Is the Optimism In Optimistic Concurrency Warranted?

Donald E. Porter,
Owen S. Hofmann,
and Emmett Witchel

Department of Computer Sciences
University of Texas At Austin

Optimism About Optimistic Concurrency

- Industry shift to multicore chips
- Renewed importance of parallel programming
- Optimistic concurrency can find more parallelism
  - How much can it improve my system?

Quantifying Potential of Optimistic Concurrency

- Build an optimistic system and measure
  - Current best option
  - Specific
- Methodology for assessing potential benefit and tuning opportunities

Key Questions

- How can optimistic concurrency help performance?
- How much does it help in practice?
- Will it help my existing lock-based system?
  - Methodology
  - Case Study

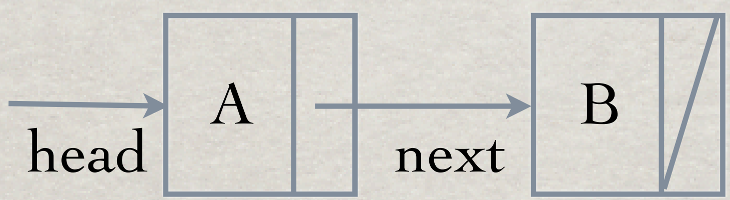
Linked List Example

Counter

```
lock(list.lock);
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
unlock(list.lock);
```

Modifier

```
lock(list.lock);
if(head.value == "A"){
  head.value = "Z";
}
unlock(list.lock);
```



Reads	Writes

Reads	Writes

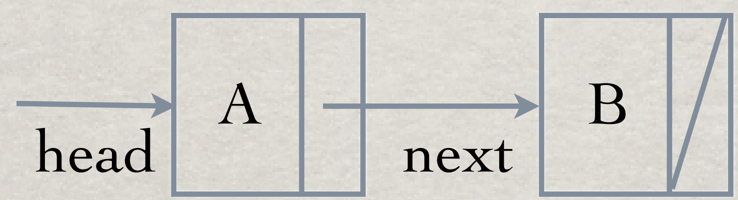
Linked List Example

Counter

```
lock(list.lock);
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
unlock(list.lock);
```

Modifier

```
lock(list.lock);
if(head.value == “A”){
  head.value = “Z”;
}
unlock(list.lock);
```



Reads	Writes

Reads	Writes

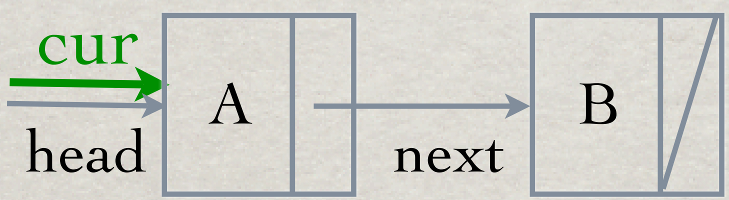
Linked List Example

Counter

```
lock(list.lock);
cur = head; ← Lock Acquire
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
unlock(list.lock);
```

Modifier

```
lock(list.lock); ← Busy Wait
if(head.value == "A"){
  head.value = "Z";
}
unlock(list.lock);
```



Reads	Writes
head	cur

Reads	Writes

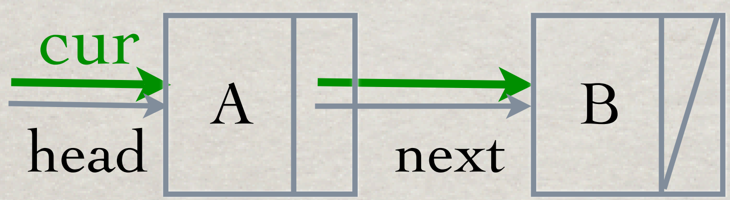
Linked List Example

Counter

```
lock(list.lock);
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
unlock(list.lock);
```

Modifier

```
lock(list.lock); ← Busy Wait
if(head.value == "A"){
  head.value = "Z";
}
unlock(list.lock);
```



Reads		Writes
head	cur	cur
node1.next		

Reads	Writes

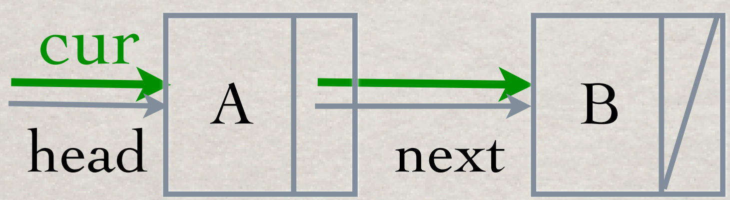
Linked List Example

Counter

```
lock(list.lock);
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
unlock(list.lock);
```

Modifier

```
lock(list.lock);  ← Busy Wait
if(head.value == "A"){
  head.value = "Z";
}
unlock(list.lock);
```



Reads		Writes
head	cur	cur
node1.next	count	count

Reads	Writes

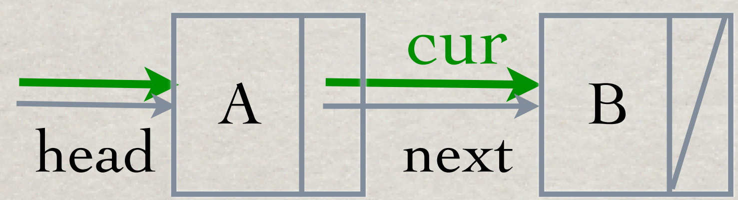
Linked List Example

Counter

```
lock(list.lock);
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
unlock(list.lock);
```

Modifier

```
lock(list.lock);  ← Busy Wait
if(head.value == "A"){
  head.value = "Z";
}
unlock(list.lock);
```



Reads		Writes
head	cur	cur
node1.next	count	count

Reads	Writes

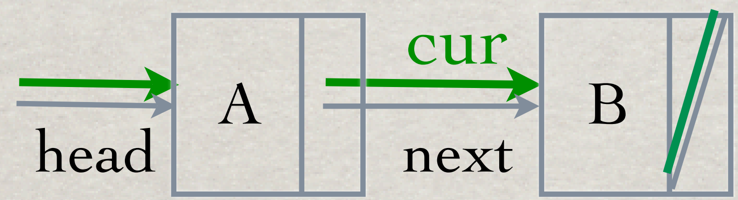
Linked List Example

Counter

```
lock(list.lock);
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
unlock(list.lock);
```

Modifier

```
lock(list.lock); ← Busy Wait
if(head.value == "A"){
  head.value = "Z";
}
unlock(list.lock);
```



Reads		Writes
head	cur	cur
node1.next	count	count
node2.next		

Reads	Writes

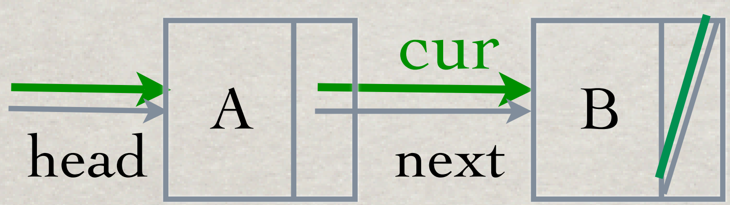
Linked List Example

Counter

```
lock(list.lock);
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
unlock(list.lock);
```

Modifier

```
lock(list.lock); ← Busy Wait
if(head.value == "A"){
  head.value = "Z";
}
unlock(list.lock);
```



Reads		Writes
head	cur	cur
node1.next	count	count
node2.next		

Reads	Writes

Linked List Example

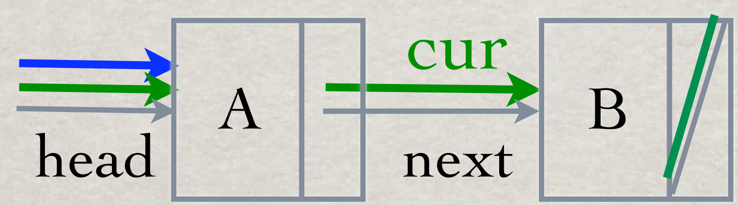
Counter

```
lock(list.lock);
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
unlock(list.lock);
```

Lock Acquire

Modifier

```
lock(list.lock);
if(head.value == "A"){
  head.value = "Z";
}
unlock(list.lock);
```



Reads		Writes
head	cur	cur
node1.next	count	count
node2.next		

Reads	Writes
head	
node1.value	

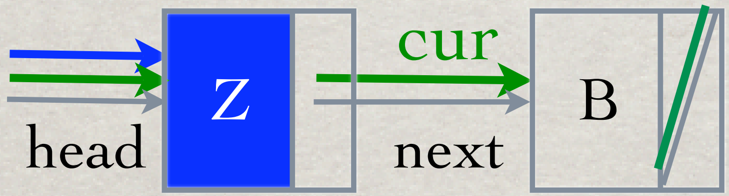
Linked List Example

Counter

```
lock(list.lock);
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
unlock(list.lock);
```

Modifier

```
lock(list.lock);
if(head.value == "A"){
  head.value = "Z";
}
unlock(list.lock);
```



Reads		Writes
head	cur	cur
node1.next	count	count
node2.next		

Reads	Writes
head node1.value	node1.value

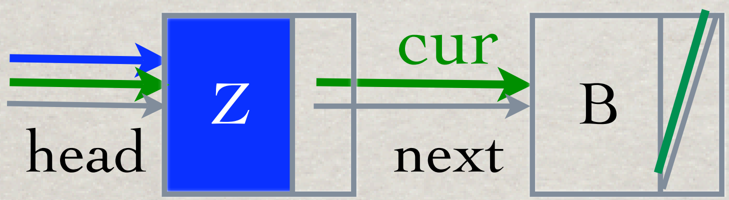
Linked List Example

Counter

```
lock(list.lock);
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
unlock(list.lock);
```

Modifier

```
lock(list.lock);
if(head.value == "A"){
  head.value = "Z";
}
unlock(list.lock);
```



Reads		Writes
head	cur	cur
node1.next	count	count
node2.next		

Reads	Writes
head	node1.value
node1.value	

Locks are Conservative

- Modifier could have safely executed concurrently with Counter
- Verified by comparing the memory locations accessed

Counter

Reads		Writes
head	cur	cur
node1.next	count	count
node2.next		

Modifier

Reads	Writes
head	node1.value
node1.value	

Optimistic Concurrency

- Can eliminate unnecessary serialization
  - Optimistically modify shared data
  - Detect unsafe accesses
  - Rollback and retry on conflict

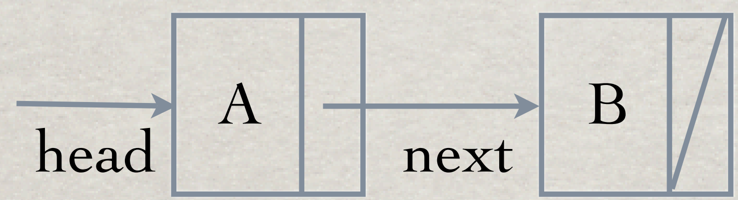
Optimistic Linked List

Counter

```
lock(list.lock);
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
unlock(list.lock);
```

Modifier

```
lock(list.lock);
if(head.value == “A”){
  head.value = “Z”;
}
unlock(list.lock);
```



Reads	Writes

Reads	Writes

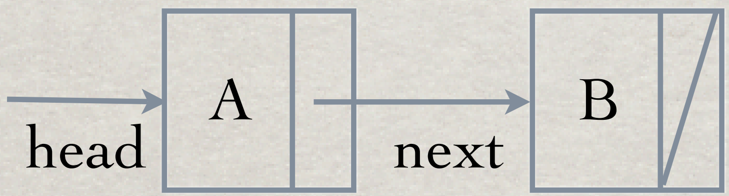
Optimistic Linked List

Counter

```
begin critical section;
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
end critical section;
```

Modifier

```
begin critical section;
if(head.value == "A"){
  head.value = "Z";
}
end critical section;
```



Reads	Writes

Reads	Writes

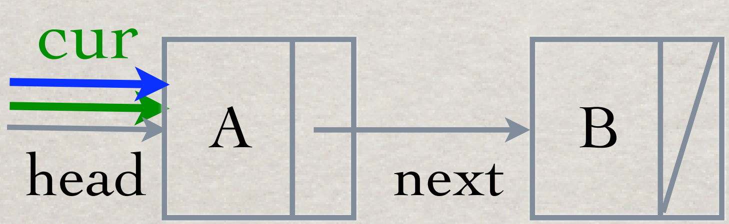
Optimistic Linked List

Counter

```
begin critical section;
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
end critical section;
```

Modifier

```
begin critical section;
if(head.value == “A”){
  head.value = “Z”;
}
end critical section;
```



Reads	Writes
head	cur

Reads	Writes
head node1.value	

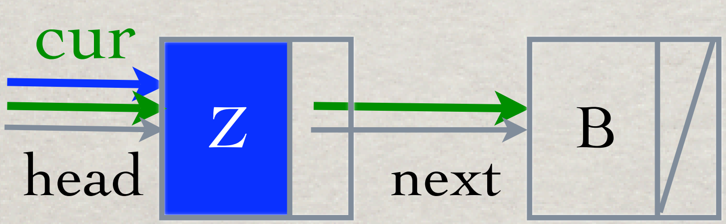
Optimistic Linked List

<u>Counter</u>

```
begin critical section;
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
end critical section;
```

<u>Modifier</u>

```
begin critical section;
if(head.value == "A"){
  head.value = "Z";
}
end critical section;
```



Reads		Writes
head	cur	cur
node1.next		

Reads	Writes
head node1.value	node1.value

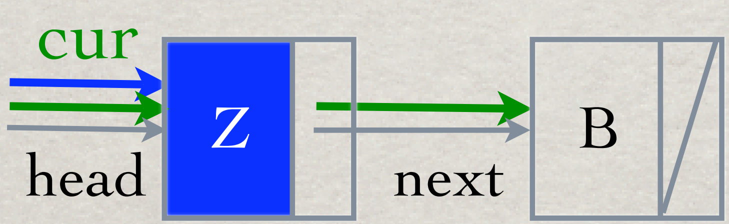
Optimistic Linked List

Counter

```
begin critical section;
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
end critical section;
```

Modifier

```
begin critical section;
if(head.value == "A"){
  head.value = "Z";
}
end critical section;
```



Reads		Writes
head	cur	cur
node1.next	count	count

Reads	Writes
head	node1.value
node1.value	

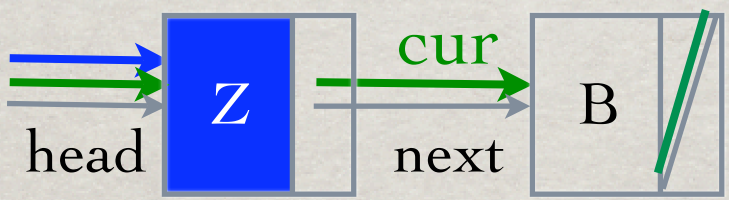
Optimistic Linked List

Counter

```
begin critical section;
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
end critical section;
```

Modifier

```
begin critical section;
if(head.value == "A"){
  head.value = "Z";
}
end critical section;
```



Reads		Writes
head	cur	cur
node1.next	count	count
node2.next		

Reads	Writes
head	node1.value
node1.value	

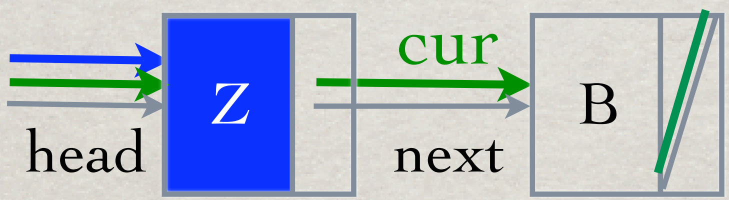
Optimistic Linked List

Counter

```
begin critical section;
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
end critical section;
```

Modifier

```
begin critical section;
if(head.value == “A”){
  head.value = “Z”;
}
end critical section;
```



Reads		Writes
head	cur	cur
node1.next	count	count
node2.next		

Reads	Writes
head	node1.value
node1.value	

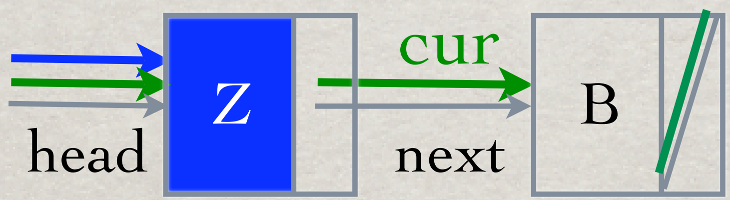
Optimistic Linked List

Counter

```
begin critical section;
cur = head;
while(cur.next != NULL){
  count++;
  cur = cur.next;
}
end critical section;
```

Modifier

```
begin critical section;
if(head.value == “A”){
  head.value = “Z”;
}
end critical section;
```



Reads		Writes
head	cur	cur
node1.next	count	count
node2.next		

Reads	Writes
head	node1.value
node1.value	

Optimistic Concurrency

- Transactional Memory
  - Modern Proposals: LogTM, TCC, VTM
- Lock-free data structures
  - Obstruction-free data structures

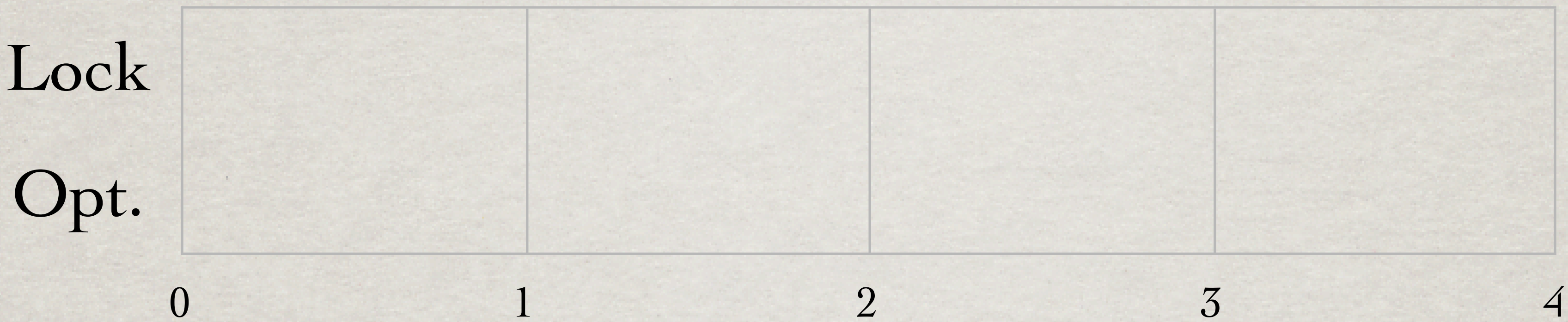
Key Questions

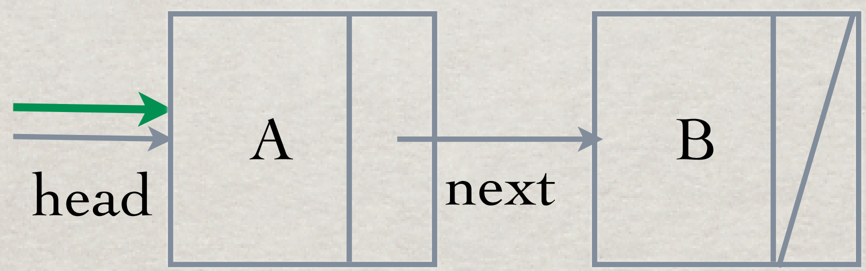
- How can optimistic concurrency help performance?
  - Eliminates unnecessary serialization
- How much does it help in practice?
- Will it help my existing lock-based system?
  - Methodology
  - Case Study

Performance Comparison

- Time lost to synchronization
  - Time spent acquiring locks
  - Time lost to restarted optimistic critical sections

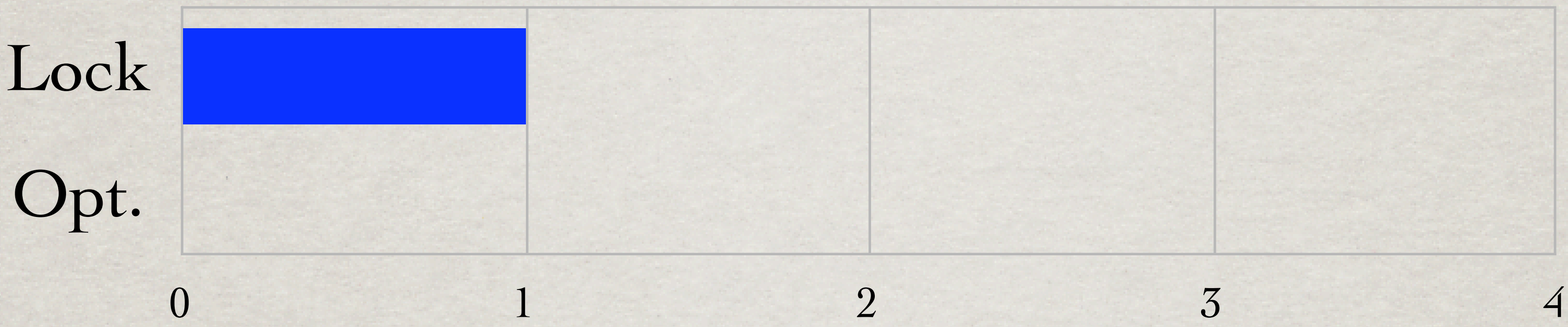
Locking Time

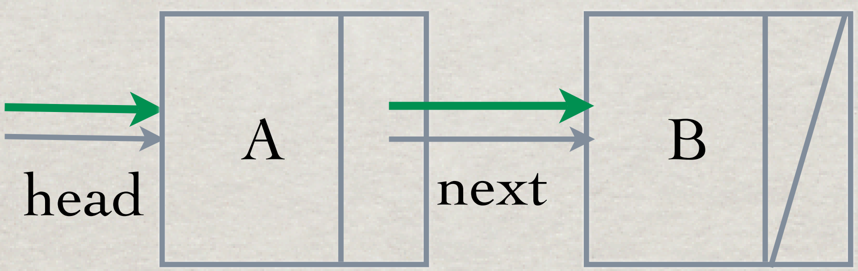




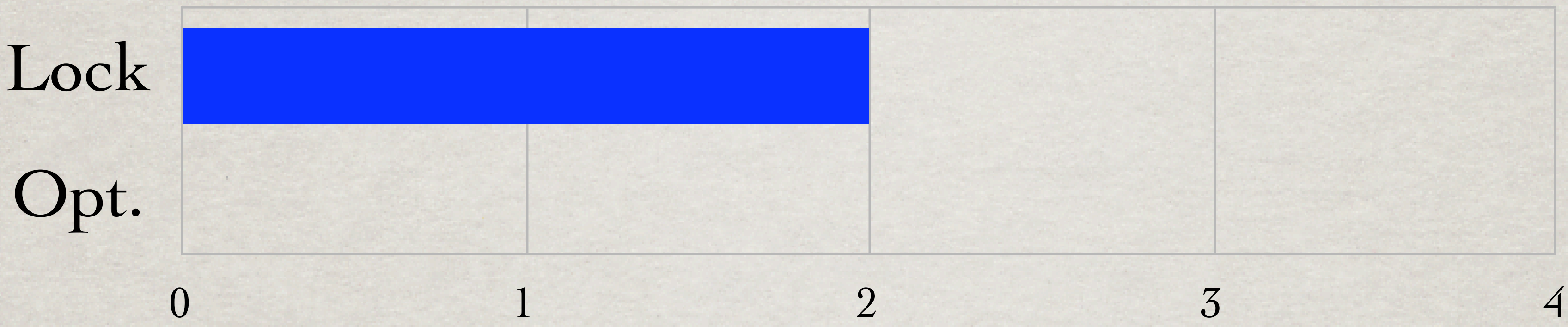
Suppose Insertion 1 acquires lock
Insertion 2 waits

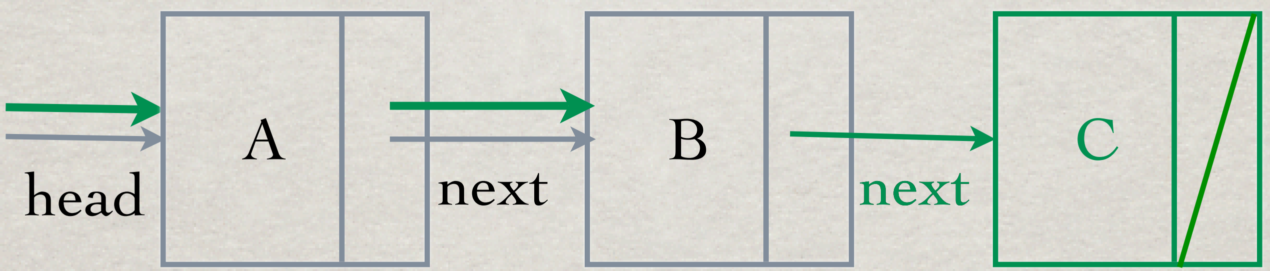
Locking Time





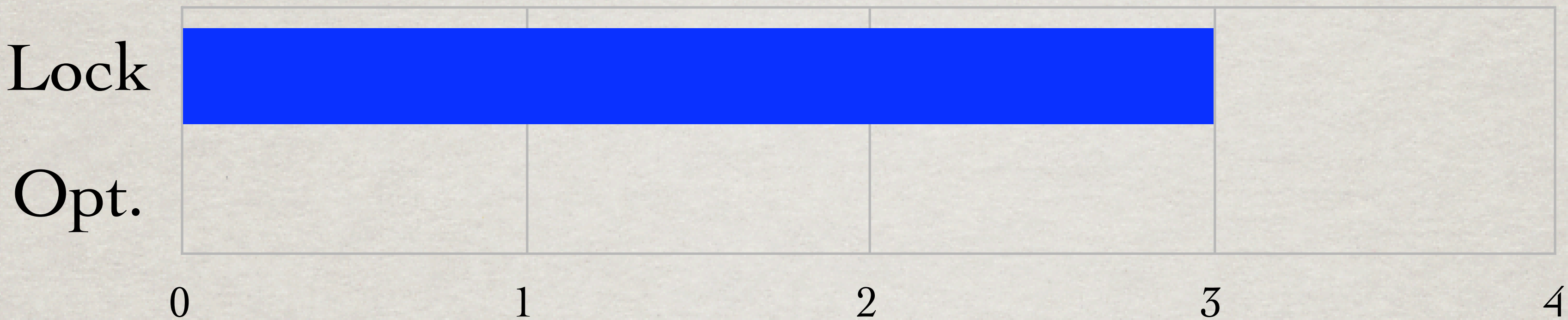
Locking Time

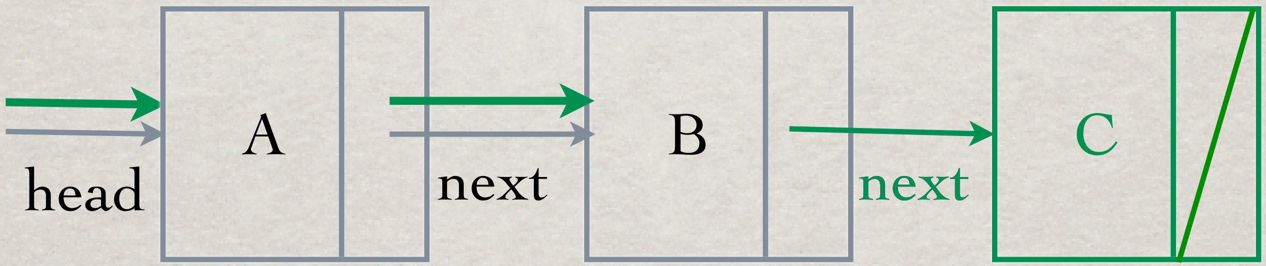




Suppose Insertion 1 acquires lock
Locking version of Insertion 2 waits

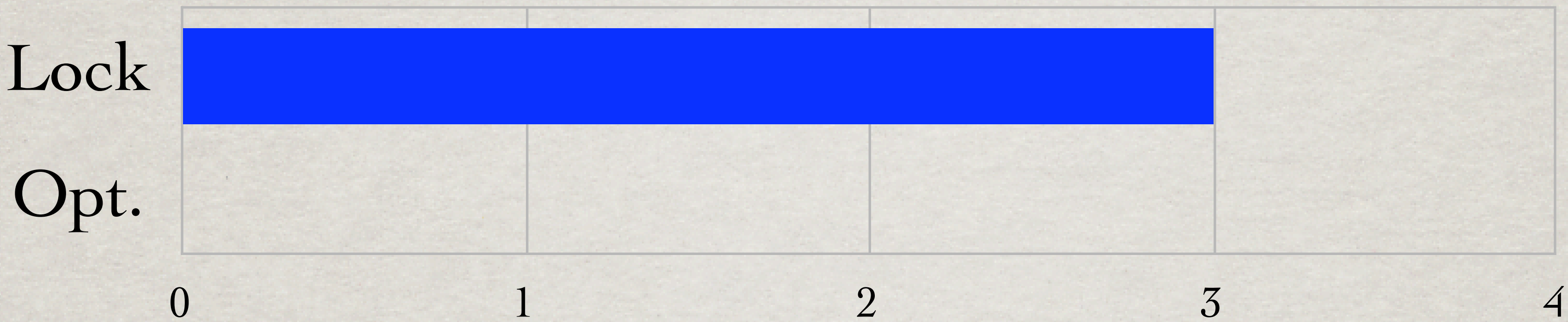
Locking Time

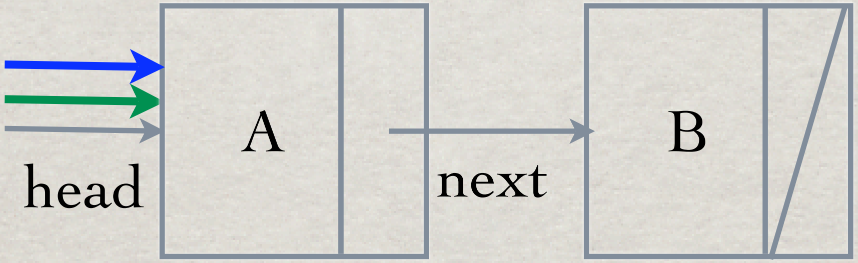




Insertion 1 releases lock
Insertion 2 acquires lock and completes

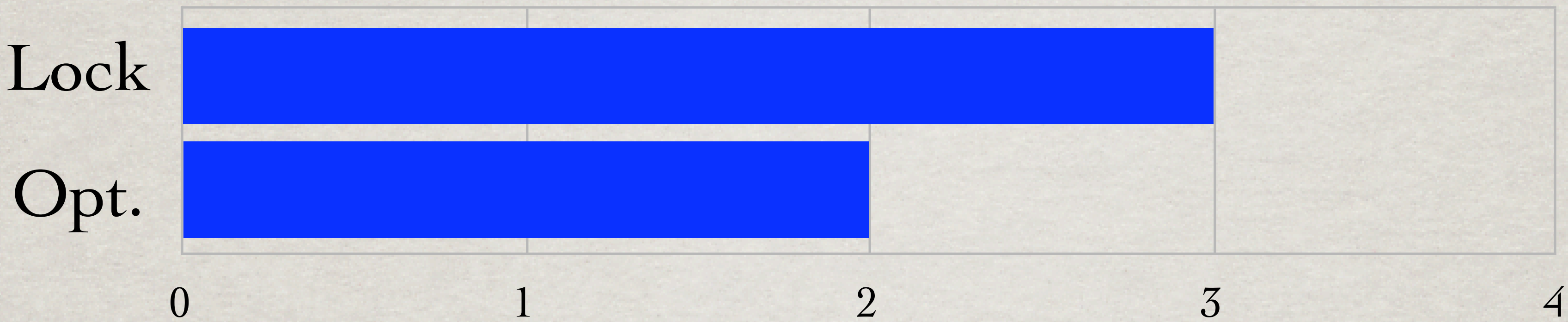
Optimistic Retry Time

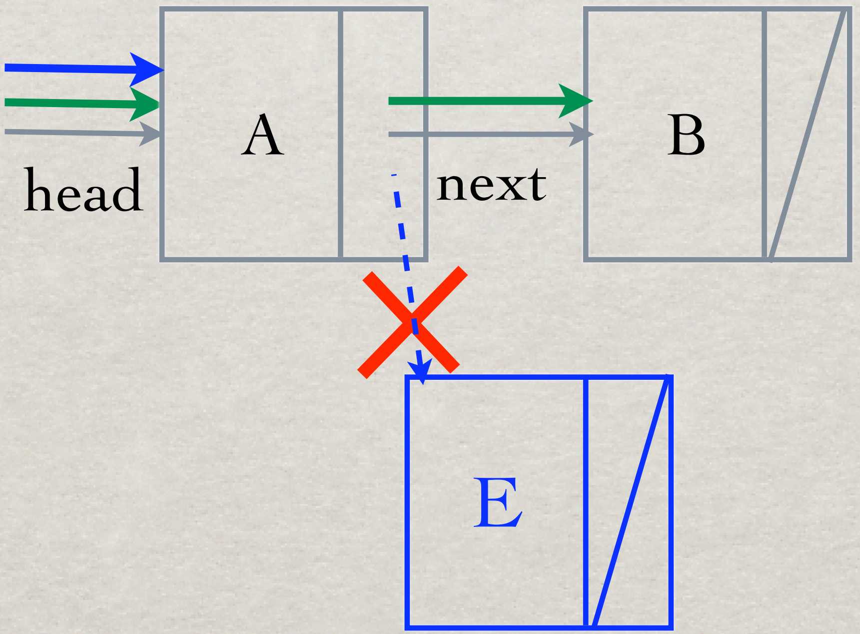




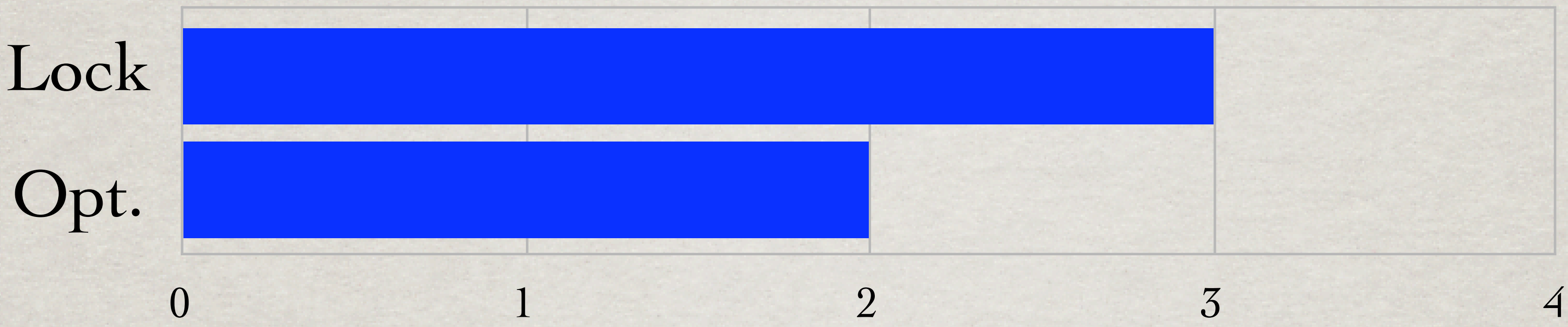
Suppose Insertion 1 always wins in a conflict
Insertion 2 speculatively executes

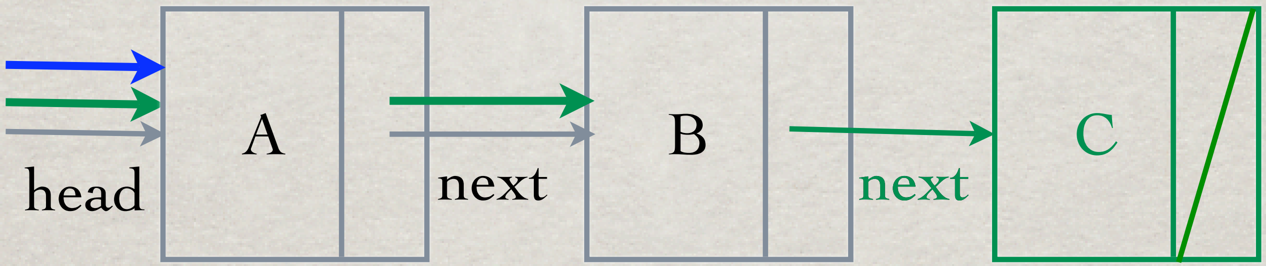
Optimistic Retry Time





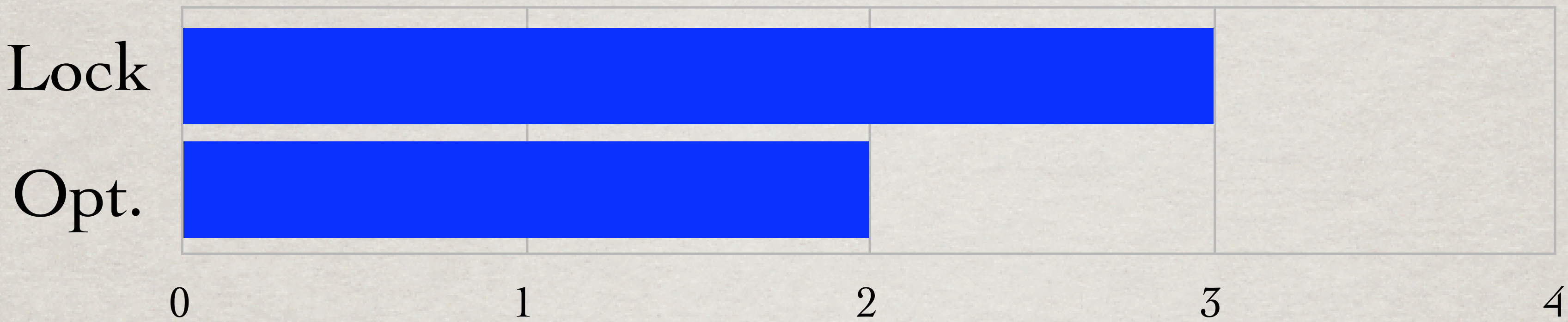
Optimistic Retry Time

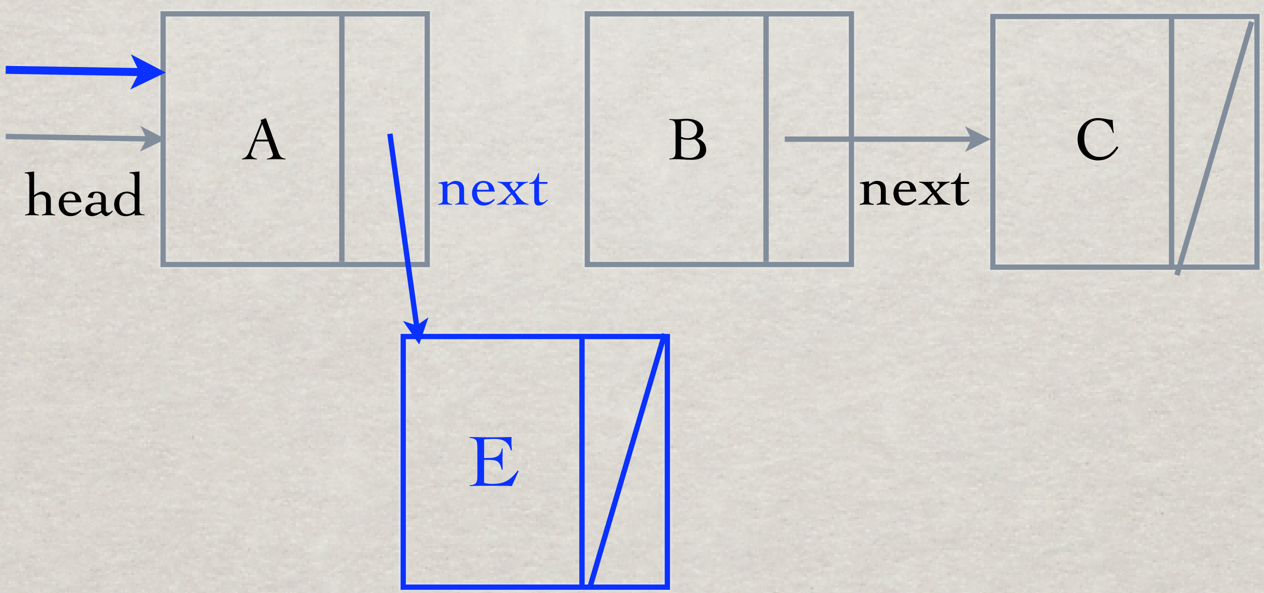




Insertion 2 rolls back and retries

Optimistic Retry Time





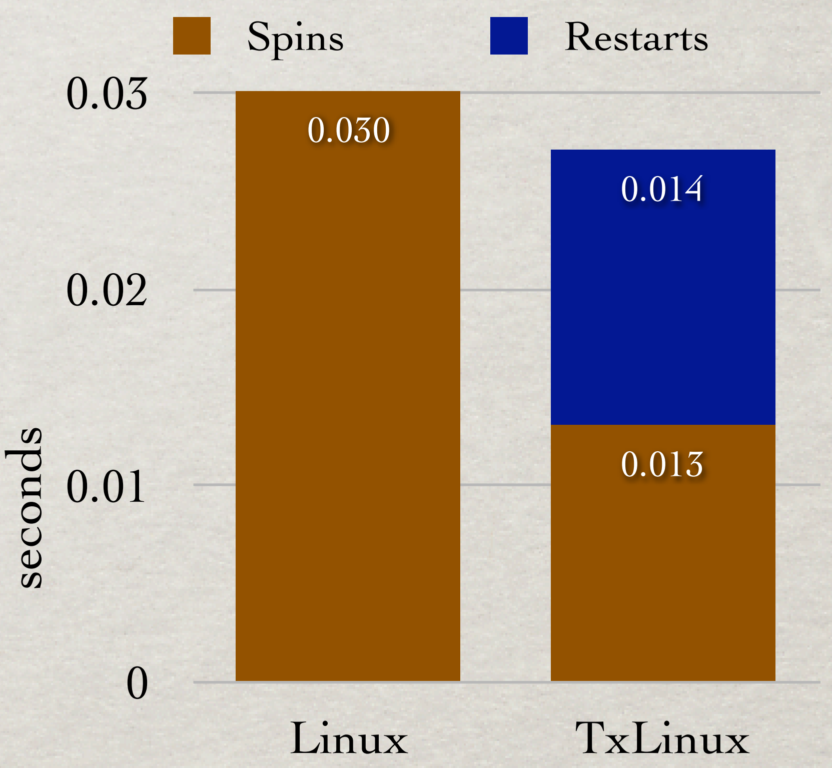
Insertion 1 has committed

Spinlocks Vs. Transactional Memory

- Compare Linux to TxLinux (ISCA 2007)
  - TxLinux converts some critical sections protected by spinlocks to hardware transactions
  - Leaves other spinlocks undisturbed
- Exercised by parallel make benchmark
  - Compile 27 source files from libFLAC 1.1.2
- Simulated 15 CPU machine

Spinlocks Vs. Transactional Memory

- 8% reduction in time wasted synchronizing
- 32% reduction in lock acquires
- Opens up new tuning opportunities



Time Wasted in Synchronization for Pmake Workload

Key Questions

- How can optimistic concurrency help performance?
  - Eliminates unnecessary serialization
- How much does it help in practice?
  - Marginal improvement for Linux running pmake
- → Will it help my existing lock-based system?
  - Methodology
  - Case Study

Address Sets and Conflicts

- **Address Set** of critical section A: the memory addresses read (R_A) and written (W_A) during A's execution

$$R_A \cup W_A$$

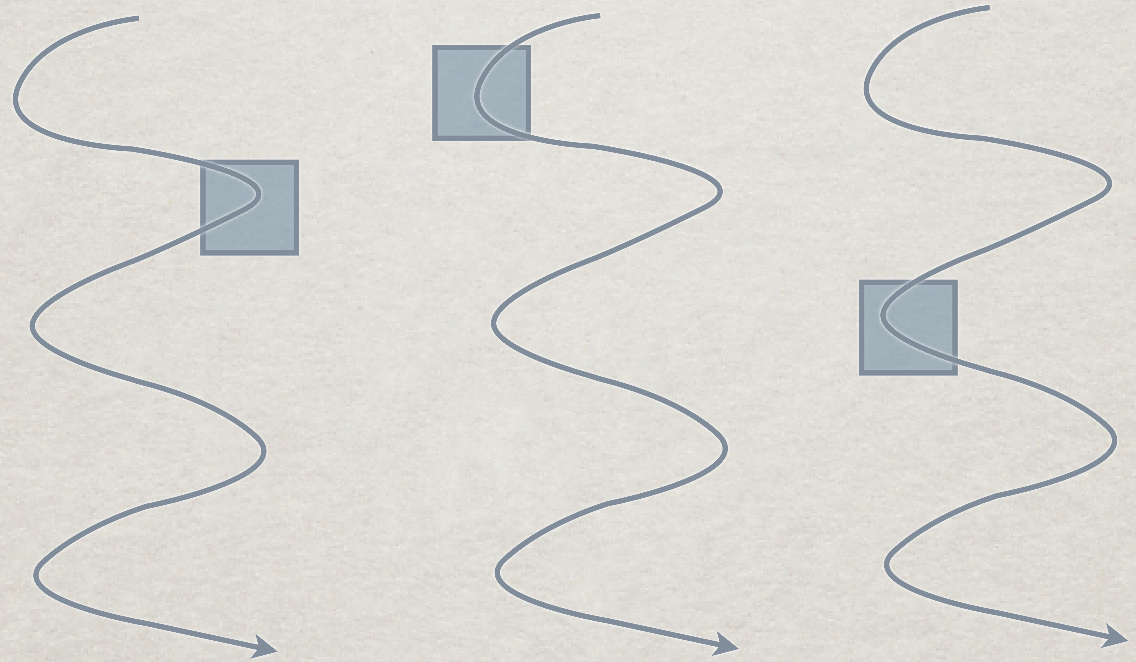
- Critical section A **conflicts** with B if:

$$W_A \cap (R_B \cup W_B) \neq \emptyset$$

Data Independence

- **Data independent** critical sections can't conflict
  - Conservative: ignores "lucky" schedules
  - Essential to optimistic performance

Measuring Data Independence

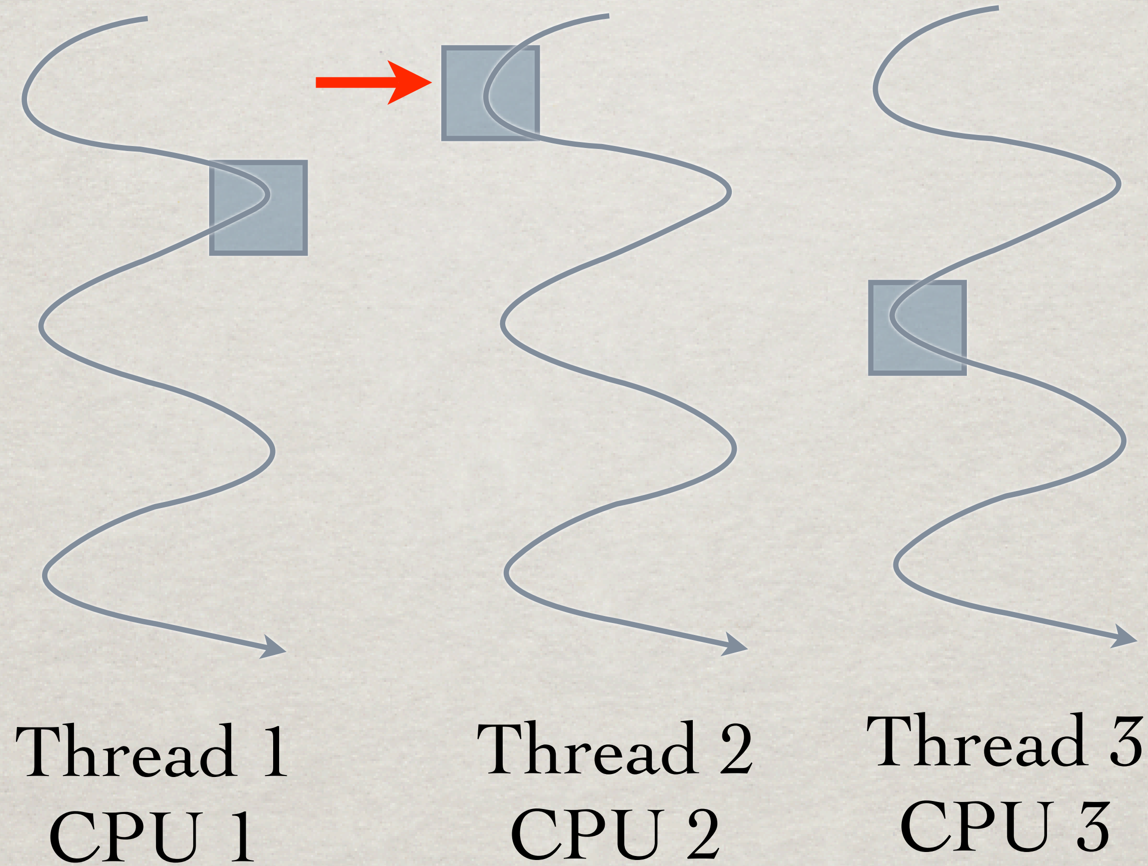


Thread 1
CPU 1

Thread 2
CPU 2

Thread 3
CPU 3

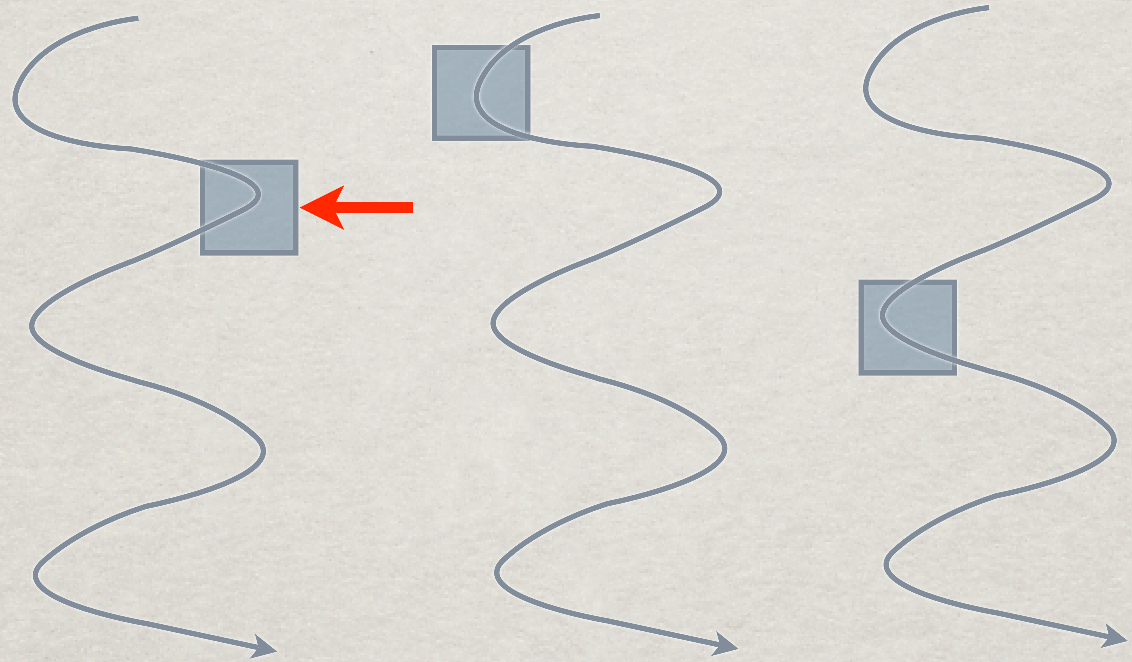
Measuring Data Independence



R	W
A	B

Measuring Data Independence

R	W
A	C
D	



Thread 1
CPU 1

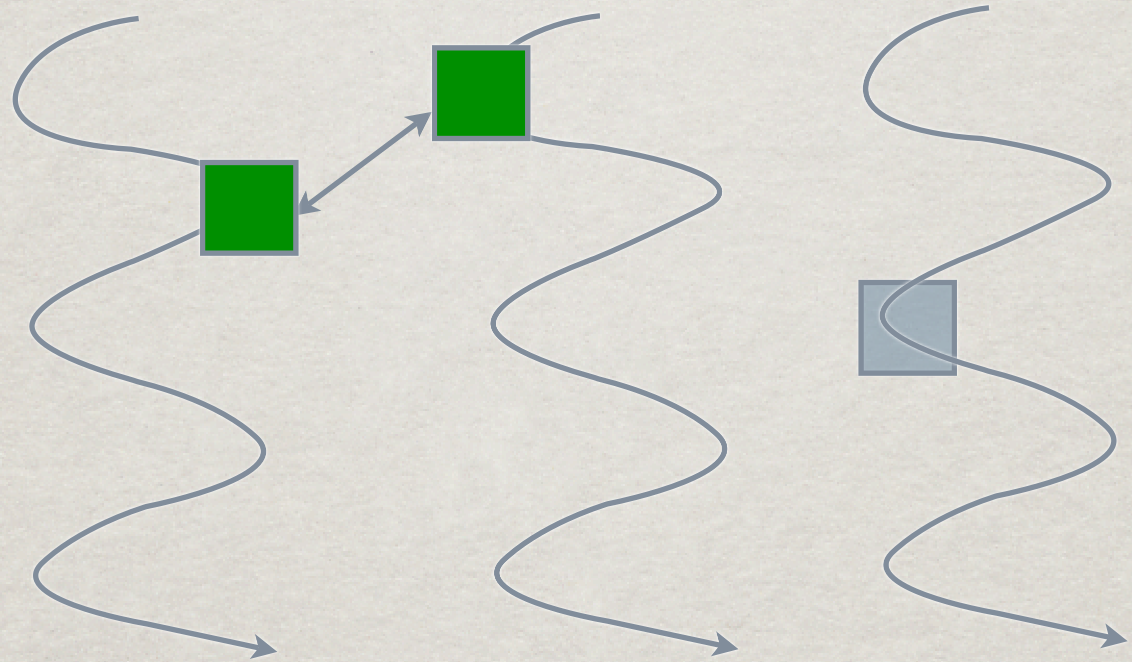
Thread 2
CPU 2

Thread 3
CPU 3

R	W
A	B

Measuring Data Independence

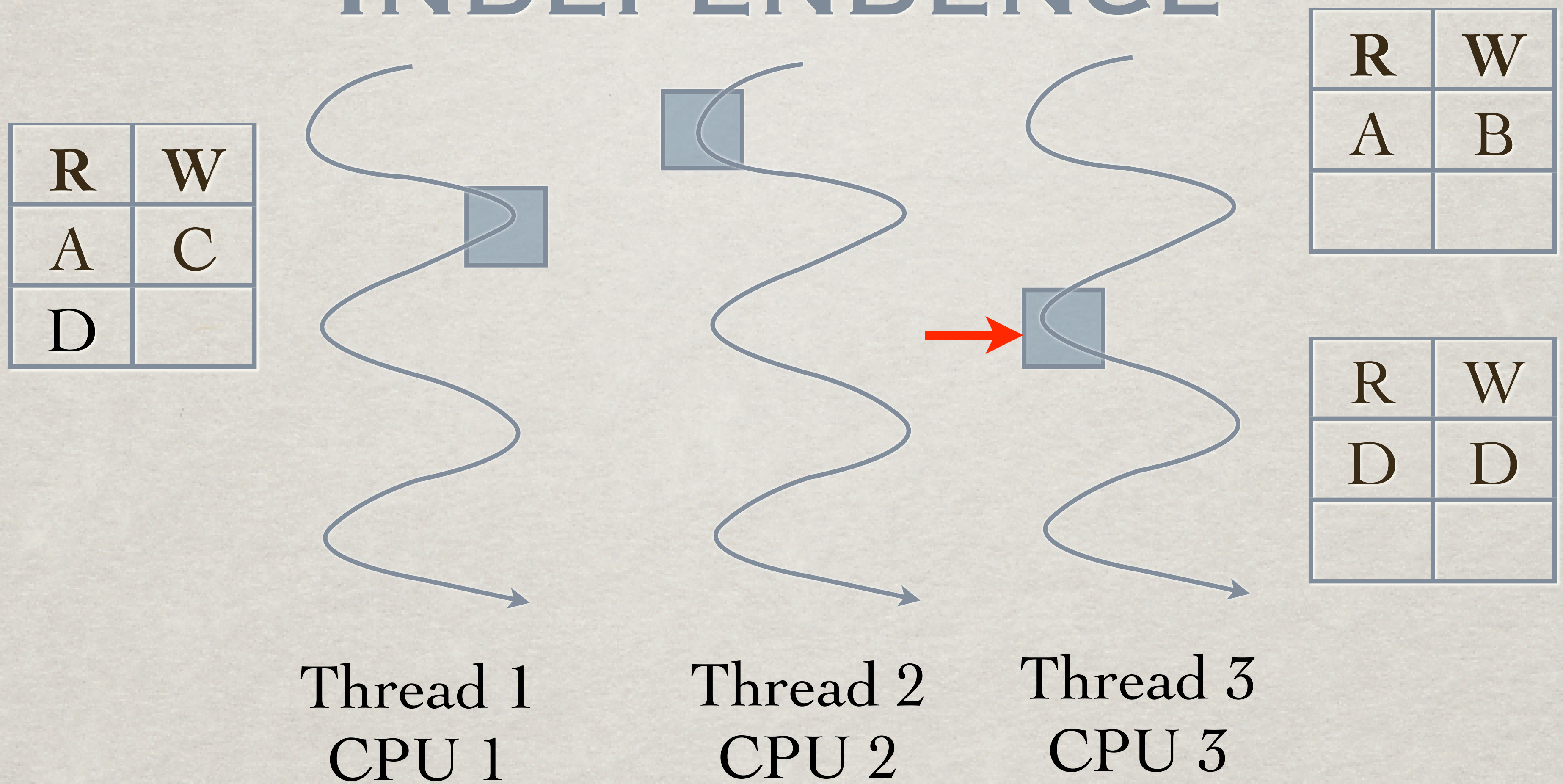
R	W
A	C
D	



R	W
A	B

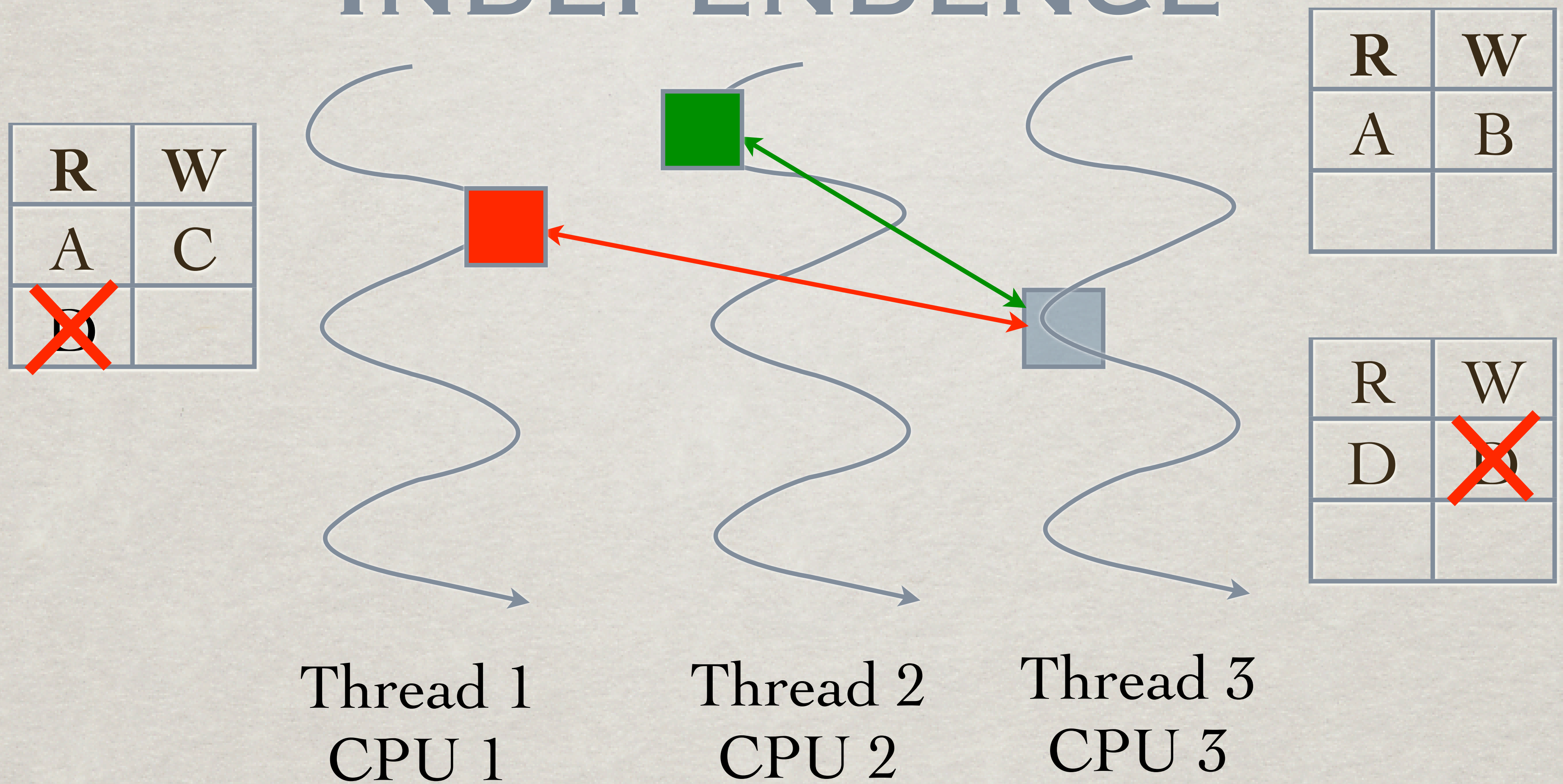
Data independence: 100%

Measuring Data Independence



Data independence: 100%

Measuring Data Independence



Data independence: 66%

Measuring Data Independence

- For each execution of a critical section:
  - Track loads and stores
  - Compare to prior address sets for same lock
  - Keep a running percentage of conflicts

Key Questions

- How can optimistic concurrency help performance?
  - Eliminates unnecessary serialization
- How much does it help in practice?
  - Marginal improvement for Linux running pmake
- Will it help my existing lock-based system?
  - Methodology: Measure data independence
  - Case Study

Case Study: The Linux Kernel

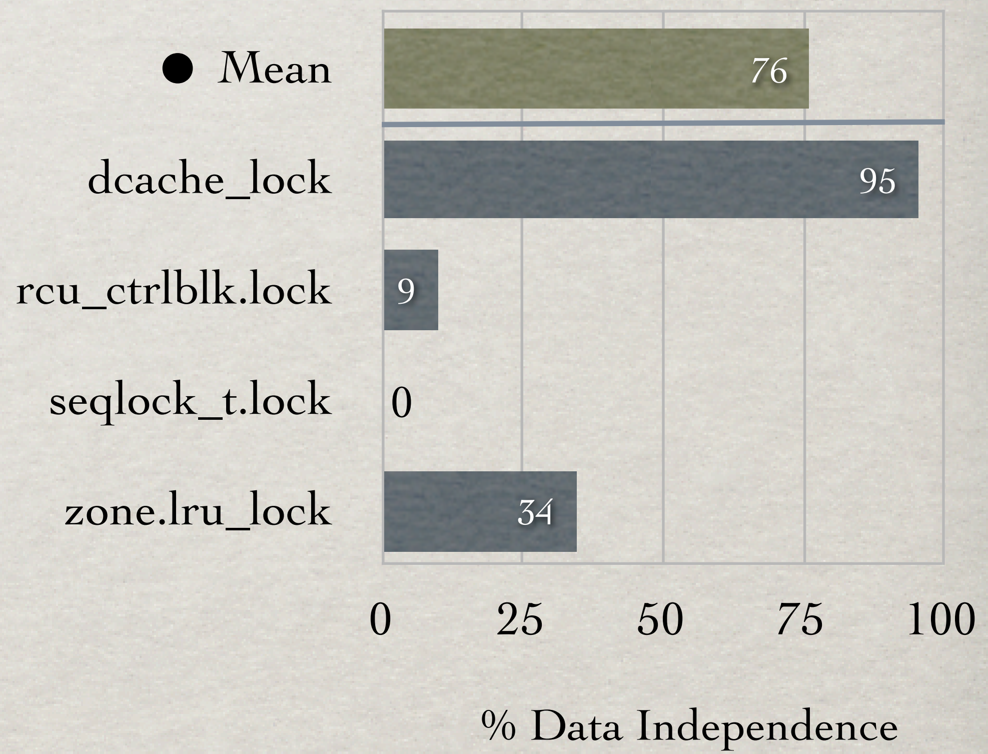
- Workload: Linux 2.6.16.1
  - Exercised by parallel make benchmark
- Simics 3.0.17
  - Full-system, execution-driven simulator
  - 15 CPU machine

Synchronization Characterization (Syncchar)

- Tracks kernel synchronization inside simulator
    - Lock acquires and releases
    - Loads and stores performed while a lock is held
    - Time lock is held
    - Time waiting for a lock
- Negligible impact on simulated system

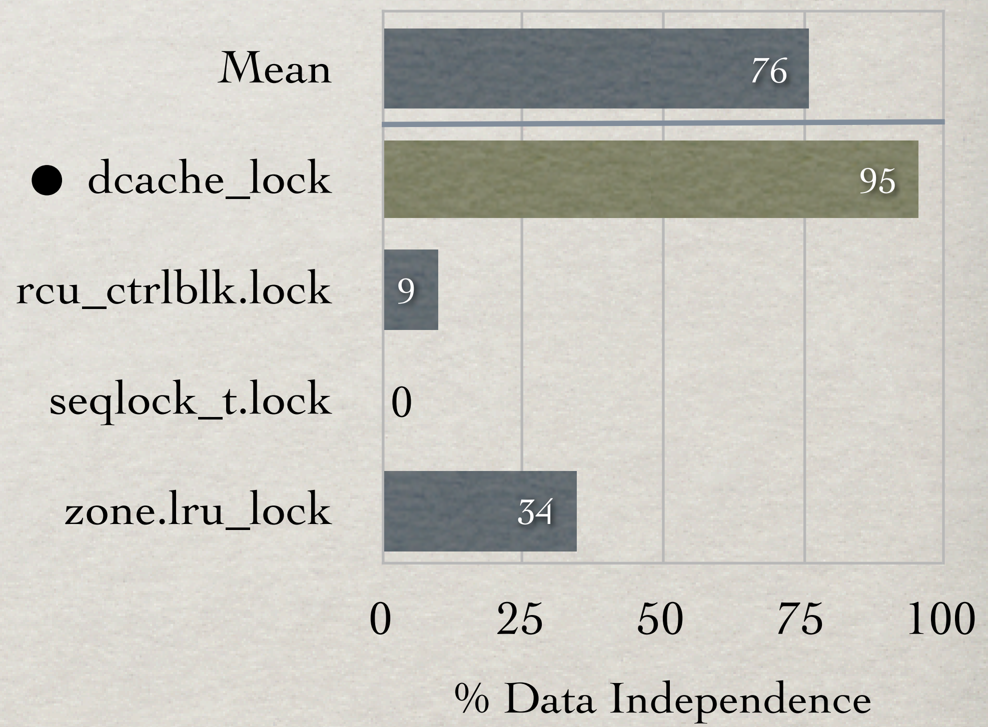
Kernel Spinlock Average

- Mean of all kernel spinlocks
  - Weighted by time lock held
- Small scalability



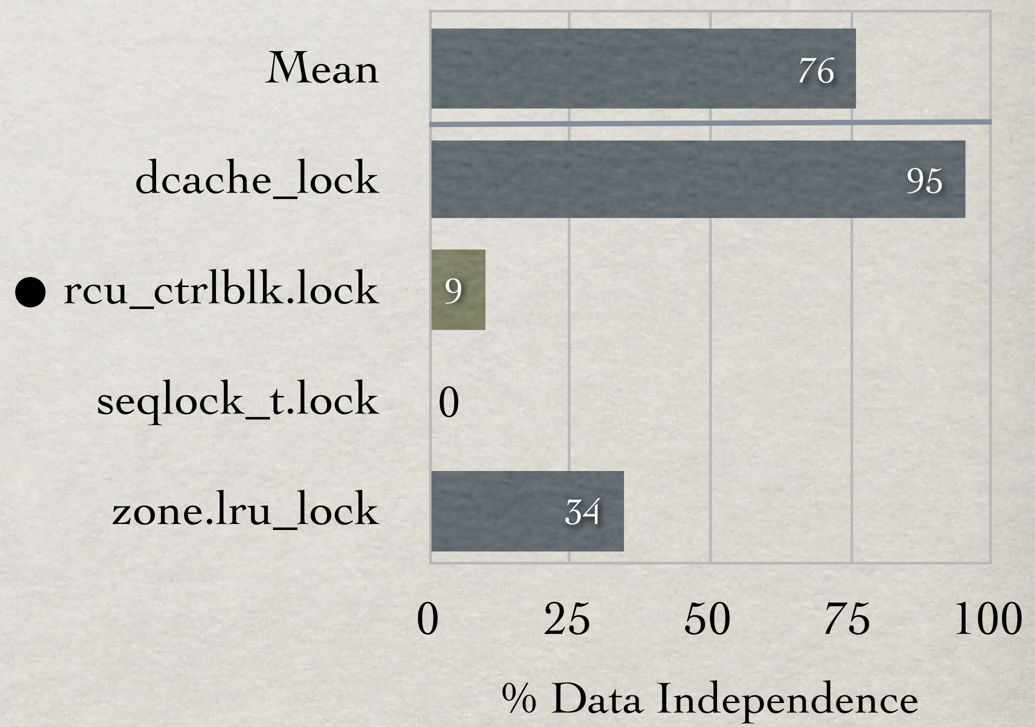
Dcache Lock

- Coarse-grained lock
- Protects cache of file names
- Large scalability



RCU Control Block Lock

- Fine-grained lock
- Protects a small, global control structure
- Short, simple critical sections
- Negligible Scalability
- Little room for optimistic improvement

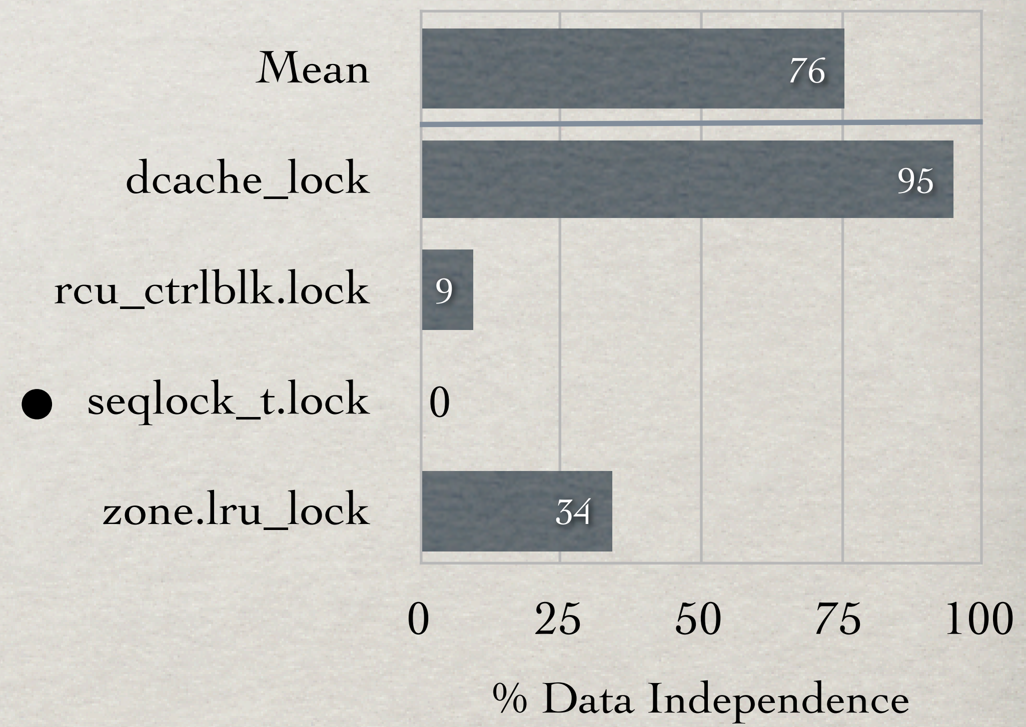


Sequence Locks

- Linux kernel synchronization primitive
- Optimistic readers
  - Read sequence number before and after reads
- Sequential writers
  - Write seq. number before and after writes
- Sequence number protected by a spinlock

Sequence Lock Internal Spinlock

- 0% Data independence for internal lock that serializes writers
- Doesn't account for optimistic readers

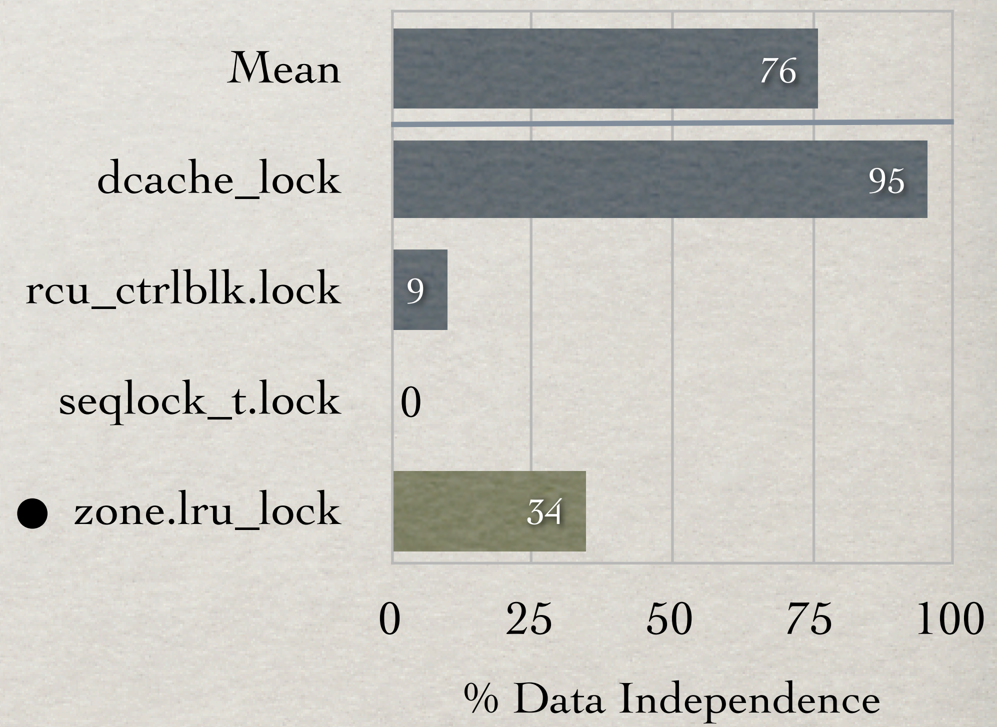


Levels of Abstraction

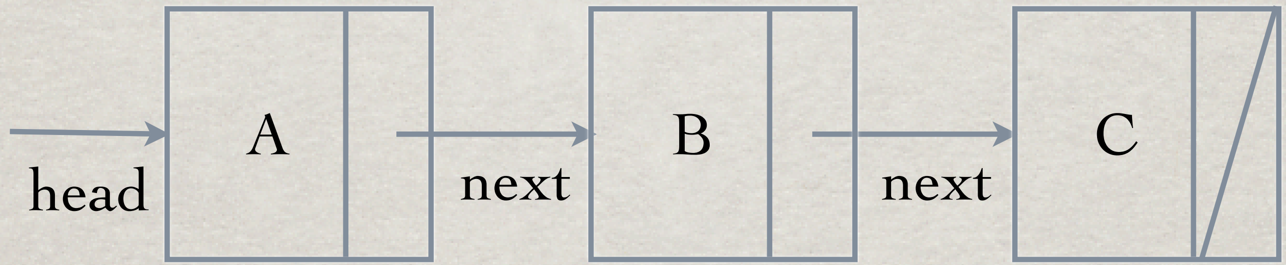
- Current work only looks at spinlocks
- Spinlocks used in some higher-level primitives
- Extend model in future work

Zone LRU Lock

- Protects two linked lists
  - Common kernel data structure
- Negligible Scalability

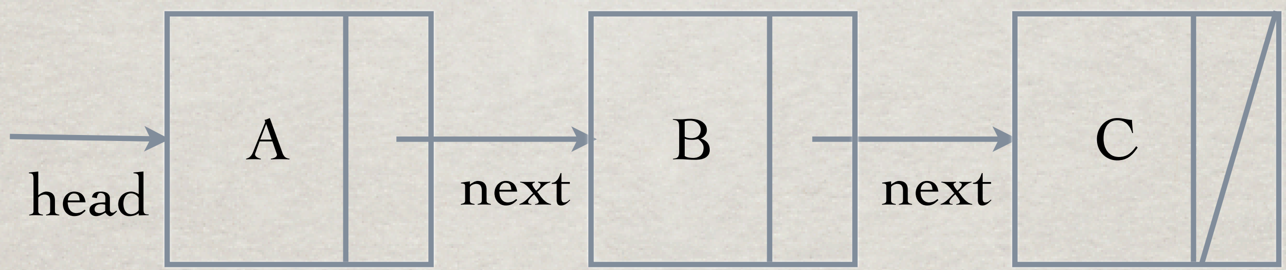


Linked List Pathology



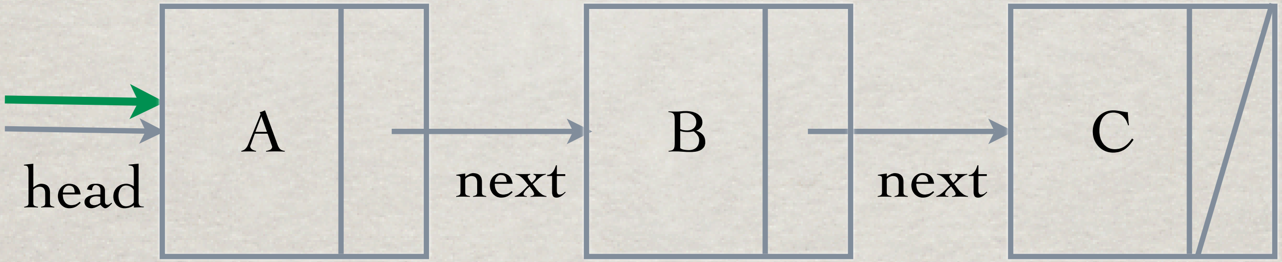
Linked List Pathology

Insertion 1



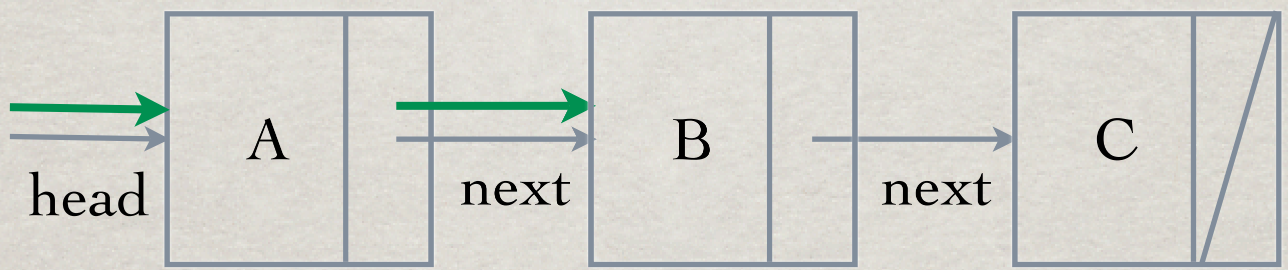
Linked List Pathology

Insertion 1



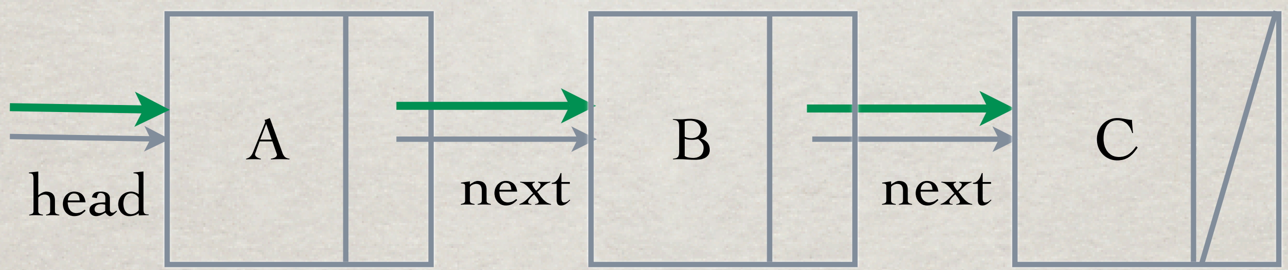
Linked List Pathology

Insertion 1



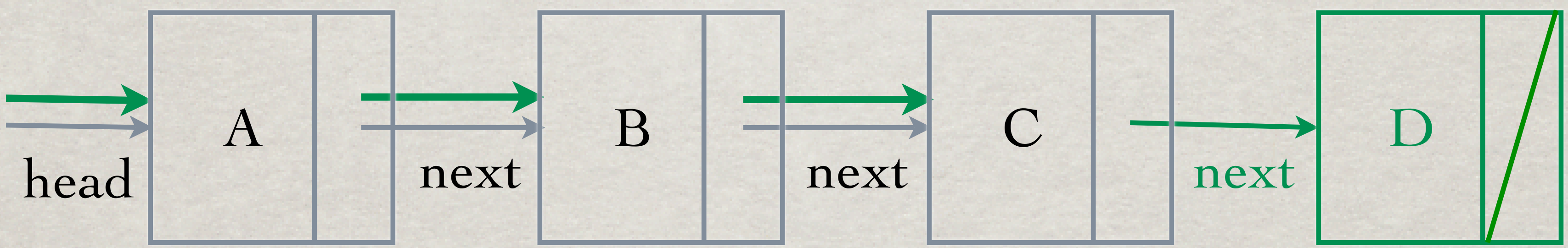
Linked List Pathology

Insertion 1



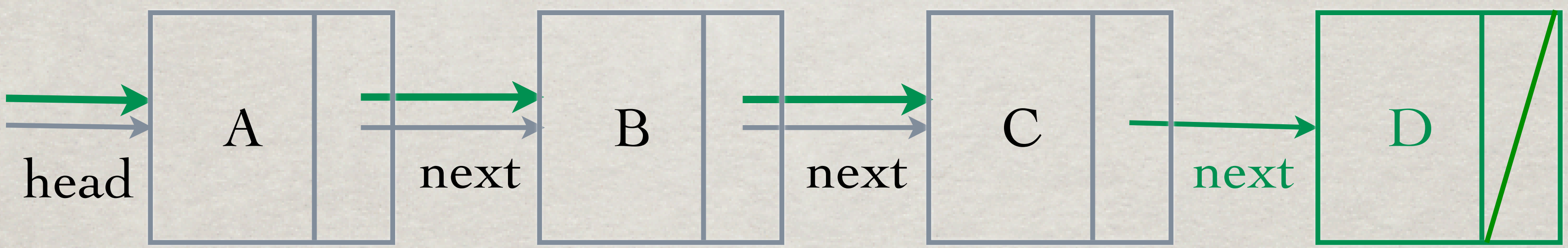
Linked List Pathology

Insertion 1



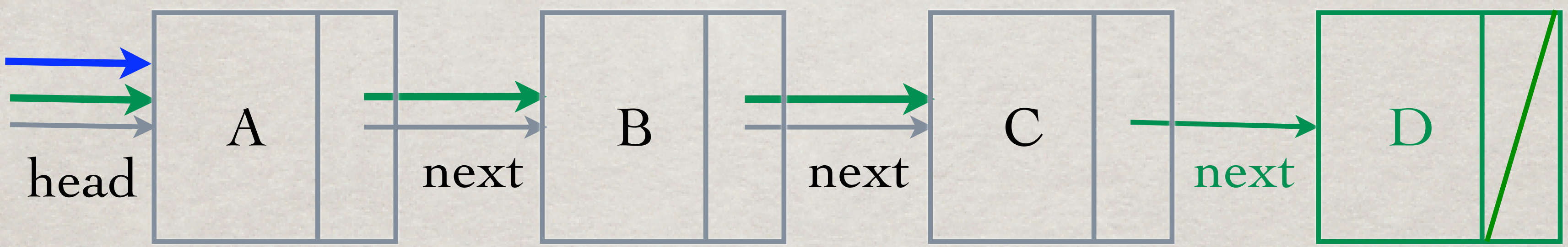
Linked List Pathology

Insertion 2



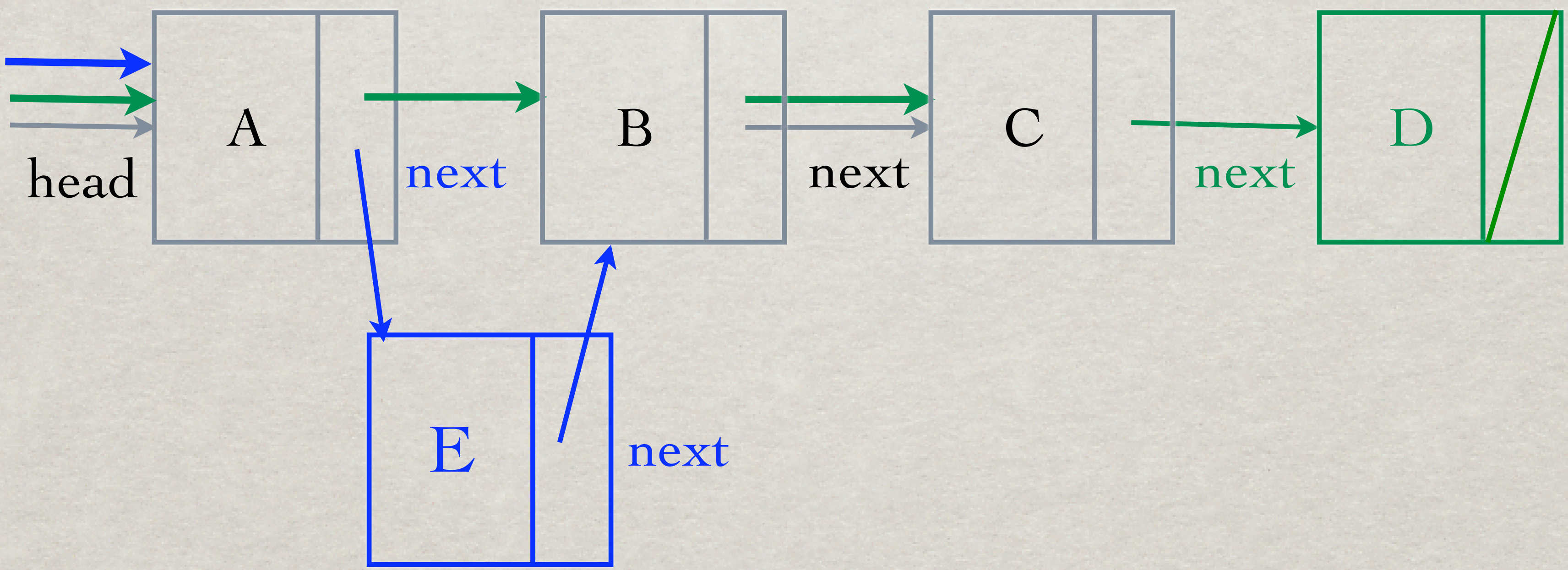
Linked List Pathology

Insertion 2



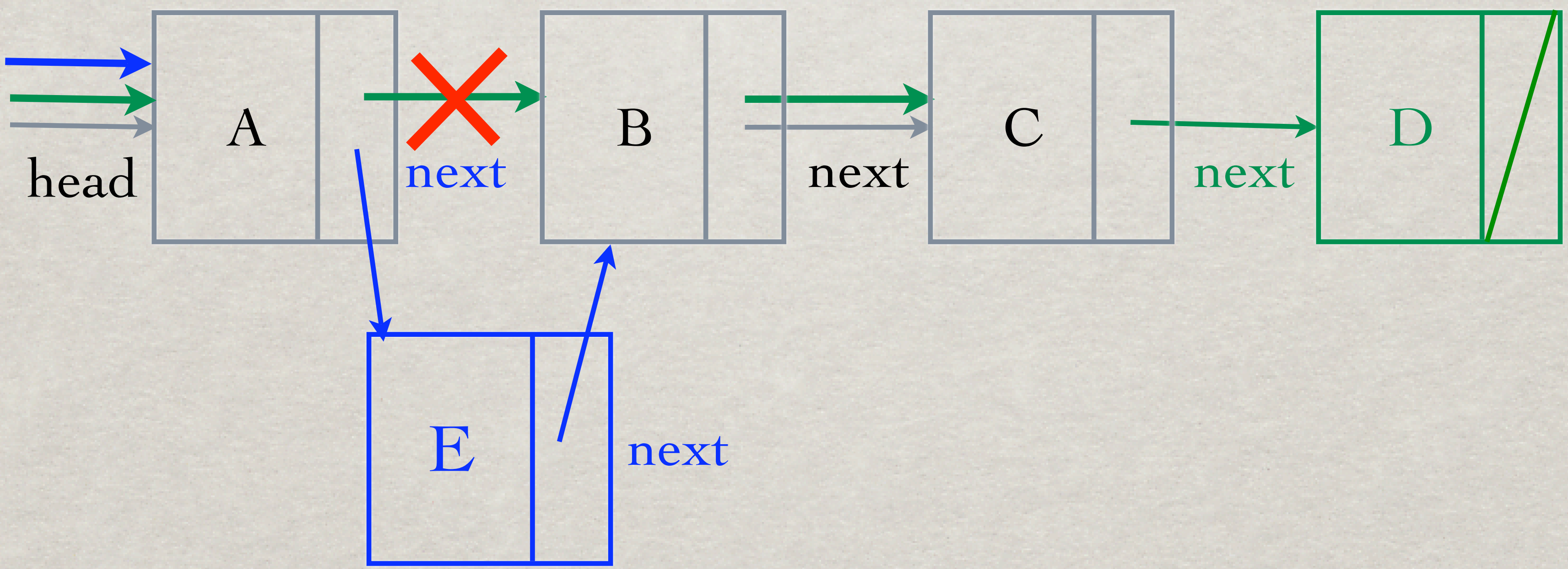
Linked List Pathology

Insertion 2



Linked List Pathology

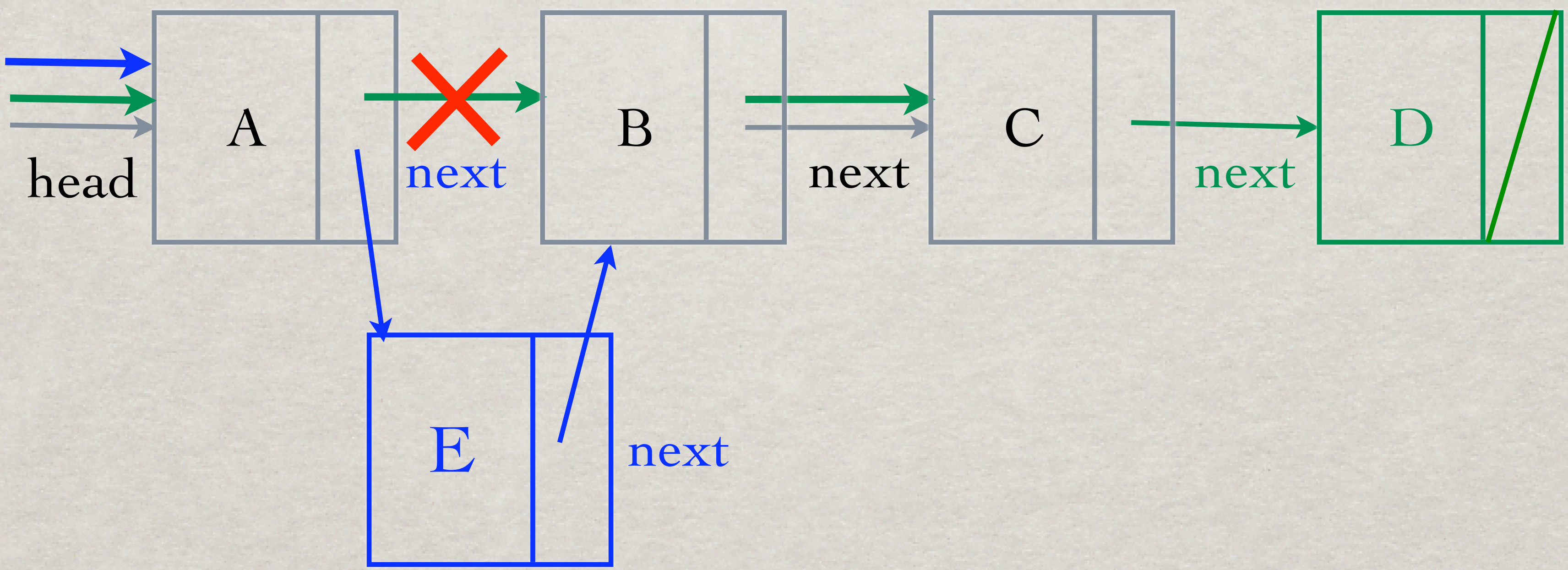
Conflict!



Linked List Pathology

Conflict!

No two insertions or deletions are data independent



Linked List Pathology

- Some common data structures are ill-suited to optimistic concurrency
- Conflict avoidance becomes first order concern
- Reorganization necessary for more concurrency

Key Questions

- How can optimistic concurrency help performance?
  - Eliminates unnecessary serialization
- How much does it help in practice?
  - Marginal improvement for Linux running pmake
- Will it help my existing lock-based system?
  - If it has high data independence

Is The Optimism Warranted?

- It depends...
- Syncchar can answer this for your system!
- For the Linux kernel running pmake:
  - 76% data independence
  - Data structure reorganization can uncover more parallelism

Questions?