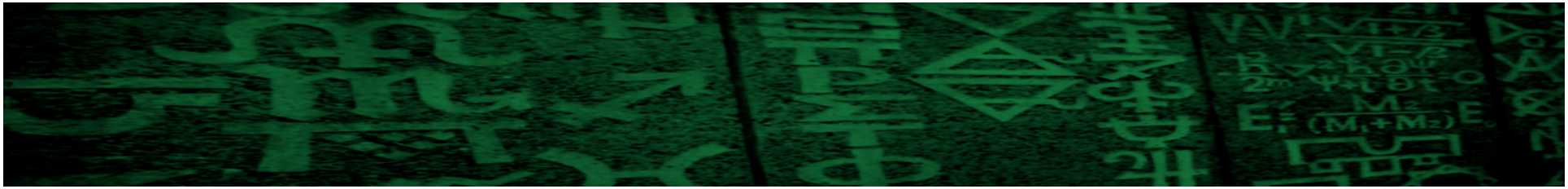
Parallel and Concurrent Real-time Garbage Collection

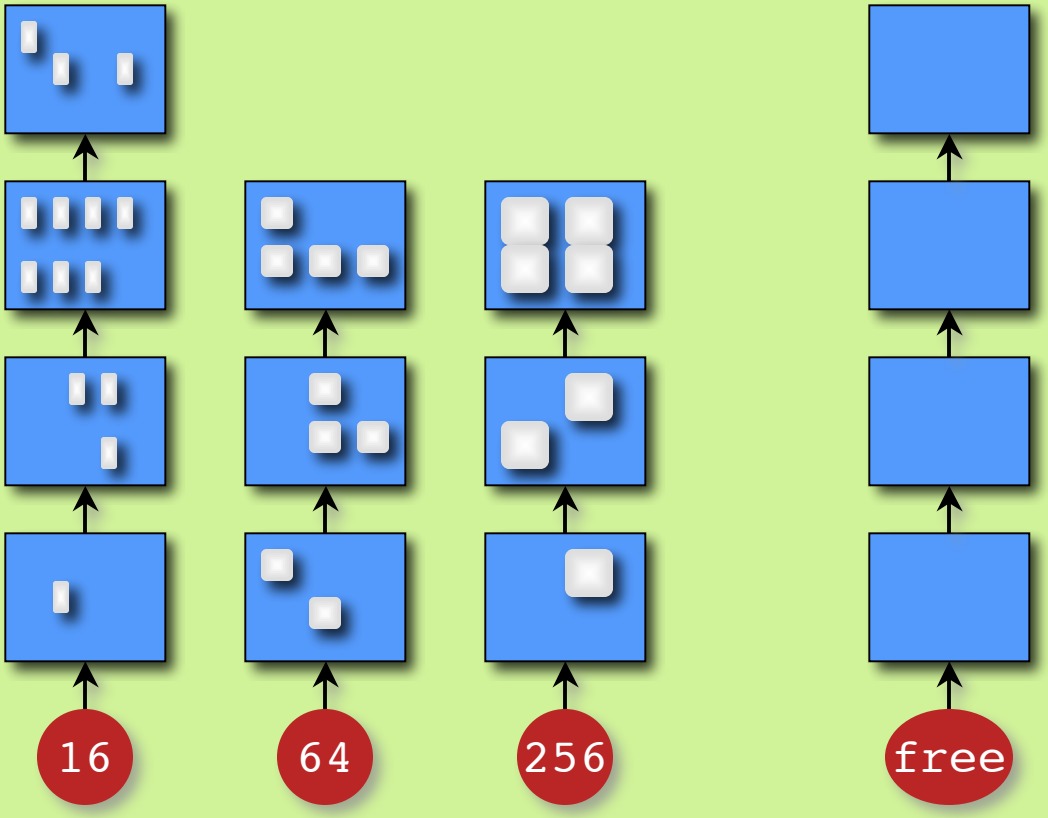
Part II:
Memory Allocation and Sweeping

David F. Bacon

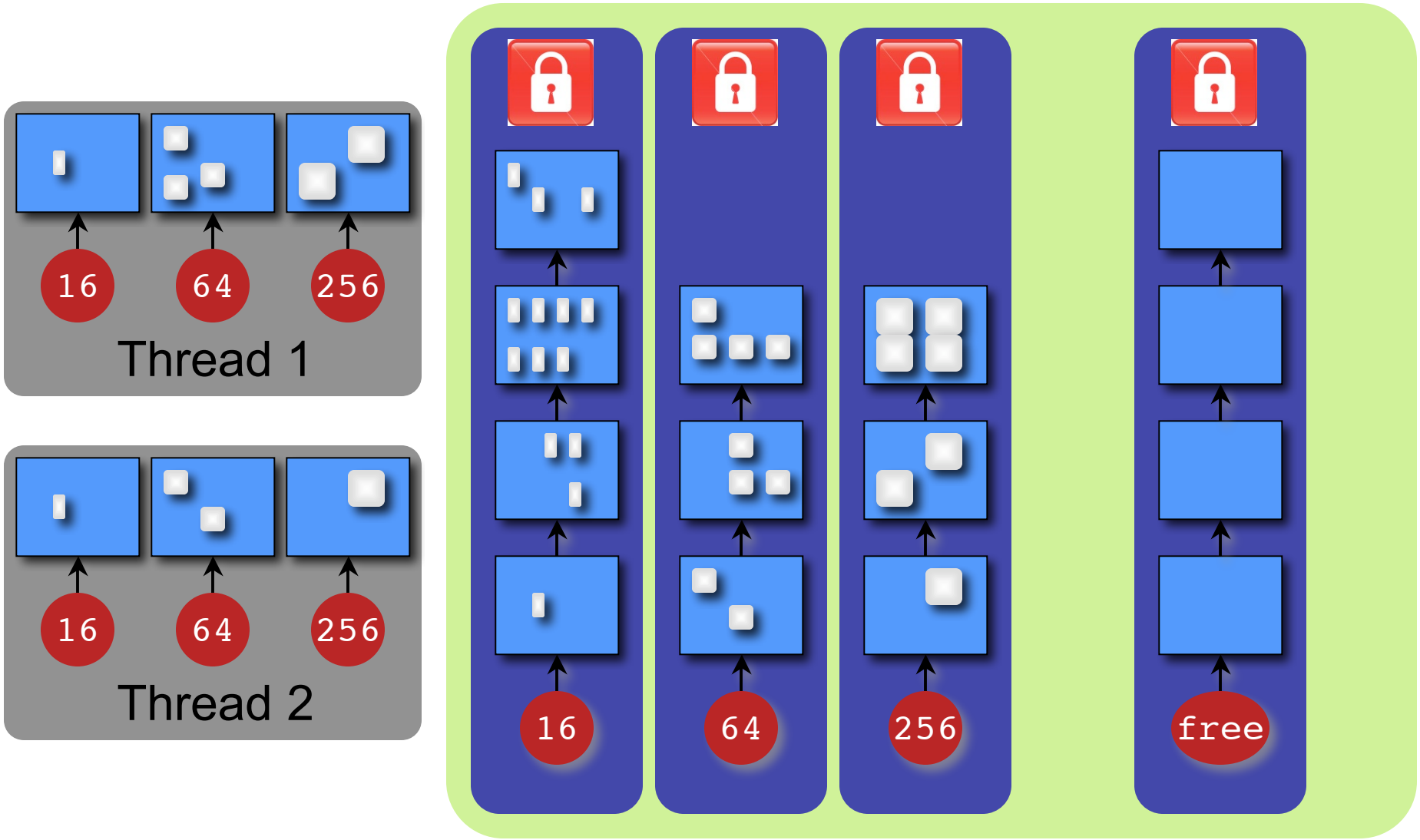
T.J. Watson Research Center

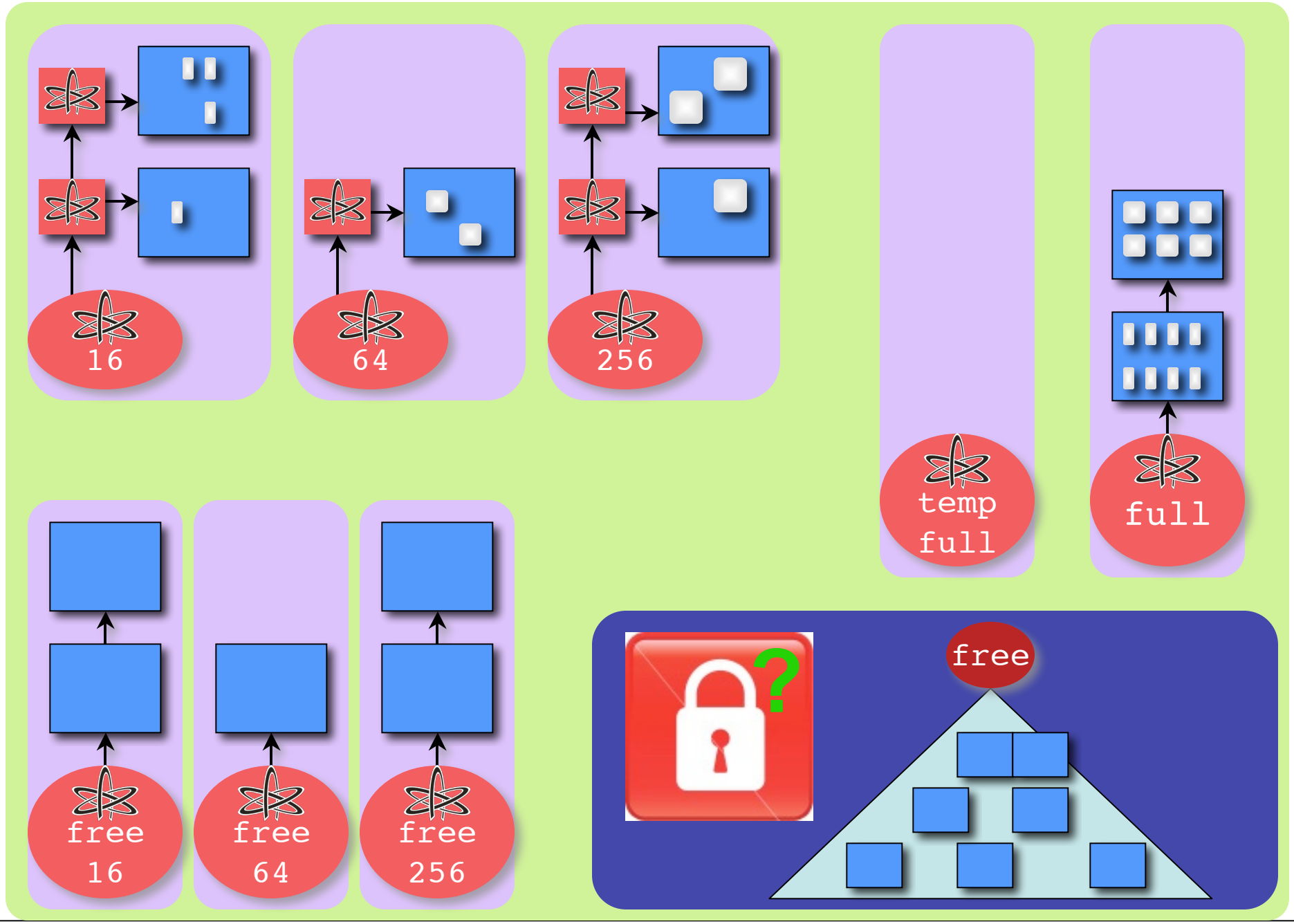


Page Data Synchronization, Take 1



Page Data, Take 2




16
64
256
temp
full
full
free
16
free
64
free
256
?
free

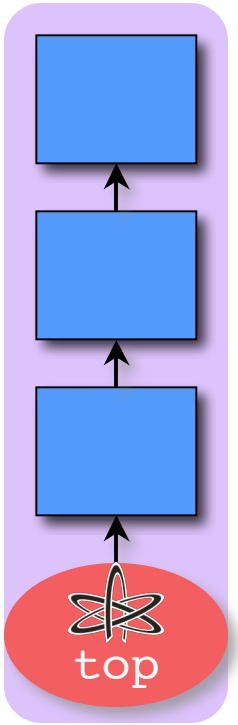
Compare-and-Swap*

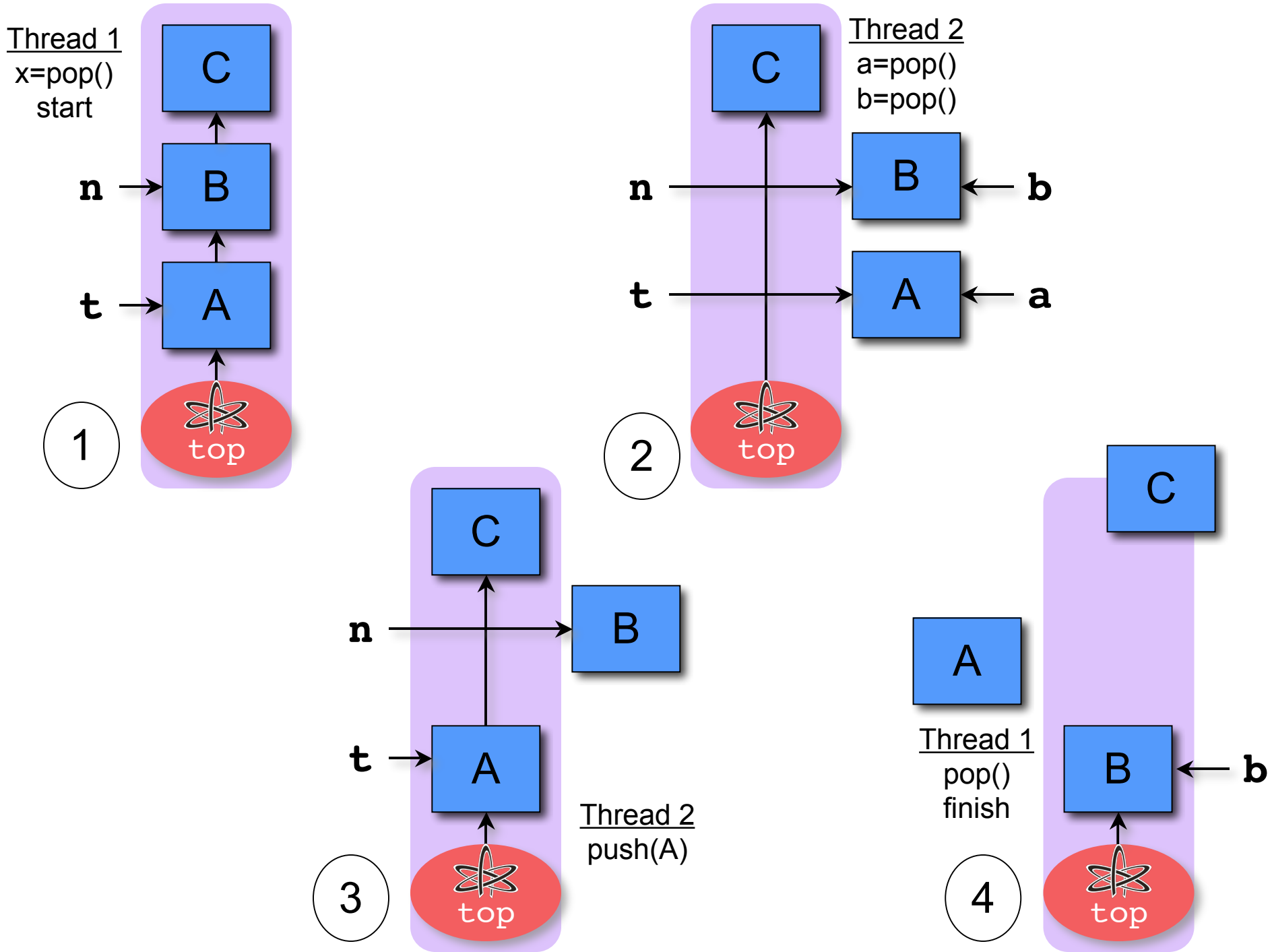
```
boolean CAS(int* addr, int old, int new) {
  atomic {
    if (* addr == old) {
      *addr = new;
      return true;
    } else {
      return false;
    }
  }
}
```

- IBM 370 since 1970, Intel 80486 since 1989
- Load-Reserved: ARM 6 since 1991, IBM POWER2 since 1993

Non-locking* Stack

```
push(Page page) {
  do {
    Page t = top;
    page->next = t;
  } while (! CAS(&top, t, page);
}

Page pop() {
  Page t = null;
  do {
    t = top;
    if (t == null) return;
    Page n = t->next;
  } while (! CAS(&top, t, n);

  return t;
}
```




Thread 1
x=pop()
start
C
B
A
n
t
top
1
Thread 2
a=pop()
b=pop()
b
a
2
Thread 2
push(A)
3
Thread 1
pop()
finish
4

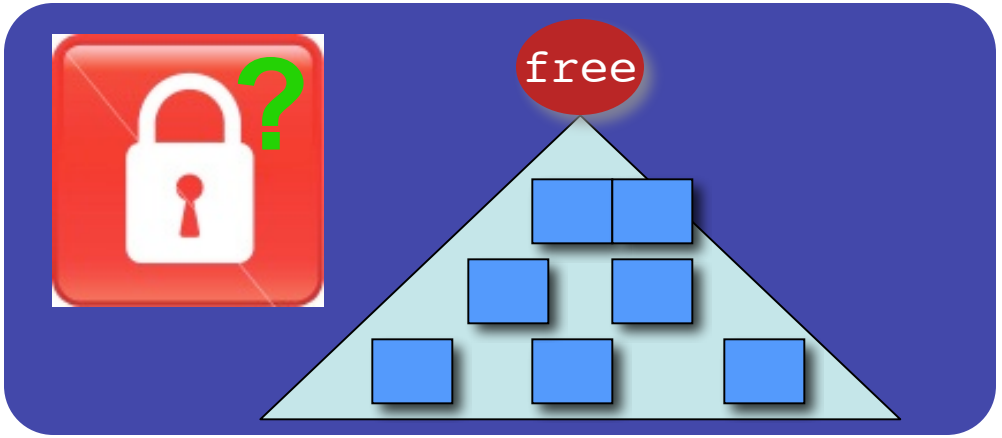
~~Correct~~ Non-locking Stack

```
push(PageId page) {
  do {
    <PageId t, short v> = top;
    page->next = t;
  } while (! CAS(&top, <t,v>, <page,(v+1) % 0xffff>);
}

PageId pop() {
  PageId t = 0;
  do {
    <PageId t, short version> = top;
    if (t==0) return 0;
  } while (! CAS(&top, <t,v>, <next(t),(v+1) % 0xffff>);

  return t;
}
```


Really Correct Non-locking Stack

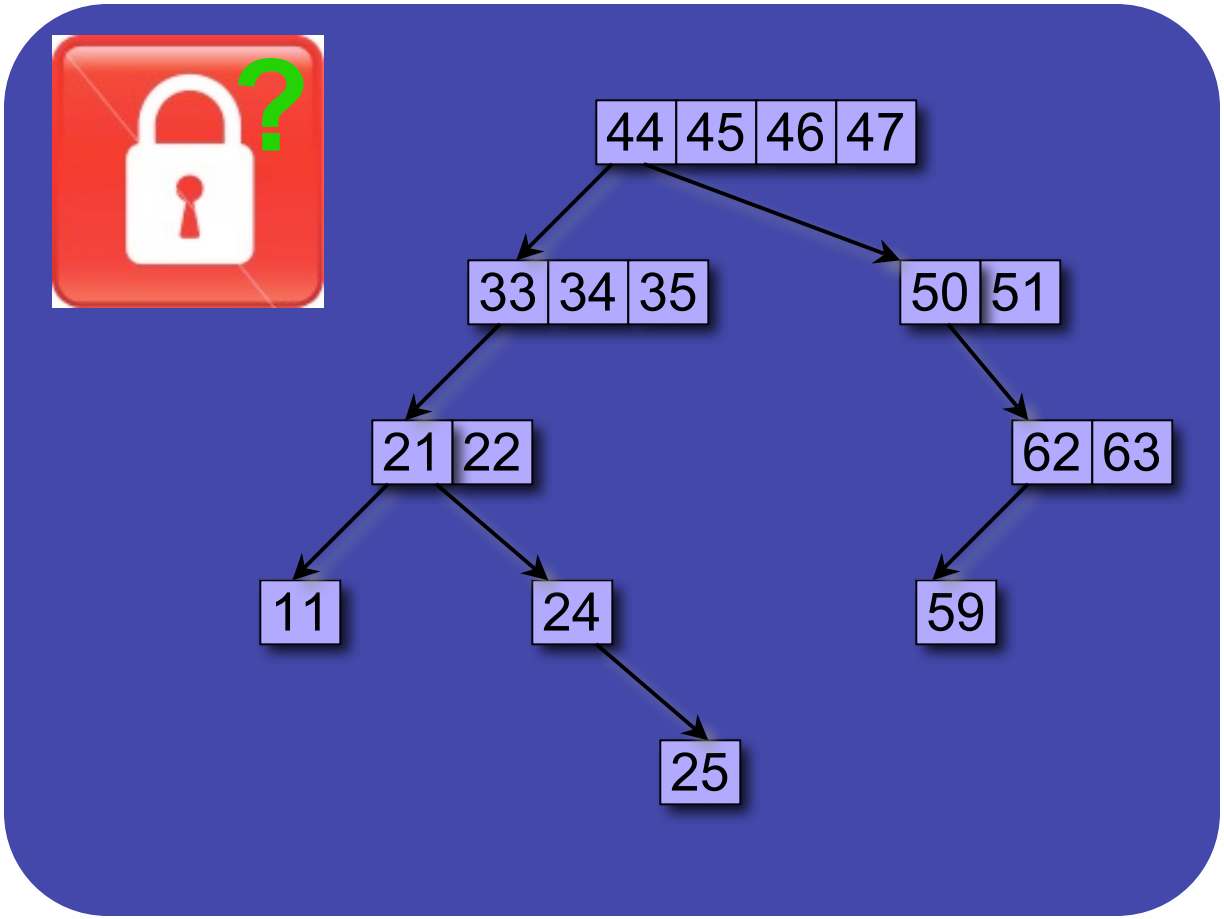
```
push(Page page) {
  do {
    Page t = LOAD_AND_RESERVE(&top);
    page->next = t;
  } while (! STORE_CONDITIONAL(&top, page));
}

Page pop() {
  Page t = null;
  do {
    Page t = LOAD_AND_RESERVE(&top);
    if (t==null) return null;
  } while (! STORE_CONDITIONAL(&top, t->next));

  return t;
}
```


Coalescing



- Ideal:
  - `getMultiplePages(n)` returns best fit
  - Does not cause mutator activities to block
  - Does not cause collection activities to block
  - Does not suffer pathology under contention
- Reality: tradeoffs between
  - Quality of fit
  - Probability of blocking
  - Overhead (number of CAS's)

Cartesian Tree + Try-lock




```
x.left.size < x.size >= x.right.size
x.left.address < x.address < x.right.address
```

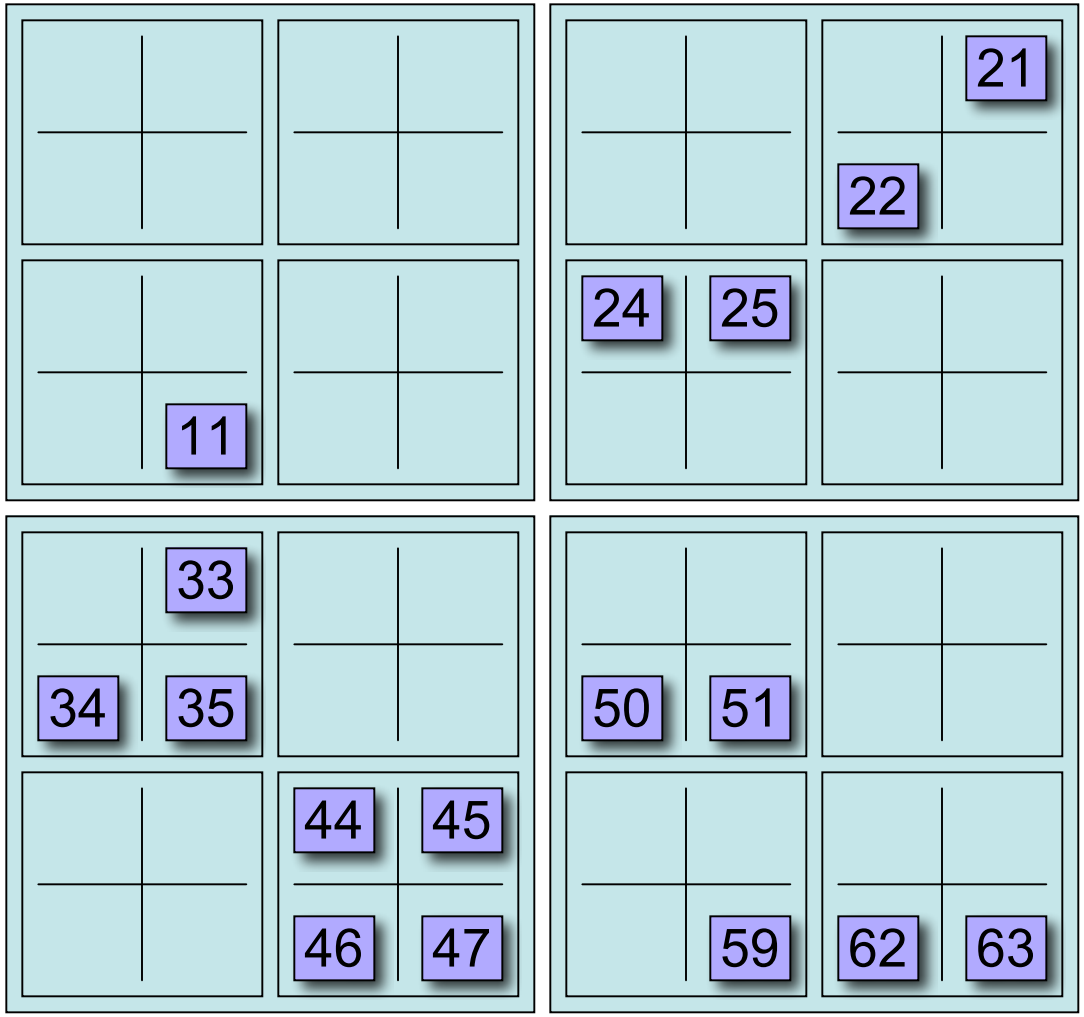
Non-locking Rebalancing Tree

- Homework…
- …but I won't use it.
- Solution would be
  - Too complex to be reliable
  - Require too many CAS operations to be fast

The Fix: Transactional Memory

```
AtomicCartesianTree extends CartesianTree {

  atomic Block getBlock() {
    return super.getBlock();
  }

  atomic void releaseBlock(Block b) {
    super.releaseBlock(b);
  }

  atomic Block[] getMultipleBlocks(int n) {
    return super.getMultipleBlocks(n);
  }
}
```


Hierarchical Non-locking Buddy?



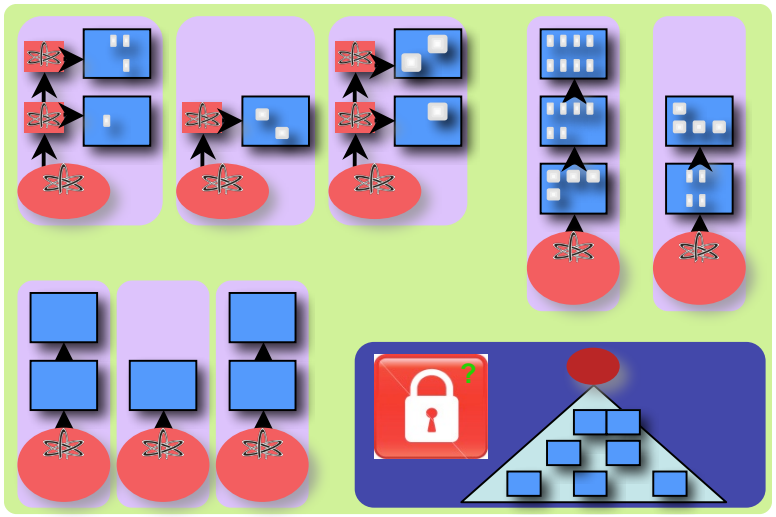
So much for allocation…

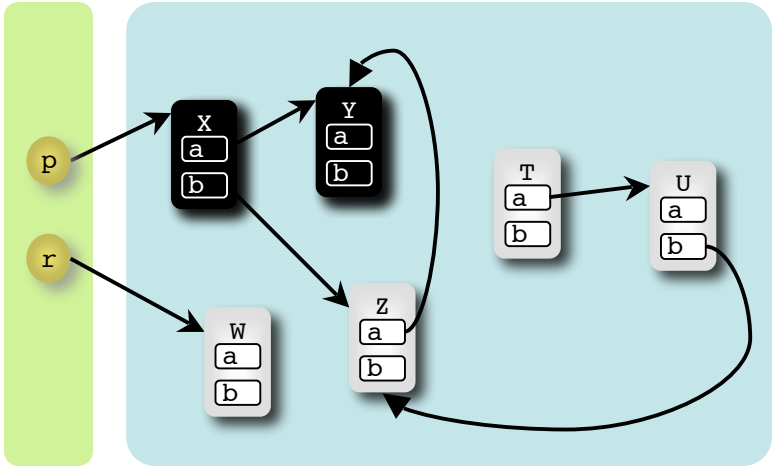
Garbage Collection

Handling the Concurrency

GC Threads

- Trace
- Collect
- Format





Application ("mutator") Threads

- Load pointer
- Store pointer
- Allocate

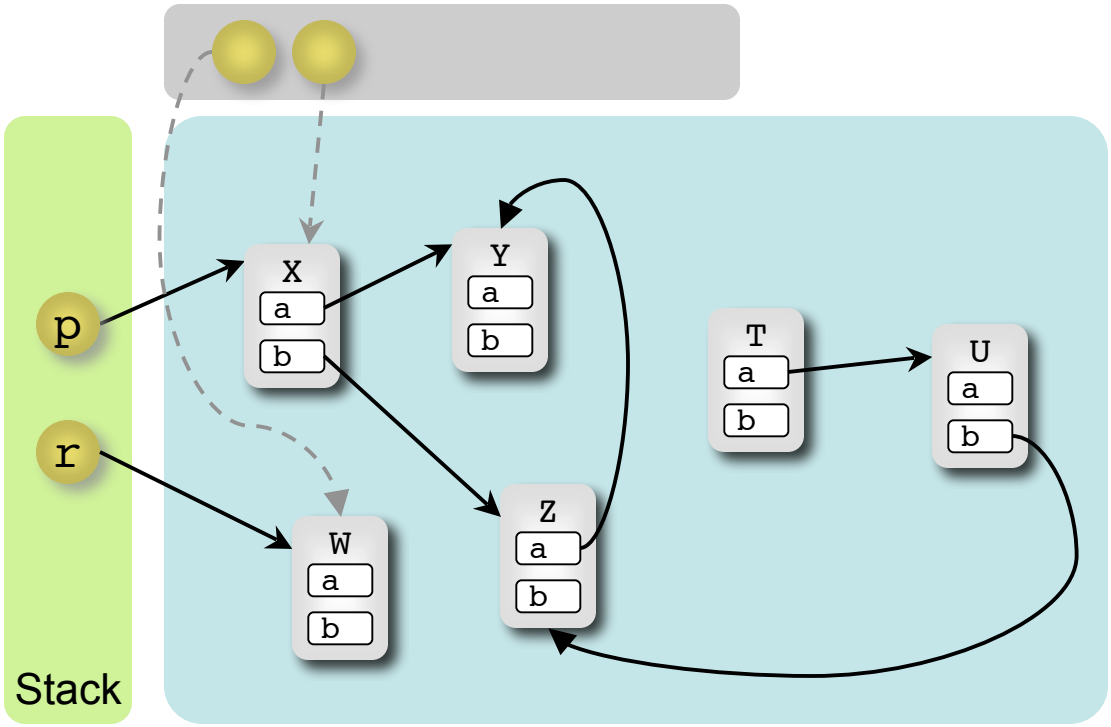
Yuasa Snapshot Algorithm (1990)

- Logically
  - Mark-and-Sweep Style Algorithm
  - Take a “copy-on-write” heap snapshot
  - Collect the garbage in that snapshot

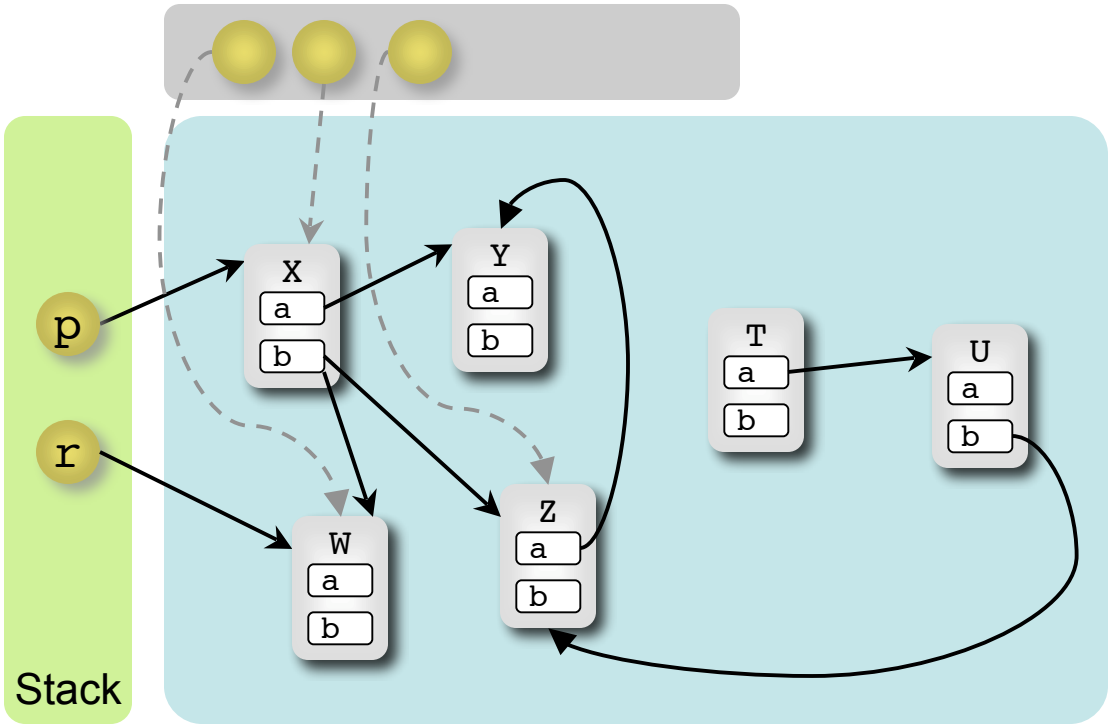
- Physically
  - Stop all threads
  - Copy their stacks (“roots”)
  - Force them to save over-written pointers
  - Trace roots and over-written pointers

* Dijktstra’75, Steele’76

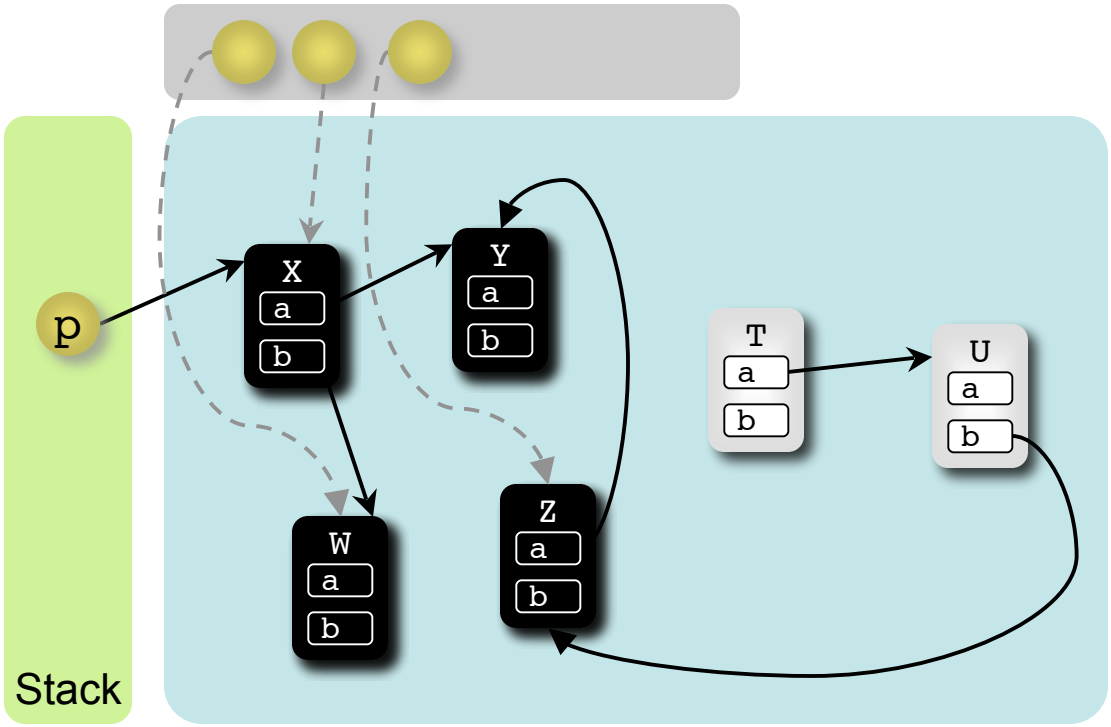
1: Take Logical Snapshot



2(a): Copy Over-written Pointers

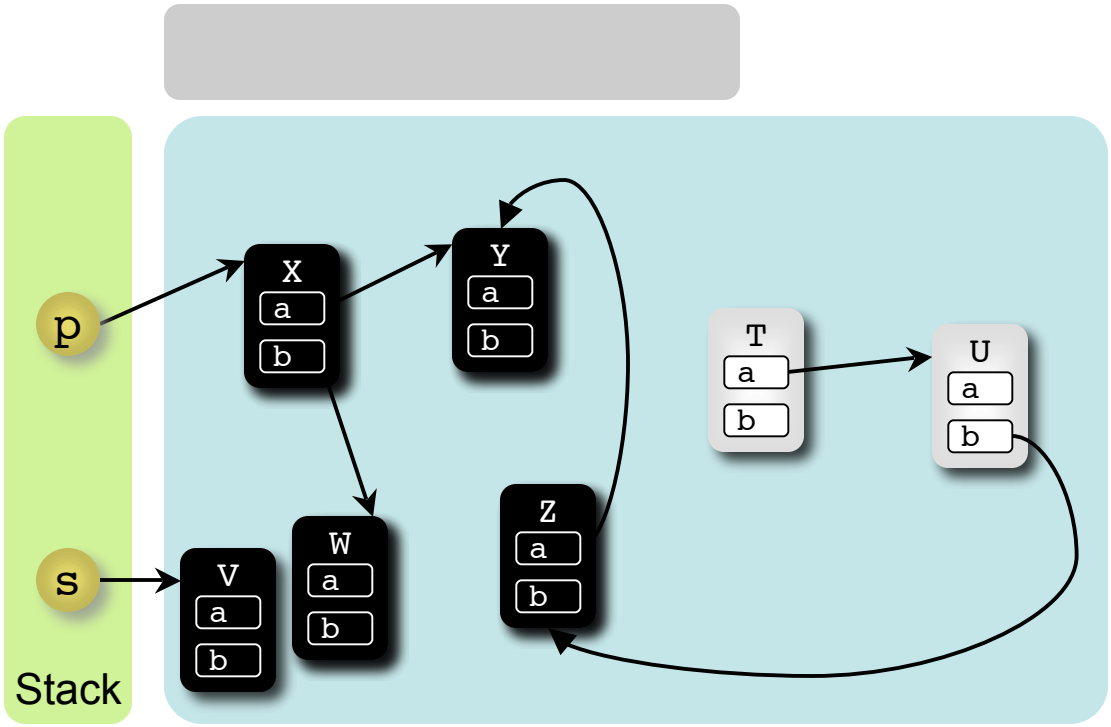


2(b): Trace

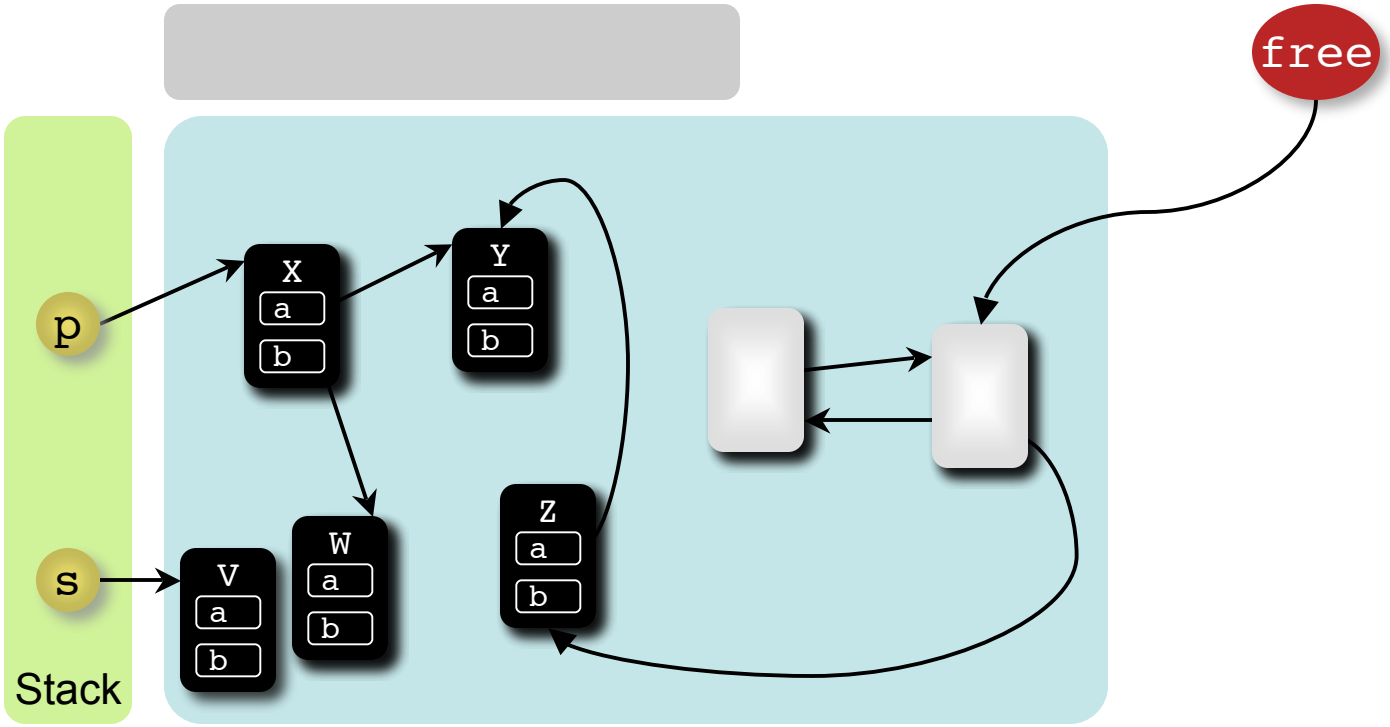


* Color is per-object mark bit

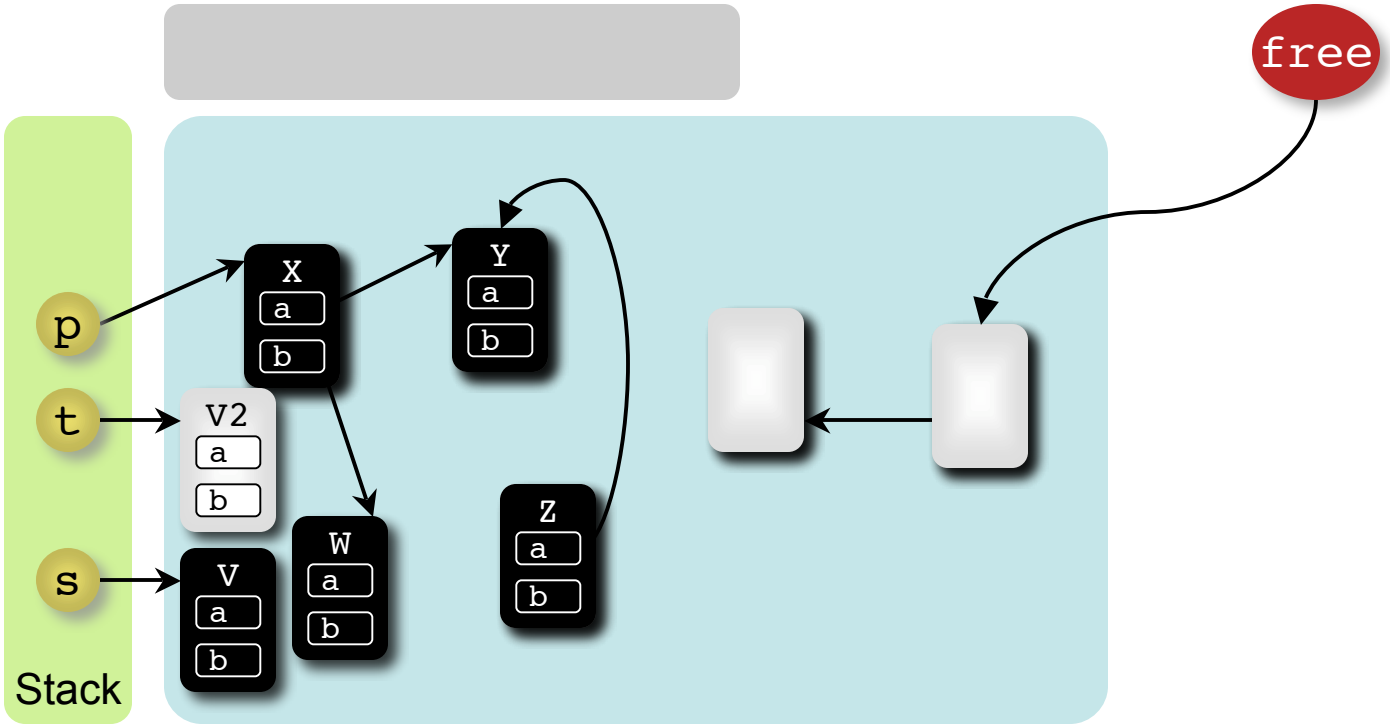
2(c): Allocate “Black”



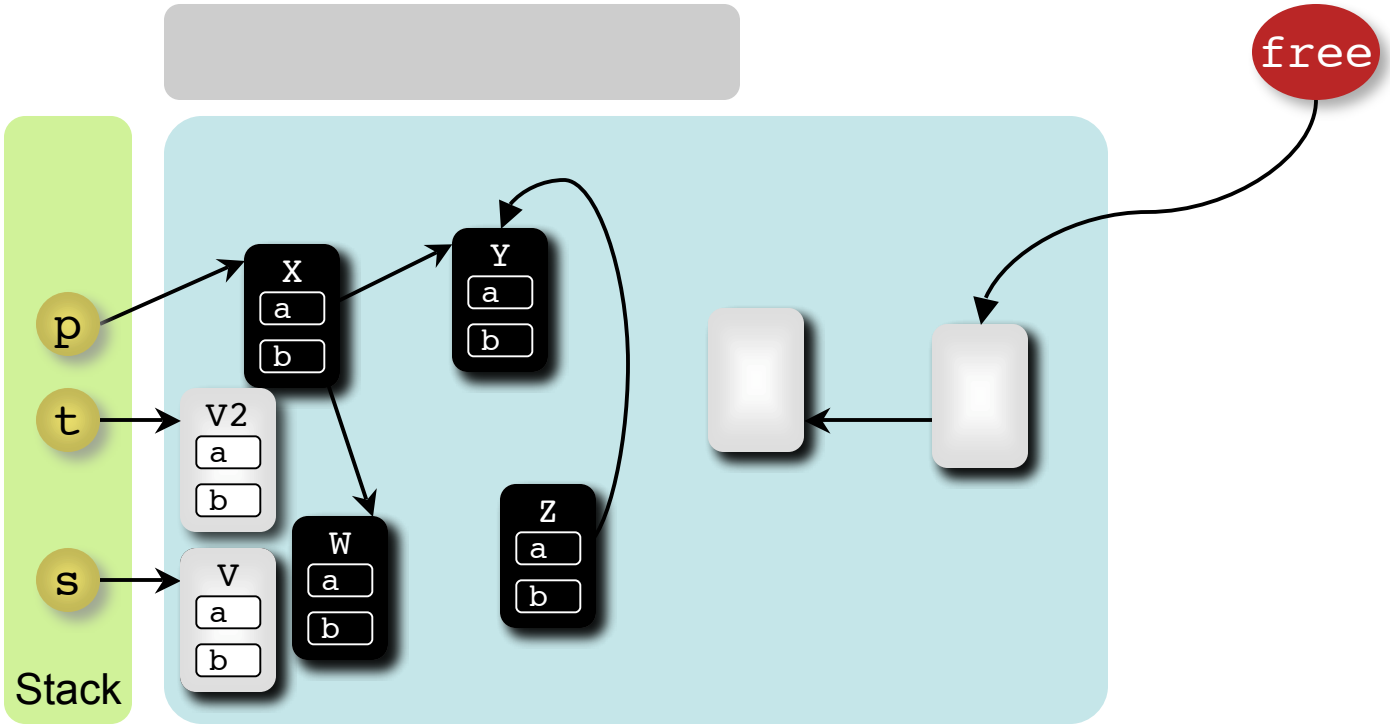
3(a): Sweep Garbage



3(b): Allocate “White”



4: Clear Marks



Yuasa Algorithm Phases

- Snapshot stack and global roots
- Trace
- Flip
- Sweep
- Flip
- Clear Marks

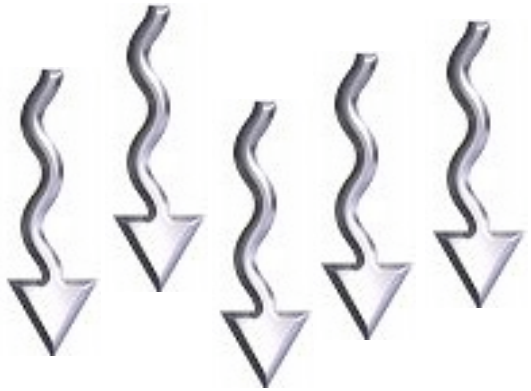
* Synchronous

Metronome-2 Concurrency

GC Master Thread

GC Worker Threads

Application Threads (may do GC work)



Metronome-2 Phases

- Initiation
  - Setup
    - turn double barrier on
- Root Scan
  - Active Finalizer scan
  - Class scan
  - **Thread scan****
    - switch to single barrier, color to black
  - Debugger, JNI, Class Loader scan
- Trace
  - **Trace***
  - **Trace Terminate*****
- Re-materialization 1
  - Weak/Soft/Phantom Reference List Transfer
  - **Weak Reference clearing**** (snapshot)
- Re-Trace 1
  - Trace Master
  - **(Trace*)**
  - **(Trace Terminate***)**
- Re-materialization 2
  - Finalizable Processing
- Clearing
  - Monitor Table clearing
  - JNI Weak Global clearing
  - Debugger Reference clearing
  - JVMTI Table clearing
  - Phantom Reference clearing
- Re-Trace 2
  - Trace Master
  - **(Trace*)**
  - **(Trace Terminate***)**
  - Class Unloading
- Flip
  - **Move Available Lists to Full List*** (contention)
    - turn write barrier off
  - **Flush Per-thread Allocation Pages****
    - switch allocation color to white
    - switch to temp full list
- Sweeping
  - **Sweep***
  - **Switch to regular Full List****
  - **Move Temp Full List to regular Full List*** (contention)
- Completion
  - Finalizer Wakeup
  - Class Unloading Flush
  - **Clearable Compaction****
  - Book-keeping

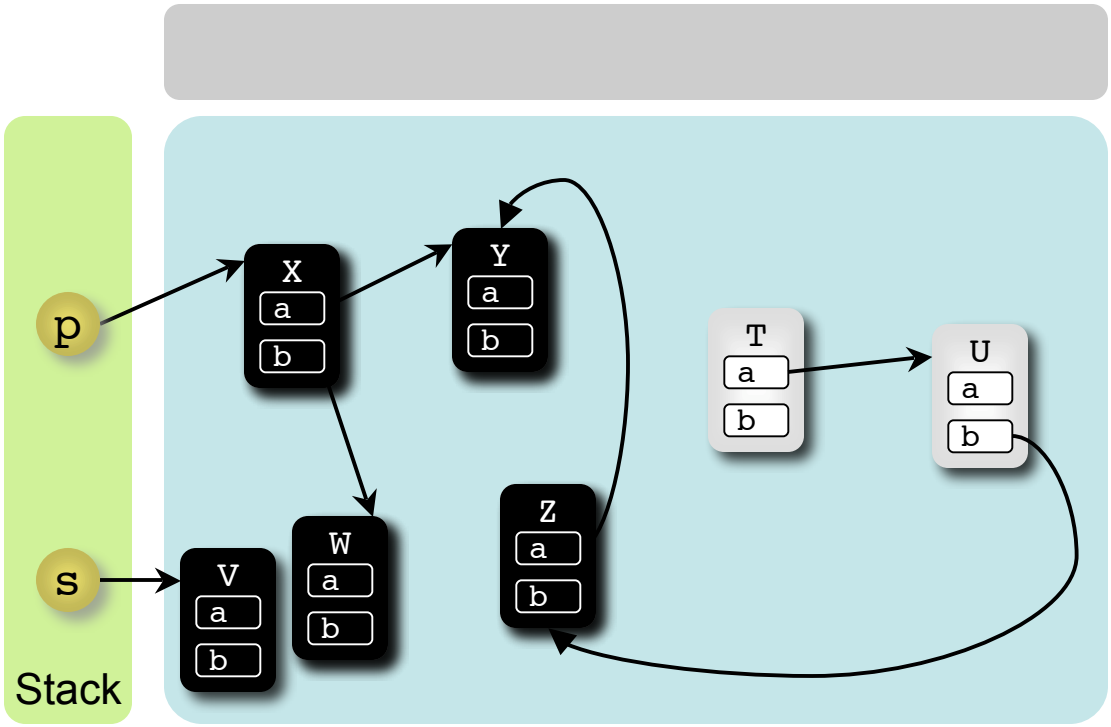
*** Parallel**
**** Callback**
***** Single actor symmetric**

Part 3: Sweep

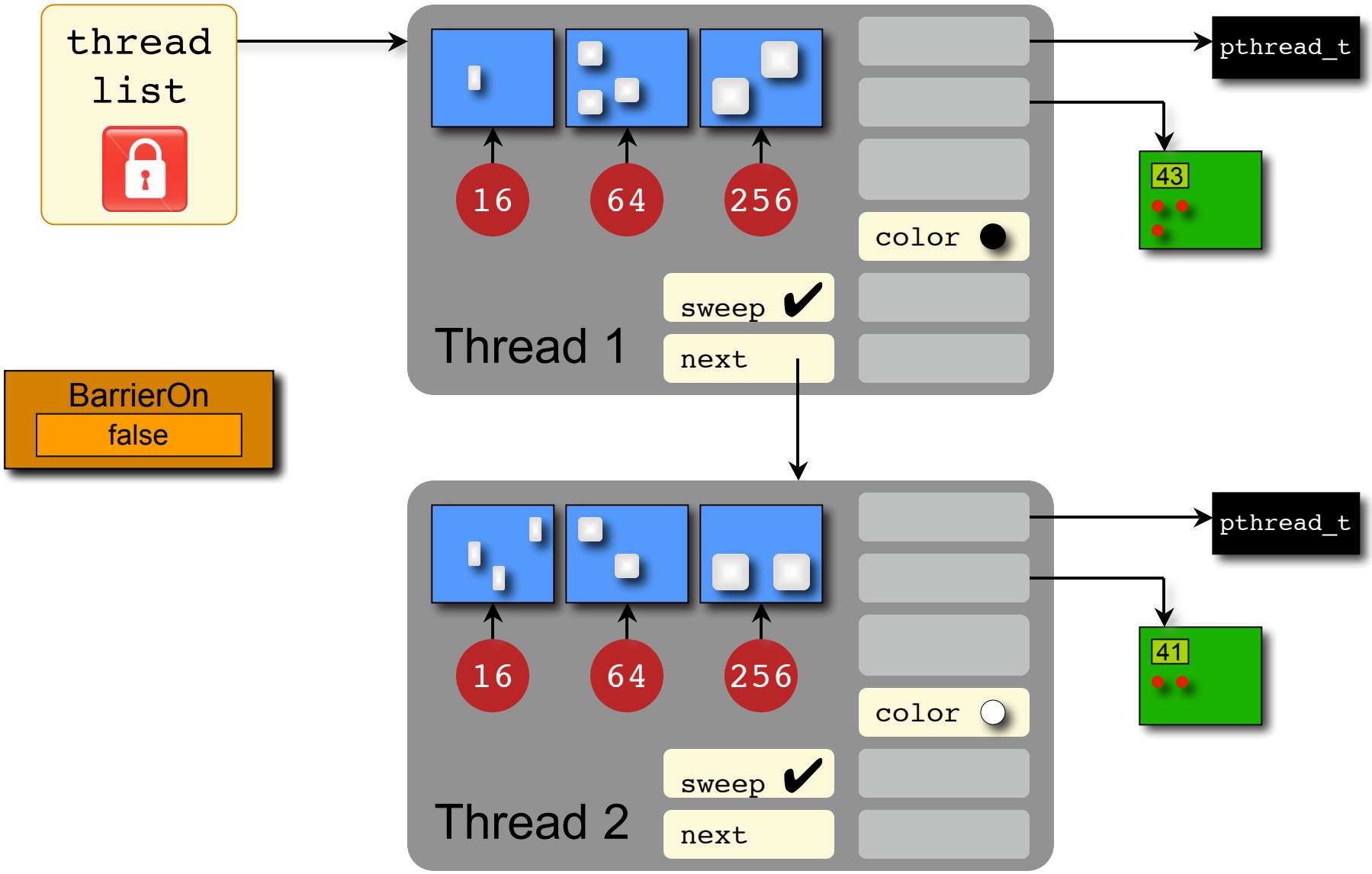
- Initiation
  - Setup
    - turn double barrier on
- Root Scan
  - Active Finalizer scan
  - Class scan
  - **Thread scan****
    - switch to single barrier, color to black
  - Debugger, JNI, Class Loader scan
- Trace
  - **Trace***
  - **Trace Terminate*****
- Re-materialization 1
  - Weak/Soft/Phantom Reference List Transfer
  - **Weak Reference clearing**** (snapshot)
- Re-Trace 1
  - Trace Master
  - **(Trace*)**
  - **(Trace Terminate***)**
- Re-materialization 2
  - Finalizable Processing
- Clearing
  - Monitor Table clearing
  - JNI Weak Global clearing
  - Debugger Reference clearing
  - JVMTI Table clearing
  - Phantom Reference clearing
- Re-Trace 2
  - Trace Master
  - **(Trace*)**
  - **(Trace Terminate***)**
  - Class Unloading
- Flip
  - **Move Available Lists to Full List*** (contention)
    - turn write barrier off
  - **Flush Per-thread Allocation Pages****
    - switch allocation color to white
    - switch to temp full list
- Sweeping
  - **Sweep***
  - **Switch to regular Full List****
  - **Move Temp Full List to regular Full List*** (contention)
- Completion
  - Finalizer Wakeup
  - Class Unloading Flush
  - **Clearable Compaction****
  - Book-keeping

*** Parallel**
**** Callback**
***** Single actor symmetric**

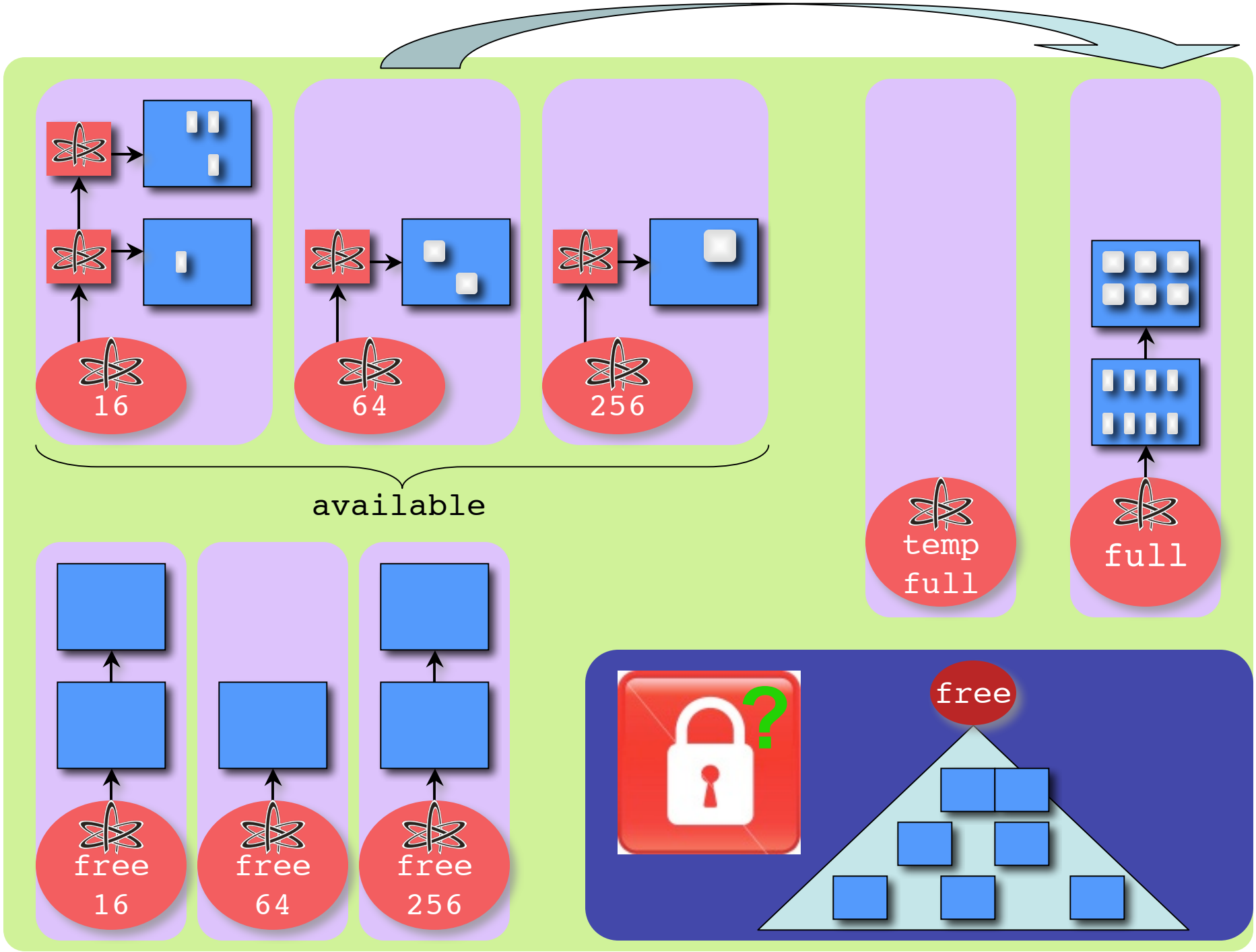
Assume Trace Has Finished

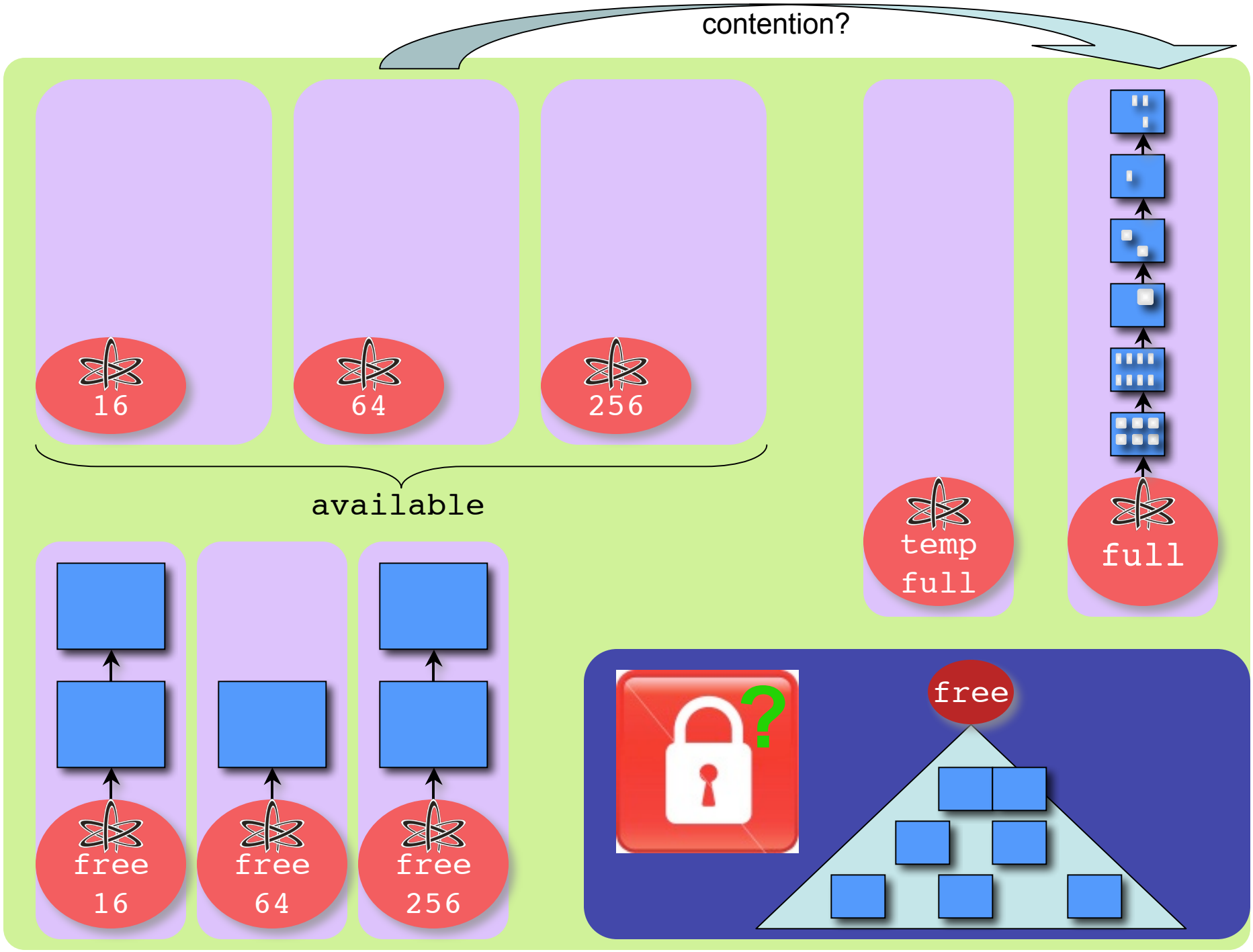


JVM Application Thread Structure



“Move Available” Phase


16
64
256
available
temp
full
full
free
16
free
64
free
256
free


contention?
16
64
256
available
temp
full
full
free
16
free
64
free
256
free

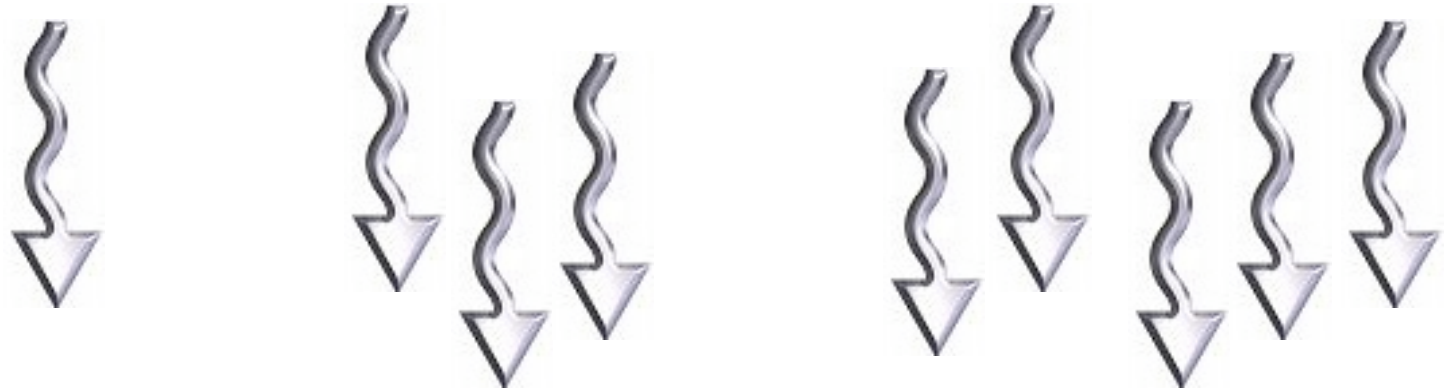
Generic Structure of Move Phase:

Shared Monotonic Work Pool

GC Master Thread

GC Worker Threads

Application Threads (may do GC work)

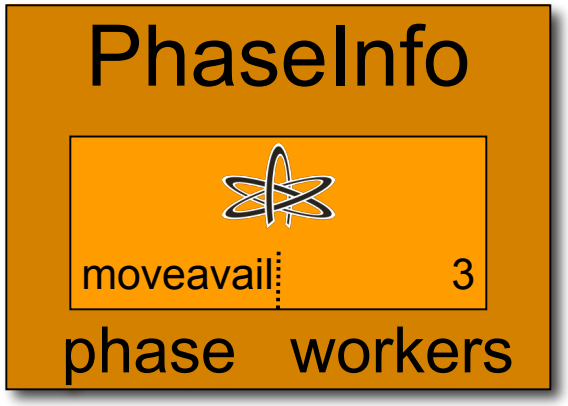


pool of work

Performing a GC Work Quantum

```
every (500us)
  tryToDoSomeWorkForGC();


tryToDoSomeWorkForGC() {
  short phase = enterPhase();
  if (phase > 0)
    doQuantum(phase);
    leavePhase();
}
```

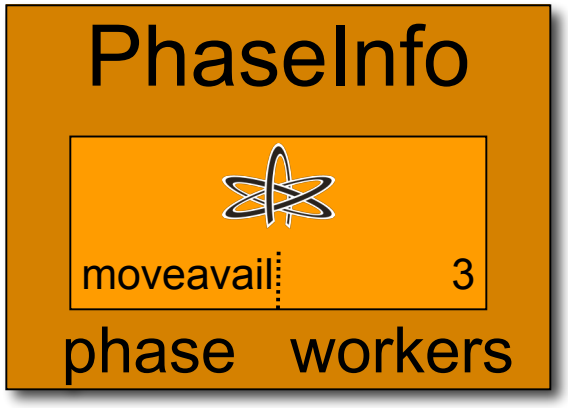

Phase Entry




```
short enterPhase() {
  thread->epoch = Epoch.newest;

  while (true)
    <short phase, short workers> = PhaseInfo;

    if (workers == phaseTable[phase].maxWorkers)
      return -1;

    if (CAS(PhaseInfo,<phase,workers>,<phase,workers+1>))
      return phase;
}
```

ABA?

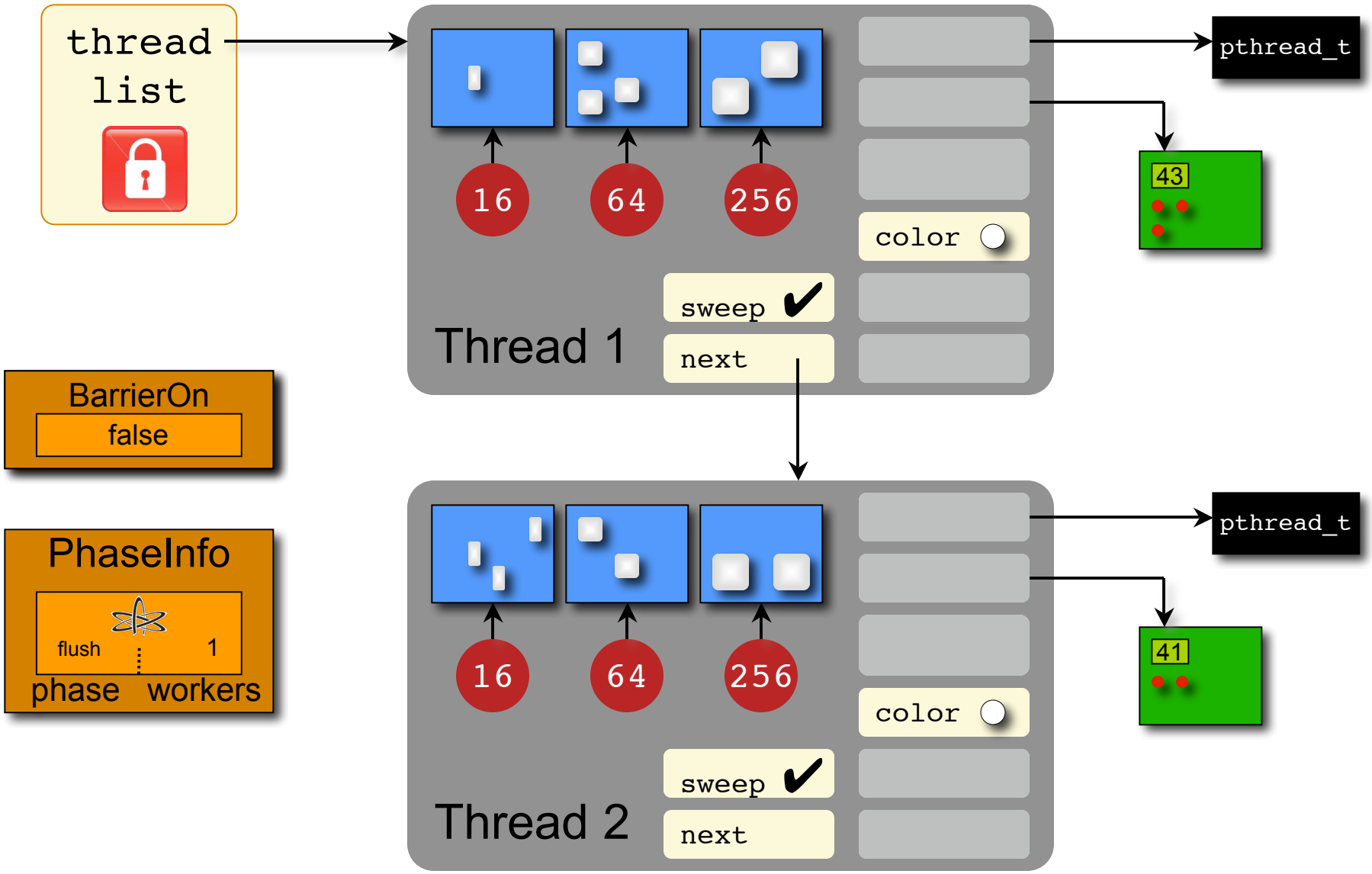
Phase Exit



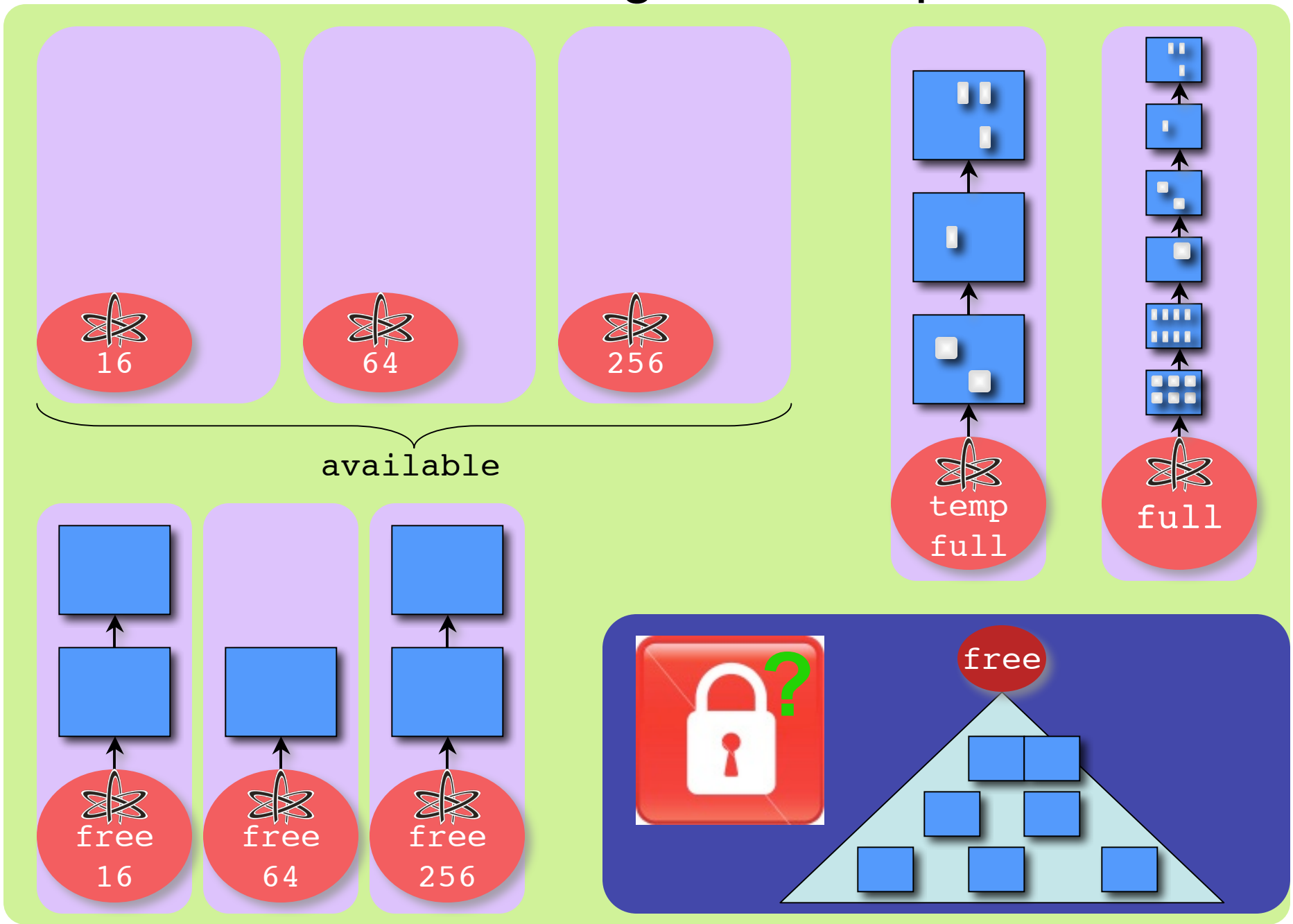

```
short leavePhase() {
  while (true)
    <short phase, short workers> = PhaseInfo;
    if (workers > 1 || moreWorkForPhase(phase))
      newPhase = <phase,workers-1>;
    else if (phase == TRACE_PHASE)
      newPhase = <phaseTable[phase].nextPhase, 1>;
    else
      newPhase = <phaseTable[phase].nextPhase, 0>;

    if (CAS(PhaseInfo, <phase,workers>, newPhase))
      return newPhase;
}
```

“Per-Thread Flush” Phase

Flush Per-Thread Pages



Put Full White Pages on temp-full List



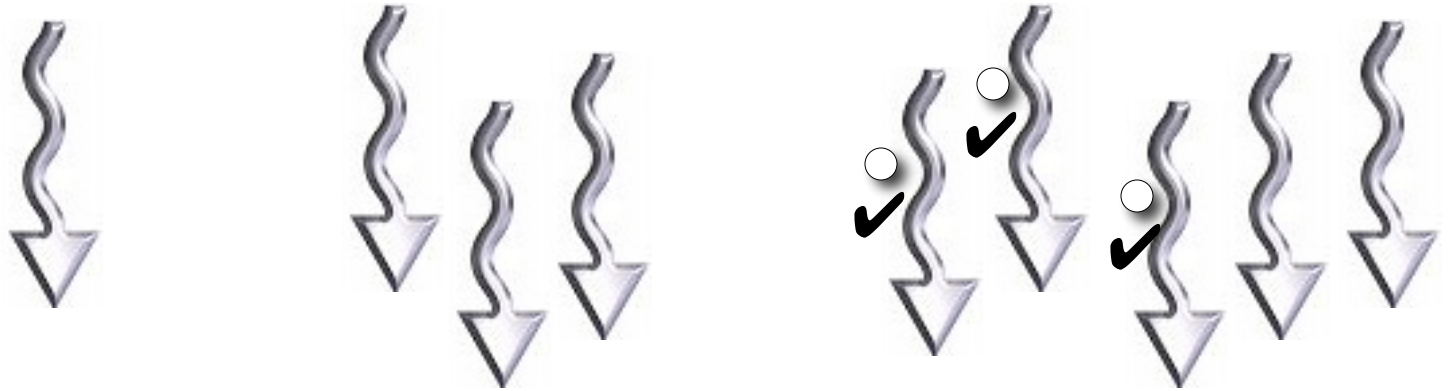
How is this Phase Different?

Per-Thread State Update

GC Master Thread

GC Worker Threads

Application Threads (may do GC work)



~~pool of work~~

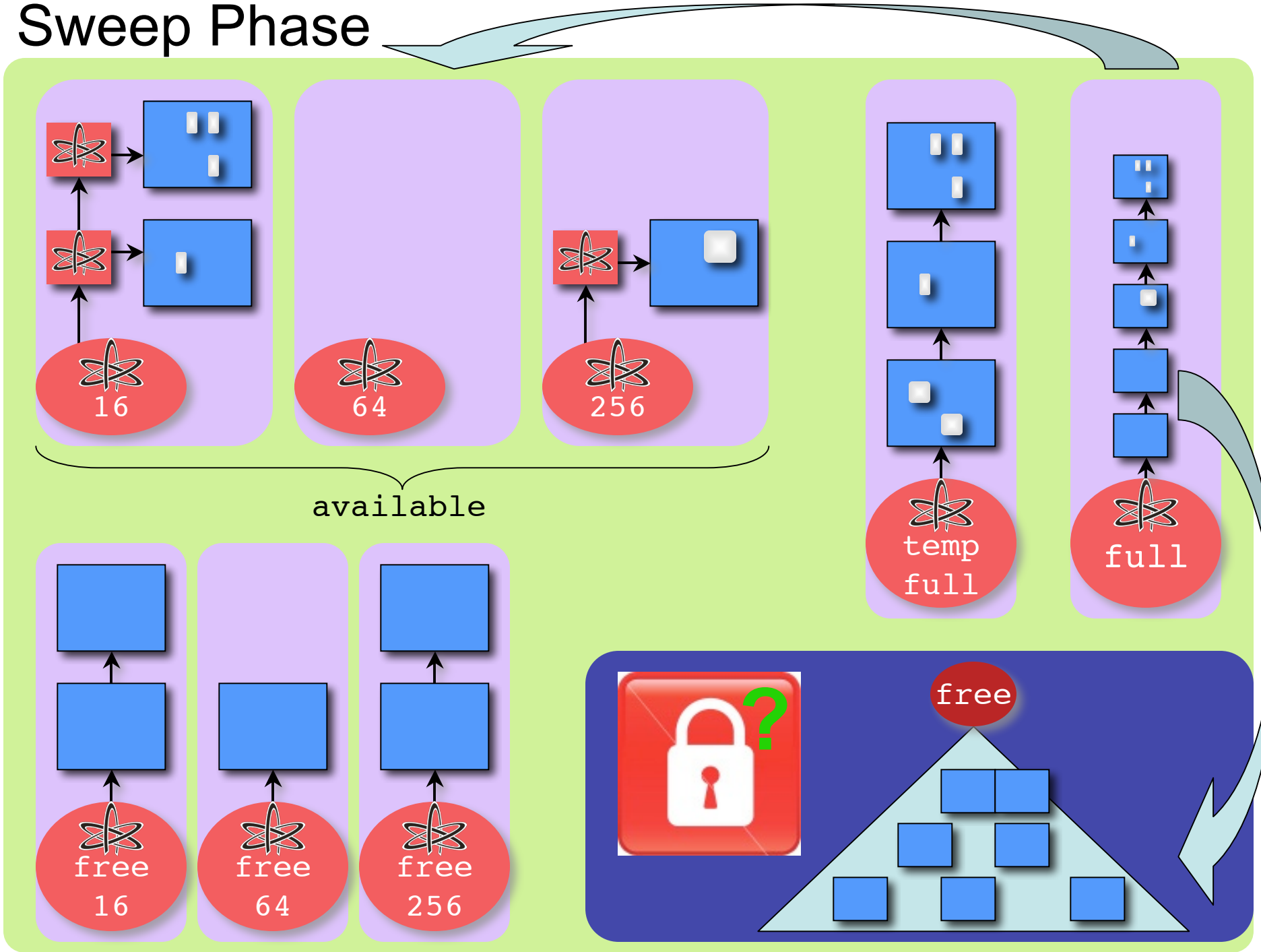
Callback Mechanism

```
bool doCallbacks(int callbackId) {
  bool alldone = true;

  LOCK(threadlist);
    for each (Thread thread in threadlist)
      if (hasCallbackWork(thread, callbackId);
        if (! thread.inVM && TRY_LOCK(thread))
          doCallbackWorkForThread(thread, callbackId);
          UNLOCK(thread);
        else
          postCallback(thread, callbackId);
          alldone = false;
  UNLOCK(threadlist);

  return alldone;
}
```

* Non-locking list with cursor?

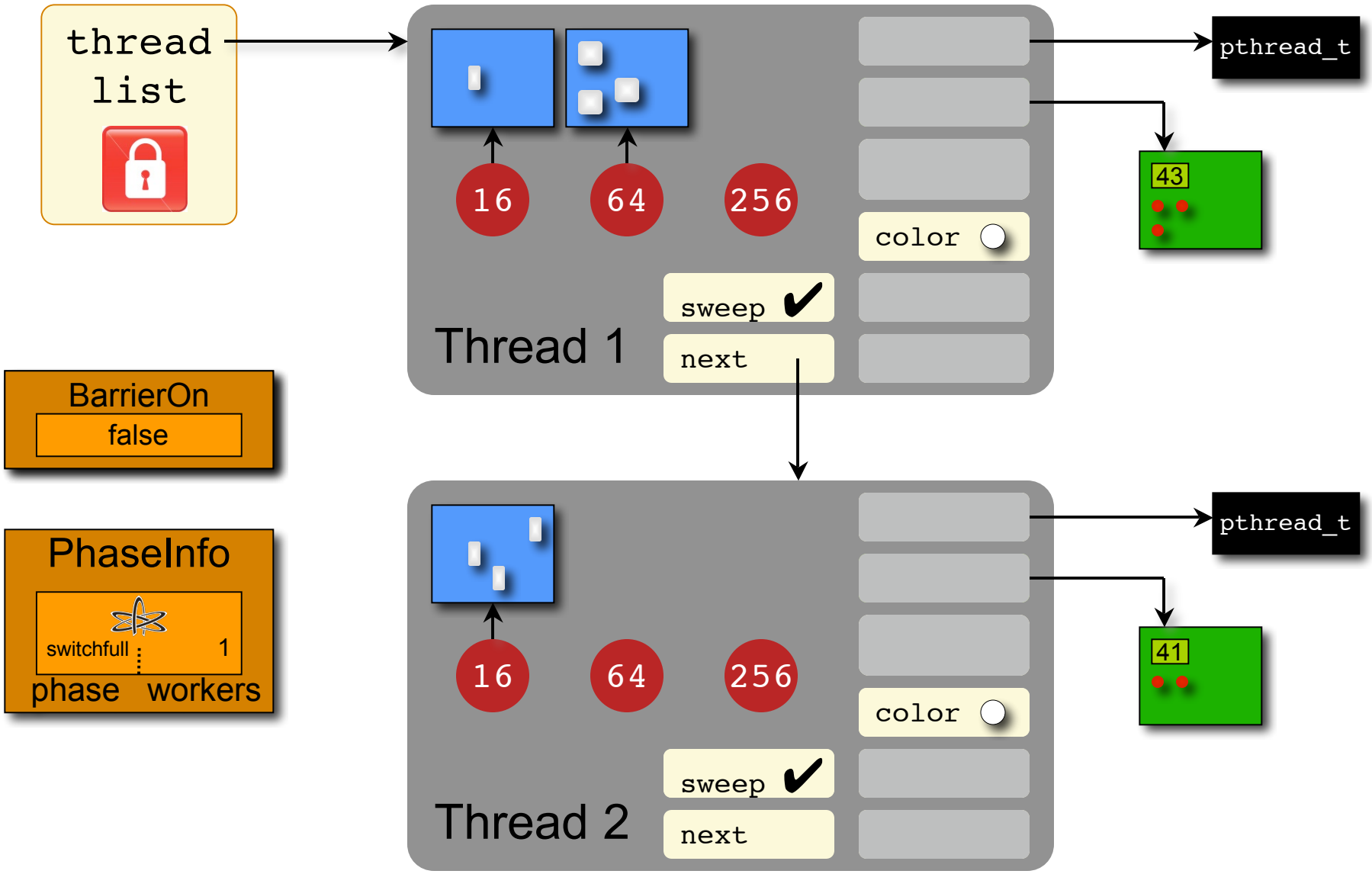
“Sweep” Phase

Sweep Phase


16
64
256
available
temp
full
full
free
16
free
64
free
256
free

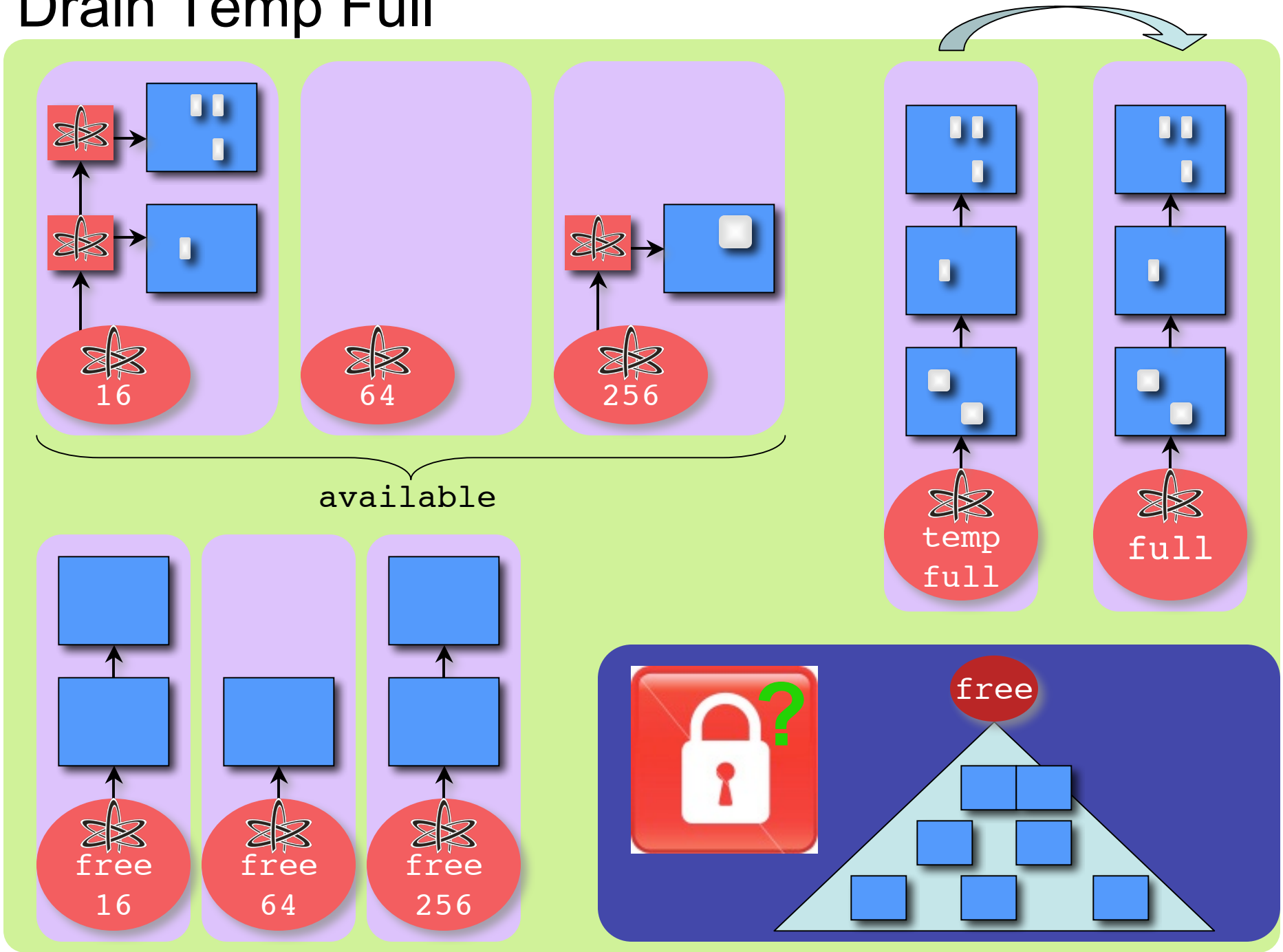
“Switch Full List” Phase

Switch Back to Normal Full List



“Drain Temp Full” Phase

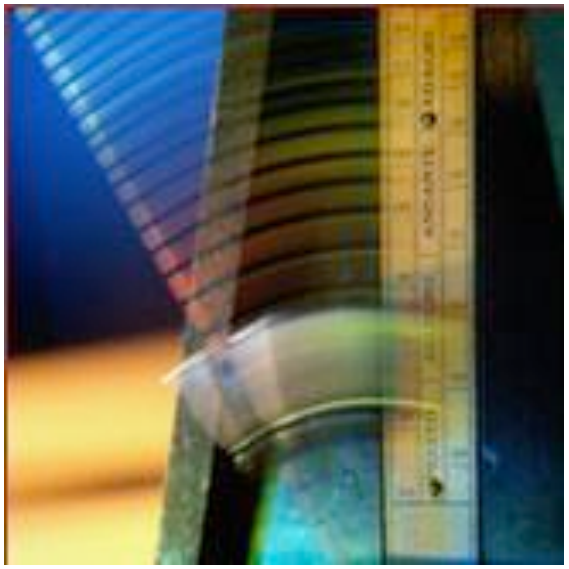
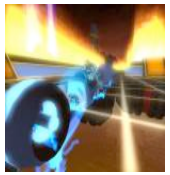
Drain Temp Full

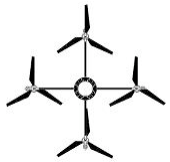


Done with Collection!!

- But…
  - That Was the Easy Part
- And there are still some problems
  - What if threads go out to lunch?





http://www.research.ibm.com/metronome

https://sourceforge.net/projects/tuningforkvp