

# Lock Reservation

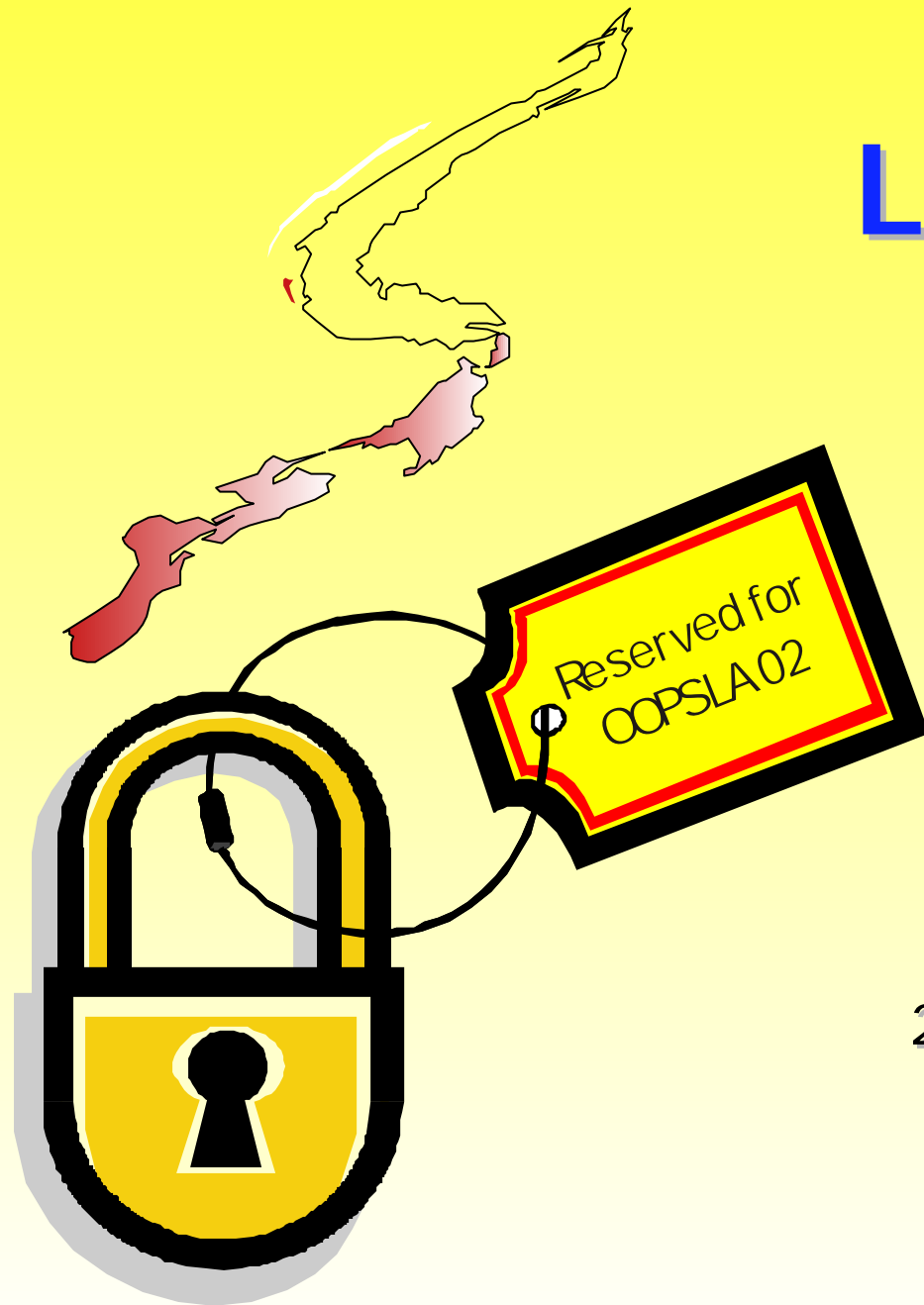
Java Locks Can Mostly Do  
Without Atomic Operations

Kikyokuni Kawachiya

Akira Koseki

Tamiya Onodera

IBM Tokyo Research Laboratory  
2002/11/07 OOPSLA 2002 Seattle

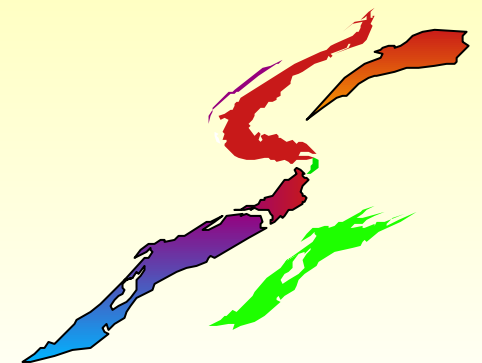


# Outline

---

Presents a new runtime technique to accelerate Java locks

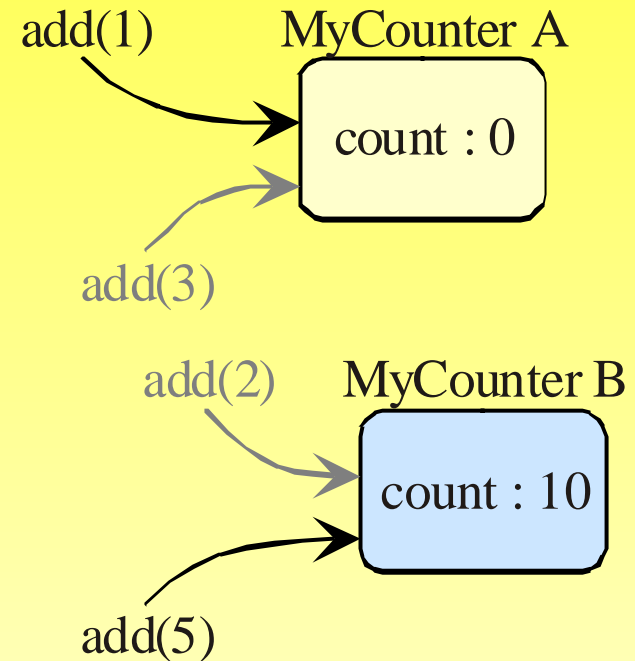
- Background: Java Locks Review
- Thread Locality of Java Locks
  - Definition and Examination
- A New Technique: Lock Reservation
  - Data Structure and Algorithm
- Performance Measurements
  - Micro- and Macro-Benchmarks
- Conclusion and Future Work



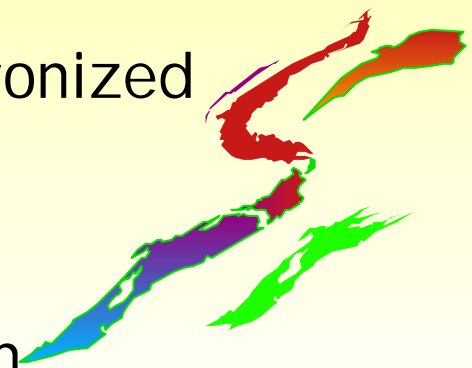
# Java Locks Review

- Specified by a *synchronized method*

```
class MyCounter {  
    int count = 0;  
  
    synchronized int add(int i) {  
        count = count + i;  
        return count;  
    }  
}
```



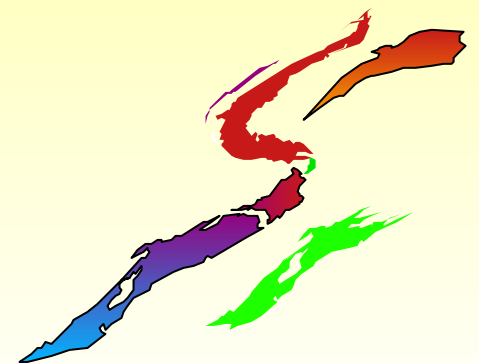
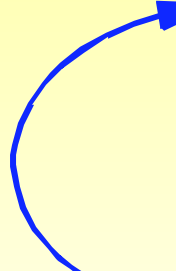
- Java adopts *monitor* semantics
  - At most one thread can execute the synchronized method
  - The object's **lock** is *acquired* before the execution and *released* after the execution



# Accelerating Java Locks

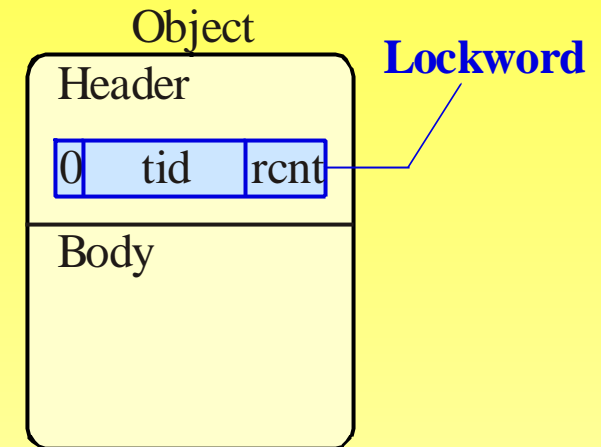
---

- Lock ops. tend to be performed frequently, because:
  - Libraries must be designed as thread-safe
  - Programmers can easily specify locks by "synchronized"
- Two approaches to reduce the lock overhead
  1. Compiler techniques to eliminate lock operations
    - By escape analysis, nested-lock analysis, ...
  2. Runtime techniques to make lock operations cheaper
    - Thin lock [Bacon98], tasuki lock [Onodera99], meta lock [Agesen99], ...
- Focus of this talk



# Runtime Techniques of Java Locks

- Characteristics of Java locks
  - Most locks are not contended
  - Most contentions are temporary
- ➔ Runtime techniques optimize the *uncontended cases*



- Atomic operation(s) such as `compare_and_swap` are still necessary
  - They are expensive in modern architectures
  - Last-remaining overhead in the uncontended cases
- ➔ Any way to remove the atomic operations?



# Any Way to Remove the Atomic Ops.?

- An assumption:

An object is synchronized only by a specific thread

→ Atomic operations are unnecessary for the object

- Not true even in single-threaded applications

- Housekeeping threads are running in Java system

- Threads may be dynamically created in future

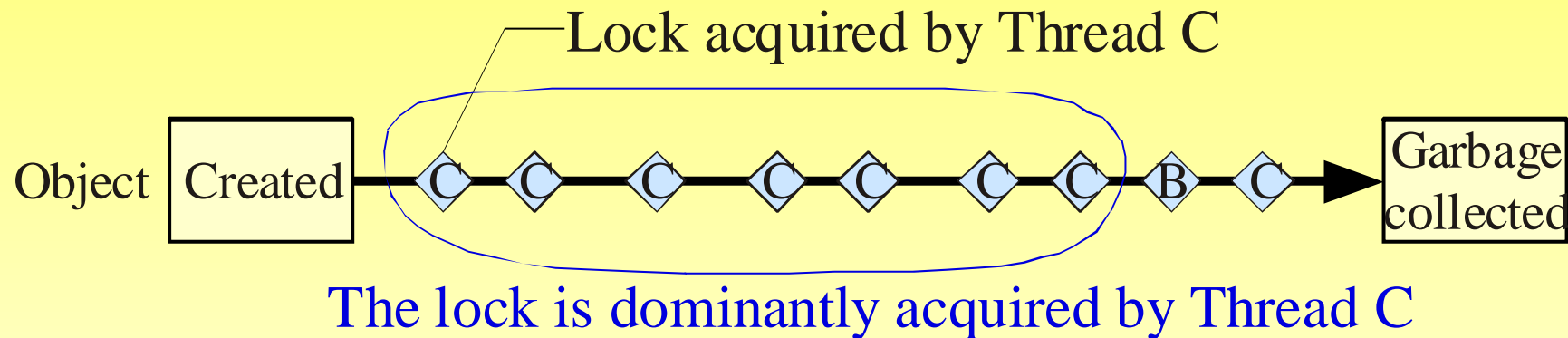
- Modify the assumption to more usable form



# Thread Locality of Java Locks

- A modified assumption:

An object is synchronized *dominantly* by a specific thread  
--- *Thread locality* of Java locks



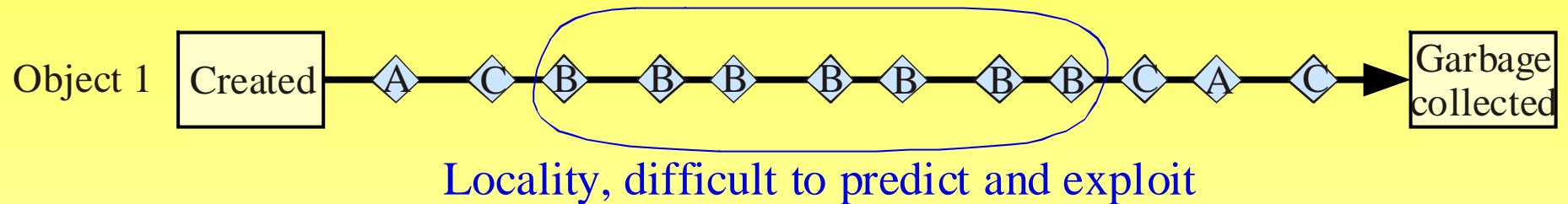
- ➔ Introduce an asymmetry to Java locks
  - Reduce the cost of lock by giving a preference to the thread



# Exploitable Thread Locality (Figure 1)

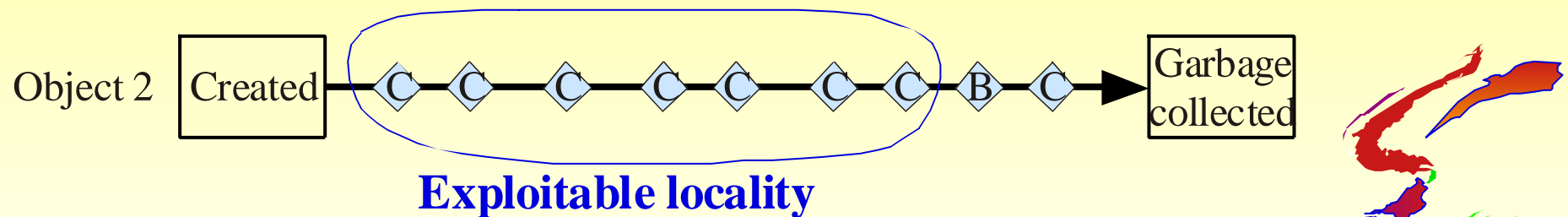
## ■ General thread locality

An object's lock is repeatedly acquired by the same thread

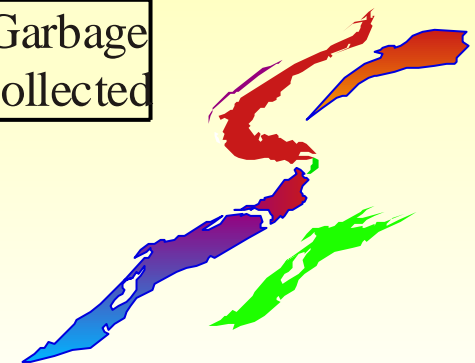


## ■ Exploitable thread locality

An object's lock continues to be acquired by the *initial locker*

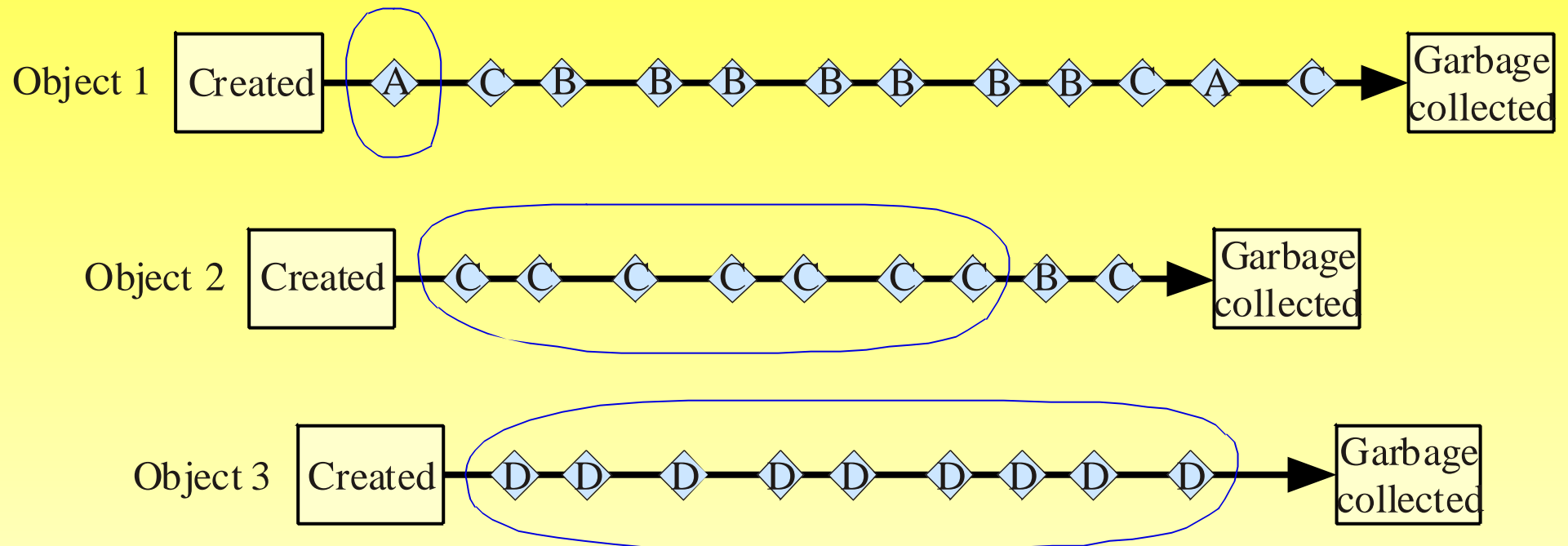


→ Give a preference to the initial locker



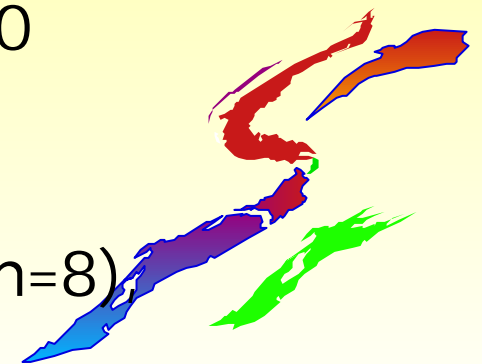
# Examine the Exploitable Thread Locality

- Count the *first repetition* in each object's lock sequence



→ Ratio of lock ops. in 1st. repetitions = 17/30

- I investigated 10 Java programs (Table 1)
  - 7 SPECjvm98 programs, SPECjbb2000 (#wh=8)  
Volano Server and Client



# Result of the Examination (Table 2)

Program name	Number of sync'd objs.	Number of lock ops.	Ratios of lock ops. in 1st. repetitions	Objects sync'd by multiple threads
SPECjvm98				
_202_jess	21,278	14,646,978	99.993%	187 (0.878%)
_201_compress	2,135	28,895	97.211%	127 (5.948%)
_209_db	66,592	162,117,521	99.9998%	52 (0.078%)
_222_mpegaudio	1,620	27,168	98.108%	91 (5.617%)
_228_jack	1,635,497	38,570,415	99.998%	144 (0.0088%)
_213_javac	1,192,734	47,062,772	99.974%	1,760 (0.148%)
_227_mtrt	3,020	3,522,926	99.557%	114 (3.775%)
SPECjbb2000	2,077,210	102,282,147	75.392%	176,318 (8.488%)
Volano Server	7,279	7,244,208	75.983%	1,888 (25.94%)
Volano Client	4,102	10,419,671	84.270%	1,640 (39.98%)

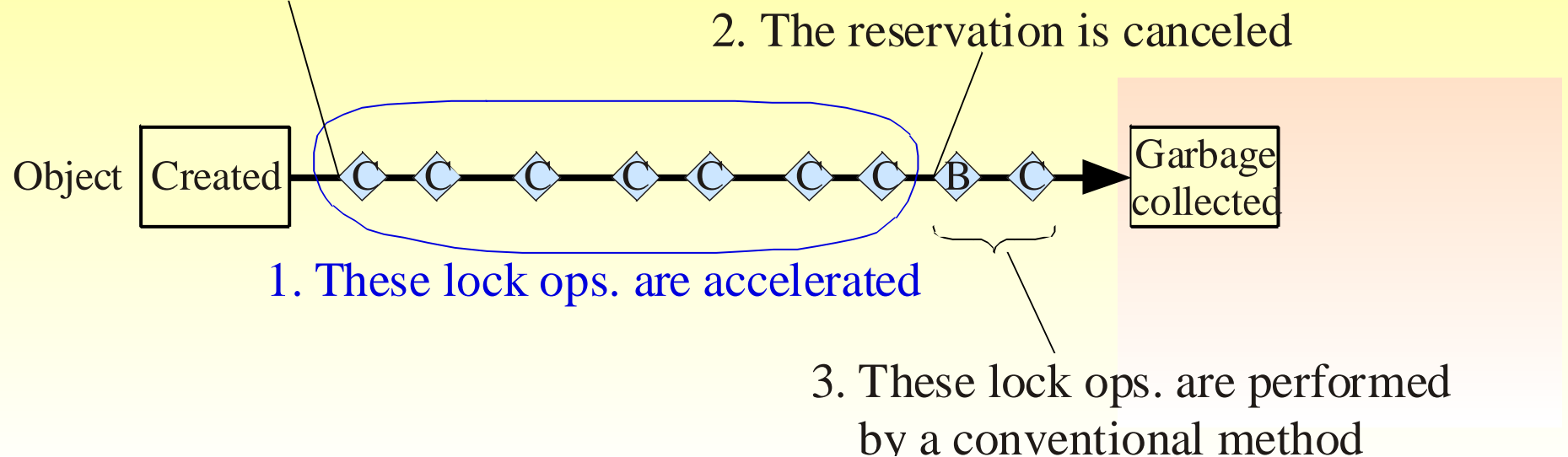
Multi-  
readed

- Most lock ops. are performed in the 1st. repetitions
  - More than 75% even for multi-threaded programs
  - ➔ Performance can be improved by accelerating this case

# Lock Reservation

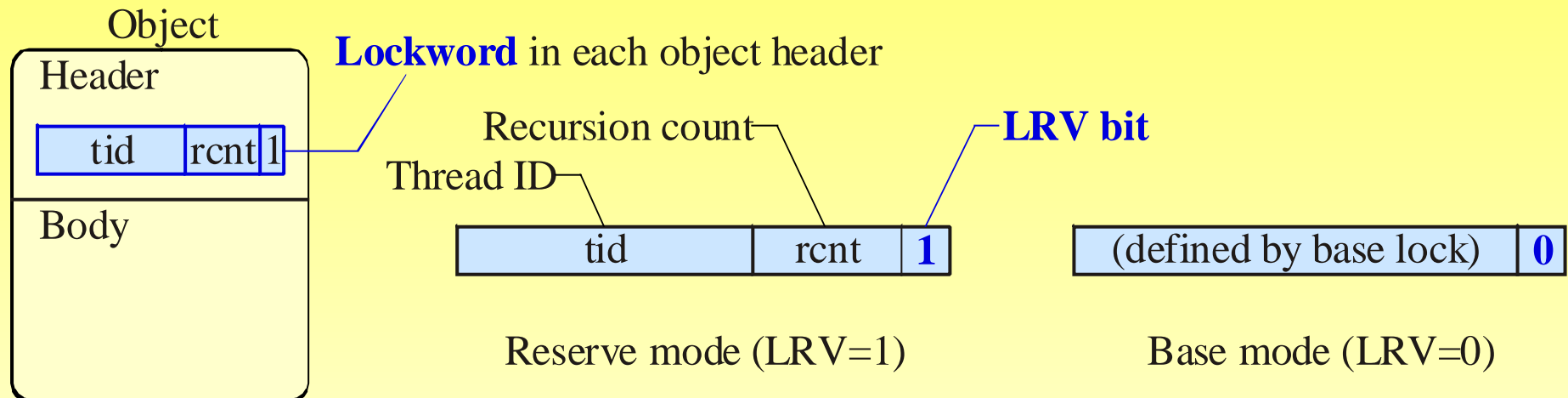
- Reserve the object's lock for the initial locker thread
  - Reduce the cost of subsequent lock ops. by the thread
- When a thread attempts to acquire an object's lock:
  1. If reserved for the thread, the lock can be *acquired very quickly*
  2. If reserved for another thread, the reservation is *anceled* first
  3. If not reserved, the lock is acquired by a conventional method

0. The lock is reserved for Thread C



# Data Structure (Figure 2)

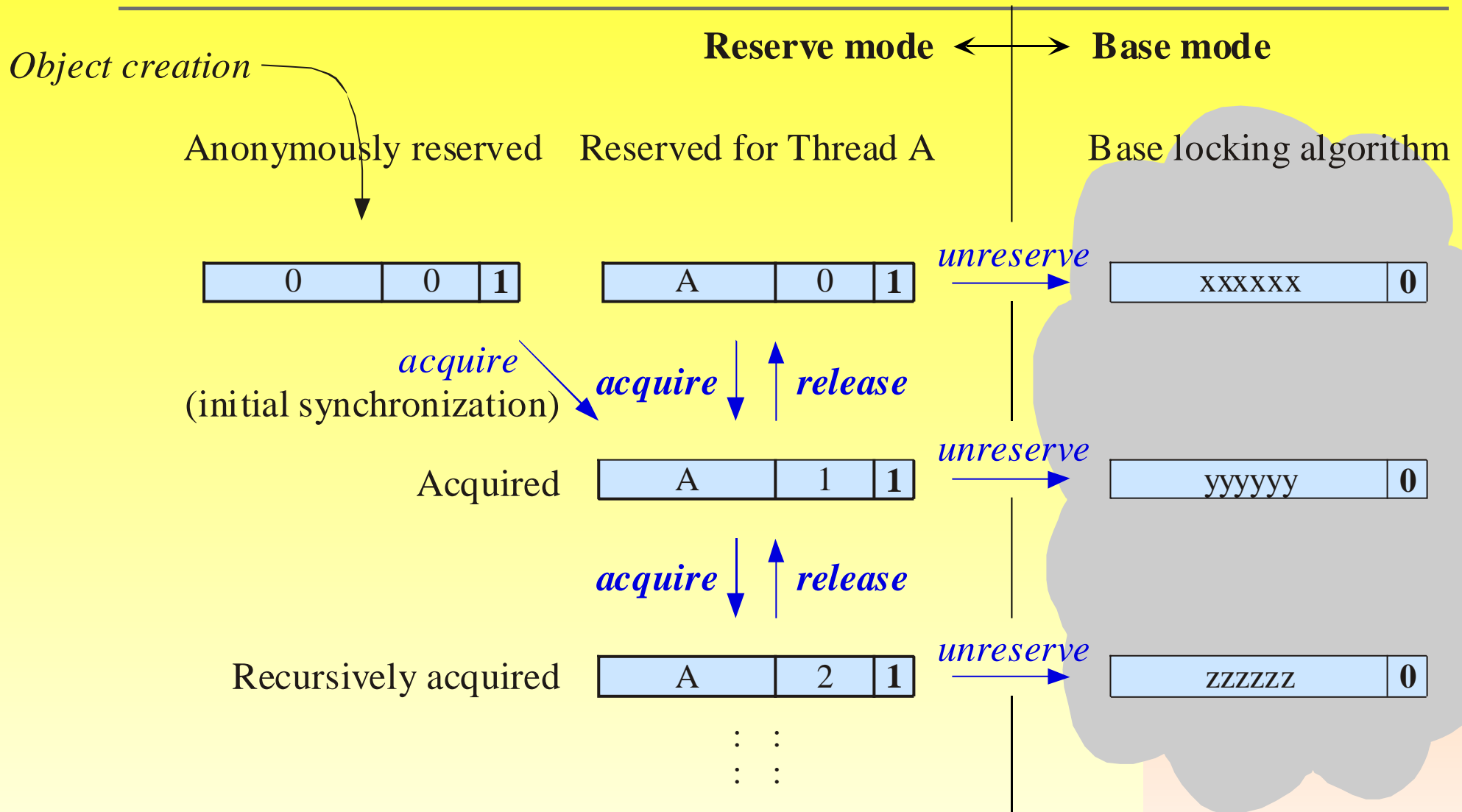
- Lock reservation can be built on any existing locks, as long as it uses a *lockword* in the object header
- LRV bit* represents whether the lock is reserved or not



## Lockword semantics in the reserve mode

- |   |    |   |   |
|---|----|---|---|
| A | 0  | 1 | (a) Reserved for Thread A, but not held |
| A | >0 | 1 | (b) Reserved for and held by Thread A   |
| 0 | 0  | 1 | (c) Reserved anonymously                |

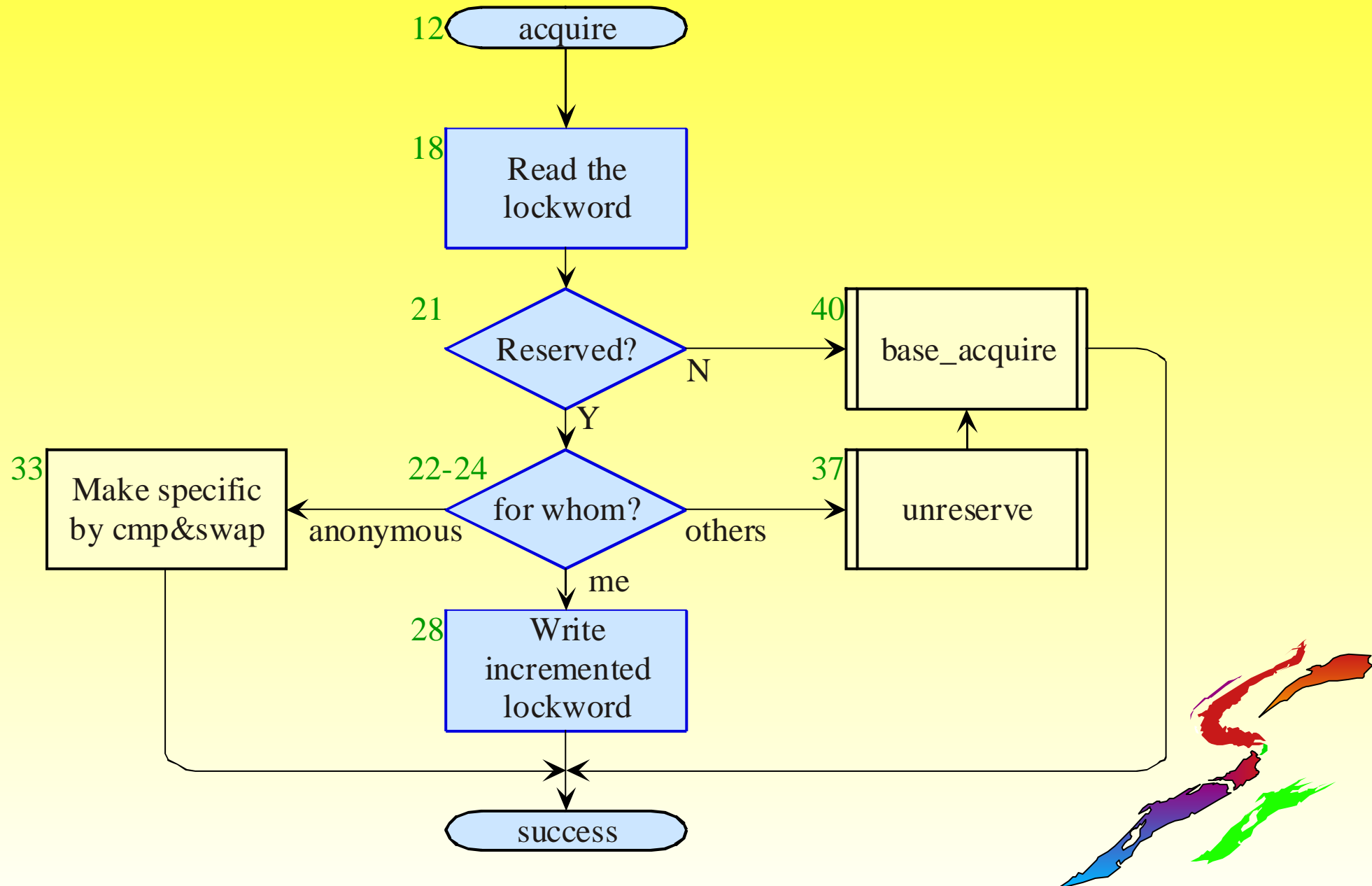
# Lock State Transitions (Figure 3)



- In reserve mode, lock can be acquired simply incrementing the rcnt field *without atomic operations*

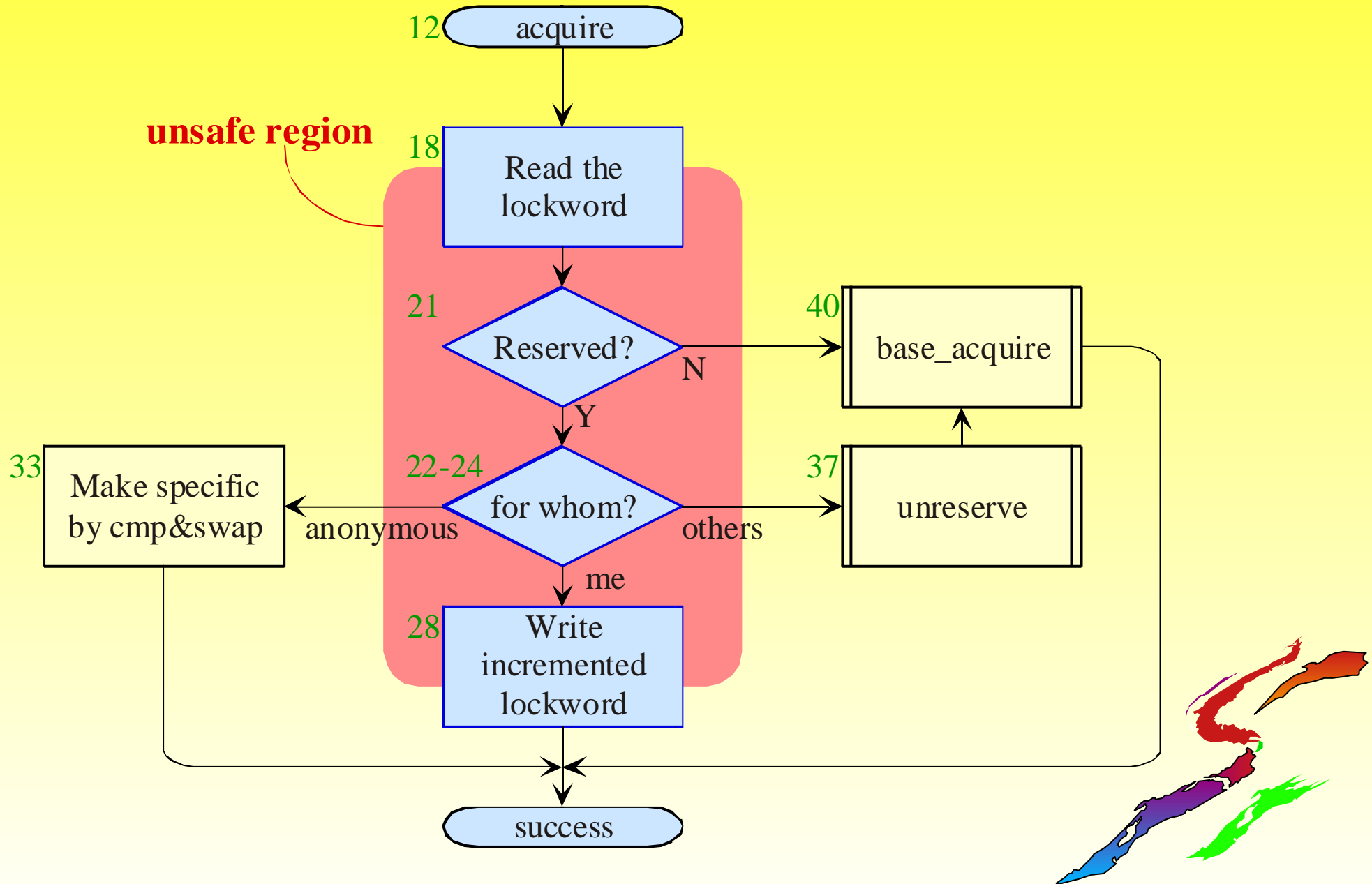
- The reservation is *anceled* when another thread tries to acquire the lock

# No Atomic Ops. in Reserve Mode

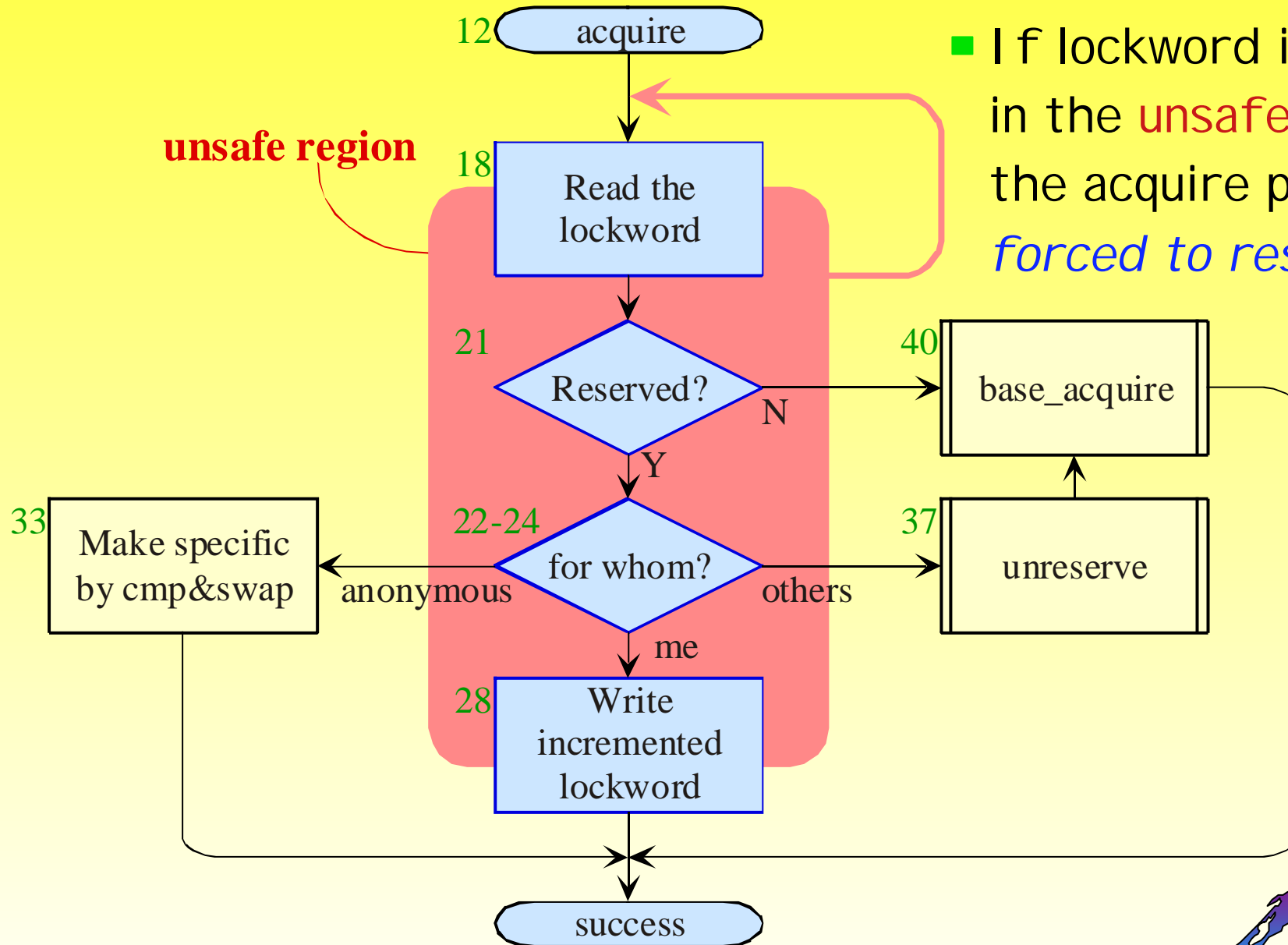


- The owner can acquire the lock by simple read-modify-write

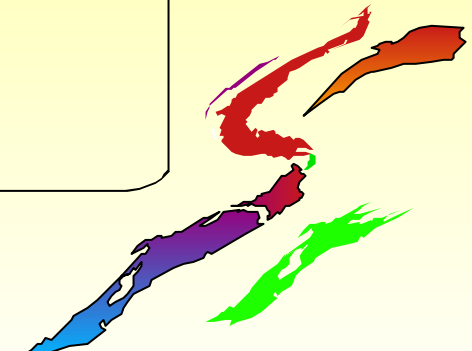
# Preserve the Consistency



# Preserve the Consistency

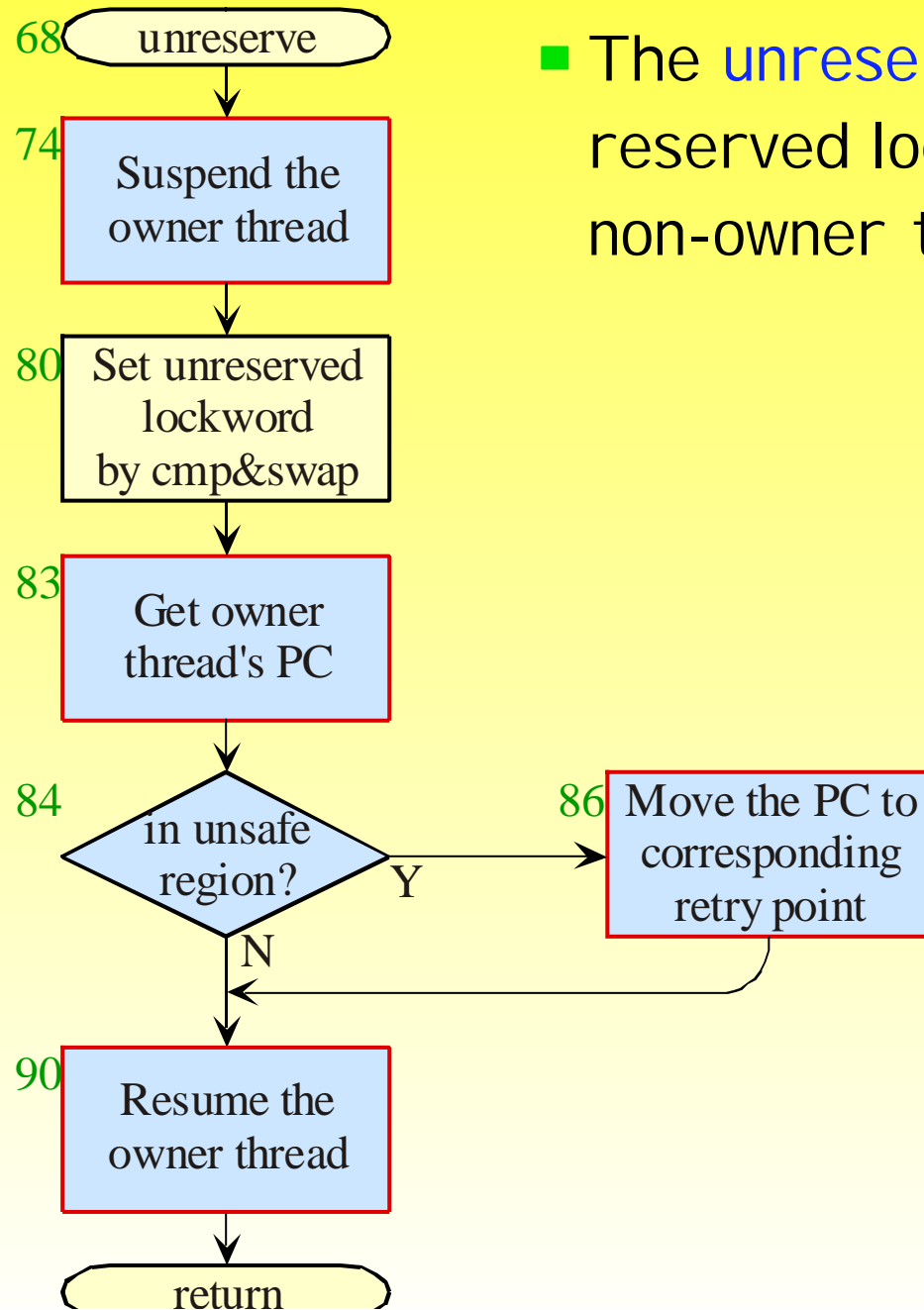


■ If lockword is modified in the **unsafe region**, the acquire process is *forced to restart*



\* These stories are basically same for release

# Reservation Cancellation (unreserve)



- The **unreserve** is the only point that reserved lockword is modified by non-owner threads



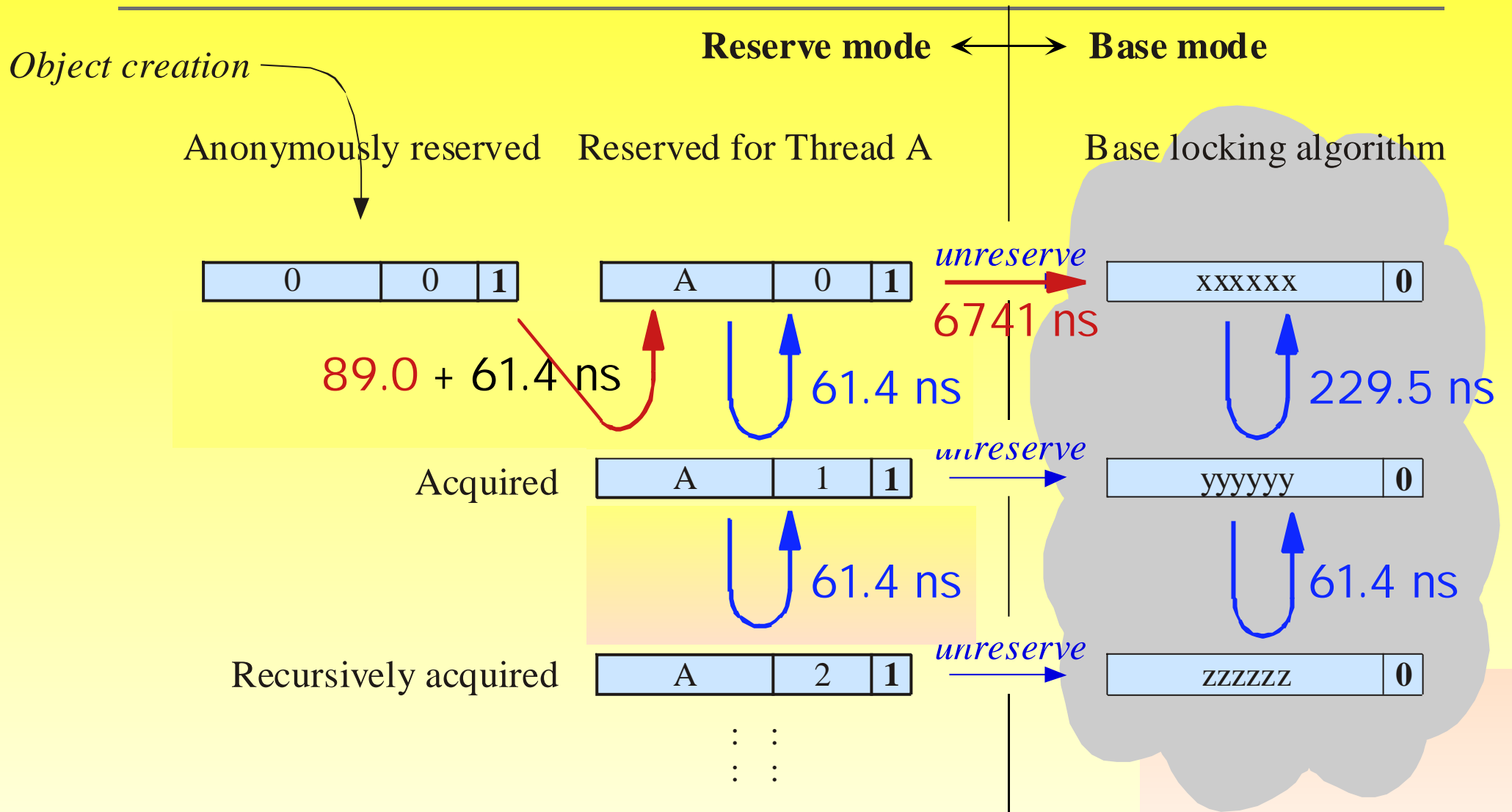
# Performance Characteristics

---

- Performance in each mode
  - When the reservation succeeds, performance is expected to improve
  - When there is no reservation, the lock is handled by the base algorithm with almost no additional overhead
  - Reservation cancellation is the only concern ...
    - However, the cancellation occurs at most once during the lifetime of an object
  
- Performance measurements
  - Implemented Lock reservation on IBM DK for Windows, Java Technology Edition, 1.3.1
  - Measured performance on Pentium4 Xeon 1.7GHz x 2, 1024MB Memory, Windows 2000 SP2

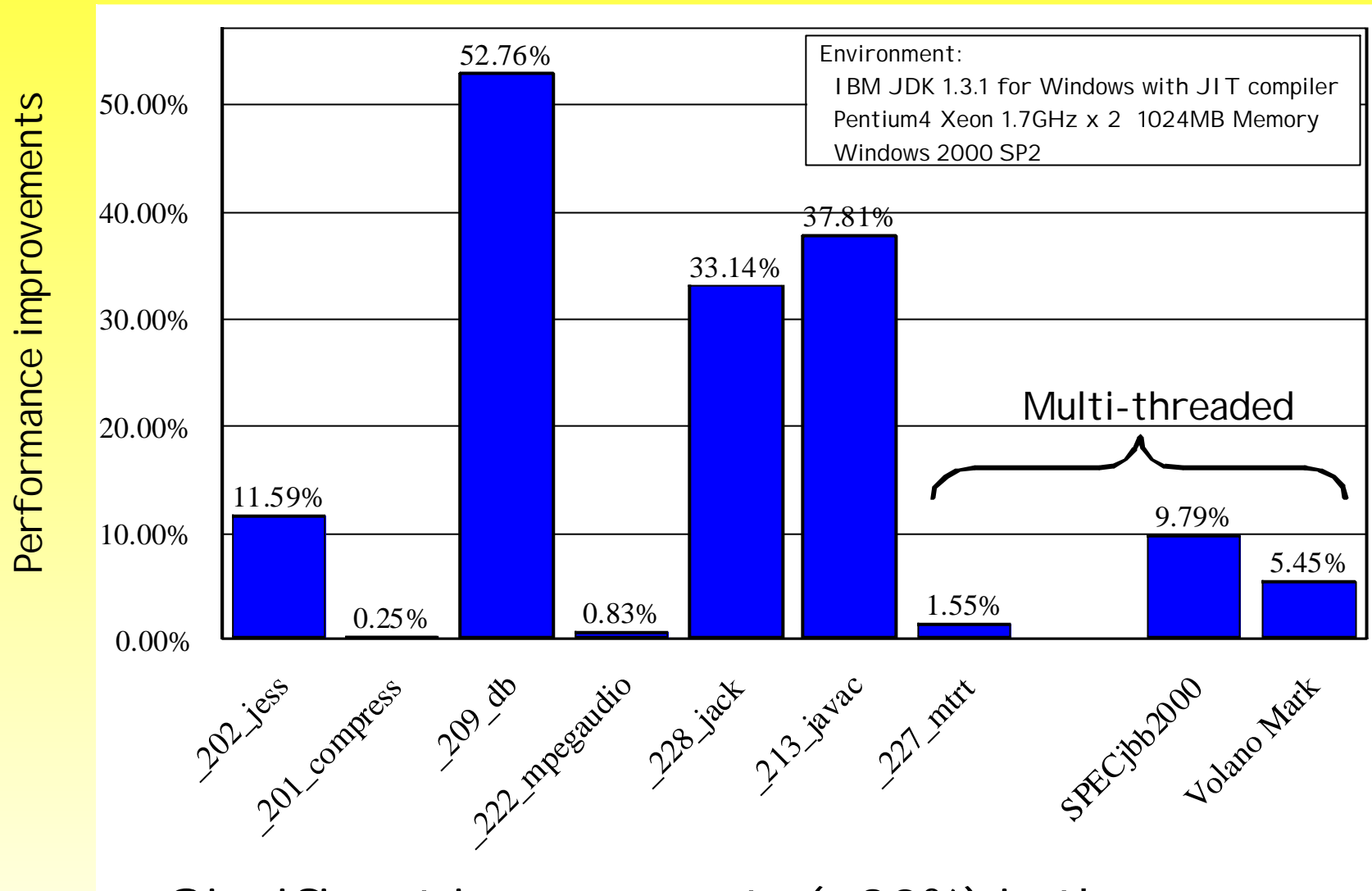


# Micro-Benchmarks (Tables 3,4)

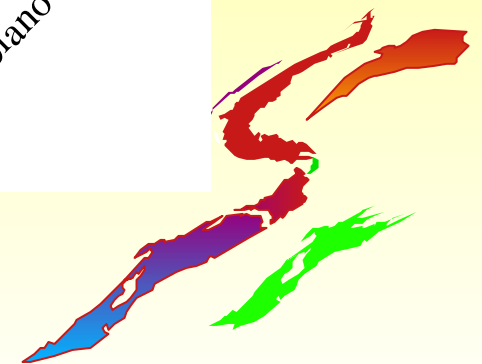


- The outermost lock cost is reduced by more than 70%
- Cancellation is quite expensive

# Macro-Benchmarks (Figure 6)



- Significant improvements (> 30%) in three SPECjvm98 programs
- 2-10% improvement even for multi-threaded programs



# Lock Statistics (Table 5)

Program name	Number of lock ops.	Ratios of accelerated locks	Ratios of cancellations
SPECjvm98			
_202_jess	14,585,409	99.289%	0.00125%
_201_compress	29,150	31.547%	0.419%
_209_db	162,079,177	99.963%	0.0000296%
_222_mpegaudio	27,480	35.837%	0.313%
_228_jack	35,207,339	91.947%	0.000395%
_213_javac	43,510,883	99.402%	0.00403%
_227_mtrt	3,523,262	99.035%	0.00284%
SPECjbb2000	335,718,621	58.544%	0.0535%
Volano Server	6,862,014	79.755%	0.0248%
Volano Client	10,381,000	84.333%	0.0138%

Multi-threaded

- Even when JIT is enabled, there still remains many locks
  - Most of them (> 58%) are accelerated by the lock reservation
- Reservation cancellation is very rare (< 0.05%)

# Concluding Remarks

---

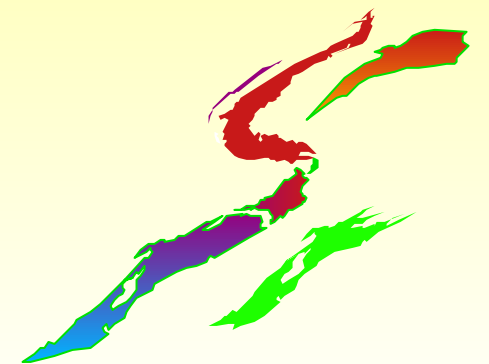
- A new lock algorithm for Java --- [lock reservation](#)
  - Optimizes Java locks by exploiting their *thread locality*
  - Allows an object's lock to be reserved for a specific thread
- Performance measurements
  - Reduced the lock cost by more than 70%
  - Accelerated more than 58% of lock ops., and achieved up to 53% performance improvements
- Future work
  - Reduce the cancellation cost
  - Refine the reservation policy
    - Avoid reservation of specific classes or creation points
  - Pursue re-reservations



# Thank You

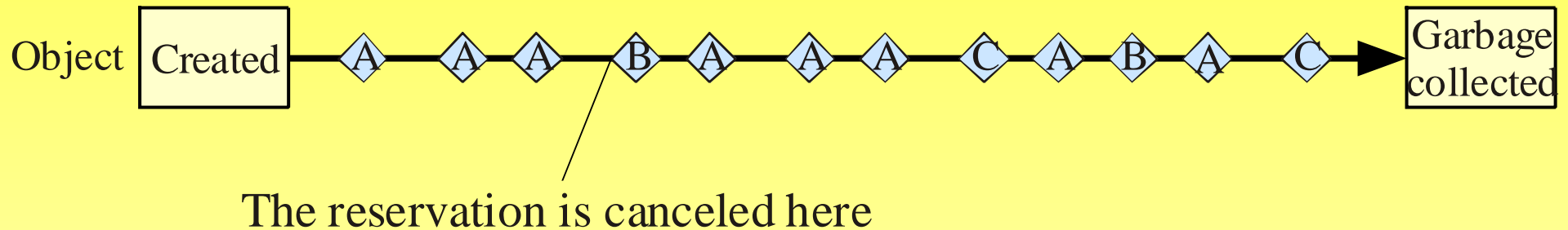
---

- Any question?



# After the Cancellation

- Locks are not *re-reserved*



Because:

- The algorithm for re-reservation would become too complicated
- There is enough thread locality even without re-reservation
- Re-reservation may cause more cancellations, which degrades performance

