CS 636: Shared-Memory Synchronization

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2025-26-II

Is the Code Thread-Safe?

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

  void add(Object x) { // Free of data races
    if (!elems.contains(x))
      elems.add(x);
  }
}

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

What is the Desired Property?

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

  void add(Object x) { // Free of data races
    if (!elems.contains(x))
      elems.add(x);
  }
}

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

atomic

Synchronization Patterns

Mutual exclusion — updates need to be serialized

```
bool lock = false;
```

```
lock_acquire():
  while TAS(&lock)
    // spin
```

```
lock_release():
  lock = false;
```

Conditional synchronization — events need to occur in a specified order

```
while !condition
  // spin
```

Other forms — e.g., synchronize across threads or control the number of simultaneous accesses to a shared resource

Desired Synchronization Properties

Mutual exclusion (safety property)

- Critical sections on the same lock from different threads do not overlap

Deadlock freedom (liveness property)

- If some threads attempt to acquire the lock, then some thread should be able to acquire the lock
- Individual threads may be infinitely delayed

Starvation freedom (liveness property)

- 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?

```java
class LockOne implements Lock {
  private boolean[] flag = new boolean[2];
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    j = 1-i;
    while (flag[j]) {}
  }
  public void unlock() {
    int i = ThreadID.get();
    flag[i] = false;
  }
}
```

- LockOne satisfies mutual exclusion
- LockOne fails deadlock-freedom, concurrent execution can deadlock

LockTwo: What could go wrong?

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

  public void unlock() {}
}
```

- LockTwo satisfies mutual exclusion
- LockTwo fails deadlock-freedom, sequential execution deadlocks

Peterson’s Algorithm

```java
class PetersonLock {
  private boolean[] flag = new boolean[2];
  private int victim;

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

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

Peterson’s Algorithm

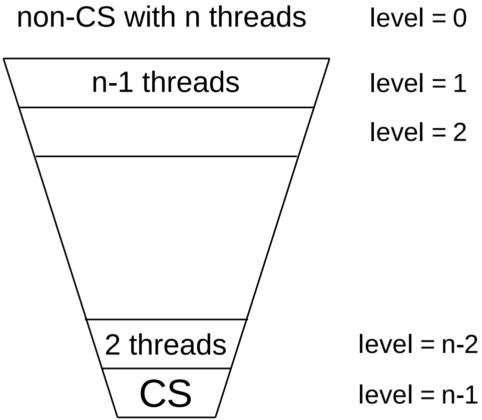
```
class PetersonLock {
  private boolean[] flag = new boolean[2];
  private int victim;

  public void lock() {
    int i = ThreadID.get();



  }

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

- Does this algorithm satisfy mutual exclusion under sequential consistency?
- What if we do not have sequential consistency?

Filter Lock for n Threads

Filter lock is a generalization of Peterson's lock to $n > 2$ 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 {
  int[] level;
  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;
  }

  public void lock() {
    int me = ThreadID.get();
    // Attempt to enter level i
    for (int i=1; i<n; i++) {
      // visit level i
      level[me] = i;
      victim[i] = me;
      // spin while conflict exists
      while ((∃k != me)
        level[k] >= i && victim[i] ==
            me) {}
    }
  }
}
```

Fairness

Starvation freedom is good, but maybe threads should not 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 the `lock()` method into two parts

  Doorway interval (DA) expresses intent to synchronize, finishes in finite steps

  Waiting interval (WA) wait for turn to synchronize, 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 $D_A^j \rightarrow D_B^k$, then $CS_A^j \rightarrow CS_B^{k+r}$

Lamport's Bakery Algorithm

```
class Bakery implements Lock {
  boolean[] choosing;
  Label[] lbl;
  public Bakery(int n) {
    choosing = new boolean[n];
    lbl = new Label[n];
    for (int i = 0; i<n; i++) {
      choosing[i] = false;
      lbl[i] = 0;
    }
  }

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

Lamport's Bakery Algorithm

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

(lbl[i], i) << (lbl[j], j)) iff

- lbl[i] < lbl[j], or
- lbl[i] = lbl[j] and i < j

Lamport's Bakery Algorithm

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

— Need to compare own label with all other threads' labels irrespective of their intent to enter the critical section

— Cost of locking increases with the number of threads

(lbl[i], i) << (lbl[j], j)) iff

- lbl[i] < lbl[j], or
- lbl[i] = lbl[j] and i < j

Lamport's Fast Lock

- Programs with highly contended locks are likely to not scale
- **Insight**: Ideally spin locks should be free of contention in well-designed systems, so optimize for the common case
- Idea:
  - Use two lock fields `fast_check` and `slow_check`
  - Acquire: Thread `t` writes its ID to `fast_check` and `slow_check` and checks for intervening writes

L. Lamport. A Fast Mutual Exclusion Algorithm. TOCS, 1987.

Fast Mutual Exclusion, Even With Contention

Lamport's Fast Lock

```
class LFL implements Lock {
  // Two checkpoints to guarantee mutual exclusion
  private int fast_check, slow_check;
  boolean[] trying;
  LFL() {
    slow_check = ⊥;
    for (int i = 0; i<n; i++)
      trying[i] = false;
  }
  public void unlock() {
    slow_check = ⊥;
    trying[ThreadID.get()] = false;
  }
}
```

Lamport's Fast Lock

```
public void lock() {
  int self = ThreadID.get();
start:
  trying[self] = true;
  fast_check = self;
  if (slow_check != ⊥) {
    // Someone else is in the CS
    trying[self] = false;
    while (slow_check != ⊥) {}
    goto start; // Retry
  }
  slow_check = self;
```

```
  // Ensure atomicity
  if (fast_check != self) {
    trying[self] = false;
    for (i ∈ T) {
      while (trying[i] == true) {}
    }
    if (slow_check != self) {
      while (slow_check != ⊥) {}
      goto start;
    }
  }
}
```

Evaluating Performance of a Lock

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

J. Preshing. Locks Aren't Slow; Lock Contention Is.

Practicality of Classical Mutual Exclusion Algorithms

A write (i.e., regular memory store) by a thread to a memory location can be overwritten without any other thread seeing the first write

Need to read and write n distinct memory locations where n is the maximum number of concurrent threads

- n is a lower bound on the number of required locations

- Motivates the need for read-write operations with stronger guarantees

Atomic Hardware Instructions

Hardware Locks

- Locks can be completely supported by hardware
- Ideas:
  1. (i) Have a set of lock lines on the bus, processor wanting the lock asserts the line, others wait, priority circuit used for arbitrating
  2. (ii) 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 lock performance
  - A thread can (i) busy wait (i.e., spin), (ii) block, or (iii) use a hybrid strategy (e.g., busy wait for some time and then block)
- Hardware locks are not popularly used
  - Limited in number due resource constraints
  - Inflexible in implementing wait strategies

We continue to rely on software locks

Can optionally make use of hardware instructions for better performance

Common Atomic Primitives

Modern architectures provide many atomic read-modify-write (RMW) instructions for synchronization

- For example, test-and-set, fetch-and-add, compare-and-swap, and load-linked/store-conditional

TAS X86, SPARC

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

swap X86, SPARC

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

Spin Lock with TAS on X86_64

```c
static inline uint8_t tas(volatile uint8_t *p) {
  uint8_t old = 1;
  asm volatile(
      "xchg %0, %1" // atomic xchg with mem
      : "+q"(old), "+m"(*p) // old is both input/output; *p is read/write
      : // no extra inputs
      : "memory" // compiler barrier for reordering
  );
  return old;
}

static volatile uint8_t lock = 0;

static inline void acquire() {
  while (tas(&lock)) { } // Add delay to reduce contention
}

static inline void release() {
  asm volatile("" ::: "memory");
  lock = 0;
}
```

Spin Lock with TAS on X86_64

```cpp
#include <atomic>

std::atomic_flag lock = ATOMIC_FLAG_INIT;

// Using sequential consistency as the default ordering

// Spin until the lock is acquired
while (std::atomic_flag_test_and_set(&lock)) {}

std::atomic_flag_clear(&lock);
```

https://en.cppreference.com/w/cpp/atomic/atomic_flag_test_and_set.html

Common Atomic Primitives

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;
```

C++ 11 onward provides `std::atomic<T>::fetch_add()`

Common Atomic Primitives

fetch_and_inc uncommon

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

fetch_and_add uncommon

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

How can we implement a mutual exclusion lock with FAI?

C++ 11 onward provides `std::atomic<T>::fetch_add()`

Compare-and-Swap (CAS) Primitive

Compare-and-Swap (CAS) compares the contents of a memory location with a given value and, only if they are the same, updates the contents of that memory location to a new given value

```
bool CAS(word* loc, word oldval, word newval) {
  atomic { // Code block will execute atomically
    res := (*loc == oldval);
    if (res)
      *loc := newval;
  }
  return res;
}
```

Compare-and-Swap (CAS) Primitive

- CAS is implemented as the compare-and-exchange (CMPXCHG) instruction in x86 architectures
  - On a multiprocessor, the LOCK prefix must be used

- CAS is a popular synchronization primitive for implementing both lock-based and nonblocking concurrent data structures

```
        xor %ecx, %ecx ; ecx=0
        inc %ecx ; ecx=1
RETRY:  xor %eax, %eax ; eax=0
        lock compxchg %ecx, &lk
        jnz RETRY
        ret
```

```
void spinLock(lock* lk) {
  // flg attribute is set when
  // the lock is acquired
  while (CAS(&lk->flg,0,1)==1) {
    // Keep spinning
  }
}
```

Compare-and-Swap (CAS) Primitive

- CAS is implemented as the compare-and-exchange (CMPXCHG) instruction in x86 architectures
  - On a multiprocessor, the LOCK prefix must be used
- CAS is a popular synchronization primitive for implementing both lock-based and nonblocking concurrent data structures

```
        xor %ecx, %ecx ; ecx=0
        inc %ecx ; ecx=1
RETRY:  xor %eax, %eax ; eax=0
        lock compxchg %ecx, &lk
        jnz RETRY
        ret
```

```
void spinLock(lock* lk) {
  // flg attribute is set when
  // the lock is acquired
  while (CAS(&lk->flg,0,1)==1) {
    // Keep spinning
  }
}
```

How can you implement fetch_and_xyz() with CAS?

Spin Lock with CAS on X86_64

```c
static inline bool cas32(volatile uint32_t *addr, uint32_t oldVal, uint32_t newVal) {
  unsigned char result;
  asm volatile(
      "lock cmpxchgl %3, %0 \n sete %1"
      : "+m"(*addr), "=q"(result), "+a"(oldVal) // loads oldVal into EAX
      : "r"(newVal)
      : "memory");
  return result;
}

uint32_t lock = 0;

void acquire() {
  while (!cas32(&lock,0,1)) { } // add delay to reduce contention
}

static inline void release() {
  asm volatile("" ::: "memory"); // compiler barrier
  lock = 0;
}
```

Load Linked (LL)/Store Conditional (SC) Instructions

LL/SC — 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;
  }
```

Load Linked (LL)/Store Conditional (SC) Instructions

LL/SC — 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

    if (res
      *a = w;
    return res;
  }
```

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

ABA Problem

Nonblocking Algorithms

Blocking algorithms (e.g., lock-based concurrent data structures)

Failure or delay of any one thread can delay other threads

- Use of locks can lead to deadlocks, livelocks, and priority inversion
- Blocked threads do not do useful work, problematic for high-priority or real-time applications
- Getting the right degree of concurrency and correctness with locks is challenging

Failure or delay of one thread cannot delay other threads in nonblocking algorithms

- Provides different progress guarantees: wait-freedom and lock-freedom
- Use RMW instructions like CAS or LL/SC for mutual exclusion
- Eliminate locks altogether

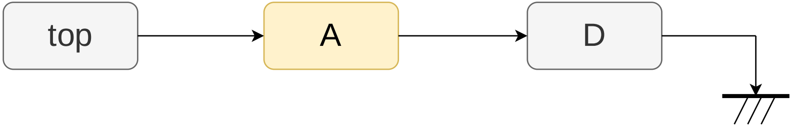
Lock-free Stack Data Structure

push

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

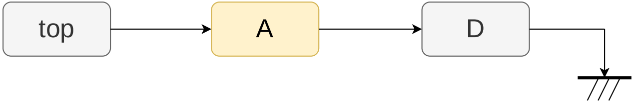
pop

```
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 to a Lock-free Stack

Thread 1 is executing `pop(A)`

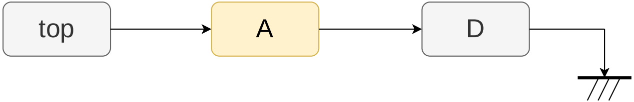


Thread 1 sees top points to A, but gets delayed while executing `pop(A)`

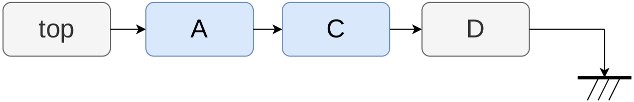
Assume that deleted nodes can be reused

Concurrent Modifications to a Lock-free Stack

1 Thread 1 is executing `pop(A)`

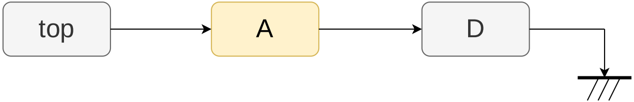


2 Other threads execute `pop(A)`, `push(C)`, and `push(A)`

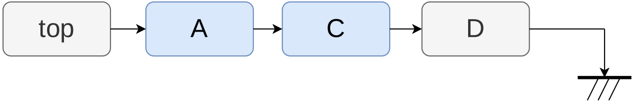


ABA Problem

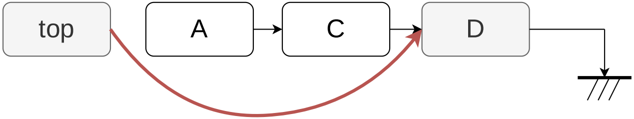
1 Thread 1 is executing pop(A)



2 Other threads execute pop(A), push(C), and push(A)



3 Thread 1's CAS succeeds



Avoiding ABA Problem

Common workaround is to add extra "tag" to the memory address being compared

- Tag can be a counter that tracks the number of updates to the reference
- Can steal lower order bits of memory address or use a separate tag field if 128-bit CAS is available

LL/SC does not suffer from the ABA problem

It checks whether a value has changed in between the interval, rather than comparing the value itself

Scalable Spin Locks

Spin Lock with TAS

```
class SpinLock {
  bool loc = false;
  public void lock() {
    while (TAS(&loc)) {
      // spin
    }
  }
  public void unlock() {
    loc = false;
  }
}
```

How can we improve the performance of TAS-based spinlocks?

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));
```

With n threads contending for a critical section, the time per acquire-release pair is $\mathcal{O}(n)$

Spin Lock with TAS and Exponential Backoff

Adapt when to retry to reduce contention

- For example, increase the backoff with the number of unsuccessful retries (implies high contention)

```
class SpinLock {
  bool loc = false;
  const int MIN = ..., MUL = ..., 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

- Adapt when to retry to reduce contention
  - For example, increase the backoff with the number of unsuccessful retries (implies high contention)

What can be some problems?

Challenges with Exponential Backoff

- Adapt when to retry to reduce contention
  - For example, increase the backoff with the number of unsuccessful retries (implies high contention)

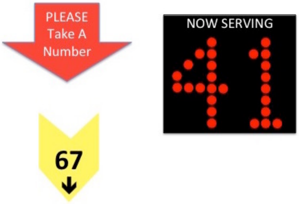
— Critical section is potentially underutilized

— Avoid concurrent threads getting into a lockstep, backoff for a random duration, possibly doubling each time till a given maximum

— Best-performing constants depend on the host machine and the application

Ticket Lock

- TAS-based locks are unfair
- Ticket lock grants access to threads based on FCFS

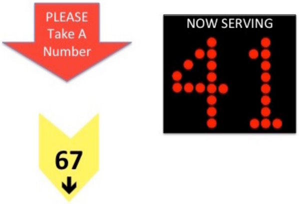



```
class TicketLock implements Lock {
  int nxt_tkt = 0;
  int serving = 0;
  public void unlock() {
    serving++;
  }
  public void lock() {
    int my_tkt = FAI(&nxt_tkt);
    while (serving != my_tkt) {}
  }
}
```

Ticket spinlocks

Ticket Lock

- TAS-based locks are unfair
- Ticket lock grants access to threads based on FCFS




```
class TicketLock implements Lock {
  int nxt_tkt = 0;
  int serving = 0;
  public void unlock() {
    serving++;
  }
```

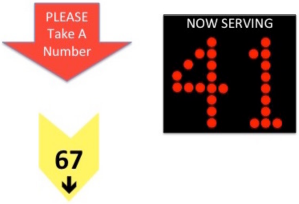
```
  public void lock() {
    int my_tkt = FAI(&nxt_tkt);
    while (serving != my_tkt) {}
  }
}
```

How is this different from Bakery's algorithm?

Ticket spinlocks

Ticket Lock

- TAS-based locks are unfair
- Ticket lock grants access to threads based on FCFS




```
class TicketLock implements Lock {
  int nxt_tkt = 0;
  int serving = 0;
  public void unlock() {
    serving++;
  }
  public void lock() {
    int my_tkt = FAI(&nxt_tkt);
    while (serving != my_tkt) {}
  }
}
```

What are some disadvantages of ticket lock?

Ticket spinlocks

Queued Locks

Key Idea

- Instead of contending on a single "`serving`" variable, make threads wait in a queue (i.e., FCFS)
- Each thread knows its order in the queue

Implementation

- Use an array-based queue
  - 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

Array-based Queued Lock

```java
public class ArrayLock {
  AtomicInteger tail;
  volatile boolean[] flag;
  ThreadLocal<Integer> mySlot = ...;
  public ArrayLock(int size) {
    tail = new AtomicInteger(0);
    flag = new boolean[size];
    flag[0] = true;
  }


```

```java
  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;
  }
}
```

- + Provides fairness
- + Invalidation traffic lower than Ticket lock

Array-based Queued Lock

```
public class ArrayLock {
  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;
  }
}
```

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

MCS Queue Lock

MCS Queue lock is the state-of-art scalable FCFS lock

- Uses linked lists instead of arrays

+ Space required to support n threads and k locks: $\mathcal{O}(n+k)$

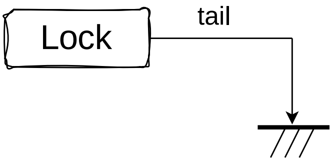
J. Mellor-Crummey and M. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. ACM TOCS, 1991.

MCS Queue Lock

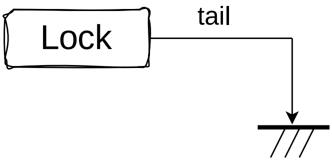
```
class QNode {
  QNode next;
  bool waiting;
}

public class MCSLock {
  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) {}
    }
  }
```

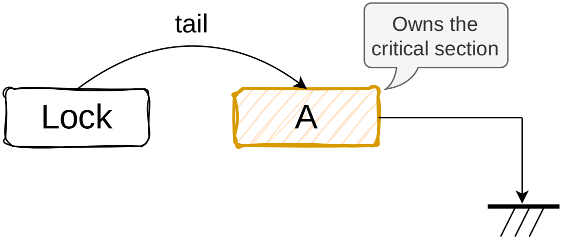
```
  public void unlock() {
    QNode node = myNode.get();
    QNode succ = node.next;
    if (succ == null)
      if (CAS(tail, node, null))
        return;
    do {
      succ = node.next;
    } while (succ == null);
    succ.waiting = false;
  }
}
```

MCS Lock Operations

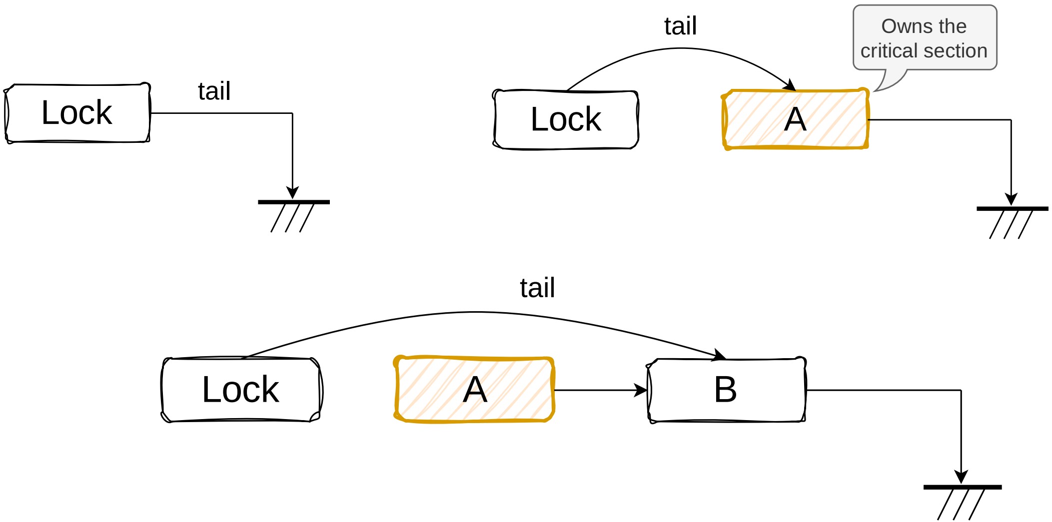


MCS Lock Operations

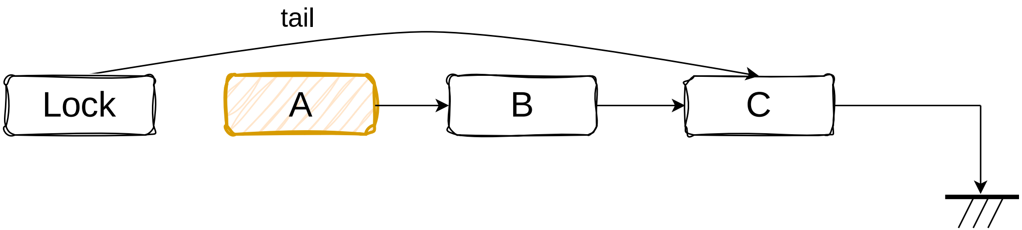




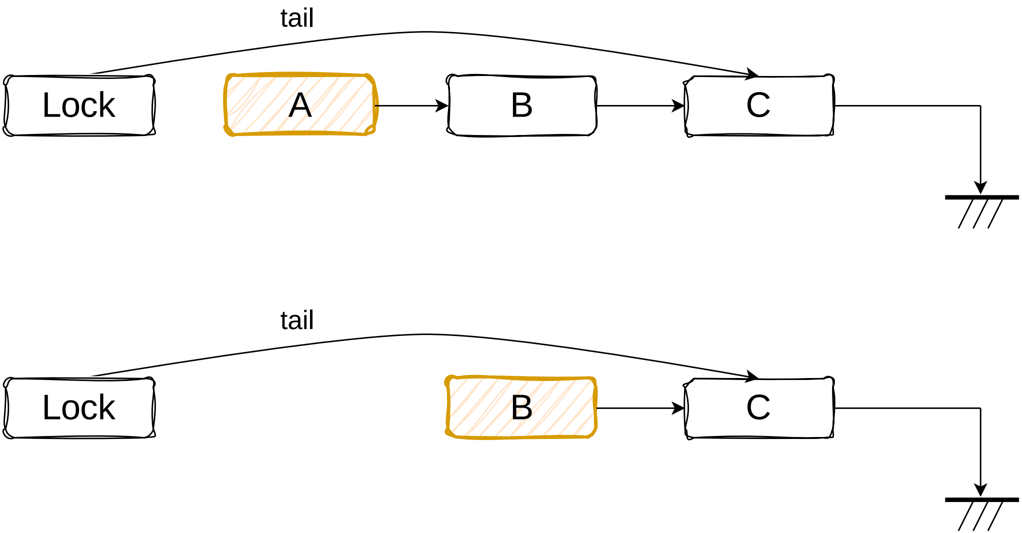
MCS Lock Operations



MCS Lock Operations



MCS Lock Operations



Properties of MCS Lock

- Threads acquire the lock in FCFS manner
- Minimizes false sharing and resource contention
- Threads joining a lock's 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
  - Guaranteed system-wide progress implies lock-freedom, allows individual threads to starve
    - Lock-free implies "locking up" the application in some way (e.g., deadlock and livelock), does not **only** imply absence of synchronization locks

Miscellaneous Lock Optimizations

Reentrant Locks

Reentrant locks can be re-acquired by the owner thread without causing a deadlock

- Freed after an equal number of releases

```java
public class ParentWidget {
  public synchronized void doWork() {
    ...
  }
}

public class ChildWidget extends ParentWidget {
  public synchronized void doWork() {
    ...
    super.doWork();
    ...
  }
}
```

Lazy Initialization In Single-Threaded Context

A variable may require the initialization to be synchronized but future uses may be read-only

```java
class Foo {
  private Helper helper = null;

  public Helper getHelper() {
    if (helper == null)
      helper = new Helper();
    return helper;
  }
  ...
}
```

Correct for single-threaded execution, what could go wrong with multiple threads?

The "Double-Checked Locking is Broken" Declaration

Lazy Initialization In Multi-Threaded Context

```java
class Foo {
  private Helper helper = null;

  public Helper getHelper() {
    if (helper == null)
      helper = new Helper();
    return helper;
  }
  ...
}
```

```java
class Foo {
  private Helper helper = null;

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

The “Double-Checked Locking is Broken” Declaration

Lazy Initialization In Multi-Threaded Context

```java
class Foo {
  private Helper helper = null;

  public Helper getHelper() {
    if (helper == null)
      helper = new Helper();
    return helper;
  }
  ...
}
```

```java
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.
Can we optimize the initialization pattern?

The "Double-Checked Locking is Broken" Declaration

Double-Checked Locking: Possible Idea

(i) Check if `helper` is initialized
- If yes, return
- If no, then obtain a lock

(ii) Double check whether `helper` has been initialized
- If yes, return
- If no, initialize `helper`, return

```java
class Foo {
  private Helper helper = null;

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

The "Double-Checked Locking is Broken" Declaration

Broken Usage of Double-Checked Locking

```
1  class Foo {
2    private Helper helper = null;
3
4    public Helper getHelper() {
5      if (helper == null) {
6        synchronized (this) {
7          if (helper == null)
8            helper = new Helper();
9        }
10     }
11     return helper;
12   }
13   ...
14 }
```

- The writes inside the constructor call of `Helper()` and to the field `helper` (line 8) can get reordered
- The constructor might be inlined, and the compiler could then reorder all the stores
- A partially created object may then become visible to other threads
- Even the hardware can reorder the stores

The “Double-Checked Locking is Broken” Declaration

Double-Checked Locking: Broken Fix

```
1  public Helper getHelper() {
2    if (helper == null) {
3      Helper h;
4      synchronized (this) {
5        h = helper;
6        if (h == null) {
7          synchronized (this) {
8            h = new Helper();
9          }
10       }
11       helper = h;
12     }
13   }
14   return helper;
15 }
```

- A release operation prevents operations from moving out of the critical section
- A release operation does not prevent `helper = h` (line 11) from being moved up (i.e., pulled into the critical section)

The “Double-Checked Locking is Broken” Declaration

Correct Use of Double-Checked Locking

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

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

The "Double-Checked Locking is Broken" Declaration

Readers-Writer Locks

Many objects are read concurrently and updated only a few times

Reader lock No thread holds the write lock

Writer lock No thread holds the reader or writer locks

```java
public interface RWLock {
  public void readerLock();
  public void readerUnlock();
  public void writerLock();
  public void writerUnlock();
}
```

Design Choices in Readers-Writer Locks

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 safely?

Readers-Writer Lock With Reader-Preference

Reader or writer preference impacts degree of concurrency

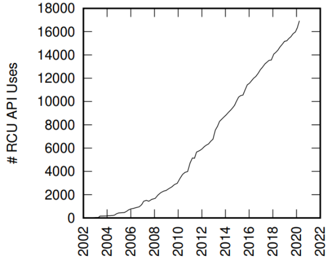
- Allows starvation of non-preferred threads

```
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)
```

RCU Locks

- Think about data structures that are mostly read, occasionally written
  - Manipulating the read counter with atomic operations is still expensive, the cost dominates performance for short critical sections
- **Idea:** Carefully update the data structure so that readers see a consistent view of data, only writers require locks
- RCU supports concurrency between a single updater and multiple readers
  - RCU is wait-free for readers and lock-free for the writer
- RCU was introduced in the Linux kernel in 2002, and is actively used throughout the kernel



What is RCU, Fundamentally?

What is RCU? – “Read, Copy, Update”

RCU Locks

- Readers can enter a CS with just a compiler or memory barrier
- Writers cannot modify data in place, instead the write operation is split
  (i) make a copy of the data,
  (ii) update the local copy, and
  (iii) publish the updated copy atomically with release semantics
- Readers can read potentially stale data
- The old data can be reclaimed only **after** all readers have completed

Using RCU Locks

Writer Thread

```c
struct foo {
  int a, b, c;
};
struct foo *gp = NULL;

/* . . . */

p = kmalloc(sizeof(*p), GFP_KERNEL);
p->a = 1;
p->b = 2;
p->c = 3;
// cannot simply use gp = p;
rcu_assign_pointer(gp, p);
```

Reader Thread

```c
// Reader-side CS

// disable preemption
rcu_read_lock();
p = rcu_dereference(gp);
// cannot simply use p = gp;
if (p != NULL) {
  do_something(p->a, p->b, p->c);
}
// enable preemption
rcu_read_unlock();
```

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

D. Bacon et al. Thin Locks: Featherweight Synchronization for Java. PLDI, 1998.

Asymmetric Locks

Often objects are accessed by most by one thread but require synchronization for (i) occasional accesses by different threads or (ii) for potential parallelization in the future

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

Monitors

Using Locks to Access a Bounded Queue

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

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

What are potential challenges?

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();
}
```

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

Monitors to the Rescue!

- Combination of methods, mutual exclusion locks, and condition variables
- Provides mutual exclusion for methods and the possibility to wait for a condition
  - Condition variables in monitors have an associated wait queue
  - Operations: wait, notify (or signal), and notifyAll (or broadcast)

```java
public synchronized void enq() {
  que.enq(x);
}
```

Condition Variables in Monitors

wait condVar, mtx

- Make the thread wait until a condition encoded by condVar is true
  (i) Releases the monitor's mutex mtx
  (ii) Moves the thread to condVar's wait queue
  (iii) Puts the thread to sleep
- Steps (i)–(iii) are atomic to prevent race conditions
- When the thread wakes up, it is assumed to hold mtx

notify condVar

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

notifyAll condVar

Moves all threads from the wait queue to the ready queue

Implementing a Monitor

Assume there is a mutex mtx, a condition variable condVar and an associated queue condVarQ, and an queue entryQ

```
monitorenter:
  enter the method
  if mtx is locked
    add this thread to entryQ
    block this thread
  else
    lock mtx

wait:
  add this thread to condVarQ
  schedule
  block this thread
```

```
monitorexit:
  // choose next thread to run
  schedule
  unlock mtx
  return from method

notify:
  if condVarQ ≠ ϕ
    remove one thread t from condVarQ
    add this thread to entryQ
    restart t
    // t will occupy the monitor next
    block this thread
```

Signaling Policies

There is a conflict between the signaling and signaled processes for access to the monitor

Signal and wait (SW)	• Signaling thread needs to reacquire the lock • Signaled thread can continue execution
Signal and urgent wait (SU)	• Like SW, but signaling thread gets to go after the signaled thread • Also called Hoare-style monitors
Signal and continue (SC)	• Signaling thread continues to hold the lock • Java implements SC only
Signal and exit (SX)	• Signaling thread exits, signaled thread can continue execution

Bill Venners. Inside the Java Virtual Machine: Thread Synchronization

Bounded Buffers with Spin Locks

```
Queue q;

Mutex mtx; // protect q
```

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

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

Bounded Buffers with Monitors

```
Queue q;

Mutex mtx; // protect q
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);
```

References

- M. Herlihy et al. The Art of Multiprocessor Programming. Chapters 1, 2, 7–8, 2nd edition, Morgan Kaufmann.
- M. L. Scott and T. Brown. Shared-Memory Synchronization. Chapters 1–7, 2nd edition, Springer Cham.
- Jeff Preshing. Locks Aren't Slow; Lock Contention Is.