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

Notes

Nothing should be said on the title slide.

Our Goal

The design of a *language to support*

Object-Oriented

Non-Blocking

Operating Systems

Notes

In this talk we will explore the design of an object-oriented non-blocking operating system. @
More precisely, we will present language support the construction of 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.

Notes

I'm interested in object-oriented operating systems in particular for reasons of safety, ease, and performance.

Recent OS research has revealed the extent to which software protection mechanisms can efficiently replace hardware or OS-mediated protection. This promises to further reduce the size of microkernels.

Object encapsulation and strong interfaces allow clean composition, so that interaction between objects is well-understood, and promise a uniform synchronization model, which we will discuss in the 2nd half of this talk.

Finally, strong encapsulation enables aggressive specialization, which has been shown to substantially increase performance. The natural locality provided by objects will also enable good performance for the object-oriented synchronization algorithm I will present in the latter half of the talk.

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.

Notes

A non-blocking operating system is immune to many deadlocks which artificially couple aspects of a system design and which otherwise require careful bookkeeping to avoid.

Further, non-blocking synchronization allows us to avoid wasting parallelism, as when a processor must wait for a lock held by another processor which is currently working on an unrelated task. In the worse cases this can lead to convoys of tasks waiting on the same locks, artificially serializing the work.

Blocking mechanisms to avoid idling processors, like semaphores, often have high costs which developers don't wish to pay.

Finally, in real-time systems, non-blocking implementations can avoid priority inversion, where a low-priority task holding a lock delays a higher-priority task.

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

Notes

We will begin this talk by surveying two existing object-oriented operating systems: Synthesis, by Massalin and Pu, and the Cache Kernel, by Greenwald and Cheriton. We will look at the interaction of their object system and synchronization mechanism, and point out errors and limitations as we find them.

We will then propose a more general approach to incorporating non-blocking synchronization into an object-oriented operating system, based on language-level support. We will build our proposal in parts, first introducing the functional arrays which it is based on. We will describe a protocol valid for synchronizing operations on single objects, then show the need for multiple object protocols and describe one. We'll complete our proposal by offering an implementation of the lock-free functional arrays used in the protocols.

We will then assess our proposal and offer conclusions.

We will begin by describing the Synthesis kernel.

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.

Notes

Synthesis was the first operating system to attempt to use only non-blocking synchronization in its implementation. Synthesis' primary goal, however, was to explore runtime specialization for high performance. Strong object encapsulation was found to be vital for the creation of multiple alternate specialized implementations of key functionality; the cost of the extra abstraction was aggressively inlined away by the runtime code generator. Massalin and Pu could not find a high-level language with the necessary support for runtime code generation and specialization, so they implemented Synthesis—including an object system for what they called quajects—directly in macro assembler for the 680x0.

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

Notes

Synthesis used quajects to represent the basic operating system objects: threads, memory segments, symbol tables, and data channels. It used three of these quaject interfaces for all synchronization in the system, which was mediated by either a LIFO stack, a FIFO queue, or a linked list.

Variant implementations of these quajects catered to the exact form of synchronization required—for example, one FIFO queue implementation targetted single-producer multiple-consumer synchronization, and another targeted the more general multiple-consumer multiple-producer case. Synthesis' runtime specialization engine selected the appropriate implementation and inlined away the abstraction cost of using these primitives.

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.

Notes

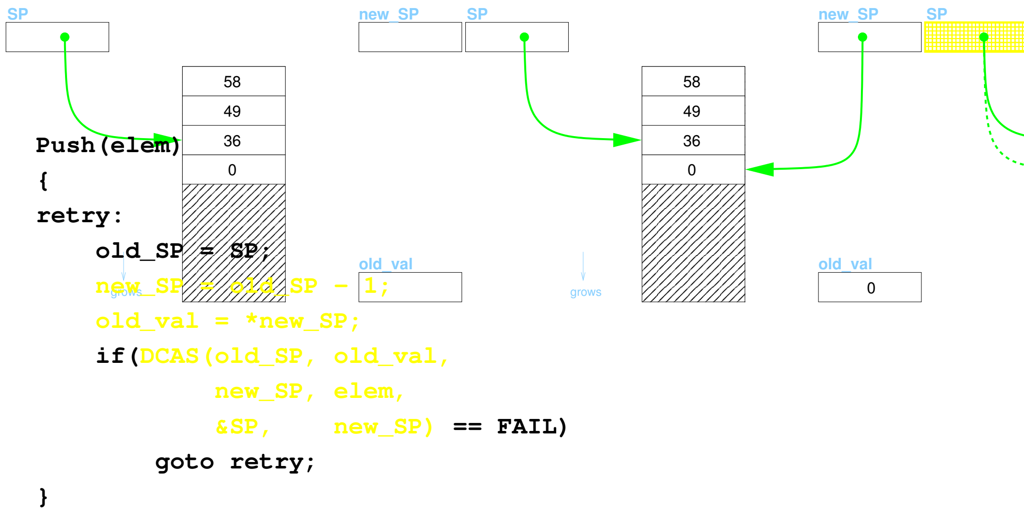
I found three synchronization errors in the published lock-free algorithms used inside Synthesis. One is an ABA problem in a last-in first-out stack which might cause a consumer to end up with stale data.

Another race was found in the multiple-producer single-consumer first-in first-out queue implementation. In this case there is the possibility that the fault lies in the published description, rather than the implementation.

Finally, quaject composition to emulate blocking i/o contains an interesting race condition which is not mentioned in any of the Synthesis publications.

We'll look at these in order.

LIFO stack: Push



Notes

We'll begin by studying the (correct) implementation of a non-blocking push onto a last-in first-out stack. We want to atomically update the stack pointer and the element at the top of the stack, while ensuring that no other simultaneous changes to the stack pointer occur. If the stack pointer were to be concurrently modified, we'd end up writing to the wrong location, possibly destroying information in the stack.

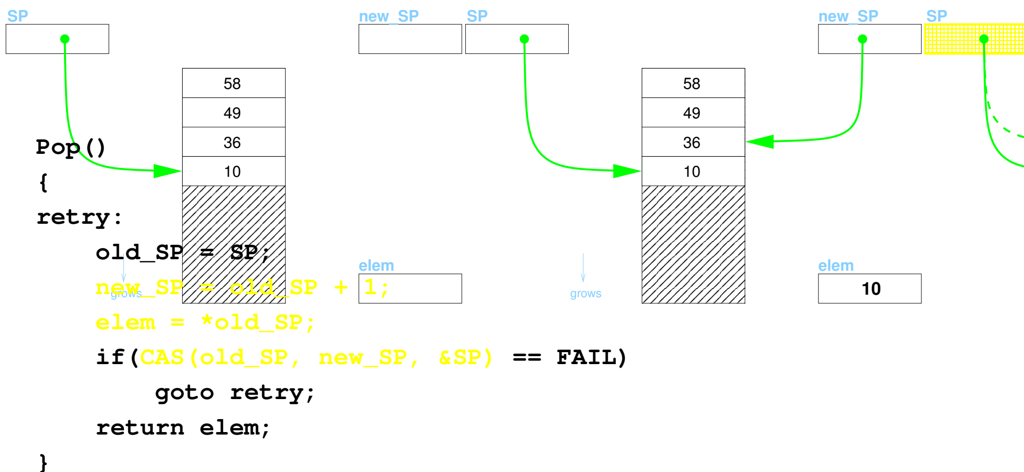
So what we'd like to do is atomically compare the stack pointer and, if the comparision succeeds, store a new stack pointer value and a new top-of-stack value.

The primitive available on the 680x0 to do this is called DCAS: double-compare and swap. It allows us to do *two* comparisons, and if both succeed, do *two* stores to the locations compared.

To use DCAS, we must @ read the old/invalid data at the top of the stack, so that we can @ use it as input to the "extra" second compare gating the @ store to the new top-of-stack.

This is safe. If the DCAS fails — likely because someone has modified the stack pointer by pushing or popping after our read and before the DCAS — then we simply loop to retry the operation. A robust implementation would include a backoff strategy as well.

LIFO stack: Pop



Notes

Now we'll look at Pop. Here we'd like to pop off the element '10' which we just pushed. @ We first read the stack pointer and the element at the top of the stack. @ We then, keeping an eye on the stack pointer for concurrent modifications, @ atomically increment the stack pointer using Compare-And-Swap.

But wait! There's a problem. @ Let's go back to just before the CAS. What if someone else @ snuck in a pop of our element (10) and then @ pushed some other element on?

Would our CAS succeed?

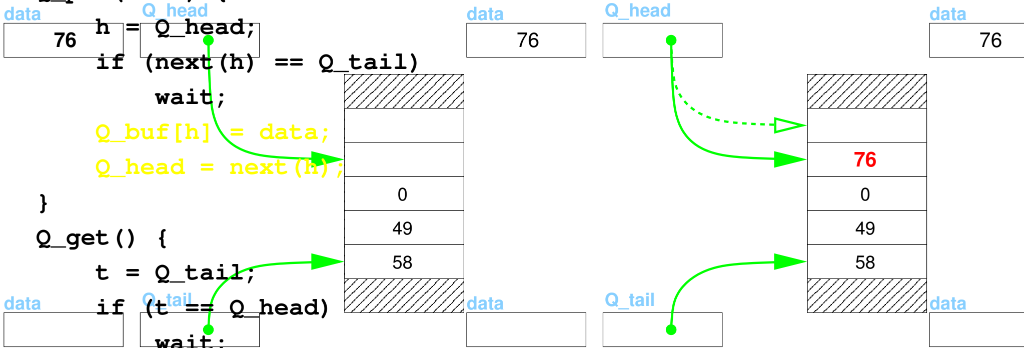
Yes! Even though we're pulling off the wrong element (10, instead of 99).

This is the code as published by Massalin and Pu. The solution is to use a DCAS instruction here, so that our operation doesn't succeed unless the top of stack remains unchanged. In general, this is known as an ABA problem, because CAS can't tell that the stack pointer's been changed to B from A if it is later reset to A again.

There might be mitigating factors for this error in the Synthesis implementation: perhaps separate threads were always bound to push and pop, so that there would never be a matched pair of push and pop conflicting with an operation. Synthesis also ran on a dual-processor machine with roughly matched clock speeds, so it might be unlikely for one processor to finish a pop *and* a push while another processor was still working on the pop.

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;
}
```

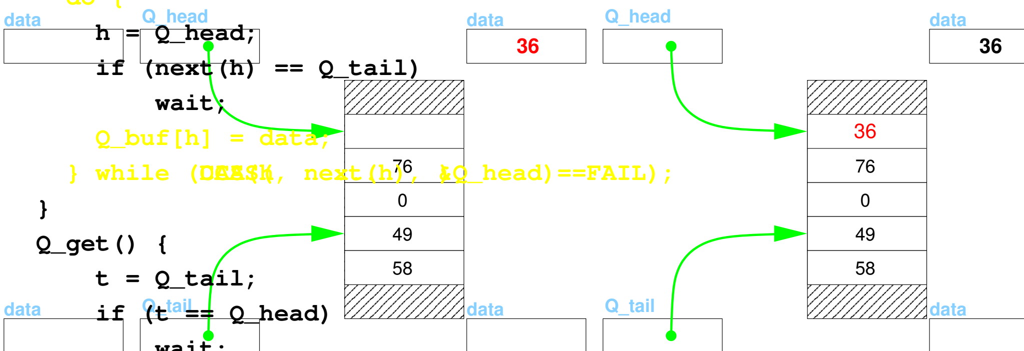


Notes

Let's move on to Synthesis' FIFO queue implementation. Synthesis actually had four variants of the FIFO queue; we'll start with the single-producer/single-consumer version. This implementation requires only atomic reads and writes (and sequential consistency).
`Q_buf` is a circular bugger of length `Q_size`.
To put an element in the queue, we need only check for overflow, @ store the new value, and @ increment the head pointer. The order of these operations is important.
The remove an element from the queue, we check for underflow, @ read the data, and @ increment the tail.
Because the head is not incremented until after the location it points to is valid, and the tail is checked against head before a value is read out, we will never read inconsistent data.
But this data structure only works if there is at most one consumer and at most one producer.

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;
}
```



Notes

@ The multiple-producer/single-consumer variant is identical except for a loop and a compare-and-swap replacing the last store. Again, we @ store the data first. Now we @ atomically update the head. If another concurrent producer has beaten us to this point, the head will already be incremented causing our compare to fail and notify us that we need to retry.
But wait: what if the concurrent producer @ got as far as writing their data to the buffer (overwriting ours) but not as far as the CAS? Now our CAS will still succeed — `Q_head` is unchanged — but the buffer has been corrupted and our datum lost.
@ The solution is to use a Double Compare-and-Swap instead of a simple CAS to protect the element we wrote as well as the head.
This queue is described in the published work on Synthesis as having "a single compare-and-swap at the end", which seems to describe the erroneous implementation. It is possible, however, that the fault is sloppy writing, not incorrect implementation. Still, this illustrates how fragile these algorithms are.

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**	⟹	Q_put	Q_get	⟸	*send-complete* interrupt
callback	**suspend**	⟸	Q_full	Q_empty	⟹	turn off *send-complete*
callback	**resume**	⟸	Q_full-1	Q_empty-1	⟹	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.
- Intended to emulate blocking I/O when callbacks

Notes

Our third race from the Synthesis implementations is found in the quaject composition mechanism, or more precisely, in how queue quajects and callbacks are intended to be used to implement blocking I/O.
Here we show a ByteQueue quaject hooked up to a consumer (a hardware UART) and a producer (a user thread). Calls to `Q_put` return immediately as long as queue is not full, otherwise the `Q_full` callback is invoked. When the queue later drains some the `Q_full-1` callback is invoked. Likewise for `Q_get` and `Q_empty`.
@ The callback mechanism is intended to allow the implementation of both synchronous (blocking) and asynchronous I/O interfaces. For synchronous I/O, the callbacks are hooked up to invoke thread suspend and resume. BUT what if queue empties between invocation of `Q_full` and the actual thread suspend? We'll try to resume a thread which isn't suspended, and then put the thread to sleep forever, waiting for a `Q_full-1` signal which will never arrive.
@ The traditional solution is to disable signals/interrupts in this critical region. However Synthesis V.1 doesn't use mutual exclusion, and so it must do something more clever. We might atomically add the thread to the "suspended" list with a compare-and-swap conditional on the queue still being full. But our list manipulation code typically already requires DCAS in order to protect against concurrent mutations. *Where will we find the extra atomic compare needed?* This question is not answered in any of the published work on Synthesis.

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.

Notes

We've seen three potential races in Synthesis, which used hand-coded non-blocking implementations of selected data structure objects for synchronization. Now we'll move on to the Cache Kernel, which uses a slightly more principled mechanism for implementing its synchronization primitives.
The Cache Kernel was a micro-microkernel with only three exported object types and a single notification mechanism: asynchronous signals. Lock-free synchronization was used not mainly for performance, as with Synthesis, but to manage the pervasive asynchrony without introducing undesirable coupling into the highly modular kernel. Further, lock-free synchronization allowed the kernel designers *carte blanche* to push system calls and OS functionality into user mode, without concern that a terminated user thread might orphan locks and deadlock the system.

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.

Notes

Unlike the ad hoc implementations in Synthesis, the Cache Kernel uses a consistent strategy for implementing non-blocking data structures. Each data structure has a version number, which counts modifications made to the data structure. One-word changes could be made on the data structure, for example, swinging a `next` pointer in a singly-linked list to insert or remove a node. Performing a one-word change begins by reading the version number. The data structure is then read to compute the appropriate change, and a DCAS instruction is used to atomically apply the change and increment the version number if and only if the data structure has remained unchanged during the entire operation (since the first read of the version number).

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;
}
```


Notes

As a concrete example, this is the Cache Kernel algorithm for deleting a node from a linked list. The first thing we do is read and store the current version number. We then traverse the list, looking for `elt`, the node we would like to delete. When we find it, we use DCAS to increment the version number and swing `p->next` from `elt` to `elt->next`.

List deletion has an exception case, too, when the node we would like to delete can not be found in the list. Before we take that exit, we double-check that the list has not been modified during the time we were looking for `elt`. If it has, then we retry the search: it may have been added while we were looking, or we may have wandered down a blind alley due to concurrent modifications. Remember that all mutations to the list occur atomically with increments of `list->version`, so there's no race here.

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.

Notes

The version-number-and-DCAS mechanism works well for small changes to small data structures, but doesn't scale well past that.

First of all, the version number protects the *entire* data structure. The larger the data structure, the more likely it is that your operation will be interrupted by a concurrent operation *somewhere* in that data structure. The Cache Kernel works around this problem by making lists of lists and hash tables of lists, so that each version number protects a smaller amount of data.

More problematic is the fact that only one-word mutations are allowed. The Cache Kernel architects went out of their way to cram all data which might need to be updated atomically into a single word, but sometimes that is impossible.

One solution is to copy-and-swap. Represent the large object by a pointer to it, and atomically update *just the pointer* to point to a new updated version. But copying the object every time a change is made gets expensive.

The Cache Kernel needs to resort to a “remove, mutate, and add” strategy to achieve acceptable performance. An object is removed from its containing data structure while it is mutated. This allows data structures to drift out-of-sync with the “real world”, and higher-level time-out and retry mechanisms must be layered on to compensate. These mechanisms have to be designed carefully to prevent deadlock! [the search mechanism can prevent the completion of the “mutate and read”]

Problems with large objects are typical of general schemes for non-blocking synchronization.

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

Notes

After reviewing the approaches used by Synthesis and the Cache Kernel to implement non-blocking synchronization, and assessing their limitations, we are ready to present our own approach to providing non-blocking synchronization to an operating system implementation.

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.

Notes

What have we learned from Synthesis and the Cache Kernel? First, that hand-coded implementations are hard to get correct! And these ad hoc implementations are going to be limited: we only have three synchronized data structures in Synthesis. The Cache Kernel has more, due to its more general methodology, but you are still forced to refactor your data structures to fit them into the limits of the version number scheme.

Further, these ad hoc implementations are brittle. You can add notes to the start of a linked list, but not in the middle. You can atomically update one word of a structure, but not two. When you resort to copy-and-swap to get larger atomic actions, your performance suddenly becomes $O(n)$.

Our solution is to integrate a general protocol for non-blocking synchronization into the language. It will be be correct, it will work for any data structure you care to write, and most importantly, its performance will be good in the face of large changes and large objects.

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!

Notes

Let's take a step backward and look at how synchronization is typically integrated with object-oriented languages. Modern o-o languages trace their treatment of synchronization back to Emerald, which introduced *monitor synchronization*. In this scheme, every object contains a monitor which can be used to enforce mutual exclusion, as well as for signalling. Certain methods are monitored, which means they take the object's mutex at entry and release it at exit. Shared variables can only be accessed by monitored methods, which (technically) eliminates data races. [Although Java doesn't enforce this safety constraint.]

However: this may technically prevent races, but it is 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;
    }
}
```


Notes

The class A here, shows what monitor synchronization looks like in Java. The `synchronized` keyword indicates that this is a monitored method. Only one thread may be hold the monitor at a time, thus only one thread may be inside `inc()` at a time. This guarantees that the increment behaves as we expect: this is a correctly synchronized method.
But look at class B, which implements the same functionality. Note that the only access to shared variable `x` is inside the monitored `get()` and `set()` methods — but this code is not safe! If n threads call `inc()`, the shared variable x may be incremented any number between 1 to n times.

Atomic Blocks

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

- Traditionally, monitors associated with each object provide mutual exclusion between concurrent accesses to the object. Instead we provide an `atomic` block, and make linearizablity

Notes

We're going to approach synchronization in a slightly different manner. Instead of guaranteeing mutual exclusion within a monitored method, @ we would like to specify synchronization as *atomic blocks*, which guarantee that the enclosed operations will be perceived as atomic by all other threads. Eliminating the explicit lock parameter also prevents some errors in the use of monitors, especially when multiple objects are involved. We'll return to this point in a few slides.
Note that atomic blocks could be implemented with locks, but we are going to use an optimistic non-blocking implementation.

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

Notes

Here's the big picture: an object-oriented language designed for the implementation of non-blocking operating systems.
We're going to start with Java, for the usual reasons.
Java is missing some crucial low-level features, like the ability to write interrupt handlers or directly address memory-mapped I/O. JEPES (Java Execution Platform for Embedded Systems) addressed these issues for Java on embedded systems, mostly via assembly hooks. The Lisaac system had a far more elegant design.
We'd like to use software protection mechanisms, which will allow us to write a single-address-space OS and remove address spaces from the kernel. There is a lot of existing research on the topic.
The crucial and novel technique here is the provision of atomic operation blocks, which we will implement with non-blocking synchronization.

Functional Arrays

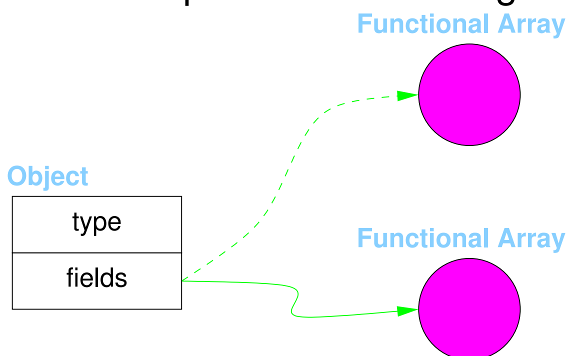
Our implementation of `atomic` blocks will be based on fast *functional arrays*.

- 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".

Notes

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.
- At start of `atomic` block load and remember

Notes

...
With a naive implementation of functional arrays, this would just be the “copy and swap” strategy of Herlihy, etc.

Problems with Multiple Objects

The case of the StringBuffer

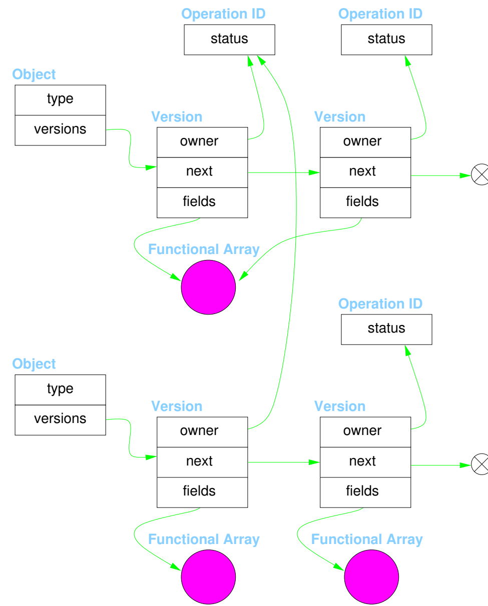
```
public final class StringBuffer ...{
    ...
    private int count;

    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(...)  { ...}
}
```


Notes

Of course we want to support operations on multiple objects. This can be tricky: see for example this code from the Java standard libraries. StringBuffer.append() is synchronized, and the operations on the parameter StringBuffer sb.length() and sb.getChars() operate atomically, but when the pieces are put together you have a race! The value for `len` we read atomically at the start of the method is not guaranteed to be the same as the length of the buffer at the point `sb.getChars()` is called.
Specifying that `StringBuffer.append()` is `atomic` is really the semantics we want. But now we have two objects, `this` and `sb`, both of whose updates must be atomic.

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*.
- At end of atomic block, attempt to set



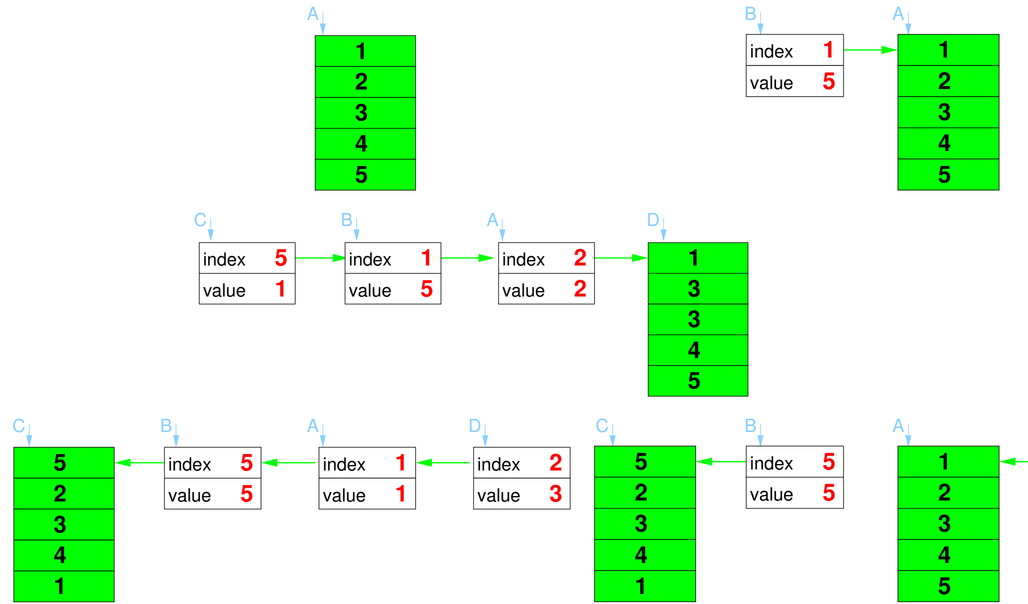
Notes

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

Notes

Functional Arrays using
Shallow Binding

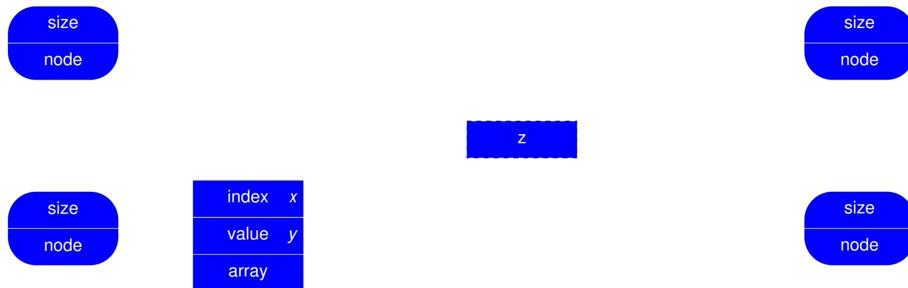


Notes

Now I'll present a lock-free fast functional array implementation. The scheme is based on trailers.

Lock-free Rotations

Unique pointer to cache acts as reservation.



DCAS

Notes

Let's look at how the rotations in the shallow-binding scheme are implemented lock-free.

A Few Optimizations

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

Notes

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

Notes

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.

Notes

The Graveyard Of Unused Slides follows this point.

Notes

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.

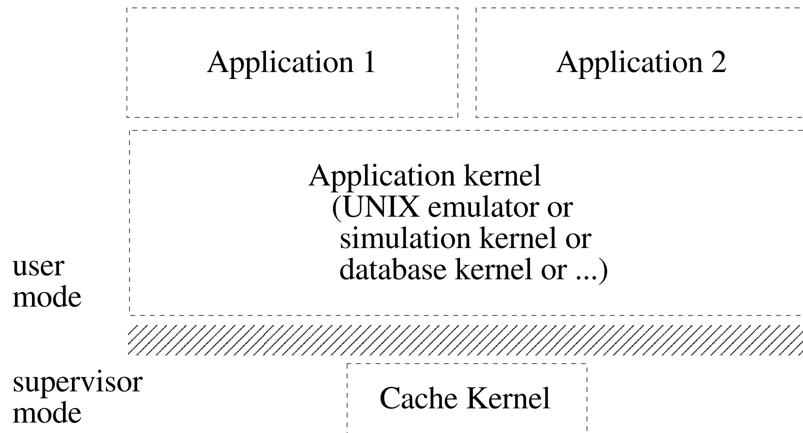
Notes

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, . . .)

Notes

V++/Cache Kernel structure



Notes

This diagram shows the Cache Kernel sitting under application kernels and applications. . . .

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;
```


Notes

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.

Notes

There are different ways multiple transactions can interact. We could allow only one active transaction at a time, only allow non-overlapping transactions, allow nested transactions, concurrent transactions, subsumed transactions, nested independent transactions, or other variations.

We'd like the investigate using this mechanism to allow a programmer to specify *optimistic* parallelism. This is much easier to make safe, although potentially just as hard to make fast.