MULTICORE PROGRAMMING

Epoch-based memory reclamation and experimental methodology

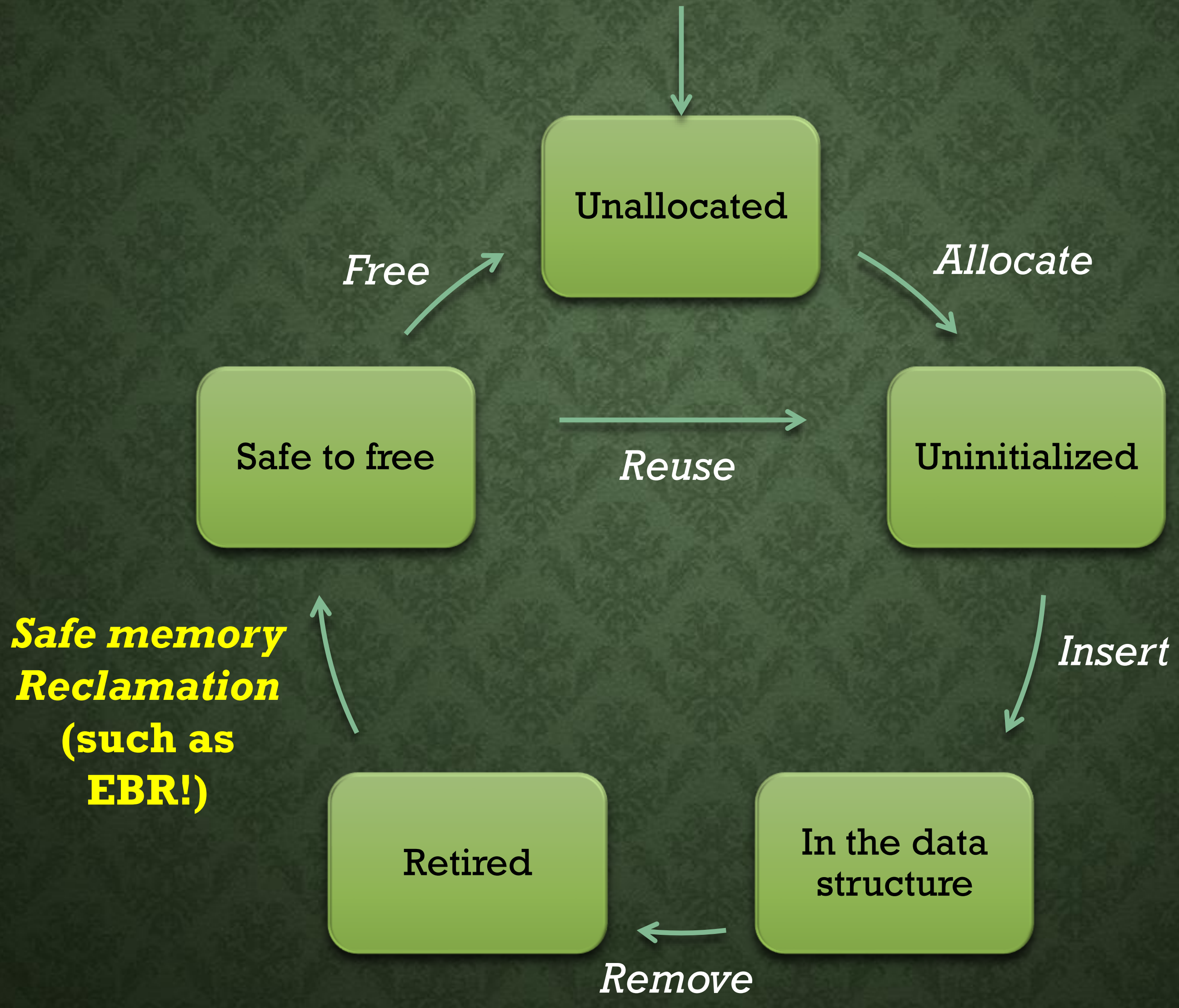
Lecture 17

Trevor Brown

THIS TIME

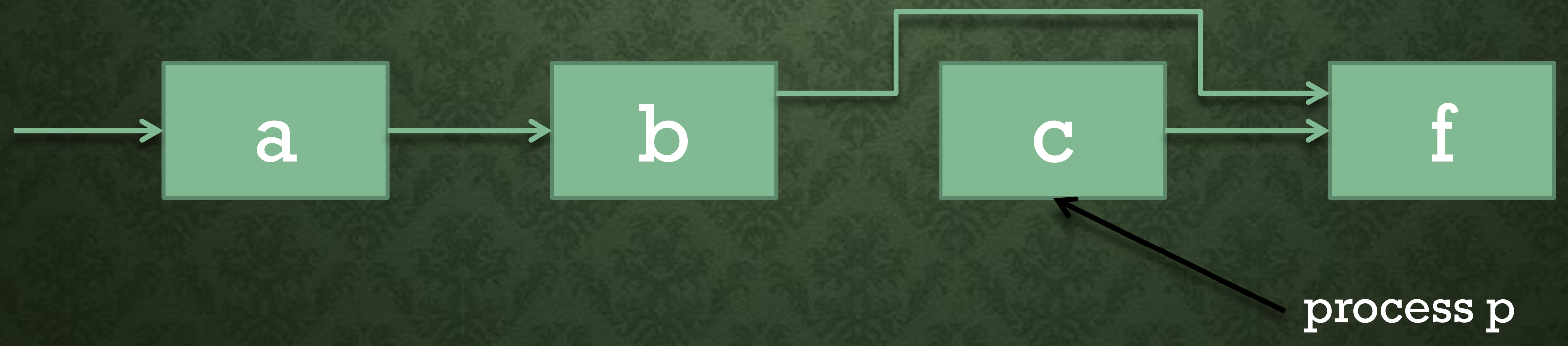
- **Epoch-based memory reclamation**
  - (**The algorithm itself**, as well as a bit more on usage)
- Time permitting:
  - A discussion on experimental methodology

LIFECYCLE OF A RECORD (OBJECT)



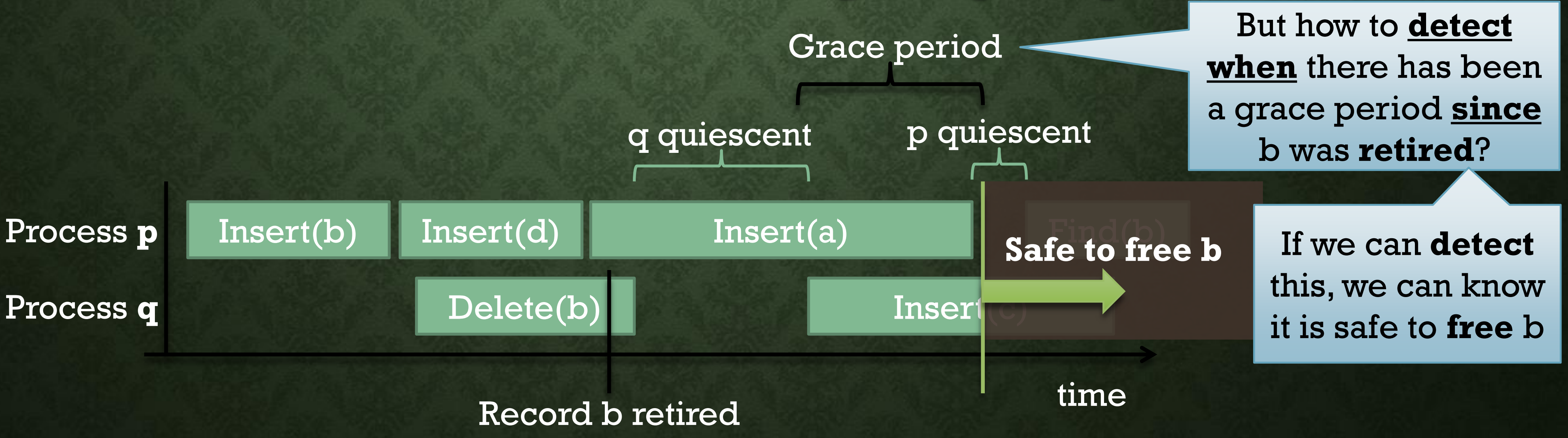
RECLAMATION WITH AND WITHOUT LOCKS

- Easy with locks: correct locking ensures that no process can access an unlinked record
- Hard without locks: processes must carefully coordinate to avoid accessing freed nodes
  - Challenge: any record you are about to free could be pointed to by another process



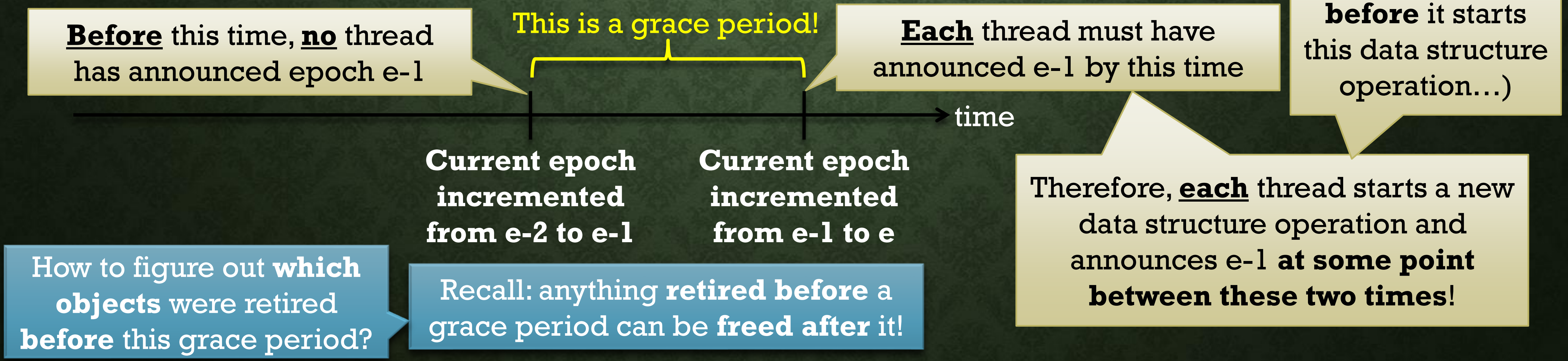
QUIESCENCE AND GRACE PERIODS

- **Definition:** a process is **quiescent** iff
  its private memory does **not** contain any pointers to records in the data structure
- **Definition:** a **grace period** is an interval **during which**
  each thread has **some time** when it is quiescent
  (different threads can be quiescent at different times---that is fine)
- **Fact:** a retired record can be freed after a **subsequent** grace period



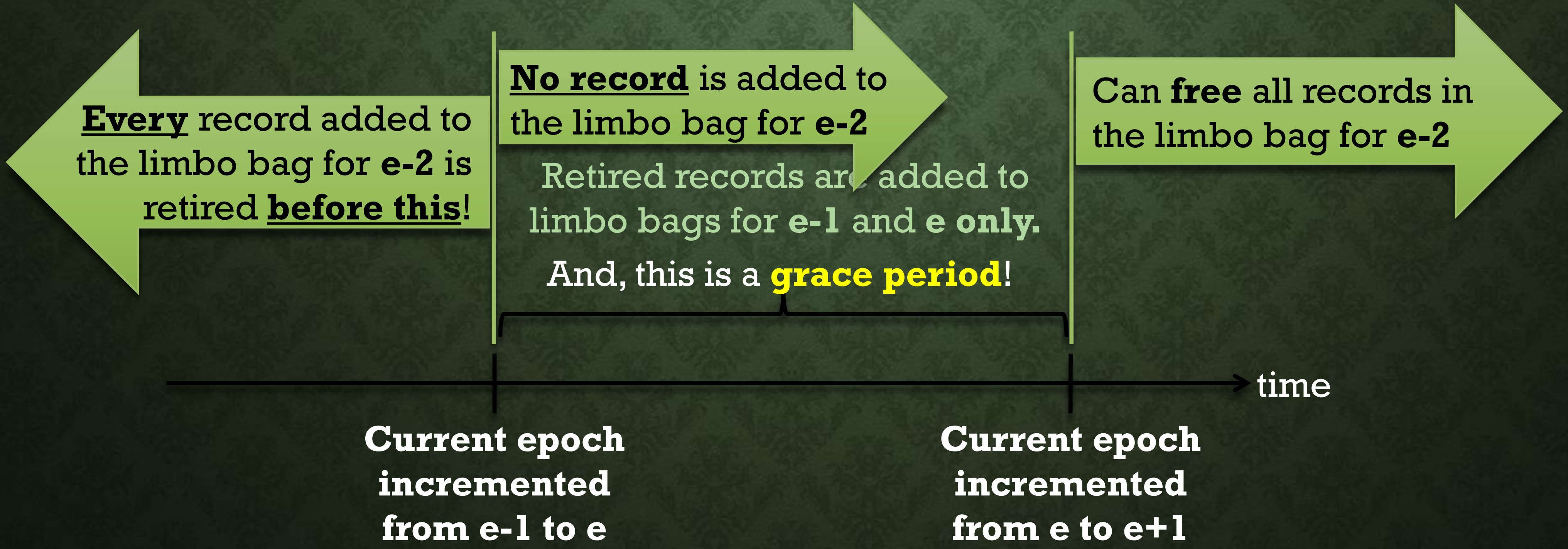
USING **EPOCHS** TO **DETECT** GRACE PERIODS

- **Key assumption:** threads are **quiescent** when **not** executing data structure operations
- The execution is divided into *epochs*,
  and the current epoch number is stored (as a global) in shared memory
- At the start of **each** data structure operation:
  1. read current epoch and **announce** it (in a global array with one slot per thread)
  2. check whether all other threads have announced it
  3. if so, **increment** the current epoch

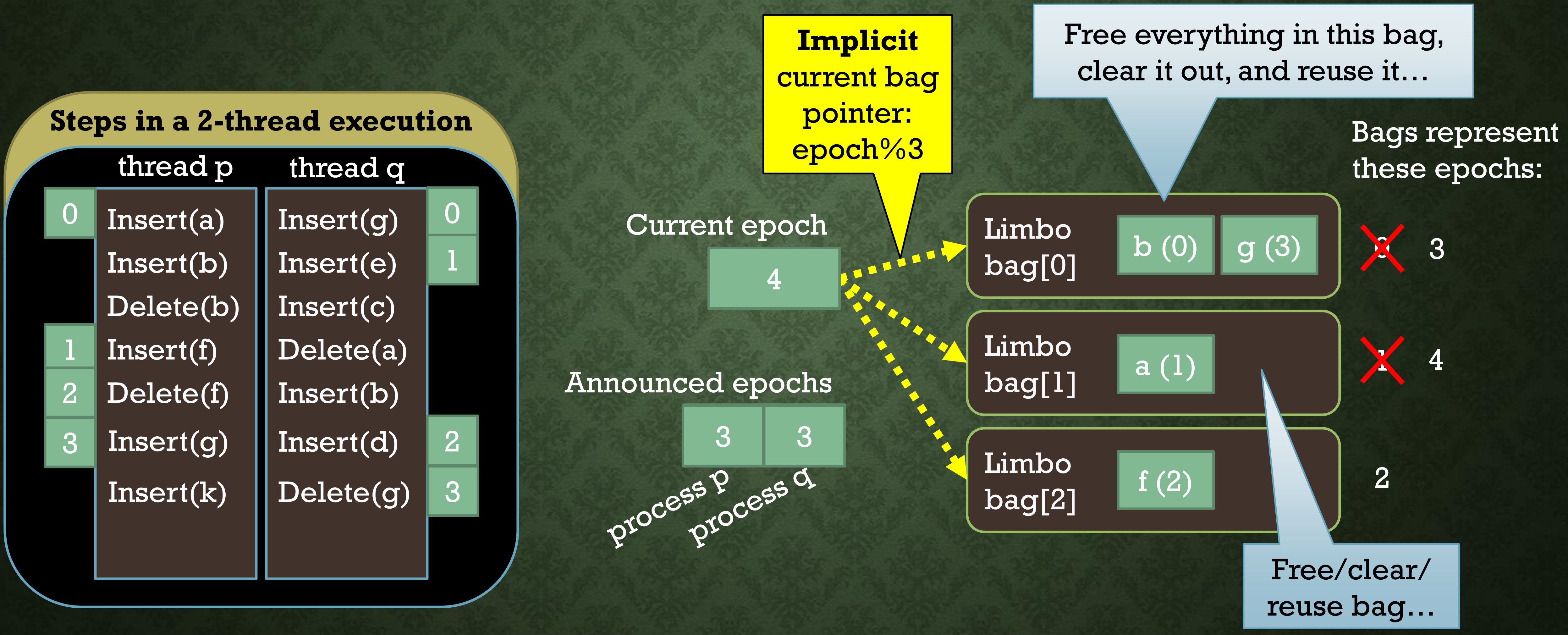


DETERMINING WHICH RECORDS ARE SAFE TO FREE

- Maintain a shared **limbo bag** for **each** of the **last 3 epochs**
- Records retired by thread p are added to the bag for the last epoch **announced** by p
  - So, if p reads epoch 7, then retires a node u, it will place u in the limbo bag for epoch 7
  - Note: by the time p retires u, the current epoch might be 8 (but it **cannot** be 9 or larger… why?)
- When the current epoch changes from e to e+1,
  we free all records in the limbo bag for e-2 (lets see why…)



EXAMPLE EXECUTION OF THE EBR ALGORITHM



IMPLEMENTATION: DATA TYPES

```
template <typename T>
class alignas(64) limbo_bag {
    ... implementation details skipped ...
public:
    void add(T * obj);
    void freeAll();
};

struct alignas(64) padded_aint {
    atomic<int>     v;
}

template <typename T, int NUM_THREADS>
class alignas(64) ebr_manager {
private:
    padded_aint          announce[NUM_THREADS];
    padded_aint          currEpoch;
    limbo_bag<T>         bags[3];
    char                 padding[64];
public:
    ebr_manager();
    void startOp();
    void retire(T * obj);
};

thread_local int tid; // assuming this is set
```

alignas to serve as padding

Sequential implementation

Note: freeAll should cause object **destructors** to be called just before the objects are freed

Type T of objects to manage/reclaim is templated

Threads' epoch announcements represent the epoch they are **conceptually running in**

Limbo bags for the current epoch and the previous two epochs

API: let's see how these are implemented

Another option instead of passing tid to every function

IMPLEMENTATION: EASY PARTS

```
template <typename T, int NUM_THREADS>
ebr_manager<T,NUM_THREADS>::ebr_manager() {
    static_assert(alignof(announce[0]) == 64);
    static_assert(alignof(announce[1]) == 64);
    static_assert(alignof(currEpoch) == 64);
    static_assert(alignof(bags[0]) == 64);
    static_assert(alignof(bags[1]) == 64);
}

template <typename T, int NUM_THREADS>
void ebr_manager<T,NUM_THREADS>::retire(T * obj) {
    assert(tid >= 0 && tid < NUM_THREADS);
    int currBag = announce[tid] % 3;
    bags[currBag].add(obj);
}
```

Paranoia: double check that alignment really implies data is **padded** the way we think it is

Paranoia: check that our tid is a valid one

announce[tid] is the epoch we are conceptually running in, and this modulo 3 is the limbo bag **for that epoch**. We put obj in that bag.

THE HARD PART (NOT OPTIMIZED)

```
template <typename T, int NUM_THREADS>
void ebr_manager::startOp() {
    assert(tid >= 0 && tid < NUM_THREADS);

    // read & announce the epoch that we will "run in"
    int seen = -1; /* -1 means epoch is "locked" */
    while (seen == -1) seen = currEpoch;
    if (seen != announce[tid]) announce[tid] = seen;

    // try to the advance currEpoch
    for (int i=0;i<NUM_THREADS;++i) {
        if (announce[i] != seen) return; // can't advance it
    }
    int exp = seen;
    if (currEpoch.compare_exchange(exp, -1 /* locked */)) {
        // we "win" the right to freeAll in the new bag
        // (while we freeAll, the "lock" prevents add()s)
        int newBag = (seen+1) % 3;
        bags[newBag].freeAll();
        currEpoch = seen+1; // "unlock" with the new epoch
    }
}
```

Paranoia: check that our tid is a valid one

currEpoch can conceptually be "locked" by a thread, who temporarily sets it to -1. So, we repeatedly read until we see an "unlocked" value.

If we see a **new** epoch, we update our announcement

Check if all threads have announced **seen**

If so, we will try to increment currEpoch **and** reclaim memory

This involves "locking" currEpoch, freeing contents of the new current bag, and "unlocking" currEpoch to the new epoch value

OPTIMIZING

```
template <typename T, int NUM_THREADS>
void ebr_manager::startOp() {
    assert(tid >= 0 && tid < NUM_THREADS);

    static thread_local numCalls = 0;
    ++numCalls;

    // read & announce the epoch that we will "run in"
    int seen = -1; /* -1 means epoch is "locked" */
    while (seen == -1) seen = currEpoch;
    if (seen != announce[tid]) announce[tid] = seen;

    // only try to advance the epoch after many operations
    const int THRESHOLD = max(100, 10*NUM_THREADS);
    if ((numCalls % THRESHOLD) == 0) {
        for (int i=0;i<NUM_THREADS;++i) {
            if (announce[i] != seen) return; // can't advance it
        }
        int exp = seen;
        if (currEpoch.compare_exchange(exp, -1 /* locked */)) {
            // we "win" the right to freeAll in the new bag
            // (while we freeAll, the "lock" prevents add()s)
            int newBag = (seen+1) % 3;
            bags[newBag].freeAll();
            currEpoch = seen+1; // "unlock" with the new epoch
        }
    }
}
```

Each thread keeps track of how many calls it has ever made to startOp()

static: accessible only within the function, but preserves its value across function calls ("= 0" happens **once**)

We try to advance the epoch only after performing THRESHOLD operations

Tradeoff: it takes longer to reclaim garbage, but we amortize the overhead of scanning announcements over many operations

Caveat: I have not run this code, and have not spent time tuning THRESHOLD

ADDING AN RAII GETGUARD() OPERATION

```
template <typename T, int NUM_THREADS>
class alignas(64) ebr_manager {
private:
    padded_aint           announce[NUM_THREADS];
    padded_aint           currEpoch;
    limbo_bag<T>          bags[3];
    char                  padding[64];
public:
    ebr_manager();
    void startOp();
    void retire(T * obj);

    class mem_guard {
        ebr_manager<T,NUM_THREADS> * mgr;
    public:
        mem_guard(ebr_manager<T,NUM_THREADS> * _mgr) : mgr(_mgr) {
            mgr->startOp();
        }
        ~mem_guard() {
            // if a "mgr->endOp()" were defined, we'd call it here
        }
    };
    mem_guard getGuard() {
        return mem_guard(this);
    }
};
```

Note: getGuard() has somewhat **limited usefulness** when there is no endOp() operation…

Just showing you how to do this RAII design pattern…

USAGE EXAMPLE 1: RECALL THE TREIBER STACK

```cpp
class stack {
    class node {
        int            key;
        node *         next;
    };
    static const int    EMPTY = -1;
    char                padding1[64];
    atomic<node *>      top;
    char                padding2[64];
public:
    stack() : top(NULL) {}
    void push(int key);
    int pop();
};
```

```cpp
void stack::push(int key) {
    node * n = new node();
    n->key = key;
    while (true) {
        node * curr = top;
        n->next = curr;
        node * exp = curr;
        if (top.compare_exchange(exp, n))
            return;
    }
}

void stack::pop() {
    while (true) {
        node * curr = top;
        if (curr == NULL) return EMPTY;
        node * next = curr->next;
        node * exp = curr;
        if (top.compare_exchange(exp, next))
            return curr->key;
    }
}
```

TREIBER STACK WITH MEMORY RECLAMATION

```
class stack {
    class node {
        int             key;
        node *          next;
    };
    static const int    EMPTY = -1;
    char                padding1[64];
    atomic<node *>      top;
    char                padding2[64];
    ebr_manager<node>   mgr;
    char                padding3[64];
public:
    stack() : top(NULL) {}
    void push(int key);
    int pop();
};
```

```
void stack::push(int key) {
    node * n = new node();
    n->key = key;
    while (true) {
        auto guard = mgr.getGuard();
        node * curr = top;
        n->next = curr;
        node * exp = curr;
        if (top.compare_exchange(exp, n))
            return;
    }
}

void stack::pop() {
    while (true) {
        auto guard = mgr.getGuard();
        node * curr = top;
        if (curr == NULL) return EMPTY;
        node * next = curr->next;
        node * exp = curr;
        if (top.compare_exchange(exp, next))
            mgr.retire(curr);
            return curr->key;
    }
}
```

EXAMPLE 2: EXPANDABLE HASH TABLE

```
void hashmap::startExpansion(t)
  if (currentTable == t) {
    t_new = createNewTableStruct(t);
    if not CAS(&currentTable, t, t_new) delete t_new;
  }
  helpExpansion(currentTable);
```

```
struct hashmap
  char padding1[64];
  atomic<table *> currentTable;
  char padding2[64];
```

Want to reclaim object types:
hashmap, table, counter, atomic<int> array

```
struct table
  char padding1[64];
  atomic<int> * data;
  atomic<int> * old;
  int capacity;
  int oldCapacity;
  counter * approxIns;
  counter * approxDel;
  char padding2[64];
  atomic<int> chunksClaimed;
  char padding3[64];
  atomic<int> chunksDone;
  char padding4[64];
```

Suppose we allocate new counter objects for each table object

EXPANDABLE HASH TABLE WITH MEMORY RECLAMATION

```
void hashmap::startExpansion(t)
  if (currentTable == t) {
    t_new = createNewTableStruct(t);
    if not CAS(&currentTable, t, t_new) delete t_new;
  }
  helpExpansion(currentTable);
```

If CAS **succeeds**, then **we unlinked t**, so we have the right to call mgr->retire(t)

```
struct hashmap
  char padding1[64];
  atomic<table *> currentTable;
  char padding2[64];
  ebr_manager<table> mgr;
  char padding3[64];
```

```
struct table
  char padding1[64];
  atomic<int> * data;
  atomic<int> * old;
  int capacity;
  int oldCapacity;
  counter * approxIns;
  counter * approxDel;
  char padding2[64];
  atomic<int> chunksClaimed;
  char padding3[64];
  atomic<int> chunksDone;
  char padding4[64];
```

Observation: if a table object **t** is safe to free, then no thread will ever access **t->old** or **t->approxIns** or **t->approxDel**

So those objects can be directly passed to free() whenever a table object is freed (carefully do this in ~table(), which is invoked by EBR)

OPTIMIZING INTO A “REAL” ALGORITHM (ONE WORTH IMPLEMENTING)

Distributed

Epoch

Based

Reclamation

Algorithm

Refers not to distributed systems, but to distributing the limbo bags across threads…

Note: the memory reclamation algorithm you were given in A5 is based on DEBRA…

SIGNIFICANT CHANGE FROM EBR

- Per-thread **quiescent bit** to allow reclamation to **continue** while a process is quiescent
  - Useful if some threads finish their work and stop, or work on something else
  - **Partial** fault tolerance
    - Crashing **while quiescent** does not block reclamation

EASY CHANGE: SCANNING EPOCH ANNOUNCEMENTS

- Amortize cost over several operations
- Each operation checks **one** epoch announcement
  (or you could check **one** announcement per **K** operations)
- After checking **n** announcements, where n is the number of threads,
  and seeing the announcements are up to date, the epoch can be advanced

EFFICIENT BAGS

- Per-process limbo bags
  - Each process rotates its limbo bags whenever its announcement changes
- Per-process free bags and one shared free bag
  - When rotating its limbo bags,
    a process appends its oldest limbo bag to its own free bag
  - Entire blocks moved to/from shared free bag
- More details (and optimizations) in the paper
  Brown, T. Reclaiming memory for lock-free data structures. PODC 2015.
  - Conference paper
  - Extended paper
  - Slides for that talk

COMPLEXITY OF DEBRA

- leaveQstate: O(1) steps ← called at start of operation
- enterQstate: O(1) steps ← called at end of operation
- retire: O(1) steps ← called after **unlinking** a record
- Reclamation operations are **wait-free!**

- However, this does **not** mean reclamation is fault tolerant!
  - A thread that crashes while non-quiescent can still block reclamation!
  - This is addressed in the **DEBRA+** algorithm (same paper as DEBRA)
  - … and more recently with Ajay Singh and Ali Mashtizadeh in **Neutralization Based Reclamation, PPoPP 2021. [paper] [talk]**