CS 636: Shared Memory Synchronization

Swarnendu Biswas

Semester 2020-2021-II
CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

What is the desired property?

```
class Set {
  final Vector elems = new Vector();

  void add(Object x) {
    if (!elems.contains(x)) {
      elems.add(x);
    }
  }

}

class Vector {
  synchronized void add(Object o) { ... }
  synchronized boolean contains(Object o) { ... }
}
```

What is the desired property?

```
Q.insert(elem):
  atomic {
    while (Q.full()) {}
    // Add elem to the Q
  }
```

```
Q.remove():
  atomic {
    while (Q.empty()) {}
    // Return data from Q
  }
```

Synchronization Patterns

- Mutual exclusion

```
lock:bool := false
```

```
Lock.acquire():
  while TAS(&lock)
    // spin
```

```
Lock.release():
  lock := false
```

- Condition synchronization

```
while ¬ condition
  // do nothing (spin)
```

- Global synchronization

Locks (Mutual Exclusion)

```
public interface Lock {
  public void lock();
  public void unlock();
}


…

public class LockImpl
implements Lock {

  …

  …
}
```

```
Lock mtx = new LockImpl(…);
…



mtx.lock();
try {
  … // body
} finally {
  mtx.unlock();
}
```

Desired Synchronization Properties

- Mutual exclusion
  - Critical sections on the same lock from different threads do not **overlap**
  - Safety property

- Livelock freedom

  If a lock is available, then **some** thread should be able to acquire it within bounded steps

Deadlock-Free

- If some thread calls **lock()**
  - And never returns
  - Then other threads must complete **lock()** and **unlock()** calls infinitely often
- System as a whole makes progress
  - Even if individuals starve

Starvation-Free

- If some thread calls lock()
  - It will eventually return
- Individual threads make progress

Desired Synchronization Properties

- Deadlock freedom
  - If a thread attempts to acquire the lock, then **some** thread should be able to acquire the lock
  - Individual threads may starve
  - Liveness property
- Starvation freedom
  - Every thread that acquires a lock **eventually** releases it
  - A lock acquire request must eventually succeed within **bounded** steps
  - Implies deadlock freedom

Classic Mutual Exclusion Algorithms

LockOne: What could go wrong?

```
class LockOne implements Lock {
  private boolean[] flag = new boolean[2];

  public void lock() {
    flag[i] = true;
    j = 1-i;
    while (flag[j]) {}
  }
}
```

Deadlock Freedom

- LockOne Fails deadlock-freedom
  - Concurrent execution can deadlock

```
flag[i] = true;     flag[j] = true;
while (flag[j]){}   while (flag[i]){}
```

  - Sequential executions OK

LockOne Satisfies Mutual Exclusion

- Assume CS_A^j overlaps CS_B^k
- Consider each thread's last (j-th and k-th) read and write in the lock() method before entering
- Derive a contradiction

From the Code

- $\text{write}_A(\text{flag[A]=true}) \rightarrow \text{read}_A(\text{flag[B]==false}) \rightarrow \text{CS}_A$
- $\text{write}_B(\text{flag[B]=true}) \rightarrow \text{read}_B(\text{flag[A]==false}) \rightarrow \text{CS}_B$

```
class LockOne implements Lock {
  …
  public void lock() {
    flag[i] = true;
    j = 1 - i;
    while (flag[j]) {}
  }
}
```

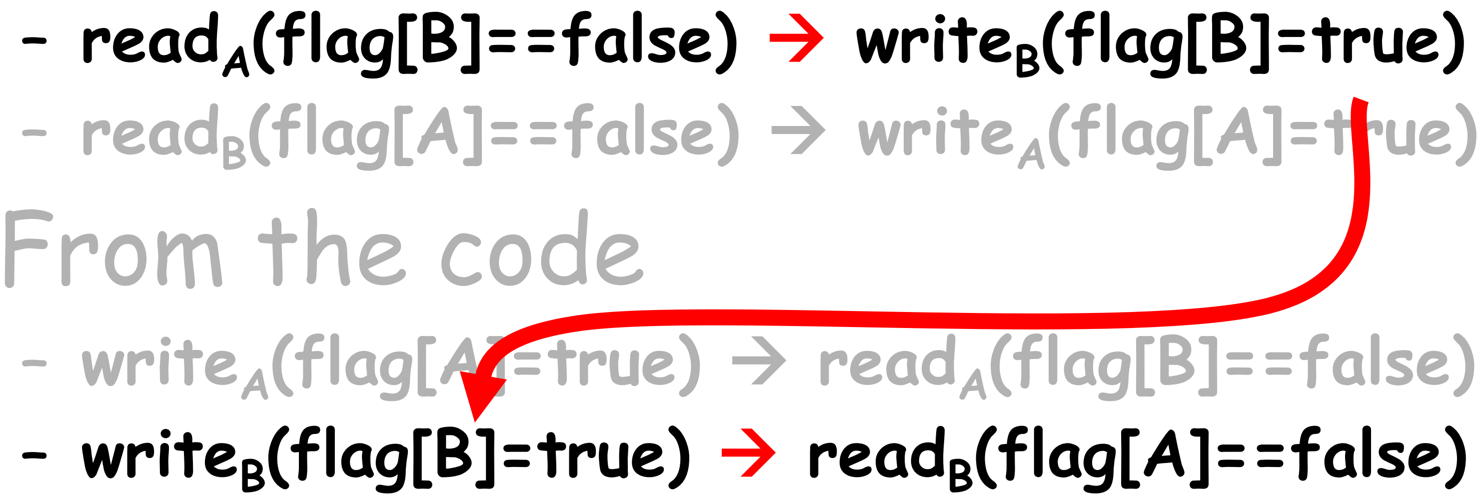
From the Assumption

- **read$_A$(flag[B]==false) → write$_B$(flag[B]=true)**
- **read$_B$(flag[A]==false) → write$_A$(flag[B]=true)**

Combining

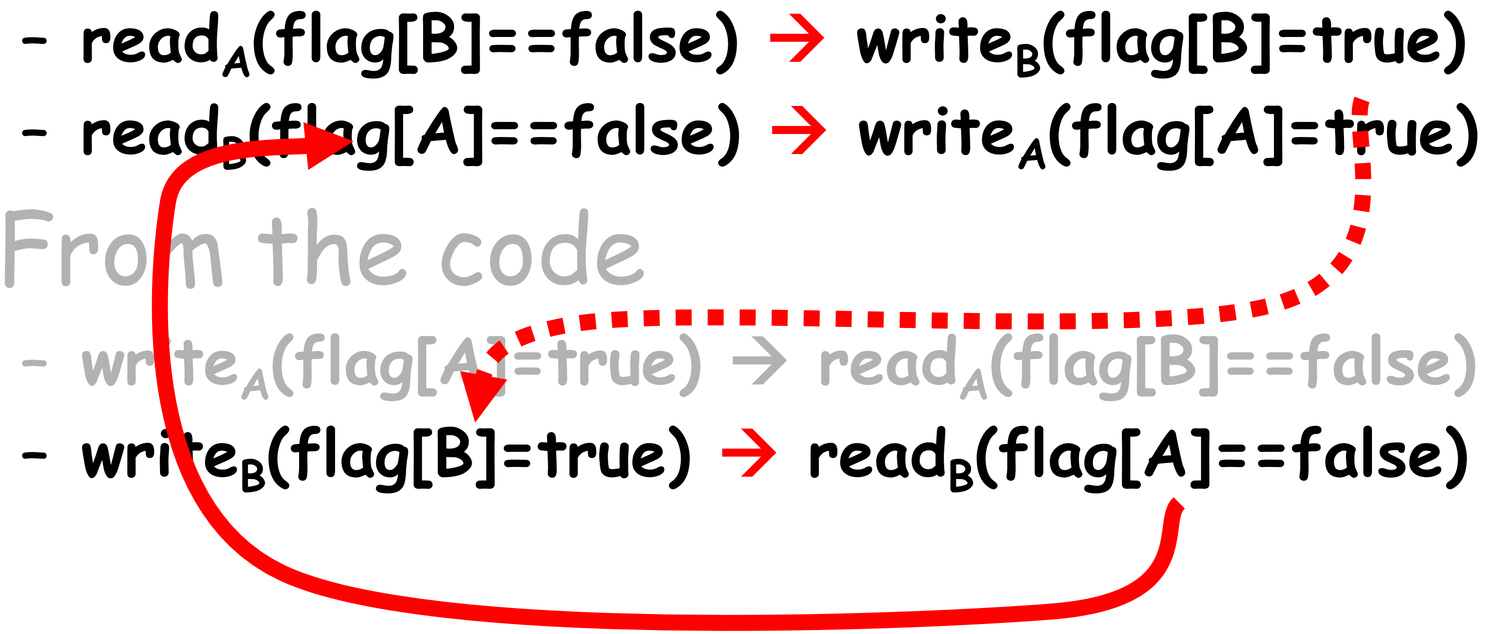
- Assumptions:
  - $\text{read}_A(\text{flag}[B]==\text{false}) \rightarrow \text{write}_B(\text{flag}[B]=\text{true})$
  - $\text{read}_B(\text{flag}[A]==\text{false}) \rightarrow \text{write}_A(\text{flag}[A]=\text{true})$
- From the code
  - $\text{write}_A(\text{flag}[A]=\text{true}) \rightarrow \text{read}_A(\text{flag}[B]==\text{false})$
  - $\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow \text{read}_B(\text{flag}[A]==\text{false})$

Combining



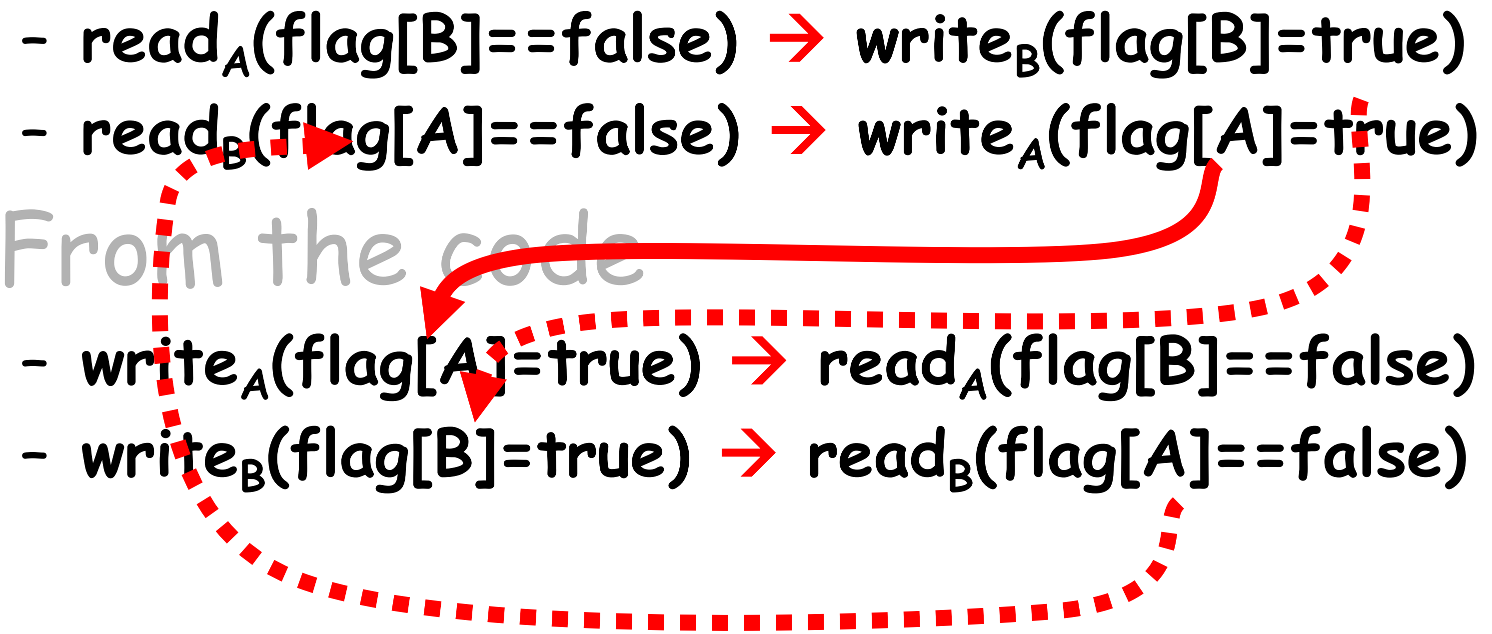
- Assumptions:
  - **read$_A$(flag[B]==false) → write$_B$(flag[B]=true)**
  - read$_B$(flag[A]==false) → write$_A$(flag[A]=true)
- From the code
  - write$_A$(flag[A]=true) → read$_A$(flag[B]==false)
  - **write$_B$(flag[B]=true) → read$_B$(flag[A]==false)**

Combining



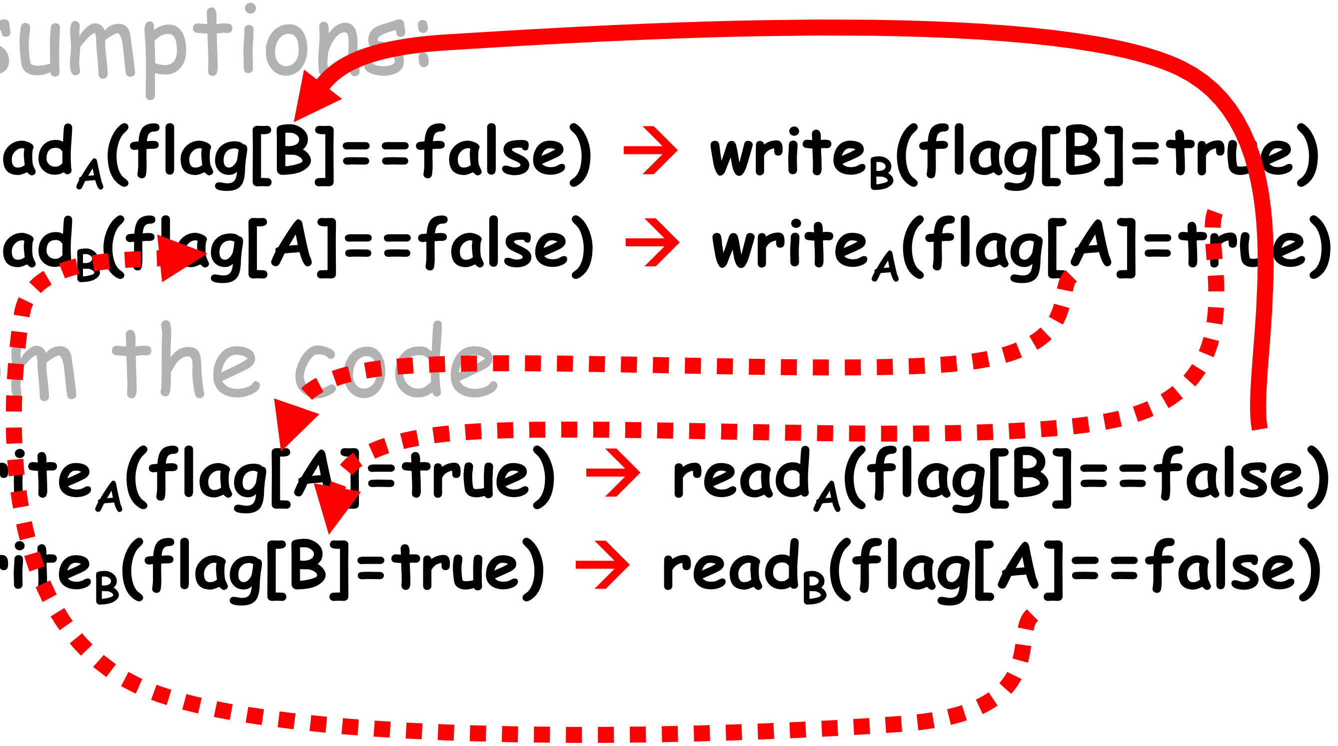
- Assumptions:
  - $\text{read}_A(\text{flag[B]==false})$ → $\text{write}_B(\text{flag[B]=true})$
  - $\text{read}_B(\text{flag[A]==false})$ → $\text{write}_A(\text{flag[A]=true})$
- From the code
  - $\text{write}_A(\text{flag[A]=true})$ → $\text{read}_A(\text{flag[B]==false})$
  - $\text{write}_B(\text{flag[B]=true})$ → $\text{read}_B(\text{flag[A]==false})$

Combining



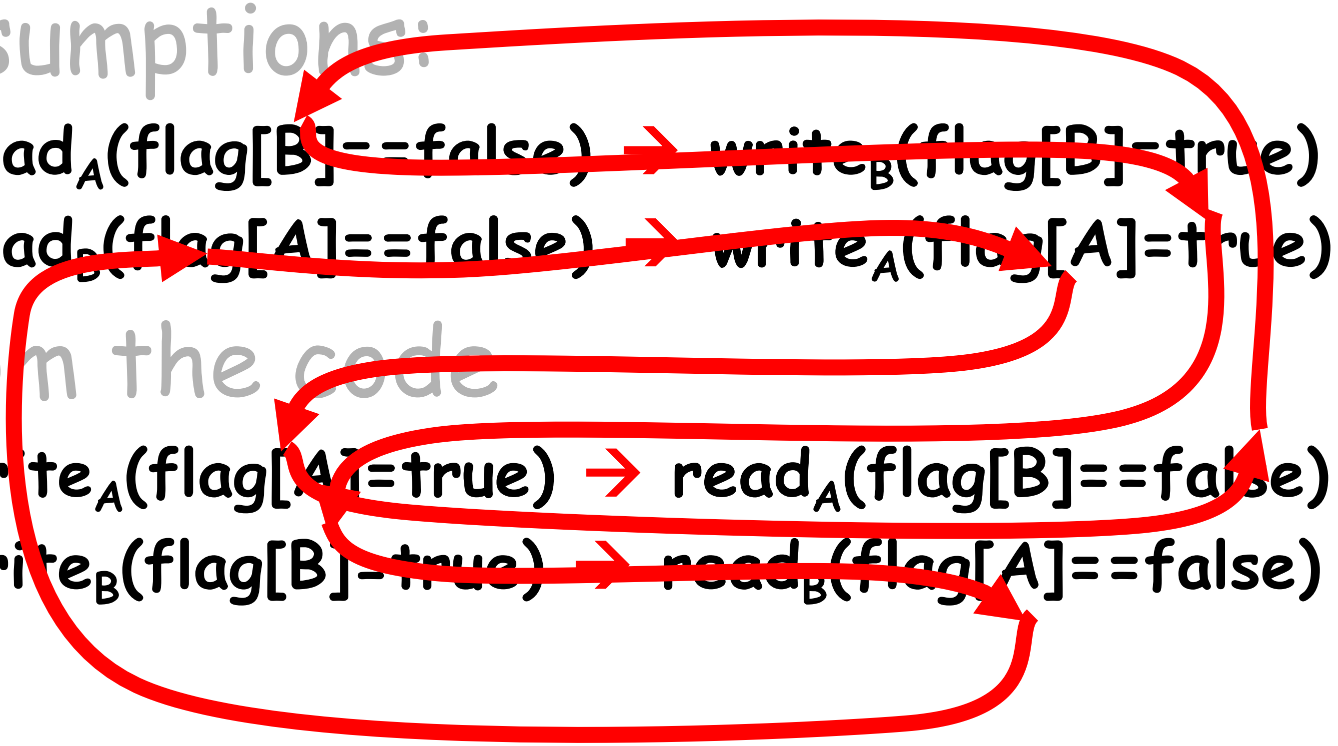
- Assumptions:
  - read$_A$(flag[B]==false) → write$_B$(flag[B]=true)
  - read$_B$(flag[A]==false) → write$_A$(flag[A]=true)
- From the code
  - write$_A$(flag[A]=true) → read$_A$(flag[B]==false)
  - write$_B$(flag[B]=true) → read$_B$(flag[A]==false)

Combining



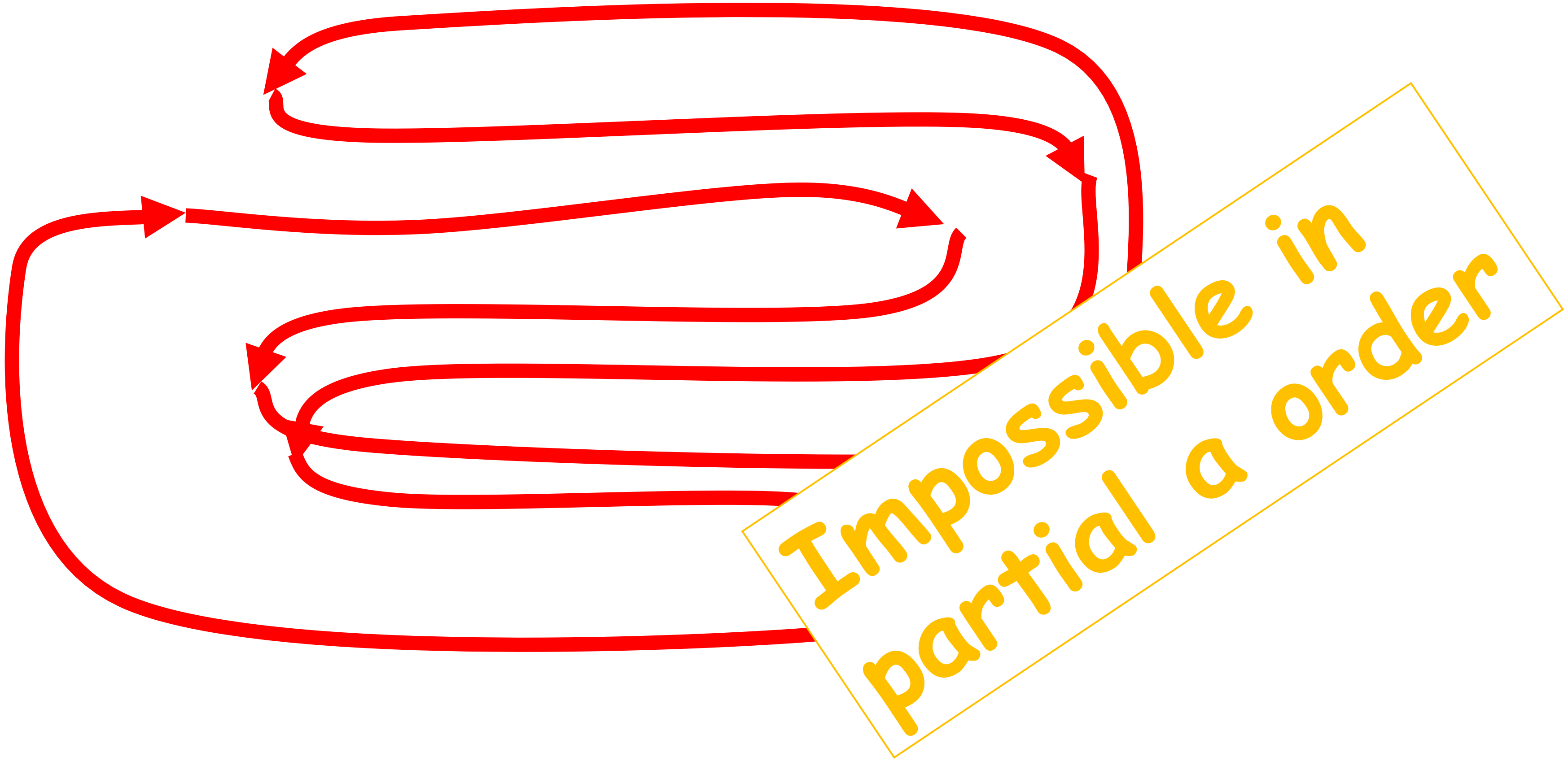
- Assumptions:
  - read_A(flag[B]==false) → write_B(flag[B]=true)
  - read_B(flag[A]==false) → write_A(flag[A]=true)
- From the code
  - write_A(flag[A]=true) → read_A(flag[B]==false)
  - write_B(flag[B]=true) → read_B(flag[A]==false)

Combining



- Assumptions:
  - read$_A$(flag[B]==false) → write$_B$(flag[B]=true)
  - read$_B$(flag[A]==false) → write$_A$(flag[A]=true)
- From the code
  - write$_A$(flag[A]=true) → read$_A$(flag[B]==false)
  - write$_B$(flag[B]=true) → read$_B$(flag[A]==false)

Cycle!



LockTwo: What could go wrong?

```
public class LockTwo implements Lock {
  private int victim;
  public void lock() {
    victim = i;
    while (victim == i) {};
  }

  public void unlock() {}
}
```

LockTwo Claims

- Satisfies mutual exclusion
  - If thread **`i`** in CS
  - Then **`victim == j`**
  - Cannot be both 0 and 1
- Not deadlock free
  - Sequential execution deadlocks
  - Concurrent execution does not

```
public void LockTwo() {
  victim = i;
  while (victim == i) {};
 }
```

Peterson’s Algorithm

```
class PetersonLock {

  static volatile boolean[] flag = new boolean[2];
  static volatile int victim;

  public void unlock() {
    int i = ThreadID.get();
    flag[i] = false;
  }
```

```
  public void lock() {
    int i = ThreadID.get();
    int j = 1-i;
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {}
  }

}
```

Mutual Exclusion

```
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
```

- If thread **0** in critical section,
  - `flag[0]=true`,
  - `victim = 1`

- If thread **1** in critical section,
  - `flag[1]=true`,
  - `victim = 0`

Cannot both be true

Starvation Free

- Thread `i` blocked only if `j` repeatedly re-enters so that

  `flag[j] == true` and `victim == i`
- When `j` re-enters
  - it sets `victim` to `j`.
  - So `i` gets in

```
public void lock() {
  flag[i] = true;
  victim    = i;
  while (flag[j] && victim == i) {};
}

public void unlock() {
  flag[i] = false;
}
```

Deadlock Free

```
public void lock() {
   …
   while (flag[j] && victim == i) {};
```

- Thread blocked
  - only at `while` loop
  - only if it is the victim
- One or the other must not be the victim

Peterson’s Algorithm

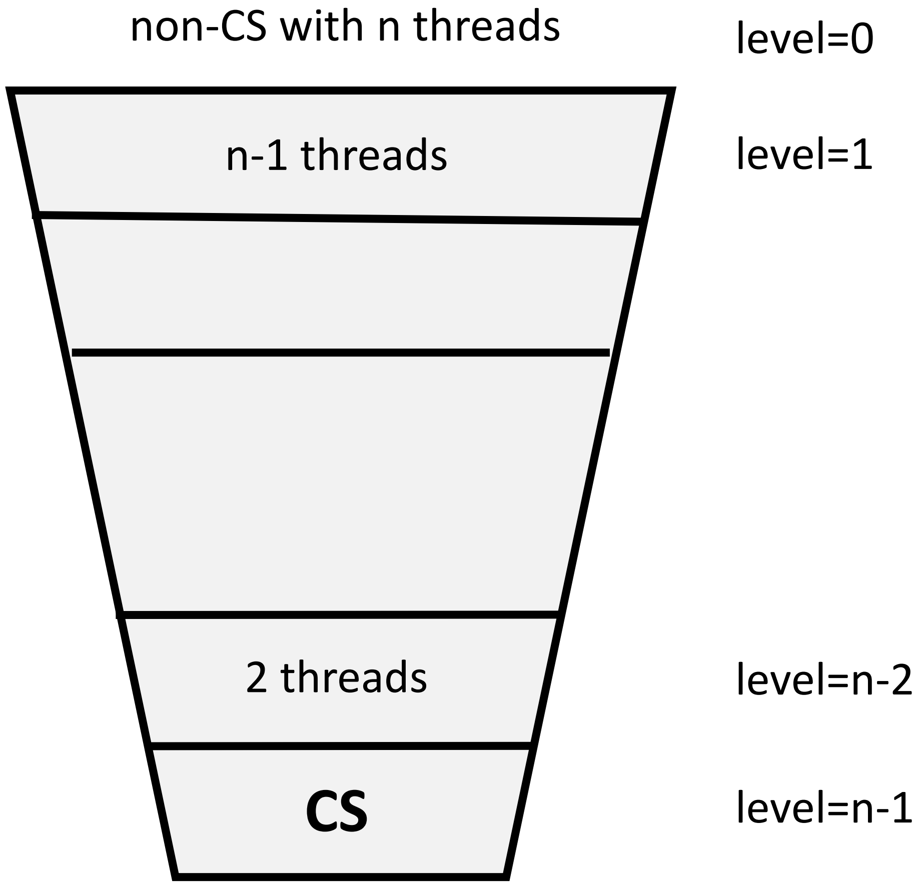
```
class PetersonLock {

  static volatile boolean[] flag = 
new boolean[2];
  static volatile int victim;

  public void unlock() {
    int i = ThreadID.get();
    flag[i] = false;
  }
```

```
public void lock() {
  int i = ThreadID.get();
  int j = 1-i;
  flag[i] = true;
  victim = i;
  while (flag[j] && victim == i) {}
}
```

- Is this algorithm correct under sequential consistency?
- What if we do not have sequential consistency?

Filter Lock for n Threads

- There are n-1 waiting rooms called "levels"
- At least one thread trying to enter a level succeeds
- One thread gets blocked at each level if many threads try to enter



Filter Lock

```
class FilterLock {

  volatile int[] level;
  volatile int[] victim;

  public FilterLock() {
    level = new int[n];
    victim = new int[n];
    for (int i = 0; i < n; i++) {
      level[i] = 0;
    }
  }
```

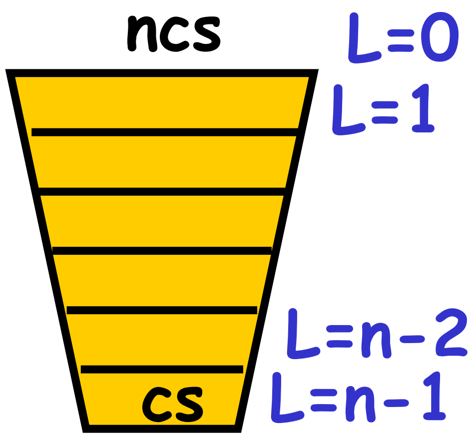
```
public void unlock() {
  int me = ThreadID.get();
  level[me]= 0;
}
```

Filter Lock

```
  …
  public void lock() {
    int me = ThreadID.get();
    for (int i = 1; i < n; i++) { // Attempt to enter level i
      level[me] = i; // visit level i
      victim[i] = me; // Thread “me” is a good guy!
      // spin while conflict exits
      while ((∃k != me) level[k] >= i && victim[i] == me) {
      }
    }
  }
}
```

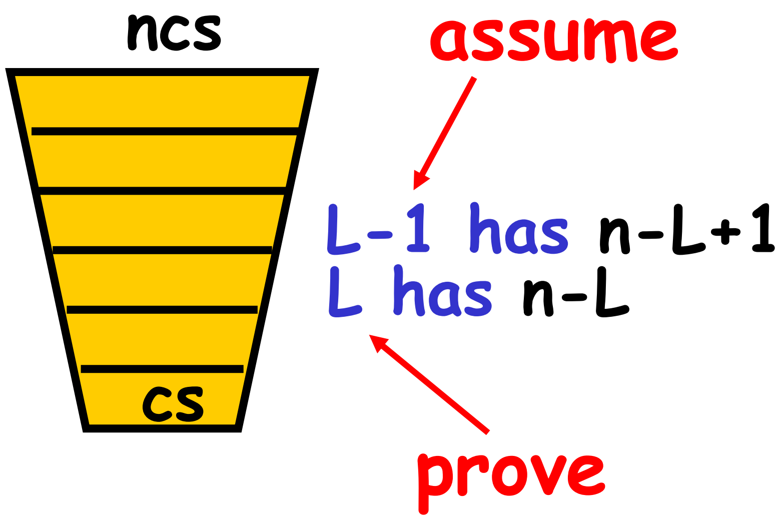
Claim

- Start at level L=0
- At most n-L threads enter level L
- Mutual exclusion at level L=n-1

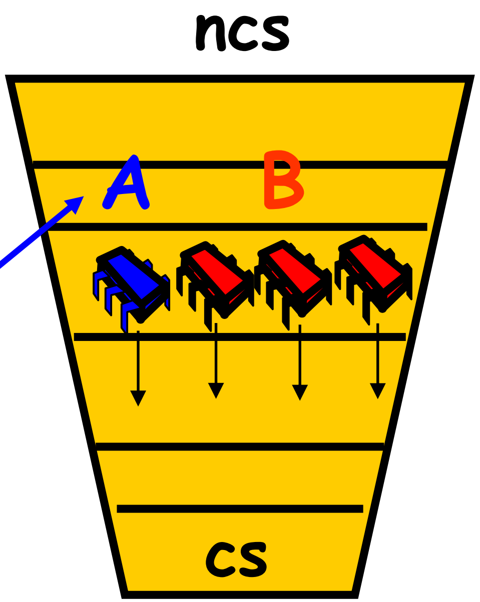


Induction Hypothesis

- No more than n-L+1 at level L-1
- Induction step: by contradiction
- Assume all at level L-1 enter level L
- A last to write victim[L]
- B is any other thread at level L



Proof Structure



Assumed to enter L-1

$n-L+1 = 4$

$n-L+1 = 4$

By way of contradiction all enter L

Last to write victim[L]

Show that A must have seen B in level[L] and since victim[L] == A could not have entered

From the Code

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow \text{write}_B(\text{victim}[L]=B)$

```
public void lock() {
 for (int L = 1; L < n; L++) {
   level[i] = L;
   victim[L]  = i;
   while ((∃ k != i) level[k] >= L)
          && victim[L] == i) {};
   }}
```

From the Code

(2) $\text{write}_A(\text{victim}[L]=A) \rightarrow \text{read}_A(\text{level}[B])$

```
public void lock() {
 for (int L = 1; L < n; L++) {
   level[i] = L;
   victim[L]  = i;
   while ((∃ k != i) level[k] >= L)
          && victim[L] == i) {};
   }}
```

By Assumption

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

By assumption, *A* is the last thread to write **victim[L]**

Combining Observations

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow \text{write}_B(\text{victim}[L]=B)$

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

(2) $\text{write}_A(\text{victim}[L]=A) \rightarrow \text{read}_A(\text{level}[B])$

Combining Observations

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow$

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

(2) $\rightarrow \text{read}_A(\text{level}[B])$

```
public void lock() {
 for (int L = 1; L < n; L++) {
   level[i]  = L;
   victim[L] = i;
   while ((∃ k != i) level[k] >= L)
          && victim[L] == i) {};
   }}
```

Combining Observations

(1) $\text{write}_B(\text{level}[B]=L)\rightarrow$

(3) $\text{write}_B(\text{victim}[L]=B)\rightarrow\text{write}_A(\text{victim}[L]=A)$

(2)

$\rightarrow\text{read}_A(\text{level}[B])$

Thus, A read level[B] ≥ L,
A was last to write victim[L],
so it could not have entered level L!

No Starvation

- Filter Lock satisfies properties:
  - Just like Peterson Alg at any level
  - So no one starves
- But what about fairness?
  - Threads can be overtaken by others

Fairness

- Starvation freedom is good, but maybe threads shouldn’t wait too much…

- For example, it would be great if we could order threads by the order in which they performed the first step of the `lock()` method

Bounded Waiting

- Divide lock() method into two parts
  - Doorway interval (D_A) – finishes in finite steps
  - Waiting interval (W_A) – may take unbounded steps

- A lock is first-come first-served if $D_A^j \rightarrow D_B^k$, then $CS_A^j \rightarrow CS_B^k$

r-Bounded Waiting

For threads A and B: if $D_A^k \rightarrow D_B^j$, then $CS_A^k \rightarrow CS_B^{j+r}$

Lamport’s Bakery Algorithm

```
class Bakery implements Lock {

  boolean[] flag;
  Label[] label;

 public void unlock() {
    flag[ThreadID.get()] = false;
  }
```

```
public Bakery(int n) {
   flag = new boolean[n];
   label = new Label[n];
   for (int i = 0; i<n; i++) {
     flag[i] = false;
     label[i] = 0;
   }
 }
```

Lamport’s Bakery Algorithm

(label[*i*], *i*) << (label[*j*], *j*)) iff label[*i*] < label[*j*] or label[*i*] = label[*j*] and *i* < *j*

```
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0], …, label[n-1]) + 1;
    while ((∃k != i) flag[k] && (label[k], k) << (label[i],i)) {}
  }

}
```

No Deadlock

- There is always one thread with earliest label
- Ties are impossible (why?)

First-Come-First-Served

- If $D_A \rightarrow D_B$ then A's label is smaller
- And:
  - $\text{write}_A(\text{label}[A]) \rightarrow$
    $\text{read}_B(\text{label}[A]) \rightarrow$
    $\text{write}_B(\text{label}[B]) \rightarrow$
    $\text{read}_B(\text{flag}[A])$
- So B is locked out while flag[A] is true

```
class Bakery implements Lock {

public void lock() {
  flag[i]  = true;
  label[i] = max(label[0],
               …,label[n-1])+1;
  while (∃k flag[k]
           && (label[i],i) >
  (label[k],k));
 }
```

First-Come-First-Served

- If $D_A \rightarrow D_B$ then A's label is sr
- A
  - –

    read$_B$(flag[A])
- So B is locked out while flag[A] is true

```
class Bakery implements Lock {
```

Deadlock-freedom together with first-come first-served implies starvation-freedom

Mutual Exclusion

- Suppose A and B in CS together
- Suppose A has earlier label
- When B entered, it must have seen
  - flag[A] is *false*, or
  - label[A] > label[B]

```
class Bakery implements Lock {

public void lock() {
  flag[i]  = true;
  label[i] = max(label[0],
                 …,label[n-1])+1;
  while (∃k flag[k]
             && (label[i],i) >
  (label[k],k));
 }
```

Mutual Exclusion

- Labels are strictly increasing so
- B must have seen flag[A] == false

Mutual Exclusion

- Labels are strictly increasing so
- B must have seen flag[A] == false
- Labeling_B → read_B(flag[A]) → write_A(flag[A]) → Labeling_A

Mutual Exclusion

- Labels are strictly increasing so
- B must have seen flag[A] == false
- Labeling_B → $\text{read}_B(\text{flag}[A])$ → $\text{write}_A(\text{flag}[A])$ → Labeling_A
- Which contradicts the assumption that A has an earlier label

Bakery Y2^{32}K Bug

```
class Bakery implements Lock {
  …
 public void lock() {
  flag[i]  = true;
  label[i] = max(label[0], …,label[n-1])+1;
  while (∃k flag[k]
           && (label[i],i) > (label[k],k));
 }
```

Bakery Y2^{32}K Bug

```
class Bakery implements Lock {
  …
 public void lock() {
  flag[i]  = true;
  label[i] = max(label[0], …,label[n-1])+1;
  while (∃k flag[k]
           && (label[i],i) > (label[k],k));
 }
```

Mutex breaks if `label[i]` overflows

Lamport’s Fast Lock

- Programs with highly contended locks are likely to not scale
- **Insight: Ideally** spin locks should be free of contention

- Idea
  - Two lock fields x and y
  - Acquire: Thread t writes its id to x and y and checks for intervening writes

Lamport’s Fast Lock

```
class LFL implements Lock {
  private int x, y;
  boolean[] trying;

  LFL() {
    y = ⊥;
    for (int i = 0; i<n; i++) {
      trying[i] = false;
    }
  }
```

```
public void unlock() {
  y = ⊥;
  trying[ThreadID.get()] = false;
}
```

Lamport’s Fast Lock

```
public void lock() {
  int self = ThreadID.get();
  start:
    trying[self] = true;
    x = self;
    if (y != ⊥) {
      trying[self] = false;
      while (y != ⊥) {} // spin
      goto start;
    }
    y = self;
    if (x != self) {
      trying[self] = false;
      for (i ∈ T) {
        while (trying[i] == true) {
          // spin
        }
      }
      if (y != self) {
        while (y != ⊥) {} // spin
        goto start;
      }
    }
}}
```

Evaluation Lock Performance

- Lock acquisition latency – Lock acquire should be cheap in the absence of contenders
- Space overhead – Maintaining lock metadata should not impose high memory overhead
- Fairness – Processors should enter the CS in the order of lock requests
- Traffic – Worst case lock acquire traffic should be low
- Scalability – Latency and traffic should scale slowly with the number of processors

Atomic Instructions in Hardware

Hardware Locks

- Locks can be completely supported by hardware

- Ideas:
  - Have a set of lock lines on the bus, processor wanting the lock asserts the line, others wait, priority circuit used for arbitrating
  - Special lock registers, processors wanting the lock acquire ownership of the registers

- What could be some problems?

Limitations with Hardware Locks

- Waiting logic is critical for the lock performance
  - A thread can (i) busy wait, (ii) block, or (iii) use a hybrid of the earlier two

- Hardware locks are not popularly used
  - Inflexible in implementing wait strategies
  - Limited in number due resource constraints

- We continue to rely on software locks
  - Can be implemented purely in software (classical load-store algorithms)
  - Can optionally make use of hardware instructions for better performance

Common Atomic (RMW) Primitives

test_and_set [x86, SPARC]

```
bool TAS(bool* loc):
  atomic {
    tmp := *loc;
    *loc := true;
    return tmp;
  }
```

swap [x86, SPARC]

```
word Swap(word* a, word b):
  atomic {
    tmp := *a;
    *a := b;
    return tmp;
  }
```

fetch_and_inc [uncommon]

```
int FAI(int* loc):
  atomic {
    tmp := *loc;
    *loc := tmp+1;
    return tmp;
  }
```

fetch_and_add [uncommon]

```
int FAA(int* loc, int n):
  atomic {
    tmp := *loc;
    *loc := tmp+n;
    return tmp;
  }
```

Implement Lock Acquire

swap

```
word Swap(word* a, word b):
  atomic {
    tmp := *a;
    *a := b;
    return tmp;
  }
```

Lock Acquire

```
while (swap(&lock, 1)) {}
```

```
// lock variable

       addi reg, r0, 1 /*r0=0*/
Lock:  xchg reg, &lock
       bnez reg, Lock
```

Common Atomic (RMW) Instructions

compare_and_swap [x86, IA-64, SPARC]

```
bool CAS(word* loc, world old, word new):
  atomic {
    res := (*loc == old);
    if (res)
      *loc := new;
    return res;
  }
```

Common Atomic (RMW) Instructions

compare_and_swap [x86, IA-64, SPARC]

```
bool CAS(word* loc, world old, word new):
  atomic {
    res := (*loc == old);
    if (res)
      *loc := new;
    return res;
  }
```

Lock Acquire

```
// lock variable

        addi reg1, r0, 0x0 /*reg1=0*/
        addi reg2, r0, 0x1 /*reg2=1*/
Lock:   lock compxchgl reg1, reg2, &lock
        bnez reg2, Lock
```

Common Atomic (RMW) Instructions

compare_and_swap [x86, IA-64, SPARC]

```
bool CAS(word* loc, world old, word new):
  atomic {
    res := (*loc == old);
    if (res)
      *loc := new;
    return res;
  }
```

How can you implement fetch_and_func() with CAS?

Common Atomic (RMW) Instructions

load_linked/store_conditional [POWER, MIPS, ARM]

```
word LL(word* a):
  atomic {
    remember a;
    return *a;
  }

bool SC(word* a, word w):
  atomic {
    res := (a is remembered, and has not been evicted since LL)
    if (res)
      *a = w;
    return res;
  }
```

Common Atomic (RMW) Instructions

load_linked/store_conditional [POWER, MIPS, ARM]

```
word LL(word* a):
  atomic {
    remember a;
    return *a;
  }

bool SC(word* a.
  atomic {
    res := (                                        ince LL)
    if (res)
      *a = w;
    return res;
  }
```

How can you implement fetch_and_func() with LL/SC?

Common Atomic (RMW) Instructions

load_linked/store_conditional [POWER, MIPS, ARM]

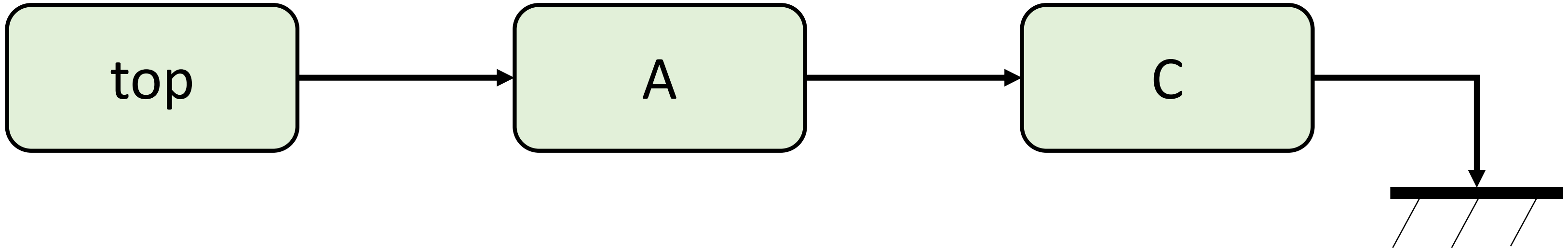
```
word LL(word* a):
  atomic {
    remember a;
    return *a;
  }

bool SC(word* a.
  atomic {
    res := (                                       ince LL)
    if (res)
      *a = w;
    return res;
  }
```

How about CAS vs LL/SC?

List Data Structure

```
void push(node** top, node* new):
  node* old
  repeat
    old := *top
    new->next := old
  until CAS(top, old, new)
```

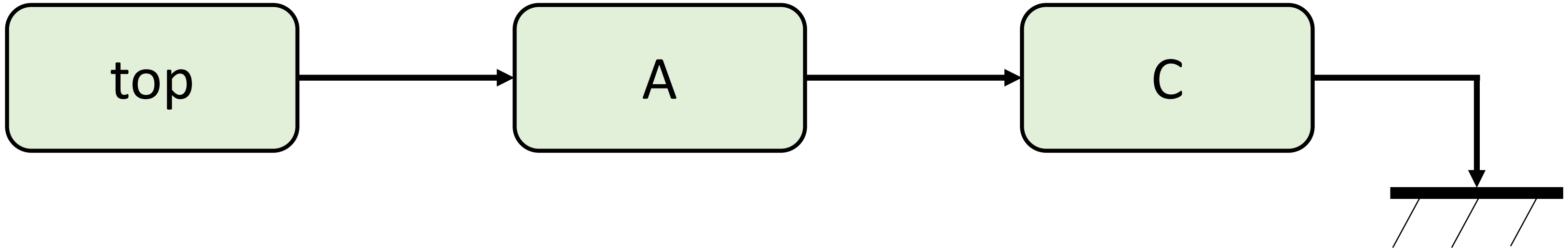
```
node* pop(node** top):
  node* old, new
  repeat
    old := *top
    if old = null return null
    new := old->next
  until CAS(top, old, new)
  return old
```

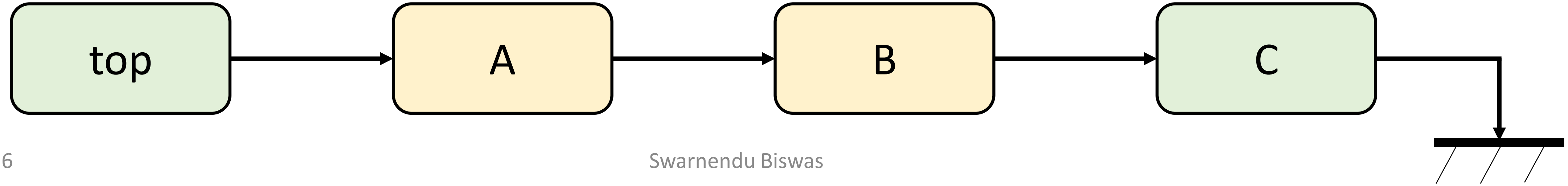


Concurrent Modifications

```
void push(node** top, node* new):
  node* old
  repeat
    old := *top
    new->next := old
  until CAS(top, old, new)
```

```
node* pop(node** top):
  node* old, new
  repeat
    old := *top
    if old = null return null
    new := old->next
  until CAS(top, old, new)
  return old
```

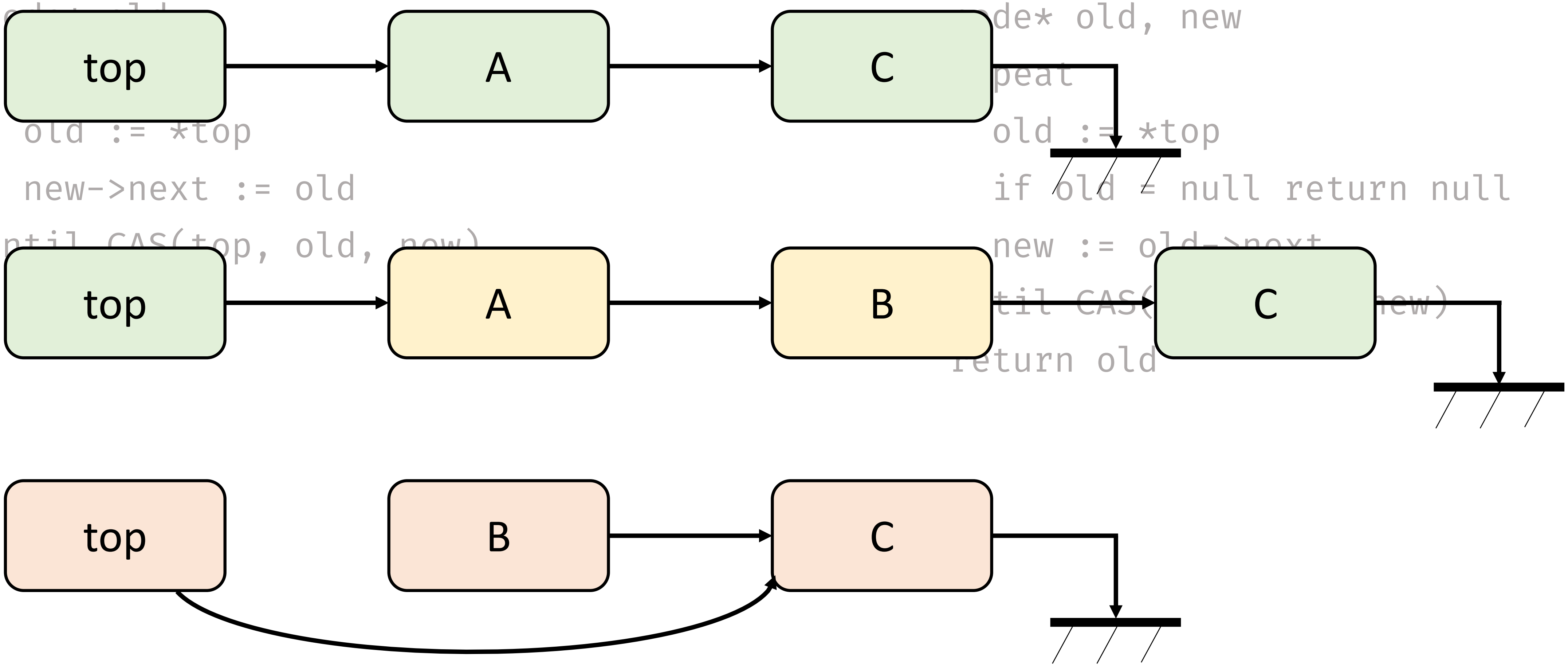




ABA Problem

```
void push(node** top, node* new):
  n
  r
    old := *top
    new->next := old
  until CAS(top, old, new)
```

```
node* pop(node** top):
  de* old, new
  peat
    old := *top
    if old = null return null
    new := old->next
  til CAS(     new)
  return old
```



Common Atomic (RMW) Instructions

compare_and_swap

- Cannot detect ABA

load_linked/store_conditional

- Guaranteed to fail
- SC can experience spurious failures
  - E.g., Cache miss, branch misprediction

Any intervening operation (e.g., bus transaction or cache replacement) to the cache line containing the address in *lock_address* register clears the *load_linked* bit. So, the subsequent SC fails.

Common Atomic (RMW) Instructions

load_linked/store_conditional [POWER, MIPS, ARM]

```
word LL(word* a):
  atomic {
    remember a;
    return *a
  }

bool SC
  atomic
    res                                  een evicted since LL)
    if (res)
      *a = w;
    return res;
  }
```

How can you reduce spurious failures in your fetch_and_func() implementation with LL/SC?

Centralized Mutual Exclusion Algorithms

Test-And-Set

- Atomically tests and sets a word
  - For example, swaps one for zero and returns the old value
- `java.util.concurrent.AtomicBoolean::getAndSet(bool val)`
- Bus traffic?
- Fairness?

```
bool TAS(bool* loc) {
  bool res;
  atomic {
    res = *loc;
    *loc = true;
  }
  return res;
}
```

Spin Lock with TAS

```
class SpinLock {
  bool loc = false;

  public void lock() {
    while (TAS(&loc)) {
      // spin
    }
  }
```

```
  public void unlock() {
     loc = false;
   }

}
```

Spin Lock with TAS

```
class SpinLock {                    public void unlock() {
  bool loc = false;

  public vo
    whil

    }
  }
```

- Delays processors not waiting for the lock
- Lock release can be delayed by spinners
- Does not support reader-writer locking
- No control over locking policy

Test-And-Test-And-Set

- Keep reading the memory location till the location **appears** unlocked
  - Reduces bus traffic – why?

```
do {
  while (TATAS_GET(loc)) {
  }
} while (TAS(loc));
```

Exponential Backoff

Larger number of unsuccessful retries

→ Higher the contention, longer the backoff

- Possibly double each time till a given maximum

Spin Lock with TAS and Backoff

```
class SpinLock {
  bool loc = false;
  const in MIN = …;
  cost int MUL = …;
  const int MAX = …;

  public void unlock() {
    loc = false;
  }
```

```
 public void lock() {
    int backoff = MIN;
    while (TAS(&loc)) {
      pause(backoff);
      backoff = min(backoff * MUL,
                    MAX);
    }
  }
}
```

Challenges with Exponential Backoff

Larger number of unsuccessful retries
→ Higher the contention, longer the backoff

What can be some problems with this?

Challenges with Exponential Backoff

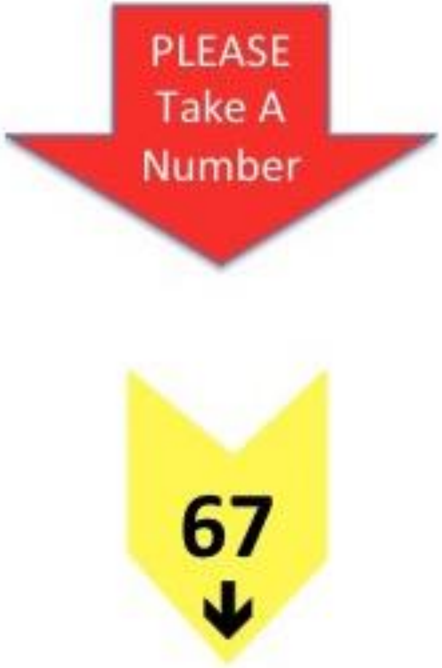
Larger number of unsuccessful retries
→ Higher the contention, longer the backoff

What can be some problems with this?

- Avoid concurrent threads getting into a lockstep, backoff for a random duration, doubling each time till a given maximum
- Critical section is underutilized

Ticket Lock

- Grants access to threads based on FCFS
- Uses `fetch_and_inc()`



Ticket Lock

```
class TicketLock implements Lock
{

  int next_ticket = 0;
  int now_serving = 0;

  public void unlock() {
    now_serving++;
  }
```

```
  public void lock() {
    int my_ticket = FAI(&next_ticket);
    while (now_serving != my_ticket) {}
  }

}
```

How is this different from Bakery's algorithm?

Ticket Lock

```
class TicketLock implements Lock
{

  int next_ticket = 0;
  int now_serving = 0;

  public void unlock() {
    now_serving++;
  }
```

```
  public void lock() {
    int my_ticket = FAI(&next_ticket);
    while (now_serving != my_ticket) {}
  }

}
```

What are some disadvantages of Ticket locks?

Scalable Spin Locks

Queued Locks

- Key idea
  - Instead of contending on a single “`now_serving`” variable, make threads wait in a queue (i.e., FCFS)
  - Each thread knows its order in the queue

Implementations

- Implement a queue using arrays
  - Statically or dynamically allocated depending on the number of threads
- Each thread spins on its **own lock** (i.e., array element), and knows the successor information

Queued Lock

```
public class ArrayLock implements
Lock {
  AtomicInteger tail;
  volatile boolean[] flag;
  ThreadLocal<Integer> mySlot = …;

  public ArrayLock(int size) {
    tail = new AtomicInteger(0);
    flag = new boolean[size];
    flag[0] = true;
  }
```

```
  public void lock() {
    int slot = FAI(tail);
    mySlot.set(slot);
    while (!flag[slot]) {}
  }

  public void unlock() {
    int slot = mySlot.get();
    flag[slot] = false;
    flag[slot+1] = true;
  }
}
```

Queued Lo

What could be a few disadvantages of array-based Queue locks?

```
public class ArrayLock impleme
Lock {
  AtomicInteger tail;
  volatile boolean[] flag;
  ThreadLocal<Integer> mySlot = …;

  public ArrayLock(int size) {
    tail = new AtomicInteger(0);
    flag = new boolean[size];
    flag[0] = true;
  }
```

```
  public void lock() {
    int slot = FAI(tail);
    mySlot.set(slot);
    while (!flag[slot]) {}
  }

  public void unlock() {
    int slot = mySlot.get();
    flag[slot] = false;
    flag[slot+1] = true;
  }
}
```

Queued Locks using Array

Can we come up with better ideas?

```
public class ArrayLock implements
Lock {
  AtomicInteger tail;
  boolean[] flag;
  ThreadLocal<Integer> mySlot = …;

  public ArrayLock(int size) {
    tail = new AtomicInteger(0);
    flag = new boolean[size];
    flag[0] = true;
  }
```

space overhead is O(nk)

```
  public void lock() {
    int slot = FAI(tail);
    mySlot.set(slot);
    while (!flag[slot]) {}
  }

  public void unlock() {
    int slot = mySlot.get();
    flag[slot] = false;
    flag[slot+1] = true;
  }
}
```

false sharing

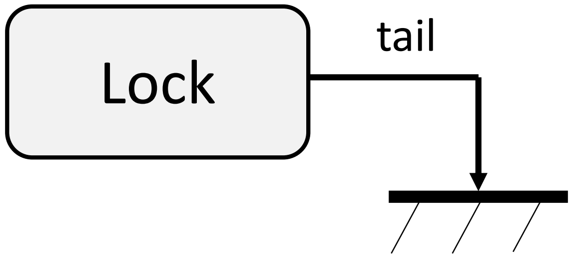
MCS Queue Lock

- Proposed by Mellor-Crumney and Scott [1991]
- Uses linked lists instead of arrays
- Space required to support n threads and k locks: **O(n+k)**
- **State-of-art scalable FIFO locks**

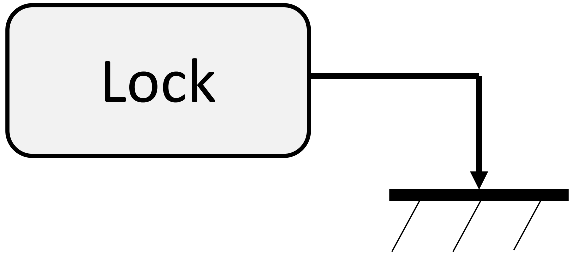
MCS Queue Lock

```
class QNode {
  QNode next;
  bool waiting;
}
public class MCSLock implements Lock {
  Node tail = null;
  ThreadLocal<QNode> myNode = …;

  public void lock() {
    QNode node = myNode.get();
    QNode prev = swap(tail, node);
    if (prev != null)
      node.waiting = true;
      prev.next = node;
      while (node.waiting) {}
  }
  public void unlock() {
    QNode node = myNode.get();
    QNode succ = node.next;
    if (succ == null)
      if (CAS(tail, node, null))
        return;
      do {
        succ = node.next;
      } while (succ == null);
    succ.waiting = false;
  }
}
```

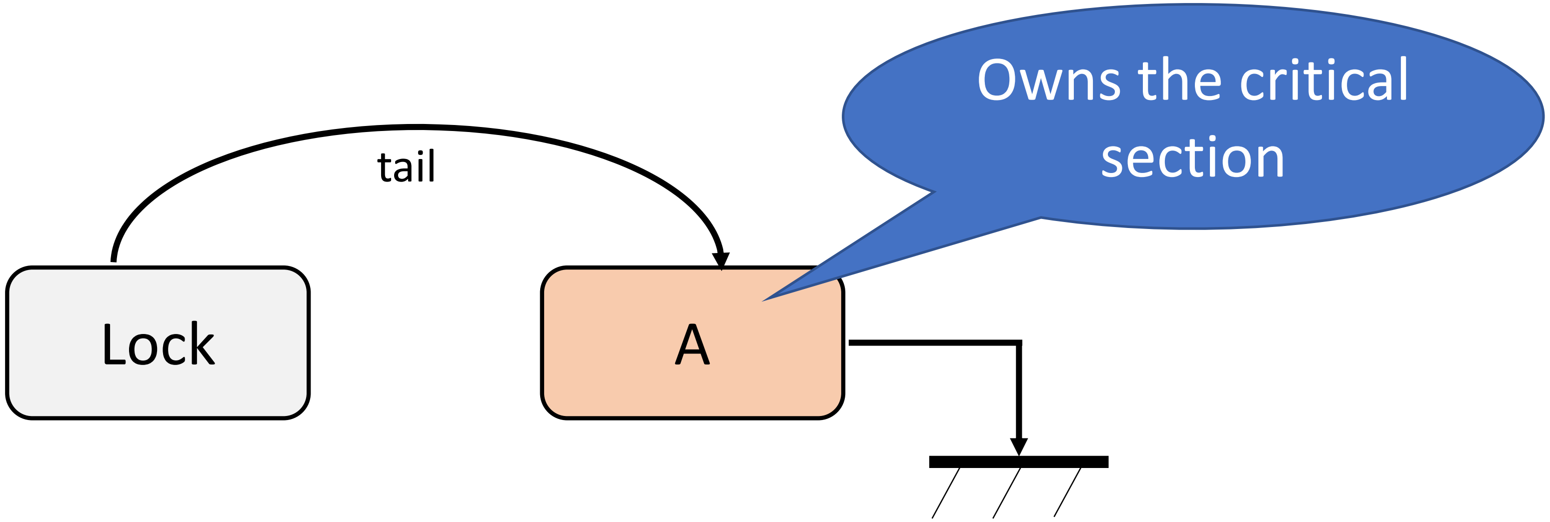
MCS Lock Operations



MCS Lock Operations



Lock



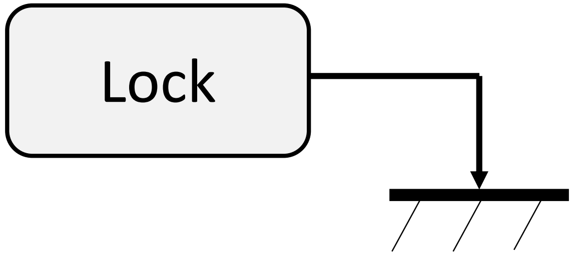
tail

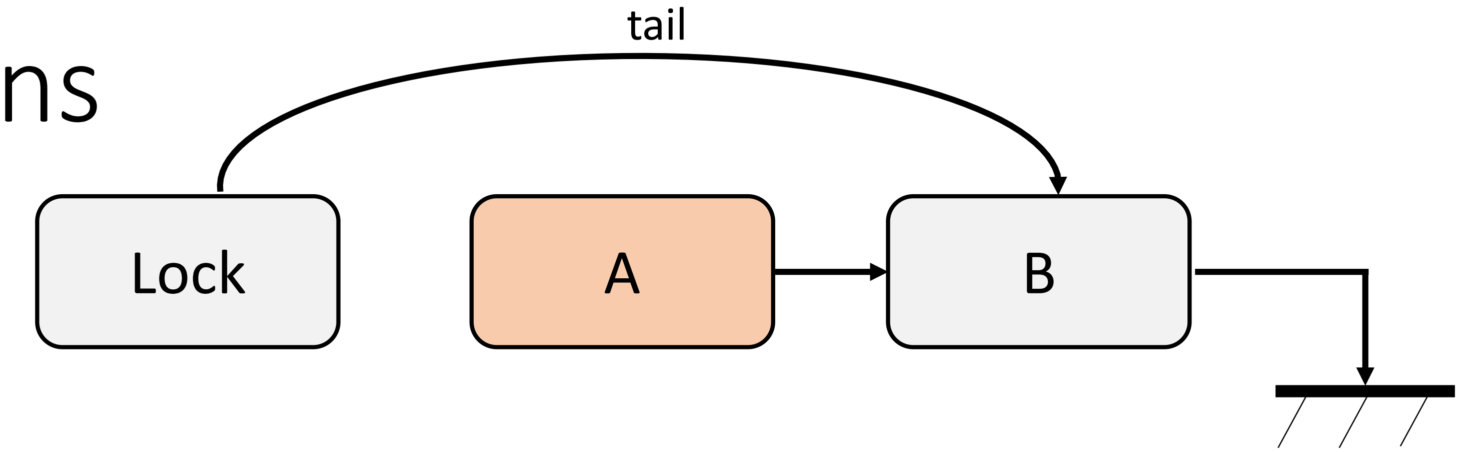
Lock

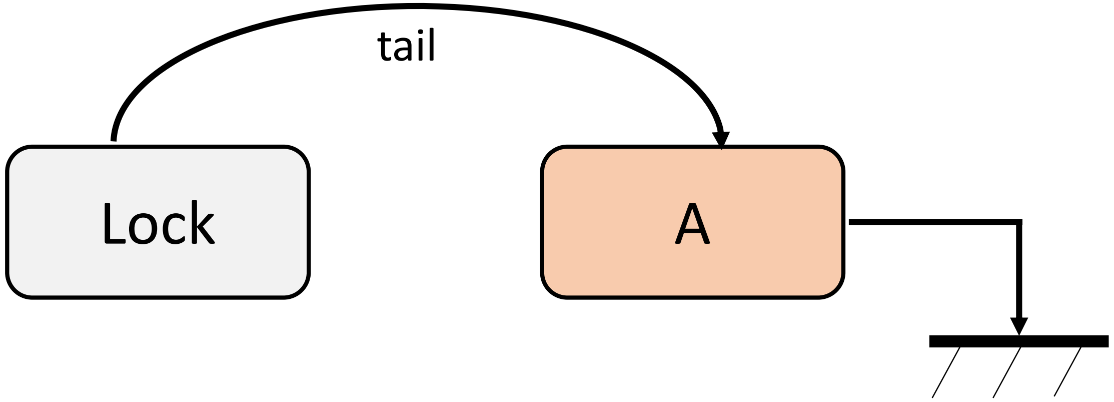
A

Owns the critical section

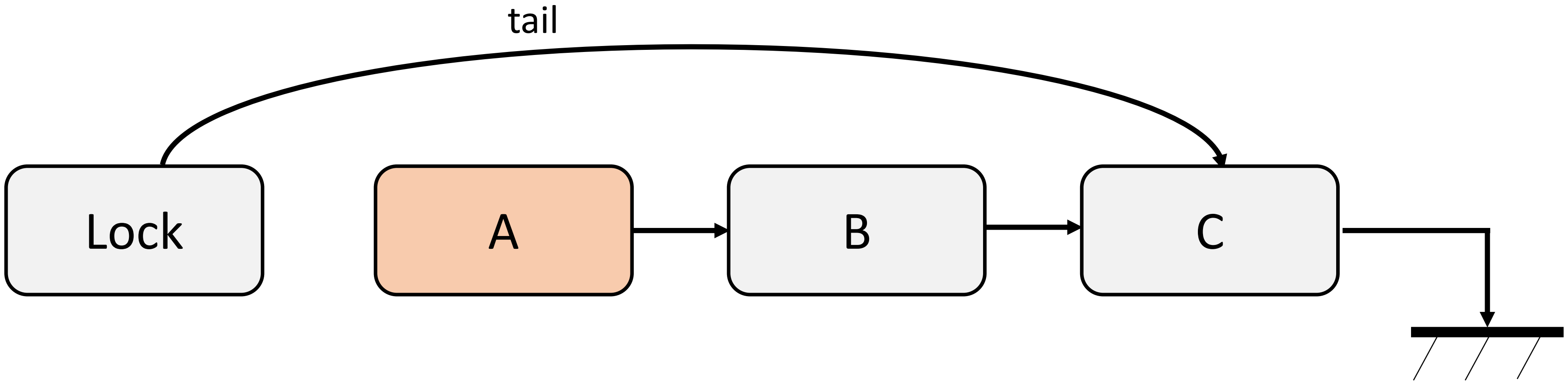
MCS Lock Operations



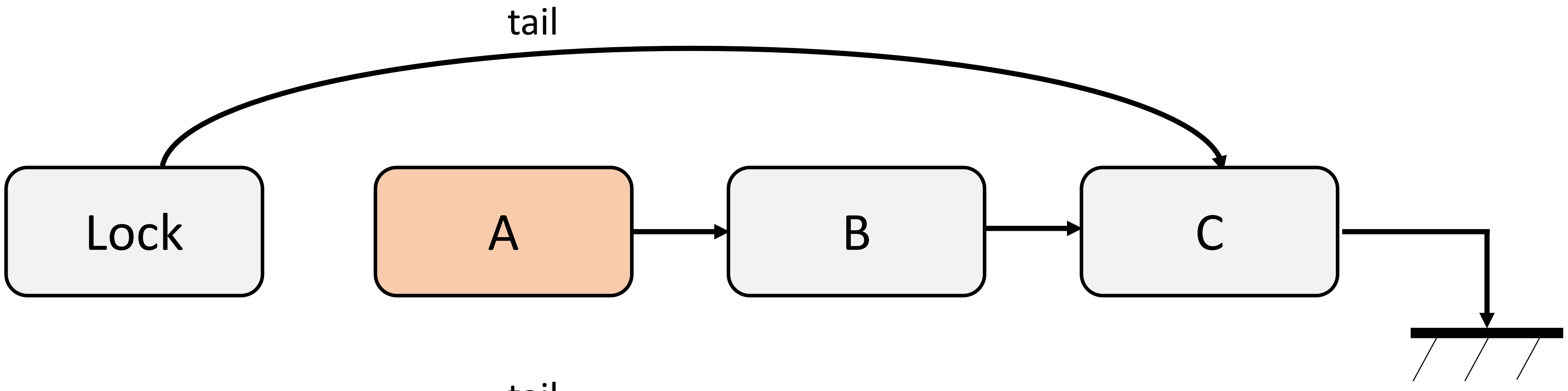


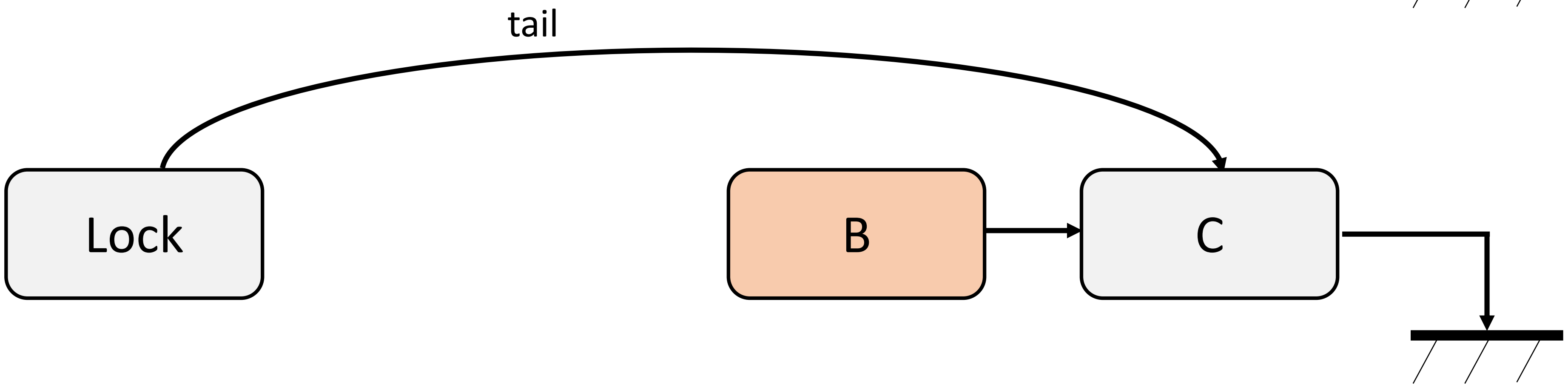


MCS Lock Operations



MCS Lock Operations





Properties of the MCS Lock

- Threads joining the wait queue is wait-free
  - Wait-freedom implies every operation has a bound on the number of steps it will take before the operation completes
  - Wait-freedom is the strongest non-blocking guarantee of progress
- Thread acquire locks in FIFO manner
- Minimizes false sharing and resource contention

Which Spin Lock should I use?

- Limited use of load-store-only locks
- Limited contention (e.g., few threads)
  - TAS spin locks with exponential backoff
  - Ticket locks
- High contention
  - MCS lock or other proposals like CLH lock

Miscellaneous Lock Optimizations

Reentrant Locks

- A lock that can be **re-acquired** by the owner thread
- Freed after an equal number of releases

```
public class ParentWidget {

  public synchronized void doWork() {
    …
  }
}
```

```
public class ChildWidget extends ParentWidget {

  public synchronized void doWork() {
    …
    super.doWork();
    …
  }
}
```

Lazy Initialization In Single-Threaded Context

```
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      helper = new Helper();
    }
    return helper;
  }
  …
}
```

Correct for single thread

What could go wrong with multiple threads?

Lazy initialization

---

http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

Lazy Initialization In Multithreaded Context

```
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      helper = new Helper();
    }
    retur
  }
  …
}
```

```
class Foo {
  private Helper helper = null;
  public synchronized Helper getHelper() {
    if (helper == null) {
      helper = new Helper();
    }
    return helper;
  }
  …
}
```

Synchronizes even after helper has been allocated

---

http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

Double-Checked Locking

- Can we optimize the initialization pattern?
  1. `Check if helper is initialized`
     - `If yes, return`
     - `If no, then obtain a lock`
  2. `Double check whether the helper has been initialized`
     - `Perhaps concurrently initialized in between Steps 1 and 2`
  3. `If yes, return`
  4. `Initialize helper, and return`

```
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      synchronized (this) {
        if (helper == null)
          helper = new Helper();
      }
    }
    return helper;
  }
  …
}
```

Broken Usage of Double Checked Locking

```
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      synchronized (this) {
        if (helper == null)
          helper = new Helper();
      }
    }
    return helper;
  }
  …
}
```

Not platform-independent when implemented in Java

Double Checked Locking: Broken Fix

```
private Helper helper = null;
public Helper getHelper() {
  if (helper == null) {
    Helper h;
    synchronized (this) {
      h = helper;
      if (h == null) {
        synchronized (this) {
          h = new Helper();
       } }
      helper = h;
    } }
  return helper;
}
```

- A release operation prevents operations from moving out of the critical section
- It does not prevent `helper = h` from being moved up

One Correct Use of Double Checked Locking

```
class Foo {
  private volatile Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      synchronized (this) {
        if (helper == null)
          helper = new Helper();
      }
    }
    return helper;
  }
  …
}
```

- Other options are to use barriers in both the writer thread (the thread that initializes `helper`) and all reader threads

Readers-Writer Locks

- Many objects are read concurrently
  - Updated only a few times

- Reader lock
  - No thread holds the write lock
- Writer lock
  - No thread holds the reader or writer locks

```
public interface RWLock {
  public void readerLock();
  public void readerUnlock();

  public void writerLock();
  public void writerUnlock();
}
```

Issues to Consider in Readers-Writer Locks

Design choices

Release preference order	Writer releases lock, both readers and writers are queued up
Incoming readers	Writers waiting, and new readers are arriving
Downgrading	Can a thread acquire a read lock without releasing the write lock?
Upgrading	Can a read lock be **upgraded** to a write lock?

Readers-Writer Locks

- Reader or writer preference
  - Impacts degree of concurrency
  - Allows starvation of non-preferred threads

```
readerLock():
  acquire(rd)
  rdrs++
  if rdrs == 1:
    acquire(wr)
  release(rd)
```

```
readerUnlock():
  acquire(rd)
  rdrs--
  if rdrs == 0:
    release(wr)
  release(rd)

writerLock():
  acquire(wr)

writerUnlock():
  release(wr)
```

Readers-Writer Lock With Reader-Preference

```
class RWLock {
  int n = 0;
  const int WR_MASK = 1;
  const int RD_INC = 2;

  public void writerLock() {
    while (¬ CAS(&n, 0, WR_MASK)) {
    }
  }
}
```

```
public void writerUnlock() {
  FAA(&n, -WR_MASK);
}

public void readerLock() {
  FAA(&n, RD_INC);
  while ((n & WR_MASK) == 1) {
  }
}

public void readerUnlock() {
  FAA(&n, -RD_INC);
}
```

Asymmetric Locks

- Often objects are locked by at most one thread
- Biased locks
  - JVMs use biased locks, the acquire/release operations on the owner threads are cheaper
    - Usually biased to the first owner thread
  - Synchronize only when the lock is contended, need to take care of several subtle issues
  - `-XX:+UseBiasedLocking` in HotSpot JVM

---

https://blogs.oracle.com/dave/biased-locking-in-hotspot

Lock Implementations in a JVM

- All objects in Java are potential locks
  - Recursive lock – lock can be acquired multiple times by the owner
  - Thin lock
    - spin lock used when there is no contention, inflated to a fat lock on contention
  - Fat lock
    - lock is contended or is waited upon, maintains a list of contending threads

Monitors

Using Locks to Access a Bounded Queue

- Consider a **bounded** FIFO queue
- Many producer threads and one consumer thread access the queue

```
mutex.lock();
try {
  queue.enq(x);
} finally {
  mutex.unlock();
}
```

What are possible problems?

Using Locks to Access a Bounded Queue

- Consider a **bounded** FIFO queue
- Many producer threads and one
  cons
  que

```
mutex.lock();
try {
```

- Producers/Consumers need to know about the size of the queue
- The design may evolve, there can be multiple queues, along with new producers/consumers
- Every producer/consumer need to follow the locking convention

Monitors to the Rescue!

- Combination of methods, mutual exclusion locks and condition variables

```
public synchronized void enque() {
  queue.enq(x);
}
```

- Provides **mutual exclusion for methods**
- Provides the possibility to **wait for a condition (cooperation)**

Condition Variables in Monitors

- Have an associated queue
- Operations
  - **wait**
  - **notify** (signal)
  - **notifyAll** (broadcast)

Condition Variable Operations

wait var, mutex

- Make the thread wait until a condition *COND* is true
  - Releases the **monior’s mutex**
  - Moves the thread to var’s wait queue
  - Puts the thread to sleep
- **Steps 1-3 are atomic to prevent race conditions**
- When the thread wakes up, it is assumed to hold mutex

Condition Variable Operations

notify var

- Invoked by a thread to assert that *COND* is true
- Moves one or more threads from the wait queue to the ready queue

notifyAll var

- Moves all threads from wait queue to the ready queue

Signaling Policies

Signal and continue (SC)	Signaler thread holds the lock Java implements SC only
Signal and wait (SW)	Signaler thread needs to reacquire the lock, signaled thread can continue execution
Signal and urgent wait (SU)	Like SW, but signaler thread gets to go after the signaled thread
Signal and exit (SX)	Signaler exits, signaled thread can continue execution

Using Monitors

- Have an associated queue
- Operations
  - **wait**
  - **notify** (signal)
  - **notifyAll** (broadcast)

```
acquire(mutex)
while (!COND) {
  wait(var, mutex)
}
…
/* CRITICAL SECTION */
…
notify(var)/notifyAll(var)
release(mutex)
```

Producer-Consumer with Monitors

```
Queue q;
Mutex mtx; // Has associated queue
CondVar empty, full;

producer:
  while true:
    data = new Data(…);
    acquire(mtx);
    while q.isFull():
      wait(full, mtx);
    q.enq(data);
    notify(empty);
    release(mtx);
```

```
consumer:
  while true:
    acquire(mtx)
    while q.isEmpty():
      wait(empty, mtx);
    data = q.deq();
    notify(full);
    release(mtx);
    …
    …
```

Contrast with Producer-Consumer with Spin Locks

```
Queue q;
Mutex mtx;

producer:
  while true:
    data = new Data(…);
    acquire(mtx);
    while q.isFull():
      release(mtx);
      …
      acquire(mtx);
    q.enq(data);
    release(mtx);
```

```
consumer:
  while true:
    acquire(mtx);
    while q.isEmpty():
      release(mtx);
      …
      acquire(mtx);
    data = q.deq();
    release(mtx);
    …
    …
```

Semaphore Implementation with Monitors

```
int numRes = N;
Mutex mtx;
CondVar zero;

P:
  acquire(mtx);
  while numRes == 0:
    wait(zero, mtx);
  assert numRes > 0
  numRes--;
  release(mtx);
```

```
V:
  acquire(mtx);
  numRes++;
  notify(zero);
  release(mtx);
```

Reader-Writer Locks with Reader-Preference

```
readerLock():
  acquire(rd)
  rdrs++
  if rdrs == 1:
    acquire(wr)
  release(rd)

readerUnlock():
  acquire(rd)
  rdrs--
  if rdrs == 0:
    release(wr)
  release(rd)
```

```
writerLock():
  acquire(wr)

writerUnlock():
  release(wr)
```

How can we construct a Reader-Writer lock with writer-preference?

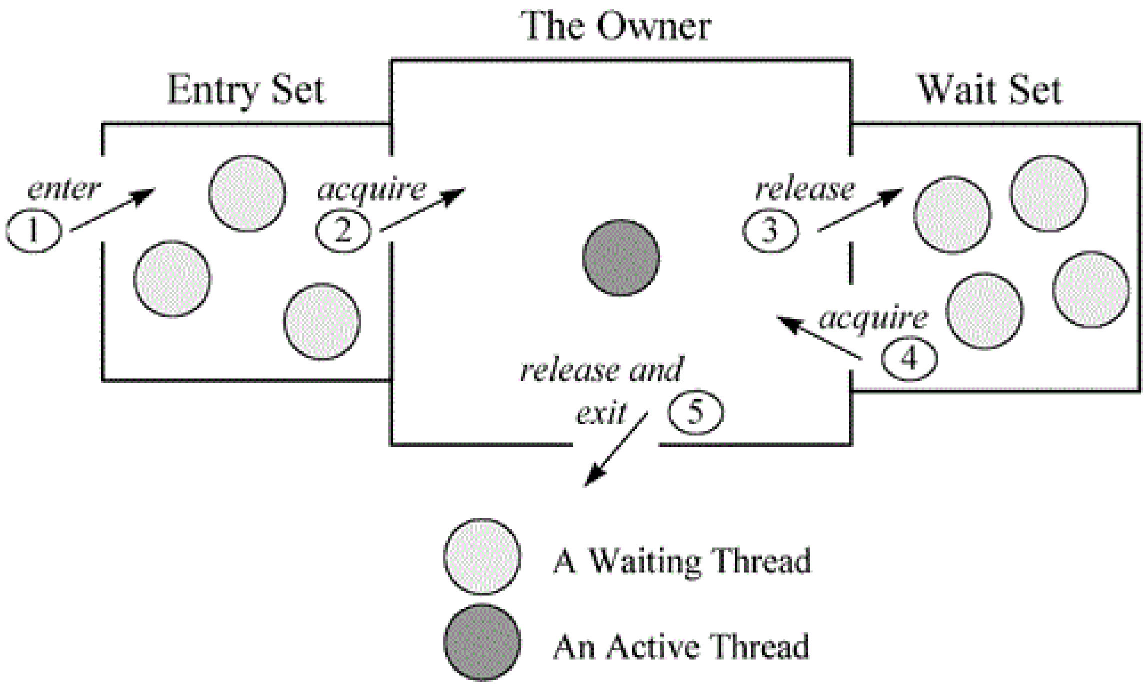
Reader-Writer Lock With Writer-Preference

```
readerLock():
  acquire(global)
  while writerFlag:
    wait(writerWait, global)
  rdrs++
  release(global)

readerUnlock():
  acquire(global)
  rdrs--
  if rdrs == 0:
    notifyAll(writerWait)
  release(global)
```

```
writerLock():
  acquire(global)
  while writerFlag:
    wait(writerWait, global)
  writerFlag = true
  while rdrs > 0:
    wait(writerWait, global)
  release(global)

writerUnlock():
  acquire(global)
  writerFlag = false
  notifyAll(writerWait)
  release(global)
```

Monitors in Java

- Java provides built-in support for monitors
  - **synchronized** blocks and methods
  - wait(), notify(), and notifyAll()
- Each object can be used as a monitor



---

https://www.artima.com/insidejvm/ed2/threadsynch.html

Bounded Buffer with Monitors in Java

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class BoundedBuffer {
  private final String[] buffer;
  private final int capacity; // Constant, length of buffer
  private int count; // Current size
  private final Lock lock = new ReentrantLock();
  private final Condition full = new Condition();
  private final Condition empty = new Condition();
```

Bounded Buffer with Monitors in Java

```
public void addToBuffer() … {
  lock.lock();
  try {
    while (count == capacity)
      full.await();
    …
    …
    empty.signal();
  } finally {
    lock.unlock();
  }
}
```

```
public void removeFromBuffer() … {
  lock.lock();
  try {
    while (count == 0)
      empty.await();
    …
    …
    full.signal();
  } finally {
    lock.unlock();
  }
 }
}
```

References

- Michael Scott. Shared Memory Synchronization. Morgan and Claypool Publishers.
- M. Herlihy and N. Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers.
- B. Goetz et al. Java Concurrency in Practice. Pearson.