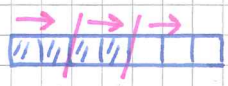
L12 Concurrent Data Structures

1) Mutual Exclusion
2) Consistency
3) Lock-Free Data Structures

Motivation: Summing Problem

float myArray[n]; // $n = 10^6$



function: float COMPUTE(float value)

want: result = $\sum_{v \text{ in myArray}}$ COMPUTE(v)

```
parallel for (int i=0; i<n; i++)
    temp ← COMPUTE(myArray[i])
    result ← temp + result
```

(result ← = Write; + result = Read)

what we want: 0 read^0 write^1 read^1 write^2

what could happen: 0 read^0 read^0 write^1 write^1 ↯

Goals: 1) Consistency with sequential Execution
2) Speedup of Multiple Threads

Mutual Exclusion (Dijkstra 1965)

Idea: shared lock variable

Properties: 1) No two threads in critical section at the same time.
2) In any non-trivial suffix, some lock or unlock call succeeds.

```
var L;
//
L.lock()
    result ← temp + result   // Critical Section
L.unlock()
```

Peterson's Mutex Algorithm

- 2 threads, steps: Read, Write, Compute()
- arbitrary interleavings of steps possible
- each thread takes infinite # steps.

In short: „Make yourself seen and be nice!"

Shared: var x
he_wants = F
she_wants = F
enum {his, her} turn

Alice's Mutex Code

```
she_wants = T
turn = his
while(he_wants == T & turn == his);
→ ACCESS(x)
she_wants = F
```

Bob's Mutex Code

```
he_wants = T
turn = hers
while(she_wants == T & turn == hers);
→ ACCESS(x)
he_wants = F
```

Proof of Mutex

Assume for contradiction that A, B both in critical section. Let B be last to write to 'turn' before CS.

$write_A(turn) <_x write_B(turn)$ (happened before)
$write_A(she\_wants) <_x write_A(turn)$
$write_B(turn) <_x read_B(she\_wants) <_x read_B(turn)$
↳ true ↳ hers

→ This contradicts B being in the CS as it would be spinning in the loop when reading these values. ↯

□

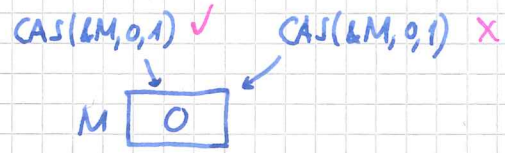
For this proof to work, we assumed Sequential Consistency: There exists a global order of operations consistent with some sequential order. In practice: enforced with MEMFENCE op.

Lock-Free Algorithms (how to get rid of the 'no thread ever dies'-assumption)

bool Compare-And-Swap (&M, old, new)

performs atomically:

if (M == old) { M = new; return T; } else { return F; }



Ex1: A Lock

```
var L ← F
Lock()
  do {} while(!CAS(&L,F,T))
Unlock()
  L ← F
```

Ex2: Summing Problem

```
for (_)
  tmp = COMPUTE(myArray[i])
  do {
     old_val = result
     new_val = result + tmp
  } while(!CAS(&result, old_val, new_val))
```

Property (Non-blocking / lock-free / latch-free)

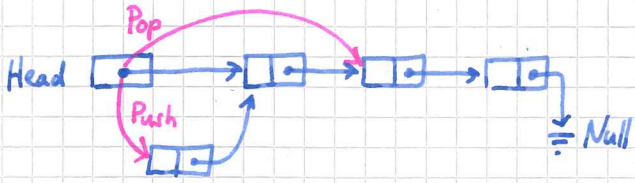
In any suffix with a large enough # steps, ∃ some operation which succeeds.

Proof (by contradiction)

Assume σ infinite, in which no operations return. ∃ thread P which takes ∞ steps without success. Case 1: CAS is successful at some point ↯

Case 2: CAS not successful → value changed → some other operation is successful. □

Lock-free Stack (Treiber, 1985)




```
struct Stack {
  Node * head; ... }
struct Node {
  int val; Node* next; }
void Push(Node*new) {
  do { new→next = head }
  while(!CAS(head, node→next, new)) }
Node* Pop() {
  Node *crt = head;
  while(crt) {
    if (CAS(head, crt, crt→next)) break;
    crt = head;
} return crt;
```

Notes:

- With CAS we get more than sequential consistency: linearizability (each operation allows serialization point within the time interval of the operation)
- issues with node's memory management (segfault, reference counting, hazard lists)