MULTICORE PROGRAMMING

Implementing multi-word CAS (KCAS)

Lecture 11

Trevor Brown

LAST TIME

- DCAS
  - Surprisingly not enough to implement doubly-linked list (at least not easily)
- KCAS
- Doubly-linked list using KCAS
  - Linearizability sketch
  - Searches are the hard part

THIS TIME

- Implementing KCAS
  - Built from CAS and double-compare-**single-swap** (DCSS)
  - Show how to implement DCSS first
- Most complex lock-free algorithm we will see
  - Uses lock-free **helping** to guarantee progress

LOCK-FREE HELPING

- Suppose
  - p starts an operation O
  - q is blocked by O
- Lock-based approach
  - q waits for p
- Lock-free approach
  - q performs O on behalf of p
  - **How does it know how to perform O?**

DESCRIPTORS

- Each operation O creates a **descriptor d**
- Descriptor **d** encodes how to perform O
  - usually contains arguments to O
  - sometimes some status information
- Help by invoking a function Help(d)
  - Completes the operation that created d

DOUBLE COMPARE SINGLE SWAP (DCSS) [HARRIS2002]

- Semantics:

```
DCSS(addr1, addr2, exp1, exp2, new2)
  atomic {
    val1 = *addr1;
    val2 = *addr2;
    if (val1 == exp1 && val2 == exp2) *addr2 = new2;
    return val2;
  }
```

Not to be confused with DCAS

```
DCSSRead(addr)
  return the value last stored in *addr by a DCSS
```

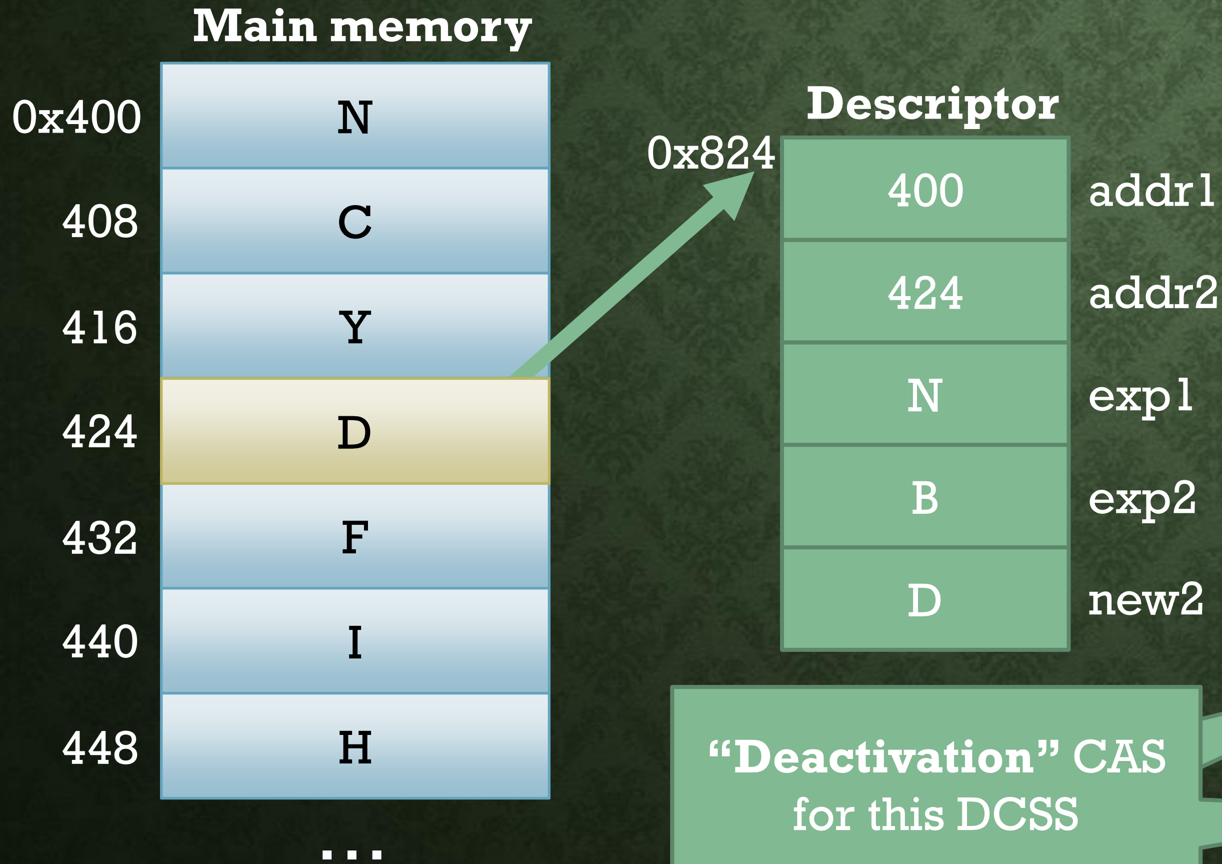
- Usage constraints:
  - **addr2** **<u>must only</u>** be modified by DCSS
  - **addr2** **<u>must only</u>** be read with DCSSRead
  - **addr1** **<u>can never</u>** be modified by DCSS

Note: no such restriction for DCAS or KCAS… just DCSS

USING CAS TO BUILD DCSS

IMPLEMENTATION SKETCH: DCSS(400, 424, N, B, D)

“**Activation**” CAS for this DCSS



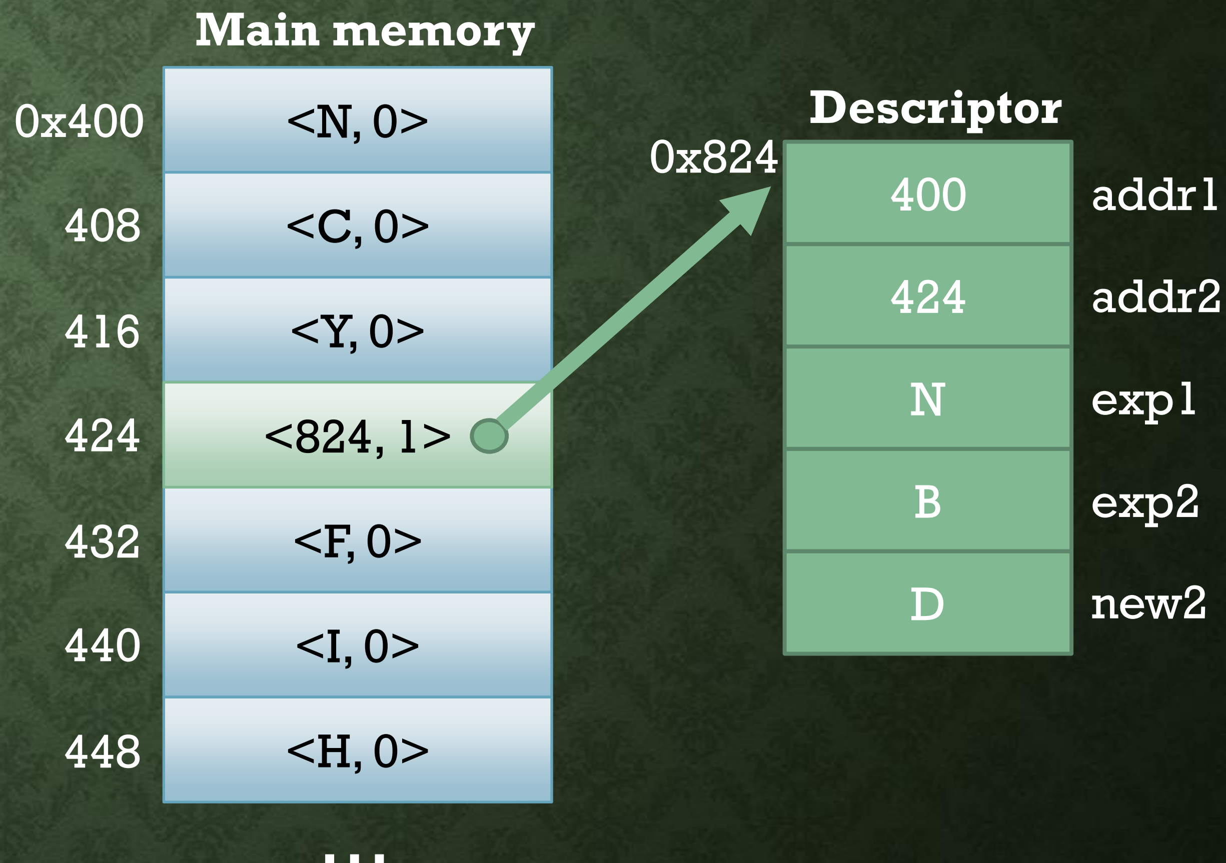

```
let d = pointer to new descriptor
CAS(addr2, exp2, d)
Help(d)
```

Help(d) function:

```
if *d.addr1 = d.exp1
    CAS(d.addr2, d, d.new2)
else
    CAS(d.addr2, d, d.exp2)
```

HOW TO TELL IF AN ADDRESS POINTS TO A DESCRIPTOR?

- Steal the least significant bit (LSB) from each field that can be modified by DCSS
- Use it to indicate addr points to a descriptor
  - if (*addr & 1) then it's a descriptor …
- What if application values USE the LSB?
  - Can **shift** values left (then can't use MSB)
  - No need to shift word-aligned pointers!
- Packing/unpacking a descriptor pointer **d**
  - pack(d): return d | 1
    [making it "look like" a descriptor pointer]
  - unpack(d): return d & ~1
    [so we can dereference it]



IMPLEMENTATION: DATA TYPES

```
struct DCSS_desc {
    atomic<word_t> * addr1;
    atomic<word_t> * addr2;
    word_t           exp1;
    word_t           exp2;
    word_t           new2;
    char             padding[24];
} __attribute__ ((aligned(64)));
```

VALUE CAS (VAL_CAS)

- Slightly different definition of CAS(addr, exp, new)
- Instead of returning true/false,
  it returns the **value that was contained in *addr**
  when the CAS occurred
  - For successful CAS, this is **exp**
  - For failed CAS, this is **different** from exp
    - → the value that caused the CAS to fail!

Note: in GCC, this is
__sync_**val**_compare_and_swap.
C++ <atomic> also implements
value CAS semantics.
Java has no equivalent!

IMPLEMENTATION: DCSSREAD

```
word_t DCSSRead(atomic<word_t> * addr)
17  word_t v;
18  while (true) {
19    v = *addr;
20    if (isDCSS(v)) DCSSHelp(unpack(v));
21    else break;
22  }
23  return v;
```

Try to read *addr.
If we do **not** see a descriptor pointer,
we are done.

If we read a descriptor pointer,
we **help** that DCSS and then retry

This continues until we see an
application value (not a descriptor)

We linearize DCSSRead at its **last** read of *addr
(where it sees an **application value**)

IMPLEMENTATION: DCSS

```
word_t DCSS(addr1, addr2, exp1, exp2, new2)
1    DCSS_desc * d = new DCSS_desc(addr1, ...);
2    word_t val2;
3    while (true) {
4      val2 = VAL_CAS(d->addr2, d->exp2, pack(d));
5      if (isDCSS(val2)) DCSSHelp(unpack(val2));
6      else break;
7    }
8    if (val2 == d->exp2) {
9      DCSSHelp(d); // finish our operation
10   }
11   return val2;
```

Create DCSS operation descriptor

Activation CAS: try to CAS our **descriptor pointer** into **addr2** (temporarily replacing **exp2**)

If the value returned from the CAS is a descriptor pointer (which means our CAS **failed**), we **help** the other DCSS, then we retry

Retry until we see an application value (not a descriptor pointer)

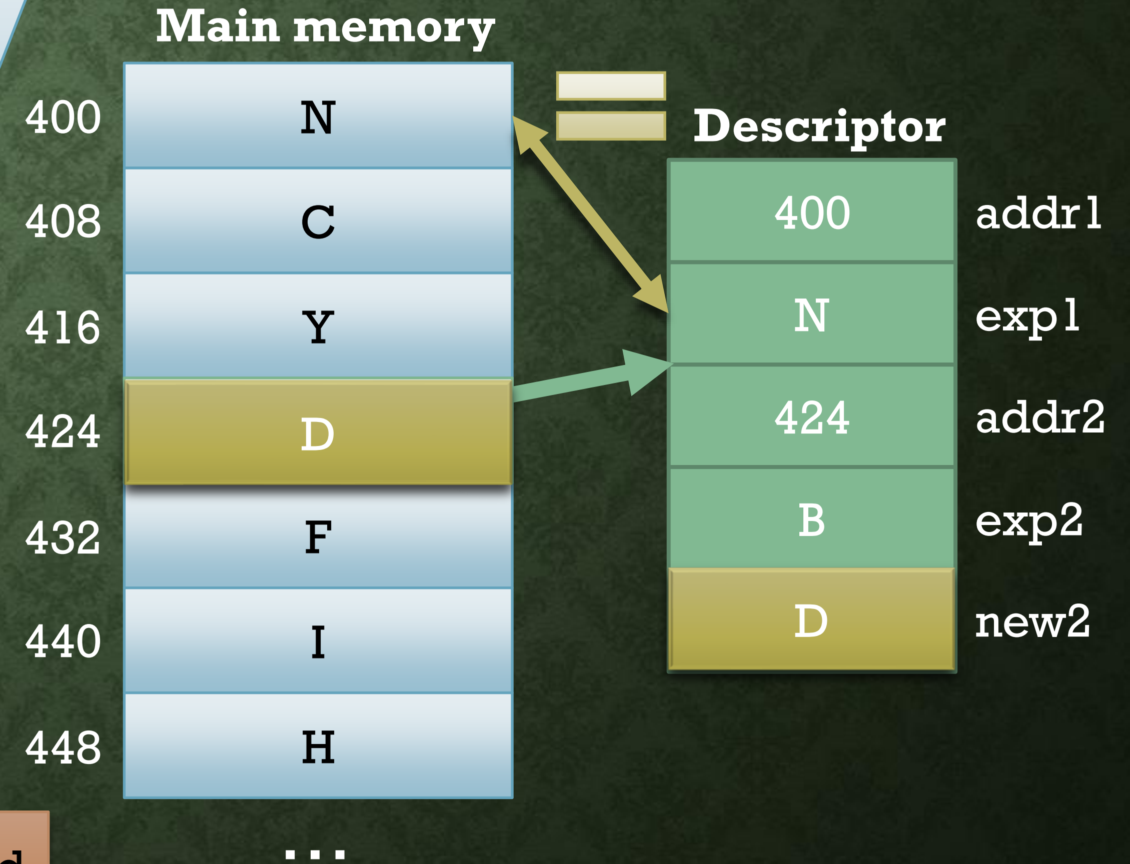
If our activation CAS succeeds, we **help** our own DCSS complete

Not handled here: how to free d

THE HELP FUNCTION: DCSS SUCCEEDS

```
void DCSSHelp(DCSS_desc * d)
12   if (*d->addr1 == d->exp1) {
13     CAS(d->addr2, pack(d), d->new2);
14   } else {
15     CAS(d->addr2, pack(d), d->exp2);
16   }
```

Deactivation CAS



Where should this DCSS be linearized?

The read at line 12… **by the thread** that does the successful CAS at line 13!

At the **read**??? You might have suggested the successful CAS… but no!

A: Activation CAS on d->addr2

B: Read *d->addr1

C: Deactivation CAS on d->addr2

Thread p

DCSS -- successful

Read *d->addr2 and see an **application value** (LP)

time

Thread q

DCSSRead

DCSSRead

```
void DCSSHelp(DCSS_desc * d)
12   if (*d->addr1 == d->exp1) {
13     CAS(d->addr2, pack(d), d->new2);
14   } else {
15     CAS(d->addr2, pack(d), 
16   }
```

This operation might be affected by the choice of LP

Makes no difference to this operation if DCSS LP is at B or C

But this operation must **help** the DCSS, so it must do its last read of *d->addr2 **after C**!

It sees the **new** value. So, it makes no difference **to <u>this</u> operation** whether DCSS LP is at B or C. Either way, the DCSS has happened and we see it.

But **B** needs to be the DCSS LP, so we know *d->addr1 == d->exp1 at the LP (required by the ADT for DCSS success)

EXPLAINING IN FURTHER DETAIL: LINEARIZING A SUCCESSFUL DCSS THAT CHANGES ADDR2 FROM D TO NEW2

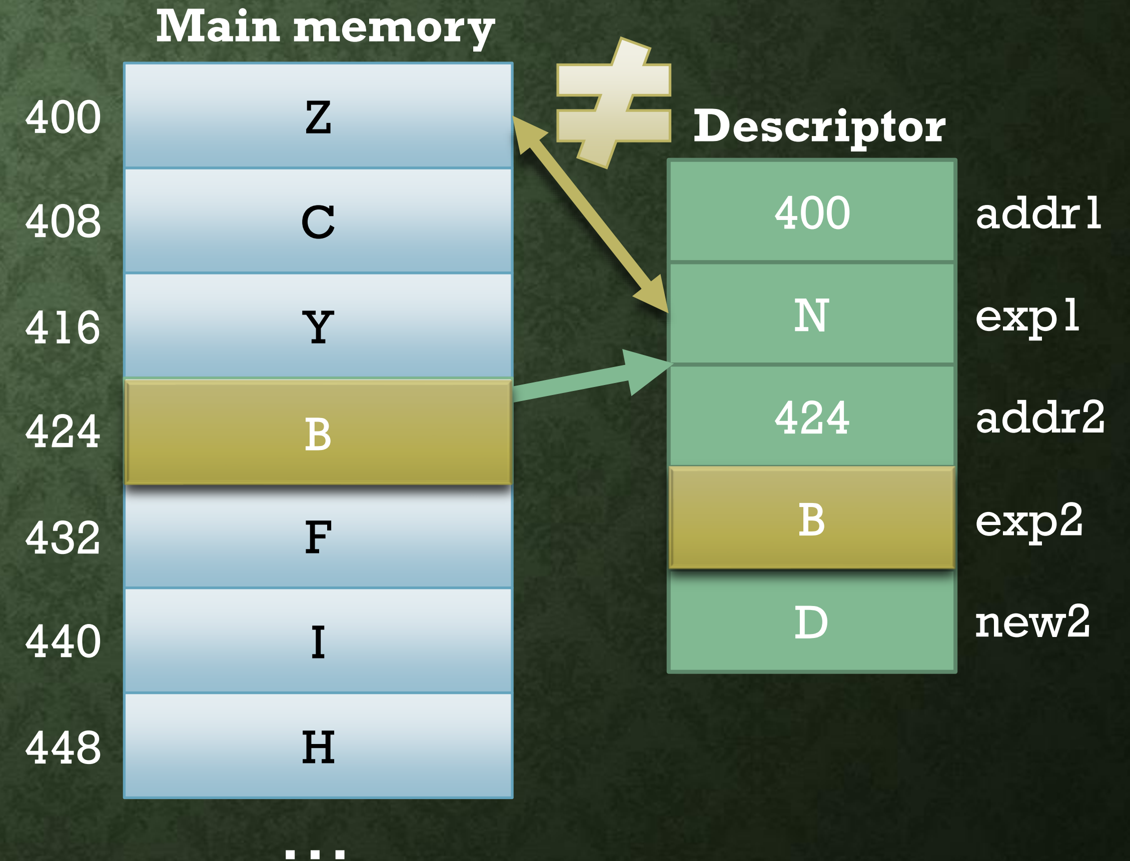
- Consider a successful DCSS operation O with descriptor d which performs a **deactivation CAS** that changes addr2 to **new2**
- Want to linearize when addr1 == exp1 and addr2 == exp2. Argue this time exists…
- There is **exactly one** successful deactivation CAS for O
- Let p be the thread that performs this successful deactivation CAS for O
- Before this CAS, p does at least one read of addr1, and the last such read sees **exp1**
- Before that read, there is a successful activation CAS for O by some thread
- At all times between the successful activation and deactivation CASs for O, addr2 points to **d** (which semantically means addr2 == exp2)
- In particular, when p last reads addr1, we have addr1 == exp1 and addr2 == exp2

THE HELP FUNCTION: DCSS FAILS

```
void DCSSHelp(DCSS_desc * d)
12  if (*d->addr1 == d->exp1) {
13    CAS(d->addr2, pack(d), d->new2);
14  } else {
15    CAS(d->addr2, pack(d), d->exp2);
16  }
```

Where should this DCSS be linearized?

The read at line 12… **by the thread** that does the successful CAS at line 15!



LINEARIZING A FAILED DCSS THAT CHANGES ADDR2 FROM D BACK TO EXP2

- Consider a DCSS operation O with descriptor d which performs a **deactivation CAS** that changes addr2 to **exp2**
- Want to linearize when addr1 != exp1 or addr2 != exp2. Argue this time exists…
- Let p be the thread that performs the deactivation CAS for O
- Before this CAS, p reads addr1 and sees a value **different** from exp1
- Linearize then

```
void DCSSHelp(DCSS_desc * d)
12   if (*d->addr1 == d->exp1) {
13     CAS(d->addr2, pack(d), d->new2);
14   } else {
15     CAS(d->addr2, pack(d), d->exp2);
16   }
```

WHAT ABOUT DCSS OPERATIONS WITH NO SUCCESSFUL ACTIVATION CAS?

```
word_t DCSS(addr1, addr2, exp1, exp2, new2)
1    DCSS_desc * d = new DCSS_desc(addr1, ...);
2    word_t val2;
3    while (true) {
4      val2 = VAL_CAS(d->addr2, d->exp2, pack(d));
5      if (isDCSS(val2)) DCSSHelp(unpack(val2));
6      else break;
7    }
8    if (val2 == d->exp2) {
9      DCSSHelp(d); // finish our operation
10   }
11   return val2;
```

Suppose VAL_CAS fails and returns an **application** value **different** from exp2

We break out of the loop

We skip over the next if-block, and return val2

Where should this DCSS be linearized?

At the (last) failed VAL_CAS by this thread (when we read the **application value** that causes the failure)

WHAT ABOUT A DCSS BY A THREAD THAT CRASHES BEFORE RETURNING?

- Let O be such a DCSS operation
- The return value of O is not a concern… (it doesn't exist)
- But O could still affect the return values of **other operations**!
  - Only if some thread performs a successful deactivation CAS for O that changes addr2 from **d** to **new2**
    - In this case we linearize O the same way as a successful DCSS!
  - Otherwise
    - No need to linearize the operation at all…
    - To all other threads, it's as if O didn't happen!

USING DCSS TO BUILD KCAS

BUILDING KCAS FROM DCSS [HARRIS2002]

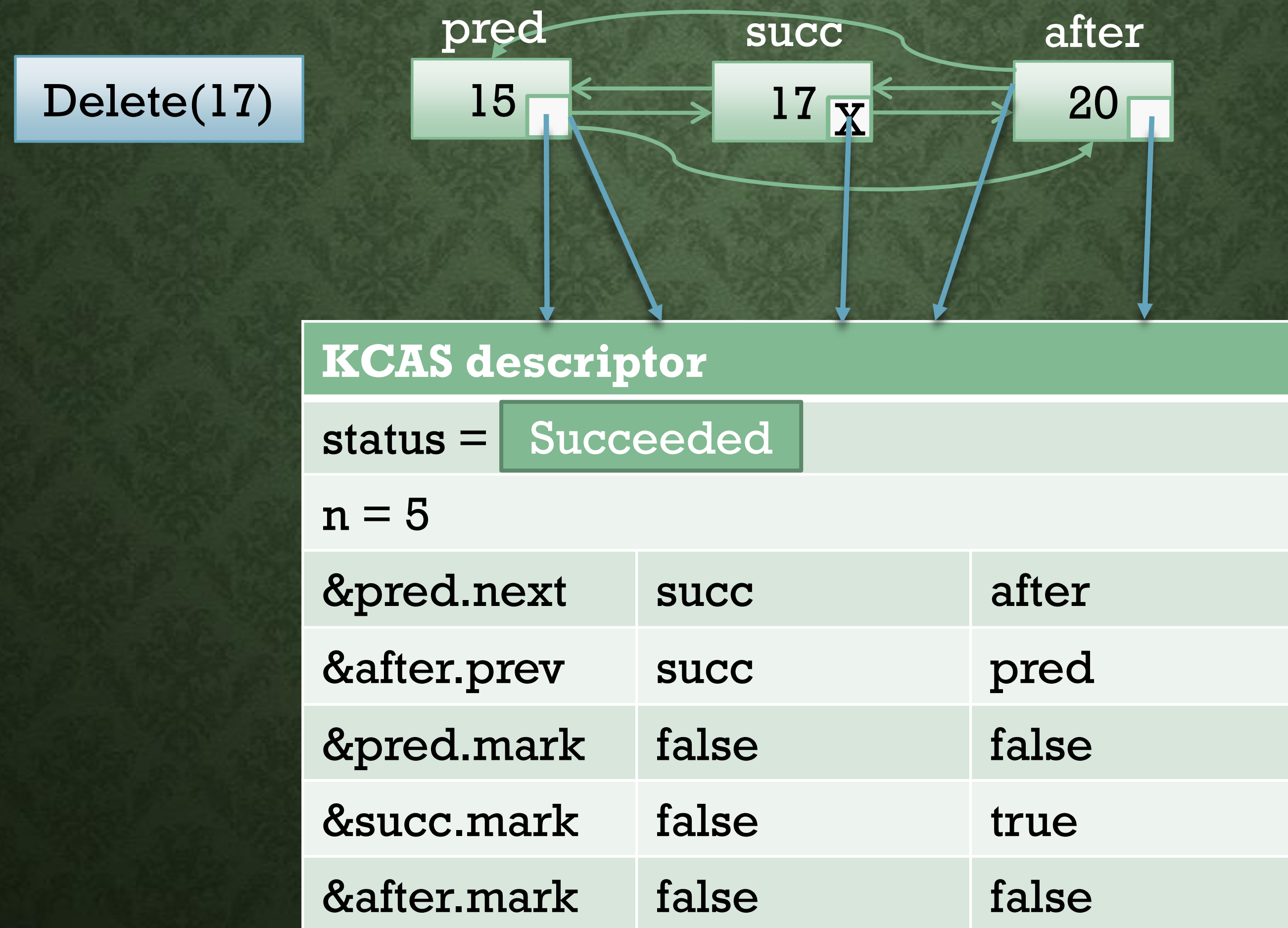
- Facilitate **helping** with **KCAS descriptor**, which stores **n rows** containing: addr, exp, new
- KCAS descriptor also contains a **status** field, with a value in **{Undecided, Succeeded, Failed}**
- The status field helps coordinate threads
- Prevents scenarios where different threads helping a KCAS have different views of memory, and one thinks the KCAS is finished, while another thinks it is still ongoing (and incorrectly makes changes twice, etc.)

KCAS descriptor		
status		
n		
addr1	exp1	new1
addr2	exp2	new2
…	…	…

KCAS ALGORITHM IDEA

- Proceeds in two phases
- Phase 1: lock-free "locking"
  - Iterate over the addresses, attempting to change each address from its expected value to a pointer **d** to the KCAS descriptor
  - If we see an unexpected value, then **status** changes to **Failed**, otherwise it changes to **Succeeded**
- Phase 2: completion
  - Iterate over the addresses, attempting to change each address from **d** to either its **new value**, or its **expected value**, respectively, depending on whether status is **Succeeded** or **Failed**

INTUITION: HOW A SUCCESSFUL KCAS WORKS: DOUBLY-LINKED LIST AS AN EXAMPLE



KCAS descriptor		
status = Succeeded		
n = 5		
&pred.next	succ	after
&after.prev	succ	pred
&pred.mark	false	false
&succ.mark	false	true
&after.mark	false	false