



# Shared Memory Synchronization

CS121 Parallel Computing  
Fall 2022



# Concurrency bugs

- In parallel systems, high performance algorithm needs high parallelism, good load balancing, memory locality, etc.
- Must also ensure multiple concurrent threads / processes operate correctly.
- Concurrency bugs can arise due to unexpected interleaving of concurrent threads.
- One of the most difficult issues to deal with in parallel / distributed computing.
  - Bugs occur at random times depending on the interleaving.
  - Bugs don't occur during testing, but they will eventually occur in system deployed system.
  - Humans have hard time anticipating and resolving concurrency bugs.
- Concurrency bugs can have very serious consequences.
  - Therac-25 radiation therapy system had a concurrency bug that led to radiation overdose and death of several patients.
  - Space shuttle aborted 20 minutes before maiden launch due to concurrency bug in its avionics software.



# Eliminating concurrency bugs

- Multiple ways, each with pros and cons.
- Critical sections and locks
  - Prevent multiple processes from accessing a block of code at the same time.
  - Easy to use, effective for some problems.
  - But cause contention, overhead and serialization.
  - Need to decide how much code to lock.
    - Too little, and may still have concurrency bug.
    - Too much, and we lose parallelism and performance.
  - If processes acquire several locks, they need to coordinate to maintain correctness, avoid deadlock.
  - Low priority that acquires a lock can delay high priority thread (priority inversion).
  - Despite these problems, locks are still the most widely used solution.



# Eliminating concurrency bugs

## ■ Transactional memory

- A block of code is defined as a transaction, i.e. the block of code either executes atomically without interleaving with other processes, or doesn't execute at all.
- Keep track of reads and writes done by a transaction. If two concurrent transactions read and write to same memory location, abort one of them, i.e. undo all the changes it made.
- Two concurrent transactions accessing different memory locations can both commit, i.e. all the changes it made are made permanent.
- Transactional memory can either be implemented in hardware (HTM) or software (STM).
  - HTM has limits on size and types of transactions it can handle.
    - Implemented in e.g. Intel Haswell, IBM Power8.
  - STM is more flexible, but can be very slow.

## ■ Write your own concurrent code, without hardware support.

- Challenging for most programmers. Not scalable in terms of productivity.
- Correct, efficient algorithms are often research level publications.



# Mutual exclusion

- Given  $n$  concurrent processes that want to perform a critical section (CS), mutual exclusion can satisfy the following properties.
  - No two processes are in CS at same time.
  - If several processes want to enter the CS, at least one succeeds in finite time (deadlock freedom).
  - If several processes want to enter the CS, every process succeeds in finite time (wait freedom).
- All (useful) mutex algorithms satisfy first and second properties.
  - Some algorithms satisfy the third property, but have lower performance.



# Mutual exclusion algorithms

- Mutex is provided by locks. But how are locks implemented?
  - Depends on the type of operations the underlying hardware supports.
  - First type of algorithm uses only read / write operations.
  - Second type uses hardware synchronization primitives such as test-and-set (TAS) or compare-and-swap (CAS), provided in most processors.
- TAS(x) tests if a Boolean variable x is true.
  - If  $x == \text{false}$ , it sets x to true.
  - Returns x's value before the TAS.
  - All these steps done atomically, without interruption from other threads.
  - `getAndSet(x,v)` is similar; atomically set x to value v.
- CAS(x,v,v') tests if variable x currently equals v. If so, it sets x to v'. Otherwise, it doesn't change x. It also returns x's current value.
  - Again, all this is atomic.
- Algorithms also depend on a processor's memory model.
  - Some processors reorder instructions to avoid stalls and obtain higher performance. This can break many lock algorithms.
  - Most lock algorithms assume memory model is sequentially consistent, i.e. the execution order of instructions from different processes is an interleaving of the instructions of each process in program order.

# A first attempt

```
1 class LockOne implements Lock {
2     private boolean[] flag = new boolean[2];
3     // thread-local index, 0 or 1
4     public void lock() {
5         int i = ThreadID.get();
6         int j = 1 - i;
7         flag[i] = true;
8         while (flag[j]) {}           // wait
9     }
10    public void unlock() {
11        int i = ThreadID.get();
12        flag[i] = false;
13    }
14 }
```

Source: The Art of Multiprocessor Programming.  
Herlihy, Shavit

- Two process lock using reads and writes.
- Each thread has an ID *i* for itself, and *j* for the other process.
- Set a flag to indicate interest in CS.
- Wait till other thread's flag unset to enter CS.
- To leave the CS, it resets the flag.

- Algorithm satisfies mutual exclusion.
  - Either thread A or B does its line 7 first.
  - E.g. if A does 7 first, then when B does its line 8, it sees flag[A] set and doesn't enter CS.
  - So only one process in CS at a time.
- Algorithm is not deadlock free.
  - If A and B both do line 7 before line 8, both will see the other's flag as true, and wait forever.

# Peterson's mutex algorithm

```
1 class Peterson implements Lock {
2     // thread-local index, 0 or 1
3     private volatile boolean[] flag = new boolean[2];
4     private volatile int victim;
5     public void lock() {
6         int i = ThreadID.get();
7         int j = 1 - i;
8         flag[i] = true;           // I'm interested
9         victim = i;               // you go first
10        while (flag[j] && victim == i) {}; // wait
11    }
12    public void unlock() {
13        int i = ThreadID.get();
14        flag[i] = false;          // I'm not interested
15    }
16 }
```

- Each thread has a flag to indicate interest in CS.
- There's a shared variable victim accessed by all the threads.
- When a thread wants to enter the CS, it first sets victim to itself to let the other thread go first.
- A thread waits while the other thread is interested in the CS, and while the victim is itself.
- To leave CS, it resets the flag.

- Algorithm satisfies mutual exclusion.
  - Let i be the process that did line 9 last.
  - Both i, j already did line 8.
  - So when i does line 10, it waits for j.
- Algorithm satisfies deadlock freedom.
  - Can't have both threads waiting at line 10, because only one thread (whichever one did line 9 last) can see victim == itself.
- Algorithm is also wait-free.
- Can build n process mutex by repeated use of 2 process mutex.





# Lamport's bakery algorithm

- $n$  process mutual exclusion.
- Based on each process getting a ticket, similar to lining up at the bakery or bank.
  - Code has two sections, doorway and waiting.
  - A process always finishes its doorway code in a bounded (in  $n$ ) number of steps.
- Satisfies first come first serve (FCFS) property:
  - If process  $i$  finishes its doorway before process  $j$  starts,  $i$  will enter the CS before  $j$ .
- Thus, this algorithm is wait free, because each process eventually finishes its doorway section, after which it's guaranteed to enter the CS before any process that starts the doorway later.

# Lamport's bakery algorithm

```
1 class Bakery implements Lock {
2     boolean[] flag;
3     Label[] label;
4     public Bakery (int n) {
5         flag = new boolean[n];
6         label = new Label[n];
7         for (int i = 0; i < n; i++) {
8             flag[i] = false; label[i] = 0;
9         }
10    }
11    public void lock() {
12        int i = ThreadID.get();
13        flag[i] = true;
14        label[i] = max(label[0], ..., label[n-1]) + 1;
15        while (( $\exists k \neq i$ )(flag[k] && (label[k], k) << (label[i], i))) {}
16    }
17    public void unlock() {
18        flag[ThreadID.get()] = false;
19    }
20 }
```

- The doorway code is lines 12 to 14. Line 15 is the waiting section.
- Each process has a flag to show interest in the CS, and an integer label.
- Each process reads the labels of all the other processes, and sets its label to be one larger than the max.
  - Several threads can be reading and setting labels at the same time, and can assign themselves the same label.
- Then the thread waits for all other interested threads with smaller labels to reset their flags.
  - Use thread ID to break ties on labels (lexicographic ordering).

- The algorithm is deadlock free.
  - At any time, some thread has the min (label, ID). That thread won't wait to enter the CS.
- The algorithm is FCFS.
  - If some thread i finishes line 14, then any other thread who starts its doorway (line 12) later will see i's label and choose a larger label. Then it will wait for i at 15.
- The algorithm satisfies mutual exclusion.
  - Suppose for contradiction both i and j are in CS. Suppose WLOG  $(\text{label}[j], j) > (\text{label}[i], i)$ .
  - When j did line 15, it saw either  $\text{flag}[i] == 0$  or  $(\text{label}[i], i) > (\text{label}[j], j)$ .
    - The latter can't happen, because i's labels are monotonically increasing.
  - So  $\text{flag}[i] == 0$ , and i hasn't done 13 yet.
  - But then when i does 14, it will see  $\text{label}[j]$  and choose a higher label, contradiction.

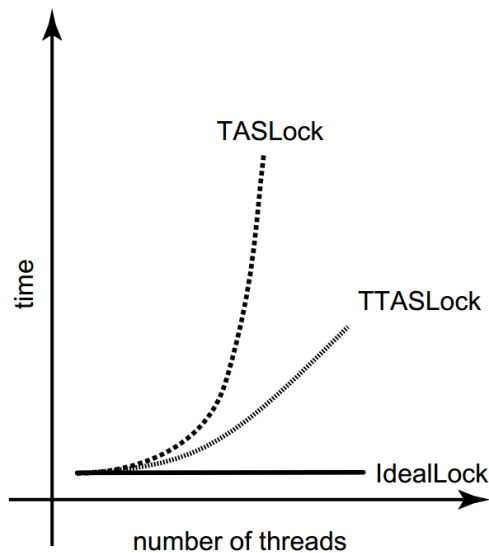
# Test-and-set based locks

- n process mutex algorithms are somewhat complicated.
- Most also assume sequential consistency, which many processors don't offer.
- Instead, we can build simpler locks using built-in hardware primitives such as test-and-set (aka TAS, aka getAndSet).
- A basic algorithm is the following.

```
1 public class TASLock implements Lock {  
2     AtomicBoolean state = new AtomicBoolean(false);  
3     public void lock() {  
4         while (state.getAndSet(true)) {}  
5     }  
6     public void unlock() {  
7         state.set(false);  
8     }  
9 }
```

- To enter CS, a process does getAndSet on variable state.
- If it's the first to arrive at the CS, it receives return value false and enters the CS. If it's not the first, it receives true and waits.
- To exit the CS, a process resets state.
- The algorithm satisfies mutual exclusion and deadlock-freedom.
- It is not wait-free, because a process that wants the CS can always get true on line 4.

# Improving performance



```
1 public class TTASLock implements Lock {
2     AtomicBoolean state = new AtomicBoolean(false);
3     public void lock() {
4         while (true) {
5             while (state.get()) {};
6             if (!state.getAndSet(true))
7                 return;
8         }
9     }
10    public void unlock() {
11        state.set(false);
12    }
13 }
```

- Simple TASLock performs poorly on multiprocessors.
  - Each getAndTest incurs cache coherency traffic.
  - Also causes processes to flush their cached copy of state, so they access memory to read state's new value.
- TTASLock uses get (read) on state instead of TS.
  - If it sees state == false, it uses getAndSet to try to set state.
  - TTASLock performs better than TASLock because the gets read the cached copy of state and don't cause coherency traffic.
    - Reading cached copy of variable is called local spinning.
  - Only when the process in CS exits and sets state to false, or at 6 when processes contend to enter the CS, is there a cache coherency broadcast.
  - Thus performance still degrades with increasing number of threads, but is much better than TSLock.

# Backoff based locks

```
1 public class BackoffLock implements Lock {
2     private AtomicBoolean state = new AtomicBoolean(false);
3     private static final int MIN_DELAY = ...;
4     private static final int MAX_DELAY = ...;
5     public void lock() {
6         Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
7         while (true) {
8             while (state.get()) {}
9             if (!state.getAndSet(true)) {
10                 return;
11             } else {
12                 backoff.backoff();
13             }
14         }
15     }
16     public void unlock() {
17         state.set(false);
18     }
19     ...
20 }
```

```
1 public class Backoff {
2     final int minDelay, maxDelay;
3     int limit;
4     final Random random;
5     public Backoff(int min, int max) {
6         minDelay = min;
7         maxDelay = min;
8         limit = minDelay;
9         random = new Random();
10    }
11    public void backoff() throws InterruptedException {
12        int delay = random.nextInt(limit);
13        limit = Math.min(maxDelay, 2 * limit);
14        Thread.sleep(delay);
15    }
16 }
```

- With previous algorithms, if a thread doesn't get the lock, it keeps trying.
- Since the lock won't be available immediately anyways, we can instead make thread back off, i.e. wait before retrying.
- To prevent all threads waiting same time and retrying together, back off for a random time period.
  - Same idea (exponential backoff) used in e.g. Ethernet.
- If still fail to get lock after backoff, then double the waiting time (line 13 in Backoff).
- Main question is how long to back off for.
  - Too short, and there are still wasted retries.
  - Too long, and threads unnecessarily delay themselves entering the CS.
- The best backoff policy is still an active area of research.

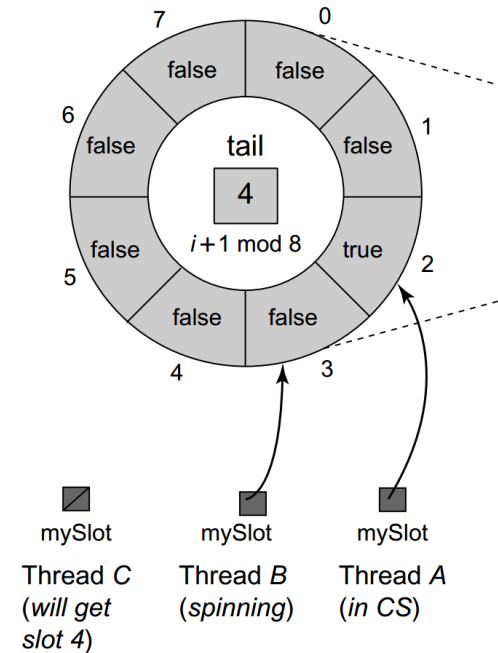
# Anderson's queue lock

```

1 public class ALock implements Lock {
2     ThreadLocal<Integer> mySlotIndex = new ThreadLocal<Integer> () {
3         protected Integer initialValue() {
4             return 0;
5         }
6     };
7     AtomicInteger tail;
8     boolean[] flag;
9     int size;
10    public ALock(int capacity) {
11        size = capacity;
12        tail = new AtomicInteger(0);
13        flag = new boolean[capacity];
14        flag[0] = true;
15    }
16    public void lock() {
17        int slot = tail.getAndIncrement() % size;
18        mySlotIndex.set(slot);
19        while (! flag[slot]) {};
20    }
21    public void unlock() {
22        int slot = mySlotIndex.get();
23        flag[slot] = false;
24        flag[(slot + 1) % size] = true;
25    }
26 }

```

- ❑ Queue locks avoid backoff's problem of having to choose the right backoff period, and avoids the TAS locks' problems of excessive cache coherency traffic.
- ❑ This queue lock algorithm requires a known upper bound size on the number of concurrent threads.
- ❑ There's a shared integer tail, initially 0, and a shared array flag.
  - ❑ Initially only flag[0]==true, all other flags are false.
- ❑ Each thread also has its own private mySlotIndex.
- ❑ To get the lock, a thread atomically increments tail and gets a slot.
  - ❑ Then it spins on flag[slot] until it becomes true, then enters the CS.
- ❑ To unset the lock, it sets its slot's flag to false, and sets the next slot (mod size)'s flag to true.
  - ❑ Then the process waiting at the next slot can enter the CS.



- ❑ Each thread spins on a different flag[slot].
- ❑ If slot changes to slot+1, then only thread previously spinning on flag[slot+1] needs to reread flag[slot+1] from memory.
- ❑ So there's much less memory traffic and we get better performance.

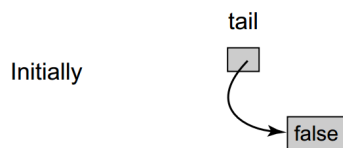
# CLH queue lock

```

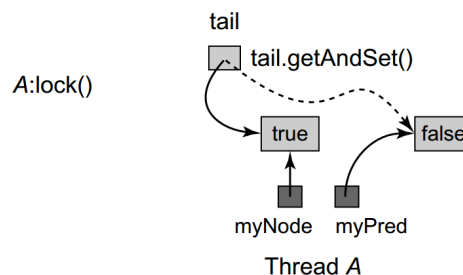
20  public void lock() {
21      QNode qnode = myNode.get();
22      qnode.locked = true;
23      QNode pred = tail.getAndSet(qnode);
24      myPred.set(pred);
25      while (pred.locked) {}
26  }
27  public void unlock() {
28      QNode qnode = myNode.get();
29      qnode.locked = false;
30      myNode.set(myPred.get());
31  }
32  }

```

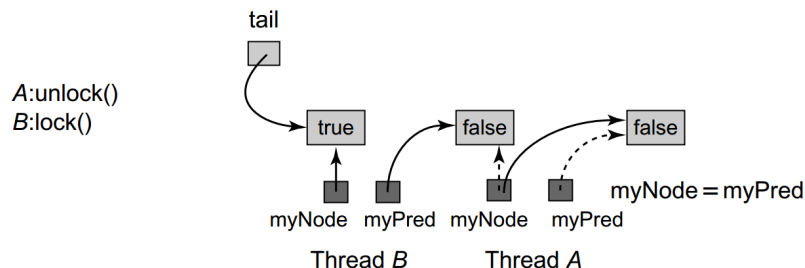
(a)



(b)



(c)



- ❑ Invented by Craig, Hagersten and Landin.
- ❑ ALock needs to know max number of concurrent threads.
- ❑ CLH lock uses a linked list, doesn't need to know a bound.
- ❑ `tail` is a shared variable, and each thread has a private `myNode` and `myPred`.
- ❑ Each thread wanting the lock first sets its `myNode.locked` to true.
  - ❑ Then does `getAndSet` on `tail` to set its predecessor `pred` to what `tail` was pointing at, and set `tail` to `myNode`.
    - ❑ So thread joins the list of nodes waiting for the lock.
  - ❑ Then it spins on `myPred` until it's unlocked.
- ❑ To unlock, a thread sets `myNode.locked` to false.
  - ❑ It also sets the node it will use the next time, i.e. `myNode`, to `myPred`.
  - ❑ This works because only this thread was spinning on `myPred` and will use a node as its `myNode`.
- ❑ Algorithm is very efficient, except in cacheless NUMA architecture.
  - ❑ Without cache, spinning on predecessor's locked field incurs remote accesses.

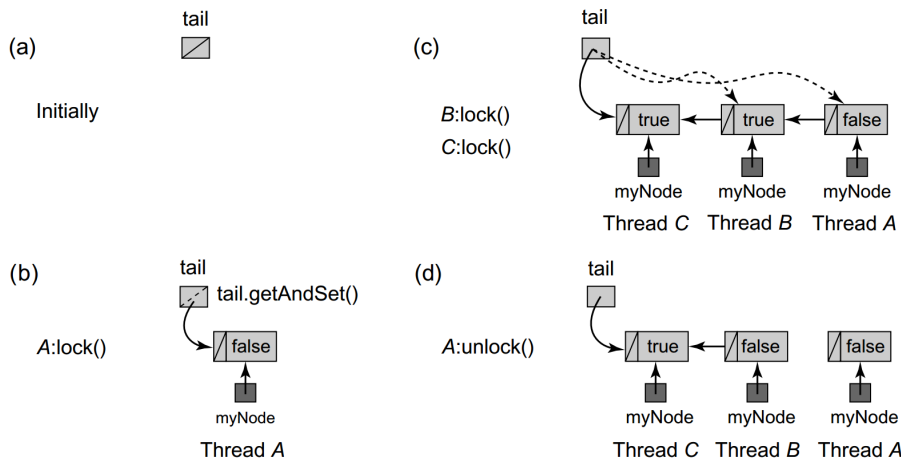
# MCS queue lock

```

18 public void lock() {
19     QNode qnode = myNode.get();
20     QNode pred = tail.getAndSet(qnode);
21     if (pred != null) {
22         qnode.locked = true;
23         pred.next = qnode;
24         // wait until predecessor gives up the lock
25         while (qnode.locked) {}
26     }
27 }
28 public void unlock() {
29     QNode qnode = myNode.get();
30     if (qnode.next == null) {
31         if (tail.compareAndSet(qnode, null))
32             return;
33         // wait until predecessor fills in its next field
34         while (qnode.next == null) {}
35     }
36     qnode.next.locked = false;
37     qnode.next = null;
38 }

```

- ❑ Invented by Mellor-Crummey and Scott.
- ❑ As with CLH lock, there is a tail shared variable, and each thread has a private myNode and myPred.
  - ❑ Each node now has a next pointer to the next thread that should enter the CS after itself.
- ❑ To get lock, a thread does getAndSet on tail to set its node's predecessor to what tail was pointing at, and set tail to myNode.
  - ❑ If pred == null, there's no thread in the CS, so this thread enters.
  - ❑ If pred != null, set predecessor's next field to myNode.
  - ❑ Spin till predecessor sets myNode.locked to false, which lets this thread enter CS.
- ❑ Advantage over CLHLock is that lock() spins on myNode.locked, which is a local variable and doesn't incur network traffic.



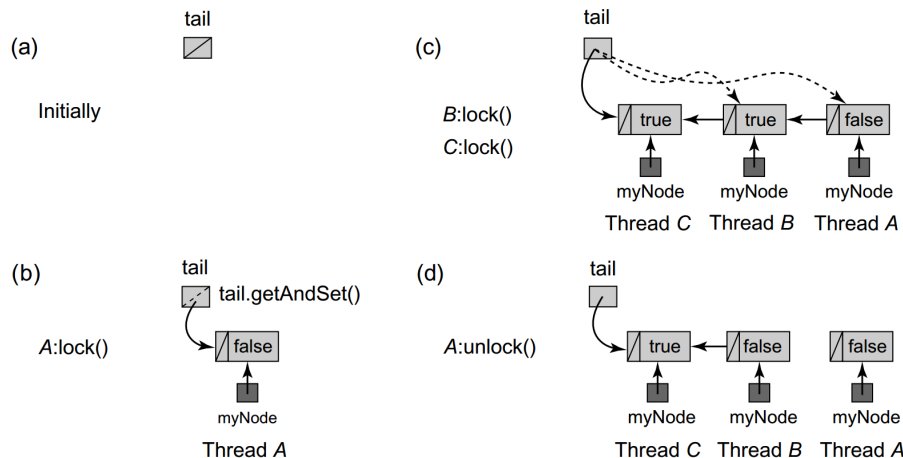


# MCS queue lock

```

18 public void lock() {
19     QNode qnode = myNode.get();
20     QNode pred = tail.getAndSet(qnode);
21     if (pred != null) {
22         qnode.locked = true;
23         pred.next = qnode;
24         // wait until predecessor gives up the lock
25         while (qnode.locked) {}
26     }
27 }
28 public void unlock() {
29     QNode qnode = myNode.get();
30     if (qnode.next == null) {
31         if (tail.compareAndSet(qnode, null))
32             return;
33         // wait until predecessor fills in its next field
34         while (qnode.next == null) {}
35     }
36     qnode.next.locked = false;
37     qnode.next = null;
38 }

```



- ❑ To unlock, check if `myNode` has predecessor (i.e. a thread waiting to enter CS) by checking if `myNode.next == null`.
    - ❑ If so, then either `myNode` doesn't have a predecessor, or the predecessor is slow to do line 23.
    - ❑ To distinguish the cases, thread does `CAS(tail, myNode, null)` in line 31.
    - ❑ If `tail == myNode`, then there's no predecessor.
      - ❑ `CAS` returns `myNode` and sets `tail` to null.
    - ❑ If `tail != myNode`, then there is a predecessor.
      - ❑ `CAS` returns current `tail`.
      - ❑ Wait for predecessor to identify itself, by setting `myNode.next` equal to its ID.
  - ❑ Let the predecessor (i.e. `myNode.next`) enter CS by setting its `locked` to false.
  - ❑ Set `myNode.next` to null. `myNode` can be reused by this thread for its next CS.
- 
- ❑ All the spinning is on local variables, so MCS is fast.
  - ❑ Compared to `CLHLock`, this algorithm does more reads and writes, and also uses `CAS`.