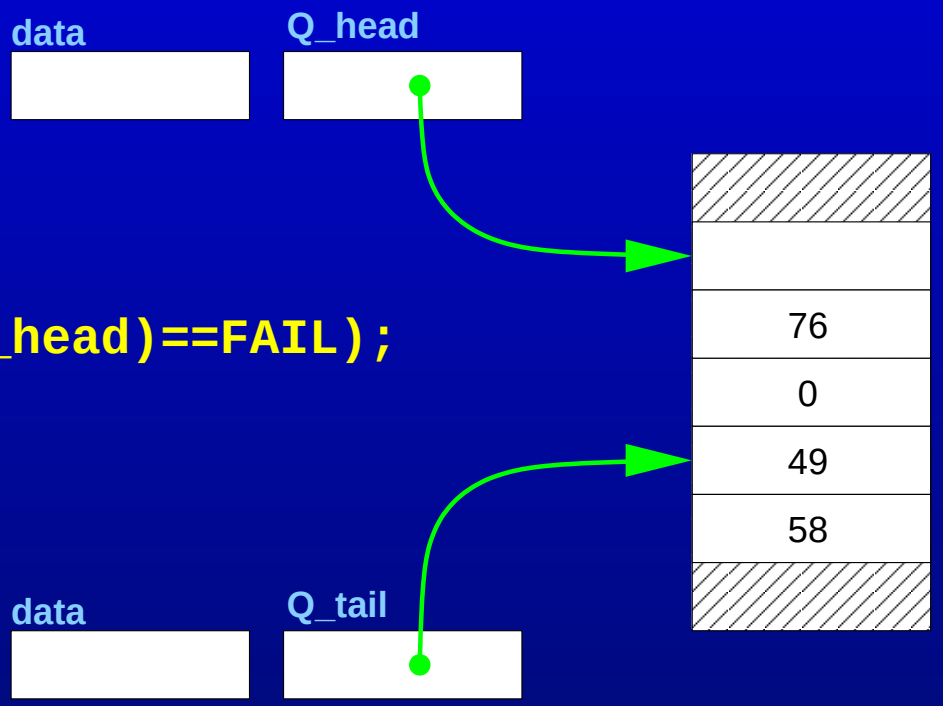
SP-SC FIFO queue

```
next(x) { return (x+1) % Q_size; }

Q_put(data) {
    h = Q_head;
    if (next(h) == Q_tail)
        wait;
    Q_buf[h] = data;
    Q_head = next(h);
}
Q_get() {
    t = Q_tail;
    if (t == Q_head)
        wait;
    data = Q_buf[t];
    Q_tail = next(t);
    return data;
}
```



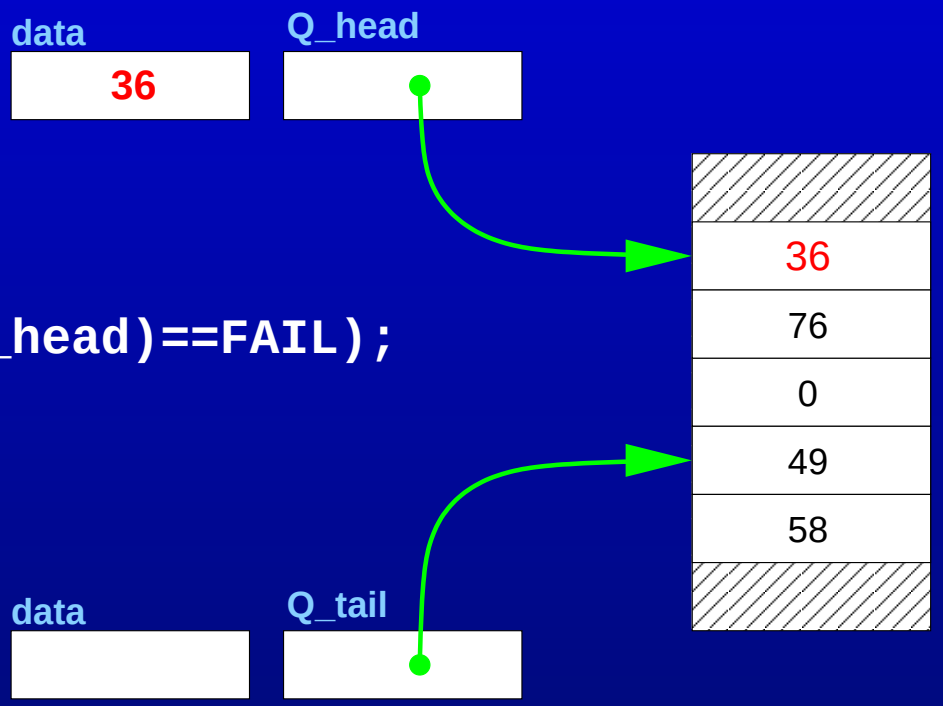
MP-SC FIFO queue

```
next(x) { return (x+1) % Q_size; }
Q_put(data) {
  do {
    h = Q_head;
    if (next(h) == Q_tail)
        wait;
    Q_buf[h] = data;
  } while (CAS(h, next(h), &Q_head)==FAIL);
}
Q_get() {
    t = Q_tail;
    if (t == Q_head)
        wait;
    data = Q_buf[t];
    Q_tail = next(t);
    return data;
}
```



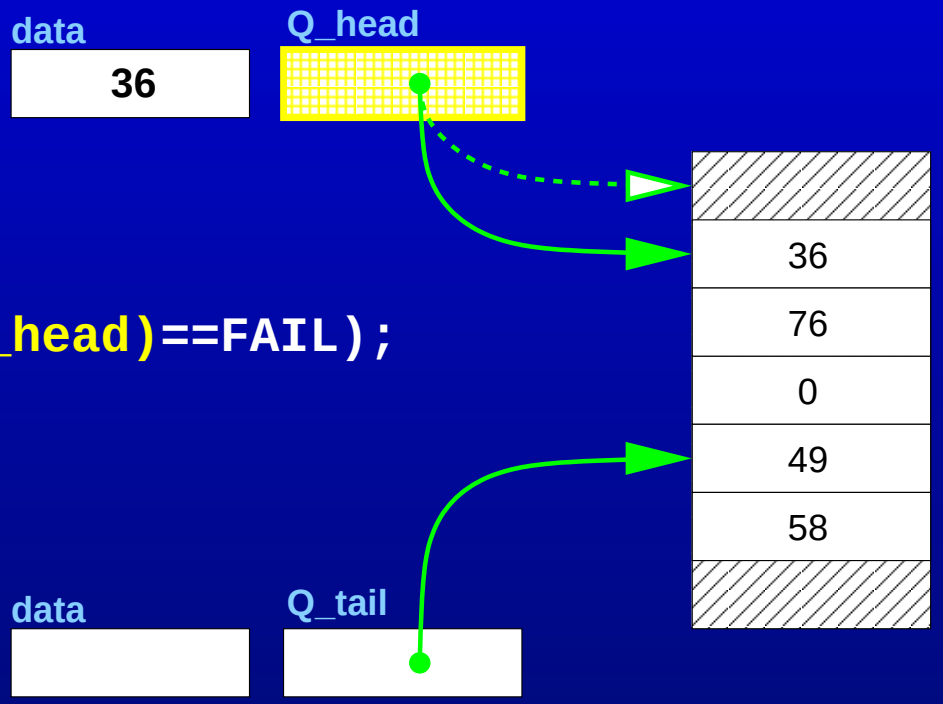
MP-SC FIFO queue

```
next(x) { return (x+1) % Q_size; }
Q_put(data) {
  do {
    h = Q_head;
    if (next(h) == Q_tail)
        wait;
    Q_buf[h] = data;
  } while (CAS(h, next(h), &Q_head)==FAIL);
}
Q_get() {
    t = Q_tail;
    if (t == Q_head)
        wait;
    data = Q_buf[t];
    Q_tail = next(t);
    return data;
}
```



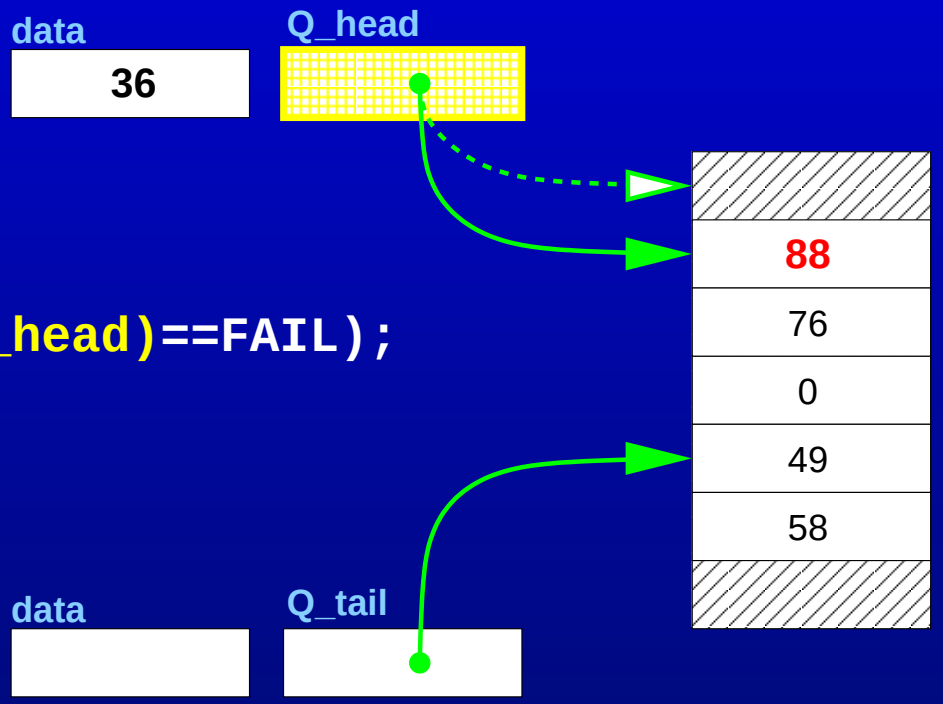
MP-SC FIFO queue

```
next(x) { return (x+1) % Q_size; }
Q_put(data) {
  do {
    h = Q_head;
    if (next(h) == Q_tail)
        wait;
    Q_buf[h] = data;
  } while (CAS(h, next(h), &Q_head)==FAIL);
}
Q_get() {
    t = Q_tail;
    if (t == Q_head)
        wait;
    data = Q_buf[t];
    Q_tail = next(t);
    return data;
}
```



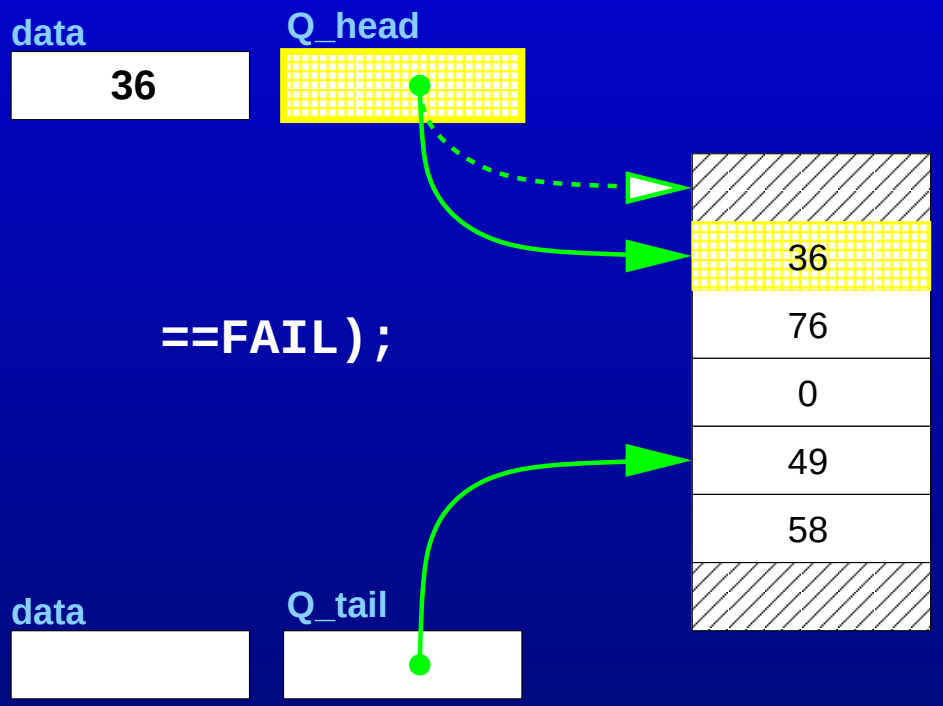
MP-SC FIFO queue

```
next(x) { return (x+1) % Q_size; }
Q_put(data) {
  do {
    h = Q_head;
    if (next(h) == Q_tail)
        wait;
    Q_buf[h] = data;
  } while (CAS(h, next(h), &Q_head)==FAIL);
}
Q_get() {
    t = Q_tail;
    if (t == Q_head)
        wait;
    data = Q_buf[t];
    Q_tail = next(t);
    return data;
}
```



MP-SC FIFO queue

```
next(x) { return (x+1) % Q_size; }
Q_put(data) {
  do {
    h = Q_head;
    if (next(h) == Q_tail)
        wait;
    Q_buf[h] = data;
  } while (DCAS(    ...    )        ==FAIL);
}
Q_get() {
    t = Q_tail;
    if (t == Q_head)
        wait;
    data = Q_buf[t];
    Q_tail = next(t);
    return data;
}
```



Quaject callback

Queue quaject hooked up to a hardware interrupt (consumer) and a user thread (producer).

Kind of Reference	User Thread		ByteQueue	ByteQueue	Device Driver		Hardware
callentry	**write**	$\Longrightarrow$	`Q_put`	`Q_get`		$\Longleftarrow$	*send-complete* interrupt
callback	**suspend**	$\Longleftarrow$	`Q_full`	`Q_empty`		$\Longrightarrow$	turn off *send-complete*
callback	**resume**	$\Longleftarrow$	`Q_full-1`	`Q_empty-1`		$\Longrightarrow$	turn on *send-complete*

- Calls to `Q_put` (`Q_get`) return immediately as long as queue is not full (empty), otherwise the `Q_full` (`Q_empty`) callback is invoked.
- When the queue later empties (fills) the `Q_full-1` (`Q_empty-1`) callback is invoked.

Quaject callback

Queue quaject hooked up to a hardware interrupt (consumer) and a user thread (producer).

Kind of Reference	User Thread		ByteQueue	ByteQueue	Device Driver	Hardware
callentry	**write**	$\Longrightarrow$	`Q_put`	`Q_get`	$\Longleftarrow$	*send-complete* interrupt
callback	**suspend**	$\Longleftarrow$	`Q_full`	`Q_empty`	$\Longrightarrow$	turn off *send-complete*
callback	**resume**	$\Longleftarrow$	`Q_full-1`	`Q_empty-1`	$\Longrightarrow$	turn on *send-complete*

- Intended to emulate blocking I/O when callbacks are hooked up to thread suspend/resume.
- BUT what if queue empties between invocation of `Q_full` and the actual thread suspend?

Quaject callback

Queue quaject hooked up to a hardware interrupt (consumer) and a user thread (producer).

Kind of Reference	User Thread		ByteQueue	ByteQueue	Device Driver		Hardware
callentry	**write**	$\Longrightarrow$	`Q_put`	`Q_get`	$\Longleftarrow$		*send-complete* interrupt
callback	**suspend**	$\Longleftarrow$	`Q_full`	`Q_empty`	$\Longrightarrow$		turn off *send-complete*
callback	**resume**	$\Longleftarrow$	`Q_full-1`	`Q_empty-1`	$\Longrightarrow$		turn on *send-complete*

- Solution #1: Disable signals/interrupts in this critical region.
- Solution #2: Atomically add to suspend list iff queue is still full. What about additional compare?

The Cache Kernel
[Greenwald and Cheriton 1996]

- Minimal microkernel with only three operating system object types:
  - Address spaces
  - Threads
  - Application kernels
- Only one interprocess notification mechanism: asynchronous signals.
- Lock-free implementation to handle large amount of asynchrony w/o coupling.
- Lock-free sync allows pushing OS functions into userland w/o deadlock when user threads are terminated.

Cache Kernel synchronization

General strategy for lock-free data structures:

- Each data structure has a version number.
- Each modification to the data structure increments the version number.

To make a one-word change:

- Read and remember v, the current version number.
- Traverse the structure to compute the change.
- Use DCAS to atomically apply the change and increment the version number, conditional on the current version number still being equal to v.

Lock-Free Linked List

```
Delete(elt) {
    do {
    retry:
        backoffIfNeeded();
        version = list->version;

        for (p = list->head; p->next != elt; p = p->next) {
            if (p==NULL) {              /* Not found */
                if (version != list->version)
                    goto retry;           /* Changed */
                return NULL;    /* Really not found */
            }
    } while (!DCAS(&(list->version), &(p->next),
                   version,           elt,
                   version+1,         elt->next));
    return elt;
}
```

Version-Number/DCAS Limitations

Version number protects *entire* data structure.

- Concurrent mutations to non-interfering sections of data structure not allowed.
- Workaround: break up data structure. List of lists, etc.

Only one-word mutations are allowed.

- Copy-and-swap larger objects.
  - Copying is expensive!
- Remove, mutate, and add.
  - Results in “best-effort” data structures which require high-level timeout and retry mechanisms.

Outline

- Survey prior non-blocking O-O Operating Systems:
  - Synthesis [Massalin and Pu 1991]
  - Cache Kernel [Greenwald and Cheriton 1996]
- A more general approach:
  - Language support for synchronization
  - Functional Arrays
  - Single-object protocol
  - Multiple-object protocol
  - Lock-free functional arrays
- Assessment and conclusions

A More General Scheme

Ad hoc impl. of lock-free data structures are:

- Hard to get right!
  - Three errors in Synthesis.
- Limited
  - Small number of hand-coded data structures.
- Brittle
  - Small number of atomic actions.
  - Forced to copy-and-swap to make larger actions atomic.
  - Copy-and-swap works poorly on large objects.

Solution: integrate synchronization into the language.

Monitor Synchronization

- Introduced by Emerald [Black et al 1986]; familiar now in Java.
- Every object contains a monitor which:
  - Enforces mutual exclusion
  - Serves as a signalling mechanism
- Certain methods are monitored
- Shared variables of objects can only be accessed by monitored methods.
  - Java doesn't enforce this.

Not sufficient to prevent unexpected parallel behavior!

Synchronization Failures

```
class A { // OK!
    int x; // shared variable
    synchronized int inc() {
        return x++;
    }
}

class B { // Race-free, but not OK.
    int x; // shared variable
    synchronized int get() { return x; }
    synchronized void set(int y) { x=y; }
    int inc() { // not monitored
        int t = get();
        t++;
        set(t);
        return t;
    }
}
```

Atomic Blocks

```
public class Count {
 private int cntr = 0;
 void inc() {
  synchronized(this) {
   cntr = cntr + 1;
  }
 }
}
```

- Traditionally, monitors associated with each object provide mutual exclusion between concurrent accesses to the object.

Atomic Blocks

```
public class Count {
 private int cntr = 0;
 void inc() {
  synchronized(this) {
   cntr = cntr + 1;
  }
 }
}
```

$\Rightarrow$

```
public class Count {
 private int cntr = 0;
 void inc() {
  atomic {
   cntr = cntr + 1;
  }
 }
}
```

- Instead we provide an `atomic` block, and make linearizablity guarantees without (necessarily) providing mutual exclusion.

Language Design for OSes

- We'll start with Java
  - C-like expressions, O-O structure, type-safety.
- Add low-level constructs
  - Interrupt linkage, fixed object layout for memory-mapped I/O
  - JEPES [Schultz et al 2003], Lisaac [Sonntag and Colnet 2002]
- Use software protection mechanisms
  - Remove address spaces from kernel
  - DrScheme [Flatt et al 1999]
- Provide atomic operation blocks
  - Implement with non-blocking synchronization

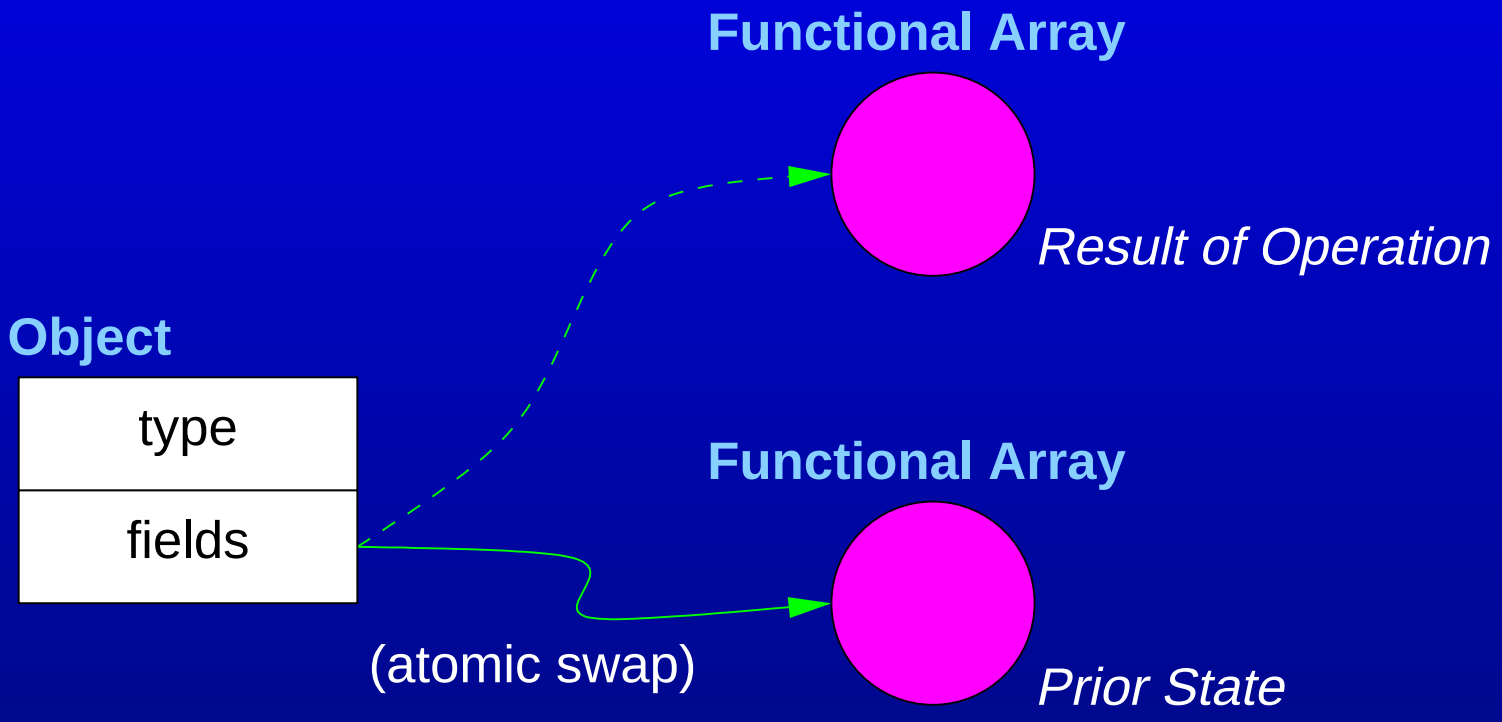
Functional Arrays

Our implementation of `atomic` blocks will be based on fast *functional arrays*.

- Functional arrays are persistent; after an element is updated both the new and the old contents of the array are available for use.
- Fundamental operation:
  $\textsc{Update}(A, i, v) : A \to \mathbb{N}_0 \to V \to A$
- Arrays are just mappings from integers to values; any persistent map can be used as a functional array.
- A *fast* functional array will have $O(1)$ access and update “for the common cases”.

Single Object Protocol

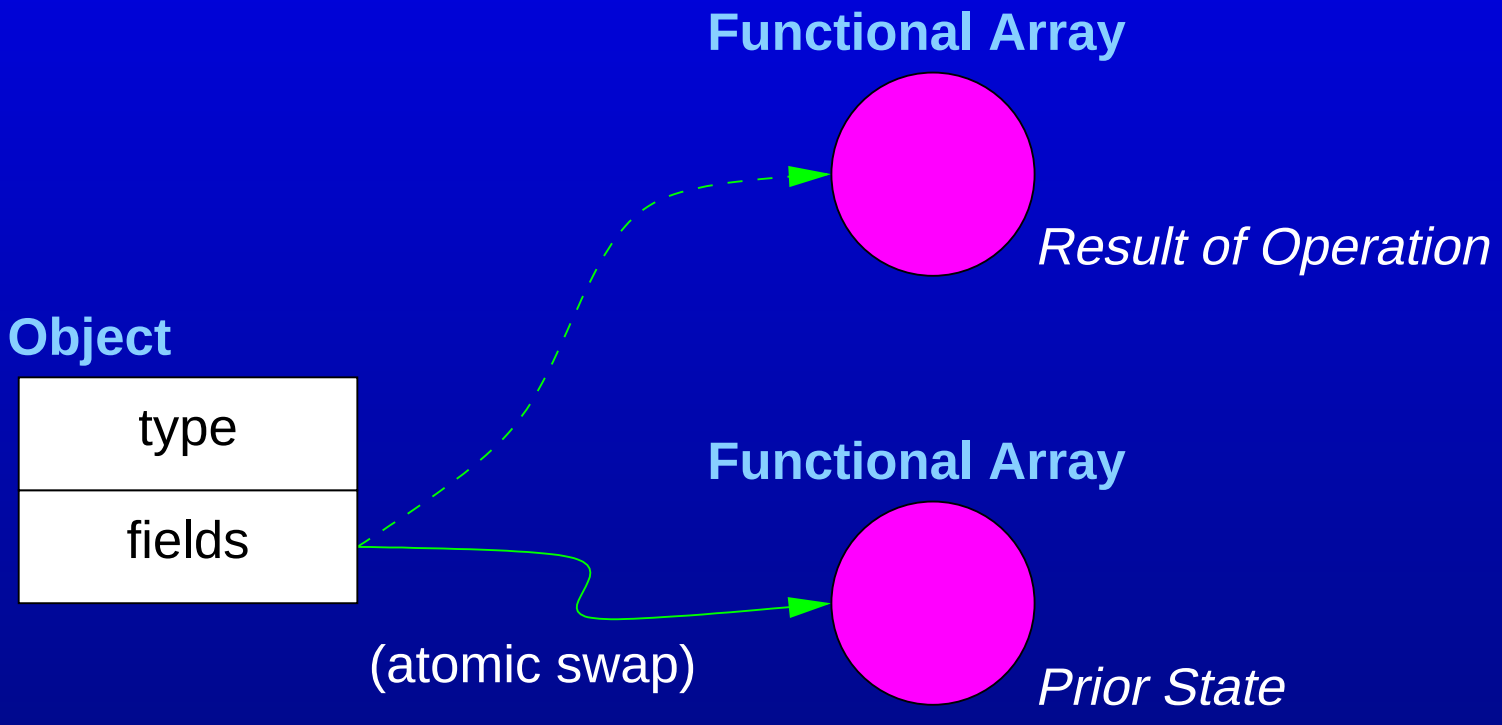
Valid for operations on a single object only.



- Object representation contains a pointer to a functional array.
- Object mutation inside `atomic` creates new functional array.

Single Object Protocol

Valid for operations on a single object only.



- At start of `atomic` block load and remember fields array pointer as *prior state*.
- At end of `atomic` block compare-and-swap the *result of operation* for the *prior state*.

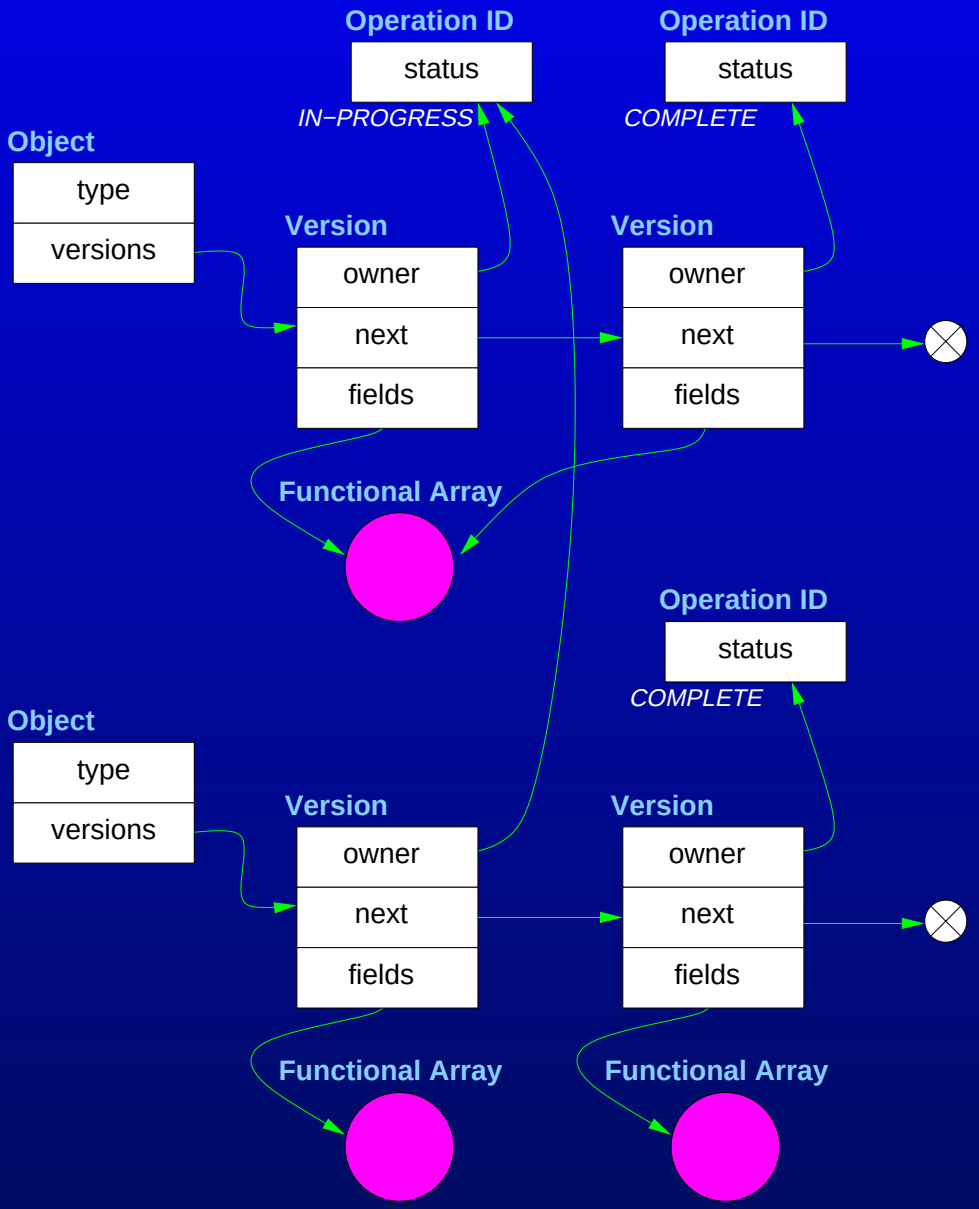
Problems with Multiple Objects

The case of the StringBuffer

```
public final class StringBuffer ...{
    ...
    private int count;

    public synchronized StringBuffer append(StringBuffer sb) {
        if (sb == null) { sb = NULL; }
        int len = sb.length(); // len may be stale.
        int newcount = count + len;
        if (newcount > value.length) expandCapacity(newcount);
        sb.getChars(0, len, value, count); // use of stale len
        count = newcount;
        return this;
    }
    public synchronized int length() { return count; }
    public synchronized void getChars(...)  { ...}
}
```

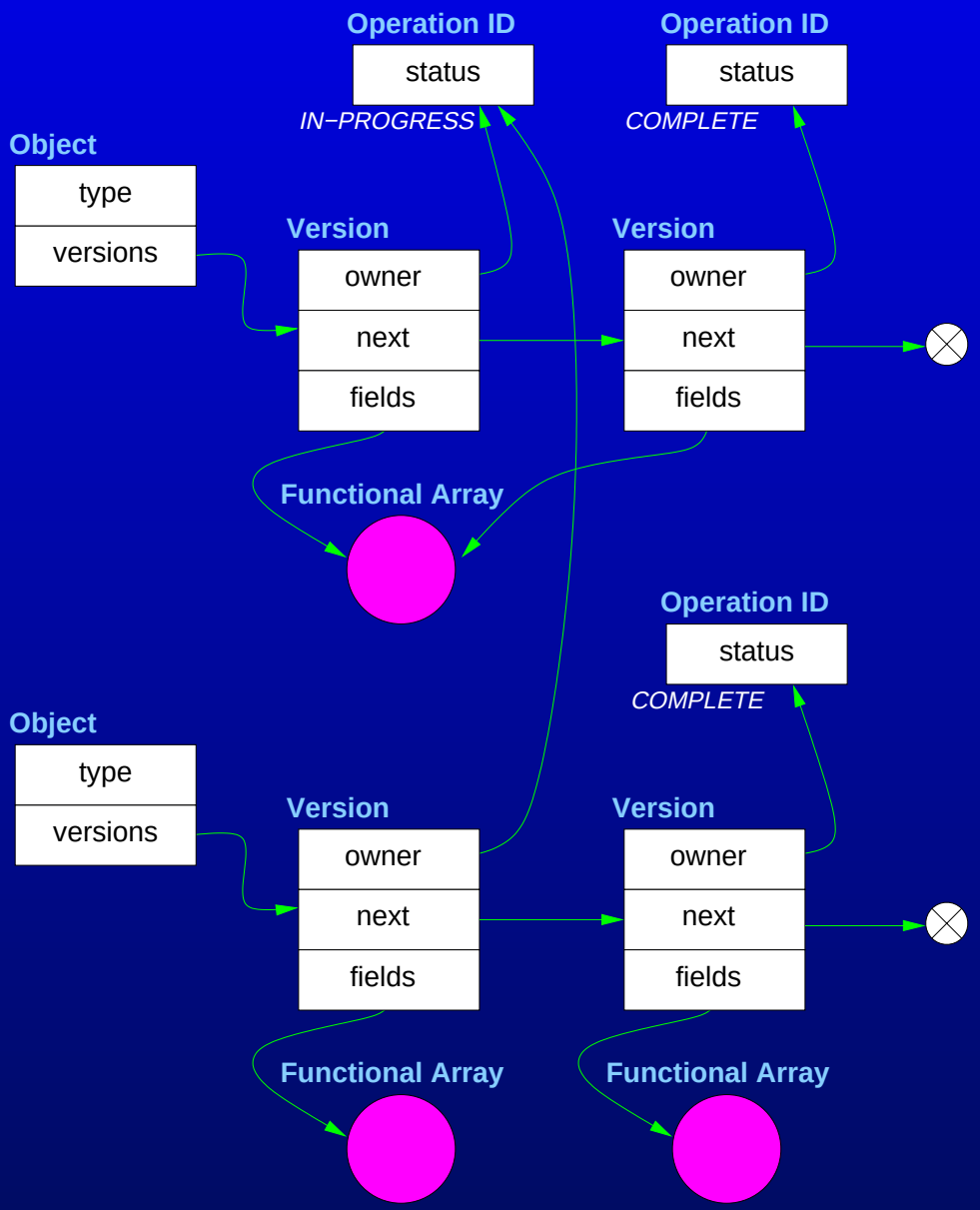
Multiple Object Protocol

- Objects point to version lists.
- Each version has an associated operation ID and field array reference.
- Operation IDs are initialized to *IN-PROGRESS* and are changed exactly once to *COMPLETE* or *DISCARDED*.



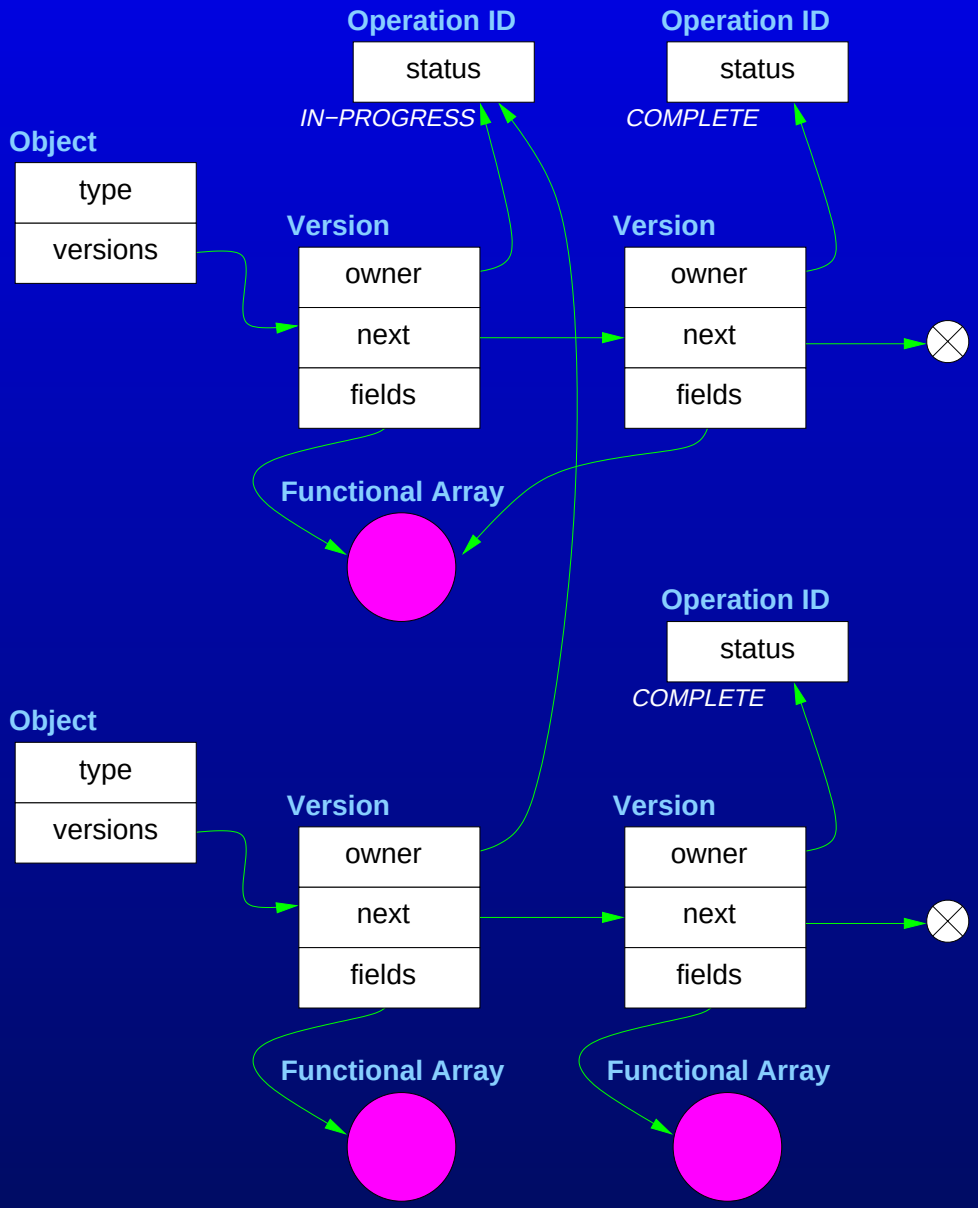
Multiple Object Protocol

- At end of atomic block, attempt to set Operation ID to *COMPLETE*.
- Value of object is value of first committed version.
- Old or *DISCARDED* versions can be trimmed.



Multiple Object Protocol

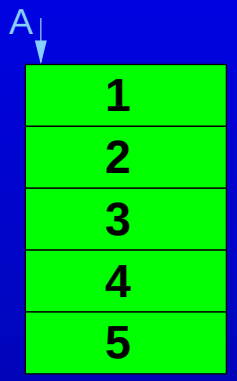
- Only one *IN-PROGRESS* version allowed on versions list, and it must be at the head.
- Before we can link a new version onto the versions list, we must ensure that every other version is either *COMPLETE* or *DISCARDED*.



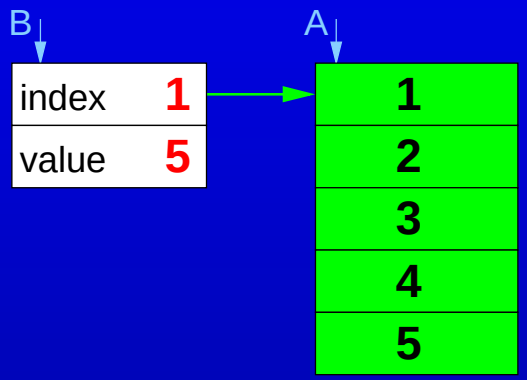
Outline

- Survey prior non-blocking O-O Operating Systems:
  - Synthesis [Massalin and Pu 1991]
  - Cache Kernel [Greenwald and Cheriton 1996]
- A more general approach:
  - Language support for synchronization
  - Functional Arrays
  - Single-object protocol
  - Multiple-object protocol
  - Lock-free functional arrays
- Assessment and conclusions

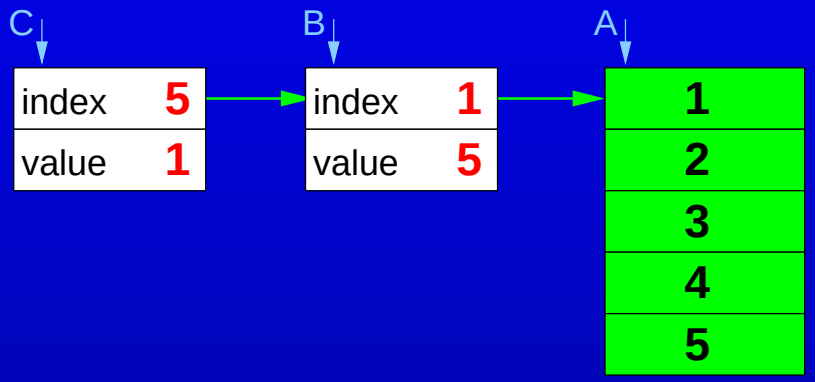
Functional Arrays using Shallow Binding



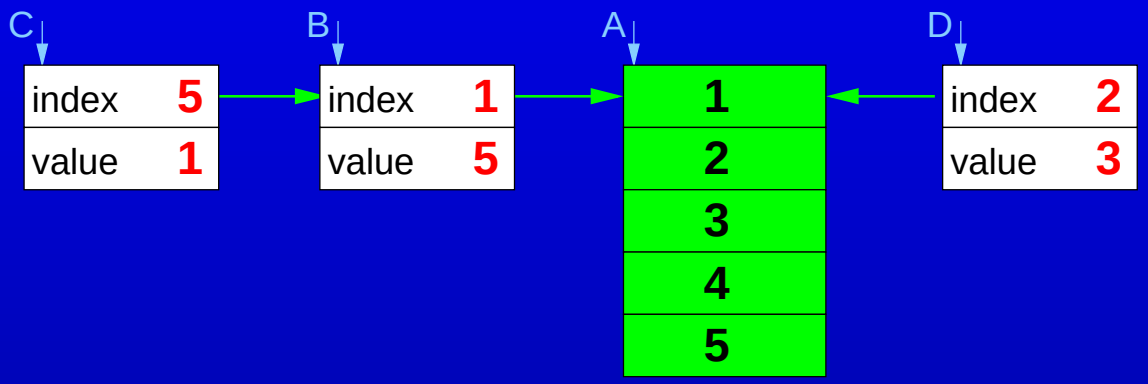
Functional Arrays using Shallow Binding



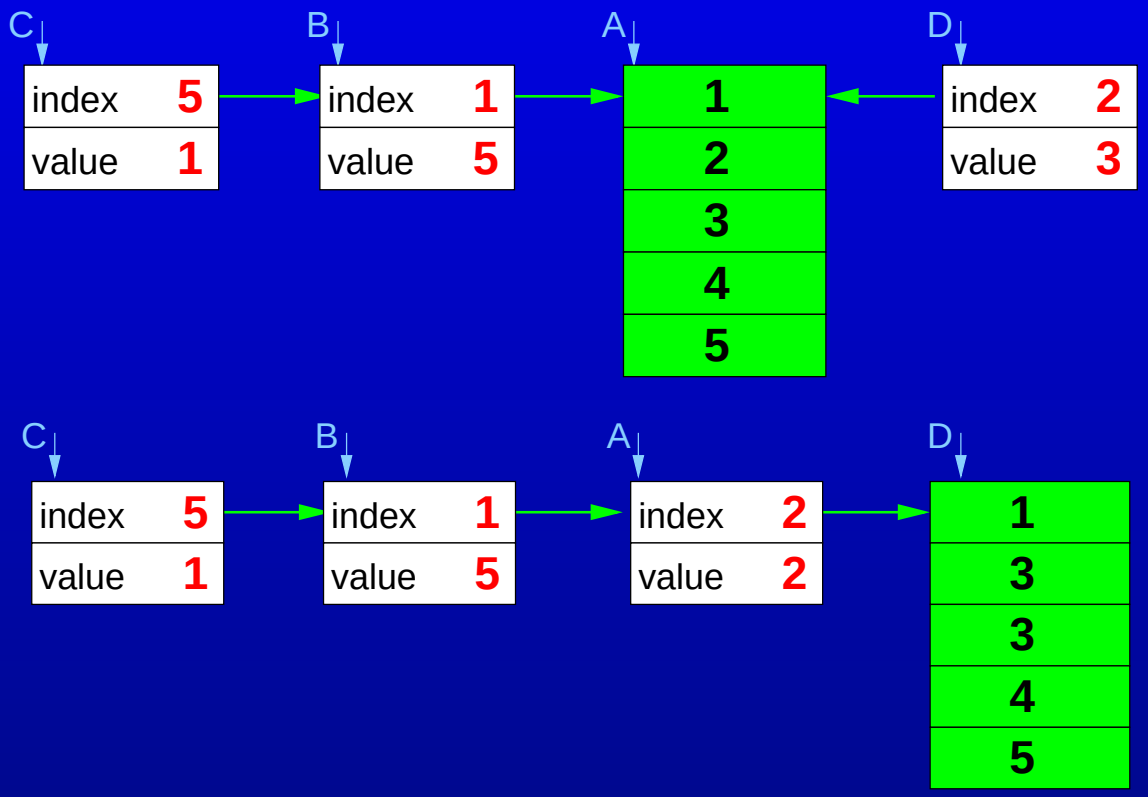
Functional Arrays using Shallow Binding



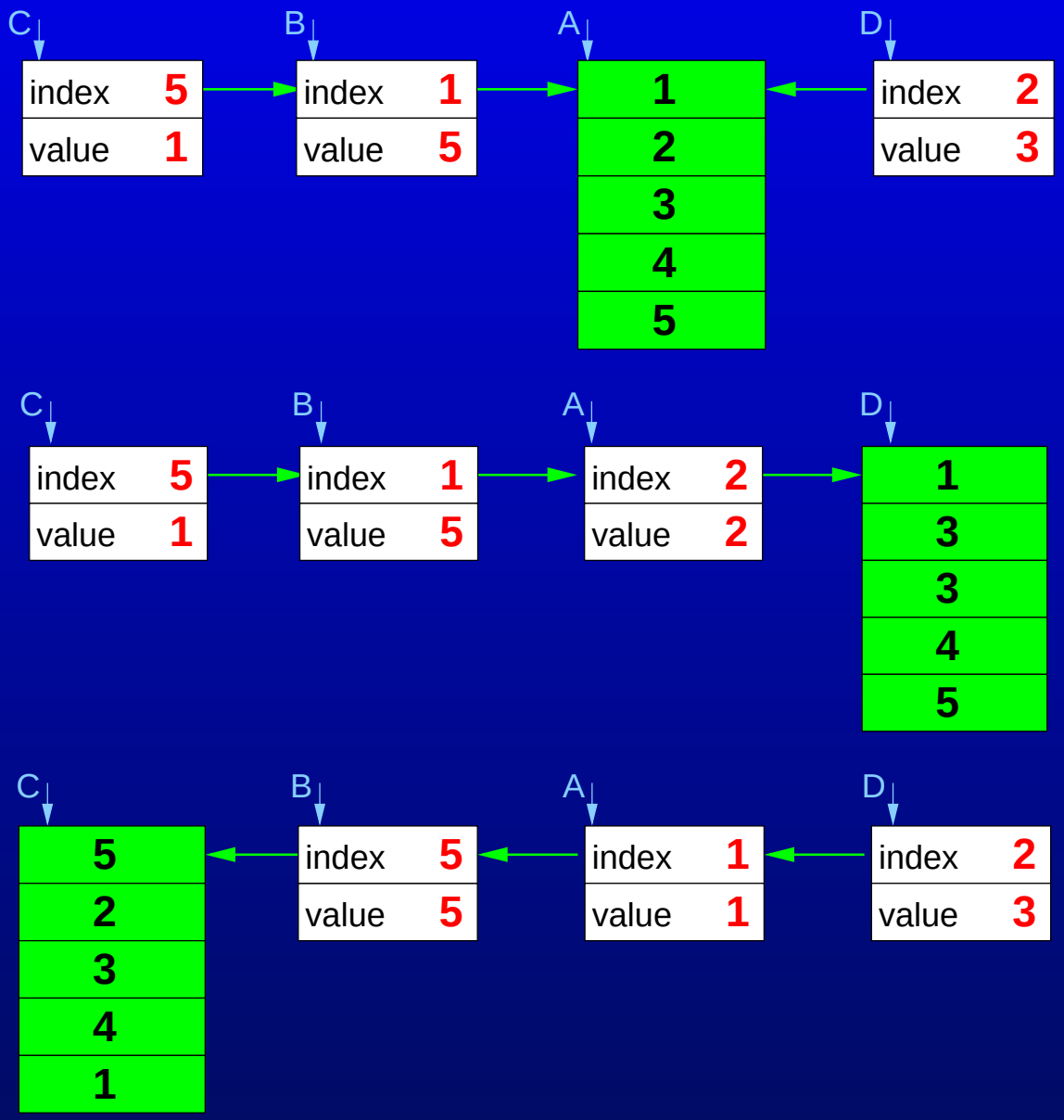
Functional Arrays using
Shallow Binding



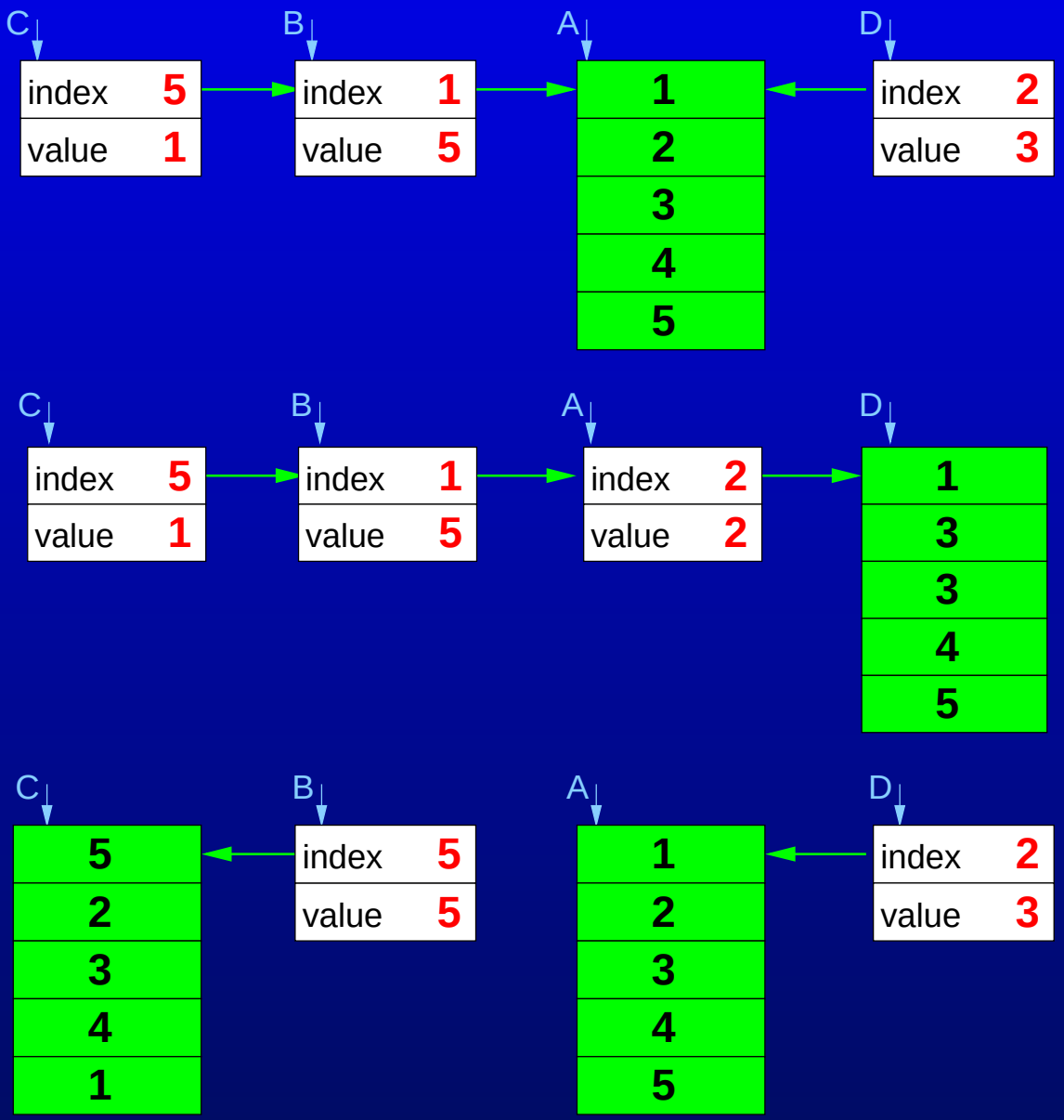
Functional Arrays using
Shallow Binding



Functional Arrays using
Shallow Binding

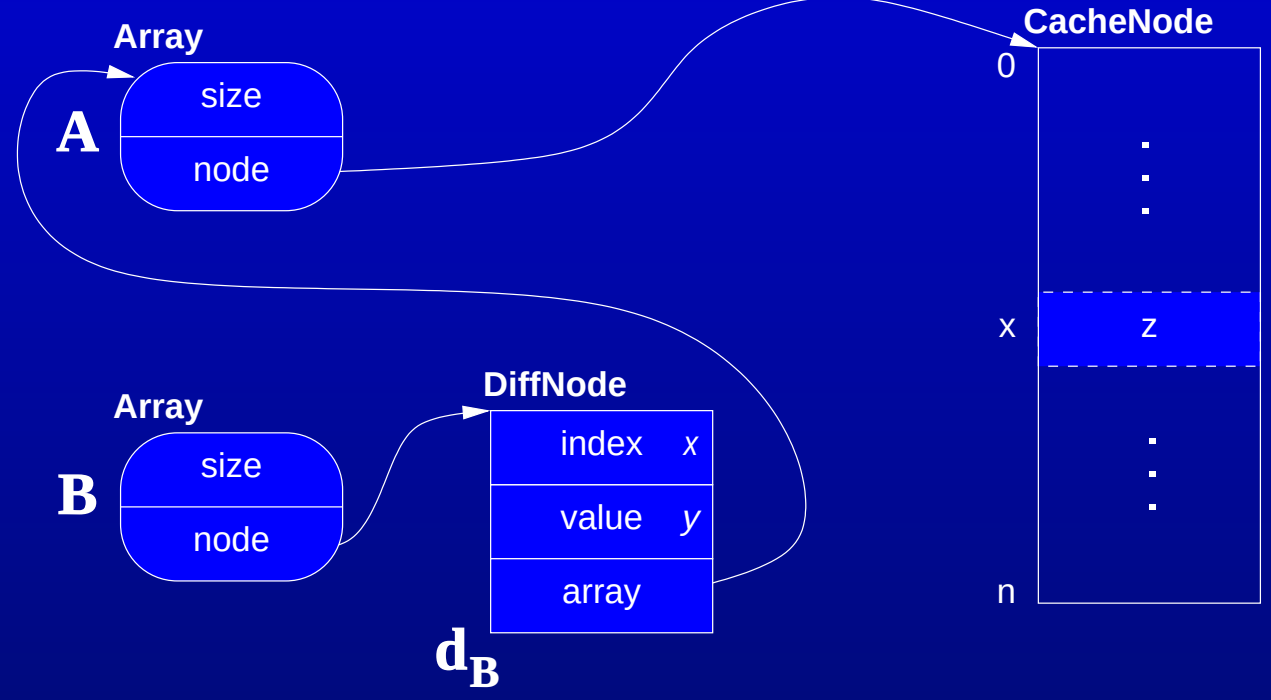


Functional Arrays using
Shallow Binding



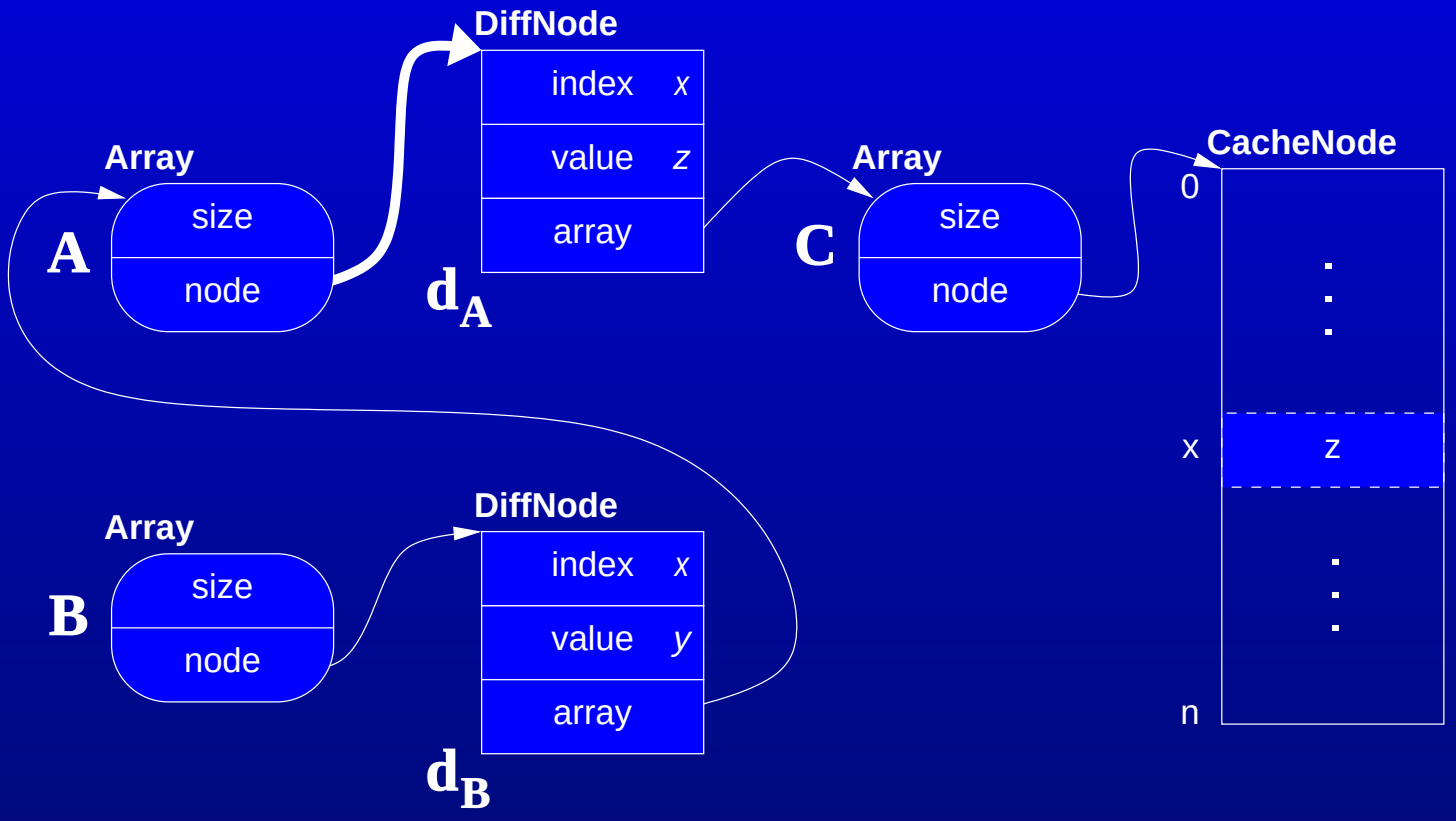
Lock-free Rotations

Unique pointer to cache acts as reservation.



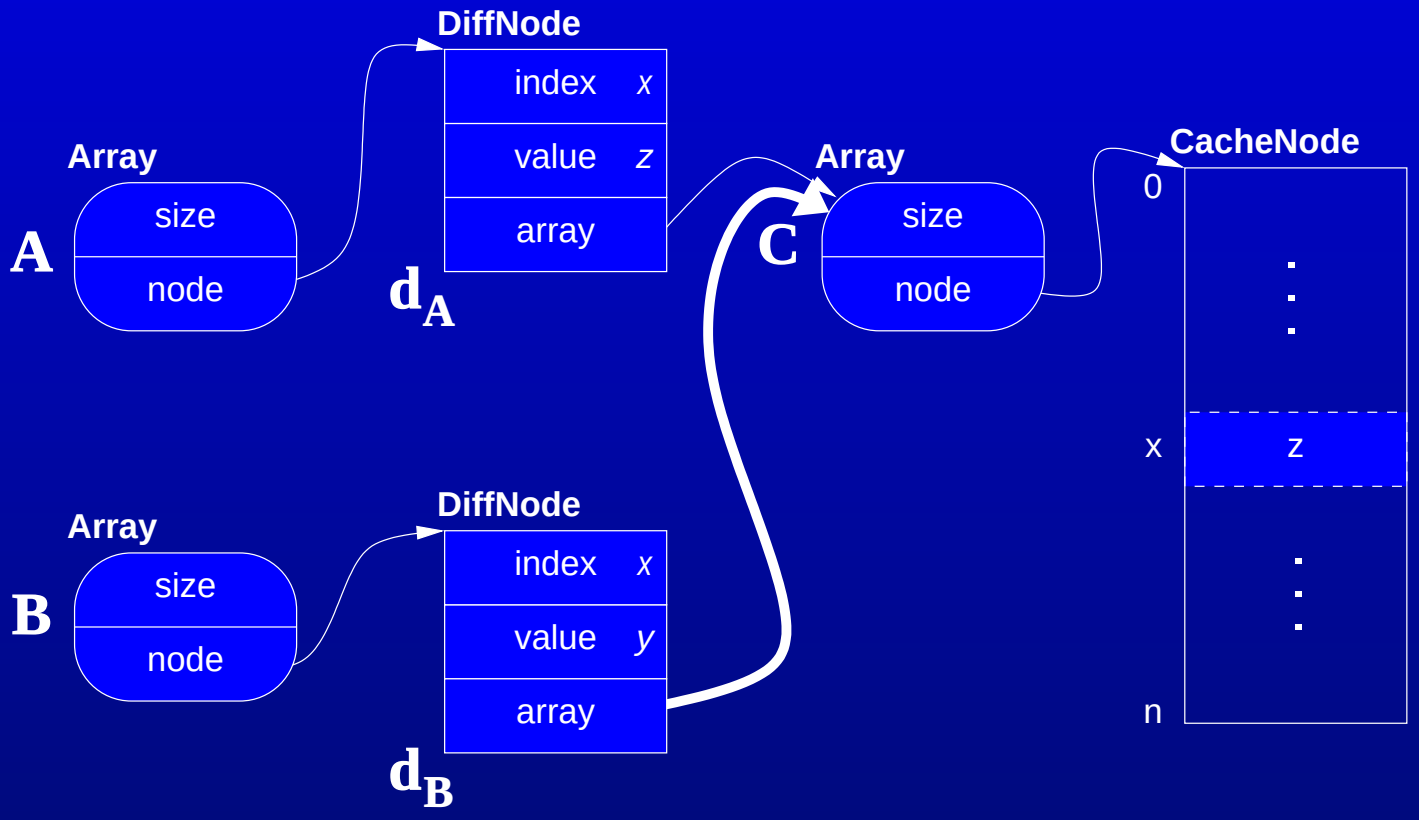
Lock-free Rotations

Unique pointer to cache acts as reservation.



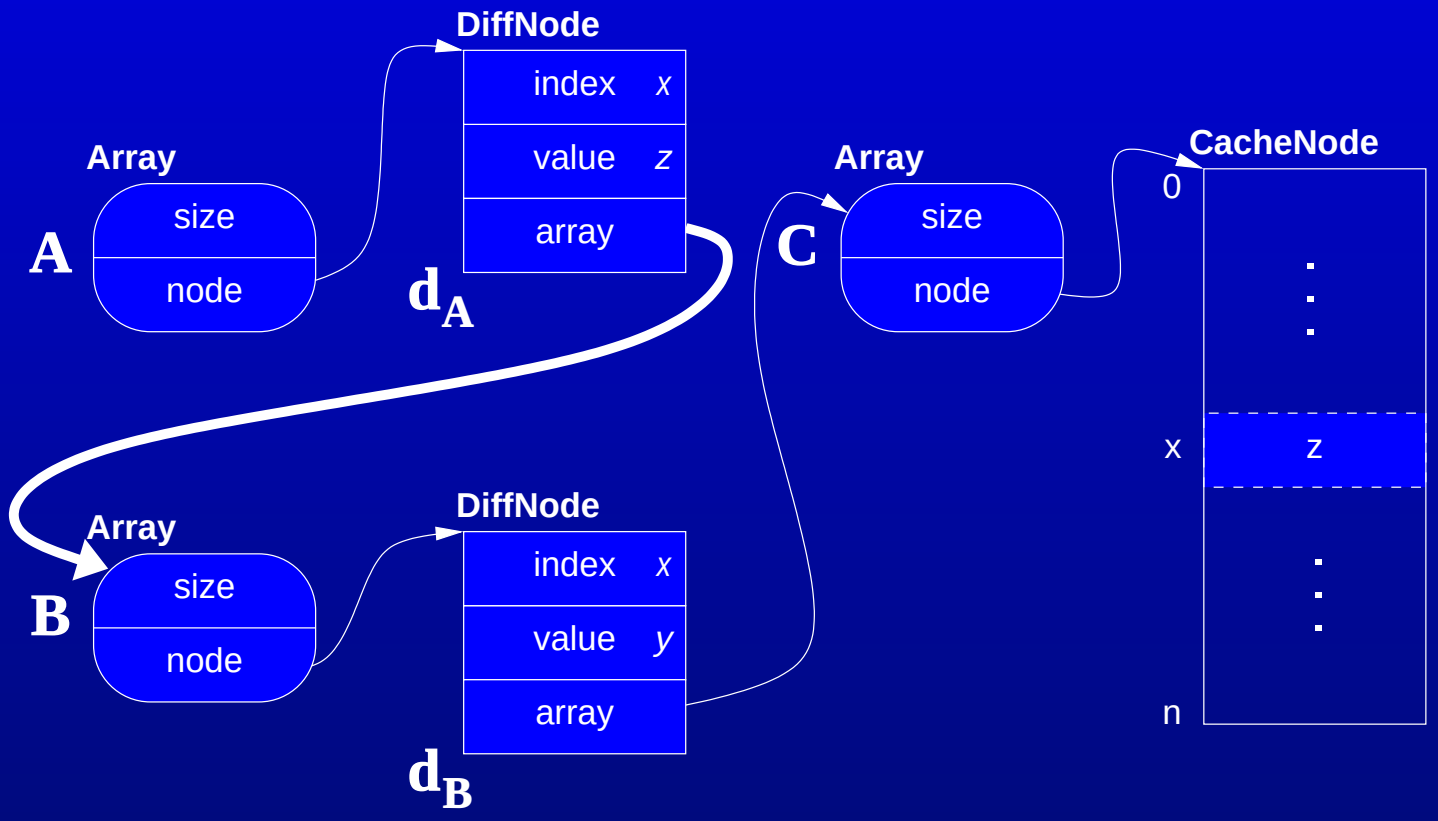
Lock-free Rotations

Unique pointer to cache acts as reservation.



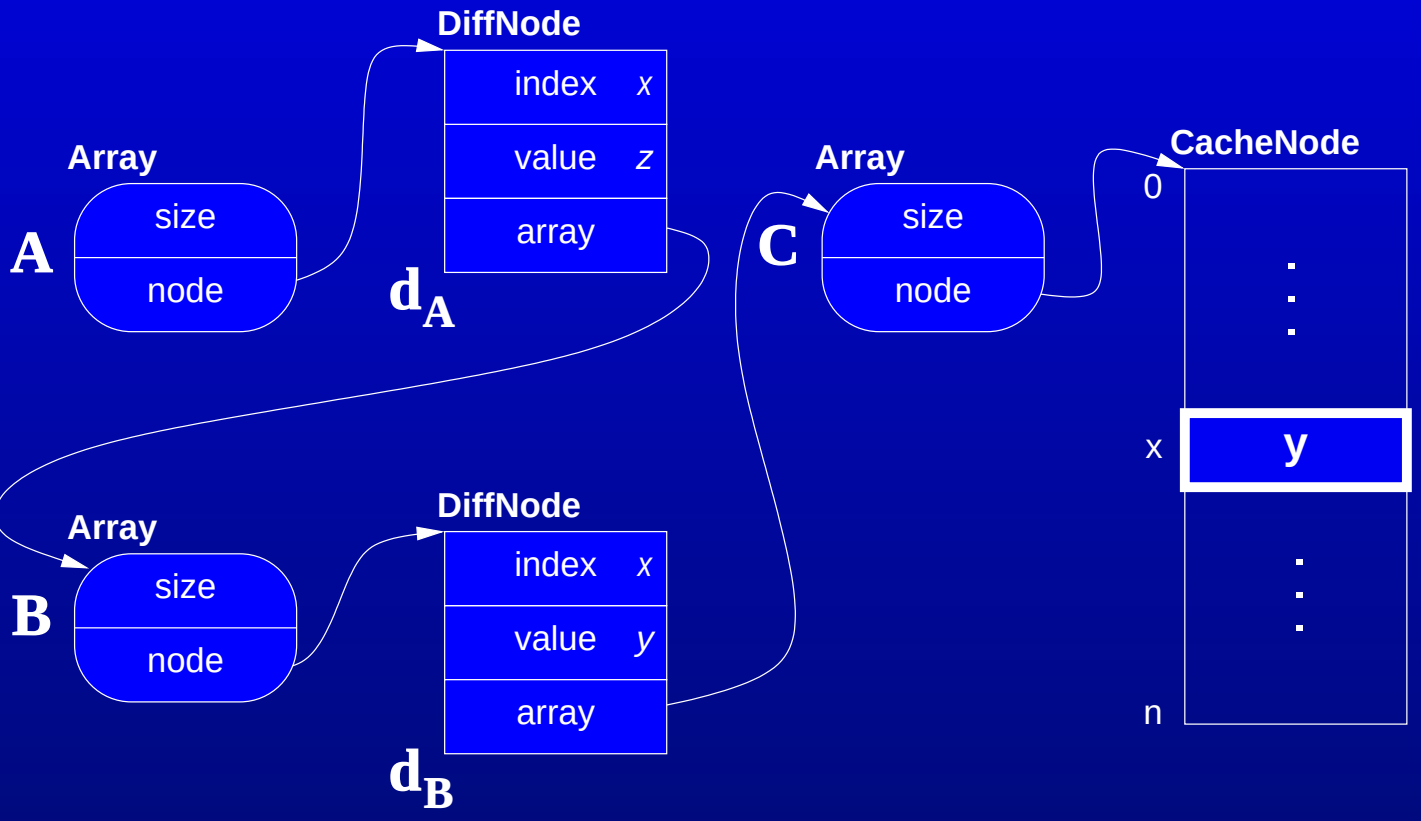
Lock-free Rotations

Unique pointer to cache acts as reservation.



Lock-free Rotations

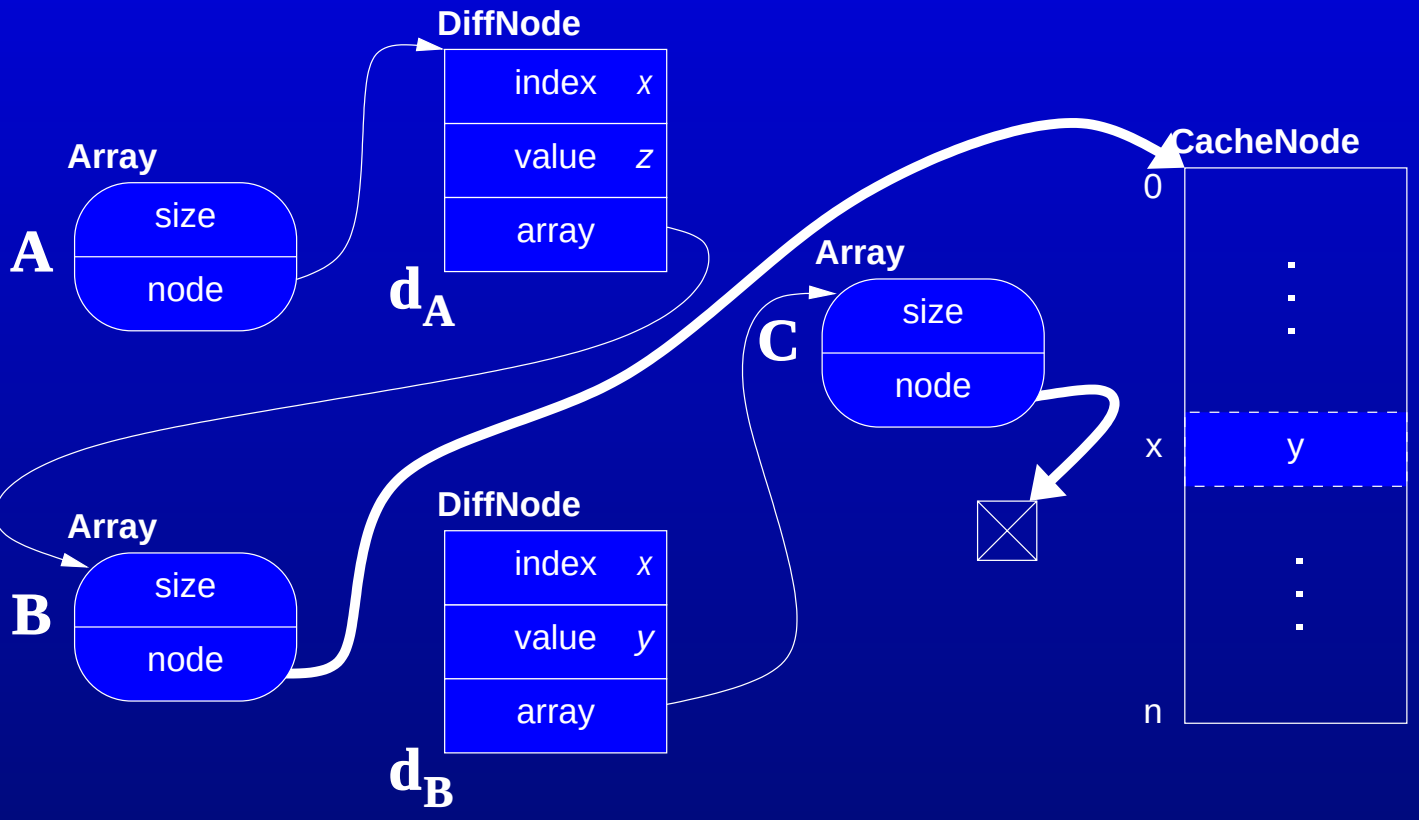
Unique pointer to cache acts as reservation.



DCAS

Lock-free Rotations

Unique pointer to cache acts as reservation.



DCAS

A Few Optimizations

- Use hardware small-transaction support to implement rotations
- Naïve functional arrays for small objects
- Only use synchronization protocol for shared data

Outline

- Survey prior non-blocking O-O Operating Systems:
  - Synthesis [Massalin and Pu 1991]
  - Cache Kernel [Greenwald and Cheriton 1996]
- A more general approach:
  - Language support for synchronization
  - Functional Arrays
  - Single-object protocol
  - Multiple-object protocol
  - Lock-free functional arrays
- Assessment and conclusions

Assessment and conclusions

- Surveyed Synthesis and Cache Kernel
  - Ad hoc implementations are hard to get right.
  - Version-number scheme is better, but very limited.
- Presented language design for non-blocking object-oriented operating system.
  - Novel feature: compiler-supported non-blocking `atomic` regions.
- Presented algorithms for implementing non-blocking `atomic` regions in O-O languages
  - Avoids “large object” problems.
  - Good complexity bounds, due to fast functional arrays.

The Graveyard Of Unused Slides follows this point.

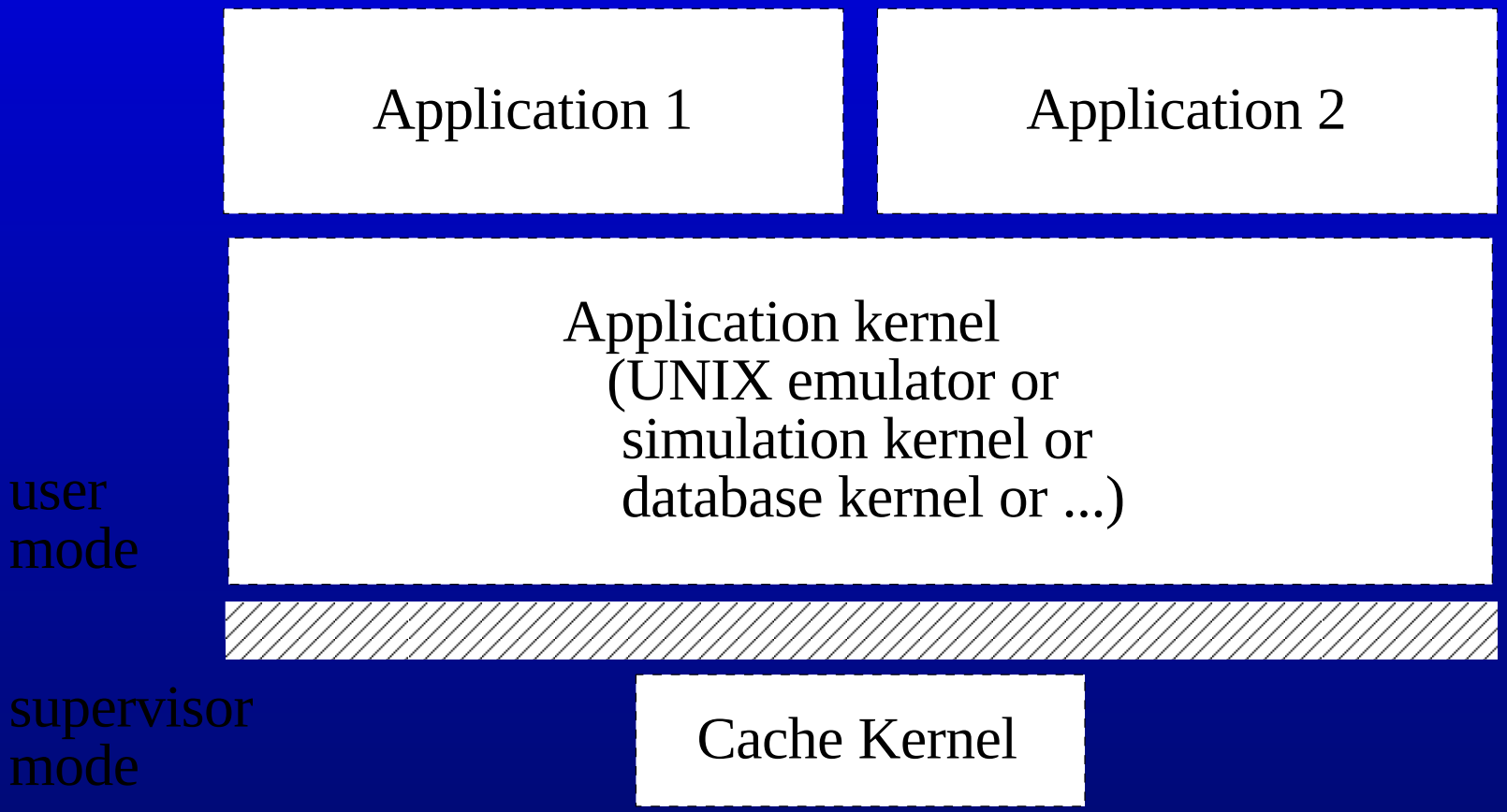
Blocking Synchronization

- Spin-locks
  - Processor runs in tight loop while waiting to enter a critical region.
  - Cheap, but wastes processor cycles.
- Semaphores
  - Maintain a waiting queue of blocked processes.
  - Queue maintenance and semaphore operations expensive.
- Hybrids
  - Spin-lock for short time periods.
  - Semaphore for long waits.

Non-blocking Synchronization

- Wait-free
  - Any process can complete any operation in finite # of steps, regardless of activities of other processes.
  - “Recursive helping.”
- Lock-free
  - *Some* process will complete in a finite # of steps.
  - Allows starvation.
- Obstruction-free
  - Processes will always complete if executed in isolation.
  - Contention can halt all progress indefinitely.
- Other (Lamport, . . .)

V++/Cache Kernel structure



Account example

```
class Account {

    int balance = 0;

    synchronized int deposit(int amt) {
        int t = this.balance;
        t = t + amt;
        this.balance = t;
        return t;
    }

    synchronized int readBalance() {
        return this.balance;
    }

    synchronized int withdraw(int amt) {
        int t = this.balance;
        t = t - amt;
        this.balance = t;
```

Optimistic parallelism

```
for (...)
  optimistically {
    ...do an iteration ...
  }
```

```
conquer(A[n], n) {
  ...
  optimistic spawn
    conquer(A, n/2);
  optimistic spawn
    conquer(A+n/2, n-n/2);
}
```

Programmer notes that the iterations or spawns are *expected* to be independent. Iff there are dynamic dependencies, the computations are serialized.