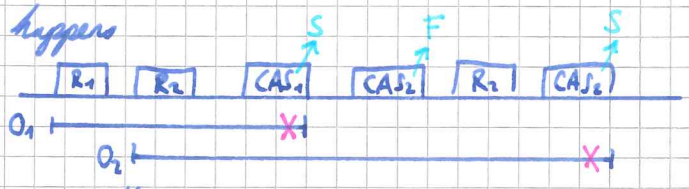
L13 Concurrent Lists

Model:
- n threads
- Read, Write, COMPUTE "steps"
- arbitrary interleaving

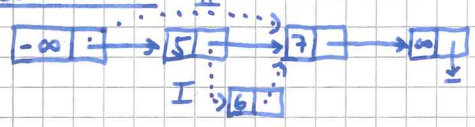
"Blocking" (lock-based) ↳ assumes threads are fairly scheduled

or "Non-Blocking" ↳ allows for thread crashes

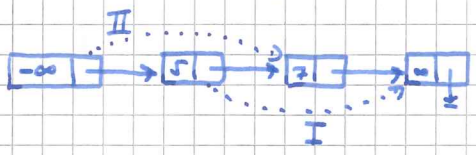
A) Program properties: something eventually happens
B) Consistency properties: linearizability



Sorted List and what can happen concurrently:



⇒ the case of the missing insert



⇒ the case of the missing remove

⇒ memory reclamation

Try 0: a single global lock on the head of the list → no parallelization at all

Try 1: Fine-grained list with a seperate lock on each node

```
bool Add(Node* node) {
    head.lock()
    Node* pred = head
    Node* curr = pred→next
    curr.lock()
    while(curr.key < node→key) {
        pred.unlock()
        pred = curr
        curr = curr→next
        curr.lock()
    }
    if (curr.key == node→key) {
        unlock_all()
        return False
    } else { //set pointers
        pred→next = node
        node→next = curr
        unlock_all()
        return True
    }
```

"hand over hand" walk through the list

Notes:
1) logical vs. physical
2) linearization
3) is it any good? ⊕ correct ⊖ slow

→ Remove operation as exercise

Try 2: Optimistic list
- sequential traversal
- locking pred, succ
- sequential traversal to check logical presence (reachability)

Try 3: Lazy list

→ add a "unmarked" bit ≈ reachable to each node

```
bool Validate(Node *pred, Node* curr)
  return (!pred→marked && !curr→marked && pred→next == curr)
```

```
bool Add(Node *node) {
  while (true) {
    Node* pred = head
    Node* curr = pred→next
    while (curr→key < node→key) {
      pred = curr; curr = curr→next; }
    pred.lock(); curr.lock()
    if (VALIDATE(pred, curr)) {
      if (curr→key == node→key) {
        unlock_all(); return FALSE; }
      else {
        node→next = curr;
        pred→next = node;
        unlock_all(); return TRUE; }}}}
```

```
bool Remove(int key) {
  while (true) {
    Node *pred = head
    Node* curr = pred→next
    while (curr→key < node→key) {
      pred = curr; curr = curr→next; }
    pred.lock(); curr.lock();
    if (VALIDATE(pred, curr)) {
      if (curr→key != key)
        unlock_all(); return FALSE; }
      else {
        curr→marked = TRUE
        pred→next = curr→next
        unlock_all()
        return TRUE; }}}}
```

Notes: 1) Not starvation-free

2) Fast in practice (contain-operation as fast as sequential)

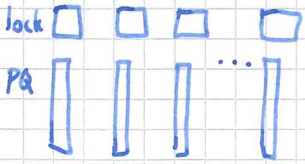
3) lock-free? Tim Harris (DISC 2002): Use LSB of next-pointer as marked bit

4) Generalizes to skiplists & Hash Tables

Concurrent PQ (priority queue)

n threads

m sequential PQs




```
insert(elem, pr)
  i ← uniform_random(1, )
  do { flag ← Q[i].try_lock() } while(!flag)
  Q[i].insert(elem, pr)
  Q[i].unlock()
```

```
T DeleteMin()
  do { i,j ← uniform_random(1,m)
       if (Q[i].min > Q[j].min) swap i,j
       flag ← Q[i].try_lock() } while(!flag)
  elim ← Q[i].DeleteMin()
  Q[i].unlock()
  return elim.
```

Thm: Under assumptions, the global rank of an element removed is $O(m)$ in expectation and $O(m \log m)$ w.p. $\geq 1 - \frac{1}{m^2}$.