MULTICORE PROGRAMMING

Proving Linearizability

Lecture 4

Trevor Brown

ANNOUNCEMENTS

- A2 should be released shortly, with a fairly short deadline
  - If a non-trivial fraction of the class needs more time, we can extend this
  - (But, we can only do this a couple of times, and A4 will likely need this)

LAST TIME

- We described a stack
- We argued that it offers lock-free progress
- We did not prove linearizability…
  - Let's do that now!
  - This is the main proof in this course   (AC/HS crosslisting)
  - Hopefully explanation is helpful for A2!

TYPICAL STRATEGY FOR LIN. PROOFS

1. Choose linearization points (LPs) for all operations
   - A linearization point is usually a **read, write or CAS (a step)**
   - Note: can pick **different** linearization points for,
     e.g., **different push()es**, or different **pop()s**
2. Prove: for each concurrent execution E,
   the **chosen** LPs induce an **equivalent linearized** (sequential) execution L
   - Recall: **equivalent** means
     all operations in L return the same values as their corresponding operations in E

OTHER (IMPORTANT) PROOF POSSIBILITIES

In some algorithms, it is **not possible** to choose an **explicit step** where an operation should be linearized

- In such algorithms, we prove:
  - for each concurrent execution E, **there exist** LPs that induce an **equivalent linearized** execution L
  - These LPs are usually **configurations** (i.e., *points in time*) rather than steps

- In practice, even if we cannot explicitly choose a **line of code (step)** as an LP, we can often argue **there exists some time during O** when the value O returned would have been the correct thing to return
  - (And we can thus linearize at that time)
- Advice: try explicit LPs first, and fall back to this if you can't find LPs that "work out"

CHOOSING LINEARIZATION POINTS FOR OUR STACK

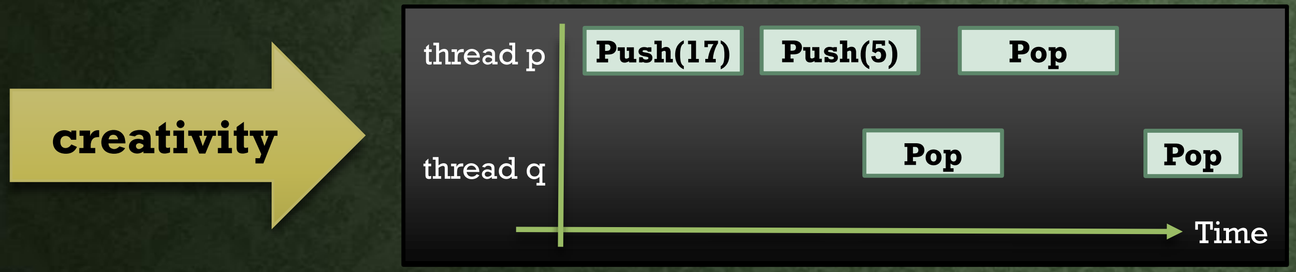
- Creativity is needed! Iterative process.
- My own thought process…
- Consider operations that **update** the stack
  - A push, or a pop that does **not** return EMPTY
  - **Heuristic question** for **updates:** can you identify a step that makes it possible for other threads to "**see**" **the effects** of this operation?
  - **Hypothesize a fixed linearization point *LP***
- Consider operations that **query** the stack (A pop that returns EMPTY)
  - **Heuristic question** for **queries:** can you identify a critical step at which the pop becomes aware of another operation's effects?
  - Either (a) **hypothesize a fixed *LP***
    or (b) argue that one must exist

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

```
int stack::pop()
  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      return curr->key;
    }
  }
```

SANITY CHECKING YOUR HYPOTHESIZED LPs

- Brainstorm a 2-thread concurrent execution E
  (you make it up, goal is to create opportunities to expose bugs)



Interactions between more than two threads / operations are also interesting. But this is a good start.

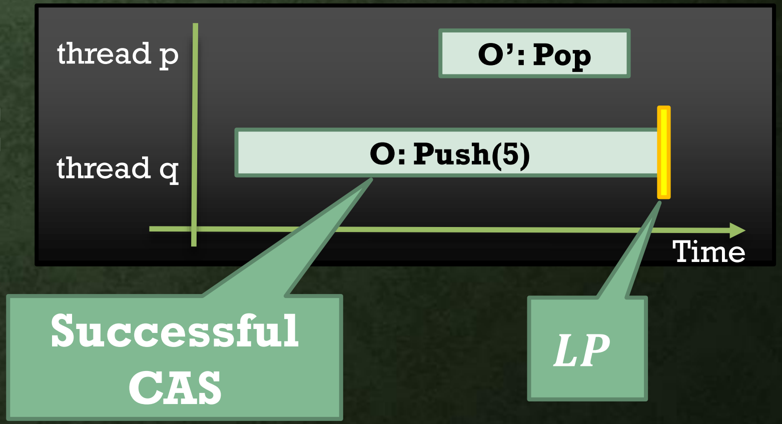
- Pick two concurrent operations O and O', and
  consider different possible thread schedules/interleavings of their steps
  - Can you find an execution wherein either operation **returns an incorrect value**?
  - (If there is any execution where it returns an incorrect value, then **chosen LPs are wrong!**)

- Let's try to see how sanity checks can catch an **incorrect LP**: the **return statement** in **push**
- Creativity: brainstorm a sample execution E, with concurrent operations O and O' (one of which should be a push, to test our LP)
  - It helps to pay special attention to LPs, and steps that change the return values of other operations.
- What value **should** Pop() return in this execution, to be consistent with the stack ADT?
- Creativity: consider different thread schedules…
  - Eventually, we might try scheduling the successful CAS of O before the start of O'
- **What will O' actually return? (See code)**
- O' will return 5 instead of EMPTY.
- This disagrees with the ADT! **So LP is wrong!**

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

```
int stack::pop()
  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      return curr->key;
  } }
```

E

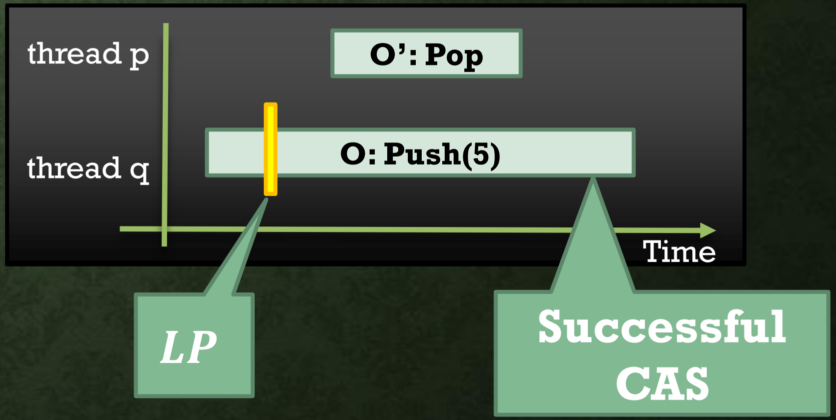


- Let's see another example of an **incorrect LP**: the **last read of top** in **push**
- **Heuristic question:** can anyone else *see* our push yet?
- Obtain sample execution via *creativity*
- What value **should** Pop() return in this execution, to be consistent with the stack ADT?
  - O' must be linearized between its own start and end, so it must be **after O**. So, it **must** return 5!
- After some *creativity* we try scheduling the LP of O before O' and the successful CAS of O after O'
- **What will O' actually return?**
- O' will return EMPTY instead of 5.
- This disagrees with the ADT! **So LP is wrong!**

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

```
int stack::pop()
  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      return curr->key;
  } }
```

E



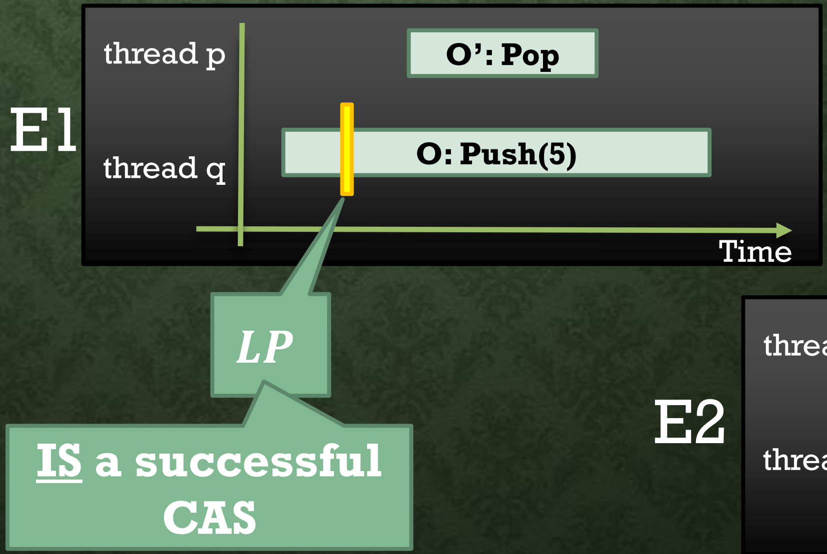
Correct LP: a **successful CAS** in **push**

- **Heuristic questions:**
  - **can anyone else <u>see</u> our push <u>before</u> the CAS?**
  - **how about <u>after</u>?**
- Creativity: try out various executions…
- **Execution E1:**
  - What value <u>**should**</u> Pop() return in this execution, to be consistent with the stack ADT?
  - **What will O' actually return?**
  - O' will return 5, as it should.
- **Execution E2:**
  - **What should/will O' return?**
  - EMPTY, as it should.
- No amount of examples will reveal a bug…

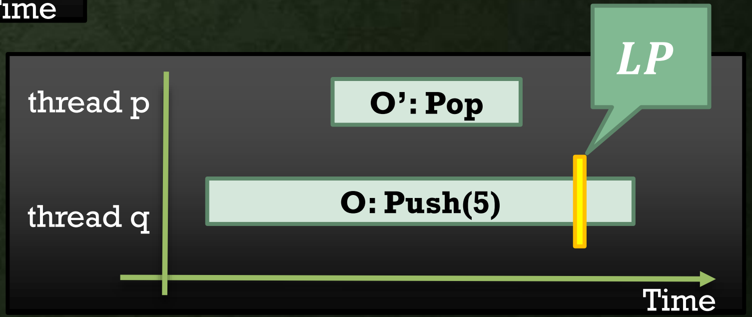
```
void stack::push(int key)
  node * n = new node(key);
  while (true) {
    node * curr = top;
    n->next = curr;
    if (CAS(&top, curr, n)) return;
  }
```

```
int stack::pop()
  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      return curr->key;
  } }
```

E1



E2



Let’s choose an **LP** for **pop**

- How about a successful CAS?
  - Can only linearize there **if** there ***is*** one…
  - If Pop returns EMPTY, there is no such CAS to linearize at!
  - (Similarly if a pop *crashes* before performing such a CAS)
- We linearize **pop** operations that **change the stack** differently from pop operations that only **query the stack**

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

```
int stack::pop()
  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      return curr->key;
  } }
```

Case 1: pop performs a successful CAS

- **Heuristic question** for **updates**: Which step makes the effect of this pop visible to other threads?
- Answer: the successful CAS
  - Before the CAS, the value is not yet popped---other threads can still see it.
  - After the CAS, they cannot.
- So we linearize at the **successful CAS**

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

```
int stack::pop()
  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      return curr->key;
  } }
```

Case 2: pop returns EMPTY

- **Heuristic question** for **queries**
  - Can you identify a **critical step S** where the pop becomes aware of the effects of other update operations?
  - Probing questions for identifying S
    - Can we change the return value of the pop by scheduling a new update operation **just before S**?
    - Are update operations **after S** essentially ignored by the pop?
- Answer: the **last read of top**
  - Suppose the stack is empty when we read top
  - Then we will return EMPTY (ignoring any updates that occur after we read top).
  - If an **update** operation changes top just before we read it, we will no longer return EMPTY
- So, we linearize at the **last read of top**

Exercise: show that it is **wrong** to choose the **last read of top** as the LP in **Case 1**.

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

```
int stack::pop()
  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      return curr->key;
  } }
```

Note: we have given linearization points for pop()s that perform a successful CAS, or return EMPTY.

What about pop()s that **crash** (without performing a successful CAS)?

SUMMARIZING SO FAR

- **<u>Plausible</u>** linearization points
  - Push: the successful CAS
  - Pop:
    - The successful CAS *if there is one*
    - Otherwise, the last read of top
- That argument was **<u>not</u>** a linearizability proof! It was a heuristic for <u>choosing LPs</u>.
- **Need to prove:** with this choice of LPs, **<u>every stack execution</u>** is linearizable.
- How do we prove this?

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

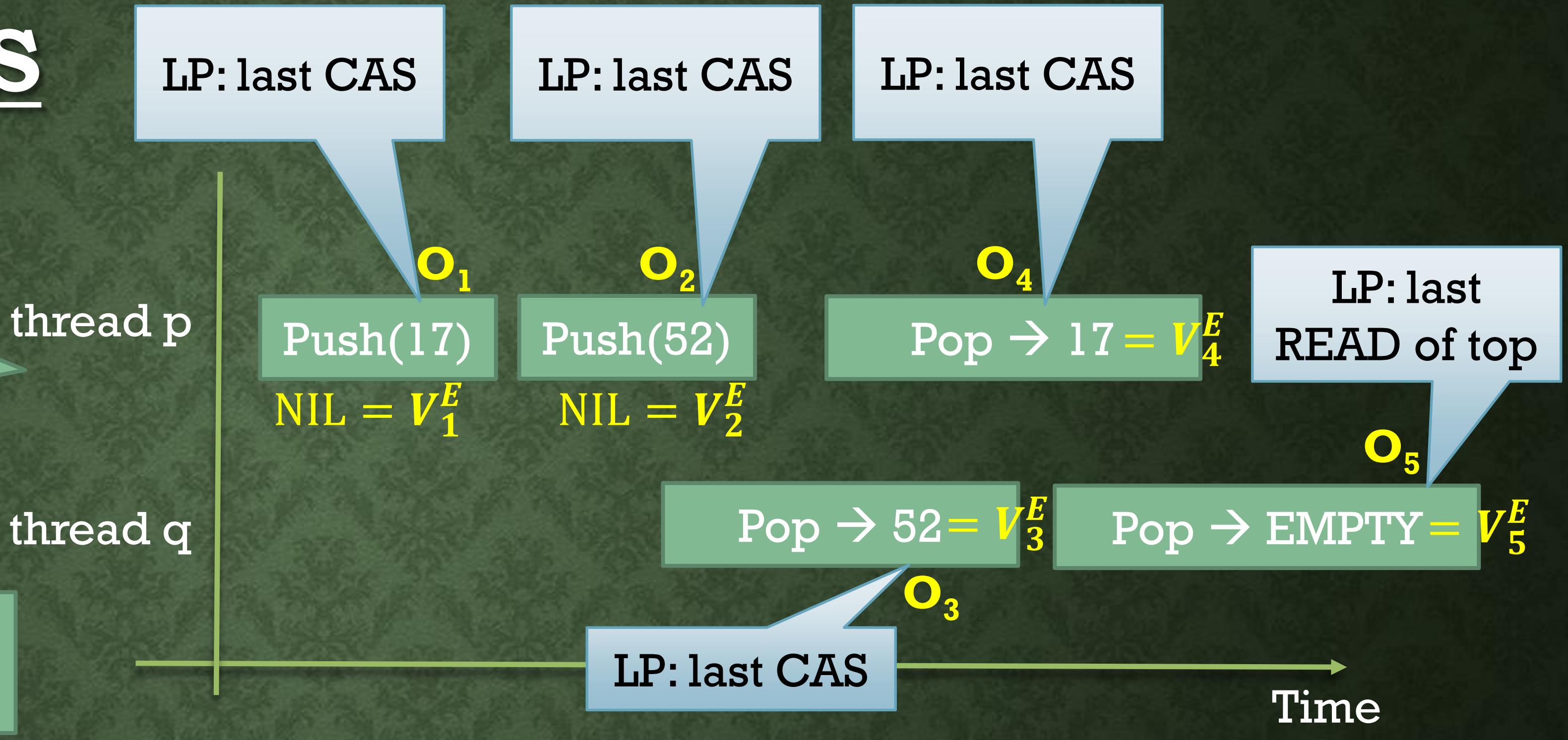
```
int stack::pop()
  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      return curr->key;
    }
  }
```

PROOF MECHANICS

- Let E be **any** concurrent execution

For example, execution E might be something like this (but you, as the theorem prover, **would not know** what E looks like)

E represents every possible execution. This example just illustrates one.



- Let L be the linearized execution induced by the LPs in E
  - In the example, the **steps of L** are **the invocations and responses of**: Push(17), Push(52), Pop, Pop, Pop
- Let $O_1, O_2, \ldots, O_k$ be the operations in order of their LPs
- Let $V_1, V_2, \ldots, V_k$ be their return values in L, as defined **by the sequential ADT**
  - In the example: NIL, NIL, 52, 17, EMPTY. (values derived directly the **definition** of a stack)
- Let $V_1^E, V_2^E, \ldots, V_k^E$ be the corresponding return values in E (must match V_i to get linearizability)

PROOF MECHANICS

- Since E is any execution, everything you prove about E is actually proved for all executions.
- **Ultimate goal is to prove $V_i^E = V_i$ for all i** (creativity needed here)
  - Usually proved **by induction** over the sequence of steps (reads/writes/CASs) in E
- What information can you use to prove this?
  - E is just "any" execution… don't know anything specific about it
  - Can only use facts that hold for **every** execution of the stack (and hence for E)
  - Sometimes some **other theoretical machinery** is used to help the proof

More on this next…

USEFUL THEORETICAL MACHINERY

This lemma can help a lot in relating V_i to V_i^E

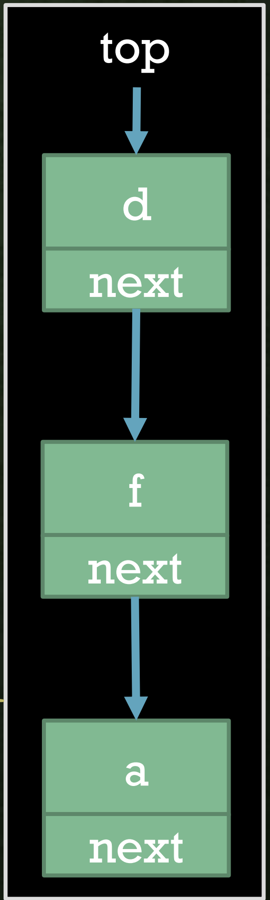
- Formal proofs often include a lemma establishing **equivalence** between a **physical state** (**PHYS**) and an **abstract state** (**ABS**)
- **ABS** is defined (somehow) by the **sequential ADT**
- **PHYS** is defined (somehow) by the **contents of memory**
- **Defining ABS for our stack:**
  - **ABS** is a sequence of keys $k_1, k_2, \dots, k_m$
  - Push(k) adds k at the start of the sequence, **shifting** other keys to the right
  - Pop() removes the first key, **shifting** other keys to the left

So if ABS $= d, f, a$ before Push(k), then ABS $= k, d, f, a$ after Push(k)

So if ABS $= d, f, a$ before Pop(), then ABS $= f, a$ after Pop()

- **Defining PHYS for our stack:**
  - **PHYS** is a sequence of keys **defined by** the sequence of nodes obtained by starting at **top** and **following next pointers**

Example: PHYS $= d, f, a$



STACK LINEARIZABILITY PROOF:
STARTING WITH A POWERFUL INVARIANT

- **Lemma: ABS = PHYS at all times in E**
- Base case: initially ABS=PHYS=empty sequence
- Proof by induction over the sequence of **all steps** $s_1, s_2, \dots, s_k$ (taken by all threads) in E:
- Let $\textbf{ABS}_i$ denote ABS after $s_1, \dots, s_i$ (& same for PHYS_i)
- IH: suppose $\textbf{ABS}_{i-1} = \textbf{PHYS}_{i-1}$
- We prove $\textbf{ABS}_i = \textbf{PHYS}_i$
- What steps can change ABS or PHYS?
- PHYS only changes at a **successful CAS**
- ABS only changes at **LPs** of ops that **change** the stack
  - These **happen to be** the steps that change PHYS!
- So WLOG assume s_i is a successful CAS.

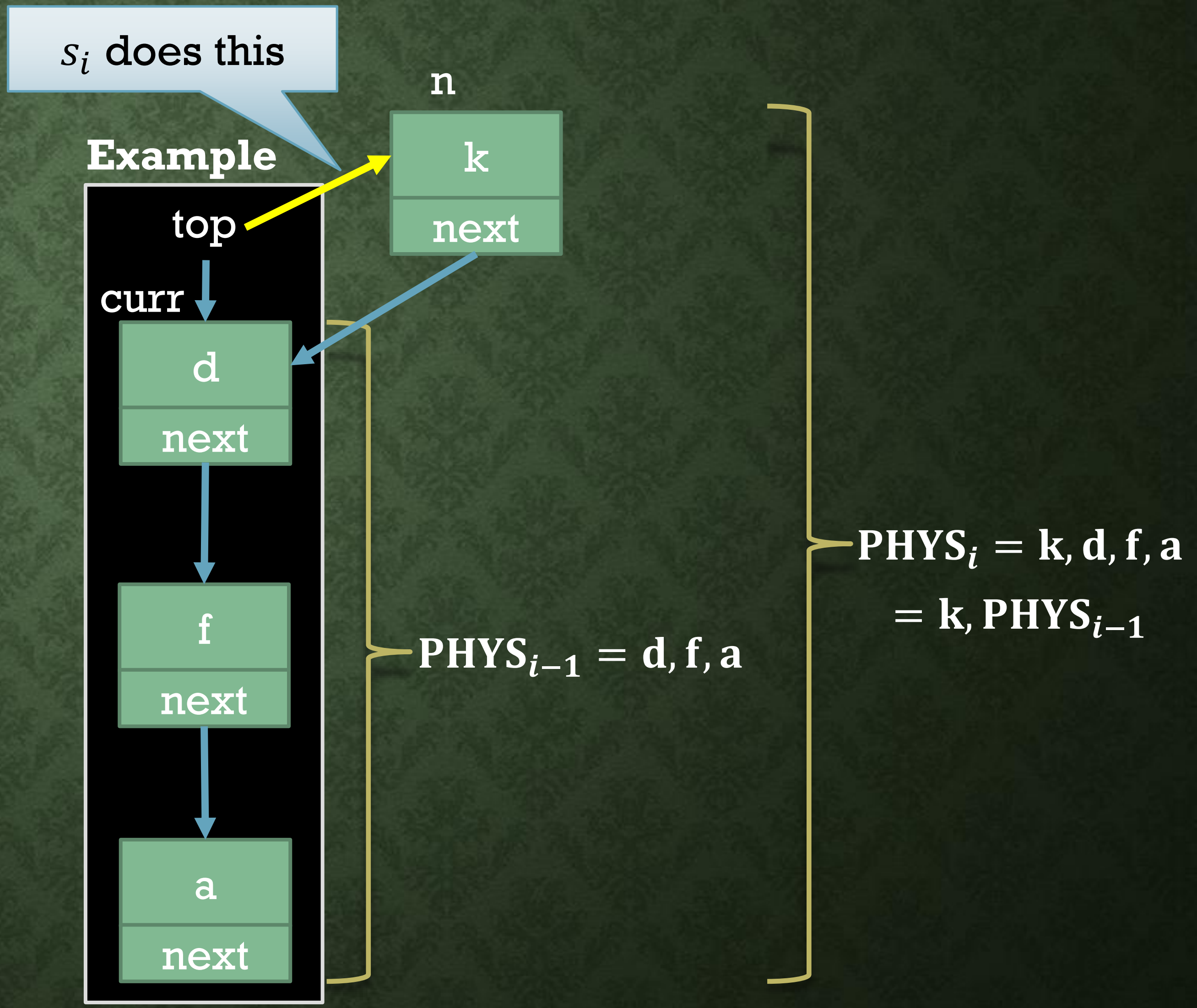
```
void stack::push(int key)
  node * n = new node(key);
  while (true) {
    node * curr = top;
    n->next = curr;
    if (CAS(&top, curr, n)) return;
  }
```

```
int stack::pop()
  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      return curr->key;
    }
```

Two cases arise…

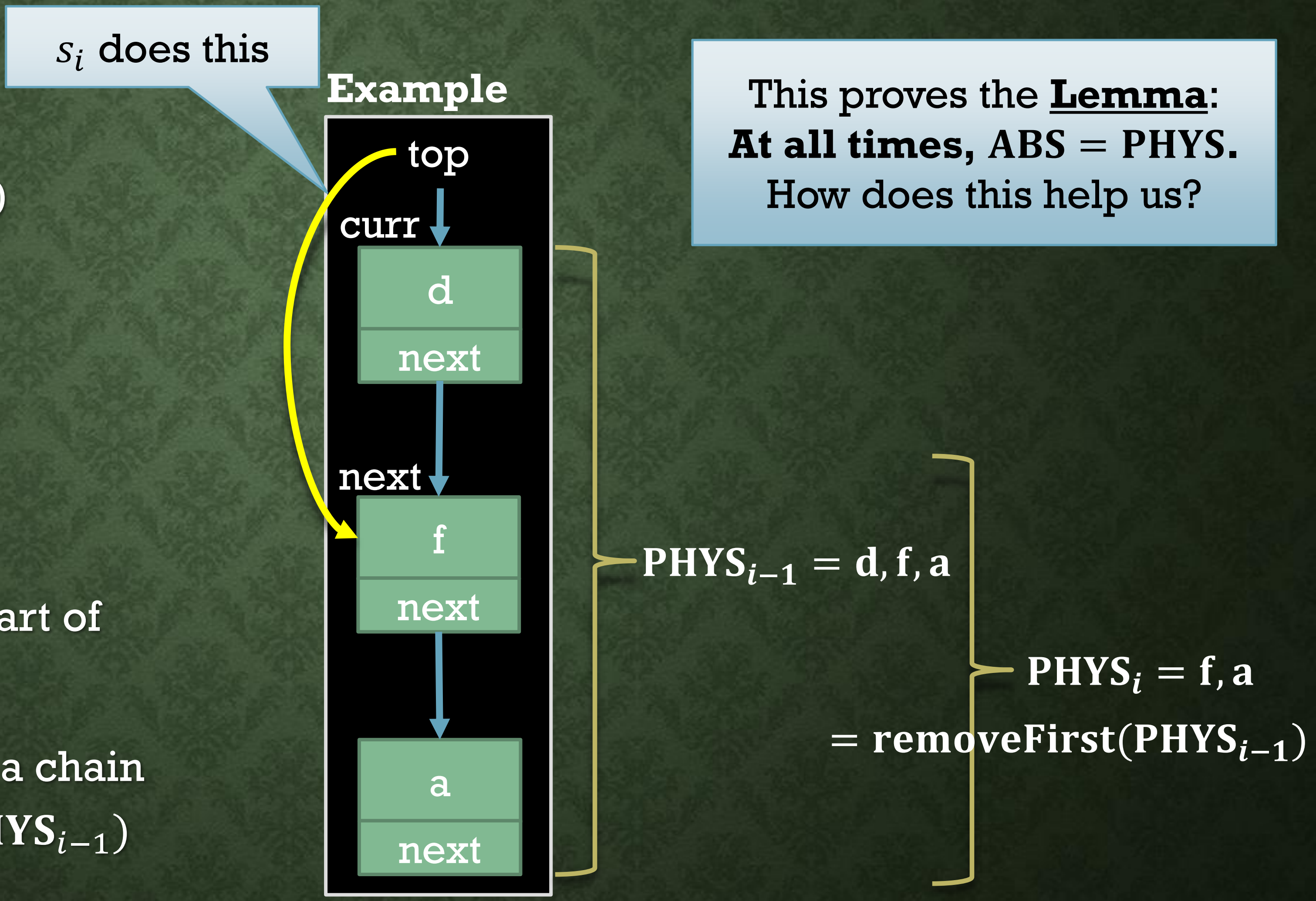
PROVING THE LEMMA: CASE 1

- IH: suppose $\mathbf{ABS}_{i-1} = \mathbf{PHYS}_{i-1}$
- We prove $\mathbf{ABS}_i = \mathbf{PHYS}_i$
- **<u>Case 1</u>:** s_i is a successful CAS in Push(k)
- How does s_i change ABS?
- $\mathbf{ABS}_i = \mathbf{k}, \mathbf{ABS}_{i-1}$ (k becomes the first key)
- How does s_i change PHYS?
  - It changes top from curr to n
- Just before s_i, top points to curr, the start of a chain of nodes that contain keys $\mathbf{PHYS}_{i-1}$
- After s_i, top points to n, which points to curr
- So, $\mathbf{PHYS}_i = \mathbf{k}, \mathbf{PHYS}_{i-1}$
- By IH, we have $\mathbf{PHYS}_i = \mathbf{k}, \mathbf{ABS}_{i-1}$, which is $\mathbf{ABS}_i$



PROVING THE LEMMA: CASE 2

- IH: suppose $\mathbf{ABS}_{i-1} = \mathbf{PHYS}_{i-1}$
- We prove $\mathbf{ABS}_i = \mathbf{PHYS}_i$
- **Case 2:** s_i is a successful CAS in Pop()
- How does s_i change ABS?
- $\mathbf{ABS}_i = \mathbf{removeFirst}(\mathbf{ABS}_{i-1})$
- How does s_i change PHYS?
  - It changes top from curr to next
- Just before s_i, top points to curr, the start of a chain of nodes that contain $\mathbf{PHYS}_{i-1}$
- After s_i, top points to next, the start of a chain of nodes that contain $\mathbf{removeFirst}(\mathbf{PHYS}_{i-1})$
- So, $\mathbf{PHYS}_i = \mathbf{removeFirst}(\mathbf{PHYS}_{i-1})$
- By IH, $\mathbf{PHYS}_i = \mathbf{removeFirst}(\mathbf{ABS}_{i-1})$, which is $\mathbf{ABS}_i$



This proves the **Lemma**:
At all times, ABS = PHYS.
How does this help us?

WHAT EXACTLY DID WE JUST PROVE?

- We have proved ABS = PHYS at all times.
- Intuition: this means the contents of memory (PHYS) **contains** the "correct" answer (ABS)
- This does **NOT** mean our operations actually **find and return** the correct answer
  - For example, if we changed the code for Pop to always return EMPTY,
    we could still prove ABS = PHYS at all times.
  - But, whenever the stack is non-empty, our return values would be completely incorrect!
- We are NOT done!
  - We still need to prove that the return values of operations actually correspond to PHYS somehow!
  - ***Then,*** our invariant PHYS = ABS will establish that our return values are **correct**.

PROVING RETURN VALUES ARE CORRECT

```
int stack::pop()
  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      return curr->key;
  } }
```

- Lemma: every (non-crashed) Pop operation O returns the first key that was in PHYS at the time O was linearized.
- Case 1: Suppose O returns EMPTY.
  - Then O is linearized at its last READ of top, which returned NULL, so **PHYS was empty** at O's LP.
- Case 2: Suppose O returns curr->key. Then O is linearized at its successful CAS.
  - Since this CAS succeeds, top pointed to curr at O's LP.
  - Thus, when O was linearized, the first key of PHYS was precisely curr->key, which O returns.
- Corollary: every Pop returns the first key that was in ABS when that Pop was linearized
- Theorem: every operation in E returns the same value as it would in L.
  - Push has no return value.
  - By the Corollary, every Pop in E returns the first key in ABS, which is exactly what it returns in L.
  - This proves the stack is <u>linearizable</u>.